Stanford Exploration Project, Report 93, November 11, 1997, pages 241–252

Geophysics in Object-Oriented Numerics (GOON): An informal conference

Jon Claerbout and Biondo Biondi[1]

keywords: *computing, inversion, algorithm, modeling*

ABSTRACT

We held an open and informal conference at SEP attempting to facilitate cooperation in introducing Object Oriented computational techniques among workers in Geophysics. Java seems attractive for the next century. For the remainder of this century a wide variety of language combination options were considered and many opinions were expressed. A few who stayed longest at the conference (perhaps the most dedicated to cooperation) developed a linkage of the Gockenbach-Symes HCL (Hilbert Class Library) in C++ to Fortran 90.

PREFACE

We held a "Workshop on object-oriented numerics and reproducible research in computational geophysics" at SEP in June 1996.

More people attended the workshop than appear in the lunch photo in Figure 1. The group is better indicated by the email addresses below.

Jon Claerbout <jon@sep.stanford.edu>,
Bill Symes <symes@masc39.rice.edu>,
Dave Nichols <nichols@gatwick.Geco-Prakla.slb.com>,
Stew Levin <salevin@dal.mobil.com>,
Martin Karrenbach <martin.karrenbach@physik.uni-karlsruhe.de>,
Mark Gockenbach <gock@math.lsa.umich.edu>,
Lydia Deng <hdeng@dix.mines.edu>,
Scott Morton <morton@cray.com>,
Joe Dellinger <jdellinger@amoco.com>,
Juan Meza <meza@ca.sandia.gov>,
Michel Kern <Michel.Kern@inria.fr>,
Jeff Scargle <jeffrey@sunshine.arc.nasa.gov>,
Stephan Matthai <matt@erdw.ethz.ch>,
proxy for Dave Donoho <donoho@playfair.stanford.edu>,
proxy for George Papanicolaou <papanico@georgep.stanford.edu>,

[1]**email:** jon@sep.stanford.edu, biondo@sep.stanford.edu

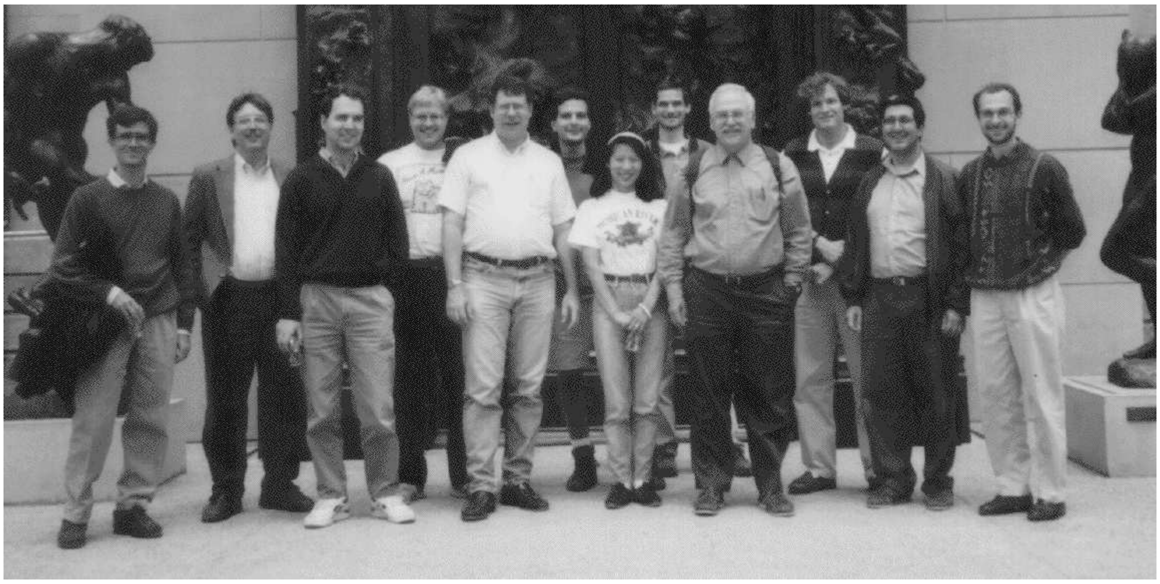

Figure 1: Picture taken after lunch on the third day of the conference. From the left: Biondo Biondi, Bill Symes, Dave Nichols, Scott Morton, Stew Levin, Hector Urdaneta, Lydia Deng, Matt Schwab, Jon Claerbout, Stephan Matthai, Michel Kern, Martin Karrenbach $\boxed{\text{jon1-goons}}$ [NR]

```
sep             <SEP mailing list>
```

