Good Practices in Scientific Computing

Paul F. Dubois

Program for Climate Model Diagnosis and Intercomparison,
Lawrence Livermore National Laboratory

Presented to:

National Center for Atmospheric Research

March 20, 2000

You are a professional programmer.

This may not be your self-identification.

It may not be your chief professional qualification.

But:

- **You program computers.**
- **You get paid for it.**

This is nothing to be ashamed of, unless you do it badly.

You can't afford much time to this profession but you can't afford no time.

Attending this lecture counts.

My advice:

- **To avoid spending time on fads, ignore all new developments for a year or even two.**
- **Do not waste 10 seconds learning about hardware; your friends will do it for you.**
- **When hiring and promoting, recognize the real role the "gurus" play in enabling the other scientists around them.**

My essay “Ten Good Practices in Scientific Programming” listed some modern approaches to consider.

1. Organize for change.
2. Write a script instead of a compiled program.
3. Use a modern source-code management system.
4. Use “Design by Contract”.
5. Use Fortran 95, not Fortran 77.
6. Steer your calculation.
7. Use multiple languages (wisely).
8. Use C++ rather than C.
9. Use templates in C++.
10. Embrace your inner programmer.

(Computing in Science and Engineering, 1(1), Jan/Feb. 1999).

Organize around change.

Change is the dominant factor in scientific programming. Our programs change much more than commercial ones.

Quality is much more important to us than to regular programmers because we cannot easily distinguish bugs from bad science.

Change tends to degrade quality.

Object technology is finally making inroads into scientific programming.

- **Expression template technology enables Fortran-speed code that is nevertheless highly abstract.**
- **Object design can make mathematical libraries and "middleware" much more usable.**
- **Steering using an object-oriented scripting language is creating "plug and play" architectures for scientific programs, responding to the growing pressures for omnibus models.**

The Fortran programmer has a lot to gain from these ideas.

See M. Gray, R. M. Roberts, "Object-based Programming in Fortran 90", Computers in Physics 11(4) July/Aug. 1997.

Object-based Fortran course by me available at:

ftp://ftp.pyfortran.sourceforge.net.

Understanding abstraction vs. performance tradeoffs can help you decide about technologies to use.

The following explanation of how to achieve performance in C++ is important because it shows us something about Fortran and Java too.

C and Fortran programmers achieve efficiency by sacrificing abstraction.

These languages are actually only efficient if you agree to stick to one abstraction, the array.

The Fortran programmer usually uses collections of arrays to represent collections of objects. The programmer creates the abstraction only in the sense that these arrays are operated upon in a consistent manner.

Consider a typical approach to calculating quantities in cells.

Arrays temperature (n), density (n), volume (n) in common.

Functions mass (), pressure () can be array-valued in F95.

I can write very efficient statements using these arrays.

Whenever I change one array I might have to remember to change other arrays to maintain consistency.

Doing that may involve other data structures, such as those that specify models for these calculations.

In a large project, this means that everyone needs to be aware of these relationships.

The process does not scale.

In an object-oriented approach, the abstract concept of the cell collection can be expressed clearly.

```
class Cells {
public:
        const Field& volume() const;
        const Field& temperature() const;
        const Field& density() const;
        void set_pressure (const Field& p);
        void set_temperature (const Field& t);
        void set_density (const Field& d);
}
```

The methods are written with assertions and class invariant checks to prevent errors.

An author who wants to change the value of the temperature simply calls set_temperature and need not be aware of the “rules”.

The compiler can inline these methods so they are efficient.

So, those “function calls” are illusory.

The Field class is an abstract container.

Operations to index and set the values.

Operators f + g, f * g, sqrt (f), f.length (), etc.

These operations can have assertions that check for length consistency, etc. while debugging.

Field-specific methods can be written such as f.gradient().

This is wonderfully abstract, but a naive implementation will perform terribly.

```
Field f, g, h

f = g + (h * sqrt (g) + 1.0) / g;
```

With a naive implementation of Field:

- **Each operation such as sqrt (g) and g + (...) requires a temporary to hold the result.**
- **Each operation will contain a loop.**

When numerical calculations were first tried in C++, they were very slow due to this abstraction penalty.

It was not uncommon to see 10 times worse than Fortran, or more.

Profiling revealed intense activity in malloc / free.

Do we give in and write a loop?

```
for (int i=0; i < f.length(); ++i) {
    f[i] = g[i] + ...;
}
```

No, this is just writing Fortran++. The same possible errors in indexing, lengths, etc. reappear. You can't read it.

Expression templates are a recent development that solves this problem.

Abstract container classes such as Field can be written so that abstract expressions such as the ones above result in nearly the same object code as the hand-written loop when optimized.

Fortran speed is achieved without sacrificing abstraction.

Your C++ code can look just like a physics textbook.

While the details are complicated, the idea is easy.

f = g + h

is Field = Field + Field

We define the operator Field + Field so that it returns not a Field but a weird type that we might call "ResultOfAddingTwoFields".

If the next operation is a divide, we have "ResultOfAddingTwoFieldsThenDividing".

Finally, assignment is overridden to produce the loop with no temporaries.

