libDsp — An Object Oriented C++ Digital Signal Processing Library

by

Daniel F. Gruhl

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1995

Author .....
Department of Electrical Engineering and Computer Science
December 21, 1994

Certified by.. .....
Barry L. Vercoe
Professor
Thesis Supervisor

Accepted by .....
F. R. Morgenthaler
Chairman, Departmental Committee on Graduate Theses

libDsp — An Object Oriented C++ Digital Signal Processing Library

by

Daniel F. Gruhl

Submitted to the Department of Electrical Engineering and Computer Science
on December 21, 1994, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis covers the design and implementation of a library of objects and functions in C++ for the manipulation of sounds. The library is meant to be machine independent and portable, though its basic layout assumes a Unix like environment. The library assumes a sound file to sound file processing scheme. It is designed to be easily extensible, reasonably optimizable and fairly complete.

This thesis includes a discussion of the design issues addressed in the development of the library, the specific details of the implementation, and a discussion of the applications currently using the library. Last will be a discussion of what directions future work on the library might proceed in.

Thesis Supervisor: Barry L. Vercoe
Title: Professor

Acknowledgments

I would like to thank Professor Vercoe and all the members of the Music and Cognition Group at the MIT Media Laboratory for their guidance and insites into the world of sound processing. I would like to thank John Buck, who was the sole instructor responsible for my learning DSP while an MIT undergraduate. I would also like to thank Melba Jezierski of the Writing Center for her patient help in converting my thesis to a form comprehendible by others beside myself.

Contents

List of Figures

Chapter 1

Introduction

1.1 Definition of Problem

A fairly common problem that appears in sound research is that a researcher spends that majority of his[1] time trying to get data into some form for processing, and getting it back from processing into a form for evaluation[2], rather than on constructing the algorithms that do the processing.

I encountered this problem myself in doing research on internote transitions in human singing. I was trying to find a way to anticipate the next note in a song. What I spent the majority of my time on was getting sound data into a form I could look at. I had to use approximately eight different programs and no less than three different machines to attempt a simple pitch track.

What I needed, and what any researcher in this field needs, are tools that will allow him to take a sample sound file on whatever machine he happens to be using, and to very quickly obtain spectrograms, statistics, plots and other data on this sample. If his job is to devise an algorithm to predict the above mentioned note transitions, the majority of his work should be on the algorithm itself, and not on the mechanics of sound file I/O. Additionally, the final product of such research (probably a program of

[1]Please note, that for simplicity, I will use masculine pronouns when I imply a neuter. This choice is based on my own masculinity, and does not mean to imply that all sound researchers are male.

[2]Often by listening to it, or plotting it.

some sort) should be portable to any of a number of machines, without much additional work.

The problem this thesis addresses, then, is to provide such a researcher with a set of tools to facilitate computer investigations into the nature and perceptions of sound.

1.2 Current Art

Several tools now exist that are in common use for sound processing. Two fairly typical examples of these are Matlab and CSound. Both allow for the manipulation of sound files, and both provide a fair number of signal processing functions. However, both have shortcomings when one attempts to use them for the kind of sound processing commonly done in sound research.

Matlab is a matrix and vector handling package that has a number of useful features. It allows arrays to be treated as objects, and manipulated with common matrix operators. It not only has a suite of built-in functions, but it is also extensible, allowing the user to define his own functions. Built into the program are graphing and file I/O functionality. This functionality makes it easy to read in a data set, and then perform any of a number of operations on it.

However, Matlab is not without its problems. Many versions of Matlab suffer from inherent memory limitations that prohibit the analysis of larger arrays[3]. Additionally, as Matlab is an interpreted language, at least as far as user defined functions are concerned, it suffers from very slow processing of loops, which are the heart of many signal processing algorithms[4]. Lastly, as Matlab (and most other packages like it) are commercial packages[5], there is a considerable financial cost associated with using

[3]Note that a 10 second sound sample taken at CD quality represents nearly a half million entry array. This is too large for the version of Matlab which I tested. The problem gets even worse when you are trying to analyze a five minute piece of music.

[4]While it is often possible to code a Matlab algorithm to use just vector operations, which Matlab can do quickly, this can be quite difficult and requires a fair degree of skill. It also doesn't always work.

[5]There does exist a package, Octave, which is free software and provides much of Matlabs functionality. However, one area it is quite weakness is signal processing, as it provides only basic fourier tranforms.

them, especially if one wants a copy of the program on each of several machines.

CSound overcomes many of these problems with Matlab-like programs as it is designed specifically for sound processing. It does a nice job of separating the concept of instrument from score, and expressing a sound, its spectrum and its spectrogram as different objects.

However, its strength, and weakness lies in its initial design as a real time sound synthesis system. It supports what might be viewed as a stream approach to sound processing, which is very well suited to real time work. Unfortunately, real time processing places a limit on the complexity of operations which may be performed, and also enforces a certain degree of causality in the operations. Secondly, as CSound is a non-extendable scripting language[6] , its functionality is limited to what operations are predefined.

1.3 Definition of Goal

While there are problems associated with existing packages, there are also a number of strengths. A well designed system will try to exploit these, without introducing the weaknesses described above. Such a set of tools should have:

- Completeness — It should be possible to do all sound manipulation in a single environment, without having to resort to chains of programs to tailor the input and output in specific ways.

- Uniformity — It should provide a set of tools that have a degree of uniformity in how they function, to make learning how to use them easier.

- Extendability — Bearing in mind that any research field constantly expands, it should be simple to expand the tool kit so that it does not become obsolete. These extensions should behave as naturally as the original toolkit.

[6]This is not strictly true. It is possible to write new "modules" in C and link them into the final code. What I mean by non-extensible is that you cannot create new functions in the environment those functions are used.

- Portability — It should be possible to use the same tool kit on any of a number of machines, with the same "look and feel".

- Speed — As sound processing is computationally intensive, the tools to do it should operate as quickly as practical, and allow easy access the underlying data to allow for the writing of fast, special purpose tools.

- Abstraction — It should be easy to abstract to any level, allowing the unimportant underlying details to be ignored. Additionally, any new tools should also be allowed to abstract for the same reason.

1.4 Overview

The second chapter will consider the design choices made in the writing of the library, the third will briefly outline the implementation details of `libDsp`, the fourth will give some examples of current programs that use `libDsp`, and the fifth will discuss new directions and extensions that could be made to the library, and what limitations are inherent to its design.

Chapter 2

Design Choices

An engineering project starts with a set of design objectives. These objectives alone, however, rarely suggest a single solution to the project. This does not imply that all the possible solutions suggested are equally good. Rather, engineering is the finding of the best solution among all those possible.

The first step is limiting the possibilities by making design choices. If each of these choices is made in an informed way, it is likely that when the design process is complete, the solution arrived at will be acceptable.

This chapter addresses what choices were made in the course of this thesis project, and why. While Chapter 3 will address the specifics of what was done, this chapter might be best described as listing what was *not* done and why.

2.1 Environment

The first set of considerations to be addressed are those involved in defining the environment in which the tool set will be implemented. These choices define what over all "shape" the tool set will take.

2.1.1 Language vs. Library

In writing a set of tools, one can either start from scratch and write a whole new language, or one can try to graft new features onto an existing language. From a designer's standpoint, it is often attractive to start fresh. This allows the designer to tailor the architecture of his language to the problem at hand, and almost always results in a cleaner, easier to understand product.

CSound is an example of where this is used to good effect. The language is structured so that there is a logical distinction between instruments and scores. Since this mirrors real life, it allows musicians to continue working on the computer in ways they are already used to.

Despite these advantages, I chose to implement my tools as a library, that is, as an extension to an existing language. First, and foremost, I wanted to encourage the use of the package. It is quite frustrating to have to rewrite all your programs to use the new "latest and greatest" language. If implemented as a library, however, existing code can continue being used, and useful tools and features can be slowly integrated as needed[1].

Second, with every new language, there is a learning curve associated with using it. A new language implies a new approach and a new way of thinking about problems. While this is true even for a library, using an existing language offers a familiar syntax and a correctly implemented library follows it as much as possible. For example, if `a = 1 + 1` is an assignment in the parent language, it helps if the extension allows two objects to be added with `result = object + object`.

Third, it is quite time consuming to implement all the trivial features of any language like $+$, $-$, $\times$, and $\div$ as they apply to the base types (ie., integers, floating point numbers, etc.). Using an existing language allows you to inherit these from the parent language as appropriate.

Fourth is a matter of speed. For a language to quickly implement arbitrary algo-

[1]Not unlike the way C++ can compile most C programs. As a result, existing code can be compiled with a C++ compiler, and the new features of the C++ language can be added to existing programs as useful.

rithms, it almost certainly needs to provide an optimizing compiler. It is, however, often prohibitively time consuming to write one from scratch. As I wanted my package to be quite fast, and I didn't have the time to write an optimizing compiler myself, I decided to use a library written in a common languages, for which there would hopefully be a good compiler already developed.

2.1.2 What language to use

Given that I wante d to develop these tools as a library, the next question was ' guage to implement the library in. The logical choice was C, as it is almost universally portable, one of my design goals.

Another of my design goals, however, was to abstract my representations as much as possible. For an example of what happens when you try to abstract too far in C, see X windows[2]. While it is possible to make such abstractions work, it is very difficult to make them work elegantly.

It made sense then to consider languages that allow for a high degree of abstraction. Scheme comes to mind, as do languages such as Smalltalk and Common Lisp. The difficulty with all of these is that they are interpreted, and therefore are fairly slow when compared to compiled C code[3].

Additionally, these languages are somewhat more esoteric and not as generally available as C. The compromise, therefore, was to use C++. While it is not as well distributed as C, it is becoming more and more prevalent, and though it does have several inherent limitations, these can be overcome in most circumstances to provide a workable environment which supports abstraction and inheritance.

2.1.3 Real Time vs. Static Processing

One big consideration was whether or not to structure `libDsp` to support real time processing. Either choice has its pluses and minuses. On the plus side for real time

[2]Specifically the X Toolkit Intrinsics.

[3]However, compiled Scheme runs ay 40% the speed of comparable C. Still, the difference between waiting 4 hours for a run and 8 hours is non-trivial.

processing is the advantage of being able to quickly get feedback on what a process is doing, and also to use `libDsp` in a real time performance type environment.

This interactiveness is not without its price. First, when attempting to operate in real time, one must make some difficult design choices for how to deal with situations where the user has requested processing faster than the machine is capable. One common solution to this problem is to throw away part of the input, so that the data comes in slowly enough to deal with.

While this is a good solution in terms of the output sound being reasonable, it is somewhat disconcerting to run an analysis on a sound file three times, and to get three different results depending on how busy the machine is at the moment. As I intended `libDsp` to be primarily an analysis tool, I felt that such variance was unacceptable.

Secondly, real time processing imposes a high degree of causality. You cannot delay the output by as much as $\frac{1}{10}$ of a second, so you are limited in how far you can "look ahead" in doing a calculation. This prohibits using any function that needs to "see into the future" to process data. Examples of such functions include automatic gain control, rescaling the volume of whole songs, and convolution of very large zero phase delay filters. As I wanted to be able to do as much as possible with `libDsp`, I felt the inability to perform these types of functions would impose an unacceptable constraint.

2.2 Completeness

Completeness, in a library, means that you should be able to use the library to do any of the common tasks that it was designed for. It is frustrating to start using a tool, and to find out it doesn't do everything you need. I felt that it was important to provide features that might be useful unless there was a compelling reason not to. When a feature might be useful, but implementing it wasn't practical due to time constraints, I tried to provide for it. For specifics of what expansions are planned, see Chapter 5.

2.2.1 Input and Output Capabilities

Provide Sound I/O functions

It is tempting to provide functions that allow the library to read in and play out its own sounds. However, there is the problem of accessing the sound hardware. It turns out there are almost as many different ways to access a sound port on a computer as there are brands of computers. No clear standard exists for how to do this[4] and as a result, new code must be written for every machine that `libDsp` would run on.

On the operating system level, different operating systems, and even different releases of the same operating system provide different ways of accessing the sound functions[5]. Thus, for a package to be truly portable, a way of accessing the sound port on all major and many minor platforms must be provided. This can become a real problem when you upgrade the package and don't have a system of every type to test your code on. You are forced to rely on other beta testers to find bugs, maintain your code, etc.

On top of this, many systems[6] provide an excellent GUI[7] interface to sound utilities. It is doubtful that any program could provide a general interface that even approaches the ease of use of the one provided by the manufacturer.

I therefore decided to work on sound files acquired from other programs, and played by other programs. If a standard for hardware access emerges, it might be worthwhile to include this ability at a later time.

File formats to support

There are quite a few file formats currently being used for sound files. Some of the more popular ones are AIFF, Wave, an μ-Law[8]. It would be nice to support these and as many other file formats as possible. I have picked the `libDsp` native sound

[4] Although the X Windows like Audiofile package may change this.

[5] In fact, to make matters even worse, some operating systems have one or more optional "sound modules" that can be installed or left out at the installer's discretion.

[6] The NeXT and Silicon Graphics Indigo come to mind.

[7] Graphical User Interface.

[8] Popular under Macs, Windows and Sun/NeXT respectively

file format to be AIFF as it provides all the information needed to generate the other sound formats using translators[9].

Further sound file support, while a definite design objective, is deemed not as important as some others and thus is discussed in Subsection 5.1.4.

Provide plotting functions

Matlab's plotting functions enhance its utility considerably, and I consider the presence of a plotting function to be essential to the usefulness of a sound analysis package.

I also wanted the package be fairly portable this precluded using even so universal a standard as X windows as it is not available on many personal computers. The answer I found was to use an external program, GnuPlot, to do the actual plotting, but to hide the specifics of this in the library.

GnuPlot provides plotting in a wide variety of environments[10]. This has allowed `libDsp` to display in a wide variety of environments with minimum additional development, by just opening a pipe to GnuPlot and letting it do the actual plotting.

Coincidentally, I found after I had implemented this that several major packages use this technique (plotting with GnuPlot) including Calc, Octave and Oleo (all Gnu packages).

2.3 Uniformity

One of my design objectives was uniformity. A uniform approach to designing a toolkit means that if a user learns the syntax for one function, and he already knows the syntax for all related functions. In practice, this is implemented by choosing library naming and calling conventions and sticking to them:

[9]Sox, the excellent sound translation package is one example. Serious thought has been given to incorporating the sox library into `libDsp` and it might become a feature at some later date (see Chapter 5 for a more complete discussion).

[10]The version I have supports plotting to over 50 different devices.

- There are only four major data types defined in `libDsp`. These all follow the convention of studifying[11] the names and all inherit from a common ancestor (`CArray`). This means that all methods common to all objects share a common interface.

- Most functions start with the prefix DSP. eg. the FFT function is `DSPFFT`. This is to avoid possible name crashes with existing functions. Most functions start with an uppercase letter and are studified.

- All functions of one type, ie. filters, signal generators, etc. use the same argument format.

If new additions to the toolkit follow these conventions and inherit from existing functions and objects whenever feasible, they should merge seamlessly with the existing code, and should be quite easy for a user to learn.

2.4 Extendibility and Abstraction

Extendibility is sort of the counterpart to completeness, as it is an admittion that any finite toolkit cannot provide all the functionality that a person might need. All is not lost however, if the toolkit is designed to allow users to add their own functions gracefully.

2.4.1 Inheritance

C++ allows objects to inherit from one another. This means that if a new object is just a special case of an existing object, none of the common code needs to be rewritten. An example of this is the `Stereo` object, which I did *not* implement in my library. Instead, it could be implemented as two Mono objects (`TimeSignals`) with a left/right selector. This simple implementation requires about 20 lines of code.

[11] Studifying — Capitalizing each word in a compound word. eg. FreqSignal. This notation comes from X windows.

Contrast this with `TimeSignal` (and `CArray` which it inherits from) with a current total of 1907 lines. The savings in coding and debugging time is apparent.

Additionally, inheritance enforces the use of similar formats for doing similar things. All signals have a size method which returns how big they are. Anything that inherits from them gets a size method from its parent, unless it provides its own. Again, this helps with uniformity.

Taken together, these two points allow new features to be added to a library with minimal work, encouraging people to extend the library as needed for their own work.

2.4.2 Abstraction

In the above section, I gave the example of how a `Stereo` object could be easily written. But perhaps more important than this is that the resulting object can be used as if it were a primitive defined in the library. In other words, the user doesn't have to think about how a `Stereo` object works, they just use it. This abstraction is one of the tools that keeps a big library from becoming overwhelming to a user.

Additionally abstraction provides another bonus. One feature I would like to add to `libDsp` at a later date is the ability to deal with very long signals. Currently, `TimeSignal` is called with the method `readAiff`. It then loads an entire signal into memory and returns. Clearly, this could be problematic with very long signals (several minutes) as you could run out of memory. The fix to this would be implemented by having `TimeSignal` only read in the portion of an array from a file that it was working on at the time. This would require a fairly major change in the way that the method `readAiff` works, as it would now have to look at the file size[12], decide whether or not to load it all in, etc. But note, it would not change the way `readAiff` was used. In fact, most users wouldn't even notice. This is because the change went on below the abstraction barrier. This ability to make improvements to a library without the user having to learn new ways to use them is key to creating a library that can grow over time.

[12]Which is quite easy to do as this information is included in the AIFF file header

2.5 Portability

One fairly major design goal I wanted to meet was that of portability. I wanted `libDsp` to run on a wide variety of systems, even though this required some sacrifice in functionality as detailed elsewhere.

2.5.1 Autoconfigure vs. System specific changes

There are three fairly typical ways of allowing a package to run on a large suite of systems. The first is to make the code so general that there are no system-dependent variations. While this is sometimes possible, it can be quite difficult because different machines provide for different functionality, even with such "standards" as the math library.

The second option is to code with `#ifdefs` all over your code to include or not include sections as appropriate to a specific machine. While allowing for a high degree of granularity in one's code, this requires a fair amount of work to write, and also access to all targeted systems for testing.

The last method, and the one I chose, was to use the package Autoconfigure from Gnu. It examines the system being installed on, and makes the changes appropriate to the code for that system. This means an install may take somewhat longer, but many more target machines are supported as a result (in fact, almost all Un*x machines are supported).

2.5.2 Imported Code

In constructing any software package, a designer realizes that some of the work that needs to be done in the package has already been done by others. It can be quite seductive to use code that someone else has written in your own package, often appearing to need just minor modifications before you can use it. I feel this approach can be highly problematic, and I will explain why I have avoided using imported code to a large degree, even when it might have been useful.

The first reason is that while someone else's code might be close to what you

need, it often needs "small" modifications. Before you can make these modifications, however, you have to understand the code the other person has written. This can often take longer than simply writing the code again from scratch yourself. Many the "obvious" assumptions the person made in writing this code turn out to be the reverse of what you assume. If you are lucky, you can give up on porting their code before you have invested too much time in it.

Second, assuming that the code is well documented, clearly written, and exactly what you need, the chances are the person who maintains it is quite proud of it. So proud, in fact, that he expects a small monetary donation for using it. This can become a problem when using each copy of your library requires several dozen small donations. Additionally, there are considerable legal hassles concerned with what you can do with someone else's code (ie., make money off it, choose not to make money off it, etc.).

Next, let's assume that the code you want to use is written by a life member of the Free Software Foundation, and you don't mind your code being copy lefted also. There is still the problem of revisions. That code that worked into your programs so well when it was in Revision 1.43 might break in Revision 1.44. When this happens, you have to choose between making extensive revisions in your own code, or not improving that part of the package when the original author does.

If it sounds as if some problems have been encountered in using others code in the past, this is true. However, by following a few guidelines, it is possible to include other's code without losing portability, which is the main concern of this section.

First, the code should be free to use and hopefully free of any restrictions on use you are unwilling to live with. Second, the code should already be portable to all the systems you want your package to target. Third, the software should be somewhat self contained. Perhaps it is best if it is a separate program all together, accessed through a pipe. Second choice would be a library which is already set up to exist as a separate unit.

Three specific examples of code that is imported into `libDsp` are as follows. First, GnuPlot is used as an auxiliary program, with a pipe as an interface. Second are the

AIFF routines used for reading and writing. While they required a small amount of rewriting to cast them as a library, and difficulties were encountered when the original code underwent revision, they proved to be a real time saver as I didn't have to learn the internals of AIFF files to use them.

Last was the Complex data type from libg++. Because this was used, it has been impossible to port `libDsp` to run on DEC Alphas. It is likely that this object will have to be completely rewritten for such a port to be possible, as gcc (and hence, g++ and libg++) has not been ported to the Alpha.

2.5.3 C++

One of my reasons for choosing C++ as a development language was its portability. By avoiding use of the more advanced features of the language (templates and exception handling, for example) it is possible to use the library on most any system. For those without native C++ compilers, there are programs that can translate C++ to C, and almost all machines have a native C compiler.

2.6 Speed

Signal processing is in general a computationally intensive task. It was therefore important to make design decisions that allow the package to be as fast as possible.

2.6.1 Simple access routines

It is fairly common to provide a "copy" type access routine to data stored in an object. This protects the user against accidentally altering data that they shouldn't be altering. Unfortunately, this almost doubles the amount of time it takes to access data.

Another common practice is to do bounds checking on array lookups. While this can save some time on debugging, the additional compare calls for *every* access to the data can again more than double run times. With this in mind, I decided that the

speed advantage of simple access routines far outweighed the error checking of a more complex scheme.

2.6.2 Assembly language type functions

There are some serious trade offs to be made between code readability and speed. These trade offs come are a result of the compiler not knowing the exact storage size of an object.

Specifically, when an existing object's value is set through a call like `object1 = object2 + object3`, there is an additional, unnecessary copy performed (`object2 + object3` yields an object which is then copied into `object1`). Normally this is not much of a problem, but in the case of large arrays, it can nearly double the run time of a program. It is therefore rather common to provide "assembly like" functions so that the above would be expressed as `add(object1, object2, object3)`. As the result of the add can be written directly into the target, this prevents the extra copy.

2.6.3 Inline functions

Judicious use has been made of inlining. Inlining is a hint to the compiler to expand the designated function as a macro rather than a formal function call. This results in somewhat larger executables, but hopefully also in faster code.

2.7 Conclusion

While there were obviously many other design choices that had to be made, I feel that these are some of the major ones. The rest will become apparent in the next chapter on Implementation, and in Appendix A, the `libDsp` user's manual.

Chapter 3

Implementation

`libDsp` is implemented as a C++ library. As such it provides classes and functions that operate on those classes. These work together to allow a user to quickly implement arbitrary signal processing algorithms.

While the library is implemented in C++, I won't spend that much time explaining the specifics of this language, but instead refer the curious to Dewhurst's book[3] which provides a good introduction.

I will now outline some terminology of object oriented programming (OOP) that is relevant to the discussion in the rest of this chapter. Object oriented programming is based on items called, not surprisingly, objects. An example of an object in the real world is a shoe, a tree or a pen. An example of things that might be objects in programming are numbers, arrays, databases and strings.

Objects can exist by themselves, can be made up of other objects, or they might be a special case of another object. For example, a dog has a tail, and has four legs, and a poodle is a special case of a dog. The two types of inheritance in C++ are just those, *has-a* and *is-a*, and they are used pretty much as they sound. You has-a wallet and you is-a human.

The reason to even bother with this becomes apparent when you consider describing something. By saying Bob *is-a* human, you are stating that he has all the characteristics common to humans. If these are defined somewhere else, you need not define them again when talking about Bob. Likewise, if everyone knows what a wallet

is, it is sufficient to just say that Bob *has-a* wallet.

Lastly, a type of an object is its *class*. Fido is an object of the class dog. Objects can do things, and these things are called *methods.* For example, a dog might have a method `bark`. All objects have a constructor method, which is run when the object comes into being, and a destructor method which is run when an object ceases to exist.

3.1 Data Types

A library has to operate on something, and those somethings are data types. `libDsp` provides four main types of objects, or classes, including the base class `CArray`, and the three working classes `TimeSignal`, `FreqSignal` and `FreqSlice`.

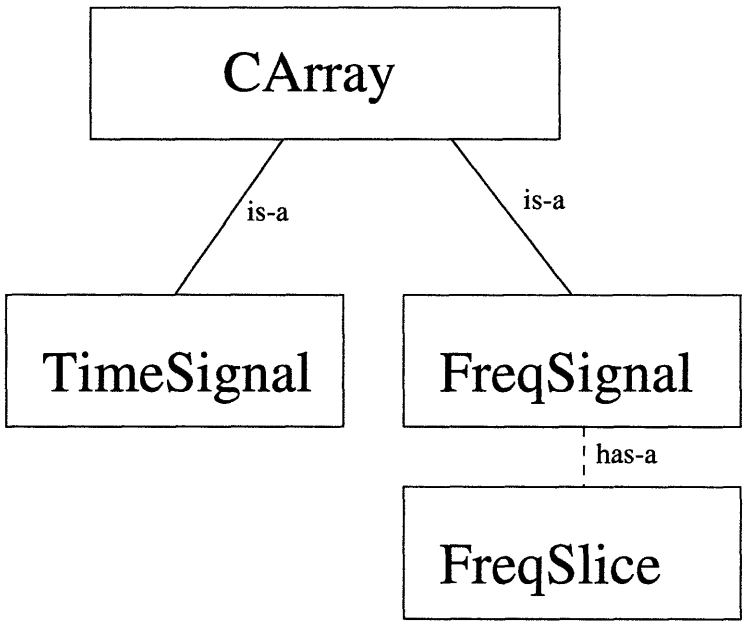

Figure 3-1: Inheritance Diagram for `libDsp`

3.1.1 CArray

`libDsp` defines three classes for general use, and one base class which these others inherit from. This base type is an array of complex numbers known as a `CArray`.

Signals in a non-real time signal processing system are often stored as arrays. I have chosen to implement my basic storage type as an array of complex, floating point numbers. Initially, this choice of using complex numbers may seem to be a bit wasteful of space, as all real signals are, by their nature, real. However, much signal processing is done in the frequency domain where a complex number is a more natural choice.

The choice of using floating point numbers as the base type rather than integers was due largely to discussions with Dan Ellis on the speed of arithmetic in a FPU[1] compared to integer arithmetic. The speed of integer arithmetic is usually the clock speed of the processor chip. This speed is hovering around the 100 MHz mark, but there doesn't seem to be much indication that it will go up by more than a factor of three or four in the near future. FPUs, on the other hand are special purpose processors that are continuing to get faster under pressure of such applications like CAD, simulation and signal processing.

For sound in particular, floating point numbers deal nicely with many of the problems associated with the huge dynamic range of perceived sounds caused by the ear's logarithmic approach to volume discrimination.

Thus the `CArray` data type, an array of complex floating point numbers. Internally, it is nothing more than a pointer to an array of type `Complex` and a count of the elements in the array. This results in an extremely light weight data object as the overhead of an additional long storage space is negligible compared to the array of complex data by itself.

What `CArray` does provide, however, is a large number of ways to access and manipulate the data stored in it.

Constructors

Three constructors are provided, allowing for construction with default size, by defining the number of elements in the `CArray`, by giving `CArray` control of an existing array of `Complex` numbers, or by a "fast pass" through a structure. This last method

[1]Floating Point Unit: A co-proccesor that handles floating point computation.

is used internally for passing without a copy between procedures and is not really intended for users.

Destructor

`CArray` has a destructor that frees any heap memory allocated for the array. The destructor, like all the following methods, is virtual, so any classes that inherit from `CArray` will also automatically call the memory freeing routine on exit. I will not mention destructors for any of the classes that inherit from `CArray` as they are provided by the base class to each member class logically unchanged.

Element Access

Array access is done through a simple inline [ ] function. Array indexing starts from 0, as this is the C standard. No error checking is provided at this stage for reasons discussed in Sub Section 2.6.1. All types that currently inherit this access operator from `CArray` use similar functionality.

Size related function

A method `size` is provided to allow for queries as to the number of array elements. Two methods for changing size are provided, `resize` and `stretch` which change the size of the array and don't copy or do copy as much data as possible, respectively. `slice` returns a copy of a portion of a `CArray`, for example, an array made up of all the elements between 10 and 100 in the original array. Last, the `giveaway` method is used internally in the constructor for the "fast pass".

Plotting

`CArray` provides the interface to the GnuPlot plotting through the method `plot`. This method provides for such sundries as title, axis labels and a number of alignment variables describing how to label and center the data (i.e., spectrums tend to look better if centered at zero and going off to $\pm\pi$, though it is more convenient to work

with them going from 0 to 2π). It is expected that these parameters and the axis labels will be filled in as appropriate by the objects that inherit from `CArray`.

Magnitude and Phase

Two methods, `MagnitudeArray` and `AngleArray` each take a double array and fill it in with the magnitude or phase of the equivalent complex cell. I have found this to be useful in a number of signal processing applications where one treats a spectrum as an object. It is also useful to pull the magnitude out of a signal if this is needed.

It would be convenient to have these methods write out to a generic array type, and they may do so at some later date, but I have yet to find an array type that I like. Additionally, any template driven array type would violate some of the portability arguments given in Chapter 2, though this should change in a year or two as templates become more standard to C++.

Arithmetic

All the built-in arithmetic operators are supported on an element by element basis between `CArray` and int, float, double, and Complex. Additionally, a method `Maxabs` which returns the maximum magnitude in the array is supplied to allow for normalization.

3.1.2 TimeSignal

A `TimeSignal` is the basic representation of a sound. It is-a `CArray` so its storage is that of a `CArray` with an additional variable to hold sampling rate. As such it represents a mono signal, for example, a sample from an external sound track, or the left channel of a CD (see Figure 3-2).

Throughout the `TimeSignal` class, ints and longs are used to represent a number of samples, and doubles a length of time. Many functions are overloaded[2] to allow the

[2]Overloading is the using of the same function name for more than one (hopefully related) functions. For example, + is overloaded, as it represents one function that knows how to add ints, another that knows how to add floats, etc.

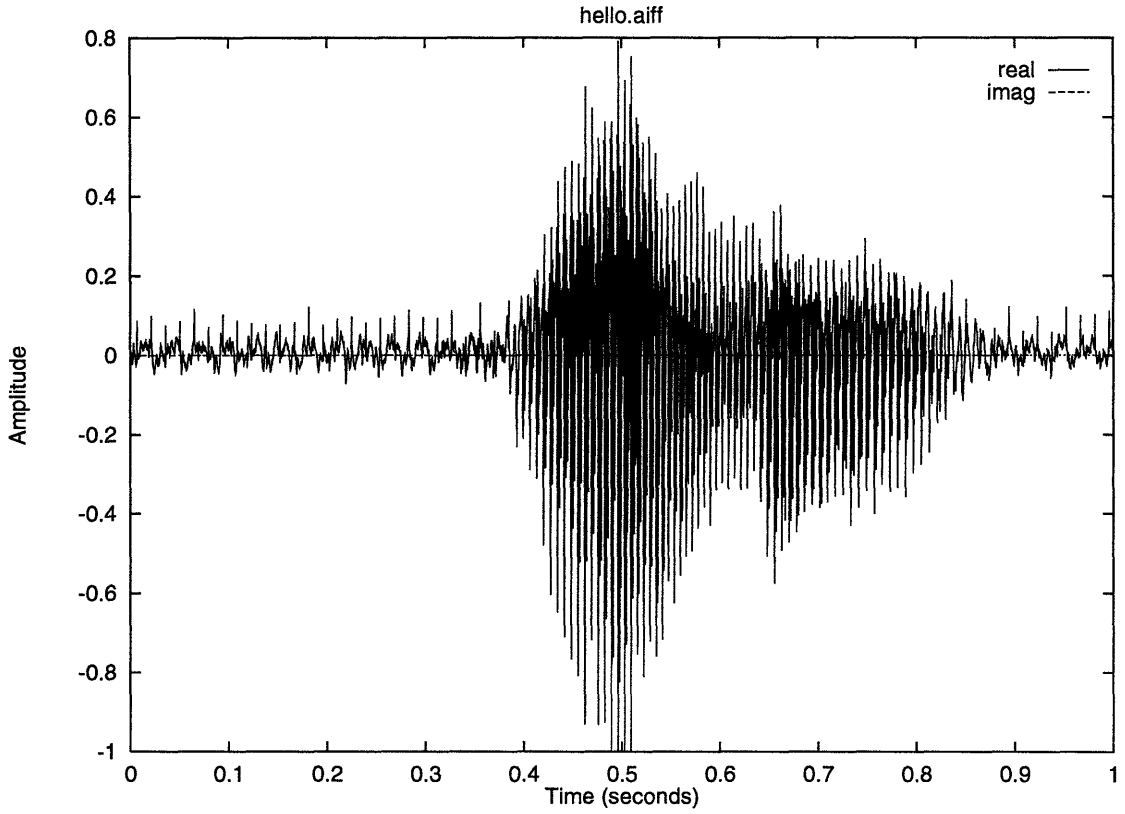

Figure 3-2: Sample TimeSignal of author saying "hello"

use of either in describing the size of a signal.

