Should C Replace FORTRAN as the Language of Scientific Programming?

Linda Wharton
CSCI 5535 Fall 1995

Abstract

Anti-FORTRAN sentiment has recently become more prevalent. Where does the attitude originate? The most probable source is academia, where C and C++ are the languages of choice. Is there a fact based justification for the attitude? FORTRAN and C are evaluated to determine whether C is a better language than FORTRAN for scientific programming. The features of FORTRAN 77, FORTRAN 90, C and C++ are compared, and evaluated as to how well they meet the requirements of the scientific programming domain. FORTRAN was designed specifically for numerical programming, and thus better meets the requirements. Three algorithms in the scientific domain are coded in both FORTRAN and C. They are evaluated on performance, readability of the code and optimization potential. In all cases the FORTRAN implementations proved superior. Is there evidence to mandate that all upgrades and new development should be done in C, rather than FORTRAN? A good computer programmer can solve any given problem in any language, however it is best to code in the language specifically designed for the problem domain. In the case of scientific programming, that language is FORTRAN.

1 Introduction

In the computer arena related to scientific programming, a prevalent attitude seems to be that FORTRAN is obsolete, and C should be used as a replacement language. I am employed as a programmer that supports meteorological research. Most of the programming code I work with is written in FORTRAN. Within the course of my work, I continually encounter prejudice against FORTRAN. Where does this attitude originate? Is there a fact based justification for the attitude? Is there evidence to mandate that all upgrades and new development should be done in C, rather than FORTRAN?

This paper first identifies possible origins of anti-FORTRAN sentiment. Although highly subjective, this topic is worth examining. Several authors [Chapra88] [Joyner92] [Morgan92] [Moylam92] have commented on this subject. Their thoughts are presented, along with my own.

Once the possible origins of anti-FORTRAN sentiment have been identified, I proceed to investigate whether there is evidence to support that C is a better language than FORTRAN for scientific programming. Scientific programming is defined as programming that performs numerical computations on large amounts of data.

The features and capabilities of FORTRAN and C programming languages are compared. C and FORTRAN have been evaluated by many authors, both in a historical sense [Wagener80] [Kernighan88], and in a functional sense [Press92] [CSEP95]. Several authors have identified possible areas where changes could be made to C to make it work more like FORTRAN in numerical applications [Leary94].

The emphasis of the evaluation is on how well each language can implement and solve scientific problems. There are currently two standards for each language. The features of these standard languages, FORTRAN 77, FORTRAN 90, C and C++, are compared.

The languages are evaluated on their performance in the scientific programming arena. Three numeric tasks are coded and implemented in both FORTRAN and C. Each implementation is evaluated on the criteria of speed of execution, how readable the code is, and how well the program lends itself to optimization. Related work on performance evaluation is presented [Sullivan95] [Haney94].

Ultimately the case will be made as to whether C should be used to replace FORTRAN as the language of scientific programming.

2 Why Not FORTRAN?

When a scientist requests a program be written by a computer programmer, there are rarely any requirements beyond obtaining code that solves the problems efficiently. A good computer programmer can solve any given problem in any language. It is easier, however, and the code runs more efficiently, when a language that was specifically designed for the problem domain is used. Any good computer language should implement the concepts behind the language cleanly and simply, and express the concepts in as few words and constructs as possible [Joyner92]. "Unfortunately, many programmers have an almost emotional attachment to 'their' language and some go as far as to contend that all others are inferior" [Chapra88]. The question here is whether the concepts behind C are better suited to scientific programming than FORTRAN.

FORTRAN 77 is the lingua franca of numerical analysis [Sullivan95], yet a movement seems afoot to declare FORTRAN obsolete and replace it with C. In one Meteorology Research Laboratory, that sentiment is so strong that the Computer Facilities Division will not run code written in FORTRAN on their operational machines. Where and when did FORTRAN obtain the stigma of being an "old language" [Morgan92]?

I have noticed that prejudice against FORTRAN mainly surfaces during interaction with two groups of people. The first group is made up of programmers who are pursuing or have recently obtained Computer Science degrees. The second group contains the system administrators who maintain computers and operationally maintain and execute programs. I find that the second group is generally a subset of the first, leading me to believe that academia is the first place to look for the source of prejudice against FORTRAN.

Table 1. shows a random sample of computer language course offerings, obtained from the World Wide Web. It lists the languages taught by the Computer Science Departments at major universities, worldwide. Only 28 percent of the universities offer FORTRAN as a language, the majority of those in the United States. In contrast, 80 percent of all universities offer C or C++.

Why is C so popular? The fact that UNIX is the most widely used operating system has played a large part in the popularity of C. Since the UNIX operating system itself is written in C, any programmer that works in a UNIX environment is much better off if they are fluent in C. Once a language has built up a large user base, it develops an unstoppable momentum, generally becoming the current language of choice. "And, of course, each generation of programming educators teaches students its favorite language" [Moylam92]. Thus, even when FORTRAN is offered, it is not likely to be taken unless a student is specifically pursuing scientific computation or parallel processing.

Table 1. Computer Languages Taught by Universities

Country	FORTRAN Taught	University	Languages Taught
USA	No	Univ CO Boulder	C
USA	No	Stanford	C, C++, Assembly, LISP
USA	No	MIT	C, Assembly
USA	Yes	Univ CA Los Angeles	FORTRAN, C, Pascal
USA	Yes	Univ IL Urbana-Champaign	FORTRAN, C, C++
USA	Yes	Univ CA Berkeley	FORTRAN, C, Pascal, LISP
USA	No	Univ NM Albuquerque	C++
USA	No	Cal Tech	C, C++
USA	Yes	Purdue University	FORTRAN, C, C++
USA	Yes	SUNY Stoneybrook	FORTRAN, C, Pascal
USA	No	Univ NC Chapel Hill	Pascal
USA	No	Univ CA Davis	C, C++, Pascal
USA	Yes	Cal Poly SLO	FORTRAN
USA	No	Univ Southern Cal	Pascal, C++
USA	No	Cornell University	Pascal, C, C++
UK	No	Univ Newcastle Tyne	C++
UK	No	Aberystwyth Univ of Wales	Ada
UK	No	Univ Sheffield	C, C++, Eiffel, LISP
UK	No	Oxford University	Orwell, C, C++
Sweden	No	Uppsala University	Standard ML, C
Australia	No	University of Sydney	Pascal
Australia	No	Monash Univ, Melbourne	C
Canada	No	Univ Saskatchewan	Pascal, Object Pascal
Canada	No	Univ Manitoba	C, C++, Pascal
Chile	Yes	Univ Chile	FORTRAN, C, Pascal

Percentages of Universities Teaching FORTRAN : Overall - 28% USA Only - 40%
Percentages of Universities Teaching C or C++ : Overall - 80%

Source: Computer Science Department Course Listings for Universities on the World Wide Web

Once students graduate with a degree in Computer Science, they generally obtain jobs as either system administrators or programmers. System administrators program in the language required by the computer systems they maintain, and programmers generally prefer to use the languages they are most familiar with to develop new programs. Currently the popular language in both instances is C (or C++).

