Stanford Exploration Project, Report 82, May 11, 2001, pages 1–199

Short Note

Birth of a C++ Project

Matthias Schwab[1]

INTRODUCTION

At the end of this summer, SEP began a collaboration on a C++ class library with Bill Symes and Mark Gockenbach from Rice University's TRIP project and Lester Dye from Stanford's Petroleum Engineering Department. The new C++ class library intends to split complex inversion algorithms into manageable pieces. Application specialists, such as geophysicists, write code dealing with their area of expertise, while numerical analysts write general solver routines. However, the authors derive their operator and solver from predefined base classes that guarantee the compatibility of the derived classes.

The C++ object oriented language (Stroustrup, 1991) is widely popular in different fields of science and engineering. Our last year's C++ project, CLOP (**?**), demonstrated the feasibility of a geophysical inversion library. CLOP's vector class is based on a software library, M++ (**?**), which unfortunately limits CLOP in various ways. Additionally, CLOP is not able always to handle nonlinear inversion conveniently.

Dye and Nichols (personal communication) as well as Gockenbach (Gockenbach, 1994) independently designed C++ libraries for large scale inversion problems. The new C++ project will merge the two libraries and complete the resulting class library.

OPTIMIZATION SOFTWARE

Good optimization software integrates numerical analysis routines and application routines conveniently, reliably, and efficiently.

CLOP's pros

In his two most recent books (**?**Claerbout, 1994), Claerbout wrote 37 routines which integrate his conjugate-gradient solver routine with the particular operator routines he discusses in various chapters. The subroutine below displays such an integration in the case of an NMO

[1]**email:** matt@sep.stanford.edu

operation `invstack()` on this page. The subroutine is crowded with computational detail, and

```
# NMO stack by inverse of forward modeling
#
subroutine invstack(  nt,model,nx,gather,rr,t0,x0,dt,dx,slow,niter)
integer it, ix, iter, nt,      nx,                           niter
real    t0,x0,dt,dx,slow, gather(nt,nx), rr(nt,nx), model(nt)
temporary real  dmodel(nt), smodel(nt), dr(nt,nx), sr(nt,nx)
do it= 1, nt
        model(it) = 0.0
do it= 1, nt
        do ix= 1, nx
                rr(it,ix) = gather(it,ix)
do iter = 0, niter {
        call imospray( 1,0,slow,x0,dx,t0,dt,nx,nt,dmodel,rr)    # nmo-stack
        call imospray( 0,0,slow,x0,dx,t0,dt,nx,nt,dmodel,dr)    # modeling
        call cgstep(iter, nt, model, dmodel, smodel, nt*nx, rr, dr, sr)
        }
return; end
```

Figure 1: Ratfor code extract: combination of a conjugate-gradient solver and an NMO operation.

personal experience shows, the code is hard to debug. Contrary, CLOP's NMO optimization code shown below `SolvNMO()` on the current page consists of basically three concise steps: first, the operator is constructed; secondly, the conjugate-gradient solver is constructed; finally, the input data is inverted using the solver. The objects smoothly interact with each other, because they are all derived from standard base classes which define the object's interfaces. Since the concepts of operator, data, and solver are cleanly separated in CLOP, the library

```
    fopNMO op(inaxlist[0],inaxlist[1],1/vel);       // make NMO operator

    HestSolver solver( op, niter );                 // construct solver object
    solver.ReportOn(cerr);                          // report progress on stderr
    floatspacearray output = solver.Solve( input ); // solve the problem
```

Figure 2: CLOP code extract: combination of a conjugate-gradient solver and an NMO operation.

allows an independent implementation of classes by experts and tends to be reusable. Additionally, a programmer using CLOP can conveniently concatenate linear operator, $C = AB$, or arrange arrays of operators to new single operators, $C = [A, \lambda I]^T$.

CLOP's cons

CLOP uses the array class of M++, a commercial C++ class library (Dyad Software Corporation, 1991). M++ is limited to in-core processing which is insufficient for many geophysical problems. Additionally, the lack of templates in M++ requires code duplication for float and complex arrays. Since M++ is proprietary, SEP's CLOP document is not as freely distributable and reproducible as we might wish. Finally, CLOP's design as a library for inversion of linear problems does not facilitate convenient formulations for many nonlinear problems. Nichols (**?**) demonstrates CLOP's strength in his velocity analysis research using L_p norms. Unlike other nonlinear inversion problems, his reweighting scheme is conceptually easy to implement in CLOP's linear framework, since it consists of a series of weighted linear operator applications. Generally in CLOP, an operator's parameters – e.g. the filter coefficients of a convolution operator – can only be changed by reconstructing the operator; CLOP's operator parameters are not conceived as a part of the model space. Besides the effort of learning C++, these various reasons discouraged potential users inside and outside of SEP to use CLOP. At SEP only Nichols continued work within the CLOP framework regularly.