In effect, we use templates to create new types in response to the program text, encoding an abstract syntax tree in the type system.

The normal compiler inlining and optimization eliminates all that machinery.

In fact, the code produced in the end will be very similar to that produced for a hand-written loop.

The expertise required to write all this machinery can be inherited so you don't even have to understand it very much.

There are four things to remember about this work.

1. C++ can be as efficient as Fortran but you need to use class libraries that have been written to exploit expression templates.
2. Templates were the key. Other object-oriented languages that don't have templates cannot solve this problem as far as we know. (.
3. If you try to use objects in Java or Fortran and overload operators you will suffer the same fate.
4. The ability of the compiler to inline and optimize is crucial for OO approaches.

When you reuse a component you increase reliability.

Reused components become more attractive targets for optimization efforts, and therefore reuse is a performance enhancer, too.

Reuse has many meanings:

- **Reuse in a new project**
- **Reuse in another part of the first project**
- **Reuse when the component survives a major upgrade**

Increasing reliability is MORE important for us than for business applications.

They know what the right answer is supposed to be.

We don't. It is hard for us to distinguish a bug from a modeling error.

Mathematical libraries are often cited as one example of successful software reuse. But if you look closely, things aren't so rosy:

Many user programs include hand-written algorithms, often of lesser quality.

The more high-level the routine, the less it is used, even though the sophistication it represents is higher.

Many users need substantial amounts of help to understand how to insert the routines in their programs.

The details of the process, such as leaking memory, error recovery, etc., are error-prone and hard to understand.

Nearly every scientific effort I have seen is not using mathematical software to the extent that it should.

The context in which the software is designed to be used simply won't recur often in functional languages. The higher the level of the routine, the less likely that conditions for its use will be satisfied.

If the language isn't safe, mistakes will happen more frequently.

Without garbage collection, Fortran and C have serious memory tracking problems, making it difficult to have optional output arrays, etc. The result is usually a clumsy interface.

It is easy to write something that looks like it is reusable:

Integrate real functions of one variable over finite intervals:

```
real function integral (f, a, b)
real a, b
external f
....some algorithm that calls f (x) for various x
integral = the answer
return
end
```

However, things are not really so simple.

Need to control the algorithm

Accuracy desired.

Maximum amount of work to be allowed.

What sort of accuracy? relative? absolute? mixed?

Order of method.

Need to have a context for the function evaluation

A significant function will need access to some state information or context in order to do its job.

Error control

Workspace and optional outputs

Here is a call to such an integrator in the Nag C Library.

```
    Nag_QuadProgress *progress;
    NagError fail;
    double a, b, ans, err, epa, epr;
    Integer npts;
    double (*f)(double);
...
    d01ajc (f, a, b, epa, epr, npts, &ans, &err, progress, &fail);
    if( fail.code != NE_NOERROR) { error handling}
```

The optional information structure returned in *progress* will be leaked memory if you call d01ajc again, so you have to add more coding for memory management.

This routine cannot pass any context for f.

More arguments, more types => more inertia

Here is the same concept in an object-oriented setting (Eiffel).

```
class ACLASS feature
    f (x: DOUBLE): DOUBLE is
        do
            Result := ...
        end;

Elsewhere:
        ac: ACLASS;
        g: GENERAL_ADAPTIVE_INTEGRATOR;
        a, b: DOUBLE;
...
        g.set_integrand (ac.~f(?));
        g.integrate (a, b);
```

The answer is in g.integral.

Any optional output accessed through other features of g.

This software is much easier to learn to use.

Reasonable defaults chosen for error tolerance, etc.

Exception raised if problem cannot be solved.

f has all its context in the object ac.

Environment tools can present the user with the other features to consider.

In the component model of programming, the problem is to find and correctly use existing software.

The user has to find the part in a catalog. When it is too hard to understand even basic usage, the part isn't used.

The part must fit into, or be easily transformed into, a context in the application code.

The features of object-oriented languages and environments work together to make this possible

Assertions become the advertisement in the catalog.

Inheritance allows adaptation to the environment, or extension by the user.

Design techniques can keep the basic interface minimal.

```
eigen: EIGENSYSTEM;
a: MATRIX;
a := ....
!! eigen.make (a);
print (eigen.eigenvectors);
print (eigen.eigenvalues);
```

We can exploit our skills and history in classification for the design of reusable components.

Here is an inheritance tree for integration:

```
INTEGRATOR
    FIXED_INTEGRATOR
        FINITE_FIXED_INTEGRATOR
        SEMI_INFINITE_FIXED_INTEGRATOR
    ADAPTIVE_INTEGRATOR
        FINITE_ADAPTIVE_INTEGRATOR
        SEMI_INFINITE_ADAPTIVE_INTEGRATOR
        ...
    MONTE_CARLO_INTEGRATOR
```

These ideas have been successful in practice.

EiffelMath, which I designed to encapsulate the NAG C library, has been used in financial models.[1]

Average number of arguments reduced from about 17 to about 2. Similar results for C++ versions of math libraries.

Many fewer bugs such as memory leaks or usage errors.

Much lower learning times.