Problems arise when system administrators and programmers join companies whose main applications involve numerical computations, and consist of a large quantity of legacy code written in FORTRAN. The system administrators don't understand FORTRAN, so they would rather not be responsible for running and maintaining FORTRAN code, and programmers would much rather work with languages they know. Thus the "old" language, FORTRAN, is disparaged as not being as good as the newer language, C, that they are more familiar with. FORTRAN 77 also requires programs be written in a restricted format, left over from the days of punch cards. This lack of a user-friendly, free-form programming style in FORTRAN 77 may also be a leading cause of animosity.

Definitively identifying the origins of anti-FORTRAN sentiment would at best be speculation, and impossible to prove. P. J. Moylam notes that "loyalty to a language is very largely an emotional issue which is not subject to rational debate" [Moylam92]. Thus, this paper now leaves the search for the emotions behind prejudice against FORTRAN and concentrates on the search for empirical information as to which language, C or FORTRAN, better supports applications in the field of scientific programming.

3 Language Comparison

According to Bjarne Stroustrup, "a language does not support a [programming] technique if it takes exceptional effort of skill to write such programs" [Stroustrup88]. Do FORTRAN and C equally support the programming techniques required for scientific programming?

3.1 Language Requirements for Scientific Programming

Scientific programming and numerical methods have many requirements for execution on computers [Chapra88] [Vandergraft83]:

- The ability to efficiently and accurately perform computations on large systems of equations, requiring multi-dimensional arrays for storage - for example, there are weather modeling programs executing hourly that perform calculations on several gigabytes worth of data.

- The ability of the user to control the precision of the data - this allows the user to control the accuracy of the data, as well as optimize the code.

- The ability to perform computations using complex type data.

- Fast execution speed on atomic mathematical functions, such as exponentiation.

- Strong emphasis on clarity and readability of code.

The final requirement is one of the most critical. The mathematical calculations and formulas executed in scientific programming are usually very complicated to begin with. When formulas need to be broken apart to allow execution in a specific language, code validation becomes very difficult.

Now that the major requirements of scientific programming have been defined, both FORTRAN and C need to be evaluated as to whether or not they can meet the criteria of a scientific programming language.

3.2 History of FORTRAN

Originally, the only language available to solve numerical equations was assembly language. Then, in the 1950s, John Backus of IBM wrote a language that would convert high-level statements containing formulas into machine instructions. This language performed **for**mula **tran**slation, and was thus named FORTRAN [Wagener80]. Backus' goal was to produce a language that was simple to understand, but almost as efficient in execution as assembly language. He was successful, in allowing scientists and engineers to write efficient, accurate numerical programs without requiring them to be computer experts [Metcalf91].

The language evolved into many dialects before first being standardized in 1966. Unfortunately, not many developers adhered to that standard. In 1978, another standard was published, that encompassed the best features of the many dialects. That standard was FORTRAN 77.

The next upgrade to FORTRAN was set for 1988, but it was stalled in committee until 1990. At that time, FORTRAN 90 was announced as an additional standard, rather than a replacement to FORTRAN 77 [Metcalf90]. Part of the new standard was that FORTRAN 90 requires the syntax of code written to the FORTRAN77 standard not conflict with the new standard. The new capabilities of FORTRAN 90, over and above those of FORTRAN 77, will be discussed in section 3.4.

3.3 History of C

The C programming language was developed in the 1970s at Bell Laboratories as a system implementation language for the UNIX operating system. The language evolved from a compact language, BCPL, which had already been used to implement the OS6 operating system at Oxford [Kernighan88]. BCPL, and thus C, are part of the family of traditional procedural languages, but include routines that are "close to the machine", and aimed directly at systems programming [Ritchie93]. C uses library routines for input-output and interactions with the operating system, allowing the language to be portable.

The impetus behind C was to create a compact, portable language that would be used to develop an operating system for the minimal memory computers of the 1970s. Dennis Ritchie believes that goal was met. "C is quirky, flawed, and an enormous success...[and] satisfied a need for a system implementation language efficient enough to displace assembly language" [Ritchie93].

C++ was first introduced in the early 1980s. It is an object-oriented language that retains C as a subset. Bjarne Stroustrup designed C++ primarily so that the author would not have to program in assembler, C or other various modern high-level languages [Stroustrup94]. His main goal was to make writing good programs easier and more pleasant for the individual programmer. Stroustrup said he chose C as the basis for C++ because it was the "best systems programming language available" [King93].

Thus, both C and C++ were designed as systems programming languages. This fact does not exclude them from being used for scientific programming, but, as section 3.4 will show, the concentration on systems programming tasks resulted in the omission of features and capabilities required to efficiently handle numerical programming tasks.

3.4 A Comparison of Features: FORTRAN 77, FORTRAN 90, C and C++

Recall that scientific programming requires: the capability to handle complex type data, should allow the user to control the precision of data, and should provide efficient mathematical operators (such as exponentiation). Table 2. lists features that are desired in current programming languages, with the features pertinent to numerical computation in bold face.

