Precision Control and Exception Handling in Scientific Computing*

K. R. Jackson and N. S. Nedialkov

Computer Science Dept., University of Toronto,
Toronto, Ontario, Canada M5S 1A4.
{krj, ned}@cs.toronto.edu

Abstract

This paper describes convenient language facilities for precision control and exception handling. Nedialkov has developed a variable-precision and exception handling library, SciLib, implemented as a numerical class library in C++. A new scalar data type, *real*, is introduced, consisting of variable-precision floating-point numbers. Arithmetic, relational, and input and output operators of the language are overloaded for reals, so that mathematical expressions can be written without explicit function calls. Precision of computations can be changed during program execution. The exception handling mechanism treats only numerical exceptions and does not distinguish between different types of exceptions.

The proposed precision control and exception handling facilities are illustrated by sample SciLib programs.

1 Introduction

Most scientific computing environments provide only very limited facilities for controlling precision, usually only a choice between single and double precision, or a poorly defined mixture of the two. Some environments have no provision at all for handling numerical exceptions — this is the case in Fortran, for example.

Although several variable-precision packages have been developed [1, 3, 4, 6, 7], most do not incorporate exception handling in an effective way. The purpose of this paper is to describe convenient language facilities for both precision control and exception handling, and to show how these facilities help to construct better numerical algorithms.

A variable-precision and exception handling library, SciLib, has been implemented in C++ [5], so the notation and examples are based on this language. More sample programs and the SciLib user's guide can be found in [9][1].

The proposed precision control provides a simple way of changing precision during program execution. It is based on a binary variable-precision floating-point arithmetic, where the first two precisions correspond to normalized IEEE single and double, respectively.

Our exception handling mechanism deals only with exceptions arising in floating-point computations. We believe that it can be effectively implemented on all modern machines, including pipelined and parallel ones.

A computing environment equipped with these two features would have the following advantages:

1. more powerful algorithms can be implemented;

2. the algorithms can be analyzed and proved correct more easily;

3. programs are easier to understand.

*This work was supported in part by the Natural Sciences and Engineering Research Council of Canada and the Information Technology Research Centre of Ontario.

[1]The thesis [9] is available by anonymous ftp from cs.toronto.edu in pub/reports/na/ned-94-msc.ps.Z as a compressed postscript file.

In addition, precision control introduces additional advantages such as the following:

1. only one version of a numerical library is needed, rather than separate versions for each precision;
2. theoretical error bounds can be easily verified.

Precision control and exception handling can be used alone or in combination. The following paradigms are illustrated in [9].

1. Use precision control without exception handling:
   (a) use higher precision temporarily;
   (b) use higher and higher precision until some criterion is satisfied.
2. First try a fast straightforward algorithm which works in most cases, but, if an exception occurs, solve the problem by
   (a) a more robust algorithm in the same precision;
   (b) repeating the calculations in higher precision;
   (c) a more robust algorithm in higher precision.

Our variable-precision floating-point numbers, the arithmetic associated with them and the language facilities for precision control are described in §2. The exception handling mechanism and language facilities for supporting it are discussed in §3. Each of §2 and §3 contains a simple example illustrating the material described there. We briefly mention a few other applications of precision control in §4 and we end with some concluding remarks in §5.

2 Precision Control

In this section, we describe our variable-precision floating-point numbers, the arithmetic associated with them and the language facilities for precision control. In particular, we discuss the relations between precisions in terms of significant digits and exponent ranges that we believe are suitable. We end this section with an example of a variable-precision SciLib program.

2.1 Floating-Point Formats and Precisions

A nonzero normalized floating-point number has the logical form

$$(-1)^s \times d_0 \,.\, d_1 d_2 \ldots d_m \times b^e \qquad (1)$$

where $s \in \{0, 1\}$, $(-1)^s$ is the *sign*; b is the *base* of the floating-point system; $d_0 \,.\, d_1 d_2 \ldots d_m$ is the *significand*, $0 \leq d_i \leq b-1$, $d_0 \neq 0$; the integer e is the *exponent*, satisfying $E_{\min} \leq e \leq E_{\max}$. The floating-point format (1) is characterized by four integer parameters: b, m, $E_{\min}$ and $E_{\max}$.

The IEEE Standard [2] uses the terms *single*, *double* and *extended* to denote different floating-point formats and the corresponding precisions. Since we use more precisions than specified by the IEEE Standard and provide a simple method of precision control, we denote the precisions by powers of two: 1, 2, 4, 8, ...

