Some thoughts on the design of software for geophysical data processing

Paul J. Fowler

ABSTRACT

Most seismic data processing is currently based on a data model of traces and trace headers. This model is probably too restrictive for future processing environments. Integration of seismic data with other forms of data in a more general database model would be more flexible and extensible. If data models are defined using a limited number of access and processing parameters, processing modules can be defined as realizations of generic filters. A limited standardization of data models in this fashion might also enable standardization of processors.

INTRODUCTION

At the 1987 S.E.G. annual meeting in New Orleans, a workshop was convened to examine whether it was feasible to consider standardizing the seismic data processing environment enough so that people could exchange processing code. As it stands, most companies have their own home-grown processing systems, each unique unto itself. This makes it very difficult for vendors to develop and sell code to implement new processing methods. It also means that if a sponsor is enamored of some idea developed at SEP or another university project, they must perforce rewrite all the code. And yet the data being handled by everyone is essentially the same, and the overall model of what is being done to it during processing is widely understood. The benefits of sharing a common standardized basis have long been recognized for programming languages, and are coming to be recognized for operating systems as well (for example, the forthcoming Posix standard). A seismic processing package shares many similarities with other large software projects such as compilers or operating systems; perhaps it is indeed time to consider how much can be standardized here too.

In this paper I present some of my thoughts in response to issues raised at the S.E.G. workshop. Although I believe that some standardization of processing software architecture is both feasible and desirable, I am not too sanguine about its immediate likelihood. Overcoming the natural inertial resistance to any disruption of the status quo requires economic motivation that probably does not exist yet. Also, there exists a common tendency in programming shops toward the "N.I.H" (Not Invented Here) syndrome, a suspicion and distrust of any software ideas that are not home grown. However, thinking about the possibility of standardization can raise issues that have larger implications that are worth examining. In particular, to standardize processors, one first has to define what one means by the term. To me, a processor is a filter that acts upon data,

where, generically, a filter is anything that accepts defined input and produces defined output. Thus, to define a processor, one first has to define the model of data being used, so that one can define the input and output to a filter. I thus spend much of this paper talking about models for geophysical data before returning to the question of standard definitions for processing.

I write this aware that my own knowledge of processing systems is limited, and that I am not putting forward any detailed proposals or solutions. I have considerable respect for the designers of some of the processing systems I have encountered, and for the amount of work that has gone into these software packages. What follows is opinions, sometimes deliberately overstated in the hope that I can raise issues and maybe stimulate discussion. I suspect that many others have already thought more deeply about these issues; if so, I would enjoy hearing what they have to say.

TRACE-ORIENTED PROCESSING

SEP currently has two commercial seismic data processing packages, SKS and DISCO. Both of these, like most seismic processing systems with which I am familiar, are based around sequential flow of data traces, using some modification of SEG-Y trace headers[1] (Barry, et al., 1975). The basic data unit is thus a seismic trace, with a prepended header (of a supposedly standard format) describing that trace. SEG-Y was designed as a standard format for archiving and exchange of seismic data on magnetic tape. It has served well, although it is often extended and modified, and may be getting a bit long in the tooth these days. Different processing systems use different formats for storing traces on disk during processing; the standardization does not carry over to disk files, which can be a significant annoyance.

The SEG-Y standard reserved only a short space (240 bytes) for the trace header, with few fields available for user definition. Because of this, it can run into problems with, say, 3-D surveys. Moreover, every read or write of a trace involves a similar read or write of the corresponding header. This can become particularly annoying when the headers are essentially meaningless, as can happen when the data is transformed by many processors. I first encountered this problem when writing a Stolt F-K migration program under SKS. What do trace headers mean after 2-D Fourier transformation of the data? My solution then was to strip off all the headers, collect them in a disk file separate from the data, and reattach them, modified as needed, on output. The trace headers in that case became an impediment, not an aid, to coding. The underlying problem was that my algorithm, in calling for a 2-D transform, changed the data in a way for which the model of simple traces and headers was not well-suited. The structure of algorithms should reflect the data structures; the data structures here were too limited.

