MANAGING SOFTWARE COMPLEXITY IN SIGNAL PROCESSING RESEARCH

Joseph Picone

Texas Instruments - Tsukuba Research and Development Center
17 Miyukigaoka, Tsukuba, Ibaraki 305 Japan
picone@trdc.ti.com

ABSTRACT

The complexity of signal processing algorithms, particularly stochastic pattern matching algorithms, has been growing exponentially in recent years. Research in signal processing is largely limited by software development time and suffers from a creative backlog. Often, point solutions are pursued, and the percentage of reusable code is low. In this paper, we describe a large hierarchical software environment, based on C++, developed to support basic research into signal processing. There are two cornerstones of the environment: the signal object file and the math classes. From these, we have built many useful higher level abstractions, including device independent audio and modular speech recognition systems. This environment heavily embraces object-oriented programming philosophies and structured programming techniques. To date, it is one of the largest such environments implemented entirely in C++.

1. INTRODUCTION

The complexity of signal processing algorithms, particularly stochastic pattern matching algorithms, has been growing exponentially in recent years, as shown in Fig.1. While disk space, memory, and CPU speed have been doubling almost every year, software complexity (measured in lines of code for lack of a better measure) is growing even faster. Applications such as a speech recognizer have grown from several thousand lines of code in the 1970's to several hundred thousand lines of code in the last five years. If we add information stored in data-driven formats, such as speech recognition acoustic and language models and application domain descriptions, the rate of growth is even more astounding. Software complexity IS the dominant problem today.

Research and experimentation in signal processing is now largely limited by the time it takes to implement complex algorithms. The majority of the time required to implement today's complex algorithms is spent on two types of programming: input/output (data-driven programming), and data structure manipulation (data structure programming). Most often, point solutions are pursued, resulting in a significant drop in the percentage of reusable code. In an effort to create a better interface between complicated data structures and functions that operate on these structures, our interest turned to object-oriented languages such as C++ in the late 1980's. This paper is not intended to be a tutorial on C++ [1], but rather a discussion of how we have used this language to solve some basic problems in signal processing software engineering.

2. AN OVERVIEW OF THE CLASS HIERARCHY

One of our goals at Texas Instruments in speech research has been to institute a group-wide software development strategy. However, the environment created from our rather informal implementation of this strategy in

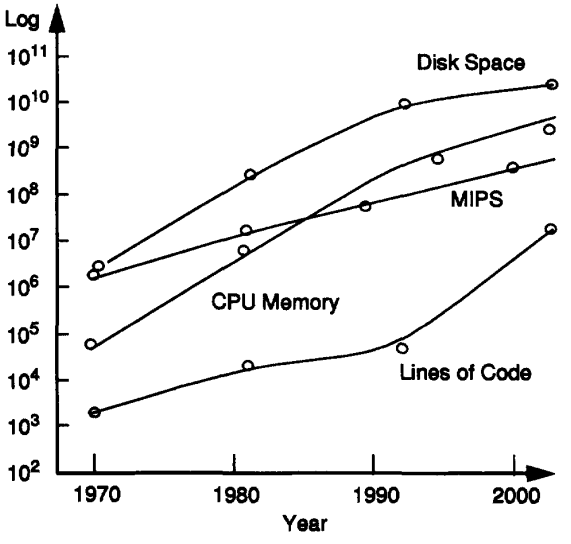

Figure 1 Some reflections on the growth in complexity of signal processing software. While disk space and CPU for one-dimensional signal processing have become plentiful, methodologies for managing large software environments are relatively undeveloped. Managing relationships between data and functions are now very important.

0-7803-0946-4/93 $3.00

its totality had evolved to many point solutions with very little underlying structure at the programming level. Several years ago, we started an initiative to redefine our environment. It became obvious as we progressed that the C programming language in itself would not be sufficient to manage the complexity of the environment. Hence, we embraced an object-oriented programming paradigm based on C++ (at a very early stage in C++).