Our implementation of variable-precision arithmetic has the following properties:

1. the base of the floating-point system is 2;
2. precisions 1 and 2 are the same as normalized *single* and *double*, respectively, in the IEEE Standard;
3. the number of significant digits of precision 2 ∗ p is more than double the number of significant digits of precision p;
4. the exponent range of precision 2 ∗ p is significantly larger than double the exponent range of precision p. In the current implementation of SciLib, $E_{\max}$(2p) and $E_{\min}$(2p) are approximately 4 times $E_{\max}$(p) and $E_{\min}$(p), respectively, except that, for p = 1, the factor is 8 times, to conform to the IEEE Standard.

More details about SciLib's variable-precision floating-point numbers may be found in [9].

2.2 Language Facilities for Precision Control

An environment for scientific computing should have the capability to change precision during program execution. In SciLib, this is accomplished by changing the precision value (power of two). We use *double precision*[2] to denote precision 2^{i+1} when its previous value was 2^i. Since the precision value is global, it can be accessed anywhere in the program. The precision can be altered at any time by the *precset* function:

precset(pexp);

where *pexp* is an integer valued expression whose value is 2^k for some $k \geq 0$. The current value of the precision is referred to as the *current* or *working* precision. It can be saved in an integer variable, say p, by the *precsav* function:

p = precsav();

The largest value of the precision is implementation dependent and is stored in a constant, MAXPREC, which can be accessed by the user. If the precision is not explicitly set, its value is 1 by default.

We have added the floating-point data type *real* to C++. An object of *real* type can contain a floating-point number in the precision set at the time the object is declared. The precision of a *real* object x can be determined and assigned to an integer variable px by the *precof* function:

px = precof(x);

Arithmetic and relational operations on *real* objects are expressed in the usual C++ notation without explicit function calls. Moreover, the precedence of operations is the same as in C++.

If an object of a type derived from *real* is declared, such as an array, class, or structure, the *real* components are created in the current precision. Since the precision can be changed during program execution, *real* objects are created at runtime.

[2]It will be clear from the context when *double* refers to IEEE double precision.

Since the precision value is global and *real* objects are declared in the current precision, functions producing results in the current precision can be easily implemented.

2.3 Variable-Precision Floating-Point Arithmetic

Our goal is to keep variable-precision arithmetic simple. The basic rule is that, if the current precision is p and either a or b is a *real* variable or constant, then SciLib computes the correctly rounded precision p approximation to the true value of a op b independent of the precisions of a and b, where op is one of the four elementary arithmetic operators $+$, $-$, $*$ or $/$. Similarly, if a is a *real* variable or constant, then SciLib computes the correctly rounded precision p approximation to $\sqrt{\mathrm{a}}$ independent of the precision of a and it computes a "reasonably accurate" precision p approximation to the true value of any other elementary function implemented so far, again independent of the precision of the argument of the function.

If the right side of an assignment statement evaluates to a precision p *real* value, and the left side of the assignment statement is a precision q *real* variable, then the precision p value is first correctly rounded to precision q before the assignment is performed. Mixed mode assignments follow similar rules, as described in [9].

For example, if the statement

$$\mathrm{a} = \mathrm{b} * \mathrm{c} + \sqrt{\mathrm{d}};$$

is executed in SciLib with current precision p, where a, b, c and d are *real* variables and b, c and d have been assigned values, then

1. the correctly rounded approximation in precision p to the true value of $\mathrm{b} * \mathrm{c}$ is computed,
2. the correctly rounded approximation in precision p to the true value of $\sqrt{\mathrm{d}}$ is computed,
3. the correctly rounded approximation in precision p to the sum of the two intermediate results described in points 1 and 2 above is computed next, and

4. this final result is correctly rounded to the precision of a and assigned to a.

The calculations described in points 1 and 2 above could be done in the reverse order or in parallel.

The default rounding mode is to round to the nearest floating-point number in the appropriate precision, where, in the case of a tie, the nearest even number is selected. Two other rounding modes are available: round to the nearest floating-point number in the appropriate precision in the direction of plus infinity or minus infinity. Like the current precision, the rounding mode value is global and can be changed by a function call.

If the absolute value of a result is greater than the largest representable positive number in the current precision, it *overflows*; if it is nonzero and its absolute value is less than the smallest positive number in the current precision, it *underflows*. Both *overflows* and *underflows* raise the exception flag, as is discussed in more detail in §3.