I use the terms "Time Base" and "Sampling Rate" interchangeably to indicate the number of samples per second in a sound.

Constructors

A selection of six different constructors is provided by `TimeSignal`. These include a default, the setting of just the size (and accepting a sampling rate of one sample per second), setting by sampling rate and number of samples, sampling rate and duration (in seconds) of signal, as well as all the constructors from a `CArray`, with a preceding sampling rate.

Miscellaneous

Provided are methods to set and query the sampling rate and to query the duration. Methods are given to normalize the whole signal, to return a 16-bit quantized version of the data, and to return the largest and smallest samples. Additionally, all the methods from `CArray` are available, as a `TimeSignal` is-a `CArray`.

File I/O

A `TimeSignal` can read and write AIFF files, and can write an ascii dump of itself. I have not yet integrated the sox library (as discussed in Subsection 2.2.1) to allow for reading and writing of arbitrary sound files, but there is no reason it couldn't be done, and I will discuss this further in Sub Section 5.1.4.

Delay

This pair of methods (a number of samples and number of seconds versions) returns a `TimeSignal` delayed by an arbitrary amount (it can be negative). The returned signal is the same length and temporal position as the original, and data that falls off the ends is lost.

Arithmetic

All the arithmetic operations from `CArray` are provided.

3.1.3 FreqSignal

A Frequency Signal, or spectrum, is the frequency space duel of a `TimeSignal`(see Figure 3-3). It maintains a variable from which the number of hertz the spectrum spans can be recovered. While, strictly speaking, assigning anything but a 0 to 2π interpretation to locations on a discrete frequency spectrum is incorrect, it is a useful illusion to preserve, as most users are interested in the corresponding real world meaning of the signal they are looking at, and realize it is only correct in the context of the signal they are working on at that time.

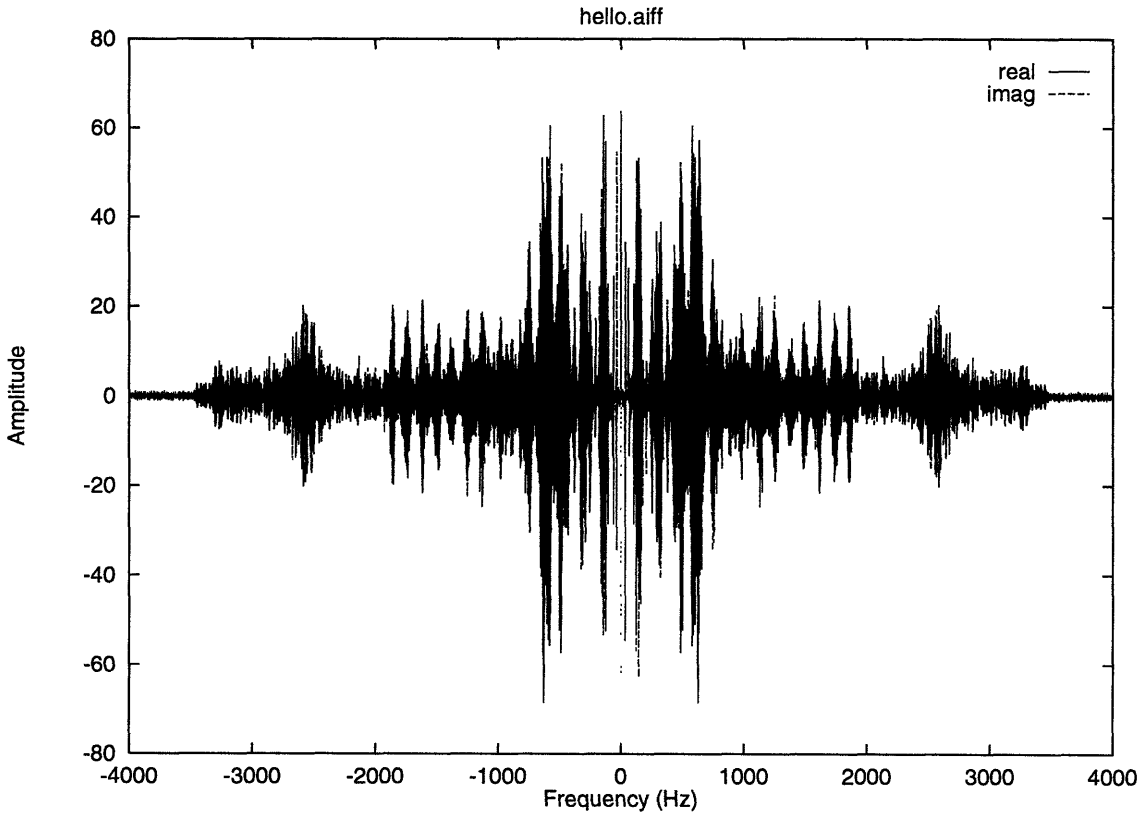

Figure 3-3: Sample FreqSignal of author saying "hello"

Constructors

As most `FreqSignals` are created by DFTs, I have provided very few constructors, the same as those provided to `CArray`, all an option of providing a spectrum width.

Miscellaneous

Provided are methods for plotting, querying the center frequency of any bin, writing an ascii dump of the data and doing all standard math on signals. In many respects, a `FreqSignal` is just another way of looking at a `CArray`. As I don't expect much I/O to be done on signals, and that most analysis will be case specific, `FreqSignal` is a much smaller object than a `TimeSignal`.

3.1.4 FreqSlice

A Frequency Slice is my name for a spectrogram, as a spectrogram can be thought of as the spectrum of slices of a `TimeSignal`(see Figure 3-4). It is created by a `Slice` function from a `TimeSignal`. It is implemented as an array of `FreqSignals` with the number of windows and delta time between windows stored.

As most operations on this type are lookups into the appropriate `FreqSignal`, no real functionality is provided except a fairly complex plotting function that produces nice spectrograms on the GnuPlot interface.

The only math provided is assignment, as no other types were deemed relevant. Also provided is a method to query the center time of any window.

3.2 Functions

Objects in space are interesting, but they become much more useful when there are functions defined to manipulate them. `libDsp` provides a number of such functions to do operations such as signal generation, Fourier transformation, filtering and signal transformation.

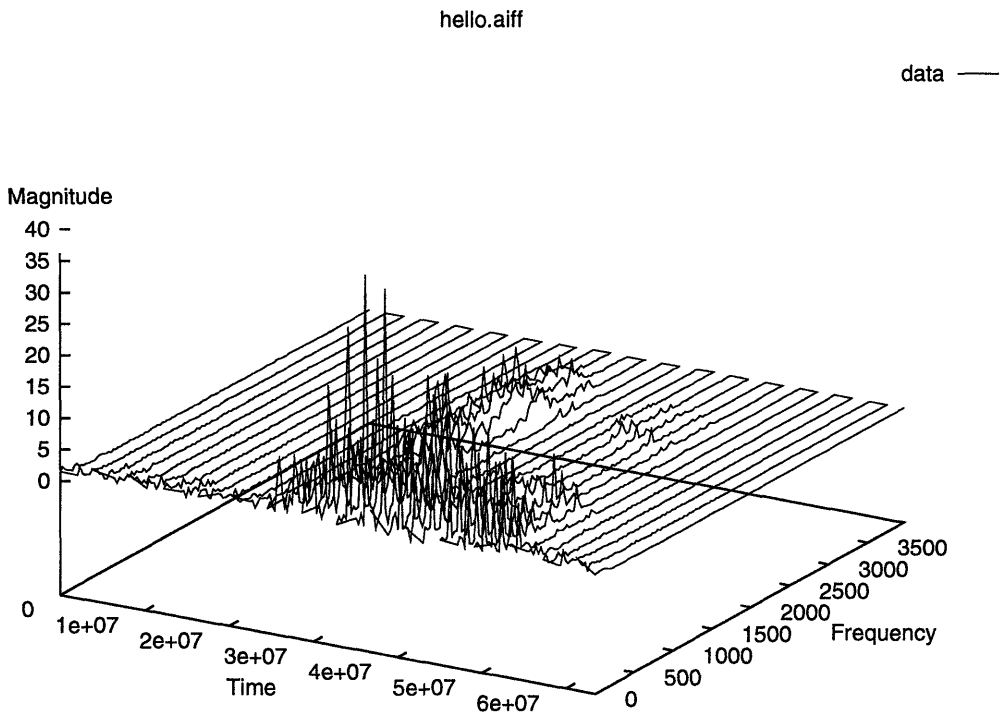

Figure 3-4: Sample FreqSlice of author saying "hello". Time is in internal units.

3.2.1 Signal Generation

A variety of signal generators are provided to do sine, cosine, square and triangle waves. There are obviously many more signal generators that could be useful including chirp, white noise, and maximal length. Because signal generators are so easy to write, and because they tend to be somewhat application specific, I haven't put much time into adding more, but rather expect users to define their own as appropriate to their work using the provided ones as templates.

Many useful algorithms can be implemented as combinations of specialized function generators and filters.

3.2.2 Fourier Transforms

Fourier transforms are functions that take a signal in time and return that signals spectrum, or vice versa. Fast Fourier Transforms, Discrete Fourier Transforms and their inverses are all provided in `libDsp`. There is a central dispatch routine that take a $\log_2$ to determine which transform to use. In the future, I hope to provide for better FFT algorithms to handle non-power of two cases, at which time the dispatch routine will be adapted to pass more signals to the more efficient FFT and less to the DFT.

3.2.3 Filters

Both IIR[3] and FIR[4] filters are provided for with a filter kernel structure. Because it is so easy to make good filters with windowing, I have provided Bartlett, Hanning and Hamming widow routines. An additional function is provided to convolve `TimeSignals`. It is envisioned that users will just create a `FreqSignal` of their desired filter response, convert it to a `TimeSignal` with an IFT, and window it with one of the above functions to get most filters.

Additionally, two non linear filters, full and half wave rectifiers, are provided.

[3]Infinite Impulse Response.
[4]Finite Impulse Response.

3.2.4 Transformation

Two functions, `Slice` and `Unslice` are provided to allow transformation between `TimeSignals` and `FreqSlices`. These rather generic spectrogram generation routines can be supplemented with specialized routines (eg. to take spectrum samples every half second) as needed, using these routines as templates.

3.3 Utility Code

There is a fair amount of code in the library that was included to make doing other things easier, though this code may have very little to do with signal processing. I will quickly outline some of the major sub-libraries here.

3.3.1 CommandLine

This is an class for parsing and querying switch settings and flags on the command line. It is used extensively in the examples, otherwise most of the example code would be command line parsing.

3.3.2 GnuPlot

A `FILE*` type interface is provided to GnuPlot. This is encapsilated by `CArray` but there is no reason it can't be used separately.

3.3.3 AIFF

The AIFF reading and writing routines were provided by Dan Ellis and converted to a sub-library to handle all the sound file I/O. I have only used a small portion of the possible functionality of his routines, though it is all available to `libDsp` users.

3.3.4 UniqueName

This is an class that provides a pseudo unique name when queried. It is used for creating temporary files when plotting with GnuPlot.

3.3.5 Miscellaneous

Functions such as zero padding and the like were written when needed and thrown in a generic toolbox sub-library. All of them are fairly special purpose.

3.4 Conclusion

This chapter has covered in broad overview the functionality of `libDsp`. The specifics of how to use the library are better addressed by the user's manual in Appendix ??.

Chapter 4

Applications

`libDsp` is a library that can be best appreciated when seen in action. It allows many simple sound manipulations to be done in correspondingly simple codes. This encourages exploration and experimentation, as the cost of trying a new idea is minimal. This kind of experimentation is exactly what `libDsp` was designed to encourage.

I will illustrate where `libDsp` can be useful by briefly showing a few of the applications were `libDsp` has already been applied. Many of these applications grew out of ideas that I had wanted to try for a while, but had been daunted by the technical hurdles to sound processing that I found existed on many systems[1]. With `libDsp` around I was able to code most of them up in under an hour.

4.1 Example Programs

The `libDsp` package comes with a number of sample programs to demonstrate some of the more basic uses of the library. I won't go into great detail here on each of them, as they are covered in the User's Manual which can be found in Appendix A.

[1]The all time winner is the NeXT, where using the sound chip requires at times sending it Motorola 56001 micro-code.

4.1.1 Intro

Intro is a short 13 line program that plots a sine wave in a graphics display window, then plots it's Fourier transform. It demonstrates the basics of what `libDsp` can do: read in sound files, process them in some way, and display the results.

4.1.2 Echo

Echo is perhaps even a more minimal program that reads in a sound file, and writes out a sound file created by taking the input sound and adding an echo. In all, this takes 3 lines of working code: one to read in the sound file, one to add the sound to delayed copies of itself, and one to write the result out. Again, since `libDsp` hides the specifics of I/O, array mathematics and delays, the amount of code needed to realize this is small.

4.1.3 AIFFdisplay

This program is a conversion of `Intro` to a command line driven display utility. It takes a sound file and plots either it, it's Fourier spectrum or its spectrogram in either a graphics window, or in a Postscript file for later printing. It is quite handy for a first pass analysis of a sound file.

4.1.4 AIFFmath

This is a program to add, subtract, multiply or divide mono AIFF files. It is included as a demonstration that without too much work, one could use a programs like Lex and Yacc or Borne Shell to implement scripting languages similar to CSound, though obviously ones that would not be as fast nor as capable of real time performance. Rather, this allows for special purpose scripting languages to be quickly assembled as needed.

In many respects it is a throw back to the kind of command line sound processing utilities that were used before `libDsp`, and shows how the existing codes for these could

be replaced by very short `libDsp` programs, probably enhancing their performance and definitely enhancing their portability and maintainability.

4.1.5 Voice Modification

This program was thrown together in 15 minutes after I had seen the movie "True Lies" and wanted to reproduce a voice distortion technique that had been used (down shifting the central frequency of the voice). The result is fun to play with, and at only half a page of code it illustrates how quickly you can implement fairly complex ideas using this library.

4.2 Data Hiding

`libDsp` was used in a body of research performed at the MIT Media Laboratory, News in the Future group over the summer of 1994 under the direction of Walter Bender. The research dealt with placing data of one form unobtrusively into the data of another, for example, text in sound. While many mediums where explored, sound received quite a bit of interest and this is where `libDsp` proved helpful.

As an aside, the work that was done on 2D image processing generated the beginnings of a library that might be included in `libDsp` at some future date (see Sub Section 5.3.2 for discussion).

4.2.1 Spread Spectrum

One of the first techniques pursued was that of Spread Spectrum signal encoding. This is a body of techniques developed for allowing sound to be transmitted via radio frequencies in a way that was uninterceptable and unjammable. It turned out to be equally useful for putting a text data stream into a sound.

`libDsp` was used for the sound file I/O, the manipulation of the sound samples, and the Fourier decoding of the result. Using the strategy mentioned in Sub Section 3.2.1, the spread spectrum systems were implemented as a group of specialized

signal generators. The resulting `TimeSignals` could be manipulated with the generic `TimeSignal` operators.

This worked quite well. The end result was fast enough to use for demonstrations to sponsors, and the C++ development environment gave us the flexibility we needed to "tweak" parameters to get the whole system working in a surprisingly short amount of time.

The code for this work is included in Appendix B.

4.2.2 Phase Coding

Another Data Hiding technique developed was based on the fact that the ear is only sensitive to relative phase, and not to absolute phase. This means that if you advance the phase for each frequency through time at a given rate, it doesn't matter what frequency you start at, the result will sound the same. Thus, you can encode information in the starting frequencies.

This technique was originally developed under Matlab, but it was found that Matlab could handle only about a 10 second sound sampled at 8kHz, so the method is being recoded in `libDsp` and seems to be coming along well.

This dual of developing theories in a Matlab-like environment and testing them in a `libDsp` environment seems very productive, and I will discuss this more in Chapter 6.

4.3 Conclusion

Though I have presented only a few examples, I hope that they illustrate the flexibility and ease of using `libDsp`. I've found that more than anything, having a package that takes care of the "grunge work" of signal processing has encouraged me to try more and more new ideas, many of which haven't worked (and thus are not included here) and a few that turned out to be quite interesting.

Chapter 5

Future Work

One of the design goals for `libDsp` was that it be as extensible as possible. A measurement of how successful I was in meeting this goal can be seen in the many areas that `libDsp` can be extended. I have divided my discussions of these possible extensions into several sections. The first covers those enhancements which are already provided for in the library. As such, implementing them involves only the coding or recoding of certain routines. Next, I try to outline what the inherent limitations of the `libDsp` design are, pointing out areas where small additions would involve a tremendous amount of work, and would perhaps be better implemented as a separate project rather than as an extention to this one. Third is a discussion of changes for which a serious redesign of the library would be involved, but which would greatly enhance its functionality.

5.1 Possible Enhancements

This section covers several areas: a variety of optimizations to the Fourier transform procedure, the infix arithmetic functions, and the plotting functions; some extensions for dealing with extremely large sound files and sound files of different formats; and additions to the existing signal generator and filter codes. All of these would be fairly straight forward to implement, and in many cases I have outlined one possible way to do so.

5.1.1 Fourier Transforms

These routines are some of the most time consuming of any in the package. The majority of the signal processing routines are $O(n)$, the FFT[1] routines are $O(n \lg n)$ and the DFTs[2] are $O(n^2)$[3].

This becomes a considerable burden when a ten second telephone quality signal has $n = 80,000$. At this n, a FFT takes roughly 16 times as long as most other processes, and a DFT takes *80,000* times as long.

Thus, since these routines are where most programs will spend the majority of their time, it is worthwhile to try to optimize them.

Native Data Format FFT

Currently, `libDsp` makes use of the FFT routines found in Numerical Recipes in C. These require a special format for the incoming data, and return the data in the same format. Thus to use these routines, the `TimeSignal` must reformat the data before sending it to the FFT, and reformat it again to put the result into a `FreqSignal`.

A speed increase might be realized if the entire FFT and IFFT[4] functions were recoded to make use of the raw `CArray` data format. This would eliminate a copy on each end, and probably lead to a speed increase in the transform routines.

Before this is undertaken, however, some experimentation should be done as to whether the Numerical Recipes data format or `libDsp`'s is faster to access. I suspect they should be nearly the same, but it might be that the cost of the two copies is insignificant compared to the difference in computation time afforded by the more optimal data format.

---

[1] Fast Fourier Transforms: A algorithm devised first by Gauss to quickly calculate transforms on signals with a power of two sample length.

[2] Discrete Fourier Transform: An algorithm for performing Fourier transforms on arbitrarily long signals

[3] This gives an idea of how much longer it takes for a routine to run with larger input. That is, if you quadruple the length of the input, an $O(n)$ routine runs four times as long, the $O(n \lg n)$ runs 8 times as long, and the $O(n^2)$ runs 16 times as long.

[4] Inverse Fast Fourier Transform.

Faster FFT for non-power of two data

Some thought needs to be given to how to speed up non-power of two transforms. Two possibilities present themselves at first glance. One would be to develop a variety of different power butterflies and use them in stages. This is somewhat complicated, though it has the potential for being the fastest algorithm. The downside to this method is that a factoring of the signal size must be done, and either an automatic butterfly constructor, or a large set of prime butterflies need to be included [ref OP and Shf].

The second option is to pad the incoming data out to a power of two. This can be quite a burden if the data is slightly larger than a power of two, (eg. 2049 samples long), as it involves nearly doubling the size (the 2049 sample signal increases to 4096 samples, and the running time by 218% over an optimal butterfly).

Despite the time it would take to implement this, it would probably pay off as either of these options would be preferable to the current scheme of taking a DFT. As discussed above, the DFT is an $O(n^2)$ operation.

Unifying FFT/IFFT code

There may not be a good reason to use a separate transforming engine in the FFT and IFFT functions. Since they are very similar, they could probably benefit by sharing a central computational routine. The existence of such a unified structure would greatly simplify the future optimizing and debugging of these functions.

5.1.2 Assembly Functions

As discussed in Sub Section 2.6.2, the infix math notation under C++ leads to an extra temporary object being created and an extra copy from it being performed. While infix notation is vastly preferable for writing code, once a code fragment is working, hand optimizing it with some assembly language style routines would make sense. Such an optimization typically leads to a 50% speed increase in the math for large arrays. `libDsp` could include several two argument and variable argument assembly

functions.

5.1.3 Very Long Sound Files

Sometimes it is necessary to write a program that will operate on sound files larger than the virtual memory of the machine you are on. In this case, implementing the change discussed in Sub Section 2.4.2 would allow processing to occur on sound files regardless of their length. As noted, such changes should be user transparent. The result would be that existing codes would run identically, but they would also work on longer sound files.

5.1.4 Automatic Sound File Conversion

Currently, `libDsp` only reads and writes AIFF files. I want to keep the I/O interface the same to allow dealing with very long sound files[5]. However, `libDsp` could convert from any input format to AIFF and back. To do this would require integrating a sound file translation utility (probably Sox) into the package.

When asked to read a non-AIFF file, it would translate the file to a temporary file (e.g., `/tmp/libDsp8273`) and read that. When done, it could just convert back. All this would, of course, be made transparent to the user.

5.1.5 Temporary Files and GnuPlot

Currently, `libDsp` uses a temporary file for plotting through GnuPlot. However, this is not necessary. It is possible to plot data directly through a pipe, and save disk space and time. Implementing this would speed plotting considerably.

5.1.6 Better Temporary File Handling

Temporary file names need to be improved, checked for uniqueness, moved to the `/tmp` directory and automatically deleted when the program terminates. None of these are

[5]see previous subsection. The AIFF routines include many functions to pull out datat from sound files a section at a time.

currently done.

5.1.7 Signal Generators

Phase in signal generators

All the signal generators could benefit from having an arbitrary phase argument, allowing the generator to not necessarily start at the beginning of a cycle.

Generic Signal Generator

A function which takes another function as an argument, and uses that function to generate a signal would be a useful addition to the signal generating package. A calling sequence like:

```
SignalGenerator a(sin, 100, 10.0, PI/2);
SignalGenerator b(cos, 100, 10.0, PI/2);
```

could be used to generate a sine or cosine wave at 100 Hertz for ten seconds with a 90° phase offset.

5.1.8 Spread Spectrum Sub-library

All the code developed for the spread spectrum research discussed in Sub Section 4.2.1 could be included as a sub-library. A number of such sub-libraries could be included to perform some of the more esoteric functions of signal processing.

5.1.9 Filters

There are many good filter construction algorithms in the public domain. Two that might be most useful would be Kaiser windows and the Parks McLauren filter design algorithm.

5.2 Inherent Limitations

Every design has some inherent limitations. I have tried to minimize these on `libDsp` by keeping the architecture as open and extensible as possible. However, there are some limitations that can't be avoided.

5.2.1 Real Time Capabilities

`libDsp` is inherently **not** a real time system. The whole design concept treats sounds as complete objects, and allows them to be manipulated without worrying whether the whole object exists yet or not. Allowing `libDsp` to be implemented with real time processing would require a basic change in the way the package is laid out. One possible approach is discussed in Sub Section 5.3.1 .

5.3 Future Research

Section 5.1 covers those extensions to `libDsp` which involve nothing more than coding. The following extentions might involve a fair amount of additional design work before they become practical.

5.3.1 I/O Stream Approach

One possible technique for real time signal processing in a C++ environment is to use I/O streams. Klee Diens at the Media Lab has supervised the development of a way to hook multiple stream filters together. One might reimplement much of `libDsp` as stream filter functions, and allow the filters to pull in from an input port and write out to an output one. If the internal filtering routines a fast enough, the result could be used for real time processing.

5.3.2 Arbitrary Dimensions

Currently, `libDsp` works only on one dimensional (ie. sound) signals. It would be useful if the library were generalized to allow for the same approach to be used on two dimensional (pictures) and three dimensional (movies/television) signals. Some preliminary work on two dimensions has been done already as part of the research into Data Hiding (see Section 4.2).

5.3.3 Hardware Hooks

There are several different types of DSP hardware installed in machines today, ranging from the integrated DSP processor in the NeXT to the various plug in boards available for the Intel PC market. The common denominator between all these is that they do signal processing orders of magnitude faster than may be done in software.

The Image Extentions to X11R6 provide an interesting approach to this problem in the world of images. The extention proscribes a family of functions, and provide working software versions of all of them. On machines where hardware is available to perform some of these tasks, those routines are recoded in the library as calls to the appropriate hardware.

The result is that code can be written that will run on all machines. On those machines that support hardware processing, the code simply runs faster.

5.4 Conclusion

`libDsp` is a very extensible design. As such, there are really no limits to what can be added to it. Just a few suggestions have been mentioned of what might be implemented. More ideas will have to come from users who use the library to do signal processing in their everyday work.

Chapter 6

Conclusion

This thesis has demonstrated that it is possible to construct a signal processing library that allows for quick and easy implementation of a wide range of signal processing applications. These applications can be created on one system, and then used on a wide variety of different systems, enhancing the portability of such applications.

Previously, there was a fairly large gap between developing an algorithm in a fast prototyping language like Matlab, and developing specialized C code or hardware to implement a useful version of it. Most algorithms run in Matlab are limited to processing just a few seconds of sound. This is often insufficient for testing and debugging a sample of useful length. The result is that often someone spends a tremendous amount of time implementing an algorithm in hardware or specialized C code, only to find that it had some problems that had to be worked out, in what now was a very costly medium (specialty C code, or worse still hardware).

`libDsp` is not meant to replace either a Matlab like environment or fast hardware or specialized C code, but rather to serve as an intermediate step between them where bugs can be caught, and parameters optimized much more cheaply than they could be before. `libDsp` also provides a natural stopping place for many algorithms that run just fine at the speed `libDsp` runs; many times it is acceptable for an algorithm to run in a few minutes, especially if you are doing experiments and not writing commercial software.

I have found that the existance of such a generalized sound processing library has

encouraged my to try sound manipulation ideas with that I would probably not have tried if each implementation taken several days to code. The code that is created can be shared among many people without modification.

This suggests that perhaps a uniform signal processing library would be a valuable addition to the libraries distributed with computers today. The use of such a standardized interface could do for sound what X windows did for graphics, with the advantage that such a library could be assembled with modern object oriented methods (as demonstrated in this thesis) to make it both powerful and easy to use.

Appendix A

libDsp user's manual

libDsp

A C++ Object Oriented DSP Library
Users Guide and Reference Manual
Edition 1.4.4 of this manual
for use with Version 2.0.0 of libDsp

Daniel F. Gruhl

This library was developed in fulfillment of the Master of Engineering Thesis Requirement for Course VI at MIT. The work was done for Prof. Barry Vercoe of the MIT Media Lab. As part of this work derives in part from the libg++ library, please see the libg++ library disclaimer for distribution of this work.

1 Introduction

libDsp is a C++ object oriented implementation of a digital signal processing library. If you understood the previous sentence, you are well on your way to using libDsp. If you didn't, you probably need some more background in either computer programming or signal processing.

The remainder of this manual assumes you have at least passing knowledge of C and C++ (though if you understand C and another Object Oriented language you shouldn't be in bad shape) and the very basics of signal processing.

1.1 History — Or where did libDsp come from

Despite any evidence to the contrary, libDsp did not "just growed". It was developed to allow for reasonably fast development of signal processing programs for music processing.

I developed it after spending a semester trying to do analysis of inter-note transitions in human singing. I found that most of my time was spent trying to chase down people to find out how their code worked, finding out what their code actually did, converting from one person's program output to another person's program input and so on...

I decided it would be nice if all the code necessary for basic signal processing was available in one place with a consistent interface that was easily extensible, reasonably fast and reasonably complete. libDsp is a result of that effort.

1.2 Format — How this manual is laid out

I find that the best way to learn how to use a library is to jump right in and use it. As a result, the first part of this manual is a series of example programs highlighting how to use the library. Following that is a reference manual listing all the functions and what they do.

1.3 Contributions

libDsp will only continue to grow if people contribute to it. I am not an expert in signal processing, nor in any of the esoteric arts of music processing. However, I am more than happy to

integrate functions you use in every day life into libDsp for others to use. If you wish to contribute, please observe the following:

1. I need a detailed description of what your code does. In english. "See program comment statements" won't fly.
2. I need to know who you are (name, address, e-mail). This is so I can get in contact with you again if I need to.
3. You need to release the code into the public domain. For now, a statement to that effect is sufficient. Later, I guess if I'm ever going to submit this to the FSF they want a written notice. libDsp is free code, so NO PROPRIETARY CODE!!!
4. Send the code to <druid@mit.edu>
5. Accept my thanks :)

1.3.1 Contributors

- The whole AIFF reading and writing routine package is courtesy of Dan Ellis <dpwe@media.mit.edu>

2 Examples

The best way to learn how to use libDsp is to play with it. To this end I will try to provide a few examples of what libDsp is typically used for. All of the code in this section can be found in the examples subdirectory.

2.1 intro

The first example is creating a signal, displaying it, taking a it's fourier transform and displaying the transformed data.

```
// intro.cc
// An introduction to what libDsp can do

#include "Dsp.h"

main(){
  // Open a plotting window
  GPlot* g = GPopen();

  // Define a signal to use
  TimeSignal a = DSPSinSignal(100, 10.24, 10.0/10.24);

  // Plot that data
  a.gplot(g, "Sample Data");

  // Wait for entry
  cout << "Press Enter to continue...\n" << flush;
  pause();

  // Take the fourier transform and store
  // it in a
  FreqSignal b = DSPFFT(a);

  // Plot the Fourier Transform
  b.gplot(g, "Fourier Transform of Sample Data");

  // Wait for enter
  cout << "Press Enter to continue...\n" << flush;
  pause();

  // Be nice, close your gnuplot window
  GPclose(g);
}
```

2.2 Echo

```
#include <Dsp.h>


main(int argc, char** argv)
{
    TimeSignal in, out;

    in.readAiff("sample.aiff");
    out = in + in.delay(2.0) + in.delay(3.0);
    out.writeAiff("sample1.aiff");

}
```

'echo' is a quick little program that implements an echo. Because the delays are double's, the mean "seconds" rather than "samples". Note, you can add the results of delay as if the were normal time signals (the extra samples fall off the end, leaving the result the same length as the original...).

2.3 AIFFmath

'AIFFmath' is a simple program that illustrates command line parsing, reading and writing AIFF files, and some simple operations with TimeSignals.