Table 2. A Comparison of Language Features

Features / Language	FORTRAN 77	FORTRAN 90	C	C++
COMPLEX Data Type	**Yes**	**Yes**	**No**	**No**
User Control of Data Precision	**No**	**Yes**	**No**	**No**
Intrinsic Exponentiation Operator	**Yes**	**Yes**	**No**	**No**
Dynamic Memory Allocation	No	Yes	Yes	Yes
Free Source Form	No	Yes	Yes	Yes
User Defined Data Types	No	Yes	Yes	Yes
Data Structures	No	Yes	Yes	Yes
Pointers	No	Yes	Yes	Yes
Recursion	No	Yes	Yes	Yes
CASE / switch Statement	No	Yes	Yes	Yes
Numeric Polymorphism	Yes	Yes	No	No

COMPLEX Data Type

The first of the three features necessary in scientific programming is the ability to handle complex type data. In FORTRAN 77 and FORTRAN 90 the COMPLEX data type is intrinsic. The language automatically performs operations on complex data. C and C++ do not have complex as an intrinsic data type.

In C, a programmer can implement complex data using **typedef struct**. Problems occur, however, when there is a requirement to perform operations, such as addition, subtraction, multiplication and division, on the data. Functions can be written, but when they are used in place of operators in equations, the readability of the code is greatly diminished (see section 5.3).

C++ can implement complex data more easily than C because it contain classes and the ability to overload operators. Operator overloading on a COMPLEX data class will improve the readability of the code somewhat, but would still not be as efficient as an intrinsic data type.

User Control of Data Precision

The ability to control the precision of data has been around in a minimal sense since multiple data type were included in computer languages. The FORTRAN 77 standard shows only one size of INTEGER data, at two bytes, but offers REAL and DOUBLE PRECISION types for floating point data, using four and eight bytes respectively. Many of the FORTRAN 77 vendors, however, have extended the capability beyond the FORTRAN 77 standard. They offer two or four byte INTEGER data, four, eight or sixteen byte REAL data, and eight or sixteen byte COMPLEX data.

In C and C++ the intrinsic data types are **short**, **int**, and **long** for integers, and **float** and **double** for floating point numbers. Again, this offers minimal control of the precision of data. Kernighan & Ritchie C actually has a major problem with precision in the case of **float** type data. More details are forthcoming in section 3.6, but the gist of the problem is that *all* float variables are converted to type double before any calculations are performed [Press92], and the programmer has no control over the conversion.

All of the above options, however, only determine the size of storage available for data. It does not control the precision of data calculations. FORTRAN 90 has a "kind" selector that allows the programmer to explicitly specify the desired precision for each variable and intrinsic functions [Dubois93]. This come in very handy in these days of increasing data word size. Where REAL may indicate 16 bits on one machine, it may indicate 32 bits on another machine. The kind selector returns control of data precision to the programmer.

Intrinsic Exponentiation Operator

Both FORTRAN 77 and FORTRAN 90 offer exponentiation as a numeric intrinsic operator. Expressions like **(A + B)**4** compile in FORTRAN to be solved with only one add and two multiplies. For some reason, both C and C++ omitted the exponentiation operator, "perhaps the most galling insult to the scientific programmer" [Press92]. What Mr. Press forgets is that C was *not* written for scientific programming. It was written for *systems* programming! Apparently computer operating systems have no need to square data. While it is true that macros and functions can be written in C to perform exponentiation, there is no way they would be as efficient as a numeric intrinsic operator. Also, several pitfalls can arise when writing your own exponentiation macros or functions that can drastically reduce the efficiency of any program that calls them.

Additional Features

There are eight additional desired features listed in Table 2:

dynamic memory allocation	pointers
free source form	recursion
user defined data types	CASE / switch statement
data structures	numeric polymorphism

These features are in high demand as capabilities of current programming languages. FORTRAN 77 contains none of the eight features, which is not surprising, considering that the language dates back to the 1950s. FORTRAN 90 has incorporated all of the features, which now levels the playing field between FORTRAN and C. C and C++ are in close contention, since they only lack one feature; that of numerical polymorphism.

Numeric polymorphism is where a function is generic over all argument types [CSEP95]. For example, the result returned by a call to **COS(x)** in FORTRAN 90 is the appropriate value of kind SINGLE, DOUBLE, IEEE or P6, depending on the kind of variable **x**. In FORTRAN 90, all computational intrinsic functions are generic over all type kinds provided. C++ doesn't have numeric polymorphism, but it does have general polymorphism.

Dynamic memory allocation, pointers, data structures, user defined data types, recursion and the CASE / switch statement are also wonderful features to have in a language. Dynamic memory allocation allows users to manage their own memory, if they so desire. One of the great features of dynamic memory allocation in FORTRAN 90 is that the compiler *automatically* takes care of allocating the correct amount of memory. This is unlike C, which requires the programmer to get it right [Ellis93].

The optional free source format of programming used by FORTRAN 90 is going to be a big success with users. No longer will labels have to start in the first column, the continuation indicator field reside in the sixth column, and source code occupy the columns from 7 to 72. With the advent of FORTRAN 90, programmers now have the option to retain their old habits, if they prefer, or feel free to program in whatever column they like [Furzer92].

This section has determined that FORTRAN 90 and C have nearly the same features. The question now is, whether or not the manner in which the features are implemented is compatible with scientific programming.

3.5 Pros/Cons of Using FORTRAN for Numeric Programming

• **Pro** My research for this paper has reinforced my belief that FORTRAN is one of the best, if not the best, language for scientific and numeric programming. It meets all of the requirements put forth in section 3.1, supporting all of the features a language needs to function properly in the scientific programming domain. The negative aspects of FORTRAN as a language, such as a restricted coding format, lack of a CASE statement, and lack of dynamic memory allocation seem to have evaporated with the advent of FORTRAN 90. An additional advantage over C is that when memory is dynamically allocated in FORTRAN 90, the compiler allocates the correct amount automatically.

The FORTRAN language originated to translate formulas, and was designed to mimic the numerical realm it sought to streamline. The result was a high level language, with type checking, that allows a secure basis for solving scientific and numeric problems. A bonus is that the program code looks very much like the equations that are being implemented, including the array syntax.

A side benefit of FORTRAN being a high level language is that it is much more easily optimized. Higher-level language compilers can out perform lower-level compilers because they have more scope to decide how best to generate executable code.

• **Con** Thus far in my search, I have not found any reason why FORTRAN should not be used for numeric programming.