These problems are not limited to 2-D Fourier transforms; similar problems could arise with slant stacks, velocity stacks, radial traces, or many other transformations just waiting for someone to invent. The problem becomes worse when one considers all the many other types of data one might want to use along with seismic data. Velocity fields

[1] I refer throughout this paper to trace headers with reference to the SEG-Y standard. SEG-D and other similar standards retain the same model of traces and headers, so I believe that what I say here should apply to them as well. My concern here is with demultiplexed data, not field tapes.

are one obvious case. Our installation of SKS has six different velocity formatting programs because different companies use different formats for the same information; in the absence of standardization, anarchy prevails. Survey and telemetry data for complex acquisition geometries are another problematic case.

The problem is going to get worse soon, not better. The tremendous advances in computational and display capabilities now becoming common in workstations, combined with the advent of efficient networks linking heterogeneous computers, are blurring the distinctions between processing and interpreting geophysical data. Concomitant with this is the need to integrate a variety of different types of data and a wide range of tools for handling these data. Forcing every data type into SEG-Y format is too procrustean a solution. Likewise, developing ad hoc formats and solutions for each new situation invites the growth of incomprehensible, incompatible, and unmaintainable code.[2] I suspect that it will prove easier to try to plan ahead now. How can data handling systems be designed to remain flexible and extensible with an expanding range of data types, and to include from the start a modicum of standardization, enough so that companies, contractors, and maybe even universities can communicate about algorithms? Various database models have been implemented for handling nonseismic data. In principle, trace headers could be stripped off seismic data as it is read in from tape, and a database constructed to describe the data, with headers only reappended when data is again output to tape. This way, it might prove possible to integrate seismic and nonseismic data much more readily than if trace headers are retained. It could eliminate redundancy (most of the contents of a trace header are normally highly predictable from previous headers), and cut back on unneeded I/O of headers. It also encourages one to think of seismic data less rigidly than just as a flow of homogeneous traces.

DATABASE MODELS FOR PROCESSING

Data in a digital computer comprises patterns of bits. How we interpret these bit patterns depends on what we believe they are intended to represent, be it integers, floating point numbers, characters, executable machine instructions, memory addresses, or whatever. This information telling us how to understand the bits found in some section of memory is either known by common agreement (e.g., SEG-Y trace headers are always supposed to be 240 bytes long, with the meanings of at least the first 180 bytes standardized), or is obtained indirectly, by reading some information from a standardized location (e.g., bytes 115-116 of the SEG-Y trace headers, and bytes 3221-3222 of the reel headers, are supposed to tell you how many samples are in each trace, and thus how large the offset between trace headers will be.) The indirect information forms a simple database, and the standards establish a common format so that the database can be

[2] I have to acknowledge here that there exist strong forces fighting against any effort to make code easy to understand, maintain, standardize, etc. Historically, computing has been controlled by centralized bureaucracies whose gurus drew their power (and job security) precisely from being the only ones who understood the magic incantations needed to use the computers successfully. The arcane operating systems used by some major manufacturers have abetted these people. Networked workstations and personal computers will work to subvert the rule of these cabals, because people who have powerful and easy-to-use machines on their desks should be harder to cow into submission. When the data processing bureaucrats and supposed gurus see their power being eroded, they will fight back. Don't give in; such people are defending their own selfish interests, and don't really give a fig about you or your company. They are best sent off to the countryside to dig potatoes in re-education camps anyway.

read. But as a database it is too inflexible and too homogeneous in its data model to serve well for the future.[3] The database (headers) is too restrictive for incorporating heterogeneous data types, and the interlacing of headers with data is clumsy. I would like to propose that data be treated simply as bytes in memory, or in disk files, and the information needed to interpret data files be maintained as a separate database, in distinct files. Moreover, the database should not be rigidly specified byte by byte; all that is needed is a general agreement on what sorts of information it contains, how to query it, and how to modify or update it. That is, any standardization should concern the external appearance of the database (how one interacts with it), not the actual format of how it is implemented by a specific programmer on a particular machine.