Following is the code for the example. Following the code, I will discuss some of the features in it.

```
/* FILE: /User/druid/src/c++/ALPHA-libDsp/libDsp/examples/AIFFmath.cc */
/* AUTHOR: Daniel F. Gruhl <druid@mit.edu> */
/* DATE LAST MODIFIED: Mon Apr 18 19:28:07 EDT 1994 */
/* DESCRIPTION: Do simple math with CArray files */


#include <stdlib.h>
#include <iostream.h>
#include "Dsp.h"

// This is the message that AIFFmath will use if it needs
// to display help.

const char* help_msg =
"Usage: %s <math command> aiff1 aiff2 aiff3\n"
"   where <math command> is one of\n"
```

```
"      -a    aiff3 = aiff1 + aiff2 \n"
"      -s    aiff3 = aiff1 - aiff2 \n"
"      -m    aiff3 = aiff1 * aiff2 \n"
"      -d    aiff3 = aiff1 / aiff2 \n\n"
"      -h    Generate this help.\n";

main(int argc, char** argv){

    // Parse Command Line
    CommandLine CL(argc, argv);
    CL.help_set(help_msg);
    CL.die_on_switch("h");
    CL.die_if_switch_count_not_between(1, 1);
    CL.die_if_switch_not_one_of(4, "a", "s", "m", "d");
    CL.die_if_param_count_not_between(3, 3);


    // Allocate Storage
    TimeSignal aiff1, aiff2, aiff3;

    // Load in working files
    aiff1.readAiff(CL.csParameter(0));
    aiff2.readAiff(CL.csParameter(1));

    // If the files are of different time bases (ie. sampling rates)
    // we have a problem...
    if (aiff1.time_base() != aiff2.time_base()){
        cerr << "ERROR: Cannot do math on aiff sampled at different rates.\n";
        exit(1);
    }

    // Resize Storage Area of smaller AIFF file
    if (aiff1.size() > aiff2.size()) aiff2.stretch(aiff1.size());
    else if (aiff1.size() < aiff2.size()) aiff1.stretch(aiff2.size());

    // Do the math
    if (CL.switch_set("a")) aiff3 = aiff1 + aiff2;
    else if (CL.switch_set("s")) aiff3 = aiff1 - aiff2;
    else if (CL.switch_set("m")) aiff3 = aiff1 * aiff2;
    else if (CL.switch_set("d")) aiff3 = aiff1 / aiff2;


    // Write out the result
    aiff3.writeAiff(CL.csParameter(2));

    cout << "Successful Completion\n";
    exit(0);
}
```

Ok, starting at the top. We need `#include <stdlib.h>` to get the proper definition for `exit`. `#include <iostream.h>` is for all the input and output we do, and `#include "Dsp.h"` is for all the libDsp definitions.

The first actual code is the declaration of *help_msg*. This message, cryptically enough, is the one we will be displaying whenever we want to give the user help. The format for the string is defined in the CommandLine object. Basically, there must be one '`%s`' which will get filled with the name of the command when it is run (from *argv[0]*) and the rest should be formatted the way you want it to print under `fprintf`.

While a command line help is no replacement for a good man page (which is no replacement for a good texinfo file...) it does provide a quick reminder of usage. You should include all the switches, what they do and what all the parameters mean, along with a brief description of what your program does.

Moving onwards, the program body starts. The `main` function needs to have the *argc* and *argv* arguments in order to read in the command line.

```
CommandLine CL(argc, argv);
```

Next, the object *CL* of type `CommandLine` is declared. The declaration includes a pass of the argument information from the command line, which cues *CL* to parse the command line.

```
CL.help_set(help_msg);
```

Next, a call is made to `help_set` to tell *CL* what to print when we need to print the user a help message.

```
CL.die_on_switch("h");
```

This code checks if the user has typed a '`-h`' requesting help from the command line. If they have, we print help and die.

```
CL.die_if_switch_count_not_between(1, 1);
```

We only want one switch. we wouldn't know what to do if we got more, so we should just die if there is less than 1 switch, or greater than 1 switch set.

```
CL.die_if_switch_not_one_of(4, "a", "s", "m", "d");
```

The only valid switches are a, s, m and d. If any other switch than these are set, we should print the help and quit.

```
CL.die_if_param_count_not_between(3, 3);
```

We need three and only three file names. The first two will be input files. The last will be the output file. If we don't get this many token's, this is an error and we should die printing the help message.

That takes care of the command line parsing. Next we allocate our TimeSignals and read two of them in from AIFF files. If any trouble is encountered during these read ins, the internals of `readAiff` will cause them to die issuing an error.

```
    if (aiff1.time_base() != aiff2.time_base()){
cerr << "ERROR: Cannot do math on aiff sampled at different rates.\n";
exit(1);
    }
```

Next we check to make sure the signals we read in where sampled at the same rate. If not, we should stop.

```
// Resize Storage Area of smaller AIFF file
if (aiff1.size() > aiff2.size()) aiff2.stretch(aiff1.size());
else if (aiff1.size() < aiff2.size()) aiff1.stretch(aiff2.size());
```

Next we pad the smaller AIFF file with zeros so that they are now the same size.

```
// Do the math
if (CL.switch_set("a")) aiff3 = aiff1 + aiff2;
else if (CL.switch_set("s")) aiff3 = aiff1 - aiff2;
else if (CL.switch_set("m")) aiff3 = aiff1 * aiff2;
else if (CL.switch_set("d")) aiff3 = aiff1 / aiff2;
```

Next we just do the math. If this looks a lot like regular `C` math, then I should be well on my way to a thesis. That's the idea.

Lastly, we write the file out and send the user a useful message. That's all there is to it.

2.4 AIFFdisplay

```
/* FILE: /User/druid/src/c++/ALPHA-libDsp/libDsp/examples/AIFFdisplay.cc */
/* AUTHOR: Daniel F. Gruhl <druid@mit.edu> */
/* DATE LAST MODIFIED: Mon Apr 25 17:20:29 EDT 1994 */
/* DESCRIPTION: */


#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include "Dsp.h"

const char* help_msg =
"usage: %s -spec -<s|p> <output.ps> filename1.AIFF... \n"
"\n"
"    -spec means to output a spectrogram\n"
"    -s means display to screen\n"
"    -p means display to postscript\n"
"    -h display this help\n"
"\n"
" This program will display 16 bit AIFF files to either the screen or\n"
" into a postscript file for later plotting.\n";

typedef enum {POSTSCRIPT=1, SCREEN=0} plot_mode;
typedef enum {True=1, False=0} boolean;

#define SIZE 128

main(int argc, char** argv){
    GPlot *g;
    plot_mode pm;
    boolean slice_it;

// Parse the command line;
// The CommandLine object is fully documented in the
// Texinfo document for libDsp

    // Loads and parses command line
    CommandLine CL(argc, argv);

    // Sets the help message that will be displayed if
    // need to the one above.
    CL.help_set(help_msg);

    // If -h has been set, give the user help
    CL.die_on_switch("h");

    // Make sure only one or two switches have been set.
    CL.die_if_switch_count_not_between(1,2);
```

```
// Make sure they are switches I recognize
CL.die_if_switch_not_one_of(3, "p", "s", "spec");

// If one f them is spec, flag that we are to compute a
// spectrogram
if (CL.switch_set("spec"))
    slice_it = True;
else
    slice_it = False;

// Decide if we are displaying to the screen or a
// postscript file
if (CL.switch_set("s")) {
    pm = SCREEN;
    g = GPopen();
} else {
    pm = POSTSCRIPT;
    g = GPfopen(CL.csParameter(0));
}


// Alocate working data space
TimeSignal indata;
FreqSlice  inslice;
long j; double d;

// Loop... Note, if postscript, we are starting with the
// 2nd parameter, as the first was the ps filename
for (long i=pm; i<CL.how_many_params(); i++){
    // Read in the AIFF file
    indata.readAiff(CL.csParameter(i));

    // If we are making a spectrogram, cut the data up
    // and fourier transform it
    if (slice_it == True){
        // This hack limits floating point
        // overflows
        d = Maxabs((CArray &) indata);
        indata /= d;
        inslice = Slice(indata, SIZE);
        for (j=0; j<inslice.Windows(); j++)
            inslice[j] /= d;
        inslice.gplot(g, CL.csParameter(i));

    } else {
        indata.gplot(g, CL.csParameter(i));
    }
```

```
        // If we are displaying to screen, pause after
        // printing each graph
        if (pm == SCREEN){
            cout << "Press <ENTER> to continue.\n" << flush;
            pause();
        }
    }

    // Close all files, kill all spawned processes
    GPclose(g);

    // And leave
    exit(0);
}
```

2.5 TimeStretch

```
#include <iostream.h>
#include <math.h>
#include "Dsp.h"
#include <String.h>

const char* help_msg =
"usage: %s <stretch percentage> <infile> <outfile>\n"
"       where strtch percentage is 1.3 for 130%%\n";

main(int argc, char** argv){
    long WINDOW_SIZE = 128;

    // Parse Command Line
    CommandLine CL(argc, argv);
    CL.help_set(help_msg);
    CL.die_on_switch("h");
    CL.die_if_switch_count_not_between(0, 1);
    CL.die_if_param_count_not_between(3, 3);

    GPlot* g = GPopen();
    double stretch = atof(CL.csParameter(0));
    String infile  = CL.Parameter(1);
    String outfile = CL.Parameter(2);
```

```
// LoadData
TimeSignal tin;
tin.readAiff(infile);

// Checkpoint
tin.gplot(g, "Data in");
cout << "Press enter to continue.\n" << flush;
pause();

FreqSlice fsin = Slice(tin, WINDOW_SIZE);

fsin.gplot(g, "Spectrogram", 1);
cout << "Press enter to continue.\n" << flush;
pause();

long New_Number_Of_Windows = (long) (stretch * (double) fsin.Windows());

FreqSlice fsout(New_Number_Of_Windows, fsin.WindowSize());

double *angle1, *angle2;
double *mag1, *mag2;
double *current_ang = new double[fsout.Windows()];

long ref, j;
double delta;
double ang, mag;

long sz = fsin[ref].size();
angle1 = new double[sz];
angle2 = new double[sz];
mag1   = new double[sz];
mag2   = new double[sz];


for (long i=1; i<fsout.Windows()-1; i++){        // HACK!!!
    ref = (long) ((double) i / stretch);
    delta = (double) i / stretch - (double) ref;

    angle1 = fsin[ref].AngleArray(angle1);
    angle2 = fsin[ref+1].AngleArray(angle2);
    mag1   = fsin[ref].MagnitudeArray(mag1);
    mag2   = fsin[ref+1].MagnitudeArray(mag2);

    for (j=0; j<fsout.WindowSize(); j++){
        ang = current_ang[j] += (angle2[j] - angle1[j]) * delta;
        mag = (mag2[j] - mag1[j]) * delta + mag1[j];
        fsout[i][j] = polar(mag, ang);
    }
```

```
        }

        delete [] angle1;
        delete [] angle2;
        delete [] mag1;
        delete [] mag2;

        TimeSignal tout;
        tout = UnSlice(fsout);

        tout.gplot(g, "Data out");
        cout << "Press enter to continue.\n" << flush;
        pause();


        tout.writeAiff(outfile);

        GPclose(g);
    }
```

3 Classes

LibDsp is based on a number of Classes designed to aide in digital signal handling. Many of the functions in these can be safely ignored by casual users of the library as they are used only by the internals of the DSP toolkit.

The idea for placing the signals in a "Hierarchy" of Classes is lifted from Barry Vercoe's Csound program. This technique allows the user a great deal of flexibility in how they deal with their data, while still hiding as much of the implementation detail as possible.

3.1 CArray - Complex data type array

CArray Base Signal Processing Class

CArray is the base class for all other signals in the libDsp library. The complex array data type provides the ability to perform vector operations such as addition, subtraction, element wise multiplication and division and cross products. The Complex data type this class is based is the GNU g++ Complex data type. I have chosen not to use the GNU AVec prototype for my vectors as I wanted the class to have more flexibility than AVec provides. I have, however, borrowed several ideas from the AVec template.

A CArray can be created one of two ways. It can be initialized to have a number of elements, or it can just be created as an empty array with the number of elements to be set later with a resize command.

Example:

```
#include <Dsp.h>

main(){
    CArray a, b(10);    //a is an empty array, b is an array of 10 elements
                        //empty arrays are actually implemented as arrays of
                        //1 element
}
```

A couple of warnings regarding CArray. First, the array has an automatic destructer. This means that when the array goes out of scope, all information in it is destroyed. Thus, if you have pointers to elements of a CArray, and the array goes out of scope, these pointers will no longer be valid.

Secondly, for those of you from Fortran land, CArray uses the standard C convention of numbering arrays starting at 0. Thus and array of 10 elements goes from 0...9, not 1...10.

Next, briefly, here are the internal data types defined and used by a `CArray`:

CArrayGiveAway Passing Variable

This is an internal variable used for fast passing in CArray. It holds a pointer to data, and size of data information. As just a pointer is passed, rather than a whole array recopied, it is a fast way to pass data if you are not again going to use the source.

Following are the various user functions provided by CArray.

`void` **CArray::CArray** (`long` *items*) Function
`void` **CArray::CArray** (`long` *items*, `Complex*` *data*) Function
`void` **CArray::CArray** (`CArrayGiveAway&` *c*) Function

These are all ways of initializing a `CArray`. The first provides just the size of the CArray in items, the second provides the size and the data (which CArray then owns) and the last is an internal passing mechanism to allow fast copying.

`long` **CArray::size** () Function

Returns the number of items in the current Complex Array.

Example:

```
#include <iostream.h>
#include <Dsp.h>

main(){
  CArray A(10);
  cout << "The variable A has " << A.size() << " elements in it.\n"
       << flush;
}
```

`long` **CArray::setsize** (`long` *newsize*) Function

This is the internal resetting routine. It should not be called by users.

void CArray::resize (long *newsize*) Function

Resize deletes the current array and creates a new array of the size requested for the object. Note, all pointers, references, etc. are lost to the original array. If you want the array data to stay the same, use stretch instead.

Example:

```
#include <Dsp.h>

main(){
        CArray a, b(10);

        b[3] = 4;

        a.resize(4);   // a is now an array of 4 elements
        b.resize(4);   // b is now an array of 4 elements. element
                       // 3 is NO LONGER set to (4, 0)!!!
}
```

void CArray::stretch (long *newsize*) Function

Stretch is sort of like resize, but it copies over as many elements as will fit into the new array. Note, that new storage space is allocated for the new array, so pointers will no longer be valid.

If stretch allocates new storage space, that space will be set to zero initially.

Example:

```
#include <Dsp.h>

main(){
        CArray a, b(10), c;

        b[3] = 2;
        b[5] = 4;
        c = b;

        a.resize(4);   // a is now an array of 4 elements
        b.resize(4);   // b is now an array of 4 elements.
                       // element 3 is still set to (2, 0)
                       // but element 5 is lost.
        c.resize(12);  // Just like b, but now there are two more
                       // elements, both zero at the end of the
                       // array.

}
```

`CArray` **CArray::slice** (`long` *first*, `long` *last*) Function

Slice returns a CArray made up of the elements of a current CArray from element first through and including element last.

Example:

```
#include <Dsp.h>

main(){
      CArray a(5), b;
      a[0] = 2;
      a[1] = 8;
      a[2] = 4;
      a[3] = 9;
      a[4] = 3;
      b = a.slice(2,4);
      // b == {4, 9, 3}
}
```

`void` **CArray::showme** (`long` *i*) Function

An internal debugging routine. Prints the value of the array item at location *i*. Useful in 'GDB' and other debuggers without full support for 'C++'.

`CArrayGiveAway` **CArray::giveaway** () Function

Passes the data from a CArray into a CArrayGiveAway. The CArray is no longer usable and should be ignored or destroyed after this routine is called.

`void` **CArray::gplot** (`GPlot*` *g*, `char*` *title*, `char*` *xlabel*, `char*` *ylabel*, `double` *offset*, `double` *delta*, `double` *stepoff*) Function

Gplot is the routine used for display of data from a CArray to the screen or a Postscript file. *g* is a gplot window opened by `GPfopen()`. *title* is the title of the plot, *xlabel* and *ylabel* are the labels applied to the corresponding axis, *offset*, *delta* and *stepoff* control the value on the left side of the graph, the delta between each data point, and the point in the data array to call index 0 (with wrap around). These are usually set by the higher level calling object.

CArray CArray::operator= (const CArray& *rhs*) Function
CArray CArray::operator= (const int& *rhs*) Function
CArray CArray::operator= (const float& *rhs*) Function
CArray CArray::operator= (const double& *rhs*) Function
CArray CArray::operator= (const Complex& *rhs*) Function

The equal operator performs in two very different ways. First, it can be used as a regular vector equator, in which case it takes on the size and contents of whatever it is supposed to equal. Secondly, it can be used to set an entire array to some constant value. In this case the array size is unchanged. For CArray, all arithmetic operators are defined for CArray and int, float, double and Complex data types.

Example:

```
#include <Dsp.h>

main(){
      CArray a(2), b(2), c(4);

      a[0] = 1;
      a[1] = 2;

      b = 3.4;        // b[0] and b[1] get set to 3.4
      c = a;          // c is resized to two elements which are
                      // set the same as a

}
```

CArray CArray::operator+ (*) Function
CArray CArray::operator- (*) Function
CArray CArray::operator* (*) Function
CArray CArray::operator/ (*) Function
CArray CArray::operator+= (*) Function
CArray CArray::operator-= (*) Function
CArray CArray::operator*= (*) Function
CArray CArray::operator/= (*) Function

All the basic math types, +, -, * and / are supported on an element by element basis. Additionally, +=, -=, *= and /= are supported, again on an element by element basis. The actual complex math is handled by the Complex data type, CArray simply provides the machinery to the complex operations on a vector basis.

Example:

```
#include <iostream.h>
#include <Dsp.h>
```

```
main(){
 CArray a(2), b, c, d, e, f;
 Complex CNum(1,2);

 a[0] = 1;
 a[1] = 2;

 b = a + 2;
 c = 4.7 - b;
 d *= a;
 e = d / b;
 f -= CNum;

 cout << a << " " << b << " " << c << " " << d
      << " " << e << " " << f << "\n";
}
```

CArray CArray::operator<< () Function

This operator formats a CArray for somewhat reasonable printing. At the moment this is a hack, and not a real iostream operator.

3.2 Time Signal

TimeSignal isa CArray. Signal processing class

A Time Signal is-a CArray with one additional variable associated with it, that being the Sampling Rate which that data was taken at. This sampling rate is known as the "time base" for the object.

Time Signals can be created somewhat differently from CArrays. Instead of specifying the size of the array, (which you can still do), you may instead specify the duration of the signal. Alternately, you can just specify the length of the signal in samples, and accept a sampling rate of 1 sample per second.

Example:

```
#include <Dsp.h>
```

```
main(){
      long samples = 1;
      double duration = 1.0;
      TimeSignal a(100, duration), b(100, samples);
                              //a is a TimeSignal sampled
                              // at 100 Hz and of a duration
                              // one second (hence it has 100
                              // samples). b is a TimeSignal
                              // sampled at 100 Hz with only
                              // one sample.
}
```

All operations that could be done with CARRAYS are fully supported for TimeSignals, with some additions:

double TimeSignal::time_base () Function

This just returns the current sampling rate of this time signal.

Example:

```
#include <iostream.h>
#include <Dsp.h>

main(){
   TimeSignal a(100, 1.0);
   cout << "a has a sampling rate of " << a.time_base() << " \n";
}
```

double TimeSignal::duration () Function

This just returns the current duration of this time signal.

Example:

```
#include <iostream.h>
#include <Dsp.h>

main(){
   TimeSignal a(100, 1.0);
   cout << "a has a sampling duration of " << a.time_base() << " \n";
}
```

void TimeSignal::data_out (char* filename) Function

This is the low level dump of a TimeSignal. The output file is ASCII write out in the following format:

```
<time in seconds of this sample> <real part> <imaginary part>
```

The procedure is called with an argument of the filename to write the data into. This routine is provided mainly for debugging purposes.

Example:

```
#include <iostream.h>
#include <Dsp.h>

main(){
   TimeSignal a(100, 2);
   a[0] = 5;
   a[1] = 3;
   a.data_out("data_file.dat");
}
```

void TimeSignal::readaiff (char* filename) Function

Read AIFF loads a 16-bit AIFF file into a time signal. Only 16 bit AIFF file are supported at this time.

Example:

```
#include <iostream.h>
#include <Dsp.h>

main(){
   TimeSignal a;
   a.readaiff("soundin.1"); // Read in an aiff file.
}
```

void TimeSignal::writeAiff (char* filename) Function

Write AIFF writes a 16-bit AIFF file from a time signal. Signal is scaled so the largest magnitude in the signal corresponds to 32,000.

Example:

```
#include <stdio.h>
#include <iostream.h>
#include "Dsp.h"
```

```
main(int argc, char** argv){
  TimeSignal a;
  a.readAiff(argv[1]);          // Read in an aiff file
  a.writeAiff(argv[2]);         // Write out an aiff file
}
```

TimeSignal **TimeSignal::delay** (long amount) Function
TimeSignal **TimeSignal::delay** (double amount) Function

Delay allows shifting of a TimeSignal. This shift can be defined in samples, or in seconds. A positive delay results in a later signal. Zeros are shifted in and the data shifted out is lost.

Example:

```
#include <Dsp.h>

main(){
      TimeSignal t(2.0, 10), r, s;
      t[5] = 1;

      r = t.delay(1);     // r[6] is now 1;
      s = r.delay(-2.0); // s[5] is again 1
                          // (-2 seconds of shift)
}
```

3.3 Frequency Signal

FreqSignal isa CArray -- Signal processing class

A Frequency Signal is-a CArray with the addition of a variable to track the central frequency in each "bin". While all internal calculations are done in 2pi radians for the signal length, for convenience interfaces to the user present the represented frequency.

Frequency Signals are created exactly the same way as CArrays, by specifying the length. All math valid for CArrays is also valid for frequencies.

Example:

```
#include <Dsp.h>
```

```
main(){
      FreqSignal a(10); // a is a frequency spectrum with 10 bins.
}
```

All operations that could be done with CARRAYS are fully supported for FreqSignals, with some additions:

`double` **FreqSignal::freq_per_bin** () Function

This just returns the current delta frequency per bin in Hz.

Example:

```
#include <iostream.h>
#include <Dsp.h>

main(){
   FreqSignal a(10);
   cout << "a has a delta freq per bin of " << a.freq_per_bin() << " \n";
}
```

`double` **FreqSignal::frequency** (`long bin`) Function

Returns the central frequency for the given bin.

Example:

```
#include <iostream.h>
#include <Dsp.h>

main(){
   FreqSignal a(100);
   cout << "a[1] has a central frequency of " << a.frequency(1) << " \n";
}
```

`Frequency` **Signal::data_out** (`char* filename`) Function

This is the low level dump of a FreqSignal. The output file is ASCII write out in the following format:

```
<frequency in radians> <real part> <imaginary part>
```

The procedure is called with an argument of the filename to write the data into. This routine is provided mainly for debugging purposes.

Example:

```
#include <iostream.h>
#include <Dsp.h>

main(){
   FreqSignal a(2);
   a[0] = 5;
   a[1] = 3;
   a.data_out("data_file.dat");
}
```

3.4 Frequency Slice

FreqSlice has a FreqSignal. Signal Processing Class

The Frequency Slice data type is just an array of frequency signals. It is used to hold a set of data for display. The format would typically be used to show a Time Signal's spectrogram.

This sample program reads in an AIFF file, takes every 5th Window of 128 sample points, takes their FFT, stores them in a FreqSignal, then plots that FreqSignal to a Postscript file.

```
#include <stdio.h>
#include <iostream.h>
#include "Dsp.h"

#define SLICES 128
#define MULT 5

main(int argc, char** argv){
  TimeSignal a, b(128);
  GPlot* g;
  char buffer[128];

  g = GPfopen(argv[2]);
  a.readAiff(argv[1]);

  FreqSlice f(SLICES, 128);
```

```
    long j;
    for (long i=0; i<SLICES; i++){
        for (j=0; j < 128; j++)
    b[j] = a[i*128*MULT + j];
        f[i] = DSPFFT(b);
        cout << i << "\n" << flush;
    }

    f.gplot(g, argv[1] );


    GPclose(g);

    cerr << "Done...\n" << flush;
}
```

`void` **FreqSlice::FreqSlice** (`long` *Windows*, `long` *samples-per-window*) Function

This is the initializer for the class. You set how many windows you need, and how many samples in each window.

`void` **FreqSlice::~FreqSlice** () Function

This is the class distructor. It just calls the destructors of every FreqSignal in the data array.

`FreqSignal` **FreqSlice::operator** (`long` *Index*) Function

Returns the *Index*th `FreqSignal` of the `FreqSlice`.

`void` **FreqSlice::set_delta_time** (`double` *dt*) Function

Sets the internal delta t between the centers of each slice.

`double` **FreqSlice::time_of_window** (`long` *window*) Function

Returns the central time of the *window*th window in the `FreqSlice`.

`void` **FreqSlice::gplot** (`GPlot*` *g*, `char*` *plot_title*, `long` *every*) Function

Plots the data stored in the `FreqSlice`. The graph has the title *plot_title* and is plotted in the GnuPlot window *g*. I found that these plots can get very messy (and not very informative) if you plot all the data of even a short signal. Thus, it plots every *every* signal instead (set this to 1 to plot everything, if set to 3 for example, this will print every third spectrum).

4 Tool Box

The DSP toolbox provides some basic tools for manipulating the data types above. These are a kind of catch all for routines that don't fall under a particular class.

4.1 Signal Generators

libDsp provides several basic signal generators. Please send me your favorites and I'll try to include them in my next release. All signal generators share the following format:

```
TimeSignal SigGen( Sampling Rate, Duration(seconds), Frequency);
```

`TimeSignal` **DSPSinWave** (`double sampling-rate, double duration,` double frequency) Function

This Simply returns a sin wave of the duration and frequency requested.

Example:

```
#include <Dsp.h>
main(){
  TimeSignal a = DSPSinWave(1000, 5.5, 60);
       // a is a sine wave sampled at 1000 hz, of duration
       // 5.5 seconds and a frequency of 60 Hz
}
```

`TimeSignal` **DSPCosWave** (`double sampling-rate, double duration, double frequency`) Function

This Simply returns a cosine wave of the duration and frequency requested.

Example:

```
#include <Dsp.h>
main(){
  TimeSignal a = DSPCosWave(1000, 5.5, 60);
       // a is a cosine wave sampled at 1000 hz, of duration
       // 5.5 seconds and a frequency of 60 Hz
}
```

`TimeSignal` **DSPSquareWave** (`double sampling-rate, double duration,` double frequency) Function

This Simply returns a square wave of the duration and frequency requested.

Example:

```
#include <Dsp.h>
main(){
  TimeSignal a = DSPSquareWave(1000, 5.5, 60);
}
```

`TimeSignal` **DSPTriangleWave(double** `sampling-rate, double duration, double frequency)` Function

This Simply returns a triangle wave of the duration and frequency requested.

Example:

```
#include <Dsp.h>
main(){
  TimeSignal a = DSPTriangleWave(1000, 5.5, 60);
}
```

4.2 Special Filters

libDsp provides some non linear filters that appear in signal processing.

`TimeSignal` **HalfWaveRectifier** (`TimeSignal& t`) Function

A Half Wave Rectifier (HWR) takes a time signal t, and returns another time signal where any non positive portion of the signal is reset to zero. This simulates the behavior of a shunt diode.

Example

```
#include <Dsp.h>
```

```
main(){
     TimeSignal t = DSPSinSignal(100, 5, 20);
     TimeSignal t2 = HalfWaveRectifier(t2);
     GPlot* g = GPopen();
     t2.gplot(g);
     pause();
     GPclose(t2);
}
```

`TimeSignal` **FullWaveRectifier** `(TimeSignal& t)` Function

A Full Wave Rectifier (FWR) takes a time signal t, and returns another time signal where the resulting signal is the absolute value of the original signal.

Example

```
#include <Dsp.h>

main(){
     TimeSignal t = DSPSinSignal(100, 5, 20);
     TimeSignal t2 = FullWaveRectifier(t2);
     GPlot* g = GPopen();
     t2.gplot(g);
     pause();
     GPclose(t2);
}
```

4.3 Fourier Transforms

Fourier transforms allow you to make a frequency representation of a time signal and vice versa

libDsp makes a distinction between FT's and IFT's. It also provides an intelligent interface to these routines. Please note, by working in powers of two you get a much faster FT.

`FreqSignal` **DSPFT** `(TimeSignal& t)` Function

The Fourier Transform is a simple dispatching routine that decides if a sample is a candidate for FFT, or rather it must avail to DFT. It is recommended that you use this whenever possible. As the FFT routine becomes more capable, this routine will begin to dispatch more signals to it.

Example:

```
#include <iostream.h>
#include "Dsp.h"
main(){
  TimeSignal swave1 = DSPSinSignal(100, 5.12, 1/5.12*10.0);
  FreqSignal spec1  = DSPFT(swave1);
  swave1.data_out("samp1");
  spec1.data_out ("samp2");
}
```

`FreqSignal` **DSPFFT** `(TimeSignal& t)` Function

The Fast Fourier Transform is the fastest way to take a FT. However, it is limited to performing only on signals of a length 2**n. When you can use it, though, it computes several orders of magnitude faster than the DFT.

Normally, you don't have to use DSPFFT directly. If you just call DSPFT, that function will dispatch to the appropriate fourier transform. However, if you know a priori that the signal is the correct length, you can get slightly faster operation with a direct call.

Example:

```
#include <iostream.h>
#include "Dsp.h"
main(){
  TimeSignal swave1 = DSPSinSignal(100, 5.12, 1/5.12*10.0);
  FreqSignal spec1  = DSPFFT(swave1);
  swave1.data_out("samp1");
  spec1.data_out ("samp2");
}
```

`FreqSignal` **DSPDFT** `(TimeSignal& t)` Function

The Discrete Fourier Transform is the slow way to take a FT. However, it is not limited to performing only on signals of a length 2**n. Normally, you don't have to use DSPDFT directly. If you just call DSPFT directly, that program will dispatch to the appropriate fourier transform. However, if you know a priori that the signal is the correct length, you can get slightly faster operation with a direct call.

Example:

```
#include <iostream.h>
#include "Dsp.h"
main(){
  TimeSignal swave1 = DSPSinSignal(100, 5.11, 1/5.12*10.0);
  FreqSignal spec1  = DSPDFT(swave1);
  swave1.data_out("samp1");
  spec1.data_out ("samp2");
}
```

4.4 Inverse Fourier Transforms

Inverse Fourier transforms allow you to make a time representation of a frequency spectrum.

libDsp makes a distinction between FT's and IFT's. It also provides an intelligent interface to these routines. Please note, by working in powers of two you get a much faster IFT.

`TimeSignal` **DSPIFT** `(FreqSignal& f)` Function

The Inverse Fourier Transform is a simple dispatching routine that decides if a sample is a candidate for IFFT, or rather if it must avail to IDFT. It is recommended that you use this whenever possible. As the IFFT routine becomes more capable, this routine will begin to dispatch more signals to it.

When calling the DSPIFT, you must supply the sampling rate, as this information is not intrinsic in the spectrum.

