CRITERIA FOR TRANSPORTABLE ALGOL LIBRARIES

Pieter W. Hemker

ABSTRACT

A rather comprehensive numerical software library (NUMAL {3}) was transported from a Philips EL-X8 computer to a CDC CYBER system. The experiences justify the following conclusion:

If (1) we use a well-defined language (e.g., ALGOL 60 or ALGOL 68), if (2) we construct well-programmed software in that language, if (3) we have a good compiler and if (4) the computer/compiler has well-designed arithmetic properties, then the transportability problem scarcely exists. This statement can also be put the other way. The requirement of orthogonality of the conditions (1) - (4) determines what can be considered as a decent programming language, a good compiler, good programming and well-behaved arithmetic.

For instance, good programming should not make use (perhaps at the cost of some efficiency) of idiosyncratic features of a language dialect, of a particular compiler or of a particular kind of machine arithmetic. From this abstract point of view, a number of useful properties of a well-structured portable software library are mentioned.

INTRODUCTION

In 1973, the Mathematical Centre in Amsterdam had to transport its numerical software library from a Philips EL-X8 computer to a CDC CYBER system. The library was written in ALGOL 60 and at that time it consisted of about 250 procedures. During the construction of the library, which started in the early sixties, a tradition in the use of ALGOL 60 was developed. This means that only correct ALGOL 60 in the sense of the Revised Report {2} was used and that features that were not clearly defined in the report were avoided as much as possible.

On the EL-X8 a reliable, efficient and rather complete ALGOL 60 compiler was available and the library was partly incorporated into the ALGOL 60-oriented running system.

When, in 1973, the library had to be adapted to the CDC CYBER system, the change-over took only a few months. In fact, it appeared that no essential changes in the code had to be made to adapt it for the CDC ALGOL 60 version 3 compiler and most of the transport work could be done automatically. Only in a few exceptional cases, some strange properties of the CDC machine arithmetic caused a procedure to fail in its new environment.

Since 1973 the numerical library NUMAL has been extended considerably and now it consists of about 450 specialist-oriented as well as general purpose routines in the field of numerical mathematics.

Restrictions of ALGOL 60

The easy transport from one machine to the other was mainly due to the strict use of ALGOL 60 and to the machine-independent way of programming. The latter means, e.g., that the relative accuracy of a

floating-point number has to be mentioned explicitly in an input parameter in the calling sequence of a procedure. This kind of machine independence, however, was not possible for all procedures: in those programs where representations of floating-point constants are necessary, they are given to only about 15 decimal places.

The use of ALGOL 60 according to the Revised Report has some apparent disadvantages:

(1) one cannot use input/output statements;

(2) communication with mass storage is impossible;

(3) double precision arithmetic is not available as a language feature.

The first two points force us to keep the library completely I/O free and to exclude all procedures that require mass storage. On the other hand double-precision arithmetic procedures have been introduced. Because of the favourable arithmetical properties of the EL-X8 computer, it was possible to write the elementary double-precision operations +, -, *, / in ALGOL 60 {1} for the EL-X8; for the CDC CYBER computer, however, these elementary procedures had to be written in machine code. These machine-dependent double-precision procedures were implemented because double precision was considered to be indispensable in several applications and the procedures only cause machine-dependence in a clearly distinguishable part of the library.

Structure

Besides the "machine independent" use of ALGOL 60, the library NUMAL has two other characteristics: it is an integrated library and it has a modular structure. By *integrated library* we mean that it is

not merely a collection of tested and documented routines, but a coherent structure in which the different parts gear into each other. The main lines of interconnection between the various parts are given in Figure 1.

By *modular structure* we denote that also on a much smaller scale programs have been divided into pieces, which can be used separately in different places. Thus, in principle, in any two places where the same effect is required, it is effectuated by only a single piece of code.

Compiler Dependence

Though the library takes into account the restrictions that are imposed by the use of strict ALGOL 60, nevertheless a number of problems arise when the library is taken to a new compiler.

We give a short list of problems that may arise:

(1) In general a different character representation and ALGOL symbol representation are used. A very simple program can take care of this conversion, but it is also possible that an incomplete character set is used; e.g., lower case letters are missing in the CDC character set.