3.6 Pros/Cons of Using C for Numeric Programming

• **Pro** The main advantage of using C for numerical programming is the capability to easily interact with the operating system for display and efficiency analysis purposes. C also has the capability of data type conversion, via casting, whereas FORTRAN 90 only allows specific conversions [Press89]. This may, or may not, be an advantage.

The numerical programming community has recognized there are areas where C is weak in numerical processing and have addressed the problem by creating a new language, Numerical C. Numerical C is a superset of ANSI-standard C and contains additional language constructs geared toward mathematical-programming paradigms [Leary94]. The constructs enable the compiler to generate more efficient code by giving the compiler more information about the algorithm and by enforcing a canonical form on the input program.

• **Con** There are several major problems with using C, or C++, for numerical programming. Mainly, the languages were designed for systems programming, not high precision, efficient number crunching and mathematical calculations.

In my opinion, the biggest problem with using C for scientific programming is the way arrays were implemented. There is no efficient way to handle multi-dimensional arrays. Even the author, Dennis Ritchie, admits that C's treatment of arrays is suspect [Ritchie93]. This was due to the fact that systems programmers rarely need arrays of more than one dimension, so they were not designed into the language [Ross93].

The lack of type checking in C makes it harder to program without errors [Moylam92]. It also puts more responsibility on the programmer to have an intimate knowledge of data types and how they are stored and manipulated on the computer.

Using pointers in C drastically reduces the possibility of code optimization. This is due to aliasing, or the compiler not knowing how many pointers reference a location in memory [Jaeschke91].

No intrinsic complex data type. As mentioned before, this can be programmed around, but will reduce the efficiency and readability of the resulting program.

There are many pitfalls in C due to the syntax of the language [Baker92]. Suppose the goal was to divide by a de-referenced pointer: double *x; z=y**/***x; The problem is that the division symbol and pointer de-reference symbol together create the syntax for the start of a comment.

Another trap occurs when a programmer saves time by omitting spaces around the assignment operator: x=-10.0; The compiler interprets the sequence of "**=-**" as "**-=**". Thus instead of assigning the value -10.0 to x, the current value of x would be decremented by 10.0. A similar error will occur when assigning the value from a de-referenced pointer: x=*a would multiply x by the address of pointer a, rather than assigning the de-referenced value of a to x.

C does not adhere to the IEEE standards for data storage [Jervis92], thus it is not possible to reliably identify NaNs, or Not-a-Number values. NaNs occur on instances such as a division by zero. Whatever bit pattern is stored in a numerical variable is in a format which the computer cannot interpret as a number. Once a NaN occurs, it has the capability to propagate quickly throughout additional numerical computations. If you cannot identify that a NaN has occurred, you cannot correct the problem that caused it in the first place.

Kernighan & Ritchie implemented a bizarre practice in their version of C. A decision was made to automatically convert **float** variables to **double** before performing any operation, including arithmetic operations as well as passing as arguments to functions. All arithmetic is done in double precision. If a **float** variable receives the result of such an arithmetic operation, the high precision is immediately thrown away. This decision means that all of the real-number standard C library functions are of type **double**, and compute to double precision. The justification for these rules was "there’s nothing wrong with a little extra precision" [Press92]. The worst problem with this is that all conversion between **float** and **double** is done *automatically*, with no hope of shutting it off. This takes the choice of data precision away from the programmer. Fortunately, ANSI C dropped the practice for arithmetic operations.

The bottom line on using C for numerical programming is that, yes, it is possible, if the programmer wants to go to a lot of extra work.

4 Methods for Evaluating Languages

Now that FORTRAN and C have been compared, and their strengths and weaknesses with respect to scientific programming have been identified, the languages are evaluated on their performance. Three numeric tasks are implemented and tested in both FORTRAN and C:

- Gaussian Elimination with Back Substitution.
- 3x3 Smoothing across a 61x61 matrix.
- Newton's Method to extract a root of a fourth degree polynomial.

The specifics about these tasks, as well as the reasoning behind their selection, are defined in section 5.

Each implementation is evaluated on the criteria of speed of execution, how readable the code is, and how well the program lends itself to optimization.

Speed of execution is important, especially in situations where calculations are repetitive. Each of the compiled programs was executed ten times. The user time parameter returned from the UNIX function, **time**, was used to determine execution time. The collected times are averaged to determine an unbiased speed of execution.

Another measure for evaluation is how readable the code is. In an ideal situation, the code would look just like the mathematical equations the program is simulating. In the worst case, the code degenerates into expressions that are no longer meaningful in the context of the problem. A pure mathematical representation of the problem will be used for comparison, when applicable.

Recall, from section 3.6, that C code using pointers does not always lend itself to optimization. The presence or absence of pointers is evaluated as to how they might affect optimization.

5 Results of Evaluation

As mentioned in section 4, three programs have been chosen for evaluation. The Gaussian Elimination program was chosen for testing because it required the ability to work with two-dimensional arrays in a non-sequential manner. The smoothing program was chosen to work on larger two dimensional arrays. And, the Newton's method program was chosen to work with complex data, and compare intrinsic versus programmer implemented **complex** data types.

5.1 Gaussian Elimination with Back Substitution

Gaussian elimination is used to solve a system of simultaneous linear equations. In a system with n equations, the coefficients of the equations are stored in a matrix, **A** that is n x n, and the constant terms are stored in a vector, **b**, that is n-dimensional. The matrix equation that represents the system is: **A x = b**. Gaussian elimination will solve for the *n*-dimensional vector **x**.

The equation **A x = b** represents the set of *n* simultaneous equations:

$$
\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\
a_{21}x_1 + a_{12}x_2 + \ldots + a_{2n}x_n &= b_2 \\
&: \\
a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n &= b_n
\end{aligned}
$$

The goal is to modify the original equations into a system of linear equations that is upper triangular, and of the following form:

$$
\begin{aligned}
c_{11}x_1 + c_{12}x_2 + \ldots + c_{1n}x_n &= d_1 \\
c_{12}x_2 + \ldots + c_{2n}x_n &= d_2 \\
&: \\
c_{nn}x_n &= d_n
\end{aligned}
$$

At this point, $x_n = d_n/c_{nn}$. The remainder of the equations can be solved for **x** by back-substituting the known values into the equations above.