Example:

```
#include <iostream.h>
#include "Dsp.h"
main(){
  TimeSignal swave1 = DSPSinSignal(100, 5.12, 1/5.12*10.0);
  FreqSignal spec1  = DSPFT(swave1);
  TimeSignal swave2 = DSPIFT(spec1, 100);
  swave1.data_out ("samp1");
  spec1.data_out  ("samp2");
  swave2.data_out ("samp3");
}
```

`TimeSignal` **DSPIFFT** `(FreqSignal& f)` Function

The Inverse Fast Fourier Transform is the fastest way to take a IFT. However, it is limited to performing only on signals of a length 2**n. When you can use it, though, it is several orders of magnitude faster than the IDFT.

Normally, you don't have to use DSPIFFT directly. If you just call DSPIFT directly, that program will dispatch to the appropriate inverse fourier transform. However, if you know a priori that the signal is the correct length, you can get faster operation with a direct call.

Example:

```
#include <iostream.h>
#include "Dsp.h"
main(){
  TimeSignal swave1 = DSPSinSignal(100, 5.12, 1/5.12*10.0);
  FreqSignal spec1  = DSPFFT(swave1);
  TimeSignal swave2 = DSPIFFT(spec1);
  swave1.data_out("samp1");
  spec1.data_out ("samp2");
  swave2.data_out("samp3");
}
```

`TimeSignal` **DSPIDFT** `(FreqSignal& f)` Function

The Inverse Discrete Fourier Transform is the slow way to take a IFT. However, it is not limited to performing only on signals of a length 2**n. Normally, you don't have to use DSPIDFT directly. If you just call DSPIFT directly, that program will dispatch to the appropriate inverse fourier transform. However, if you know a priori that the signal is the correct length, you can get faster operation with a direct call.

Example:

```
#include <iostream.h>
#include "Dsp.h"
main(){
  TimeSignal swave1 = DSPSinSignal(100, 5.11, 1/5.12*10.0);
  FreqSignal spec1  = DSPDFT(swave1);
  TimeSignal swave2 = DSPIDFT(spec1);
  swave1.data_out("samp1");
  spec1.data_out ("samp2");
  swave2.data_out("samp3");
}
```

5 Gnu Plot

The standard output device used to display data is the very excellent GnuPlot program. To use these functions, you must have gnuplot on your system and in your execution path. If this is the case, all you have to do is open a gnuplot window (by using GPopen), then pass that valid window to the object you want to plot. When you are done, remember to use GPclose to terminate the gnuplot process.

Additionally, you have the option of opening a gnuplot window that prints to a PostScript file instead of the screen.

An example of these plotting routines:

```
#include <Dsp.h>
#include <String.h>

int MyPlotTimeSignal(TimeSignal& t){
     static GPlot* g1 = GPopen();
     static GPlot* g2 = GPfopen("dump.ps");
     String buffer;

     t.gplot(g1);
     cout << "Save this plot to the dump file?" << flush;
     cin >> buffer;
     if (buffer[0] == 'y' || buffer[0] == 'Y')
       t.gplot(g2);
     return 1;
}
```

GPlot File Reference Pointer

A `GPlot` is used just like `FILE`, ie. normally you use a GPlot* to point to a gnuplot window.

`GPlot*` GPopen () Function

Returns a new, open GnuPlot X11 window which can now be plotted in (technically, the window doesn't open until plotting is actually done in it. What is happening is that a pipe is opened to a gnuplot process. Gnuplot doesn't open a window until the first time you plot in it.)

`GPlot*` **GPfopen** (`char*` *filename*) Function

Returns a new, open GnuPlot window which will plot PostScript to the file names in *filename*. Otherwise, this is identical to `GPopen`.

`GPlot*` **GPclose** (`GPlot*` *gp*) Function

Closes (and flushes) the plotting window opened by either `GPopen` or `GPfopen`.

`char*` **GPUniqueName** () Function

Returns a pointer to a string containing a pseudo unique file name for holding plotting information. Used by internal GnuPlot functions to pass data. Note, the return value is a pointer to a static piece of memory which is overwritten every call (i.e. `strcpy` the name if you ever want to use it again).

`void` **pause** () Function

The pause command is provided to allow the program to, you guessed it, pause between graphs. While it was developed for this purpose, it can pause the program at any time it is convient. Pause is held until the return key is pressed, but no message to this effect is printed by the pause program.

```
#include <iostream.h>
#include "Dsp.h"

main(){
  GPlot* g = GPopen();

  // Plot the time signal
  TimeSignal a = DSPSinSignal(100.0, 2.56, 10.0);
  a.gplot(g, "Sample Data");
  cout << "Press <enter> to continue\n" << fflush ;
  pause();

  // Plot its frequency representation
  FreqSignal b = DSPFT(a);
  b.gplot(g, "Fourier Transform of Sample Data");
  cout << "Press <enter> to continue\n" << fflush ;
  pause();

  GPclose(g);
}
```

6 CommandLine

While writing the examples code, I found that an awful lot of one's coding time was spent parsing the command line. This, inevitably was also where all the bugs showed up, because it's quickly hacked together.

`CommandLine` is an object that handles the parsing one might want to do on a command line.

Some quick nomenclature notes. A switch is something on a command line proceeded by a single dash. A parameter is anything on the command line not proceeded by a dash. All switches must appear before parameters (in fact, after the first parameter is scanned, everything else on the command line is treated as a parameter.)

CommandLine Utility Class

`CommandLine` is a class for parsing and asking questions about a command line.

6.1 CommandLine Functions

`void` **CommandLine::help_set** (`String` *help-message*) Function

`help_set` sets the internal help message of CommandLine to be the string in *help-message*. This string must be formatted as follows: it is being printed by `fprintf` so all standard formatting conventions apply. Secondly, there **MUST** be one '`%s`' in the string. This will be replaced by the name of the program when help is printed.

Example:

```
char* help_msg = "USAGE: %s <infile> <outfile>\n";
```

`int` **CommandLine::parse** (`int` *argc*, `char**` *argv*) Function

`parse` is the parsing engine called by the base initializer. It takes the *argc* and *argv* that are passed to main and figures out what switches and parameters have been set.

`int` **CommandLine::switch_set** (`String` *s*) Function

`switch_set` is a query as to whether or not a particular switch has been set on the command line. Returns 1 if it has and 0 if it hasn't.

`long` **CommandLine::how_many_switches** () Function

`how_many_switches` returns the number of switches that were set on the command line.

`long` **CommandLine::how_many_params** () Function

`how_many_params` returns the number of parameters that were set on the command line.

`String` **CommandLine::Parameter** (`long` *which-one*) Function

`Parameter` returns the *which-one*th parameter from the command line. Note, parameter numbering starts at 0, ie. the first parameter is parameter number 0. Asking for a parameter beyond the last one on the line or one less than 0 results in a fatal error and terminates the program.

`String` **CommandLine::Switch** (`long` *which'one*) Function

`Switch` returns the *which_one*th switch from the command line. Note, switch numbering starts at 0, ie. the first switch is switch number 0. Asking for a switch beyond the last one on the line or one less than 0 results in a fatal error and terminates the program.

`char*` **CommandLine::csParameter** (`long` *which'one*) Function

`csParameter` returns the *which_one*th parameter from the command line. Note, parameter numbering starts at 0, ie. the first parameter is parameter number 0. Asking for a parameter beyond the last one on the line or one less than 0 results in a fatal error and terminates the program. The `char *` is a reference to a static buffer, which of course will change each time the function is called (so use `strcpy` to save it if you will need it again.)

`char*` **CommandLine::csSwitch** (`long` *which'one*) Function

`Switch` returns the *which_one*th switch from the command line. Note, switch numbering starts at 0, ie. the first switch is switch number 0. Asking for a switch beyond the last one on the line or one less than 0 results in a fatal error and terminates the program.The `char *` is a reference to a static buffer, which of course will change each time the function is called (so use `strcpy` to save it if you will need it again.)

`void` **CommandLine::die** () Function

`die` terminates execution of the program and returns an error status of 1 to the calling shell.

`void` **CommandLine::die_on_switch** (`String` *switch*) Function

`die_on_switch` prints the help message and terminates execution if *switch* was set on the command line. Most commonly, you will place one of these in your code with the switch '`"h"`' so that people may type '`foo -h`' to get help.

`void` **CommandLine::die_if_switch_not_one_of** (`int` *how-many*, ...) Function

`die_if_switch_not_one_of` allows you to quickly scan the switches that were set, and if any of them are not on the allowed list, to print the help message and die. *how-many* is set to the number of valid switches in the list, and then the valid switches are listed afterwards.

Example

```
CL.die_if_switch_not_one_of(4, "a", "s", "m", "d");
```

The above example will print help and halt the program if any switch beside a, s, m or d was set.

`void` **CommandLine::die_if_switch_count_not_between** (`long` *low*, `long` *high*) Function

`die_if_switch_not_between` prints the help and halts the program if the number of switches set is not between *low* and *high* inclusive.

`void` **CommandLine::die_if_param_count_not_between** (`long` *low*, `long` *high*) Function

`die_if_switch_not_between` prints the help and halts the program if the number of parameters set is not between *low* and *high* inclusive.

6.2 CommandLine Variables

The following are the variables internal to CommandLine. You can't touch them, but their descriptions are provided here for the curious.

`String` **CommandLine::command_name** Variable

This variable gets the name of the command (from *argv[0]*). It's assigned when the `parse` method runs.

String* CommandLine::switches Variable

This array holds the switches that have been set.

String* CommandLine::s_values Variable

This array holds the values switches have been set to. It is not yet used.

String* CommandLine::parameters Variable

This array holds the parameters that have been set.

String CommandLine::help_msg Variable

help_msg holds the help message that get's displayed when `CommandLine` detects an error and decides to die.

long CommandLine::s_count Variable

s_count holds the number of switches that have been set.

long CommandLine::p_count Variable

s_count holds the number of parameters that have been set.

Concept Index

Function Index

C

D

F

G

H

I

P

S

T

Variable Index

C

Data Type Index

C

F

G

T

Program Index

Table of Contents

Appendix B

Spread Spectrum Codes

B.1 BitGen.cc

This is the maximal length bit stream generator:

```
#ifndef _BITGEN_H
#define _BITGEN_H
#include "BitGen.h"

unsigned long WatsonMod2[21][4] =
{{0,       }, //11
 {1, 1,    }, //22
 {1, 1,    }, //33
 {1, 1,    }, //44
 {1, 2,    }, //55                                                    10
 {1, 1,    }, //66
 {1, 1,    }, //77
 {3, 4, 3, 2,}, //88
 {1, 4     }, //99
 {1, 3     }, //110
 {1, 2     }, //111
 {3, 6, 4, 1 }, //112
 {3, 4, 3, 1 }, //113
 {3, 5, 3, 1 }, //114
 {1, 1     }, //115                                                   20
 {3, 5, 3, 2 }, //116
 {1, 3     }, //117
 {3, 5, 2, 1 }, //118
 {3, 5, 2, 1 }, //119
 {1, 3     }, //220
} ;


BitGen::BitGen(long length, long seed){
   Length  = length;                                                  30
```

```
    Seed    = seed;
    if (seed & (1L << length)) Current = 1; else Current = 0;
    State = 1;
    Mask = 0;
    for (long i=0; i < WatsonMod2[length-1][0]; i++){
        Mask |= 1L << WatsonMod2[length-1][1 + i]-1;
    }
}

void
BitGen::time_advance()
{
    if (State & 1L << Length){
        State = ((State ^ Mask) << 1) | 1L;
        Current = 1;
    } else {
        State <<= 1;
        Current = 0;
    }

}
#endif
```

B.2 direct_sequence.cc

This is the directsequence code itself:

```
#include "Dsp.h"
#include "BitGen.h"

// Return a TimeSignal sampled at rate srate of length time, made up
// of a carrier frequency Wc, a PseudoRandom noise signal at at Time
// Rate Tc (chip), of bits bits and seed seed. The data is at a rate
// Td and is held in memory starting at data (1 bit/byte) and of
// data_bits length.

TimeSignal
direct_sequence_encode
  (double srate, double time, double Wc, double Tc, double Td, long
   bits, unsigned long seed, long data_bits, char* Data)
{
    long i;

    TimeSignal carrier = DSPCosWave(srate, time, Wc);

    TimeSignal chip(srate, time);
    BitGen bg(bits, seed);
    unsigned long chip_num = 0;
    double current_time;
    unsigned long current_chip;
    for (i=0; i<chip.size(); i++){
        current_time = (double) i / srate;
```

```
        current_chip = (unsigned long) (current_time / Tc);
        while (current_chip > chip_num){
            chip_num++;
            bg.time_advance();
        }
        if (bg.current()) chip[i] =  1.0;
        else              chip[i] = -1.0;
    }

    TimeSignal data(srate, time);
    unsigned long data_num = 0;
    unsigned long current_data;
    for (i=0; i<data.size(); i++){
        current_time = (double) i / srate;
        current_data = (unsigned long) (current_time / Td);
        while (current_data > data_num){
            data_num++;
        }
        if (Data[data_num % data_bits] == '1') data[i] =  1.0;
        else                                   data[i] = -1.0;
    }

    return (carrier * chip * data);
}

char*
direct_sequence_decode
  (TimeSignal& signal, double Wc, double Tc, double Td, long
   bits, unsigned long seed)
{
    double srate = signal.time_base();
    double time  = signal.duration();

    TimeSignal chip(srate, time);
    BitGen bg(bits, seed);
    unsigned long chip_num = 0;
    double current_time;
    unsigned long current_chip;
    for (long i=0; i<chip.size(); i++){
        current_time = (double) i / srate;
        current_chip = (unsigned long) (current_time / Tc);
        while (current_chip > chip_num){
            chip_num++;
            bg.time_advance();
        }
        if (bg.current()) chip[i] =  1.0;
        else              chip[i] = -1.0;
    }

    TimeSignal unspread = signal   / chip;
    TimeSignal unmod    = unspread / DSPCosWave(srate, time, Wc);

    unsigned long data_num = 0;
    unsigned long current_data;
```

```
    Complex sum = 0;
    char *rval = new char[ (long) (((double) unmod.size())/Td) + 1 ];
    for (i=0; i<unmod.size(); i++){
        current_time = (double) i / srate;
        current_data = (unsigned long) (current_time / Td);
        while (current_data > data_num){
            if (abs(sum) > 0)
                rval[data_num] = '1';
            else
                rval[data_num] = '0';
            sum = 0;
            data_num++;
        }
        sum += unmod[i];
    }
    return rval;
}
```

B.3 main.cc

and finally a main section that was used to test the above code:

```
#include "Dsp.h"
#include "BitGen.h"
#include <stdio.h>
#include <stdlib.h>
GPlot* gp;

TimeSignal
DTfill_time_signal
  (double SampRate, double time, long dsize, Complex* data)
{
    unsigned long length =  (unsigned long) (SampRate * time);
    Complex* rval = new Complex[length];

    for (long i=0; i<length; i++)
        rval[i] = data[i % dsize];

    return TimeSignal(SampRate, length, rval);

}

TimeSignal
DTchip_signal
  (double SampRate, double time, long bits, double Dt)
{
```

```
    unsigned long length =  (unsigned long) (SampRate * time);
    Complex* rval = new Complex[length];
    BitGen bg(bits, 1);


    double ctime;                                                    30
    long   cstep;
    long   step=0;
    for (long i=0; i<length; i++){
          ctime = ( (double) i / SampRate );
          cstep = (long) (ctime / Dt);
          while (cstep > step){
             step++;
             bg.time_advance();
          }
          if (bg.current()){                                         40
             rval[i] = (Complex) (1.0);
          } else {
             rval[i] = (Complex) (-1.0);
          }
    }

    return TimeSignal(SampRate, length, rval);

}
                                                                     50

TimeSignal
DTdata_signal
   (double SampRate, double time, long data_bits, char* data, double Dt)
{
    unsigned long length =  (unsigned long) (SampRate * time);
    Complex* rval = new Complex[length];

    double ctime;
    long   cstep;                                                    60
    long   step=0;
    for (long i=0; i<length; i++){
          ctime = ( (double) i / SampRate );
          cstep = (long) (ctime / Dt);
          while (cstep > step){
             step++;
          }
          if (data[step%data_bits] == '1'){
             rval[i] = (Complex) (1.0);
          } else {                                                   70
             rval[i] = (Complex) (-1.0);
          }
    }

    return TimeSignal(SampRate, length, rval);
}

char*
```

```
DTget_data
  (TimeSignal& ts, double Dt, double Wc)                                    80
{
        char msg[512];

   long  dbits = (long)(ts.duration() / Dt);
   char* rval =  new char[dbits + 8];

   // Calculate how big a fourier transform to use
   long bits    = (long) (log(Dt * ts.time_base()) / log(2.0));
   long samples = (long) pow(2.0, (double) bits);
        //  printf("Samples => %ld\n", samples);                            90

   // Allocate space
   TimeSignal working(ts.time_base(), samples);
   FreqSignal fworking;

   // Figure the bins to pull the signal out of
   Wc /= (ts.time_base()) / (double) samples;  // Adjust to bin space

   double bin1 = Wc;
   double bin2 = (double) samples - bin1;                                   100
   long   off1 = (long) bin1;
   double del1 = bin1 - (double) off1;
   long   off2 = (long) bin2;
   double del2 = bin2 - (double) off2;
/*   printf("Bins: %lg (%ld + %lg) and %lg (%ld + %lg)\n", bin1, off1,
           del1, bin2, off2, del2);
           */
   long i, j, offset;
   Complex sum1, sum2, sum;
   for (i=0; i<dbits; i++){                                                 110
        offset = (long)(i * Dt * ts.time_base()) ;

        for (j=0; j<samples; j++)
           working[j] = ts[offset+j];
        fworking = DSPFT(working);

#ifdef PLOT_ME
        sprintf(msg, "Bit %ld", i);
        fworking.gplot(gp, msg);
#endif                                                                      120
#ifdef HPoint
        for (j=0; j<samples; j++)
           if (real(fworking[j]) + imag(fworking[j]) > 200.0)
                printf("Hot Point --- [%ld] %ld\n", i, j);
#endif

        sum1 = (fworking[off1 + 1] - fworking[off1]) * del1 +
           fworking[off1];
        sum2 = (fworking[off2 + 1] - fworking[off2]) * del2 +
           fworking[off2];                                                  130
        if (real(sum1 + sum2) > 0)
           rval[i] = '1';
```

```
        else
            rval[i] = '0';
    }

    rval[dbits] = '\0';
    return rval;
}

TimeSignal
DT_BP_filter
  (double srate, long points, double start, double stop)
{
    long i;
    FreqSignal working(points);
    // to convert radians -> Hz
    double tfactor = 1.0 / (2.0 * PI) * srate / 2.0;
    start /= tfactor;
    stop  /= tfactor;
    for (i=0; i<working.size(); i++)
        if ((working.frequency(i) > start &&
             working.frequency(i) < stop) ||
            (working.frequency(i) > -stop &&
             working.frequency(i) < -start))
            working[i] = 1.0; else working[i] = 0.0;
    TimeSignal rval = DSPIFT(working, srate);
    return DSPWindowHanning(rval);
}

/*
main(){
    TimeSignal t = DT_BP_filter(8000, 128, 900, 1100);
    gp = GPfopen("foo.ps");
    t.gplot(gp, "Filter");
    GPclose(gp);
}
*/

char*
make_data(long bits)
{
    char* buffer;
    buffer = new char[bits + 1];
    for (long i=0; i<bits; i++){
        if ( ( (double) rand() / (double) RAND_MAX ) > .5 )
            buffer[i] = '0';
        else
            buffer[i] = '1';
    }
    buffer[bits] = '\0';
    return buffer;
}

main(int argc, char** argv){
    long i;
```

```
   srand(atoi(argv[4]));
   double nlevel = atof(argv[1]);

   gp = GPfopen("sample.ps");                                          190

   TimeSignal midata;
   midata.readAiff("sample1.aiff");
   double mymax = 0.0;
   for (long ii = 0; ii < midata.size(); ii++){
         if (abs(midata[ii]) > mymax) mymax = abs(midata[ii]);
   }
   midata /= mymax;

                                                                       200
   double srate = 8000;
   double Wc    = 1000;
   double time  = midata.duration();
   double factor = atof(argv[2]);
   double Td    = factor / 8000.0 * atof(argv[3]) ;
   double Tc    = factor / 8000.0 ;
   char *Data   = make_data(time / Td);
   long data_bits = strlen(Data);

   TimeSignal chip    = DTchip_signal(srate, time, 10, Tc);            210
   TimeSignal data    = DTdata_signal(srate, time, data_bits, Data, Td);
   TimeSignal carrier = DSPCosWave(srate, time, Wc);
   TimeSignal encode  = data * carrier;
   //eencode.gplot(gp, "Pre mess");

#ifdef PLOT_ME
   FreqSlice fs;
   fs = Slice(encode, 128);
   char msg[512];
   sprintf(msg, "spectrogram, slice 30 before spread with %s", argv[1]);  220
   fs[30].gplot(gp, msg);
#endif

   TimeSignal sample  = chip * data * carrier;

   //ssample.gplot(gp, "Post spread");

#ifdef PLOT_ME
   fs = Slice(sample, 128);
   sprintf(msg, "spectrogram, slice 30 after spread with %s", argv[1]);  230
   fs[30].gplot(gp, msg);
#endif

//     double noise;
//     for (i=0; i<sample.size(); i++){
//         noise = (double) rand() / (double) RAND_MAX;
//         noise = noise * nlevel - nlevel / 2;
//         sample[i] += noise;
//    }
                                                                       240
```

```
   sample += nlevel * midata;
   sample.writeAiff("encoded.aiff");

#ifdef PLOT_ME
   fs = Slice(sample, 128);
   sprintf(msg, "spectrogram, slice 30 noise introduced with %s", argv[1]);
   fs[30].gplot(gp, msg);
#endif

  //ssample.gplot(gp, "Post mess");                                    250

   TimeSignal unspread    = sample * chip;

#ifdef PLOT_ME
   fs = Slice(unspread, 128);
   sprintf(msg, "spectrogram, slice 30 unspread with %s", argv[1]);
   fs[30].gplot(gp, msg);
   //uunspread.gplot(gp, "Post spread");
#endif
                                                                      260
#ifdef FILTER_ME
   double fstart = 900;
   double fstop  = 1100;
   TimeSignal filt = DT_BP_filter(8000, 512, fstart, fstop);
   TimeSignal filtered = ApplyFIR(unspread, filt);
#ifdef PLOT_ME
   fs = Slice(unspread, 128);
   sprintf(msg, "spectrogram, slice 30 uns and filt with %s filt %lg"
            " to %lg", argv[1], fstart, fstop);
   fs[30].gplot(gp, msg);                                             270
#endif
#endif

   char* decoded     = DTget_data(unspread, Td, Wc);

   double right = 0.0;
   double wrong = 0.0;
   for (i=0; i<strlen(decoded); i++){
         if (decoded[i] == Data[i]) right += 1.0; else wrong += 1.0;
   }                                                                  280
   printf("NSR <%s> Bits/Chip <%s> N <%s> Bits <%d> Correct<%lg%%> \n",
          argv[1], argv[2], argv[3], strlen(decoded), right / (right + wrong));

}
```

Appendix C

libDsp source code

C.1 Files

Following is a list of the files in the `libDsp` distribution:

```
libDsp
libDsp/INSTALL
libDsp/NEWS
libDsp/README
libDsp/TODO
libDsp/install.sh
libDsp/configure.in
libDsp/configure
libDsp/Makefile.in
libDsp/doc
libDsp/doc/Makefile
libDsp/doc/examples
libDsp/doc/examples/echo.cc
libDsp/doc/examples/AIFFdisplay.cc
libDsp/doc/examples/Makefile
libDsp/doc/examples/AIFFmath.cc
libDsp/doc/examples/MakeAIFF.cc
libDsp/doc/examples/TimeStretch.cc
libDsp/doc/examples/intro.cc
libDsp/doc/examples/vdistort.cc
libDsp/doc/examples/vdistort
libDsp/doc/examples/.cvsignore
libDsp/doc/examples/intro
libDsp/doc/examples/echo
libDsp/doc/examples/TimeStretch
libDsp/doc/examples/Makefile.in
libDsp/doc/echo.cc.texi
libDsp/doc/Makefile.in
```

```
libDsp/doc/libDsp.texi
libDsp/libsrc/Makefile.in
libDsp/libsrc
libDsp/libsrc/aifif
libDsp/libsrc/aifif/edif.cc
libDsp/libsrc/aifif/genutils.cc
libDsp/libsrc/aifif/macsndf.cc
libDsp/libsrc/aifif/xxmpeg.cc
libDsp/libsrc/aifif/aifif.c
libDsp/libsrc/aifif/Makefile
libDsp/libsrc/aifif/aifif.cc
libDsp/libsrc/aifif/byteswap.cc
libDsp/libsrc/aifif/edif.c
libDsp/libsrc/aifif/IEEE80.cc
libDsp/libsrc/aifif/IEEE80.c
libDsp/libsrc/aifif/byteswap.c
libDsp/libsrc/aifif/genutils.c
libDsp/libsrc/aifif/macsndf.c
libDsp/libsrc/Makefile.in
libDsp/libsrc/objects
libDsp/libsrc/objects/Makefile.in
libDsp/libsrc/objects/CommandLine.cc
libDsp/libsrc/objects/FFT.cc
libDsp/libsrc/objects/FreqSlice.cc
libDsp/libsrc/objects/GPlot.cc
libDsp/libsrc/objects/SigGen.cc
libDsp/libsrc/objects/Slice.cc
libDsp/libsrc/objects/UniqueName.cc
libDsp/libsrc/objects/FreqSignal.cc
libDsp/libsrc/objects/TimeSignal.cc
libDsp/libsrc/objects/DSPtools.cc
libDsp/libsrc/objects/Filters.cc
libDsp/libsrc/objects/CArray.cc
libDsp/include
libDsp/include/CommandLine.h
libDsp/include/Consts.h
libDsp/include/DSPtools.h
libDsp/include/Dsp.h
libDsp/include/FFT.h
libDsp/include/Filters.h
libDsp/include/FreqSignal.h
libDsp/include/FreqSlice.h
libDsp/include/GPlot.h
libDsp/include/IEEE80.h
libDsp/include/SigGen.h
```

```
libDsp/include/Slice.h
libDsp/include/TimeSignal.h
libDsp/include/UniqueName.h
libDsp/include/aifif.h
libDsp/include/byteswap.h
libDsp/include/dpwe-aiff.h
libDsp/include/dpwelib.h
libDsp/include/edif.h
libDsp/include/genutils.h
libDsp/include/vecops.h
libDsp/include/CArray.h
libDsp/include/.cvsignore
libDsp/include/Stereo.h
libDsp/include/Makefile.in
libDsp/examples
libDsp/examples/echo
libDsp/examples/echo/Makefile
libDsp/examples/echo/main.cc
libDsp/examples/echo/Makefile.in
libDsp/examples/MakeAIFF
libDsp/examples/MakeAIFF/Makefile.in
libDsp/examples/MakeAIFF/main.cc
libDsp/examples/MakeAIFF/Makefile
libDsp/examples/Makefile.in
libDsp/examples/AIFFdisplay
libDsp/examples/AIFFdisplay/main.cc
libDsp/examples/AIFFdisplay/Makefile.in
libDsp/examples/intro
libDsp/examples/intro/Makefile.in
libDsp/examples/intro/main.cc
libDsp/examples/AIFFmath
libDsp/examples/AIFFmath/Makefile.in
libDsp/examples/AIFFmath/main.cc
libDsp/examples/NOTES
libDsp/examples/vdistort
libDsp/examples/vdistort/main.cc
libDsp/examples/vdistort/Makefile.in
```

C.2 libDsp/INSTALL

To Create the library, just type make in the main directory. The .a file will be made in the libsrc directory. To compile a program use

```
g++ -o foo -I$(BASE)/include foo.cc -L$(BASE)/libsrc -lDsp
```

```
where $(BASE) is whereever the main directory tree is.

type make doc
```

C.3 libDsp/NEWS

```
Version 2.0.0 ---
* Added install

* Cleanup of manual in general

* Major restructuring of files witin release

* Begining of port to GNU autoconf to allow libDsp to
        compile out of the box on a number of platforms

        * Removed all examples from manual and placed them in seperate
        files so that they will be tested before each release

        * Brought most makefiles up to GNU spec

Version 1.3.5 ---
* Lots of bug fixes, including virtual in base class
destructors, and passes by argument rather than by value
        in operators.


Version 1.3.0 ---
>>>>>>> * CommandLine object added -- This is a really neat hack I
wish I had come up with a lot earlier. It is an auto command
line parser that takes much of the pain out of providing a
usable user interface.

       * Major cleanup of documentation

        * Aliased SinSignal --> SinWave

* Aliased CosSignal --> CosWave

* Added Examples section, including
          ** intro -- how it works
  ** AIFFmath -- simple math on AIFF files
          ** TimeStretch -- Increases or decreases signal duration
```

```
  w/o changing frequency of data.

* Added delay to time signals.

* Cleaned up plotting of Freq Slice

* Cleaned up bug in scaling DSPIDFT


Version 1.2.1 ---
* Fixed bug in write AIFF

Version 1.2.0 ---
* Added Freq Slice data type -- This data type is what one
uses to hold the "break the signal up into frames, FFT the
        frames and display them" way of looking at a signal. Right
        now I don't have a routine to go from Time Signal directly
        to FreqSlice, because I find I usually want something like
        every 10th slice displayed to increase plotting speed. If
        anyone has a good idea for geberalizing this, let me know.

* Added Write AIFF

Version 1.1.0 ---
* Added AIFF reading

        * Added Square Wave and Triangle Wave generators.

* Added full and half wave rectifiers

* Added GPfopen: This allows you to open a gplot file for
  ploting to postscript rather than the screen.

Version 1.0.0 --- Base Release
```

C.4 libDsp/README

```
====================

README -- This file

NEWS -- What's new in this release
```

```
./include - all the include files for libDsp

./libsrc  - the C++ source files for libDsp

./doc - the texi manual for libDsp

./examples - The example code you should look at

./testsrc - These are test files I have used with libDsp's
        development. They are included for now until I can get around
        to writing a decent tutorial to give the enthusiastic some
        examples of how libDsp works.
```

C.5 libDsp/TODO

```
* Add SOX to read/write any (reasonable) format.

* Change GNUplot to pass by pipe rather than file
```

C.6 libDsp/install.sh

This is provided incase the target system is missing an install program.

```
#!/bin/sh

#
# install - install a program, script, or datafile
# This comes from X11R5; it is not part of GNU.
#
# $XConsortium: install.sh,v 1.2 89/12/18 14:47:22 jim Exp $
#
# This script is compatible with the BSD install script,
# but was written from scratch.
#


# set DOITPROG to echo to test this script

# Don't use :- since 4.3BSD and earlier shells don't like it.
doit="${DOITPROG-}"


# put in absolute paths if you don't have them in your path;
```

```
# or use env. vars.

mvprog="${MVPROG-mv}"
cpprog="${CPPROG-cp}"
chmodprog="${CHMODPROG-chmod}"
chownprog="${CHOWNPROG-chown}"
chgrpprog="${CHGRPPROG-chgrp}"
stripprog="${STRIPPROG-strip}"
rmprog="${RMPROG-rm}"

instcmd="$mvprog"
chmodcmd=""
chowncmd=""
chgrpcmd=""
stripcmd=""
rmcmd="$rmprog -f"
mvcmd="$mvprog"
src=""
dst=""

while [ x"$1" != x ]; do
    case $1 in
-c) instcmd="$cpprog"
    shift
    continue;;

-m) chmodcmd="$chmodprog $2"
    shift
    shift
    continue;;

-o) chowncmd="$chownprog $2"
    shift
    shift
    continue;;

-g) chgrpcmd="$chgrpprog $2"
    shift
    shift
    continue;;

-s) stripcmd="$stripprog"
    shift
    continue;;
```

```
*)  if [ x"$src" = x ]
    then
src=$1
    else
dst=$1
    fi
    shift
    continue;;
    esac
done

if [ x"$src" = x ]
then
echo "install:  no input file specified"
exit 1
fi

if [ x"$dst" = x ]
then
echo "install:  no destination specified"
exit 1
fi


# If destination is a directory, append the input filename; if
# your system does not like double slashes in filenames, you may
# need to add some logic

if [ -d $dst ]
then
dst="$dst"/`basename $src`
fi

# Make a temp file name in the proper directory.

dstdir=`dirname $dst`
dsttmp=$dstdir/#inst.$$#

# Move or copy the file name to the temp name

$doit $instcmd $src $dsttmp

# and set any options; do chmod last to preserve setuid bits

if [ x"$chowncmd" != x ]; then $doit $chowncmd $dsttmp; fi
```

```
if [ x"$chgrpcmd" != x ]; then $doit $chgrpcmd $dsttmp; fi
if [ x"$stripcmd" != x ]; then $doit $stripcmd $dsttmp; fi
if [ x"$chmodcmd" != x ]; then $doit $chmodcmd $dsttmp; fi

# Now rename the file to the real destination.

$doit $rmcmd $dst
$doit $mvcmd $dsttmp $dst


exit 0
```

C.7 libDsp/configure.in

```
dnl Process this file with autoconf to produce a configure script.
AC_INIT(examples)
AC_PROG_CXX
AC_LN_S
AC_PROG_INSTALL
AC_OUTPUT(
 Makefile
 doc/Makefile
 doc/examples/Makefile
   examples/Makefile
                examples/AIFFdisplay/Makefile
 examples/AIFFmath/Makefile
                examples/MakeAIFF/Makefile
                examples/TimeStretch/Makefile
                examples/echo/Makefile
                examples/intro/Makefile
                examples/vdistort/Makefile
 libsrc/Makefile
 libsrc/aifif/Makefile
 libsrc/objects/Makefile
 include/Makefile
        )
```

C.8 libDsp/configure

This file is created by the Autoconfigure utility.