(2) Even good compilers have some restrictions. It is wise not to use all the ALGOL 60 features that are permitted by the Revised Report; e.g., most ALGOL 60 compilers do not handle "own dynamic arrays." However, when a reasonably complete compiler is available, the restrictions imposed by it introduce only minor problems for numerical programs and they are easily eliminated.

(3) It is not defined by the Revised Report how independent compilation of procedures should be handled. In order to adapt the texts of the original procedures for use in the CDC system, all externally declared procedures needed to be referenced by a code declaration inside the procedure body.

These points all have to do with the peculiarities of a particular compiler, whereas the library was constructed with no particular compiler in mind. The only thing we can do - if we want the library to run on another machine - is to find (or to insist on the construction of) a compiler with a negligible number of anomalies.

Arithmetic

Two more points have to be kept in mind when we consider the transportation of a library to a new environment (i.e., computer + compiler), viz., machine arithmetic and elementary functions.

When an algorithm is coded independently of a particular machine environment, a guarantee with respect to its performance can only be given under certain assumptions on the machine arithmetic. Weak arithmetic can spoil a sound algorithm. For instance, on CDC, a program failed because, using CDC arithmetic, one can obtain real numbers a and b such that $a \neq 0.0$. $b \geq 1.0$ and $a \times b = 0.0$. In implementing programs on existing machines, one has to reckon with this kind of peculiarity that makes the construction of truly portable (machine independent) software almost impossible.

It would be expedient if a clear terminology existed to denominate machine arithmetic characteristics, so that computer/compilers could be

classified according to their arithmetical properties. Such a classification would enable a programmer to guarantee his code for an environment in which the arithmetical properties belonged to a certain class.

We do not intend to start such a classification here, but to make the idea more clear we shall mention some useful requirements for floating-point arithmetic. A *minimal set* of requirements should be

$$fl(a \circ b) = a(1+\alpha) \circ b(1+\beta)$$

where $fl(a \circ b)$ denotes the result of a floating point operation: $\circ$ = +, -, *, /; α and β are numbers depending on a and b respectively and on $\circ$ such that $|\alpha| \leq \varepsilon$, $|\beta| \leq \varepsilon$ where ε is a machine parameter, the "relative machine precision." In general these minimal requirements are not adequate. Additional requirements would be, e.g., *monotonicity*; i.e.,

$$a > b \rightarrow c + a > c + b;$$

$$c > 0,\ a > b \rightarrow c * a > c * b;$$

etc.

Machine arithmetic can be called *optimal* with respect to +, -, *, /, if, as a result of any of these operations between two floating-point numbers, the nearest representable number is delivered; if the result lies exactly between two representable numbers one of these should be chosen in a uniquely determined way.

Closely related to the machine arithmetic are questions with respect to overflow and underflow (i.e., situations where the operands a and b in the elementary operations are such that the *arithmetic requirements cannot be satisfied*). For these cases it is expedient if a user can make a choice among 3 options: (1) hard failure action (i.e., after an error message the computation is stopped); (2) soft

failure action (i.e., after a message the computation goes on), or (3) no action (computation goes on without any message). If the computation is continued, the value delivered might be some kind of "undefined" or some "near" representable number.

Another question related to arithmetic is the conversion between the machine representation of a real number and its decimal representation in I/O or in a program text. A good compiler should allow all machine representable floating-point numbers to be converted to distinct decimal representations, and vice versa. To illustrate, the CDC ALGOL 60 version 3 compiler violates this requirement since floating-point numbers are represented by a 48-bit binary mantissa, whereas the compiler ignores the 15th and further digits of any decimal representation. Thus, there are certain floating-point numbers that cannot be distinguished by their decimal representation.

Like machine arithmetic, the elementary (i.e., compiler-provided) functions should satisfy certain clearly defined specifications. A *minimal requirement* (which can be imposed on any function) is

$$\text{"computed value of" function } (x,y,\ldots) = \text{function } (x(1+\xi),\ y(1+\eta),\ldots)\ (1+\phi)$$

where $|\xi| \leq e$, $|\eta| \leq e$, ..., and $|\phi| \leq f$; e is some number related to the relative machine accuracy and f is the relative function accuracy. For most elementary functions one can impose either e = 0 or f = 0, but f = 0 is preferable, since, e.g., it causes arcsin(sin(x)) always to deliver a value, whereas e = 0 might result in a call of arcsin(y) with y > 1.0. For monotonic functions, preservation of monotonicity could also be required.