The algorithm to solve **A x = b** is as follows:

1. Read number of equations (**n**) and their coefficients (**A** and **b**)
2. Convert **A** to upper triangular form
3. Solve the new system by back-substitution
4. Print results

Step one is accomplished using data listed in the code at compile time. In actuality, this code would be implemented as a subroutine that would reside in a math library.

Step two uses subroutines/functions **pivot** and **tridiag** to manipulate the matrix.

Step three uses subroutine/function **back**.

Step four uses a print routine that displays matrices.

FORTRAN Implementation of Gaussian Elimination with Back Substitution [CSEP95]: program/subroutine descriptions (full code in Appendix A, Part 1.)

Program **testg**: determine the correct processing of the subroutines: pivot.f, triang.f, and back.f. The subroutines determine the solution to a series of simultaneous equations.

Declaration and initialization of variable **matrix**:

```
      INTEGER IMAX, JMAX
      PARAMETER (IMAX = 5, JMAX = 6)
      REAL  matrix(IMAX, JMAX)
      REAL  solvec(IMAX)
      INTEGER i, j, n

      DATA ( (matrix(i,j), j = 1, JMAX), i = 1, IMAX)
     +    /4.0, 2.0,-2.5, 4.0, 1.0,-0.1,
     +     3.0,-1.0, 0.5, 0.0, 5.0, 1.5,
     +     1.5, 2.5, 3.0,-1.0, 5.0, 3.0,
     +     1.0, 2.0,-1.0, 2.0,-1.0, 0.1,
     +     5.0, 3.0, 3.0, 2.0,10.0, 4.0/
```

Subroutine **tridiag**: performs the lower decomposition of an input matrix.

Subroutine **back**: computes a solution vector from an augmented matrix that has already undergone lower decomposition.

Subroutine **pivot**: determines the largest value in the first column of an augmented matrix and moves the row with the largest value in the first column to first row. The process is then repeated for the successive rows and columns, and for each iteration, the column position and the row position are decremented by 1 (That is, 1st Column-1st Row then 2nd Column-2nd Row, then 3rd Column-3rd Row, etc.)

```
      SUBROUTINE pivot(matrix, n)
      INTEGER i, j, k, n
      REAL  matrix(n, n + 1), maxval, tempval

      do 10, j = 1, n
        maxval = matrix(j,j)
        do 20, i = j + 1, n
          if ( maxval .lt. matrix(i,j) ) then
             maxval = matrix(i,j)
             do 30, k = 1, n + 1
                tempval = matrix(i,k)
                matrix(i,k) = matrix(j, k)
                matrix(j, k) = tempval
 30          continue
          endif
 20     continue
 10   continue
      end
```

As you can see from subroutine **pivot**, the FORTRAN implementation is very straightforward. Both **matrix** and **n** are passed into the routine, allowing the code to manipulate the values in **matrix**. The array subscripts are manipulated in the same manner as they would be if the problem was being manually solved on paper.

C Implementation of Gaussian Elimination with Back Substitution - no parameters passed [CSEP95]: function descriptions (full code in Appendix A, Part 2.)

Declaration and initialization of variable **matrix**:

```
#include <stdio.h>
#define IMAX 5
#define JMAX 6

float matrix[IMAX][JMAX] = {
                    { 4.0, 2.0,-2.5, 4.0, 1.0,-0.1 },
                    { 3.0,-1.0, 0.5, 0.0, 5.0, 1.5 },
                    { 1.5, 2.5, 3.0,-1.0, 5.0, 3.0 },
                    { 1.0, 2.0,-1.0, 2.0,-1.0, 0.1 },
                    { 5.0, 3.0, 3.0, 2.0,10.0, 4.0 }
                      };
float solvec[IMAX]      = { 0.0, 0.0, 0.0, 0.0, 0.0 };

main()
```

Program **main()**: same function as program **testg** in FORTRAN implementation.
Function **pivot**: same function as subroutine **pivot** in FORTRAN implementation.

```
void pivot()
{
      int i, j, k;
      float maxval, tempval;

      for ( j = 0; j < IMAX; j++) {
            maxval = matrix[j][j];
            for ( i = ( j + 1 ); i < IMAX; i++) {
                  if ( maxval < matrix[i][j] ) {
                        maxval = matrix[i][j];
                        for( k = 0; k <= IMAX; k++) {
                              tempval = matrix[i][k];
                              matrix[i][k] = matrix[j][k];
                              matrix[j][k] = tempval;
                        }
                  }
            }
      }
}
```

Function **tridiag**: same function as subroutine **tridiag** in FORTRAN implementation.
Function **back**: same function as subroutine **back** in FORTRAN implementation.

In the C implementation, the array **matrix** is declared globally, as shown above. The function **pivot()** is a void function that receives no parameters and returns no value. Instead, it makes use of the fact that **matrix** is a global variable, allowing the array to be accessed with the array notation type syntax shown above.

The global declaration of **matrix** is not realistic in real world applications. In a realistic application, the function **pivot()** would be a generic program, residing in a library. It would have no information of what name the matrix variable was declared as, nor the name of the variable that defines the array size. To be implemented correctly, the function **pivot()** would need to be called with a pointer to the matrix variable and a second variable indicating the size of the matrix.

C Implementation of Gaussian Elimination with Back Substitution - parameters passed: function description (full code in Appendix A, Part 3.)

Function **pivot**: same function as subroutine **pivot** in C implementation, modified to receive data as parameters rather than globally.

```
void pivot(float *pm, int imx, int jmx)
{
        int i, j, k;
        float maxval, tempval;

        for ( j = 0; j < imx; j++) {
          maxval = *(pm + (j*jmx) + j);
          for ( i = ( j + 1 ); i < imx; i++) {
            if ( maxval < *(pm+(i*jmx)+j) ) {
              maxval = *(pm+(i*jmx)+j);
              for( k = 0; k <= imx; k++) {
                tempval = *(pm+(i*jmx)+k);
                *(pm+(i*jmx)+k) = *(pm+(j*jmx)+k);
                *(pm+(j*jmx)+k) = tempval;
              }
            }
          }
        }
}
```

Evaluation:

Performance

The average execution time of each program is as follows:

FORTRAN - avg = .039u
C with global variables - avg = .061u
C with parameters passed - avg = .070u

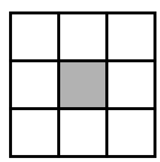
On average, the FORTRAN code ran faster than the C code. The time to execute the C code raised slightly when the array was passed as a parameter. This seems to indicate that FORTRAN handles arrays more efficiently. Full testing results are in Appendix B.