C.9 libDsp/Makefile.in

```
srcdir = @srcdir@
VPATH  = @srcdir@

CXX = @CXX@

INSTALL = @INSTALL@
INSTALL_PROGRAM = @INSTALL_PROGRAM@
INSTALL_DATA = @INSTALL_DATA@

DEFS = @DEFS@
LIBS = @LIBS@

CFLAGS = -g
LDFLAGS = -g

prefix = /usr/local
exec_prefix = $(prefix)
binprefix =
manprefix =
incprefix =

bindir = $(exec_prefix)/bin
libdir = $(exec_prefix)/lib
mandir = $(prefix)/man/man1
manext = 1
infodir = /usr/local/info

SHELL = /bin/sh

LIBDIRS = include libsrc
SRCDIRS = doc examples $(LIBDIRS)

DISTFILES = INSTALL NEWS configure.in README configure\
            Makefile.in $(SRCDIRS)

all: library

install:
echo foo
for var in $(LIBDIRS); do cd $$var; $(MAKE) install; cd ..; done

installdirs:
```

```
uninstall:

check:
@echo No tests are supplied at this time.

library:
for var in $(LIBDIRS); do\
     cd $$var;\
     $(MAKE) all;\
     cd ..; done

Makefile: Makefile.in config.status
$(SHELL) config.status
config.status: configure
$(SHELL) config.status --recheck
configure: configure.in
cd $(srcdir); autoconf

TAGS:

clean:
for var in $(SRCDIRS); do\
     cd $$var;\
     $(MAKE) clean;\
     cd ..; done

mostlyclean: clean

distclean: clean
rm -f Makefile config.status

realclean: distclean

dist:
```

C.10 libDsp/doc/

The contents of this subdirectory create the manual found in Appendix A.

C.11 libDsp/libsrc/Makefile.in

```
INSTALL = @INSTALL@
INSTALL_PROGRAM = @INSTALL_PROGRAM@
INSTALL_DATA = @INSTALL_DATA@

C++ = @CXX@
C++FLAGS = -g
LDFLAGS = -g

DEFS = @DEFS@

LIBOBJS = objects/*.o aifif/*.o

dirs=objects aifif

prefix = /usr/local
incdir = ../include
srcdir = @srcdir@
libdir = $(prefix)/lib/${SYS}

VPATH=$(dirs)

lib=libDsp.a

INCS=-I${incdir} -I/usr/include/local

# Suffix Rules the way I want them
.c.o:
${C++} ${C++FLAGS} ${INCS} -c $< -o $@
.cc.o:
${C++} ${C++FLAGS} ${INCS} -c $< -o $@

INSTALL=cp

all:$(lib)

install:$(lib)
for var in $(dirs); do cd $$var; $(MAKE) HEAD=$(head)/.. install\
            ; cd ..; done
$(INSTALL) $(lib) $(libdir)

OBJS=aifif/*.o objects/*.o

$(lib):objs
```

```
ar ruv $@ $(OBJS)
ranlib $@

clean:
rm -f $(lib)
for var in $(dirs); do cd $$var; $(MAKE) clean ; cd ..; done

objs:
for var in $(dirs); do cd $$var; $(MAKE) DEFS=$(DEFS) all ;\
        cd ..; done


# Fun with dependencies
CArray.o     : CArray.cc     $(incdir)/Dsp.h
DSPtools.o   : DSPtools.cc   $(incdir)/Dsp.h
FFT.o        : FFT.cc        $(incdir)/Dsp.h
FreqSignal.o : FreqSignal.cc $(incdir)/Dsp.h
FreqSlice.o  : FreqSlice.cc  $(incdir)/Dsp.h
SigGen.o     : SigGen.cc     $(incdir)/Dsp.h
Slice.o      : Slice.cc      $(incdir)/Dsp.h
TimeSignal.o : TimeSignal.cc $(incdir)/Dsp.h
GPlot.o      : GPlot.cc      $(incdir)/GPlot.h
UniqueName.o : UniqueName.cc $(incdir)/UniqueName.h
```

C.12 libDsp/libsrc/aifif/

These routines are courtisy of Dan Elis and not specifically part of this thesis.

C.13 libDsp/libsrc/objects/Makefile.in

```
CXX=@CXX@

C++FLAGS = -g

head=../../..
```

```
incdir=../../include
srcdir=.
libdir=$(head)/lib/${SYS}

lib=libDsp.a

INCS=-I${incdir} -I/usr/include/local

SRCS=$(wildcard *.c) $(wildcard *.cc)
OBJS=$(patsubst %.c,%.o,$(wildcard *.c))\
     $(patsubst %.cc,%.o,$(wildcard *.cc))

# Suffix Rules the way I want them
.c.o:
${CXX} ${C++FLAGS} ${INCS} $(DEFS) -c $< -o $@
.cc.o:
${CXX} ${C++FLAGS} ${INCS} $(DEFS) -c $< -o $@

all:$(OBJS)

install:$(OBJS)

clean:
rm -f ${OBJS}



# Fun with dependencies
CArray.o      : CArray.cc     $(incdir)/Dsp.h
DSPtools.o    : DSPtools.cc   $(incdir)/Dsp.h
FFT.o         : FFT.cc        $(incdir)/Dsp.h
FreqSignal.o  : FreqSignal.cc $(incdir)/Dsp.h
FreqSlice.o   : FreqSlice.cc  $(incdir)/Dsp.h
SigGen.o      : SigGen.cc     $(incdir)/Dsp.h
Slice.o       : Slice.cc      $(incdir)/Dsp.h
TimeSignal.o  : TimeSignal.cc $(incdir)/Dsp.h
GPlot.o       : GPlot.cc      $(incdir)/GPlot.h
UniqueName.o  : UniqueName.cc $(incdir)/UniqueName.h
```

C.14 libDsp/libsrc/objects/CommandLine.cc

```
#include <stdarg.h>
#include <stdio.h>
#include <iostream.h>
#include "CommandLine.h"
#include <stdlib.h>

void
CommandLine::die_if_param_count_not_between(long low, long high){
    if (how_many_params() < low || how_many_params() > high){
        issue_help();
        die();
    }
    return;
}


void
CommandLine::die_if_switch_not_one_of(int how_many,...){
    va_list ap;
    int i, j;

    if (how_many < 1) {
        cerr << "CommandLine::die_if_flag_not_one_of - incorrect how_many\n";
        exit(1);
    }
    String choices[how_many];
    va_start(ap, how_many);

    for (i=0; i<how_many; i++)
        choices[i] = (char *) va_arg(ap, char*);

    va_end(ap);

    int flag = 0;
    for (i=0; i<how_many_switches(); i++){
        for (j=0; j<how_many; j++)
            if (switches[i] == choices[j]){
                flag = 1;
                break;
            }
        if (!(flag)){
            issue_help();
            die();
        }
    }
    return;
}

void
CommandLine::die_if_switch_count_not_between(long low, long high){
    if (how_many_switches() < low || how_many_switches() > high){
```

```
            issue_help();
            die();
    }
    return;
}

void
CommandLine::die_on_switch(String flag){
    if (switch_set(flag)){
            issue_help();
            die();
    }
}

void
CommandLine::die(){
    exit(1);
}

void
CommandLine::issue_help(){
    fprintf(stderr, (char *) help_msg, (char *) command_name);
    return;
}

CommandLine::CommandLine(int argc, char** argv){
    parse(argc, argv);
    return;
}

CommandLine::~CommandLine(){
    delete [] switches;
    delete [] s_values;
    delete [] parameters;
    return;
}

int
CommandLine::parse(int argc, char** argv){
    command_name = argv[0];
    switches = new String[argc];
    s_values = new String[argc];
    parameters = new String[argc];
    s_count = 0;
    p_count = 0;

    long i=1;
    long still_switches_p = 1;

    while (i < argc) {

        // Snag a switch
        if (still_switches_p && argv[i][0] == '-') {
            switches[s_count++] = argv[i] + 1;
```

```
        } else {
          still_switches_p = 0;
          parameters[p_count++] = argv[i];
        }

        i++;
    }

    return argc;
}


// Was the switch set?
int
CommandLine::switch_set(String swtch){

    // Deal with trivial case
    if (s_count == 0) return 0;

    // Scan switches
    for (long i=0; i<s_count; i++) {
        if (switches[i] == swtch)
            return 1;
    }

    return 0;
}

// return parameter
String
CommandLine::Parameter(long ww){
    if (ww >= p_count || ww < 0) {
        cerr << "CommandLine::parameter - requested out of range"
            << "parameter.  bye.\n";
        exit(1);
    }

    return parameters[ww];
}

// return parameter
String
CommandLine::Switch(long ww){
    if (ww >= s_count || ww < 0) {
        cerr << "CommandLine::switch - requested out of range"
            << "switch.  bye.\n";
        exit(1);
    }

    return switches[ww];
}

// return parameter
char*
```

```
CommandLine::csParameter(long ww){
   static char buffer[256];
   if (ww >= p_count || ww < 0) {
        cerr << "CommandLine::parameter - requested out of range"
           << "parameter.  bye.\n";
        exit(1);
   }

   strcpy(buffer, parameters[ww]);
   return buffer;
}

// return parameter
char*
CommandLine::csSwitch(long ww){
   static char buffer[256];
   if (ww >= s_count || ww < 0) {
        cerr << "CommandLine::switch - requested out of range"
           << "switch.  bye.\n";
        exit(1);
   }

   strcpy(buffer, switches[ww]);
   return buffer;
}
```

C.15 libDsp/libsrc/objects/FFT.cc

```
/* FILE: /User/druid/src/c++/libDsp/libsrc/FFT.cc */
/* AUTHOR: Daniel F. Gruhl <druid@mit.edu> */
/* DATE LAST MODIFIED: Tue Feb  1 11:57:57 EST 1994 */
/* DESCRIPTION: This file provides the fourier transform and inverse */
/* transform functions for the libDsp library */


static char rcsid[] = "$Id:  FFT.cc,v 1.2 1994/08/01 12:27:06 druid Exp $";

#include "Dsp.h"
#include <math.h>
#include "FreqSignal.h"

long
DSPFlipAddress(int bits, long number){
  long rval=0, mul = (long) pow( 2.0, (double) bits - 1);
  for (int i=0; i<bits; i++){
```

```
    rval += mul * (number % 2);
    number /= 2;
    mul /= 2;
  }
  return rval;
}

FreqSignal
DSPFT(TimeSignal& t){
  double testval = t.size();
  double tval1 = log(testval) / log(2.0);
  if ( tval1 == (long) tval1)
    return DSPFFT(t);
  else
    return DSPDFT(t);
}

TimeSignal
DSPIFT(FreqSignal& f, double base){
//      f.data_out("dbg1.dat");
    double testval = (double) f.size();
    double tval1 = log(testval) / log(2.0);
    if ( tval1 == (long) tval1)
          return DSPIFFT(f, base);
    else
          return DSPIDFT(f, base);
}

// Adapted from FFT in Numerical Recipies in C
FreqSignal
DSPFFT(TimeSignal& t){
  unsigned long i;
  float*   wdata = new float[t.size() * 2 + 1];
  Complex* rval  = new Complex[t.size()];
  const Complex Ci(0,1);

  for (i=0; i<t.size(); i++){
    wdata[i*2 + 1] = real(t[i]);
    wdata[i*2 + 2] = imag(t[i]);
  }

  four1(wdata, t.size(), 1);
  for (i=0; i<t.size(); i++)
    rval[i] = wdata[i*2 + 1] + Ci*wdata[i*2 + 2];

  delete [] wdata;
  return FreqSignal((long) t.size(), rval, t.time_base()/t.size() );
}

TimeSignal
DSPIFFT(FreqSignal& f, double base){
  unsigned long i;
  float*   wdata = new float[f.size() * 2 + 1];
  Complex* rval  = new Complex[f.size()];
```

```
  const Complex Ci(0,1);

  for (i=0; i<f.size(); i++){
    wdata[i*2 + 1] = real(f[i]);
    wdata[i*2 + 2] = imag(f[i]);
  }

  four1(wdata, f.size(), -1);

  for (i=0; i<f.size(); i++)
    rval[i] = (wdata[i*2 + 1] + Ci*wdata[i*2 + 2]) / f.size();

  delete [] wdata;
  return TimeSignal(base, (long) f.size(), rval);
}


#define SWAP(a,b) tempr=(a); (a)=(b); (b)=tempr
void four1(float data[], unsigned long nn, int isign)
{
  unsigned long n, mmax, m, j, istep, i;
  double wtemp, wr, wpr, wpi, wi, theta;
  float tempr, tempi;

  n=nn << 1;
  j=1;
  for (i=1; i<n; i+=2){
    if (j > i) {
      SWAP(data[j], data[i]);
      SWAP(data[j+1], data[i+1]);
    }
    m=n >> 1;
    while (m >= 2 && j > m) {
      j -= m;
      m >>= 1;
    }
    j += m;
  }

  mmax=2;
  while ( n > mmax){
    istep=mmax << 1;
    theta = isign*(6.28318530717959/mmax);
    wtemp=sin(0.5*theta);
    wpr = -2.0*wtemp*wtemp;
    wpi=sin(theta);
    wr=1.0;
    wi=0.0;
    for (m=1; m<mmax; m+=2){
      for (i=m; i<=n; i+=istep){
          j=i+mmax;
          tempr=wr*data[j] - wi*data[j+1];
          tempi=wr*data[j+1] + wi*data[j];
          data[j] = data[i] - tempr;
```

```
            data[j+1] = data[i+1] - tempi;
            data[i] += tempr;
            data[i+1] += tempi;
        }
        wr=(wtemp=wr)*wpr-wi*wpi+wr;
        wi=wi*wpr+wtemp*wpi+wi;
      }
      mmax=istep;
    }
}


const Complex DSPi(0, 1);

inline Complex
DSPTW(long super, long sub){
        return pow(exp(-DSPi * (2.0 * PI / (double) sub)), (double) super);
}

FreqSignal
DSPDFT(TimeSignal& t){
        Complex* f = new Complex[t.size()];

        for (long k = 0; k<t.size(); k++){
                f[k] = 0.0;
                for (long n = 0; n<t.size(); n++){
                        f[k] += t[n] * DSPTW((long)(n*k), (long)t.size());
                }
        }
        return FreqSignal(t.size(), f, t.time_base()/t.size()  );
}

TimeSignal
DSPIDFT(FreqSignal& f, double base){
        Complex* t = new Complex[f.size()];

        for (long k = 0; k<f.size(); k++){
                t[k] = 0.0;
                for (long n = 0; n<f.size(); n++){
                        t[k] -= f[n] * DSPTW(-(long)(n*k), (long)f.size());
                }
        }
        for (k =0; k<f.size(); k++) t[k] /= -f.size();
        return TimeSignal(base, f.size(), t);
}
```

C.16 libDsp/libsrc/objects/FreqSlice.cc

```
#include <iostream.h>
```

```
#include <stdlib.h>
#include "FreqSignal.h"
#include "FreqSlice.h"
#include <string.h>

double
maxabs(FreqSlice& f){
   double d = 0.0;
   for (long i=0; i<f.Windows(); i++)                                    10
        if ( Maxabs(f[i]) > d) d = Maxabs(f[i]);
   return d;
}

FreqSlice::FreqSlice(long Windows, FreqSignal* Data, double Deltatime){
   windows = Windows;
   data = Data;
   deltatime = Deltatime;
}
                                                                         20
void
FreqSlice::gplot(GPlot* gp, char* plot_title, long every=1){
   char datafile[256];
   FILE *dfile;

   strcpy(datafile, GPUniqueName());

   dfile  = fopen(datafile, "w");
   if (!dfile){
        cerr << "FreqSlice::gplot - Cannot open plotting file.  bye...\n";   30
        exit(1);
   }

   long i,j, k = data[0].size()/2;
   for (i=0; i<windows; i+=every)
        if ((i/every)%2){
           for (j=0; j<k; j++)
                fprintf(dfile, "%lg %lg %lg\n",
                        time_of_window(i), operator[](0).frequency(j),
                        abs(data[i][j]));                                40
        } else {
           for (j=k-1; j>=0; j--)
                fprintf(dfile, "%lg %lg %lg\n",
                        time_of_window(i), operator[](0).frequency(j),
                        abs(data[i][j]));
        }
   fclose(dfile);

   fprintf( (FILE*) gp, "set title '%s'\n", plot_title);
   fprintf( (FILE*) gp, "set xlabel '%s'\nset ylabel '%s'\nset zlabel '%s'\n",   50
             "Time", "Frequency", "Magnitude");
   fprintf( (FILE*) gp, "set parametric\n");
   fprintf((FILE*) gp, "splot '%s' title 'data' with lines\n",(char*)datafile);
   fflush( (FILE*) gp);
}
```

```
FreqSlice::FreqSlice(long Windows=1, long samples_per_window=1){
  windows = Windows;
  data = new FreqSignal[windows];
  if (data == NULL) {
    cerr << "FreqSignal::FreqSignal - Cannot allocate storage space.  Bye...\n";
    exit(1);
  }
  for (long i=0; i<Windows; i++)
    data[i].resize(samples_per_window);
}

FreqSlice::~FreqSlice(){
    if (data != NULL){
          delete [] data;
          data = NULL;
    }
}

FreqSignal&
FreqSlice::operator[](long index){
  return data[index];
}

FreqSlice&
FreqSlice::operator= (FreqSlice& f){
   if (data != NULL){
          delete [] data;
          data = NULL;
    }
   windows   = f.Windows();
   deltatime = f.DeltaTime();
   data      = new FreqSignal[windows];
   for (long i=0; i<windows; i++)
      data[i] = f[i];
}
```

C.17 libDsp/libsrc/objects/GPlot.cc

```
/* FILE: /User/druid/src/c++/libDsp/libsrc/GPlot.cc */
/* AUTHOR: Daniel F. Gruhl <druid@mit.edu> */
/* DATE LAST MODIFIED: Tue Feb  8 09:51:55 EST 1994 */
/* DESCRIPTION:  */

char static rcsid[] = "$Id:  GPlot.cc,v 1.1 1994/06/27 15:02:34 druid Exp $";

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```

```
#include "Dsp.h"
#include "UniqueName.h"

/* typedef FILE GPlot; */

GPlot*
GPopen(){
  FILE* hold;
  hold = popen("gnuplot", "w");
  if (hold == NULL) {
    cerr << "GPopen:  ERROR::  Could not open pipe to gnuplot\n";
    exit(1);
  }
  fprintf(hold, "set terminal X11\n");
  fflush(hold);
  return (GPlot*) hold;
}

GPlot*
GPfopen(char* filename){
  FILE* hold;
  hold = popen("gnuplot", "w");
  if (hold == NULL) {
    cerr << "GPopen:  ERROR::  Could not open pipe to gnuplot\n";
    exit(1);
  }
  fprintf(hold, "set terminal postscript\n");
  fprintf(hold, "set output '%s'\n", filename);
  fflush(hold);
  return (GPlot*) hold;
}

GPlot*
GPclose(GPlot* gp){
  fprintf((FILE*) gp, "exit\n");
  fflush( (FILE*) gp);
  pclose( (FILE*) gp);
}

int
GPPlotFile(GPlot* gp, char* filename, char* title, char* xaxis,
           char* yaxis, char* data, char* plottype){
  fprintf( (FILE*) gp, "set title '%s'\n", title);
  fprintf( (FILE*) gp, "set xlabel '%s'\nset ylabel '%s'\n",
           xaxis, yaxis);
  fprintf( (FILE*) gp, "plot '%s' title '%s' with %s\n",
           filename, data, plottype);
  fflush( (FILE*) gp);
}

char*
GPUniqueName(){
//     static UniqueName un;
   static long num = 0;
```

```
    num++;
    static char buffer[128];
//      char* tmp = un.tmp_file_name("GPlot");
//      strcpy(buffer, tmp);
    sprintf(buffer, "GPlot%ld", num);
//      free(tmp);                                                    70
    return buffer;
}

void
pause(){
    char buffer[128];
    gets(buffer);
}

                                                                      80
```

C.18 libDsp/libsrc/objects/SigGen.cc

```
/* FILE: /ti/class/druid/src/c++/libDsp/DSPtools.cc */
/* AUTHOR: Daniel F. Gruhl <druid@mit.edu> */
/* DATE LAST MODIFIED: Wed Jan  5 08:56:57 EST 1994 */
/* DESCRIPTION: This file contains the functions implemented in the  */
/* libDsp toolkit. Please see the libDsp texi file for full          */
/* odcumentation of these functions. */

static char rcsid[] = "$Id:  SigGen.cc,v 1.2 1994/08/01 12:27:08 druid Exp $";

#include <math.h>                                                     10
#include "SigGen.h"
#include "Dsp.h"

TimeSignal
DSPTriangleWave(double base, double time, double freq){
  long size = (long) (base * time);
  Complex* rval = new Complex[size];
  long SamplesPerCycle = (long) (freq * base) * 2;
  double delta = 1.0 / SamplesPerCycle;
  long QuarterSample = SamplesPerCycle / 4;                           20
  long ref;

  for (long i=0; i<base*time; i++){
    ref = (i + 3 * QuarterSample + 1) % SamplesPerCycle;
    if (ref < 2 * QuarterSample)
      rval[i] = 1 -  4.0 *  ref * delta;
    else
```

```
    rval[i] = -1 + 4.0 *  (ref - 2 * QuarterSample) * delta;
  }
  return TimeSignal(base, size, rval);
}

TimeSignal
DSPSquareWave(double base, double time, double freq){
  long size = (long) (base * time);
  Complex* rval = new Complex[size];
  long SamplesPerCycle = (long) (freq * base) * 2;
  for (long i=0; i<base*time; i++){
    if ( i % SamplesPerCycle < SamplesPerCycle / 2)
      rval[i] = 1.0;
    else
      rval[i] = -1.0;
  }
  return TimeSignal(base, size, rval);
}

TimeSignal
DSPSinSignal(double base, double time, double freq){
  long size = (long) (base * time);
  Complex* rval = new Complex[size];
  for (long i=0; i<base*time; i++)
    rval[i] = sin( freq / base * ((double) i) * 2.0 * PI );
  return TimeSignal(base, size, rval);
}

TimeSignal
DSPCosSignal(double base, double time, double freq){
  long size = (long) (base * time);
  Complex* rval = new Complex[size];
  for (long i=0; i<base*time; i++)
    rval[i] = cos( freq / base * ((double) i) * 2.0 * PI );
  return TimeSignal(base, size, rval);
}
```

C.19 libDsp/libsrc/objects/Slice.cc

```
// This may look like C, but it's really -*- C++ -*-
/* FILE: /User/druid/src/c++/libDsp/libsrc/Slice.cc */
/* AUTHOR: Daniel F. Gruhl <druid@mit.edu> */
/* DATE LAST MODIFIED: Wed May 18 08:27:58 EDT 1994 */
/* DESCRIPTION: This file does the sliceing of a TimeSignal into a      */
/* Frequency Slice. Yes... I know there should be an elegant way to do  */
/* this, but I can't think of a way to incorperate it right now. */


static char rcsid[] = "$Id:  Slice.cc,v 1.2 1994/08/01 12:27:08 druid Exp $";
```

```
#include "Dsp.h"
#include <iostream.h>
#include <stdio.h>
#include <math.h>

TimeSignal
UnSlice(FreqSlice& F){
   long windows     = F.Windows();
   long windowsize  = F.WindowSize();
   Complex *rval    = new Complex[windows * windowsize];
   double time_base = F.DeltaTime() / windowsize;

   long i, j, index;
   TimeSignal working;
   FreqSignal fworking;

   for (i=0; i<windows; i++){
         fworking = F[i];

//          fworking.data_out("dbg.dat");

         working = DSPIFT(fworking, time_base);
         index = i * windowsize;
         for (j=0; j<windowsize; j++)
            rval[j + index] = working[j];
   }

   return TimeSignal(time_base, windows * windowsize, rval);
}

FreqSlice
Slice(TimeSignal& T, int windowsize){
   // First off, find out how big our array we are going to be
   // dumping into is. We round up, because we will pad the last
   // entry with zeros to bring it to it's full length

   long slices = (long) ceil( ((double) T.size()/ (double)
                                        windowsize) );

   // Now allocate memory for slicing into. Because C++ doesn't
   // support argument passing to arrays on ititialization, we have
   // to do it by hand afterwards.

   FreqSignal* rval = new FreqSignal[slices];
   for (long i=0; i<slices; i++) rval[i].resize(windowsize);

   // Now, we do our slicing
   TimeSignal slicechunk(T.time_base(), windowsize);
   long index, j;

   for (i=0; i<slices-1; i++){
         index = i * windowsize;
         for (j=0; j<windowsize; j++)
```

```
            slicechunk[j] = T[index + j];
        rval[i] = DSPFT(slicechunk);
    }

    index = (slices - 1) * windowsize ;
    for (i=0; i<slicechunk.size(); i++) slicechunk[i] = 0.0;
    for (i=index; i<T.size(); i++)
        slicechunk[i - index] = T[i];
    rval[slices-1] = DSPFT(slicechunk);

    return FreqSlice(slices, rval, T.time_base()*windowsize);
}
```

C.20 libDsp/libsrc/objects/UniqueName.cc

```
#include <std.h>
#include <stdio.h>
#include <string.h>
#include "UniqueName.h"

UniqueName::UniqueName(){
    snum = max_snum++;
    ref_num = 1000;
}

int
UniqueName::number(){
    FILE* p = popen("echo $$", "r");
    int id;
    fscanf(p, "%d", &id);
    fclose(p);
    return id;
}

char*
UniqueName::name(){
    char buffer[256];
    sprintf(buffer, "%d%d%d", snum, ref_num++, number());
    return strdup(buffer);
}

char*
UniqueName::tmp_file_name(char* header){
    char buffer[256];
    char* nm = name();

    sprintf(buffer, "%s%s", header, nm);
    free(nm);
```

```
    return strdup(buffer);
}
```

C.21 libDsp/libsrc/objects/FreqSignal.cc

```
/* FILE: /ti/class/druid/src/c++/libDsp/FreqSignal.cc */
/* AUTHOR: Daniel F. Gruhl <druid@mit.edu> */
/* DATE LAST MODIFIED: Wed Jan  5 08:55:07 EST 1994 */
/* DESCRIPTION: This file provides the functions for the Frequency  */
/* Signal data type. Please see the libDsp texi file for full       */
/* documentation.  */

char static rcsid[] = "$Id:  FreqSignal.cc,v 1.2 1994/08/01 12:27:07 druid Exp $";

                                                                        10
#include "FreqSignal.h"
#include <stdio.h>

void FreqSignal::gplot(GPlot *g, char* title)
{
   CArray::gplot(g, title, "Frequency (Hz)", "Amplitude",
                 - Freq_Per_Bin * size() / 2.0, Freq_Per_Bin, size()/2);
}

FreqSignal::FreqSignal(long size, Complex* data)                        20
    : CArray( size, data )
{
 Freq_Per_Bin = 2.0 * PI / size;
}

FreqSignal::FreqSignal(long size, Complex* data, double FPB)
    : CArray( size, data )
{
 Freq_Per_Bin = FPB;
}                                                                       30

FreqSignal::FreqSignal(long size)
    : CArray( size )
{
 Freq_Per_Bin = 2.0 * PI / size;
}

FreqSignal::FreqSignal(CArrayGiveAway& c)
    : CArray( c )
{                                                                       40
 Freq_Per_Bin = 2.0 * PI / c.num_items;
}

void
FreqSignal::resize(long size)
{
 Freq_Per_Bin = 2.0 * PI / size;
```

```
  CArray::resize(size);
}

FreqSignal::FreqSignal()
    : CArray( 1 )
{
  Freq_Per_Bin = 0;
}

double
FreqSignal::frequency(long bin){
        if (bin > size() / 2.0)
            return  -Freq_Per_Bin * (double) (size() - bin);
        return Freq_Per_Bin * (double) bin;
}

void
 FreqSignal::data_out(char* filename){
   FILE* outfile = fopen(filename, "w");
   for (int i = 0; i<size(); i++)
     fprintf(outfile, "%lg\t %lg \t %lg\n",
             frequency(i), real(operator[](i)), imag(operator[](i)));

   fclose(outfile);
 }


FreqSignal&
FreqSignal::operator=(FreqSignal& rhs){
  Freq_Per_Bin = rhs.freq_per_bin();
  CArray::operator=( (CArray&) rhs );
  return *this;
}

FreqSignal&
FreqSignal::operator=(CArray& rhs){
  CArray::operator=( rhs );
  return *this;
}


FreqSignal&
FreqSignal::operator=(CArrayGiveAway& rhs){
  CArray::operator=( rhs );
  Freq_Per_Bin = 2.0 * PI / size();
  return *this;
}

FreqSignal
operator+(FreqSignal& lhs, FreqSignal& rhs){
  CArray intval = (CArray&) lhs + (CArray&) rhs;
  return FreqSignal(intval.giveaway());
}
```

```
FreqSignal
operator+(FreqSignal& lhs, int& rhs){
  CArray intval = (CArray&) lhs + rhs;
  return FreqSignal(intval.giveaway());
}

FreqSignal
operator+(FreqSignal& lhs, float& rhs){
  CArray intval = (CArray&) lhs + rhs;
  return FreqSignal(intval.giveaway());
}

FreqSignal
operator+(FreqSignal& lhs, double& rhs){
  CArray intval = (CArray&) lhs + rhs;
  return FreqSignal(intval.giveaway());
}