As with machine arithmetic, a short and clear description of the properties of elementary functions (independent of the particular algorithm used for their computation) is important, so that a programmer will be able to guarantee his code under certain well defined standard conditions.

Like overflow and underflow, an improper call of an elementary function (such as sqrt(-2.0) or ln(-6.8E-8)) should result in one of 3 (optional) actions: (1) hard failure, (2) soft failure, or (3) no action. If the computation continues, the function should deliver "undefined" or rather a "near" representable number (e.g., ln(x) delivers ln(|x|) if x < 0.0).

Portability

Although standardization and classification of machine properties has not yet reached a sufficiently developed state and compilers are not perfect, the experience with the transportation of the ALGOL library NUMAL justifies the following conclusion.

If we use (1) a well-defined language (or only a sublanguage with well-defined effects) and (2) a compiler which interprets the language correctly; if we have available (3) an environment (hardware + compiler) with well-behaved and well-defined arithmetic, and if we construct (4) well-programmed software, then portability is scarcely a problem.

We can think of these four aspects as independent of each other and the realization can be a task for different groups of people. The numerical or software specialist can raise standards for (1), (2), or (3) but his prime interest should be (4). If he cannot work independently of a particular machine he will not be able to create truly portable software.

This conclusion can be used as a starting point for and a philosophy behind the construction of software packages. In fact this idea is not new at all and it is even partly realized in some sense in the Handbook for Numerical Computation {4}. Indeed, the ALGOL texts in this book appear to be almost completely portable and they were easily implemented on the CDC CYBER-system*, except for (1) the construction of a double-length inner product, and (2) the change of some machine constants that were mentioned in the program texts. The result, however, is rather a collection of procedures than a coherent structure and the literal ALGOL text could not be made to run very efficiently.

Portability and Efficiency

The final efficiency of a code is to a high degree dependent on the compiler used. So, renouncing all special abilities of a particular environment, we will never obtain the most efficient code. On the other hand, if we exploit the special features we may not expect portability. Hence portable software will not be the most efficient on all computers. However, a great deal of the disadvantage of portable programming can be eliminated by the exploitation of the modular structure of a library. An enormous amount of work is essentially done by the very basic routines such as matrix-vector operations, polynomial evaluations, etc. This yields the possibility of speeding up the codes essentially by replacing the isolated, compiler translated, innermost pieces of a library by hand translated code. In this way a 2 to 3 times faster runtime was obtained for the NUMAL library.

---

*This was done, mainly for reference purposes, by the computing centers of the Universities of Utrecht and Groningen.

Portability and User Convenience

If ALGOL 60 routines from NUMAL are run on a particular machine, parameters such as the relative machine accuracy have to be specified. This and the complete absence of I/O are not very attractive to the average user. This disadvantage can be overcome by a user interface, i.e., a piece of software which establishes the connection between a non-specialist user and the set of numerical routines. This program (which itself could be machine independent to a certain extent), sets the machine parameters, interprets performance indicators and, possibly, selects a particular numerical procedure from among those available. By adding such a user interface on top of the numerical routines, we obtain *a structured library in 3 levels:* (1) The *user interface,* (2) The *numerical routines* consisting of algorithms coded for portability, and (3) a machine-dependent *speed-up part.*

ALGOL 60 - ALGOL 68

Although the realization of a numerical library according to the above mentioned criteria should be possible in any well-standardized language in which algorithms can be expressed, we have concentrated on languages of the ALGOL family. Our library in ALGOL 60 is available for external use; program texts and descriptions are distributed to subscribers and a version with a speed-up part, adapted to the CDC ALGOL 60 version 3 compiler,is maintained.

At this moment we are considering the possible construction of a software library, satisfying the described criteria, in the full language ALGOL 68. This language provides a number of useful features that are missing from ALGOL 60; e.g., multiple precision, file handling

and I/O routines. Other features, such as operation declarations, allow for particularly clear and well-structured programming that is not confused by opaque jumping or administrative details. In this way ALGOL 68 programming could combine reasonably efficient coding with a clear and realistic description of numerical algorithms.