The environment we describe in this paper has been developed entirely in C++ for general signal processing research. It has been successfully applied to topics ranging from finite precision effects in digital audio [3] to sophisticated speech recognition algorithms [4]. This paper is NOT intended to advocate one particular programming environment, however. Instead, it is about the invaluable experiences we have learned in understanding the deficiencies of modern day programming languages, and about some modest successes we have had in simplifying the complexity of our software. Our goal in this work has been to achieve an environment that does not constrain the individual programmer, and requires a minimum of *a priori* knowledge. C++ represents a significant step forward in this respect.

Of course there are many approaches one can take to building a good structured programming environment. One of our most important criterion was a desire to have a rich programming environment rather than a fancy interactive programming tool. In our experience, interactive program tools (often referred to as CASE tools) are simply not used by researchers doing state of the art algorithm development. We seek an environment that is rich in low-level software that can be easily pieced together to form complex applications, and an environment that does not have a steep learning curve.

A second important decision point was the choice of a compiled language, such as C++, versus an interpretive one, such as LISP. Major factors affecting this choice were: applications such as speech recognition are still computationally intensive; compiled languages tend to be more efficient than interpretive ones; memory management is still a major issue; the industry trend is toward C++. In this sense, our approach is quite different than that taken in [5], or those being considered with other languages such as Smalltalk. We also chose to follow an industry trend and adopted AT&T C++ as our programming language. We have recently converted to the public domain language supported by the GNU foundation, g++ (it is not productive to debate the rationale behind this decision here).

There are numerous other factors that contributed to our decision to adopt C++. One decision that cannot be under-emphasized (and learned from experience) is the use of a strongly-typed language. Normally, in speech processing, we like to read data from files, and process it accordingly (data-driven programming). For example, in many applications we would like to transparently process sampled data or "feature data" depending on the data in the file. This is difficult to implement cleanly in any strongly-typed language, because, in general, you need to know the type of something before you can process it. Languages such as LISP, of course, excel at doing this, because they allow run-time type-checking. It has been our (painful) experience that a strongly-typed language, as advertised [1], drastically reduces debugging time. In hindsight, we cannot imagine implementing an environment with thousands of objects with a weakly typed language.

The general architecture, or class hierarchy, of our environment is show below in Fig. 2. The strict hierarchical relationships between classes is important in preserving a well-organized environment and a simple I/O interface. The Sof class, which coordinates file I/O, is at the bottom of the hierarchy - all other classes refer to it.

Immediately above Sof are the mathematics classes: scalars, vectors, and matrices. These implement vector and linear algebra and other useful mathematical operations (i.e., dynamic programming and LMS analysis). The next

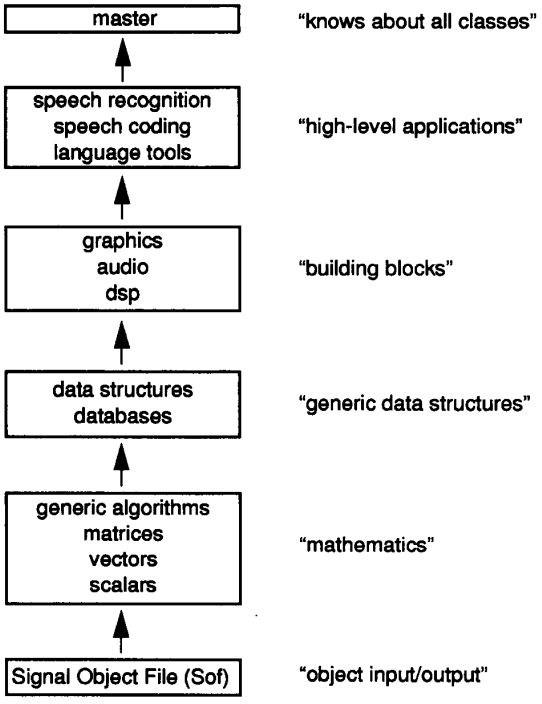

Figure 2. A structured programming implementation of a signal processing class hierarchy. All objects can read/write themselves to files - hence, the file I/O class is at the bottom of the hierarchy. All mathematics is performed using a well-developed set of mathematics operators that support all of the basic C operators and linear algebra constructs.

level up are the dsp building blocks. Signal processing constructs, such as signals, filters, and fundamental frequency are contained in the dsp classes, and are built from (inherit) the math classes. Above these classes are more complex systems, such as tools that process text in a language independent manner. Finally, at the highest level, there are the master classes. These are a special set of classes that are allowed to know about all other classes in the environment. Only in very special situations involving file I/O are these necessary.