Users reported quick and easy use in their codes.

1. Not work for LLNL.

Steering means giving the user full control of the application's assets using a real programming language.

Lotus 1-2-3 (1982)

What made it such a hit? You could program it.

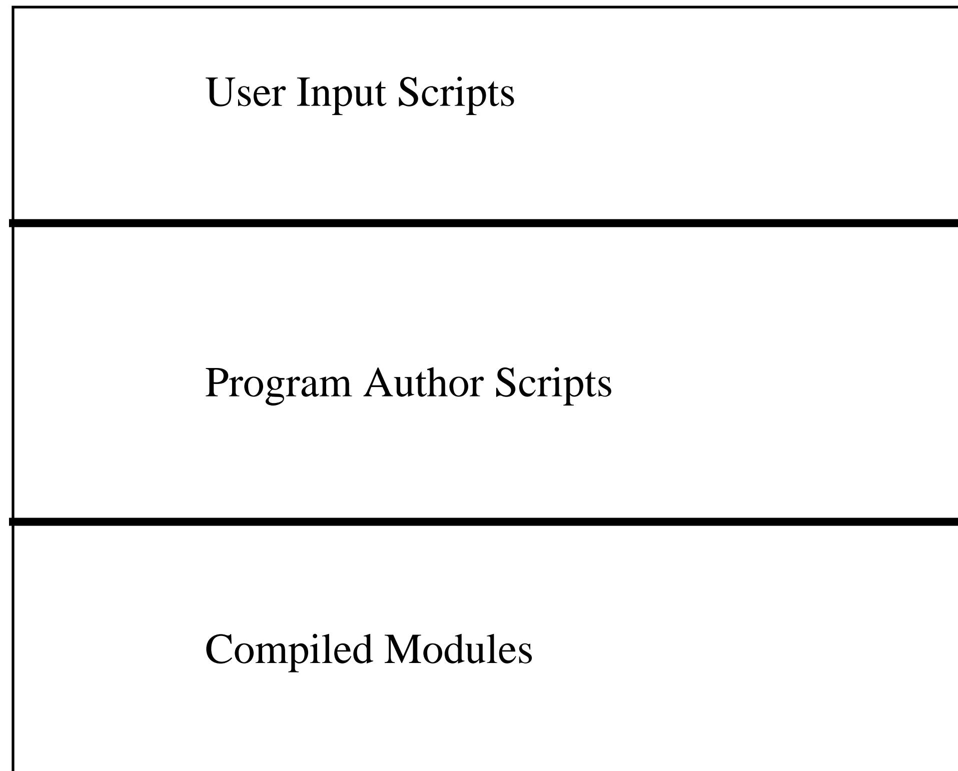
Basis (1984)

Used in over 200 codes, including several of LLNL' s largest. Fortran 95 -like array language interpreter.

Maple 6 (2000)

Symbolic algebra engine re-engineered to enhance extension by compiled code, link to NAG library.

The advent of steering has changed how programs are organized.



This concept greatly improves the productivity and power of both code groups and their users.

- **The creativity of the users is enabled.**
- **The code development group is no longer a bottleneck as users can add many capabilities by themselves.**
- **A graphical debugger is built in, and algorithms can be prototyped in the interpreter.**
- **We can respond to business opportunities quickly.**
  - -> **Two LLNL applications in medicine were first developed by using the Basis interface to teach an existing code to do something new.**

We use the Python language as a development environment.

We determined our requirements to be:

Availability

- **Portable to our exotic hardware.**
- **Free or cheap (important for University collaborators, programs).**

Language Characteristics

- **Interactive or script execution.**
- **Reasonable performance, especially numeric performance.**
- **Easy to learn and read.**
- **Able to interact with at least C, C++, and Fortran.**
- **Scalable to large projects.**

Leverage

- **Available libraries of existing components.**
- **An active user community.**
- **Possibility of distributed / parallel computing.**

Python meets all our major criteria well.

- Very small, easily learned language.
- Free (even if we make commercial products with it).
- Available on Windows, Mac, Unix, our weird computers.
- Rapidly growing international user community.
- Excellent object-oriented facilities and built-in types.
- Fast array facility.
- Scales extremely well to large projects.
- Vast libraries of components to use.

It is easy to add built-in functions and objects in C, C++ and Fortran.

SWIG produces interfaces to C and C++.

Pyfort (pyfortran.sourceforge.net) produces interfaces to routines in F77 or with implicit interfaces in F90.

Pyfort produces a Python module from a F90-like input syntax that describes your Fortran routine.

```
module remap
       subroutine pop_remap (num_links, noutput, input, output,
             remap_matrix, src_address,dst_address)
      integer:: num_links = size (remap_matrix, 2)
      integer noutput
      real*8 remap_matrix (3,num_links), input (num_links)
       integer src_address (num_links),  dst_address (num_links)
      real*8, intent (out):: output( num_links)
      end
end module remap
```

From Python you call it like this:

```
from remap import pop_remap
output = pop_remap (nout, in, remapm, src, dst)
```

Python has proven to be a powerful program-development environment.

EOF analysis of data in a climate dataset.