INTRODUCTION

Through the years we have been disappointed by the number of occasions that inversion theory has not yielded successful results. The success ratio of inversion theory seems to match that of wildcat drilling. When results are disappointing it is hard to know the cause. Often we cannot confidently eliminate even one of the major possible causes: (1) inadequate data, (2) insufficient theoretical model, (3) failure of numerical techniques, (4) error in coding. To help students overcome the last two causes of frustration, one of us (JFC) developed the draft of an electronic book (TDF) which guides students through several tens of inverse problems using SEP's methods of "reproducible research". Basing the book on Fortran 77 has led to some painful defects that should be remedied before moving on to more extensive and realistic practical examples.

The principal difficulty with TDF in Fortran 77 is the way the inverse theory becomes intermingled with the application. Ideally, the geophysicist would simply set up the problem and then invoke a choice of canned mathematical and numerical solvers. Simply invoking external expertise is easy in Fortran where the geophysicist calls an external subroutine such as FFT. Fortran sets roadblocks, however, when the external intelligence, an optimizer for example, should invoke geophysical application code. Oversimplifying, it is easy when new code calls old code but hard when old code must call the new code. Using C++, there have been somewhat successful attempts to build environments isolating geophysics and mathematics at Rice University, at Colorado Mines, and here at Stanford. Considering the difficulties we at SEP have been having building and sustaining a seismic C++ research environment, we contacted Bill Symes at Rice and he confirmed a willingness to cooperate and attempt to gather a community of like minded researchers. Lydia Deng from Colorado Mines and Martin Karrenbach from Karlsruhe University also participated enthusiastically. From this joint effort sprung our report of this conference.

Where to start

A preconceived notion was that we would attempt to build on the C++ HCL (Hilbert Class Library) framework built by Bill Symes and Mark Gockenbach. This would allow us to leave numerical analysis and optimization to the mathematicians. This may be where we will end out; we hope so; but there are many other possibilities. We were surprised and disappointed to learn that the other three academic groups had the same difficulty we did, namely that a community of active C++ programmers did not emerge at their sites either. In other words, most geophysicists are too busy to go up the C++ learning curve and we cannot easily do what some of the contractors have done. Bill Symes at Rice has the most active academic geophysical C++ group, and the style there is that the math students code operators in Fortran 77 and the system built and maintained by Symes does the input/output, and optimization.

Fortran 90 (F90) is a potential alternative to C++ for breaking the constraints imposed by F77 shortcomings to geophysical inversion research. F90 is an advance over F77 in that it includes the modern idea of information hiding and complex data structures. The most important example of this is an operator like a sparse matrix and its adjoint being initialized by a geophysicist and being used by a mathematical optimizer. Fortran 90 should also offer us some advantages over C++ since compilers exist to parallelize F90, or the closely related HPF (High Performance Fortran), code on a variety of architectures, including our SGI Power Challenge and Connection Machine CM-5.

Both C++ and Fortran 90 have fairly recently matured to the stage where we can purchase compilers on all our general purpose workstations. F90 cannot be expected to reach the levels of conceptual abstraction possible in C++, but as a practical matter, we don't know how badly we really need such powerful abstractions. Also, it is too early to know if F90 will become a standard.

JAVA

The JAVA computer language recently appeared in connection with networking and it appears to solve many of the problems connected with C and C++. A strong objection that practical people have to C and C++ is that they may find themselves in a baffling situation where they are unable to debug their code. A byproduct of Java's design for network security is that weird machine-dependent behavior is theoretically impossible. If you make an error, you should get a meaningful message at compile time or at run time, unlike C, C++, and Fortran, where weird machine-dependent behavior often perplexes us. Supposedly, Java is C++ with the removal of dangerous, obscure, and unnecessary features.