3. THE SIGNAL OBJECT FILE (SOF)

At the heart of any good speech recognition environment is a flexible data file format. At TI, we have progressed through three generations file systems: SP_SOFT, SD_SOFT, and SDF_SOFT [6]. As our software requirements advanced, each of these systems became inadequate. Our newest generation, the Signal Object File (Sof) melds many of the features of these previous systems, and introduces three fundamentally new capabilities: transparent support of ASCII and machine-independent binary files; the ability to store multiple instances of arbitrary user-defined objects in a file; and the ability to retrieve pieces of a complicated object.

In an object-oriented system, it is very important to be able to store multiple instances of an object. For example, a simple revisioning system has been implemented by a class called Revision_history. Most Sof files will contain multiple instances of this object describing changes that have been made to the file. We also use this feature to support large text databases: sentences in a text database are simply written to a file as text objects. Each file may contain thousands of sentences.

Before we implemented the Sof class, a survey of most object-oriented I/O schemes turned up one disappointing fact: each piece of data had to be tagged with some identification bits. This is usually prohibitively expensive for vector-oriented data types such as speech signals. We would like a storage format that does not introduce any overhead per signal sample, and a format that supports storage of vector data in "machine-dependent" sizes, such as short integers or one-byte unsigned characters.

The Sof class is nothing more than an index manager: it maintains a list of objects in the file (class names and object tag) and their locations in the file. The actual I/O methods are member functions of each class - they are not generic member functions of the Sof class. Since each class is a hierarchy of other, lower-level classes, implementing these methods for each class is straightforward - execute the I/O methods for each object in the class to be stored externally in a file. At the very bottom of the hierarchy are the scalar classes - these know how to read/write themselves to binary or ASCII files in a machine-independent manner.

ASCII files are quite simple. An example is shown in Fig. X. They simply consist of data prefaced by an object tag. An index is constructed in the fly when the file is opened. Binary files are somewhat more complicated than ASCII files. These contain, in addition to the data, an index of object names in the file and the starting locations of these objects in the file. After a file is opened, processing of ASCII and binary files are identical!

ASCII files are provided so that editable and readable files can be supported. All data files in the environment, whether parameter specifications (typically ASCII files), signal files, text databases, or recognition models, are Sof files. Users need not write any special parsing algorithms for parameter files - something that usually consumes a large amount of time. Sof provides a single, uniform interface to all files.

One other important aspect of file I/O for signal processing is real-time I/O. For audio applications, we need to be able to read/write them to audio devices in real-time with a minimum delay (< 0.25 secs). Fortunately, as the speed of disks and Unix workstations has increased, this becomes easier to do. In fact, file I/O is generally in the noise now - except for real-time applications. Since Sof defers I/O to each class, those classes that need to do partial I/O or special I/O, have been augmented with special functions to do this. Since most of the classes requiring to do this use vector data, they normally inherit the basic vector data I/O methods.

4. USEFUL BUILDING BLOCKS

Such an environment is simply not useful without a well-developed set of mathematics classes. Our scalar classes currently consist of the following "C" integral types: one byte characters, complex numbers, four and eight byte floats, signed integers (one through four bytes long), a logical variable, unsigned integers, and a void pointer. Some special classes are also included: a four-byte bit field (bits_4), a Q point arithmetic class (Qfloat_8), and a TMS320C30 DSP floating point simulator (TMS320C30_float). Each of these types is also supported with a corresponding vector and matrix class. all normal C

```
SOF2.0
# Signal 0 #
 (Sample_frequency = 8000.000000 Hz)
 (Sample_type = 1 int_2)
 (Sample_precision = 16 bits)
 (Sample_compression = 2 linear)
 (Num_channels = 2)
5 3
255 -367 1024 -699 27
-266 -555 1024
```

Figure 3. An ASCII Sof file containing two objects: a comment object and a signal object.

operators are overloaded, and all useful linear algebra constructs are supported. Hence, it is simple to write "a=b+c-d" and operate on scalars, vectors, or matrices.