Learning from UNIX

Before attempting to outline a processing database model in any further detail, I would like to discuss two examples from which I have borrowed concepts. The first is the UNIX operating system. UNIX treats data files essentially as I have proposed above, as simply clumps of bits. The user is free to read data files in any manner he or she chooses; the operating system takes care of such things as buffering, blocking, and maintaining current position pointers in a file, unless the user explicitly chooses to manipulate them. The operating system also maintains a simple "database" telling what additional disk blocks are used for a given file, what the file permissions are, and such like. This simplicity allows the user to make silly errors - the most common probably is to attempt to display non-characters file (such as executable files), which upsets most terminals' digestion severely. On the other hand, it also allows UNIX to use a particularly simple hierarchical tree structure for files (because directories are just files whose database contains a notation that they are to be treated a little differently). It also allows UNIX to treat a wide variety of I/O devices as simply special cases of ordinary files. Thus, for example, one can send output to one's terminal screen, to a disk file, to a tape, or to a plotter, all using the same command syntax, because they are all named as files. The operating system simply checks the database description of the specified output file, and calls the appropriate driver programs to interpret the bytes into the necessary form. In some cases, such as terminals, the operating system, after identifying the output file as a terminal, scurries off to consult a secondary database file to find out the peculiarities of that particular terminal so that it can know how to address it in the correct dialect of gobbledygook used by that terminal.

What features of the UNIX file system are germane to the problem of geophysical processing? First is the idea that data files can be widely varying beasts, but can treated very similarly if one maintains a guidebook to their idiosyncracies. The way that data is stored in a file does not have to be a programmer's primary concern, provided he or she knows how to consult the database, implicitly or explicitly. Thus, for example, one might specify a surface location and want to call up a seismic gather near this location (either an actual profile, an interpolated one, or some other processed version), a segment

[3] I emphasize again that this is not a complaint against SEG-Y or its kin as *exchange* formats, but rather a warning about the limitations of trying to use them as processing database formats. It is probably an excellent idea to keep trace headers interlaced with seismic data on archival or exchange tapes, so that in the event of damage to the tape, as much data as possible can still be recovered.

of a gravity survey, an interpreted model, and a picture of rays traced from one or more horizons. Each of these will probably be stored differently. A survey database could be consulted first, to relate the specified location to the locations of geophysical experiments actually available, which would provide keys for querying the database files describing each type of data. The data desired could then be retrieved according to the information about its storage found in the database. The final desired data might also result from further computation on retrieved data. For an example of the latter, one might have horizons and velocities stored as data, and a call for a ray trace picture might fire up one or more programs to compute and display the rays. The database thus might contain not just descriptions of data files, but possibly also instructions on how to find and execute other files (programs) in more complicated ways. The possible use of a secondary database, such as one for survey data or for velocity fields, is another concept that could be borrowed from UNIX. The primary database does not have to contain all the details; it can instead redirect queries to other specialized databases as needed. This last concept should allow tremendous flexibility and extensibility - one does not have to guess in advance what future data will look like, but only need know how to construct and query files to describe that data when the need arises.

Learning from Seplib

The most commonly used system for experimental data processing at SEP is a home-grown known as Seplib. It has been briefly described in a previous SEP report by Claerbout (1986). It uses a simple database system, and relies on UNIX pipes to connect different processors. The database used is in the form of an ASCII file, called a history or header file, that contains lines of the form "parameter=value". These parameters tell where to find a data set ("in=filename"), or describe some aspect of the incoming data ("n1=1024"), or describe some value of a parameter used by some previous processor. Each processor reads the history file describing the data, reads in further parameters from the operating system command line or from a designated file of processing parameters, processes the data, and updates the history file by indicating the name of that process, when it was applied, what processing parameters were used, and what changes were made in any of the parameters describing the data. So that programs written by various individuals could communicate, the names of several of the most basic parameters were standardized. For example, "n1" always specifies the number of samples on the fastest storage axis of the data (the number of samples per trace for conventional data).

In its original form, Seplib was designed with a simple picture of data, because it assumed that data was homogeneous, at most three dimensional, and always regularly sampled on every axis. It has not been hard to extend it (although much of the extension has been done in an ad hoc fashion). Because parameters can have character string values as well as integer or floating point values, one can read in a name of a file that contains values of an irregularly sampled data axis, and use that file as a secondary database. Likewise, the ability to read and write to more than one input and output file was soon added. Another extension was to write tools to read and utilize not just the current value of a parameter, but prior values as well.

Seplib is a useful tool for rapid development and testing of geophysical algorithms in a UNIX environment. It is too limited in conception to provide a ready basis for production environments of the present or the future. However, several ideas from it are

worth highlighting. It utilizes a simple database to tell where data is located, and to describe how to interpret that data. It also maintains a simple history of what has been done to bring the data to its current status.