FreqSignal
operator+(FreqSignal& lhs, Complex& rhs){
  CArray intval = (CArray&) lhs + rhs;
  return FreqSignal(intval.giveaway());
}

FreqSignal
operator+(int& lhs, FreqSignal& rhs){
  CArray intval =  lhs + (CArray&) rhs;
  return FreqSignal(intval.giveaway());
}

FreqSignal
operator+(float& lhs, FreqSignal& rhs){
  CArray intval =  lhs + (CArray&) rhs;
  return FreqSignal(intval.giveaway());
}

FreqSignal
operator+(double& lhs, FreqSignal& rhs){
  CArray intval =  lhs + (CArray&) rhs;
  return FreqSignal(intval.giveaway());
}



FreqSignal
operator+(Complex& lhs, FreqSignal& rhs){
  CArray intval =  lhs + (CArray&) rhs;
  return FreqSignal(intval.giveaway());
}


FreqSignal
operator-(FreqSignal& lhs, FreqSignal& rhs){
  CArray intval = (CArray&) lhs - (CArray&) rhs;
```

```
  return FreqSignal(intval.giveaway());
}

FreqSignal
operator-(FreqSignal& lhs, int& rhs){
  CArray intval = (CArray&) lhs - rhs;
  return FreqSignal(intval.giveaway());
}

FreqSignal
operator-(FreqSignal& lhs, float& rhs){
  CArray intval = (CArray&) lhs - rhs;
  return FreqSignal(intval.giveaway());
}

FreqSignal
operator-(FreqSignal& lhs, double& rhs){
  CArray intval = (CArray&) lhs - rhs;
  return FreqSignal(intval.giveaway());
}

FreqSignal
operator-(FreqSignal& lhs, Complex& rhs){
  CArray intval = (CArray&) lhs - rhs;
  return FreqSignal(intval.giveaway());
}

FreqSignal
operator-(int& lhs, FreqSignal& rhs){
  CArray intval =  lhs - (CArray&) rhs;
  return FreqSignal(intval.giveaway());
}

FreqSignal
operator-(float& lhs, FreqSignal& rhs){
  CArray intval =  lhs - (CArray&) rhs;
  return FreqSignal(intval.giveaway());
}

FreqSignal
operator-(double& lhs, FreqSignal& rhs){
  CArray intval =  lhs - (CArray&) rhs;
  return FreqSignal(intval.giveaway());
}

FreqSignal
operator-(Complex& lhs, FreqSignal& rhs){
  CArray intval =  lhs - (CArray&) rhs;
  return FreqSignal(intval.giveaway());
}



FreqSignal
```

```
operator*(FreqSignal& lhs, FreqSignal& rhs){
  CArray intval = (CArray&) lhs * (CArray&) rhs;
  return FreqSignal(intval.giveaway());
}

FreqSignal
operator*(FreqSignal& lhs, int& rhs){
  CArray intval = (CArray&) lhs * rhs;
  return FreqSignal(intval.giveaway());
}

FreqSignal
operator*(FreqSignal& lhs, float& rhs){
  CArray intval = (CArray&) lhs * rhs;
  return FreqSignal(intval.giveaway());
}

FreqSignal
operator*(FreqSignal& lhs, double& rhs){
  CArray intval = (CArray&) lhs * rhs;
  return FreqSignal(intval.giveaway());
}

FreqSignal
operator*(FreqSignal& lhs, Complex& rhs){
  CArray intval = (CArray&) lhs * rhs;
  return FreqSignal(intval.giveaway());
}

FreqSignal
operator*(int& lhs, FreqSignal& rhs){
  CArray intval =  lhs * (CArray&) rhs;
  return FreqSignal(intval.giveaway());
}

FreqSignal
operator*(float& lhs, FreqSignal& rhs){
  CArray intval =  lhs * (CArray&) rhs;
  return FreqSignal(intval.giveaway());
}

FreqSignal
operator*(double& lhs, FreqSignal& rhs){
  CArray intval =  lhs * (CArray&) rhs;
  return FreqSignal(intval.giveaway());
}

FreqSignal
operator*(Complex& lhs, FreqSignal& rhs){
  CArray intval =  lhs * (CArray&) rhs;
  return FreqSignal(intval.giveaway());
}
```

```
FreqSignal
operator/(FreqSignal& lhs, FreqSignal& rhs){
  CArray intval = (CArray&) lhs / (CArray&) rhs;
  return FreqSignal(intval.giveaway());
}

FreqSignal
operator/(FreqSignal& lhs, int& rhs){
  CArray intval = (CArray&) lhs / rhs;
  return FreqSignal(intval.giveaway());
}

FreqSignal
operator/(FreqSignal& lhs, float& rhs){
  CArray intval = (CArray&) lhs / rhs;
  return FreqSignal(intval.giveaway());
}

FreqSignal
operator/(FreqSignal& lhs, double& rhs){
  CArray intval = (CArray&) lhs / rhs;
  return FreqSignal(intval.giveaway());
}

FreqSignal
operator/(FreqSignal& lhs, Complex& rhs){
  CArray intval = (CArray&) lhs / rhs;
  return FreqSignal(intval.giveaway());
}

FreqSignal
operator/(int& lhs, FreqSignal& rhs){
  CArray intval =  lhs / (CArray&) rhs;
  return FreqSignal(intval.giveaway());
}

FreqSignal
operator/(float& lhs, FreqSignal& rhs){
  CArray intval =  lhs / (CArray&) rhs;
  return FreqSignal(intval.giveaway());
}

FreqSignal
operator/(double& lhs, FreqSignal& rhs){
  CArray intval =  lhs / (CArray&) rhs;
  return FreqSignal(intval.giveaway());
}

FreqSignal
operator/(Complex& lhs, FreqSignal& rhs){
  CArray intval =  lhs / (CArray&) rhs;
  return FreqSignal(intval.giveaway());
}
```

```
FreqSignal&
FreqSignal::operator+=(FreqSignal& rhs){
  CArray::operator+=((CArray&) rhs);
  return *this;
}

FreqSignal&
FreqSignal::operator+=(int& rhs){
  CArray::operator+=(rhs);
  return *this;
}

FreqSignal&
FreqSignal::operator+=(float& rhs){
  CArray::operator+=(rhs);
  return *this;
}

FreqSignal&
FreqSignal::operator+=(double& rhs){
  CArray::operator+=(rhs);
  return *this;
}

FreqSignal&
FreqSignal::operator+=(Complex& rhs){
  CArray::operator+=(rhs);
  return *this;
}

FreqSignal&
FreqSignal::operator-=(FreqSignal& rhs){
  CArray::operator-=((CArray&) rhs);
  return *this;
}

FreqSignal&
FreqSignal::operator-=(int& rhs){
  CArray::operator-=(rhs);
  return *this;
}

FreqSignal&
FreqSignal::operator-=(float& rhs){
  CArray::operator-=(rhs);
  return *this;
}

FreqSignal&
FreqSignal::operator-=(double& rhs){
  CArray::operator-=(rhs);
  return *this;
}
```

```
FreqSignal&
FreqSignal::operator-=(Complex& rhs){
  CArray::operator-=(rhs);
  return *this;
}

FreqSignal&
FreqSignal::operator*=(FreqSignal& rhs){
  CArray::operator*=((CArray&) rhs);
  return *this;
}

FreqSignal&
FreqSignal::operator*=(int& rhs){
  CArray::operator*=(rhs);
  return *this;
}

FreqSignal&
FreqSignal::operator*=(float& rhs){
  CArray::operator*=(rhs);
  return *this;
}

FreqSignal&
FreqSignal::operator*=(double& rhs){
  CArray::operator*=(rhs);
  return *this;
}

FreqSignal&
FreqSignal::operator*=(Complex& rhs){
  CArray::operator*=(rhs);
  return *this;
}

FreqSignal&
FreqSignal::operator/=(FreqSignal& rhs){
  CArray::operator/=((CArray&) rhs);
  return *this;
}

FreqSignal&
FreqSignal::operator/=(int& rhs){
  CArray::operator/=(rhs);
  return *this;
}

FreqSignal&
FreqSignal::operator/=(float& rhs){
  CArray::operator/=(rhs);
  return *this;
}
```

```
FreqSignal&
FreqSignal::operator/=(double& rhs){
  CArray::operator/=(rhs);
  return *this;
}

FreqSignal&
FreqSignal::operator/=(Complex& rhs){
  CArray::operator/=(rhs);
  return *this;
}
```

C.22 libDsp/libsrc/objects/TimeSignal.cc

```
// This file may look like C, but its really -*- C++ -*-
/* FILE: /ti/class/druid/src/c++/libDsp/TimeSignal.cc */
/* AUTHOR: Daniel F. Gruhl <druid@mit.edu> */
/* DATE LAST MODIFIED: Wed Jan 5 08:46:47 EST 1994 */
/* DESCRIPTION: This file provides the procedures for the Time Signal */
/* data type. For a complete discussion of this Class please see the */
/* libDsp texi file. */

static
char rcsid[] = "$Id:  TimeSignal.cc,v 1.3 1994/08/01 12:27:09 druid Exp $";

#include "TimeSignal.h"
#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>
#include <math.h>
#include "aifif.h"

TimeSignal::~TimeSignal(){
#ifdef _FREE_TRACK
   cout << "Timesignal Pre Free data == " << whatdata() << "\n";
#endif
   Dealocate();
#ifdef _FREE_TRACK
   cout << "Timesignal Post Free data == " << whatdata() << "\n";
#endif
}

void
TimeSignal::normalize(){
   RangeVal r;
   r = maxposneg();

   double nval;
   if (fabs(r.maxpos) > fabs(r.maxneg)){
         nval = fabs(r.maxpos);
```

```
    } else {
        nval = fabs(r.maxneg);
    }

    for (long i=0; i<size(); i++)
        operator[](i) /= nval;
}

void
TimeSignal::gplot(GPlot *g, char* title){
    if (TimeBase == 0) TimeBase = 1.0;
  CArray::gplot(g, title, "Time (seconds)", "Amplitude", 0, 1.0 / TimeBase);
}


TimeSignal::TimeSignal(long size)
    : CArray( size )
{
  TimeBase = 1.0;
}

TimeSignal::TimeSignal(double base, long size)
    : CArray( size )
{
  TimeBase = base;
}

TimeSignal::TimeSignal(double base=1, double time=1)
    : CArray( (long) (base * time) )
{
  TimeBase = base;
}

TimeSignal::TimeSignal(double base, long size, Complex* data)
        : CArray ( size, data )
{
  TimeBase = base;
}

TimeSignal::TimeSignal(double base, CArray c)
        : CArray ( c )
{
  TimeBase = base;
}

RangeVal
TimeSignal::maxposneg(){
    RangeVal rval;
    rval.maxpos = 0;
    rval.maxneg = 0;
    for (long i=0; i<size(); i++)
        if ( real(operator[](i)) > rval.maxpos ) rval.maxpos =
                                                    real(operator[](i));
        else if ( real(operator[](i)) < rval.maxneg ) rval.maxneg =
```

```
                    real(operator[](i));
    return rval;
}

Quantized
TimeSignal::QData(){
    Quantized rval;
    rval.num = size();
    rval.data = new short[rval.num];
    double ScaleNum;                                                    100
    RangeVal rv = maxposneg();

    if (rv.maxpos == rv.maxneg) ;
    else {
        if (fabs(rv.maxpos) > fabs(rv.maxneg)) ScaleNum = 32000.0 / rv.maxpos;
        else ScaleNum = 32000.0 / fabs(rv.maxneg);

        for (long i=0; i<rval.num; i++)
            rval.data[i] = (short) (real(operator[](i) * ScaleNum));
    }                                                                   110
    return rval;
}

int
printRangeVal(RangeVal rv){
    printf("%lg %lg\n", rv.maxpos, rv.maxneg);
}
int
printComplex(Complex v){
    printf("%lg + %lg i\n", real(v), imag(v));                          120
}

TimeSignal::TimeSignal(double base, CArrayGiveAway& c)
    : CArray(c){
      TimeBase = base;
}

//TTimeSignal
//TTimeSignal::slice(long first, long last){
//    return TimeSignal(TimeBase, CArray::slice(first, last));          130
//}}

void
 TimeSignal::data_out(char* filename){
   FILE* outfile = fopen(filename, "w");
   for (int i = 0; i<size(); i++)
     fprintf(outfile, "%lg\t %lg \t %lg\n",
              (double) i / TimeBase, real(operator[](i)), imag(operator[](i)));

   fclose(outfile);                                                     140
 }

TimeSignal&
TimeSignal::operator=(TimeSignal& rhs){
```

```
  TimeBase = rhs.time_base();
  CArray::operator=( (CArray&) rhs );
  return *this;
}

TimeSignal&                                                          150
TimeSignal::operator=(CArray& rhs){
  CArray::operator=( rhs );
  return *this;
}


TimeSignal
operator+(TimeSignal& lhs, TimeSignal& rhs){
  if (lhs.time_base() != rhs.time_base() ||
      lhs.size() != rhs.size()){                                     160
    cerr << "Fatal Error:  <TimeSignal::operator+> Tried to add";
    cerr << "   TimeSignals with different parameters.  Bye...";
    exit(1);
  }
  CArray intval = (CArray&) lhs + (CArray&) rhs;
  return TimeSignal(rhs.time_base(), intval.giveaway());
}

TimeSignal
operator+(TimeSignal& lhs, int& rhs){                                170
  CArray intval = (CArray&) lhs + rhs;
  return TimeSignal(lhs.time_base(), intval.giveaway());
}

TimeSignal
operator+(TimeSignal& lhs, float& rhs){
  CArray intval = (CArray&) lhs + rhs;
  return TimeSignal(lhs.time_base(), intval.giveaway());
}
                                                                     180
TimeSignal
operator+(TimeSignal& lhs, double& rhs){
  CArray intval = (CArray&) lhs + rhs;
  return TimeSignal(lhs.time_base(), intval.giveaway());
}

TimeSignal
operator+(TimeSignal& lhs, Complex& rhs){
  CArray intval = (CArray&) lhs + rhs;
  return TimeSignal(lhs.time_base(), intval.giveaway());             190
}

TimeSignal
operator+(int& lhs, TimeSignal& rhs){
  CArray intval =  lhs + (CArray&) rhs;
  return TimeSignal(rhs.time_base(), intval.giveaway());
}
```

```
TimeSignal
operator+(float& lhs, TimeSignal& rhs){
  CArray intval =  lhs + (CArray&) rhs;
  return TimeSignal(rhs.time_base(), intval.giveaway());
}

TimeSignal
operator+(double& lhs, TimeSignal& rhs){
  CArray intval =  lhs + (CArray&) rhs;
  return TimeSignal(rhs.time_base(), intval.giveaway());
}


TimeSignal
operator+(Complex& lhs, TimeSignal& rhs){
  CArray intval =  lhs + (CArray&) rhs;
  return TimeSignal(rhs.time_base(), intval.giveaway());
}


TimeSignal
operator-(TimeSignal& lhs, TimeSignal& rhs){
  if (lhs.time_base() != rhs.time_base() ||
      lhs.size() != rhs.size()){
    cerr << "Fatal Error:  <TimeSignal::operator-> Tried to add";
    cerr << "   TimeSignals with different parameters.  Bye...";
    exit(1);
  }
  CArray intval = (CArray&) lhs - (CArray&) rhs;
  return TimeSignal(rhs.time_base(), intval.giveaway());
}

TimeSignal
operator-(TimeSignal& lhs, int& rhs){
  CArray intval = (CArray&) lhs - rhs;
  return TimeSignal(lhs.time_base(), intval.giveaway());
}

TimeSignal
operator-(TimeSignal& lhs, float& rhs){
  CArray intval = (CArray&) lhs - rhs;
  return TimeSignal(lhs.time_base(), intval.giveaway());
}

TimeSignal
operator-(TimeSignal& lhs, double& rhs){
  CArray intval = (CArray&) lhs - rhs;
  return TimeSignal(lhs.time_base(), intval.giveaway());
}

TimeSignal
operator-(TimeSignal& lhs, Complex& rhs){
  CArray intval = (CArray&) lhs - rhs;
```

```
  return TimeSignal(lhs.time_base(), intval.giveaway());
}

TimeSignal
operator-(int& lhs, TimeSignal& rhs){
  CArray intval = lhs - (CArray&) rhs;
  return TimeSignal(rhs.time_base(), intval.giveaway());
}                                                                    260

TimeSignal
operator-(float& lhs, TimeSignal& rhs){
  CArray intval = lhs - (CArray&) rhs;
  return TimeSignal(rhs.time_base(), intval.giveaway());
}

TimeSignal
operator-(double& lhs, TimeSignal& rhs){
  CArray intval = lhs - (CArray&) rhs;                               270
  return TimeSignal(rhs.time_base(), intval.giveaway());
}

TimeSignal
operator-(Complex& lhs, TimeSignal& rhs){
  CArray intval = lhs - (CArray&) rhs;
  return TimeSignal(rhs.time_base(), intval.giveaway());
}

                                                                     280

TimeSignal
operator*(TimeSignal& lhs, TimeSignal& rhs){
  if (lhs.time_base() != rhs.time_base() ||
      lhs.size() != rhs.size()){
    cerr << "Fatal Error:  <TimeSignal::operator*> Tried to add";
    cerr << "   TimeSignals with different parameters.  Bye...";
    exit(1);
  }
  CArray intval = (CArray&) lhs * (CArray&) rhs;                     290
  return TimeSignal(rhs.time_base(), intval.giveaway());
}

TimeSignal
operator*(TimeSignal& lhs, int& rhs){
  CArray intval = (CArray&) lhs * rhs;
  return TimeSignal(lhs.time_base(), intval.giveaway());
}

TimeSignal                                                           300
operator*(TimeSignal& lhs, float& rhs){
  CArray intval = (CArray&) lhs * rhs;
  return TimeSignal(lhs.time_base(), intval.giveaway());
}

TimeSignal
```

```
operator*(TimeSignal& lhs, double& rhs){
  CArray intval = (CArray&) lhs * rhs;
  return TimeSignal(lhs.time_base(), intval.giveaway());
}

TimeSignal
operator*(TimeSignal& lhs, Complex& rhs){
  CArray intval = (CArray&) lhs * rhs;
  return TimeSignal(lhs.time_base(), intval.giveaway());
}

TimeSignal
operator*(int& lhs, TimeSignal& rhs){
  CArray intval =  lhs * (CArray&) rhs;
  return TimeSignal(rhs.time_base(), intval.giveaway());
}

TimeSignal
operator*(float& lhs, TimeSignal& rhs){
  CArray intval =  lhs * (CArray&) rhs;
  return TimeSignal(rhs.time_base(), intval.giveaway());
}

TimeSignal
operator*(double& lhs, TimeSignal& rhs){
  CArray intval =  lhs * (CArray&) rhs;
  return TimeSignal(rhs.time_base(), intval.giveaway());
}

TimeSignal
operator*(Complex& lhs, TimeSignal& rhs){
  CArray intval =  lhs * (CArray&) rhs;
  return TimeSignal(rhs.time_base(), intval.giveaway());
}



TimeSignal
operator/(TimeSignal& lhs, TimeSignal& rhs){
  if (lhs.time_base() != rhs.time_base() ||
      lhs.size() != rhs.size()){
    cerr << "Fatal Error:  <TimeSignal::operator/> Tried to add";
    cerr << "    TimeSignals with different parameters.  Bye...";
    exit(1);
  }
  CArray intval = (CArray&) lhs / (CArray&) rhs;
  return TimeSignal(rhs.time_base(), intval.giveaway());
}

TimeSignal
operator/(TimeSignal& lhs, int& rhs){
  CArray intval = (CArray&) lhs / rhs;
  return TimeSignal(lhs.time_base(), intval.giveaway());
}
```

```
TimeSignal
operator/(TimeSignal& lhs, float& rhs){
  CArray intval = (CArray&) lhs / rhs;
  return TimeSignal(lhs.time_base(), intval.giveaway());
}

TimeSignal
operator/(TimeSignal& lhs, double& rhs){
  CArray intval = (CArray&) lhs / rhs;
  return TimeSignal(lhs.time_base(), intval.giveaway());
}

TimeSignal
operator/(TimeSignal& lhs, Complex& rhs){
  CArray intval = (CArray&) lhs / rhs;
  return TimeSignal(lhs.time_base(), intval.giveaway());
}

TimeSignal
operator/(int& lhs, TimeSignal& rhs){
  CArray intval =  lhs / (CArray&) rhs;
  return TimeSignal(rhs.time_base(), intval.giveaway());
}

TimeSignal
operator/(float& lhs, TimeSignal& rhs){
  CArray intval =  lhs / (CArray&) rhs;
  return TimeSignal(rhs.time_base(), intval.giveaway());
}

TimeSignal
operator/(double& lhs, TimeSignal& rhs){
  CArray intval =  lhs / (CArray&) rhs;
  return TimeSignal(rhs.time_base(), intval.giveaway());
}

TimeSignal
operator/(Complex& lhs, TimeSignal& rhs){
  CArray intval =  lhs / (CArray&) rhs;
  return TimeSignal(rhs.time_base(), intval.giveaway());
}

TimeSignal&
TimeSignal::operator+=(TimeSignal& rhs){
  if (time_base() != rhs.time_base() ||
      CArray::size() != rhs.size()){
    cerr << "Fatal Error:  <TimeSignal::operator/> Tried to add";
    cerr << "   TimeSignals with different parameters.  Bye...";
    exit(1);
  }
  CArray::operator+=((CArray&) rhs);
  return *this;
}
```

```
TimeSignal&
TimeSignal::operator+=(int& rhs){
  CArray::operator+=(rhs);
  return *this;
}

TimeSignal&
TimeSignal::operator+=(float& rhs){
  CArray::operator+=(rhs);
  return *this;
}

TimeSignal&
TimeSignal::operator+=(double& rhs){
  CArray::operator+=(rhs);
  return *this;
}

TimeSignal&
TimeSignal::operator+=(Complex& rhs){
  CArray::operator+=(rhs);
  return *this;
}

TimeSignal&
TimeSignal::operator-=(TimeSignal& rhs){
  if (time_base() != rhs.time_base() ||
      size() != rhs.size()){
    cerr << "Fatal Error:  <TimeSignal::operator/> Tried to add";
    cerr << "   TimeSignals with different parameters.  Bye...";
    exit(1);
  }
  CArray::operator-=((CArray&) rhs);
  return *this;
}

TimeSignal&
TimeSignal::operator-=(int& rhs){
  CArray::operator-=(rhs);
  return *this;
}

TimeSignal&
TimeSignal::operator-=(float& rhs){
  CArray::operator-=(rhs);
  return *this;
}

TimeSignal&
TimeSignal::operator-=(double& rhs){
  CArray::operator-=(rhs);
  return *this;
}
```

```
TimeSignal&                                                                    470
TimeSignal::operator-=(Complex& rhs){
  CArray::operator-=(rhs);
  return *this;
}

TimeSignal&
TimeSignal::operator*=(TimeSignal& rhs){
  if (time_base() != rhs.time_base() ||
      size() != rhs.size()){
    cerr << "Fatal Error:  <TimeSignal::operator/> Tried to add";              480
    cerr << "   TimeSignals with different parameters.  Bye...";
    exit(1);
  }
  CArray::operator*=((CArray&) rhs);
  return *this;
}

TimeSignal&
TimeSignal::operator*=(int& rhs){
  CArray::operator*=(rhs);                                                     490
  return *this;
}

TimeSignal&
TimeSignal::operator*=(float& rhs){
  CArray::operator*=(rhs);
  return *this;
}

TimeSignal&                                                                    500
TimeSignal::operator*=(double& rhs){
  CArray::operator*=(rhs);
  return *this;
}

TimeSignal&
TimeSignal::operator*=(Complex& rhs){
  CArray::operator*=(rhs);
  return *this;
}                                                                              510

TimeSignal&
TimeSignal::operator/=(TimeSignal& rhs){
  if (time_base() != rhs.time_base() ||
      size() != rhs.size()){
    cerr << "Fatal Error:  <TimeSignal::operator/> Tried to add";
    cerr << "   TimeSignals with different parameters.  Bye...";
    exit(1);
  }
  CArray::operator/=((CArray&) rhs);                                           520
  return *this;
}
```

```
TimeSignal&
TimeSignal::operator/=(int& rhs){
  CArray::operator/=(rhs);
  return *this;
}

TimeSignal&                                                              530
TimeSignal::operator/=(float& rhs){
  CArray::operator/=(rhs);
  return *this;
}

TimeSignal&
TimeSignal::operator/=(double& rhs){
  CArray::operator/=(rhs);
  return *this;
}                                                                        540

TimeSignal&
TimeSignal::operator/=(Complex& rhs){
  CArray::operator/=(rhs);
  return *this;
}

int
TimeSignal::readAiff(char* filename){
  AIF_STRUCT *aifs;                                                      550
  short *buf;
  int i;
  long pvb[16], pvl;

  // Open reading file.
  aifs = aifNew();
  if ((aifOpenRead(aifs, filename))){
    cerr << "Cannot open AIFF file <" << filename << "> for reading.  Bye...\n";
    exit(1);
  }                                                                      560

  // Read in the parameters of this file
  pvl = 0;
  pvb[pvl++] = AIF_P_FILETYPE;    ++pvl; /* 1  */
  pvb[pvl++] = AIF_P_NFRAMES;     ++pvl; /* 3  */
  pvb[pvl++] = AIF_P_SAMPSIZE;    ++pvl; /* 5  */
  pvb[pvl++] = AIF_P_CHANNELS;    ++pvl; /* 7  */
  pvb[pvl++] = AIF_P_SAMPRATE;    ++pvl; /* 9  */
  pvb[pvl++] = AIF_P_COMPID;      ++pvl; /* 11 */
  pvb[pvl++] = AIF_P_COMPNAME;    ++pvl; /* 13 */                        570
  aifGetParams(aifs, pvb, pvl);



#ifdef _PRINT_AIFF_DATA
  printf("-*- Reading In Data -*-\n");
```

```
  printf("Sampling Rate:     %g\n", FLOATofLONG(pvb[9]));
  printf("Channels:               %ld\n", pvb[7]);
  printf("Bits:                        %ld\n", pvb[5]);
  printf("Frames in file:  %ld\n", pvb[3]);                                    580
  printf("Summery of params:\n");
   for (long prms = 0; prms < 14; prms++)
      printf("\tparam[%ld] = %ld\n", prms, pvb[prms]);
#endif

  if (pvb[5] != 16){
     cerr << "Tried to read non 16 bit signal.  Bye...\n";
     exit(1);
  }
                                                                             590
  resize(pvb[3]);

  TimeBase = (double) FLOATofLONG(pvb[9]);

  buf = new short[pvb[3]];
  aifReadFrames(aifs, buf, pvb[3]);
  aifClose(aifs);

  for (i=0; i<pvb[3]; i++)
    (*this)[i] = (Complex) buf[i];                                           600

  delete [] buf;
  aifFree(aifs);

  return 1;
}

int
TimeSignal::writeAiff(char* filename){
  AIF_STRUCT *aifs;                                                          610
  BYTE       *buf;
  long pvb[16], pvl;
  Quantized  q;

  aifs  = aifNew();

  pvl = 0;
  pvb[pvl++] = AIF_P_FILETYPE;    ++pvl;  /* 1  */
  pvb[pvl++] = AIF_P_NFRAMES;     ++pvl;  /* 3  */
  pvb[pvl++] = AIF_P_SAMPSIZE;    ++pvl;  /* 5  */                          620
  pvb[pvl++] = AIF_P_CHANNELS;    ++pvl;  /* 7  */
  pvb[pvl++] = AIF_P_SAMPRATE;    ++pvl;  /* 9  */

  pvb[1] = 1;
  pvb[3] = (long) size();
  pvb[5] = 16;
  pvb[7] = 1;
  float srate = (float) TimeBase;
  pvb[9] = LONGofFLOAT( srate );
  aifSetParams(aifs, pvb, pvl);                                              630
```

```
#ifdef _PRINT_AIFF_DATA
  printf("-*- Write Out Data -*-\n");
  printf("Sampling Rate:     %g\n", FLOATofLONG(pvb[9]));
  printf("Channels:               %ld\n", pvb[7]);
  printf("Bits:                        %ld\n", pvb[5]);
  printf("Frames in file:  %ld\n", pvb[3]);
  printf("Summery of params:\n");
   for (long prms = 0; prms < 14; prms++)
      printf("\tparam[%ld] = %ld\n", prms, pvb[prms]);                    640
#endif

  if ((aifOpenWrite(aifs, filename, UNK_LEN) < 0)){
    cerr << "Cannot open AIFF file <" << filename << "> for writing.  Bye...\n";
    exit(1);
  }

  q = (*this).QData();
  buf = (BYTE *) q.data;
  aifWriteFrames(aifs, buf, pvb[3]);                                      650
  aifClose(aifs);
  aifFree(aifs);
  delete [] buf;
  return 1;
}


void
TimeSignal::resize(long newsize){
  CArray::resize(newsize);                                                660
}

void
TimeSignal::stretch(long newsize){
  CArray::stretch(newsize);
}

TimeSignal
TimeSignal::delay(double seconds){
    long samples = (long) rint(seconds/time_base());                      670
    TimeSignal rval = delay(samples);
    return TimeSignal((*this).time_base(), rval.giveaway());
}

TimeSignal
TimeSignal::delay(long samples){
    long i;

    Complex* rval = new Complex[ (*this).size() ];
                                                                          680
    long start, stop, off;

    off   = -samples;
    start = -samples;
```

```
    stop  = (*this).size() + samples;

    if (start < 0) start = 0;
    if (stop >= (*this).size()) stop = (*this).size() - 1;

    for (i=start; i<stop; i++)                                          690
        rval[i] = (*this)[i - off];

    return TimeSignal((*this).time_base(), (*this).size(), rval);
}
```

C.23 libDsp/libsrc/objects/DSPtools.cc

```
/* FILE: /ti/class/druid/src/c++/libDsp/DSPtools.cc */
/* AUTHOR: Daniel F. Gruhl <druid@mit.edu> */
/* DATE LAST MODIFIED: Tue Feb  1 11:57:40 EST 1994 */
/* DESCRIPTION: This file contains the functions implemented in the */
/* libDsp toolkit. Please see the libDsp texi file for full        */
/* odcumentation of these functions.                                */

static char rcsid[] = "$Id:  DSPtools.cc,v 1.3 1994/08/01 12:27:05 druid Exp $";

#include <math.h>                                                       10

#ifdef NEEDS_GNU_COMPLEX
#include "GnuExtras/Complex.h"
#else
#include <Complex.h>
#endif

#include "TimeSignal.h"

TimeSignal                                                              20
FullWaveRectifier(TimeSignal& t){
  Complex* c = new Complex[t.size()];
  register long s = t.size();
  for (long i=0; i<s; i++)
    c[i] = fabs(real(t[i]));
  return TimeSignal(t.time_base(), s, c);
}

TimeSignal
HalfWaveRectifier(TimeSignal& t){                                       30
  Complex* c = new Complex[t.size()];
  register long s = t.size();
  for (long i=0; i<s; i++){
    if (real(t[i]) > 0.0)
      c[i] = t[i];
    else
      t[i] = 0.0 ;
  }
  //  ^ Magic for, if t is positive, assign it to c, otherwise make c = 0
```

```
  return TimeSignal(t.time_base(), s, c);
}


TimeSignal
ZeroPad(long num, TimeSignal& T){
  Complex* c = new Complex[num];
  for (long i=0; i<T.size(); i++)
    c[i] = T[i];
  for (i = T.size(); i<num; i++)
    c[i] = 0.0;
  return TimeSignal(T.time_base(), num, c);
}
```

C.24 libDsp/libsrc/objects/Filters.cc

```
#include "Filters.h"


TimeSignal
Filter(TimeSignal& t, FilterKernal k){
   Complex *w, *outdata;
   long fbnum, i, j;

   // Make and initialize storage
   if (k.an > k.bn) fbnum = k.an;
   else fbnum = k.bn;
   w = new Complex[fbnum];
   for (i=0; i<fbnum; i++)  w[0] = 0;

   // Allocate output storage
   outdata = new Complex[t.size()];

   // Do filter run
   for (i=0; i<t.size(); i++){
         //CCalculate new feedback term
         w[0] = t[i];
         for (j=1; j<k.an; j++)
            w[0] += k.a[j] * w[j];

         // And the feed forward term
         for (j=0; j<k.bn; j++)
            outdata[i] += k.b[j] * w[j];

         // And shuffle the stored data
         for (j=fbnum-1; j>0; j--)
            w[j] = w[j-1];
   }
```

```
    delete [] w;
    return TimeSignal(t.time_base(), t.size(), outdata);
}

TimeSignal
ApplyFIR(TimeSignal& t, TimeSignal& filt){                                    40
    Complex *c = new Complex[t.size()];
    long i, j, index;
    long num = filt.size();

    for (i=0; i<t.size(); i++){
        c[i] = 0;
        for (j=0; j<num; j++){
            index = i - num/2 + j;
            if (index > 0 || index < t.size())
                c[i] += t[index] * filt[j];                                   50
        }
    }

    return TimeSignal(t.time_base(), t.size(), c);
}

// From Op and Sch p. 447
TimeSignal
DSPWindowBartlett(TimeSignal& t){
    Complex* c = new Complex[t.size()];                                       60
    long M = t.size();

    // Actually do the window
    for (long i=0; i<=M/2; i++)
        c[i] = t[i] * ( 2.0 / ( (double) M ) * ( (double) ((i + M/2) % M)));
    for (i=M/2+1; i<M; i++)
        c[i] = t[i] * ( 2.0 - 2.0 / (double) M * (double) ((i + M/2) % M));

    return TimeSignal(t.time_base(), t.size(), c);
}                                                                             70

// From Op and Sch p. 447
TimeSignal
DSPWindowHanning(TimeSignal& t){
    Complex* c = new Complex[t.size()];
    long M = t.size();

    // Actually do the window
    for (long i=0; i<M; i++)
        c[i] = t[i] * ( .5 - .5 * cos                                         80
                        ( 2 * PI * (double) ( (i + M/2) % M) / (double) M));

    return TimeSignal(t.time_base(), t.size(), c);
}

// From Op and Sch p. 447
TimeSignal
DSPWindowHamming(TimeSignal& t){
```

```
    Complex* c = new Complex[t.size()];
    long M = t.size();                                                    90

    // Actually do the window
    for (long i=0; i<M; i++)
        c[i] = t[i] * ( .54 - .46 * cos
                        ( 2 * PI * (double) ((i + M/2) % M) / (double) M));

    return TimeSignal(t.time_base(), t.size(), c);
}

                                                                          100
```

C.25 libDsp/libsrc/objects/CArray.cc

```
// This may look like C code, but it's really -*- C++ -*-
/* FILE: /ti/class/druid/src/c++/libDsp/CArray.cc */
/* AUTHOR: Daniel F. Gruhl <druid@mit.edu> */
/* DATE LAST MODIFIED: Tue May 17 08:18:27 EDT 1994 */
/* DESCRIPTION: This file contains the functions necessary for the     */
/* Complex array data type. For a full discussion of this data type,   */
/* please see the libDsp texi file.                                    */

char static rcsid[] = "$Id:  CArray.cc,v 1.2 1994/08/01 12:27:05 druid Exp $";
                                                                          10
#include "CArray.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <String.h>

/*
@setfilename carray-info
SECTION
     CArray                                                               20

DESCRIPTION
     A CArray is an array of Complex variables.