Readability
Both the FORTRAN code and the C code without parameters have readability representative of the original mathematical expression. The C code with parameters loses all recognizable contact with the original array subscript notation. This is due to the inability of C to pass multi-dimensional arrays as parameters.

Optimization Potential
The FORTRAN code has a high potential for optimization, as does all FORTRAN code. The C code with parameters is suspect as to how well it will optimize. The array has to be passed as an explicit pointer, which could compromise optimization efforts. The C code without parameters should optimize slightly better than the C code with parameters due to the fact that the globally declared array is accessed by its own name rather than an alias.

5.2 3x3 Smoothing across a 61x61 matrix.

The goal in a smoothing routine is to avoid drastic jumps in the values of neighboring data. The method used is to read in one array, and output the smoothed data to another array. The mechanics of calculation are as follows:



The values in the nine squares are averaged. The averaged value is then assigned to the output array in the location of the shaded square. The grid points on the edge of the array are directly transferred to the output array without being averaged. These points are intentionally ignored because there are not a sufficient number of grid points to calculate an average.

This program offers an opportunity to illustrate how FORTRAN 90 implements the capability of numeric polymorphism, which is the ability for one subroutine to function generically on any type of data. To accomplish this, the following interface block would be added:

```
INTERFACE SMOOTH                           ! SMOOTH is the generic name

 INTEGER FUNCTION SMOOTH_INT(AA)           ! for procedures
  INTEGER :: AA(:,:)                       !    SMOOTH_INT
 END FUNCTION SMOOTH_INT                   !    SMOOTH_SINGLE

 INTEGER FUNCTION SMOOTH_SINGLE(AA)
  REAL(SINGLE) :: AA(:,:)                  ! AA is an assumed shape two-
 END FUNCTION SMOOTH_SINGLE                ! dimensional array in each case.

END INTERFACE
```

Programmers would access the routines via the generic name **smooth**, using either INTEGER or SINGLE type data. The interface transparently sends calls with INTEGER data to subroutine **smooth_int**, and sends calls with SINGLE data to the subroutine **smooth_single**.

FORTRAN Implementation of 3x3 Smoothing [CSEP95]:
program/subroutine descriptions (full code in Appendix A, Part 4.)

Program **testsm** : program to test the subroutine smooth.f, which performs a 3 x 3 averaging operation on an arbitrary sized matrix.

Subroutine **smooth**:computes a 3x3 average from an input matrix. Note that, this subroutine does not average the values at the edges of the matrix.

```
      SUBROUTINE smooth(output, input, n, m)
      INTEGER i, j, n, m
      REAL upper, mid, lower
      REAL output(n,m), input(n,m)

      do 10, i = 2, n - 1
       do 20, j = 2, m - 1
          upper = input(i-1,j-1)+input(i-1,j)+input(i-1,j+1)
          mid   = input(i  ,j-1)+input(i  ,j)+input(i  ,j+1)
          lower = input(i+1,j-1)+input(i+1,j)+input(i+1,j+1)

          output(i,j) = (upper + mid + lower) / 9.0
20      continue
10     continue

      do 30, i = 1, n
          output(i,1) = input(i,1)
          output(i,m) = input(i,m)
30     continue

      do 40, j = 2, m-1
          output(1,j) = input(1,j)
          output(n,j) = input(n,j)
40     continue

      end
```

This subroutine is straightforward. It executes the smoothing code on the interior of the array, storing the results into the output array. The next steps then copy the data on the perimeter of the input array to the output array. FORTRAN allows the 61x61 arrays to be easily passed in and out of the subroutine as parameters.

C Implementation of 3x3 Smoothing:
program/function descriptions (full code in Appendix A, Part 5.)

Program **main**(): same function as program **testsm** in FORTRAN implementation.
Function **smooth**: same function as subroutine **smooth** in FORTRAN implementation.

```
void smooth(void)
{
        int i, j;
        float upper, mid, lower;

        for ( i = 1; i < (IMAX - 1); i++) {
           for ( j = 1; j < (JMAX - 1); j++) {
               upper = input[i-1][j-1] + input[i-1][j] + input[i-1][j+1];
               mid   = input[i  ][j-1] + input[i  ][j] + input[i  ][j+1];
               lower = input[i+1][j-1] + input[i+1][j] + input[i+1][j+1];

               output[i][j] = ( upper + mid + lower ) / 9.0;
           }
        }
        for ( i = 0; i < IMAX; i++) {
           output[i][0     ] = input[i][0     ];
           output[i][JMAX-1] = input[i][JMAX-1];
        }
        for ( j = 0; j < JMAX; j++) {
           output[0     ][j] = input[0     ][j];
           output[IMAX-1][j] = input[IMAX-1][j];
        }
}
```

Evaluation:

Performance

The average execution time of each program is as follows:

FORTRAN - avg = .033u
C - avg = .074u

Again, on average, the FORTRAN code ran faster than the C code. Full testing results are in Appendix B.

Readability

The FORTRAN code and C code both have readability representative of the original mathematical expression.

Optimization Potential

The FORTRAN code has a high potential for optimization. The C code should optimize unambiguously due to the fact that the globally declared array is accessed by name rather than by an alias.

5.3 Newton's Method to extract a root of a fourth degree polynomial

This program uses Newton's method to systematically search for a root of a fourth degree polynomial. Such a root may be a complex number so **complex** data elements are employed. The method starts with an initial guess to the root, then uses Newton's formula to generate the next, and hopefully better, approximation to the root. The program verifies that the method converges, terminating execution if it starts to diverge.

FORTRAN Implementation of Newton's Method [Wagener80]:
program description (full code in Appendix A, Part 6.)