At the present time, we see two big disadvantages of Java: (1) it seems to be 5-30 times slower than Fortran or C, (2) it is immature and (in our experience) embedded in unstable environments.

Scott Morton, Cray's representative to the geophysical world, shared with us his thoughts on a prototyper's philosophy including opinions of Arch Robison (designer of KAI's C++ optimizer). As for performance of Java, Robison expected it to match C by the year 2000. If his company does Java, he said, it will run faster than C++. He forecast that in 1997 there would be more academic conference papers of the my-extension-to-Java kind than of the my-extension-to-C++ kind. For 1998 he predicts that Java will predominate introductory computing languages for computer science majors. Robison was quoted saying that in 1996 local bookstore shelf space for Java exceeds that of Fortran, and in 1997 it will exceed that of C++. By 1997 the tools for Java will be so much better than the C++ tools that the only reason for C, C++, and F90 will be performance and legacy codes. By the year 2000, time to develop applications should be so much better in Java than in C++ that Java shops will have better survival statistics. By 2001 more parallel programs will be written in Java than in any other language.

We are delighted to find Mr. Robison, a powerful figure in the world of compilers, confidently confirming that our dreams are just around the corner, but 2001 is not here yet, and we seek to define now our first steps beyond F77.

DEBACLE AVERTED

After one week, it seemed that our workshop was failing to foster cooperation. Two distinct camps were emerging: one favoring a pure C++ solution and the other an F90 solution. At SEP we were experiencing the same cultural divide that exists in the programming world at large. Fortran and C have had an uneasy marriage for 20 years. Now their offspring, F90 and C++ are estranged. F90 is the community of those who value high performance computing, continuity with past numeric codes, easy of learning, and unambiguous code. C++ is the community of popular modern computer science, namely, powerful abstractions.

Most SEP students were opposed to the adoption of a pure C++ system. The reasons for such resistance are rooted in recent SEP history, and go beyond the actual shortcomings of C++ as a tool to implement numerical algorithms. First, several previous attempts to build a critical mass of C++ programmers at SEP have not been successful, inducing skepticism about the chances of a new effort. Second, our new Power Challenge has compilers generating efficient parallel multi-threaded executables from F90 code, but not from C++ code. The iterative nature of the inversion algorithms that we use, combined with the desire of working on 3-D problems, makes the efficient use of the available computing resources even more important for inversion than for our usual research. A related reason is that much of the utility software layered on the top of Seplib90 (Biondi et al., 1996), our new library for 3-D irregularly sampled data sets, had been already developed in F90 (Clapp and Crawley, 1996). Porting these libraries to C++ would be feasible, but an unwelcome chore.

On the other hand, despite the valiant efforts of several students, no simple and robust solution was found to implement in F90 several attractive capabilities that HCL easily supports in C++. Building a complex geophysical operator by combining simple ones can tremendously increase the speed of development and testing of sophisticated inversion methods. Complex operators can be built by assembling operators into an operator array, such as the $\mathrm{C} = [\mathrm{A}, \mathrm{I}]'$, or into an operator chain, such as C=AB. F90 does not support a general method for assembling operators. Even if specialized routines can be written for implementing particular assembly patterns, recursive combination, such as building chains of operator matrices, are impossible. Further, multiple instances of the same type of operator cannot be initialized with different set of parameters. Therefore, operator chains, such as $\mathrm{C} = \mathrm{A}_1\mathrm{B}\mathrm{A}_2$, cannot easily be constructed in F90. These difficulties are mostly consequences of the the restrictions on the use of subroutines and functions in F90. Function names can be passed around as arguments, but functions themselves cannot be components of derived types (F90 equivalent of C structures). Unfortunately, it seems that also Fortran 95 will inherit this severe handicap affecting the Fortran bloodline.

The logical solution of our mid-workshop impasse was combining C++ with F90; to take advantage of the attractive characteristics of both while bypassing their respective limitations. However, the binding of the F90 with C++ is difficult and non-standard. Because the two languages cater to different communities, compiler writers have not set a standard for linking object code generated from the two languages. Further, there are no defined interfaces specifying how to pass structures across language boundaries. Fortunately, our dynamic duo, Dave Nichols and Martin Karrenbach, overcame the conflict between Fortran 90 and C++ communities by finding a way to link them, at least on our SGI computer, and allowing crude cooperation until about 2001 when JAVA might subsume both. GOON was born.