Acknowledgements

I am grateful to Walter Hoffmann for a number of valuable remarks.

We are aware of the fact that we made a number of critical remarks about the performance of the ALGOL 60 features available on CDC; however, in our opinion, CDC offers ALGOL facilities superior to those of other major manufacturers which have not recognized ALGOL as an important language.

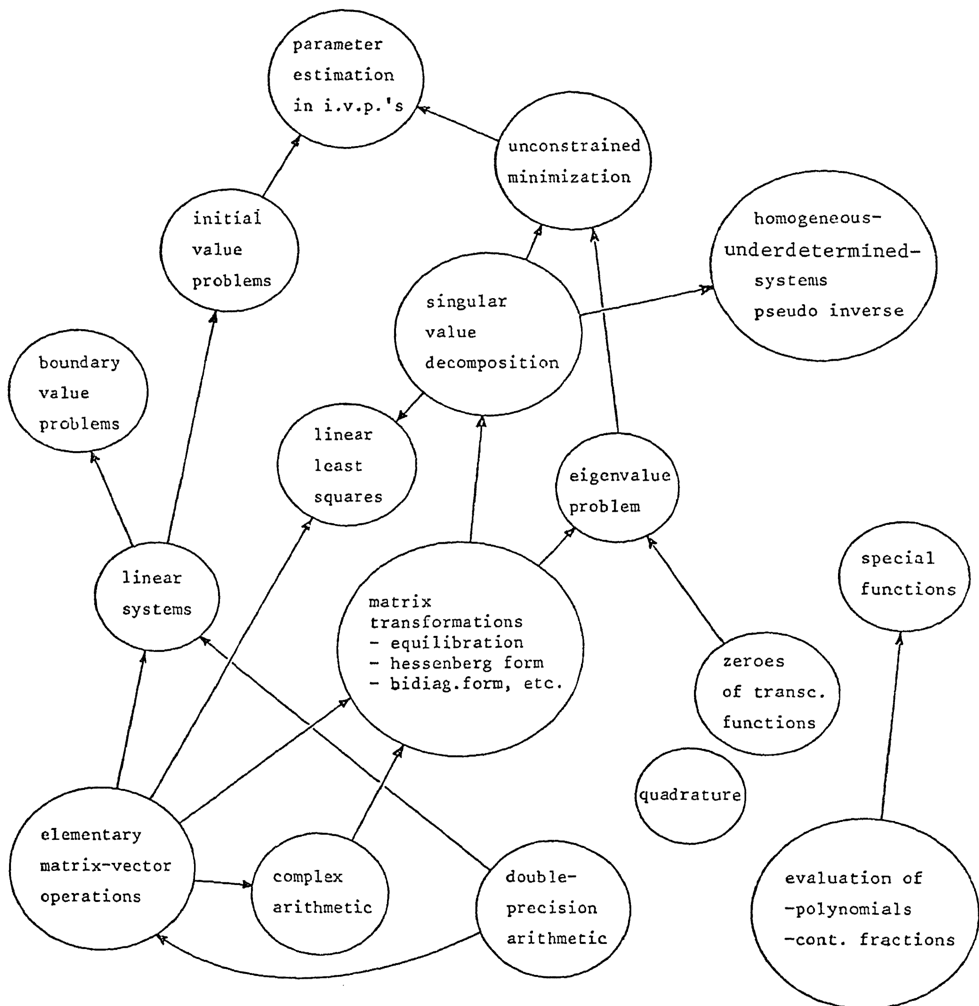

Figure 1

The main lines of interdependence between the different parts of the library NUMAL.

REFERENCES

1. Dekker, T. J., A Floating-Point Technique for Extending the Available Precision, Numer. Math. 18 (1971) 224-242.

2. Naur, P., Ed., Revised Report on the Algorithmic Language ALGOL 60, A/S Regnecentralen, Copenhagen, 1964.

3. NUMAL, A Library of Numerical Procedures in ALGOL 60, Mathematical Centre, 1974.

4. Wilkinson, J. H., and Reinsch, C., Handbook for Automatic Computation, Vol. 2, Linear Algebra, Springer-Verlag, Heidelberg, 1971.