About the only other exotic feature we use from the C++ language is the virtual function mechanism. In many situations, it is desirable to hide implementation specifics for a specific device. For example, in audio, we want to play any signal to any audio device, independent of the format of the signal or the type of audio device. We have chosen to implement this feature using a virtual function approach. The "parent" object is created that contains a standard set of virtual functions that represent the software interface to a device. At run-time, when a particular device is requested, the virtual functions are overloaded with the specific device properties. The specific implementation for a device is completely under programmer control, and is independent of all other device implementations. The system is easily extended to new devices by modifying only one file in the parent class. We support, for example, four significantly different audio systems with this approach. This methodology can also be implemented in C using function pointer tables, but the code is much more complex. In C++, this mechanism is built into the language.

5. MORE INTERESTING ABSTRACTIONS

One of the most fundamental data types to signal processing is the class called Signal. In our approach, a Signal is represented internally in the CPU as a floating point matrix (four byte float). This is sufficient precision for most signal processing applications, and more precision than used to store signals in files. A signal is implicitly multi-dimensional: each channel of the signal represents a row in the matrix. Hence, we inherit the wealth of vector operations available in the vector classes, and make processing of the channels of a signal straightforward.

Signals have the following attributes: sample frequency, sample type (the number of bytes), sample precision (the number of bits), and sample compression (linear, log, etc.). For example, an 8 bit μ-law codec sampling at 8 kHz would return sample frequency = 8 kHz, sample type = 1 byte unsigned integer (uint_1), sample precision = 8 bits, and sample compression = μ-law.

It is important, for efficiency reasons, that signals are stored in compressed forms in files. During I/O, signals are transformed from their external compressed form into a common 4-byte float form. Signals are converted on-the-fly from multi-channel to single-channel, as need be, to be able to be played on less capable audio systems. The Signal class is, in fact, one of the basic objects in our system.

The next step up from signals is a class called Signal_model. This consists of a hierarchy of three classes: signal_measurements, signal_parameters, and signal_observations [4]. The first class represents the output of the spectral analysis; the second class represents the output of operations such as time differentiation, differencing, etc.; the third class represents the output of any statistical conditioning (decorrelation, variance-weighting, etc.). Most of the common spectral analysis techniques used in speech recognition systems today (LPC, cepstrum, filter bank) are included in this class. Signal_model is actually built from many of the classes that exist in the DSP class library.

Perhaps the greatest benefit of structured programming surfaces when implementing a large scale system such as a speech recognizer. The top-level class in our current implementation is Recognizer - denoting a general class capable of operating on signals or signal_models. Within this class, search algorithms are implemented using the virtual function approach previously described. Each search algorithm (i.e., Viterbi beam search, N-best search, LR parser) is implemented as a sub-class linked to the parent class by a set of virtual functions. Signal processing, is implemented via the signal_model class, and is completely specified at run-time by a parameter file. Since each algorithm implementation controls its own data space, new programmers can easily add new algorithms to the system without needing to understand someone else's "obscure" implementation.

6. SUMMARY

This paper has served as mainly a high-level introduction to our signal processing research environment. The current environment consists of 500,000 lines of code, and is continuously growing. There are many useful abstractions for signal processing not discussed here. We emphasize that the construction presented here is targeted mainly for research simulations - so we have tried to keep the level of programming required very simple.

7. REFERENCES

1. B. Stroustrup, *The C++ Programming Language* (2nd Ed.), Addison-Wesley Publishing Co., New York, New York, 1991.

2. J. Picone, "On The Precision Requirements Of Signal Interpolators For High Quality Digital Audio Systems," *Texas Instruments Incorporated*, Dallas, Texas, TI Tech. Rep. No. CSC-TR-92-002, pp. 1-32, April 14, 1992.

3. J. Picone, "Signal Modeling Techniques in Speech Recognition," submitted to the *Proceedings of the IEEE* in June 1992.

4. G.E. Kopec, "The Signal Representation Language," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 33, no. 4, pp. 921-931, August 1985.

5 J. Picone, "A Functional Specification for SDF_SOFT," *Texas Instruments Incorporated*, Dallas, Texas, TI Tech. Rep. No. CSC-TR-88-002, pp. 1-12, May 15, 1989.