Program **newton** : extract a root of a fourth degree polynomial by using Newton's method.

```
      program newton

      integer n, loop
      real a,b,c,d,e
      complex x, root, f, df

      data n, x /1, (1.0,1.0)/

      a = 1.0
      b = 0.0
      c = 1.0
      d = 0.0
      e = -1.0
      loop = 1

      do while (loop .eq. 1)
        f = (((a*x+b)*x+c)*x+d)*x+e
        df = ((4*a*x+3*b)*x+2*c)*x+d
        if (abs(df) .lt. 0.001) then
          print *, 'derivative too small -- terminate search'
          loop = 0
          goto 900
        endif
        root = x-f/df
        print *, root
        if (abs(root-x)/abs(root) .lt. 0.00001) then
          print *, 'root found'
          loop = 0
          goto 900
        endif
        x = root
        n = n + 1
        if (n .gt. 40) then
          print *, 'too many iterations -- terminate search'
          loop = 0
          goto 900
        endif
      enddo
900   continue
      end
```

Of particular note in the above program are the two lines in which the function and derivative are evaluated:

```
f = (((a*x+b)*x+c)*x+d)*x+e
df = ((4*a*x+3*b)*x+2*c)*x+d
```

Because FORTRAN has COMPLEX as an intrinsic data type, and implements numeric polymorphism on all intrinsic data, the above two lines are all that is needed to ensure proper evaluation of the equations. In contrast, the next program shows what is required to evaluate the same formula in C, which does not have **complex** as an intrinsic data type.

C Implementation of Newton's Method:
program/function descriptions (full code in Appendix A, Part 7.)

Program **main()**: same function as program **newton** in FORTRAN implementation. calls functions **fcn_eva**l and **deriv_eval** to evaluate the the above equations.

```
 #include <stdio.h>
#include <math.h>

struct FCOMPLEX {
  float r;
  float i;
};
typedef struct FCOMPLEX fcomplex;

        fcomplex fcn_eval(float, float, float, float, float, fcomplex);
        fcomplex deriv_eval(float, float, float, float, float, fcomplex);
```

```
main()
{
        int n;
        float a,b,c,d,e;
        fcomplex x, root, f, df;

        while (1) {
          f = fcn_eval(a,b,c,d,e,x);
          df = deriv_eval(a,b,c,d,e,x);
          if (Cabs(df) < 0.001) {
            printf("derivative too small -- terminate search\n");
            break;
          }
          root = Calc_root(x,f,df);
          printf("(%f,%f)\n",x.r,x.i);
          if (Cabs(Csub(root,x))/Cabs(root) < 0.00001) {
            printf("root found\n");
            break;
          }
          x = root;
          n = n + 1;
          if (n > 40) {
            printf("too many iterations -- terminate search\n");
            break;
          }
```

Note that evaluating the equations containing complex variables require function calls for operations such as addition, subtraction, and multiplication.

C Implementation of Newton's Method (continued):
program/function descriptions (full code in Appendix A, Part 7.)

Function **fcn_eval:** evaluates the equation f = (((a*x+b)*x+c)*x+d)*x+e.
Function **deriv_eval:** evaluates the equation df = ((4*a*x+3*b)*x+2*c)*x+d.

```
fcomplex fcn_eval(float a, float b, float c, float d, float e, fcomplex x)
{
      fcomplex f2c,ans;

/*      f = (((a*x+b)*x+c)*x+d)*x+e;  */
      f2c.i = 0.0;
      f2c.r = a;
      ans = Cmul(f2c,x);
      f2c.r = b;
      ans = Cadd(f2c,ans);
      ans = Cmul(ans,x);
      f2c.r = c;
      ans = Cadd(f2c,ans);
      ans = Cmul(ans,x);
      f2c.r = d;
      ans = Cadd(f2c,ans);
      ans = Cmul(ans,x);
      f2c.r = e;
      ans = Cadd(f2c,ans);
      return ans;
}
```

```
fcomplex deriv_eval(float a, float b, float c, float d, float e, fcomplex x)
{
      fcomplex f2c,ans;

/*     df = ((4*a*x+3*b)*x+2*c)*x+d; */
      f2c.i = 0.0;
      f2c.r = 4*a;
      ans = Cmul(f2c,x);
      f2c.r = 3*b;
      ans = Cadd(f2c,ans);
      ans = Cmul(ans,x);
      f2c.r = 2*c;
      ans = Cadd(f2c,ans);
      ans = Cmul(ans,x);
      f2c.r = d;
      ans = Cadd(f2c,ans);
      return ans;
}
```

Unfortunately, the above two functions bear no direct resemblance to the equations they are trying to evaluate.

Evaluation:

Performance

The average execution time of each program is as follows:

FORTRAN - avg = .026u
C - avg = .047u

Due to the **complex** data type not being intrinsic in C, the FORTRAN code ran almost twice as fast as the C code. Full testing results are in Appendix B.

Readability

The FORTRAN code has readability representative of the original mathematical expression. The C code is very difficult to understand. It is not immediately clear what the functions are trying to evaluate.

Optimization Potential

There should be no problem optimizing the code in either of the languages, since there are no pointers or arrays involved in the calculations.

6 Related Work

The Computational Science Education Project evaluated FORTRAN 77, FORTRAN 90, C and C++ and ranked them according to numerical robustness [CSEP95]. FORTRAN 90 was ranked first, based on having numeric polymorphism, real "kind" type parameterization and decimal precision selection. FORTRAN 77 ranked second due to its intrinsic support of complex variables, and C++ nudged out C for third place due to its capabilities in the general area of polymorphism.

Scott Haney broached the subject of whether or not C++ is fast enough for scientific computing. He evaluated C, C++ with linear addressing, and C++ with indirect addressing against FORTRAN 77 in three tasks:

- multiplication of 100 x 100 real matrices
- multiplication of 100 x 100 complex matrices
- computing the inductances for a series of square coils

These calculations were performed on six different platforms. Results showed that C was closer to the speed of FORTRAN 77 than C++, and the C++ results ranged from 150 - 700% slower than FORTRAN 77. Only if you execute C code on a Cray C90 can you hope to match FORTRAN 77 [Haney94].

Ben Zorn and Stephen Sullivan evaluated "Numerical Analysis Using Non-Procedural Paradigms" [Sullivan95]. They benchmarked Gaussian elimination on 1000 x 1000 sparse matrices in, among other languages, C++, FORTRAN 77 and FORTRAN 90. The results showed that FORTRAN 77 was slightly faster than C++ with static array allocation.