GOON DEFINITION

The guiding principle of GOON is to take advantage of two modern computer languages, C++ and F90, and build a flexible and efficient system for testing geophysical inversion ideas. In GOON, C++ and F90 have two distinct and complementary roles. The main program and the solvers are adopted from HCL and are written in C++. C++ is used to assemble complex geophysical operators from simpler ones, by creating operator matrices and operator chains, and to pass these composite operators to a solver module. F90 is used for efficient and unambiguous coding of geophysical operators.

The lack of standardization of the interface to pass complex data structures between C++ and F90 is a major obstacle when binding C++ with F90 modules. GOON overcomes this difficulty by strictly separating the part of the system (written in F90) that operates on the data, from the part of the system (written in C++) that orchestrates the computations by manipulating pointers to F90 data structures. The complete separation between the brains (C++) and the brawn (F90) is facilitated by taking full advantage of the HCL abstract formulation of inverse problems as relating vectors belonging to Hilbert spaces. In HCL the data and the model are instances of vectors belonging to two, possibly coincident, Hilbert spaces. For a seismic data set the vector space is defined by the trace geometry (coordinates), and the data values (ordinates) identify individual vectors belonging to the space. All the operations performed by the solver on the data and model vectors can be expressed as vector functions, that can be easily coded in F90 and made accessible to the C++ solvers as library functions. Therefore, the C++ modules never need direct access to the data and model vectors.

In addition to enable a natural split between the C++ and the F90 modules, the Hilbert space abstraction makes the inversion system independent from the particular representation of the Hilbert vectors and thus it facilitates adding support for new data or model representations. At present, GOON supports both vectors representing regularly sampled data sets (Regular Gridded Functions or RGF) (Schroeder and Schwab, 1996) and irregularly sampled data sets (Irregular Gridded Functions or IGF) (Urdaneta and Karrenbach, 1996). To support a new data or model representation (e.g. layer-based velocity function) we would not need to modify the system, but only to add a new vector library for the new data type. For example, we plan to generalize the present RGF and IGF vector types to out-of-core vectors and apply inversion theory to realistic-scale 3-D problems. To add this new functionality, we will need only to add to the vector libraries the appropriate I/O calls, while leaving the main part of the system unchanged.

The GOON project is multi-faced and the software is still in the phase of early development. In this report several papers present different aspects of the GOON project; some of the papers are more tightly related to GOON than others, but all the projects have contributed in furthering SEP's understanding of inversion theory, and its effective implementation using modern computer languages. A brief guide to these papers is:

- Introduction to HCL by Hector Urdaneta and Matt Schwab (1996).
- Regular gridded function (RGF) in HCL by Joel Schroeder and and Matt Schwab (Convolution, Prediction error Filter, missing Data, linear and non-linear Solver, F77 operator wrapper) (1996).

- Irregular gridded function (IGF) in HCL by Martin Karrenbach and Hector Urdaneta (1996).
- Object-oriented F90 library for Seplib90 data sets by Bob Clapp and Sean Crawley (1996).
- Example of geophysical operator (NMO) in F90 (1996).

FEEDBACK

Two months after the conference a spirited free-for-all broke out on the local electronic bulletin board. Reformatting from the email `msgs` bulletin format to LaTeXbelow required me to exercise small editorial judgments that detract only slightly from exchange:

Bill Harlan: I'm sorry that F90 has sidetracked your project because I can't find anyone outside of the SEP interested in using F90. Is it really easier to learn to use F90 than C? Which is more likely to be useful after graduation?

If you use C++, then you can use as little of it as you prefer, and know that additional power is available should you need it. (Some will say that F90 is appropriate for parallelism. Any rules for parallelism you learn this year are unlikely to be useful in five years. Students shouldn't be worrying much about efficiency anyway.)