General database models

I have deliberately used (abused?) the phrase "database" here without any effort to define it well. In particular, I have run roughshod over the formal distinctions between data, database, and database management systems; I hope my meaning survives despite my ignorance of much of the terminology and theory of proper databases.[4] There exists a large literature about design and organization of database and database management systems. Geophysicists wishing to get their feet wet might start with Hatton (1984) or Dillahunty, et al. (1980) before braving the abysses to be found in a computer science library. My principal goal here has been to suggest that processing systems based on SEG-Y trace headers define the permissible data model so rigidly as to make extension for future (and present) needs very difficult. A better model might be to separate the data files themselves from the files that tell one how to find and understand the part of the data that one needs.

What form should the descriptive "database" files take? One needs first to be able to know how to find a particular piece of data, that is, in what file to look, at what offset into that file to start reading data, what size chunks (in bytes) the data comes in, and how many of these chunks are to be read. First, of course, one has to tell what data files or address ranges in memory are germane. Also, one has to specify what a basic data object looks like (integer, floating point number, or maybe some more complicated structure). The definition of the basic data object may well vary for different types of data, and may even vary with different times that one accesses data; one person's complex number is another person's pair of floating point numbers, for a trivial example.

Once the basic data object is defined, most geophysical data I can think of can be indexed by specifying the values of a relatively few parameters, usually the surface location coordinates, line number, shot number, depth, time, offset, or suchlike. These parameters, even if unevenly sampled, can be thought of as defining generalized axes for the data space. These parameters define the link between the location of data in a file and its implied location in the processor's model of a data space, so I will call them *access* parameters. Thus, given, say, two surface location coordinates for a shot and two for a phone, the access parameters would have to tell in what order traces are stored, so that one can find where in memory or disk the appropriate trace is stored, and how many samples long it is. Note that the number of access parameters specified depends on what type of data chunk one wants; in this example, to get a shot profile, one would specify two coordinates, to get a trace, four, and to get a particular sample, five. Note also that, because computer storage in both disk and memory is ultimately addressed as a (linear) offset from some reference point, a hierarchy of access parameters

[4] For example, Rick Ottolini tells me that I am arguing for what would be termed dynamic or self-defining database schemata, instead of the external, fixed schema imposed by SEG-Y trace headers. I tried to consult a couple of books from our computer science library on the subject of databases in the hope of at least gleaning useful jargon. Alas, I found them as impenetrable as a computer scientist probably would find these reports!

must be maintained, from the slowest changing to the fastest. Finally, note that interpreting access parameters may utilize secondary data files or databases to keep track of the intricacies of things like variable shooting geometries.

Some operations such as subsampling, windowing, interpolation, or padding, could change the *values* of the access parameters. Other operations, such as change of coordinates, resorting, Fourier transformation, or slant stacking could change the *definitions* of the appropriate access parameters to be used. Thus, the set of access parameters must be flagged as to which ones are currently active. Inactive parameters should be retained, however, as well as previous values of all parameters, so that, for example, after padding and Fourier transforming, one can inverse transform and automatically have the same data sampling one began with if so desired. As Claerbout (1986) points out, this becomes particularly important when one goes in and out of any transform space repeatedly.

Besides parameters that tell one how to read data files, a processing database needs to keep track of what has been done to the data to get it into its shape. This means keeping track (probably as dynamic stacks) of what processors have been applied to the data, and when. There will also be many acquisition and processing parameters that are not needed to define the "size" or "shape" of the data access, but do help tell how the data got the particular values it has. Examples of the latter might include mutes, tapers, weights, filters, etc., applied during acquisition or processing. The current and previous values of these *processing* parameters should be maintained also.

Under this model, a processor is a filter acting on data. The input(s) and output(s) can be described in terms of the active access parameters; a processor is fundamentally characterized by what access parameters it uses, and what changes it makes in them. *Specifying a processor as a filter in this manner should be conducive to the creation of fairly simple standards for defining processors.* That is, we may never agree on just how an NMO routine should work, but we might be able to agree that a generic (2-D) NMO processor accepts data in the form of midpoint gathers with access parameters that define data axes of time, offset, and midpoint, and that the output data have the same description. And I do not think that agreeing on standard names for some of these common parameters, or maybe even on a standard syntax[5] for routines to retrieve or change values for these access parameters, is necessarily too utopian to hope for. We in SEG have previously managed to agree on such details as that a value of 2 in bytes 139-140 of a SEG-Y header would indicate that a $\cos^2$ taper was used on a vibroseis sweep. Who knows, we might even be able to agree on the names of a few processors and their processing parameters someday, too (but don't count on it!)