FORTRAN 90 was somewhat slower, which they attributed to relatively new compilers that have not perfected their optimization techniques yet.

7 Conclusion

Now that all of the facts have been gathered, and examples have been presented, let's evaluate the results and answer the questions that were posed.

First, where does the anti-FORTRAN attitude originate? I believe its roots are in both academia and the mass mind. C and, more recently, C++ have been the language of choice in universities since the late 1980s. The fact that C is taught in 80 percent of the universities sampled, versus FORTRAN currently taught at only 28 percent of the universities, bears this out. Programmers prefer to work in languages they are familiar with. Thus, once students graduate into the world marketplace, it stands to reason they would prefer to use a language they know, as well as express their opinions and influence people in the work force toward that language. This is the mass mind factor.

Second, is there a fact based justification for the attitude? I found none in the research and evaluations presented in this paper. Several authors who have compared FORTRAN and C in the area of scientific programming hold the same belief. While FORTRAN 77 still requires a rigid, static style of programming, FORTRAN 90 provides an upgraded alternative that is more user friendly.

Several authors have voiced their opinion on whether or not C should be used for scientific programming:

> "Better languages [than C] exist for higher level functions such as scientific work" [Joyner92].
>
> "Our message to C users is: look at FORTRAN again. It's still the best there is for engineering and science" [Morgan92].
>
> "Scientific programming in C may someday become a bed of roses; for now, watch out for the thorns!"[Press92].

And finally, is there evidence to mandate that all upgrades and new development should be done in C, rather than FORTRAN? A good computer programmer can solve any given problem in any language, however it is best to code in the language specifically designed for the problem domain. Oftentimes the language in which an application is implemented is dictated by issues not related to the problem being solved. Too much emphasis is being placed on what language programmers and system administrators are most comfortable with, and not what language is best for solving a given problem.

Should C replace FORTRAN as the language in scientific programming? NO!!!

References

[Baker92] Baker, L., *C Mathematical Function Handbook*, McGraw-Hill, New York, NY, 1992.

[CSEP95] Computational Science Education Project, *Fortran 90 and Computational Science*, http://csep1.phy.ornl.gov/csep.html,1995.

[Chapra88] Chapra, S. C., Canale, R. P., *Numerical Methods for Engineers*, Second Edition, McGraw-Hill, New York, NY, 1988.

[Dubois93] Dubois, P. F., Busby, L., Portable, Powerful FORTRAN Programs, *Computers in Physics*, 7, 1 (January 1993) 38-43.

[Ellis93] Ellis, T. M. R., *Fortran 77 Programming with an Introduction to the Fortran 90 Standard,* Second Edition, Addison-Wesley, Reading, MA, 1993.

[Furzer92] Furzer, I., Fortran 90: The View from Academia, *Chemical Engineer*, 527, (September 1992) 26.

[Haney94] Haney, S. W., Is C++ Fast Enough for Scientific Computing?, *Computers in Physics*, 8, 6, (November 1994) 690.

[Jaeschke91] Jaeschke, R., Standard C: A Status Report, *Dr. Dobbs Journal of Software Tools for the Professional Programmer*, 16, 8 (August 1991), 16.

[Jervis92] Jervis, Robert, Numerical Extensions to C, *Dr. Dobbs Journal of Software Tools for the Professional Programmer*, 17, 8 (August 1992), 26.

[Joyner92] Joyner, I., C++?? A Critique of C++, Second Edition, Unisys, North Ryde, Australia, 1992.

[Kernighan88] Kernighan, B. W., Ritchie, D. M., *The C Programming Language*, Second Edition, Prentice-Hall, Englewood Cliffs, NY, 1988.

[King93] King, K. N., The History of Programming Languages, *Dr. Dobbs Journal of SoftwareTools for the Professional Programmer*, 18, 8 (August 1993), 18.

[Leary94] Leary, K., Numerical C and DSP, *Dr. Dobbs Journal of Software Tools for the Professional Programmer*, 19, 8 (August 1994), 18.

[[Metcalf90] Metcalf, M., Reid, J., *Fortran 90 Explained*, Oxford University Press, New York, NY,1990.

[Metcalf91] Metcalf, M., Reid, J., Whither Fortran?, *Fortran Forum*, 10, 2, (July 91) 18.

[Morgan92] Morgan, N., Fortran 90: the end of schizophrenic engineers?, *Chemical Engineer*, 527, (September 1992) 25.

[Moylam92] Moylam, P. J., A Case Against C, Technical Report EE9240, Department of Electronic and Computer Engineering, University of Newcastle, Australia, July 1992.

[Press89] Press, W. H., Flannery, B. P., Teukolsky, S. A., Vetterling, W. T, *Numerical Recipes: The Art of Scientific Computing (FORTRAN Version)*, Cambridge University Press, Cambridge, U.K. 1989.

[Press92] Press, W. H., Flannery, B. P., Teukolsky, S. A., Vetterling, W. T, *Numerical Recipes in C*, Second Edition, Cambridge University Press, Cambridge, U.K. 1992.

[Ritchie93] Ritchie, D. M., The Development of the C Language, *Second ACM History of Programming Languages Conference* (April 1993).

[Ross93] Ross, John W., Calling C Functions with Variably Dimensioned Arrays, *Dr. Dobbs Journal of SoftwareTools for the Professional Programmer*, 18, 8 (August 1993), 52.

[Stroustrup88] Stroustrup, B., What is Object-Oriented Programming?, *IEEE Software*, 5, 3, (May 1988), 10.

[Stroustrup94] Stroustrup, B., *The C++ Programming Language*, Second Edition, Addison-Wesley, Reading, MA, 1994.

[Sullivan95] Sullivan, Stephen J. and Zorn, Benjamin G., Numerical Analysis Using Non-Procedural Paradigms, *ACM Transactions on Mathematical Software*, 21, 3 (September 1995), 267.

[Vandergraft83] Vandergraft, J. S., *Introduction to Numerical Computations*, Second Edition, Academic Press, Inc., New York, NY, 1983.

[Wagener80] Wagener, J. L., *Fortran 77 Principles of Programming*, Wiley & Sons, New York, NY, 1980.