     Don't you wish you had some?

*/


/*                                                                        30
FUNCTION
      CArray::AngleArray
```

```
SYNOPSIS
      DblVector& CArray::AngleArray,
       ();

DESCRIPTION
      Return a @var{DblVector} of the phase angle of every point in the
      CArray.                                                          40
*/


double*
CArray::AngleArray(double* mdata){
   double* rval = mdata;
   if (rval == NULL){
         cerr << "CArray::PhaseArray --- cannot allocate memory...\n";
         exit(1);
   }                                                                   50
   for (long i = 0; i<size(); i++)
         rval[i] = arg(data[i]);
   return rval;
}

/*
FUNCTION
    CArray::MagnitudeArray

SYNOPSIS                                                              60
    DblVector& CArray::MagnitudeArray();

DESCRIPTION
      Return a @var{DblVector} of the magnitude of every point in the
      CArray.
*/

double*
CArray::MagnitudeArray(double* mdata){
   double* rval = mdata;                                              70
   for (long i = 0; i<size(); i++)
         rval[i] = abs(data[i]);
   return rval;
}

//CContructors and destructors for CArray. These provide for the
//ccreation and destruction of Complex Arrays

//CCArray::CArray(){
//   num_items = 0;                                                   80
//   data = new Complex[1];      // If the array does not exist I just
//                              // give it one element. This simplifies
//                              // the destruction and resizing commands.
//}}

long
```

```
CArray::setsize (long items){
 num_items = items;
 data = new Complex[items];
#ifdef _FREE_TRACK                                                    90
   cerr << "CARRAY is Creating " << (int) data << "\n" << flush;
#endif
 return items;
}

CArray::CArray (long items=1){
 if (items < 0) {
   cerr << "FATAL ERROR: <CArray::CArray(long)> Attempted create\n";
   cerr << "    an array with a negative number of elements.  Bye...\n";
   exit(1);                                                           100
 }
 setsize(items);
}

CArray::~CArray(){
   Dealocate();
}

void
CArray::Dealocate(){                                                  110
#ifdef _FREE_TRACK
   cerr << "CARRAY is Thinking of Freeing " << (int) data << "\n" << flush;
#endif
 if (data != NULL){
#ifdef _FREE_TRACK
    cerr << "CARRAY is Freeing " << (int) data << "\n" << flush;
#endif
    delete [] data;
 }
   data = NULL;                                                       120
}

// Fun with sizing
 // Return the number of elements in the stored array. I know this
 // isn't really the "size" in bytes but it's the convention I'm using.
long
CArray::size(){
        return num_items;
}
                                                                      130
 // Resize just trashes the array and creates a new one.
void
CArray::resize(long newsize){
 delete [] data;
 setsize(newsize);
}

 // Stretch makes a new array of a given size and copies the elements
 // over to the new array. Truncation of extra elements is done
void                                                                  140
```

```
CArray::stretch(long newsize){
  Complex* hold = data;
  setsize(newsize);
  for (int i=0; i< ( newsize < num_items ? newsize : num_items); i++)
    data[i] = hold[i];
  delete [] hold;
  return;
}

CArray                                                              150
CArray::slice(long first, long last){
  long size = last - first + 1;
  Complex* c = new Complex[size];
  for (long i=0; i<size; i++)
    c[i] = data[i+first];
  return CArray(size, c);
}


// Operator Magic                                                   160
// Returns the actual array element indexed.
Complex&
CArray::operator[] (long index){
#ifdef DEBUG1
  if (index < 0 || index >= num_items){
    cerr << "FATAL ERROR: <CArray::operator[]> Attempted to index\n";
    cerr << "    element outside of array bounds.  Bye...\n";
    exit(1);
  }
#endif                                                              170
        return data[index];
}

  // Equal in many flavors...
CArray&
CArray::operator=(CArray& rhs){
  if (this==&rhs) return *this; // If someone has been clever....
  if (num_items != rhs.size()) {
    delete [] data;
    num_items = rhs.size();                                         180
    data = new Complex[num_items];
  }
  for (int i=0; i<num_items; i++)
    data[i] = rhs[i];

  return *this;
}


  // Assigning a int                                               190
CArray&
CArray::operator=(int& rhs){
  for (int i=0; i<num_items; i++)
    data[i] = rhs;
```

```
  return *this;
}

  // Assigning a int
CArray&
CArray::operator=(float& rhs){
  for (int i=0; i<num_items; i++)
    data[i] = rhs;
  return *this;
}

  // Assigning a int
CArray&
CArray::operator=(double& rhs){
  for (int i=0; i<num_items; i++)
    data[i] = rhs;
  return *this;
}

  // Assigning a int
CArray&
CArray::operator=(Complex& rhs){
  for (int i=0; i<num_items; i++)
    data[i] = rhs;
  return *this;
}

CArray&
CArray::operator=(CArrayGiveAway& c){
  if (data != NULL) delete [] data;
  data = c.data;
  num_items = c.num_items;
  c.data = NULL;
  c.num_items = 0;
  return *this;
}

// operator +=... Define for all
CArray&
CArray::operator+=(CArray& rhs){
  if (size() != rhs.size()){
    cerr << "Fatal Error:  <CArray::operator+=> tried to add two arrays\n";
    cerr << "     of different size.  Bye...";
    exit(1);
  }

  for (long i=0; i<num_items; i++)
    data[i] += rhs[i];

  return *this;
}

CArray&
CArray::operator+=(int& rhs){
```

```
  for (long i=0; i<num_items; i++)
    data[i] += rhs;
  return *this;
}

CArray&
CArray::operator+=(float& rhs){
  for (long i=0; i<num_items; i++)
    data[i] += rhs;
  return *this;
}

CArray&
CArray::operator+=(double& rhs){
  for (long i=0; i<num_items; i++)
    data[i] += rhs;
  return *this;
}

CArray&
CArray::operator+=(Complex& rhs){
  for (long i=0; i<num_items; i++)
    data[i] += rhs;
  return *this;
}

// operator -=... Define for all
CArray&
CArray::operator-=(CArray& rhs){
  if (size() != rhs.size()){
    cerr << "Fatal Error:  <CArray::operator+=> tried to add two arrays\n";
    cerr << "     of different size.  Bye...";
    exit(1);
  }

  for (long i=0; i<num_items; i++)
    data[i] -= rhs[i];

  return *this;
}

CArray&
CArray::operator-=(int& rhs){
  for (long i=0; i<num_items; i++)
    data[i] -= rhs;
  return *this;
}

CArray&
CArray::operator-=(float& rhs){
  for (long i=0; i<num_items; i++)
    data[i] -= rhs;
  return *this;
}
```

```
CArray&
CArray::operator-=(double& rhs){
  for (long i=0; i<num_items; i++)
    data[i] -= rhs;
  return *this;
}

CArray&
CArray::operator-=(Complex& rhs){
  for (long i=0; i<num_items; i++)
    data[i] -= rhs;
  return *this;
}

// operator *=... Define for all
CArray&
CArray::operator*=(CArray& rhs){
  if (size() != rhs.size()){
    cerr << "Fatal Error:  <CArray::operator+=> tried to add two arrays\n";
    cerr << "     of different size.  Bye...";
    exit(1);
  }

  for (long i=0; i<num_items; i++)
    data[i] *= rhs[i];

  return *this;
}

CArray&
CArray::operator*=(int& rhs){
  for (long i=0; i<num_items; i++)
    data[i] *= rhs;
  return *this;
}

CArray&
CArray::operator*=(float& rhs){
  for (long i=0; i<num_items; i++)
    data[i] *= rhs;
  return *this;
}

CArray&
CArray::operator*=(double& rhs){
  for (long i=0; i<num_items; i++)
    data[i] *= rhs;
  return *this;
}

CArray&
CArray::operator*=(Complex& rhs){
  for (long i=0; i<num_items; i++)
```

```
    data[i] *= rhs;
  return *this;
}
                                                                    360
// operator /=... Define for all
CArray&
CArray::operator/=(CArray& rhs){
  if (size() != rhs.size()){
    cerr << "Fatal Error:  <CArray::operator+=> tried to add two arrays\n";
    cerr << "     of different size.  Bye...";
    exit(1);
  }

  for (long i=0; i<num_items; i++)                                  370
    data[i] /= rhs[i];

  return *this;
}

CArray&
CArray::operator/=(int& rhs){
  for (long i=0; i<num_items; i++)
    data[i] /= rhs;
  return *this;                                                     380
}

CArray&
CArray::operator/=(float& rhs){
  for (long i=0; i<num_items; i++)
    data[i] /= rhs;
  return *this;
}

CArray&                                                             390
CArray::operator/=(double& rhs){
  for (long i=0; i<num_items; i++)
    data[i] /= rhs;
  return *this;
}

CArray&
CArray::operator/=(Complex& rhs){
  for (long i=0; i<num_items; i++)
    data[i] /= rhs;                                                 400
  return *this;
}



// operator +... First, adding two complex arrays. They MUST be the
// same size.
CArray
operator+(CArray& lhs, CArray& rhs){                                410
```

```
  if (lhs.size() != rhs.size()){
    cerr << "Fatal Error:  <operator+(CArray,CArray)> Tried to add";
    cerr << "Two streams of non identical size.  Bye...";
  }
  Complex *result = new Complex[lhs.size()];
  for (int i=0; i<lhs.size(); i++)
    result[i] = lhs[i] + rhs[i];
  return CArray(lhs.size(), result);
}

CArray
operator+(CArray& lhs, int& rhs){
  Complex* result = new Complex[lhs.size()];
  register long l = lhs.size();
  for (int i=0; i<l; i++)
    result[i] = lhs[i] + rhs;
  return CArray(lhs.size(), result);
}

CArray
operator+(CArray& lhs, float& rhs){
  Complex* result = new Complex[lhs.size()];
  for (int i=0; i<lhs.size(); i++)
    result[i] = lhs[i] + rhs;
  return CArray(lhs.size(), result);
}

CArray
operator+(CArray& lhs, double& rhs){
  register long l = lhs.size();
  Complex* result = new Complex[lhs.size()];
  for (int i=0; i<l; i++)
    result[i] = lhs[i] + rhs;
  return CArray(lhs.size(), result);
}

CArray
operator+(CArray& lhs, Complex& rhs){
  Complex* result = new Complex[lhs.size()];
  for (int i=0; i<lhs.size(); i++)
    result[i] = lhs[i] + rhs;
  return CArray(lhs.size(), result);
}

CArray
operator+(int& lhs, CArray& rhs){
  Complex* result = new Complex[rhs.size()];
  for (int i=0; i<rhs.size(); i++)
    result[i] = lhs + rhs[i];
  return CArray(rhs.size(), result);
}
CArray
operator+(float& lhs, CArray& rhs){
  Complex* result = new Complex[rhs.size()];
```

```
  for (int i=0; i<rhs.size(); i++)
    result[i] = lhs + rhs[i];
  return CArray(rhs.size(), result);
}
CArray
operator+(double& lhs, CArray& rhs){
  Complex* result = new Complex[rhs.size()];
  for (int i=0; i<rhs.size(); i++)
    result[i] = lhs + rhs[i];
  return CArray(rhs.size(), result);
}
CArray
operator+(Complex& lhs, CArray& rhs){
  Complex* result = new Complex[rhs.size()];
  for (int i=0; i<rhs.size(); i++)
    result[i] = lhs + rhs[i];
  return CArray(rhs.size(), result);
}


// operator -... First, subtracting two complex arrays. They MUST be the
// same size.
CArray
operator-(CArray& lhs, CArray& rhs){
  if (lhs.size() != rhs.size()){
    cerr << "Fatal Error:  <operator+(CArray,CArray)> Tried to add";
    cerr << "Two streams of non identical size.  Bye...";
  }
  Complex* result = new Complex[lhs.size()];
  for (int i=0; i<lhs.size(); i++)
    result[i] = lhs[i] - rhs[i];
  return CArray(lhs.size(), result);
}

CArray
operator-(CArray& lhs, int& rhs){
  Complex* result = new Complex[lhs.size()];
  for (int i=0; i<lhs.size(); i++)
    result[i] = lhs[i] - rhs;
  return CArray(lhs.size(), result);
}

CArray
operator-(CArray& lhs, float& rhs){
  Complex* result = new Complex[lhs.size()];
  for (int i=0; i<lhs.size(); i++)
    result[i] = lhs[i] - rhs;
  return CArray(lhs.size(), result);
}

CArray
operator-(CArray& lhs, double& rhs){
  Complex* result = new Complex[lhs.size()];
  for (int i=0; i<lhs.size(); i++)
```

```
    result[i] = lhs[i] - rhs;
  return CArray(lhs.size(), result);
}

CArray
operator-(CArray& lhs, Complex& rhs){
  Complex* result = new Complex[lhs.size()];
  for (int i=0; i<lhs.size(); i++)
    result[i] = lhs[i] - rhs;
  return CArray(lhs.size(), result);
}

CArray
operator-(int& lhs, CArray& rhs){
  Complex* result = new Complex[rhs.size()];
  for (int i=0; i<rhs.size(); i++)
    result[i] = lhs - rhs[i];
  return CArray(rhs.size(), result);
}

CArray
operator-(float& lhs, CArray& rhs){
  Complex* result = new Complex[rhs.size()];
  for (int i=0; i<rhs.size(); i++)
    result[i] = lhs - rhs[i];
  return CArray(rhs.size(), result);
}

CArray
operator-(double& lhs, CArray& rhs){
  Complex* result = new Complex[rhs.size()];
  for (int i=0; i<rhs.size(); i++)
    result[i] = lhs - rhs[i];
  return CArray(rhs.size(), result);
}

CArray
operator-(Complex& lhs, CArray& rhs){
  Complex* result = new Complex[rhs.size()];
  for (int i=0; i<rhs.size(); i++)
    result[i] = lhs - rhs[i];
  return CArray(rhs.size(), result);
}

// operator *... First, multiply two complex arrays. They MUST be the
// same size.
CArray
operator*(CArray& lhs, CArray& rhs){
  if (lhs.size() != rhs.size()){
    cerr << "Fatal Error:  <operator+(CArray,CArray)> Tried to add";
    cerr << "Two streams of non identical size.  Bye...";
  }
  Complex* result = new Complex[lhs.size()];
  for (int i=0; i<lhs.size(); i++)
```

```
    result[i] = lhs[i] * rhs[i];
  return CArray(lhs.size(), result);
}

CArray
operator*(CArray& lhs, int& rhs){
  Complex* result = new Complex[lhs.size()];
  for (int i=0; i<lhs.size(); i++)                                    580
    result[i] = lhs[i] * rhs;
  return CArray(lhs.size(), result);
}

CArray
operator*(CArray& lhs, float& rhs){
  Complex* result = new Complex[lhs.size()];
  for (int i=0; i<lhs.size(); i++)
    result[i] = lhs[i] * rhs;
  return CArray(lhs.size(), result);                                  590
}

CArray
operator*(CArray& lhs, double& rhs){
  Complex* result = new Complex[lhs.size()];
  for (int i=0; i<lhs.size(); i++)
    result[i] = lhs[i] * rhs;
  return CArray(lhs.size(), result);
}
                                                                      600
CArray
operator*(CArray& lhs, Complex& rhs){
  Complex* result = new Complex[lhs.size()];
  for (int i=0; i<lhs.size(); i++)
    result[i] = lhs[i] * rhs;
  return CArray(lhs.size(), result);
}

CArray
operator*(int& lhs, CArray& rhs){                                     610
  Complex* result = new Complex[rhs.size()];
  for (int i=0; i<rhs.size(); i++)
    result[i] = lhs * rhs[i];
  return CArray(rhs.size(), result);
}

CArray
operator*(float& lhs, CArray&rhs){
  Complex* result = new Complex[rhs.size()];
  for (int i=0; i<rhs.size(); i++)                                    620
    result [i] = lhs * rhs[i];
  return CArray(rhs.size(), result);
}

CArray
operator*(double& lhs, CArray& rhs){
```

```
Complex* result = new Complex[rhs.size()];
  for (int i=0; i<rhs.size(); i++)
    result[i] = lhs * rhs[i];
    return CArray(rhs.size(), result);                                    630
}

CArray
operator*(Complex& lhs, CArray& rhs){
Complex* result = new Complex[rhs.size()];
  for (int i=0; i<rhs.size(); i++)
    result[i] = lhs * rhs[i];
    return CArray(rhs.size(), result);
}
                                                                          640
// operator /... First, divide two complex arrays. They MUST be the
// same size.
CArray
operator/(CArray& lhs, CArray& rhs){
  if (lhs.size() != rhs.size()){
    cerr << "Fatal Error:  <operator+(CArray,CArray)> Tried to add";
    cerr << "Two streams of non identical size.  Bye...";
  }
Complex* result = new Complex[lhs.size()];
  for (int i=0; i<lhs.size(); i++)                                        650
    result[i] = lhs[i] / rhs[i];
    return CArray(lhs.size(), result);
}

CArray
operator/(CArray& lhs, int& rhs){
Complex* result = new Complex[lhs.size()];
  for (int i=0; i<lhs.size(); i++)
    result[i] = lhs[i] / rhs;
    return CArray(lhs.size(), result);                                    660
}

CArray
operator/(CArray& lhs, float& rhs){
Complex* result = new Complex[lhs.size()];
  for (int i=0; i<lhs.size(); i++)
    result[i] = lhs[i] / rhs;
    return CArray(lhs.size(), result);
}
                                                                          670
CArray
operator/(CArray& lhs, double& rhs){
Complex* result = new Complex[lhs.size()];
  for (int i=0; i<lhs.size(); i++)
    result[i] = lhs[i] / rhs;
    return CArray(lhs.size(), result);
}

CArray
operator/(CArray& lhs, Complex& rhs){                                     680
```

```
Complex* result = new Complex[lhs.size()];
  for (int i=0; i<lhs.size(); i++)
    result[i] = lhs[i] / rhs;
    return CArray(lhs.size(), result);
}


CArray
operator/(int& lhs, CArray& rhs){
Complex* result = new Complex[rhs.size()];                              690
  for (int i=0; i<rhs.size(); i++)
    result[i] = lhs / rhs[i];
    return CArray(rhs.size(), result);
}

CArray
operator/(float& lhs, CArray& rhs){
Complex* result = new Complex[rhs.size()];
  for (int i=0; i<rhs.size(); i++)
    result[i] = lhs / rhs[i];                                           700
    return CArray(rhs.size(), result);
}

CArray
operator/(double& lhs, CArray& rhs){
Complex* result = new Complex[rhs.size()];
  for (int i=0; i<rhs.size(); i++)
    result[i] = lhs / rhs[i];
    return CArray(rhs.size(), result);
}                                                                       710

CArray
operator/(Complex& lhs, CArray& rhs){
Complex* result = new Complex[rhs.size()];
  for (int i=0; i<rhs.size(); i++)
    result[i] = lhs / rhs[i];
    return CArray(rhs.size(), result);
}

ostream&                                                                720
operator<<(ostream& output, CArray& carray){
  char buffer[4096], buf1[128];
  strcpy(buffer, "{ ");
  long mval = carray.size();
  Complex mval1;
  for (int i=0; i<mval; i++){
    mval1 = carray.operator[](i);
    sprintf(buf1, "( %lg, %lg ) ", real(mval1),
              imag(mval1));
    strcat(buffer, buf1);                                               730
  }
  strcat(buffer, "} ");
  return output << buffer;
}
```

```
CArrayGiveAway
CArray::giveaway(){
  CArrayGiveAway rval;
  rval.data = data;
  rval.num_items = num_items;                                          740

  num_items = 0;
  data = NULL;

  return rval;
}


//  This Code predates when I knew about the operator functions....
//CComplex&                                                            750
//CCArray::it(long index){
//        return data[index];
//}}

void
CArray::showme(long i){
  printf(" (%lg, %lg)\n", real(data[i]), imag(data[i]));
}

                                                                       760
void
CArray::gplot(GPlot* gp, char* title, char* xlabel, char* ylabel,
              double offset, double range, long stepoff=0){
  char realdata[256], imagdata[256];
  FILE *rdata, *idata;
  strcpy(realdata, GPUniqueName());
  strcpy(imagdata, GPUniqueName());
  rdata = fopen(realdata, "w");
  idata = fopen(imagdata, "w");
                                                                       770
  //  This writes out the data for plotting. It allows for the data to
  //  be offset shifted (used  in FreqSignal to put the 0 frequency in
  //  the middle of the plot).

  for (long i=0; i<num_items; i++){
    fprintf(rdata, "%lg %lg\n", offset + i * range,
            real(data[(i+stepoff)%num_items]));
    fprintf(idata, "%lg %lg\n", offset + i * range,
            imag(data[(i+stepoff)%num_items]));
  }                                                                    780

  fclose(rdata);
  fclose(idata);
  fprintf( (FILE*) gp, "set title '%s'\n", title);
  fprintf( (FILE*) gp, "set xlabel '%s'\nset ylabel '%s'\n",
           xlabel, ylabel);
  fprintf( (FILE*) gp, "plot '%s' title 'real' with lines,", (char *)realdata);
  fprintf( (FILE*) gp, " '%s' title 'imag' with lines\n",(char *)imagdata);
```

```
 fflush( (FILE*) gp);
 }                                                                    790

double
Maxabs(CArray& c){
   double d = 0.0;
   for (long i=0; i<c.size(); i++)
         if (abs(c[i]) > d) d = abs(c[i]);
   return d;
}
```

C.26 libDsp/include/

The files not listed here are part of the aiff distribution mentioned above.

C.27 libDsp/include/Makefile.in

```
prefix = /usr/local
head   = $(prefix)
incdir = $(head)/include
srcdir = @srcdir@

INSTALL = @INSTALL@
INSTALL_PROGRAM = @INSTALL_PROGRAM@
INSTALL_DATA = @INSTALL_DATA@

HEADERS=*.h

all:

install:
$(INSTALL_DATA) $(HEADERS) $(incdir)

clean:
```

C.28 libDsp/include/CommandLine.h

```
/* FILE: /User/druid/src/c++/ALPHA-libDsp/libDsp/libsrc/Command.cc */
/* AUTHOR: Daniel F. Gruhl <druid@mit.edu> */
/* DATE LAST MODIFIED: Mon Apr 18 14:31:03 EDT 1994 */
/* DESCRIPTION: These function parse command lines in a reasonable way. */


#ifndef _COMMANDLINE_H
```

```
#define _COMMANDLINE_H

//    command_name                                                    10
//                <-switch>
//                <parameter>

#include <stdio.h>
#include <stdarg.h>
#include <iostream.h>
#include <String.h>


                                                                      20
class CommandLine {
   String  command_name;
   String* switches;
   String* s_values;
   String* parameters;
   String  help_msg;
   long    s_count;
   long    p_count;
public:
   CommandLine(int argc, char** argv);                                30
   ~CommandLine();
   int    parse(int argc, char** argv);
   int    switch_set(String s);
   long   how_many_switches(){return s_count;};
   long   how_many_params(){return p_count;};
   String Parameter(long ww);
   String Switch(long ww);
   char*  csParameter(long ww);
   char*  csSwitch(long ww);
   void   help_set(String h_msg){help_msg = h_msg;};                  40
   void   issue_help();
   void   die();
   void   die_on_switch(String flag);
   void   die_if_switch_count_not_between(long low, long high);
   void   die_if_switch_not_one_of(int how_many,...);
   void   die_if_param_count_not_between(long low, long high);
};


#endif                                                                50
```

C.29 libDsp/include/Consts.h

```
// This file may look like C, but it's really -*- C++ -*-
/* FILE: /ti/class/druid/src/c++/libDsp/Consts.h */
```

```
/* AUTHOR: Daniel F. Gruhl <druid@mit.edu> */
/* DATE LAST MODIFIED: Wed Jan  5 08:58:09 EST 1994 */
/* DESCRIPTION: This file provideds constatnts that may not otherwise */
/* be available to the libDsp library. */

// $Id: Consts.h,v 1.1 1994/06/27 15:01:46 druid Exp $

#ifndef PI
#define PI 3.14159265358979323846264338
#endif
```

C.30 libDsp/include/DSPtools.h

```
// This file may look like C, but it's really -*- C++ -*-
/* FILE: /ti/class/druid/src/c++/libDsp/DSPtools.h */
/* AUTHOR: Daniel F. Gruhl <druid@mit.edu> */
/* DATE LAST MODIFIED: Wed Jan  5 08:56:01 EST 1994 */
/* DESCRIPTION: This file infludes the headers you need to run the */
/* libDsp toolkit. For a full description of the functions in this */
/* toolkit, please see the libDsp texi file. */

// $Id: DSPtools.h,v 1.1 1994/06/27 15:01:47 druid Exp $

#ifndef _DSPTOOLS_H
#define _DSPTOOLS_H
#include <math.h>
#include "TimeSignal.h"
#include "FreqSignal.h"
#include "FreqSlice.h"
#include "Consts.h"

TimeSignal ZeroPad(long num, TimeSignal& T);
TimeSignal FullWaveRectifier(TimeSignal& t);
TimeSignal HalfWaveRectifier(TimeSignal& t);

#endif //_DSPTOOLS_H
```

C.31 libDsp/include/Consts.h

```
// This file may look like C, but it's really -*- C++ -*-
/* FILE: /ti/class/druid/src/c++/libDsp/Consts.h */
/* AUTHOR: Daniel F. Gruhl <druid@mit.edu> */
/* DATE LAST MODIFIED: Wed Jan  5 08:58:09 EST 1994 */
/* DESCRIPTION: This file provideds constatnts that may not otherwise */
/* be available to the libDsp library. */
```

```
// $Id: Consts.h,v 1.1 1994/06/27 15:01:46 druid Exp $

#ifndef PI
#define PI 3.14159265358979323846264433
#endif
```

C.32 libDsp/include/Dsp.h

```
// This file may look like C, but it's really -*- C++ -*-
/* FILE: /User/druid/src/c++/libDsp/include/Dsp.h */
/* AUTHOR: Daniel F. Gruhl <druid@mit.edu> */
/* DATE LAST MODIFIED: Tue Feb 1 12:09:05 EST 1994 */
/* DESCRIPTION: This is where all the headers are pulled together for */
/* one easy include in your favorite code... */

// $Id: Dsp.h,v 1.2 1994/07/01 16:19:52 druid Exp $

#ifndef _DSP_H
#define _DSP_H

#ifdef NEEDS_GNU_COMPLEX
#include "GnuExtras/Complex.h"
#else
#include <Complex.h>
#endif
#include "CArray.h"
#include "TimeSignal.h"
#include "FreqSignal.h"
#include "Consts.h"
#include "FFT.h"
#include "SigGen.h"
#include "FreqSlice.h"
#include "DSPtools.h"
#include "GPlot.h"
#include "Slice.h"
#include "aifif.h"
#include "Filters.h"
#include "CommandLine.h"
#include "DblVectorAVec.h"
#include "DblVectorVec.h"
#endif
```

C.33 libDsp/include/FFT.h

```
// This file may look like C, but it's really -*- C++ -*-
```

```
/* FILE: /User/druid/src/c++/libDsp/include/FFT.h */
/* AUTHOR: Daniel F. Gruhl <druid@mit.edu> */
/* DATE LAST MODIFIED: Tue Feb  1 12:10:08 EST 1994 */
/* DESCRIPTION: This is the header file for the FFT, IFFT, DFT and */
/* IDFT routines */