David Lumley: Which language? Probably the one that lets you code your ideas most efficiently, and can be road-tested on full size data sets in reasonable run times. This is some non-unique combination of user preference and run-time efficiency. Dave Nichols and I had an interesting experience with multiple suppression: we used Dave's C++ code to find the best objective function weighting scheme, etc., and used my Fortran code with those optimal parameters for the full-size runs on the Mobil AVO data because it was 30x faster than Dave's optimized C++ code. Both C and Fortran were useful.

Specific parallelizing rules maybe. Generic rules and concepts should be valid longer.

Depending on what you're doing, learning to develop efficient code can be extremely useful as a student. If you're scoping out something abstract like the behavior of an objective function, fine. But nobody will care how great your nifty C++ code is if it takes 24 cpu hours to run on a single shot gather (e.g., who really cares about 1-D full waveform inversion?). Learning to develop prototype code that can do something new efficiently is basic training for an applied research job in the industry. People who have this expertise when they go job hunting have a huge advantage over others who may also be bright, but can't test their ideas on data efficiently.

Dave Nichols: I would hesitate to use the word "optimized" about the C++ code we used. It was compiled with optimization on but it wasn't designed for fast operation. It was designed for maximum flexibility.

I could write a C++ code for the velocity stack inversion problem that gets to within a factor of two of Fortran code. It won't be quite as flexible as the code we had but it is still be usable within the C++ operator framework.

So my conclusions are:

1. There is always a tradeoff between generality and efficiency.
2. It is certainly true that Fortran compilers are better at optimizing many problems.
3. Out of the factor of 30× I would say that 2-3× was due to C++ vs Fortran, 2-3× was due to the way we designed the CLOP classes (lots of creation and deletion of objects in every iteration) and 4-7× was due to the use of a very general code. It is possible to write much less efficient code in C++ than in Fortran because of all the extra options you get!
4. Why did we use David's Fortran code rather than optimize the C++? Because once we had found a good set of parameters he wrote the Fortran code in a morning!

(This doesn't address the parallelism issue at all.)

Biondo Biondi: I think that we should not cast GOON as an F90-based system. I believe that one of the potential advantages of GOON is to exploit C++, F90 and C (yes C!) for what they do best.

C++ orchestrates the whole application, and enables us to write general and reusable code for the most of the constituent blocks, such as optimization algorithms and vector space functions and operators. Even if we write operators in F90, we can reuse them and combine them within the same application only thanks to the abstraction capabilities of C++.

Writing seismic operators in F90 instead of C/C++ is arguably easier in the absolute sense, but it is certainly easier for most SEPers. Being able to run them fast because compilers are available that can optimize/parallelize such F90 operators without much parallel/multi-threading coding is an additional, and I believe important, benefit.

All the low level I/O libraries for both the RGF and IGF vector classes are written in C. Such coding is what C was invented for, and it is where it adds the most value.

The potential problem with GOON is that we may have tried to be too smart in our effort of exploiting all the available languages at their best, and built a system that is difficult to use and maintain.

My hunch is that the whole team has succeeded in beating the odds, and actually GOON will be fairly simple to learn for new users, and maintainable, at least by their initial developers.

When the first "pure" users will try to use GOON (soon I hope) we will have some answers to these questions and have better feelings whether we have created a monster or a winner. I am very optimistic!

Bill Harlan: To SEP students: I think your continued use of F90, and inevitable dependence on it, will seriously limit the number of your collaborators. The only arguments I have heard in favor of F90 are (1) ease of use and (2) parallelism. As I said before, C and C++ are not harder to learn at a level comparable to F90, and you have much more potential to learn over the years.

No, I do not think the "general principles" you learn from F90 parallelization will continue to be useful. The sort of parallel machines you are likely to use (the Power Challenge) are not massively parallel. On 8 to 16 nodes, the most popular parallel styles use things called "threads" and "semaphores" (or worse PVM), which you are unlikely to see in F90 anyway. These styles are too complicated and are evolving too rapidly for students to keep up with, unless you have a strong interest in computer science.

Again, C is EASY to learn and use. Read Kernighan and Ritchie on a weekend. Use it like Fortran for a while. Don't feel obliged to use all the gadgets in C++ any time soon. You've got years to pick up more programming tricks. But start out on a path with a little more potential for growth. Do it for your own sake, even if group projects continue to use F90. Believe me, C[++] will look much better on your resume, will not take any more of your time, and will continue to be useful when you graduate.