I have ignored the problem of how processors are linked together. This can be handled either by the operating system (e.g., UNIX provides the pipeline utilities used by

[5] Syntax for subroutine calls is of course dependent on the programming language used. I suppose one could implement any new processing system in Fortran, but until the standard Fortran definition starts providing dynamic allocation, pointers, and heterogeneous data structures, such an implementation will be painful. Do we really want to implement the software of the 1990's using the programming language of the 1950's? We could also agree to make Sumerian cuneiform the official language of the S.E.G. But do we really want to? There, I have vented my spleen about Fortran, and if I am lucky, all the ardent Fortran fans out there will have eyesight too poor (or common sense too abundant) to bother reading these tiny sarcastic footnotes.

Seplib), or by a separate supervisory program that builds a pipeline out of smaller processing modules (e.g., the SKS translator or the DISCO monitor). If processing modules are to be designed as specific realizations of standardized filters, they must remain as ignorant as possible of their interaction with other filters. Hence they must worry as little as possible about where they are getting their data from, that is, whether the data is currently on disk, in memory, in an array processor, or off on vacation in Jamaica.[6]. Fortran predates modern operating systems, so Fortran programmers are used to worrying about many details of I/O that most more recent programming languages rely on the operating system to provide. I think seismic processing systems should adopt the latter view: data flow is the concern of the supervisory program and the operating system, not the individual modules. The module should ask to be provided with a described chunk of data, and not care how this demand is handled. Unless modules can be shielded from the (usually language and system dependent) details of data I/O, standardization may prove well nigh impossible.

CONCLUSIONS

I have tried to argue that standardization of seismic processing depends first on adopting a simple and general data model. I would like to see the model of interlaced traces and headers abandoned in favor of some form of database system that can handle more heterogeneous data types comfortably. If this can be done, processing modules could be written as general filters, with the primary level of standardization being a common way of describing what data comes in and what data goes out. I have suggested only in a fuzzy way how I think this might be accomplished. I hope that with this paper I can provoke others into telling me why I am wrong, and how to do it better!

ACKNOWLEDGEMENTS

I thank the organizers of the SEG workshop, Elmer Eisner and Eike Reitsch, as well as the panelists, Ray Farrell, Les Hatton, Rick Ottolini, and Ben Thigpen. Les Hatton has greatly influenced my thinking about software design through the years. Rick Ottolini encouraged me to organize my inchoate thoughts in writing; he and Joe Dellinger both provided useful comments on a draft version of this paper. Jon Claerbout and Stew Levin were the principal architects of Seplib, which is a much cleverer system than I originally realized.

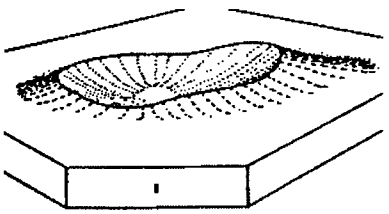
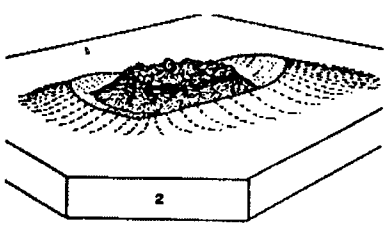
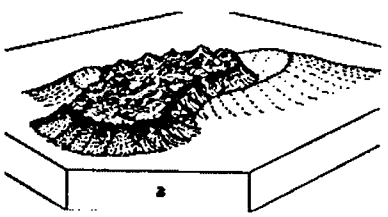
Foolish ideas herein are of course my own responsibility, not that of any of these other individuals.

[6] Ideally, the rapid growth of giant high speed core memories will overtake the growth of seismic data sets, and allow us to assume that all data is available in core. Then programmers could get their work done much faster, and take more vacations in Jamaica themselves. I am not holding my breath waiting.

REFERENCES

Barry, K.M., Cavers, D.A., and Kneale, D.W., 1975, Recommended standards for digital tape formats: Geophysics, **40**, 344-352.

Claerbout, J.F., 1986, A canonical program library: SEP-50.

Dillahunty, R.C., Fash, J.L., Parsley, L.R., and Townsend, D.W., 1980, Geophysical data bases: systems considerations: Geophysical Prospecting, **28**, 495-512.

Hatton, Les, 1984, Computer science for geophysicists, Part V: databases and expert systems: First Break, **2**, no. 1, pp. 9-15.