A Note on the Portability of Mathematical Software

Michael A. Malcolm
Lawrence D. Rogers

Department of Computer Science
University of Waterloo

Research Report CS-76-28

May 1976

A Note on the Portability of Mathematical Software

Michael A. Malcolm
Lawrence D. Rogers

Department of Computer Science
University of Waterloo

There is a continuing debate concerning the problems of implementing mathematical software so that it is portable over a reasonable number of computers. The problems can be divided into two categories: (1) writing code so it can be compiled and executed on each target machine, and (2) writing code that will produce acceptable results on each target machine regardless of the mathematical properties of its number representation and arithmetic. Both these problems present difficulties in porting mathematical software and are of specific concern to numerical analysts trying to create high quality library programs to be used on a wide class of machines.

An excellent paper on the subject of numerical subroutine libraries by W. J. Cody (1974) focuses on the portability problem and adopts a pessimistic view. Cody states that "it is possible to write Fortran programs which are transportable with minimum change provided one knows each of these local options and takes great pains to program wherever possible in only the common intersection of the target dialects of Fortran. This is beyond the knowledge of the average programmer let alone the average numerical analyst."

We agree that Fortran is presently the most logical vehicle for porting numerical software; however, the problem of using a portable subset of Fortran (called PFORT), while difficult, is largely a mechanical one. Software tools such as the PFORT Verifier (see Ryder (1974)) are now available to aid the programmer in determining if a Fortran program is syntactically portable. More recently, preprocessors have been developed which compile well-defined source languages into portable Fortran. One such preprocessor, for a language called RATFOR, has been developed at Bell Laboratories by B. W. Kernighan (1975). RATFOR extends PFORT with a well-chosen set of control structures. The preprocessor, which is written in RATFOR (see Kernighan and Plauger (1976)), translates RATFOR statements into PFORT. Unfortunately, the preprocessor detects only those syntax errors which occur in the new control statements. The burden of writing the rest of the program (declarations, input/output, etc.) in PFORT remains with the RATFOR programmer aided by the PFORT verifier. For this reason, P. Y. T. Chan (1976) has developed a compiler for a completely specified language, called JOTS. The JOTS compiler generates PFORT as its object code, which has been verified by the PFORT Verfier. The JOTS programmer needs no knowledge of PFORT or even Fortran, and should never experience a syntax error from a Fortran compiler or have a need to use the PFORT verifier. Hence the task that was once "beyond the knowledge of the average programmer" is becoming automated.

A more difficult problem is that of acheiving sufficient numerical accuracy on each of the machines to which a program is ported. This problem is not well understood in general, but in our experience it is easily solved for nearly all numerical programs.

The problem is illustrated by Cody's example of computing the function $|z|$ for complex $z = x + iy$. Under the topic "designing a practical computational algorithm", Cody (1974) describes how to compute $|z|$ using a specific machine, the IBM 360. He shows how to avoid unnecessary rounding errors due to the floating-point representation used on the IBM 360. We will review the development of Cody's algorithm and then investigate its portability.

Instead of calculating $|z| = (x^2 + y^2)^{1/2}$ directly, let $u = \max(|x|, |y|)$ and $v = \min(|x|, |y|)$ and then set $|z| = u(1 + (v/u)^2)^{1/2}$. This is the usual trick which avoids overflow and reduces the importance of underflow.

On the IBM 360, a floating-point number has the representation $16^e \times f$, where e is an integer and the fraction f satisfies $1/16 \leqslant |f| < 1$. Cody observes that $w = 1 + (v/u)^2$ satisfies $1 \leqslant w \leqslant 2$. Hence, the intermediate result $1 \leqslant \sqrt{w} \leqslant \sqrt{2}$ always has three leading zero bits in its IBM 360 representation. This can be avoided by scaling w thereby reducing the size of the relative error in $\sqrt{w}$. Cody suggests using

$$|z| = 2u(.25 + (v/2u)^2)^{1/2}$$

which ensures that the result of the intermediate square root has a leading bit of unity. This scaled formula yields up to three additional bits of accuracy on the IBM 360.

Since the development of the scaled formula is based on observations about the IBM 360 number representation scheme, one is left with the impression that an accurate modulus computation is highly machine dependent. However, by considering the same computation using a more general floating-point representation we can see that Cody's scaled formula will produce good results on any contemporary machine. We can also gain greater insight into the choice of scale factor.

Consider any floating-point system using representations of the form $\beta^e \times f$ where the radix $\beta > 1$ and exponent e are integers and the fraction f satisfies $\beta^{-1} \leqslant |f| \leqslant 1$. Also, let the scaled w be defined as

$$w(\alpha) = \alpha^{-2} + (v/\alpha u)^2 .$$

To avoid an unnecessary loss of relative precision in the intermediate $\sqrt{w(\alpha)}$ we wish to choose α so that $p \leqslant \sqrt{w(\alpha)} < 1$ for the maximum possible value of p. Since $\alpha^{-2} \leqslant w(\alpha) \leqslant 2\alpha^{-2}$, we should choose α to be as small as possible subject to $\alpha > \sqrt{2}$.

However, if we choose $\alpha = 1.5$, for example, the resulting formula has the undesirable property of introducing rounding error into $|z|$ for pure real or imaginary z which are exactly representable. This problem can be avoided by either testing explicitly in the algorithm for $v = 0$ or by choosing α such that both α and α^{-2} are exactly representable. It is not hard to prove that the smallest value of α greater than $\sqrt{2}$ for which both α and α^{-2} are exactly representable in number systems where $\beta = 2^n$ $(n \geqslant 1)$ or $\beta = 10$, is $\alpha = 2$, Cody's choice. Since $\sqrt{w(2)} \geqslant 1/2$, the leading bit of f for any system with $\beta = 2^n$ $(n \geqslant 1)$ is unity. For $\beta = 10$ the leading digit of f is at least 5. Hence, the possible loss of relative precision in $\sqrt{w(2)}$ due to small fraction values cannot occur in any contemporary floating-point system.

The morale of this discussion is that the numerical aspects of making mathematical software portable are often solved by simply designing the algorithms for a general floating-point system characterized by parameters (e.g., β in the above example). We have successfully applied the same general concept to non-numerical problems of portable programming. In these cases we have found it advantageous to use such parameters as word size (in bits), type of integer representation, and number of characters per word. Using these parameters appropriately, programs can be made highly portable – often machine independent – over a large class of machines. Many of the machines in this class have yet to be designed or built. Of course, certain machines are not included in the portability set using this technique; e.g., Cody's formula for the modulus computation is not appropriate for machines with $\beta = 3$. And unusual architectures, such as parallel computers, may not be utilized well by portable software.

References

Chan, P. Y. T. (1976), "A portable Fortran preprocessor", M. Math Thesis, Department of Computer Science, University of Waterloo, Waterloo, Ontario.

Cody, W. J. (1974), "The construction of numerical subroutine libraries", *SIAM Review*, vol. 16, 36-46.

Kernighan, B. W. (1975), "RATFOR – a preprocessor for a rational Fortran", *Software – Practice and Experience*, vol. 5, 395-406.

Kernighan, B. W. and P. J. Plauger (1976), *Software Tools*. Addison Wesley.

Ryder, B. G. (1974), "The PFORT verifier", *Software – Practice and Experience*, vol. 4, 359-377.