Matt Schwab: I learn a new language when it changes my way of looking at my problems. In comparison to a procedural language, a strong object oriented language (C++, Java) does offer a very distinct viewpoint. Object orientation might allow a good researcher to tackle problems beyond "do looping".

For "do looping" I like to use Fortran77: I know it already and it parallelizes fine on our Power Challenge (not necessarily on other parallel machines). Do I need to learn F90?

If I had to learn "a first computer language" I hope I would start with C and/or C++ (as I suggested to Curt recently). It is a good language to know and C++ and Fortran are easy to add. Even new languages (e.g. Java) try to ease the transition for C programmers.

What language facilitates good collaboration? That depends on who you want to collaborate with: there exists a lot of F77 code (especially at SEP). On the other hand, the non-SEP researcher I collaborated with at TRIP (Rice) or SCCM (Stanford) coded C++ and C. All the code that I saw at TOTAL was C or C++. F90 and Java are "young" languages and I have hard time predicting their success. Java seems to be more fun ... and I like fun.

I feel C-C++-Java is the way to go in the long term. Biannual SEP reports, however, are a very short term thing. *Smiles*.

(Bill, I am glad you raise these issues because I wish I had learnt more about C, C++ in my early days at SEP.)

Sergey Fomel: What language facilitates good collaboration?

This question (related to the concept of reproducible research) is one of the most important to ask ourselves. Jon Claerbout has been using Fortran in teaching for more than 20 (25?) years. Nothing made me more enthusiastic than watching the whole building of TDF, BEI, PVI, etc. raising up from scratch with no locked rooms. That effect owes a lot to Fortran's clarity.

Reproducible research means an ability to take other people's code, understand 100% of it immediately (as a precise description of the algorithm), and modify it to continue the previous (unfinished) research. If we lose any part of this ability, SEP may become different from what it is now (moving into CWP's shadow?)

It is true that Fortran77 suffered from the lack of object-oriented features. Luckily, most of them appear in Fortran90, which is quite popular in the engineering world and flexible enough to solve all of our problems.

Learning how to solve problems is more important than including computer languages in your resume. (Don't our future employers think so?) At the learning stage, we need a language, which is object-oriented but yet easy for exchanging codes among different people. Am I wrong thinking that C++ doesn't meet both criteria?

Being as tired of the F90/C++ discussion as anyone else, I just wanted to present a student's point of view. No offense to religious C++ers.

Bob Clapp: I feel a clarification is necessary: The idea that GOON locks us into Fortran90 is wrong. The base of GOON is written in C++, we have an entire SEP77 style class also written in C++. All GOON offers as opposed to HCL is that it is also possible to write in Fortran90. Someone wanting to do all their work in C++ could still easily use GOON, the only current limitation imposed is that our current interaction with SEP90 is only written in Fortran90.

Stew Levin: Shuki raises an interesting example when he mentions his FPS work as being of little practical value two years after graduation. (Mine was three years ...) In my own case, the concepts behind the FPS programming (pipelining operations, vector functions, and asynchronous communication) still serve me in good stead, in support of David Lumley's point. Similarly with the NCUBE parallel computer that used to be in in the basement of computer science. I am happy though that for the most part the balance between geophysics and computing has shifted away from computing over the past decade at SEP.

REFERENCES

Biondi, B., Clapp, R., and Crawley, S., 1996, Seplib90: Seplib for 3-D prestack data: SEP–**92**, 343–364.

Clapp, R. G., and Crawley, S., 1996, SEPF90: SEP–**93**, 293–304.

Fomel, S., Crawley, S., and Clapp, R., 1996, A generic NMO program: SEP–**93**, 305–316.

Schroeder, J., and Schwab, M., 1996, HCL and regular data: SEP–**93**, 263–272.

Schwab, M., and Urdaneta, H., 1996, Hilbert Class Library: Ideas behind the design: SEP–**93**, 253–262.

Urdaneta, H., and Karrenbach, M., 1996, An IGF90 tutorial: SEP–**93**, 273–292.