// $Id: FFT.h,v 1.1 1994/06/27 15:01:52 druid Exp $

#ifndef _FFT_H
#define _FFT_H

#include "Dsp.h"
#include <math.h>

long DSPFlipAddress(int bits, long number);
FreqSignal DSPPower_of_two_FFT(TimeSignal& t);
void four1(float data[], unsigned long nn, int isign);
inline Complex DSPTW(long super, long sub);
FreqSignal DSPDFT(TimeSignal& t);
FreqSignal DSPFFT(TimeSignal& t);
FreqSignal DSPFT(TimeSignal &t);
TimeSignal DSPIFT(FreqSignal& f, double base);
TimeSignal DSPIFFT(FreqSignal& f, double base);
TimeSignal DSPIDFT(FreqSignal& f, double base);
#endif // _FFT_H
```

C.34 libDsp/include/Filters.h

```
#ifndef _FILTERS_H
#define _FILTERS_H

#include "Dsp.h"

typedef struct {
   long an;
   Complex *a;
   long bn;
   Complex *b;
} FilterKernal;

// This is for an FIR or IIR filter
TimeSignal Filter(TimeSignal& t, FilterKernal k);

TimeSignal ApplyFIR(TimeSignal& t, TimeSignal& filt);
TimeSignal DSPWindowBartlett(TimeSignal& t);
TimeSignal DSPWindowHanning(TimeSignal& t);
TimeSignal DSPWindowHamming(TimeSignal& t);
#endif
```

C.35 libDsp/include/FreqSignal.h

```
// This may look like C code, but it's really -*- C++ -*-
/* FILE: /ti/class/druid/src/c++/libDsp/FreqSignal.h */
/* AUTHOR: Daniel F. Gruhl <druid@mit.edu> */
/* DATE LAST MODIFIED: Wed Jan 5 08:53:36 EST 1994 */
/* DESCRIPTION: This file contains the header information for the */
/* Frequency Signal data type. Please see the libDsp texi file for */
/* full documentation of this class. */

// $Id: FreqSignal.h,v 1.1 1994/06/27 15:01:54 druid Exp $

#ifndef _FREQSIGNAL_H
#define _FREQSIGNAL_H

#include "CArray.h"
#include "Consts.h"

class FreqSignal : public CArray {
  double Freq_Per_Bin;
 public:
  FreqSignal(long size);
  FreqSignal(long size, Complex* data);
  FreqSignal(long size, Complex* data, double FPB);
  FreqSignal(CArrayGiveAway& c);
  FreqSignal();
  double frequency(long bin);
  double freq_per_bin(){return Freq_Per_Bin;};
  void data_out(char* filename);
  void resize(long size);
  void gplot(GPlot *g, char* title);

  FreqSignal& operator=(FreqSignal& rhs);
  FreqSignal& operator=(CArray& rhs);
  FreqSignal& operator=(CArrayGiveAway& rhs);
  friend FreqSignal operator+(FreqSignal& lhs, FreqSignal& rhs);
  friend FreqSignal operator+(FreqSignal& lhs, int& rhs);
  friend FreqSignal operator+(FreqSignal& lhs, float& rhs);
  friend FreqSignal operator+(FreqSignal& lhs, double& rhs);
  friend FreqSignal operator+(FreqSignal& lhs, Complex& rhs);
  friend FreqSignal operator+(int& lhs, FreqSignal& rhs);
  friend FreqSignal operator+(float& lhs, FreqSignal& rhs);
  friend FreqSignal operator+(double& lhs, FreqSignal& rhs);
  friend FreqSignal operator+(Complex& lhs, FreqSignal& rhs);

  friend FreqSignal operator-(FreqSignal& lhs, FreqSignal& rhs);
  friend FreqSignal operator-(FreqSignal& lhs, int& rhs);
  friend FreqSignal operator-(FreqSignal& lhs, float& rhs);
  friend FreqSignal operator-(FreqSignal& lhs, double& rhs);
  friend FreqSignal operator-(FreqSignal& lhs, Complex& rhs);
```

```
  friend FreqSignal operator-(int& lhs, FreqSignal& rhs);
  friend FreqSignal operator-(float& lhs, FreqSignal& rhs);
  friend FreqSignal operator-(double& lhs, FreqSignal& rhs);
  friend FreqSignal operator-(Complex& lhs, FreqSignal& rhs);

  friend FreqSignal operator*(FreqSignal& lhs, FreqSignal& rhs);
  friend FreqSignal operator*(FreqSignal& lhs, int& rhs);
  friend FreqSignal operator*(FreqSignal& lhs, float& rhs);
  friend FreqSignal operator*(FreqSignal& lhs, double& rhs);
  friend FreqSignal operator*(FreqSignal& lhs, Complex& rhs);
  friend FreqSignal operator*(int& lhs, FreqSignal& rhs);
  friend FreqSignal operator*(float& lhs, FreqSignal& rhs);
  friend FreqSignal operator*(double& lhs, FreqSignal& rhs);
  friend FreqSignal operator*(Complex& lhs, FreqSignal& rhs);

  friend FreqSignal operator/(FreqSignal& lhs, FreqSignal& rhs);
  friend FreqSignal operator/(FreqSignal& lhs, int& rhs);
  friend FreqSignal operator/(FreqSignal& lhs, float& rhs);
  friend FreqSignal operator/(FreqSignal& lhs, double& rhs);
  friend FreqSignal operator/(FreqSignal& lhs, Complex& rhs);
  friend FreqSignal operator/(int& lhs, FreqSignal& rhs);
  friend FreqSignal operator/(float& lhs, FreqSignal& rhs);
  friend FreqSignal operator/(double& lhs, FreqSignal& rhs);
  friend FreqSignal operator/(Complex& lhs, FreqSignal& rhs);

   FreqSignal& operator+=(FreqSignal& rhs);
   FreqSignal& operator+=(int& rhs);
   FreqSignal& operator+=(float& rhs);
   FreqSignal& operator+=(double& rhs);
   FreqSignal& operator+=(Complex& rhs);

   FreqSignal& operator-=(FreqSignal& rhs);
   FreqSignal& operator-=(int& rhs);
   FreqSignal& operator-=(float& rhs);
   FreqSignal& operator-=(double& rhs);
   FreqSignal& operator-=(Complex& rhs);

   FreqSignal& operator*=(FreqSignal& rhs);
   FreqSignal& operator*=(int& rhs);
   FreqSignal& operator*=(float& rhs);
   FreqSignal& operator*=(double& rhs);
   FreqSignal& operator*=(Complex& rhs);

   FreqSignal& operator/=(FreqSignal& rhs);
   FreqSignal& operator/=(int& rhs);
   FreqSignal& operator/=(float& rhs);
   FreqSignal& operator/=(double& rhs);
   FreqSignal& operator/=(Complex& rhs);

};

FreqSignal operator+(FreqSignal& lhs, FreqSignal& rhs);
FreqSignal operator+(FreqSignal& lhs, int& rhs);
FreqSignal operator+(FreqSignal& lhs, float& rhs);
```

```
FreqSignal operator+(FreqSignal& lhs, double& rhs);
FreqSignal operator+(FreqSignal& lhs, Complex& rhs);
FreqSignal operator+(int& lhs, FreqSignal& rhs);
FreqSignal operator+(float& lhs, FreqSignal& rhs);
FreqSignal operator+(double& lhs, FreqSignal& rhs);
FreqSignal operator+(Complex& lhs, FreqSignal& rhs);

FreqSignal operator-(FreqSignal& lhs, FreqSignal& rhs);                    110
FreqSignal operator-(FreqSignal& lhs, int& rhs);
FreqSignal operator-(FreqSignal& lhs, float& rhs);
FreqSignal operator-(FreqSignal& lhs, double& rhs);
FreqSignal operator-(FreqSignal& lhs, Complex& rhs);
FreqSignal operator-(int& lhs, FreqSignal& rhs);
FreqSignal operator-(float& lhs, FreqSignal& rhs);
FreqSignal operator-(double& lhs, FreqSignal& rhs);
FreqSignal operator-(Complex& lhs, FreqSignal& rhs);

FreqSignal operator*(FreqSignal& lhs, FreqSignal& rhs);                    120
FreqSignal operator*(FreqSignal& lhs, int& rhs);
FreqSignal operator*(FreqSignal& lhs, float& rhs);
FreqSignal operator*(FreqSignal& lhs, double& rhs);
FreqSignal operator*(FreqSignal& lhs, Complex& rhs);
FreqSignal operator*(int& lhs, FreqSignal& rhs);
FreqSignal operator*(float& lhs, FreqSignal& rhs);
FreqSignal operator*(double& lhs, FreqSignal& rhs);
FreqSignal operator*(Complex& lhs, FreqSignal& rhs);

FreqSignal operator/(FreqSignal& lhs, FreqSignal& rhs);                    130
FreqSignal operator/(FreqSignal& lhs, int& rhs);
FreqSignal operator/(FreqSignal& lhs, float& rhs);
FreqSignal operator/(FreqSignal& lhs, double& rhs);
FreqSignal operator/(FreqSignal& lhs, Complex& rhs);
FreqSignal operator/(int& lhs, FreqSignal& rhs);
FreqSignal operator/(float& lhs, FreqSignal& rhs);
FreqSignal operator/(double& lhs, FreqSignal& rhs);
FreqSignal operator/(Complex& lhs, FreqSignal& rhs);

#endif //_FREQSIGNAL_H                                                     140
```

C.36 libDsp/include/FreqSlice.h

```
// This may look like C code but it's really -*- C++ -*-

// Nomenclature:
//   windows - each slice of time spectrum converted to spectrum is a
//         window

#ifndef _FREQSLICE_H
#define _FREQSLICE_H
#include "FreqSignal.h"
#include <String.h>                                                        10
#include "GPlot.h"
```

```
class FreqSlice {
   long windows;
   FreqSignal *data;
   double deltatime;
public:
   long   Windows()    {return windows;};
   long   WindowSize() {return data[0].size();};
   double DeltaTime()  {return deltatime;};
   void set_delta_time(double dt){deltatime = dt;};

   FreqSlice(long Windows=1, long samples_per_window=1);
   FreqSlice(long Windows, FreqSignal*data, double deltatime);
   ~FreqSlice();

   FreqSignal& operator[](long index);
   void gplot(GPlot* g, char* plot_title, long every=1);
   double time_of_window(long window){return deltatime * window;};
   FreqSlice& operator=( FreqSlice& f);
};

double maxabs(FreqSlice& f);

#endif
```

C.37 libDsp/include/GPlot.h

```
#ifndef _GPLOT_H
#define _GPLOT_H

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>

// $Id: GPlot.h,v 1.1 1994/06/27 15:01:56 druid Exp $

typedef FILE GPlot;

GPlot* GPopen();
GPlot* GPfopen(char* filename);
GPlot* GPclose(GPlot* gp);
int GPPlotFile(GPlot* gp, char* filename, char* title,
               char* xaxis, char* yaxis, char* data, char* plottype);
char* GPUniqueName();
void pause();
#endif
```

C.38 libDsp/include/SigGen.h

```
// This may look like C code, but it's really -*- C++ -*-

#ifndef _SIGGEN_H
#define _SIGGEN_H

// $Id: SigGen.h,v 1.1 1994/06/27 15:01:59 druid Exp $

#include "Dsp.h"
TimeSignal DSPSquareWave(double sampling_rate, double duration, double freq);
TimeSignal DSPTriangleWave(double sampling_rate, double duration, double freq);
TimeSignal DSPSinSignal(double sampling_rate, double duration, double freq);
TimeSignal DSPCosSignal(double sampling_rate, double duration, double freq);

inline TimeSignal
DSPSinWave(double sampling_rate, double duration, double freq){
   return DSPSinSignal(sampling_rate, duration, freq);
}

inline TimeSignal
DSPCosWave(double sampling_rate, double duration, double freq){
   return DSPCosSignal(sampling_rate, duration, freq);
}

#endif // _SIGGEN_H
```

C.39 libDsp/include/Slice.h

```
// This may look like C code, but it's really -*- C++ -*-

// $Id: Slice.h,v 1.1 1994/06/27 15:02:00 druid Exp $

#ifndef _SLICE_H
#define _SLICE_H
#include "Dsp.h"
FreqSlice  Slice(TimeSignal& T, int num);
TimeSignal UnSlice(FreqSlice& F);
#endif
```

C.40 libDsp/include/TimeSignal.h

```
// This file may look like C code, but its really -*- C++ -*-
/* FILE: /ti/class/druid/src/c++/libDsp/TimeSignal.h */
/* AUTHOR: Daniel F. Gruhl <druid@mit.edu> */
/* DATE LAST MODIFIED: Wed Jan 5 09:06:46 EST 1994 */
/* DESCRIPTION: This is the time signal data type. It would be most */
/* often read in from an external sound file. For a full description */
/* of the Class, please see the libDsp texi document. */


// $Id: TimeSignal.h,v 1.4 1994/07/25 13:07:15 druid Exp $

#ifndef _TIMESIGNAL_H
#define _TIMESIGNAL_H

#ifdef NEEDS_GNU_COMPLEX
#include "GnuExtras/Complex.h"
#else
#include <Complex.h>
#endif
#include "CArray.h"

typedef struct {
    double maxpos;
    double maxneg;
} RangeVal;

typedef struct {
    short* data;
    long num;
} Quantized;

class TimeSignal : public CArray {
    double TimeBase;
public:
    TimeSignal(long size);
    TimeSignal(double base, long size);
    TimeSignal(double base=1, double time=1);
    TimeSignal(double base, long size, Complex* data);
    TimeSignal(double base, CArray c);
    TimeSignal(double base, CArrayGiveAway& c);
    ~TimeSignal();

    void   set_time_base(double tb){TimeBase = tb;};
    double time_base(){return TimeBase;};
    void   data_out(char* filename);
    double duration(){return size()/TimeBase;};
    void   gplot     (GPlot *g, char* title);
    int    readAiff  (char* filename);
    int    writeAiff (char* filename);
```

```
    void   resize    (long newsize);
    void   stretch   (long newsize);
    RangeVal   maxposneg ();
    Quantized  QData     ();
    TimeSignal delay     (double seconds);
    TimeSignal delay     (long   samples);
    void       normalize ();
//  TimeSignal slice(long first, long last);

    TimeSignal& operator=(TimeSignal& rhs);
    TimeSignal& operator=(CArray& rhs);
    friend TimeSignal operator+(TimeSignal& lhs, TimeSignal& rhs);
    friend TimeSignal operator+(TimeSignal& lhs, int& rhs);
    friend TimeSignal operator+(TimeSignal& lhs, float& rhs);
    friend TimeSignal operator+(TimeSignal& lhs, double& rhs);
    friend TimeSignal operator+(TimeSignal& lhs, Complex& rhs);
    friend TimeSignal operator+(int& lhs, TimeSignal& rhs);
    friend TimeSignal operator+(float& lhs, TimeSignal& rhs);
    friend TimeSignal operator+(double& lhs, TimeSignal& rhs);
    friend TimeSignal operator+(Complex& lhs, TimeSignal& rhs);

    friend TimeSignal operator-(TimeSignal& lhs, TimeSignal& rhs);
    friend TimeSignal operator-(TimeSignal& lhs, int& rhs);
    friend TimeSignal operator-(TimeSignal& lhs, float& rhs);
    friend TimeSignal operator-(TimeSignal& lhs, double& rhs);
    friend TimeSignal operator-(TimeSignal& lhs, Complex& rhs);
    friend TimeSignal operator-(int& lhs, TimeSignal& rhs);
    friend TimeSignal operator-(float& lhs, TimeSignal& rhs);
    friend TimeSignal operator-(double& lhs, TimeSignal& rhs);
    friend TimeSignal operator-(Complex& lhs, TimeSignal& rhs);

    friend TimeSignal operator*(TimeSignal& lhs, TimeSignal& rhs);
    friend TimeSignal operator*(TimeSignal& lhs, int& rhs);
    friend TimeSignal operator*(TimeSignal& lhs, float& rhs);
    friend TimeSignal operator*(TimeSignal& lhs, double& rhs);
    friend TimeSignal operator*(TimeSignal& lhs, Complex& rhs);
    friend TimeSignal operator*(int& lhs, TimeSignal& rhs);
    friend TimeSignal operator*(float& lhs, TimeSignal& rhs);
    friend TimeSignal operator*(double& lhs, TimeSignal& rhs);
    friend TimeSignal operator*(Complex& lhs, TimeSignal& rhs);

    friend TimeSignal operator/(TimeSignal& lhs, TimeSignal& rhs);
    friend TimeSignal operator/(TimeSignal& lhs, int& rhs);
    friend TimeSignal operator/(TimeSignal& lhs, float& rhs);
    friend TimeSignal operator/(TimeSignal& lhs, double& rhs);
    friend TimeSignal operator/(TimeSignal& lhs, Complex& rhs);
    friend TimeSignal operator/(int& lhs, TimeSignal& rhs);
    friend TimeSignal operator/(float& lhs, TimeSignal& rhs);
    friend TimeSignal operator/(double& lhs, TimeSignal& rhs);
    friend TimeSignal operator/(Complex& lhs, TimeSignal& rhs);

    TimeSignal& operator+=(TimeSignal& rhs);
    TimeSignal& operator+=(int& rhs);
    TimeSignal& operator+=(float& rhs);
```

```
    TimeSignal& operator+=(double& rhs);
    TimeSignal& operator+=(Complex& rhs);

    TimeSignal& operator-=(TimeSignal& rhs);
    TimeSignal& operator-=(int& rhs);
    TimeSignal& operator-=(float& rhs);
    TimeSignal& operator-=(double& rhs);                                    110
    TimeSignal& operator-=(Complex& rhs);

    TimeSignal& operator*=(TimeSignal& rhs);
    TimeSignal& operator*=(int& rhs);
    TimeSignal& operator*=(float& rhs);
    TimeSignal& operator*=(double& rhs);
    TimeSignal& operator*=(Complex& rhs);

    TimeSignal& operator/=(TimeSignal& rhs);
    TimeSignal& operator/=(int& rhs);                                       120
    TimeSignal& operator/=(float& rhs);
    TimeSignal& operator/=(double& rhs);
    TimeSignal& operator/=(Complex& rhs);

};

TimeSignal operator+(TimeSignal& lhs, TimeSignal& rhs);
TimeSignal operator+(TimeSignal& lhs, int& rhs);
TimeSignal operator+(TimeSignal& lhs, float& rhs);
TimeSignal operator+(TimeSignal& lhs, double& rhs);                         130
TimeSignal operator+(TimeSignal& lhs, Complex& rhs);
TimeSignal operator+(int& lhs, TimeSignal& rhs);
TimeSignal operator+(float& lhs, TimeSignal& rhs);
TimeSignal operator+(double& lhs, TimeSignal& rhs);
TimeSignal operator+(Complex& lhs, TimeSignal& rhs);

TimeSignal operator-(TimeSignal& lhs, TimeSignal& rhs);
TimeSignal operator-(TimeSignal& lhs, int& rhs);
TimeSignal operator-(TimeSignal& lhs, float& rhs);
TimeSignal operator-(TimeSignal& lhs, double& rhs);                         140
TimeSignal operator-(TimeSignal& lhs, Complex& rhs);
TimeSignal operator-(int& lhs, TimeSignal& rhs);
TimeSignal operator-(float& lhs, TimeSignal& rhs);
TimeSignal operator-(double& lhs, TimeSignal& rhs);
TimeSignal operator-(Complex& lhs, TimeSignal& rhs);

TimeSignal operator*(TimeSignal& lhs, TimeSignal& rhs);
TimeSignal operator*(TimeSignal& lhs, int& rhs);
TimeSignal operator*(TimeSignal& lhs, float& rhs);
TimeSignal operator*(TimeSignal& lhs, double& rhs);                         150
TimeSignal operator*(TimeSignal& lhs, Complex& rhs);
TimeSignal operator*(int& lhs, TimeSignal& rhs);
TimeSignal operator*(float& lhs, TimeSignal& rhs);
TimeSignal operator*(double& lhs, TimeSignal& rhs);
TimeSignal operator*(Complex& lhs, TimeSignal& rhs);

TimeSignal operator/(TimeSignal& lhs, TimeSignal& rhs);
```

```
TimeSignal operator/(TimeSignal& lhs, int& rhs);
TimeSignal operator/(TimeSignal& lhs, float& rhs);
TimeSignal operator/(TimeSignal& lhs, double& rhs);
TimeSignal operator/(TimeSignal& lhs, Complex& rhs);
TimeSignal operator/(int& lhs, TimeSignal& rhs);
TimeSignal operator/(float& lhs, TimeSignal& rhs);
TimeSignal operator/(double& lhs, TimeSignal& rhs);
TimeSignal operator/(Complex& lhs, TimeSignal& rhs);

#endif //_TIMESIGNAL_H
```

C.41 libDsp/include/UniqueName.h

```
#ifndef _UNIQUE_NAME_H
#define _UNIQUE_NAME_H

#include <stdlib.h>
#include <iostream.h>
#include <string.h>

class UniqueName {
   /* static */ unsigned int max_snum;
   unsigned int snum;
   unsigned int ref_num;
 public:
   UniqueName();
   char* tmp_file_name(char* header);
   int number();
   char* name();


};

#endif
```

C.42 libDsp/include/CArray.h

```
// This may look like C code, but it is really -*- C++ -*-
/* FILE: /ti/class/druid/src/c++/libDsp/CArray.h */
/* AUTHOR: Daniel F. Gruhl <druid@mit.edu> */
/* DATE LAST MODIFIED: Wed Jan 5 08:58:53 EST 1994 */
/* DESCRIPTION: This file contains the header information for the */
/* Complex Array data type. For a full discussion of this Class, */
/* please see the libDsp.texi file. */
```

```
// $Id: CArray.h,v 1.2 1994/07/01 16:19:50 druid Exp $

#ifndef _CARRAY_H
#define _CARRAY_H

/* Includes that will be needed in the following code */
#ifdef NEED_GNU_COMPLEX
#include "GnuExtras/Complex.h"
#else
#include <Complex.h>
#endif
#include <math.h>
#include "GPlot.h"

// This construct exists to allow quick passing of data from one
// CArray to another, by simply passing the data array as a pointer,
// rather than copying an entire array.

typedef struct {
  Complex* data;
  long     num_items;
} CArrayGiveAway;


// This is the actual class definition for CArray

class CArray {
 private:
  Complex* data;        // Pointer to the array of complex numbers
                        // stored in the CArray
  long num_items;       // The number of items in the CArray
 public:
  // Make and Destroy
  CArray(long items = 1);         // Serves as a default constructor,
                                  // as well as one to use if you just
                                  // know the size of the array you need.
  CArray(long size, Complex* Data){data = Data; num_items=size;};
                          // Used most often when a CArray is
                                  // being created in a return
                                  // statement. An inline function.
  CArray(CArrayGiveAway& c){data = c.data; num_items=c.num_items;};
                          // An inline to take a CArray.
  virtual ~CArray();              // The destructor. It mainly frees
                                  // the data.
  // Fun with sizing
    virtual void Dealocate();
  virtual long setsize(long newsize);
  virtual long size();                // How many elements are in the array?
  virtual void resize(long newsize);     // Trash the data array and make a
                                  // new one of the requested size.
virtual  void stretch(long newsize);    // Trash the data array but copy as
                                  // much of it as you can into the
                                  // new array.
```

```
virtual CArray slice(long first, long last); // Return a CArray which is a
                                    // slice of the current array
                                    // between first and last, inclusive.
virtual void showme(long i);          // Exclusivly a debugging option
                                    // which allows you to look at
                                    // alement i in the data array.
virtual CArrayGiveAway giveaway();    // Give away the current data.
                                    // Relinquish your pointer to it.
virtual void gplot(GPlot* g, char* title, char* xlabel, char* ylabel,
        double offset, double delta, long stepoff=0);
                    // Plot the CArray: g is the gnuplot
                                    // to plot into, title, xlabel and
                                    // ylabel are the graph title, etc.
                                    // offset is the first element
                                    // coresponds to. delta is what the
                                    // increment is for each following element.

virtual  double* AngleArray(double* mdata);
  // Return a DblVector of the angular
                                    // component at each point
virtual  double* MagnitudeArray(double* mdata);
 // Return a DblVector of the
                                    // magnitude at each point

  // The following material is for TESTING only. Expect it to
  // disappear at ANY time....
  long whatdata(){return (long) data;};
  // End of testing

 // Operators --- The following are the operators that are used to
 // mathimatically manipulate CArrays and the objects that inherit
 // from them.
  // Operator []
virtual Complex& operator[](long index);
  // Operator =
 CArray& operator=( int& rhs);
 CArray& operator=( float& rhs);
 CArray& operator=( double& rhs);
 CArray& operator=( Complex& rhs);
 CArray& operator=( CArray& rhs);
 CArray& operator=( CArrayGiveAway& c);
  // Operator +
 CArray& operator+=( CArray& rhs);
 CArray& operator+=( int& rhs);
 CArray& operator+=( float& rhs);
 CArray& operator+=( double& rhs);
 CArray& operator+=( Complex& rhs);
  // Operator -=
 CArray& operator-=(CArray& rhs);
 CArray& operator-=( int& rhs);
 CArray& operator-=( float& rhs);
 CArray& operator-=( double& rhs);
 CArray& operator-=( Complex& rhs);
  // Operator *=
```

```
CArray& operator*=(CArray& rhs);
CArray& operator*=( int& rhs);
CArray& operator*=( float& rhs);
CArray& operator*=( double& rhs);
CArray& operator*=( Complex& rhs);                                    120
 //  Operator /=
CArray& operator/=(CArray& rhs);
CArray& operator/=( int& rhs);
CArray& operator/=( float& rhs);
CArray& operator/=( double& rhs);
CArray& operator/=( Complex& rhs);

//  Binops - All of these are declared as friends for obvious
//  implementation reasons (so you can say  a + b rather than
//  a.operator+(b) )                                                  130

//  Plus Operators
friend CArray operator+(CArray& lhs, CArray& rhs);

friend CArray operator+(CArray& lhs, int& rhs);
friend CArray operator+(CArray& lhs, float& rhs);
friend CArray operator+(CArray& lhs, double& rhs);
friend CArray operator+(CArray& lhs, Complex& rhs);

friend CArray operator+(int& lhs, CArray& rhs);                       140
friend CArray operator+(float& lhs, CArray& rhs);
friend CArray operator+(double& lhs, CArray& rhs);
friend CArray operator+(Complex& lhs, CArray& rhs);

//  Minus Operators
friend CArray operator-(CArray& lhs, CArray& rhs);

friend CArray operator-(CArray& lhs, int& rhs);
friend CArray operator-(CArray& lhs, float& rhs);
friend CArray operator-(CArray& lhs, double& rhs);                    150
friend CArray operator-(CArray& lhs, Complex& rhs);

friend CArray operator-(int& lhs, CArray& rhs);
friend CArray operator-(float& lhs, CArray& rhs);
friend CArray operator-(double& lhs, CArray& rhs);
friend CArray operator-(Complex& lhs, CArray& rhs);

//  Multiply Operators
friend CArray operator*(CArray& lhs, CArray& rhs);
                                                                      160
friend CArray operator*(CArray& lhs, int& rhs);
friend CArray operator*(CArray& lhs, float& rhs);
friend CArray operator*(CArray& lhs, double& rhs);
friend CArray operator*(CArray& lhs, Complex& rhs);

friend CArray operator*(int& lhs, CArray& rhs);
friend CArray operator*(float& lhs, CArray& rhs);
friend CArray operator*(double& lhs, CArray& rhs);
friend CArray operator*(Complex& lhs, CArray& rhs);
```

```
                                                                    170
  // Divide Operators
  friend CArray operator/(CArray& lhs, CArray& rhs);

  friend CArray operator/(CArray& lhs, int& rhs);
  friend CArray operator/(CArray& lhs, float& rhs);
  friend CArray operator/(CArray& lhs, double& rhs);
  friend CArray operator/(CArray& lhs, Complex& rhs);

  friend CArray operator/(int& lhs, CArray& rhs);
  friend CArray operator/(float& lhs, CArray& rhs);                 180
  friend CArray operator/(double& lhs, CArray& rhs);
  friend CArray operator/(Complex& lhs, CArray& rhs);
};

ostream& operator<<(ostream& output, CArray& carray);

CArray operator+(CArray& lhs, CArray& rhs);
CArray operator+(CArray& lhs, int& rhs);
CArray operator+(CArray& lhs, float& rhs);
CArray operator+(CArray& lhs, double& rhs);                         190
CArray operator+(CArray& lhs, Complex& rhs);

CArray operator+(int& lhs, CArray& rhs);
CArray operator+(float& lhs, CArray& rhs);
CArray operator+(double& lhs, CArray& rhs);
CArray operator+(Complex& lhs, CArray& rhs);

CArray operator-(CArray& lhs, CArray& rhs);

CArray operator-(CArray& lhs, int& rhs);                            200
CArray operator-(CArray& lhs, float& rhs);
CArray operator-(CArray& lhs, double& rhs);
CArray operator-(CArray& lhs, Complex& rhs);

CArray operator-(int& lhs, CArray& rhs);
CArray operator-(float& lhs, CArray& rhs);
CArray operator-(double& lhs, CArray& rhs);
CArray operator-(Complex& lhs, CArray& rhs);

CArray operator*(CArray& lhs, CArray& rhs);                         210

CArray operator*(CArray& lhs, int& rhs);
CArray operator*(CArray& lhs, float& rhs);
CArray operator*(CArray& lhs, double& rhs);
CArray operator*(CArray& lhs, Complex& rhs);

CArray operator*(int& lhs, CArray& rhs);
CArray operator*(float& lhs, CArray& rhs);
CArray operator*(double& lhs, CArray& rhs);
CArray operator*(Complex& lhs, CArray& rhs);                        220

CArray operator/(CArray& lhs, CArray& rhs);
```

```
CArray operator/(CArray& lhs, int& rhs);
CArray operator/(CArray& lhs, float& rhs);
CArray operator/(CArray& lhs, double& rhs);
CArray operator/(CArray& lhs, Complex& rhs);

CArray operator/(int& lhs, CArray& rhs);
CArray operator/(float& lhs, CArray& rhs);                                    230
CArray operator/(double& lhs, CArray& rhs);
CArray operator/(Complex& lhs, CArray& rhs);

double Maxabs(CArray& c);
#endif //_CARRAY_H
```

240

C.43 libDsp/examples/

These are the example discussed in the users manual.

Bibliography

[1] Gail Anderson and Paul Anderson. *The UNIX C Shell Field Guide.* Prentice Hall, 1986.

[2] Borland. *Borland C++ ver 2.0 Library Reference*, 1991.

[3] Stephan C. Dewhurst and Kathy T. Stark. *Programming in C++.* Prentice Hall, 1989.

[4] Samuel P. Harbison and Guy L. Steele Jr. *C A Refrence Manual.* Prentice Hall, 1991.

[5] Al Kelley and Ira Pohl. *A Book on C.* The Benjamin/Cummings Publishing Company, Inc., 1990.

[6] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc.* O'Reilly & Associates, Inc., 1990.

[7] Savid MacKenzie, Roland McGrath, and Noah Friedman. *Autoconf*, 1994.

[8] Scott Meyers. *Effective C++.* Addison-Wesley Publishing Company, Inc., 1992.

[9] Alan V. Oppenheim and Ronald W. Schafer. *Discrete-Time Signal Processing.* Prentice Hall, 1989.

[10] Andrew Oram and Steve Talbott. *Managing Projects with make.* O'Reilly & Associates, Inc., 1991.

[11] Chet Ramey. *Bash - The GNU shell.*

[12] Steve Teale. *C++ IO Streams Handbook.* Addison-Wesley Publishing Company, 1993.

[13] Sandra Loosemore with Roland McGrath, Andrew Oram, and Richard M. Stallman. *The GNU C Library Reference Manual*, 1993.