The new C++ class library

At the end of this summer, Lester Dye of the Department of Petroleum Engineering, Mark Gockenbach from Rice Universities' Applied Mathematics Department, and I agreed to collaborate on the development of a new C++ library for large scale inversion problems. Bill Symes, Dave Nichols, Hector Urdaneta, Jon Claerbout, and Martin Karrenbach plan to follow our progress and contribute as we proceed. A first draft of the new library is the result of merging earlier libraries by Dye, Nichols, and Gockenbach. Since different application areas in various scientific and engieneering disciplines share mathematical structures, the class design and the terminology of the new library strictly follow mathematical concepts and terminology. The three fundamental classes are a vector class, a general operator class, and a solver class. The vector class obeys the abstract mathematical definition of a vector, such as the addition and scaling of rule, and is not limited to an array of numbers. In geophysical applications however, a vector object will often represent data such as a SEPcube. The operator class represents a processing step, e.g. NMO, and is usually applied to a vector. A solver will take an operator and a vector and compute the corresponding inverted data. The vector class is currently being tested, though the design of the new operator and solver classes has just begun. In the new C++ library, a vector is represented by two base classes: the vector space class and the vector class. A vector space object contains the characteristics of the space such as its dimensionality. A vector object represents a vector and contains all operation that can be expected from vectors (e.g., adding another vector to it, or scaling itself by a scalar). The vector object also contains a pointer to its corresponding space. A Banach space class and a Banach class are derived from the vector space class and the vector class by adding a method to compute the norm of the vector. Hilbert space class and a Hilbert class, in turn, are derived from Banach classes but they contain an additional inner product method. Since a vector and its methods may vary in different applications, these classes exclusively define an interface (basically, all their methods are "virtual"). For each type of of data, e.g. SEPcube, TRIP data set, or a finite element mesh, a programmer derives a data class from a vector class and then implements all the functionality for the given interfaces. The collaboration to develop a new C++ library will be restricted to a common framework in the form of base classes and the implementation of some fundamental test classes (e.g. a matrix multiplication operator class, the

SEPcube-like NField vector class, and a conjugate-gradient solver class). The collaboration will not extend to research-type operators and solvers.

Some general design issues

When designing a class library, the authors are constantly balancing different quality criteria like efficiency or programmer friendliness. Here are some of the decisions we made when designing the vector class hierarchy and its first practical realization, the NField class. Mathematical symbols in the vector base class are not overloaded to avoid the creation of temporaries during execution time. Instead, the library uses functions like `add()` and `mult()`. Additionally, certain methods of computing linear combinations of vectors are defined in the vector base class. Consequently, these linear combinations can be used by solver classes. Since the classes are defaulted to an expression using `add()` and `mult()`, a derived class can choose to implement them to improve efficiency, but does not have to. NField class objects always reside in-place. When, for example, the matrix is transposed, only the offset and stride length of an index object are changed. The NField elements remain at their memory or disk location. Moving the elements to speed up excess along a certain dimension can be accomplished by copying the vector after transposition. Each object is required to have an `inspect()` method which prints some statements about the status of the object. The inspect method of an NField object may return the number of samples, their type, and the first few sample values. The vector objects are overloading the standard C++ input and output operator.

REFERENCES

Claerbout, J. F., 1994, Applications of three-dimensional filtering: SEP.

Dyad Software Corporation, . **M++ Class Library**. Renton, Washington, 1991.

Gockenbach, M. S., 1994, Object-oriented design for optimization and inversion software: a proposal: TRIP–**1994**, 1–24.

Stroustrup, B., 1991, The C++ programming language: Addison Weslsy Co., Massachusetts.

APPENDIX A

Here is a quiz FORTRAN user at SEP enjoyed taking: Below you see the contents of three files.

- What does the main routine `main()` on page 6 accomplish?
- Which of the functions in file vector.cc `methods()` on the next page is not used in the main routine?

- Assume, I would have given you only the file vector.h and I would have explained to you only what the “public” functions do. Could you have written the main routine?

```
class vector {
private:
  int sz;
  int *v;
public:
  vector(int dim);
  ~vector();
  int& operator() (int index);
  int add(vector& anotherVector);
};
```

Figure A-1: A simple vector class: header.

```
#include <iostream.h>
#include "vector.h"
vector::vector(int s) {
  if (s <= 0) cerr << "bad vector size" << endl;
  sz = s;
  v = new int[s];
}
vector::~vector() { delete[] v; }
vector::add( vector & u) {
  if ( u.sz != sz ) cerr << "vector dimensions incompatible" << endl;
  for (int i = 0; i < sz; i++) v[i] = v[i] + u.v[i];
}
int& vector::operator()(int i) {
  if ( i<0 || sz<=i) cerr << "index out of range" << endl;
    return v[i];
}
```

Figure A-2: A simple vector class: methods.

Answers:

- The main routine adds trace1 = trace1 + trace2 and then it prints trace1: five 5s. Give it a try!
- All of the functions are used. The destructor vector is used at the termination of the program. The indexing operator () resolves an expression `trace(index)`.
- Yes, you could have. The main routine only uses public functions of the vector class definition.

```
#include "vector.h"
#include <iostream.h>
main() {
vector trace1(5);
vector trace2(5);
for (int i=0; i < 5; i++) trace1(i) =     i;
for (    i=0; i < 5; i++) trace2(i) = 5 - i;
trace1.add(trace2);
for ( i=0; i < 5; i++)
  cout << trace1(i) << endl;
}
```

Figure A-3: A simple vector class: main routine.