2.4 An Example: Double Precision Accumulation of a Dot Product.

To illustrate the use of precision control, consider the double precision accumulation of the dot product of two vectors a and b, as shown in Fig. 1. Two *real* arrays, a and b, are declared in the current precision, p, and initialized to contain values of precision p. Then the precision is saved in an integer variable p, and the current precision is increased to $2 * \mathrm{p}$. The real variable tmp is declared in precision 2∗p and initialized to 0. Then the dot product is accumulated in tmp in double precision. That is, each multiplication and addition is performed in precision $2 * \mathrm{p}$, and the intermediate results are stored in precision $2 * \mathrm{p}$ in tmp. In the last statement, the *real* variable dot is declared in precision p, the value in the variable tmp is rounded from precision $2 * \mathrm{p}$ to precision p and then assigned to the variable dot.

```
real a[n], b[n];
// initialize a and b
int p = precsav();
precset(2*p);
real tmp = 0;

for ( int i = 0; i < n; i++)
    tmp = tmp + a[i]*b[i];

precset(p);
real dot = tmp;
```

Figure 1: Double precision accumulation of a dot product.

3 Exception Handling

In this section, the exception handling mechanism and a language construct for its support are described and illustrated by two sample SciLib programs.

A numerical exception can arise if a divide-by-zero, overflow or underflow occurs. If any of the above situations arise, a global *exception flag* is raised. Initially the flag is cleared. In addition, a user can explicitly raise the exception flag by executing the *setexc* function:

```
setexc();
```

The exception handler is implemented as an *enable-handle-end* construct [8] as shown in Fig. 2. The semantics of the construct are simple: if an exception occurs in the *enable* block, the exception flag is cleared, and the *handle* block is executed; otherwise the *handle* block is skipped, and the program continues after the *end* statement. Nesting of the construct is allowed.

When the program control is transferred to the *handle* block is implementation dependent: it can be transferred immediately after the exception arises in trapping mode or after the execution of all statements in the *enable* block in propagate mode. The current version of SciLib uses the propagate mode, but the choice does not affect any of the examples in this paper.

The *enable* block is entered if there are no numerical exceptions before it; otherwise the program terminates with an appropriate message, unless the whole construct is nested within another *enable* block.

In general, the *enable* block tries to execute a fast straightforward algorithm, which solves the problem in most cases. If a numerical exception occurs during the execution of the *enable* block, the *handle* block tries to solve the problem in another, more reliable, way.

```
enable
    try a fast algorithm
handle
    must be an exception
    try another algorithm
end
```

Figure 2: The exception handling construct.

The user should be cautious about using any values in the *handle* block that were computed in the *enable* block, since the associated statements might not have been executed in trapping mode, and, even if they were, the expected value might not have been computed if an exception arose earlier in the *enable* block. We should add the warning that, even if the user is careful about this, an optimizing compiler might introduce an error if a common expression occurs in both the *enable* and *handle* blocks, since the compiler might use the value computed earlier in the *enable* block instead of recomputing the expression in the *handle* block.

The exception handling mechanism treats only numerical exceptions and does not distinguish between different types of numerical exceptions. In addition, since it only detects if an exception has occurred and does not try to determine which exception occurred first, it can work efficiently on pipelined or parallel machines. Moreover, we should add that we have not found it useful to know which exception has occured.

3.1 Two Examples of the Hypotenuse Function.

Consider the function *hypot* in Fig. 3 that computes the hypotenuse function $\sqrt{x^2 + y^2}$. It first tries to compute the result in the current precision in a straightforward way in the *enable* block. If an overflow or underflow exception occurs, the computation is repeated in the *handle* block in double precision. Although sqrt(x∗x+y∗y) cannot overflow or underflow in precision $2 * p$, an overflow might occur in the *handle* block when the result is rounded to precision p and assigned to answer. However, unlike the exceptions that might occur in the *enable* block, the exception in the *handle* block arises only if the answer is outside the range of precision p floating-point numbers. The calling function can detect this exception by using another *enable-handle-end* construct.

```
real hypot(const real &x, const real &y)
{
    real answer;
    enable
       answer = sqrt(x * x + y * y);
    handle /* an overflow or underflow
              has occurred */
       int p = precsav();
       precset(2 * p);
       answer = sqrt(x * x + y * y);
       precset(p);
    end
    return answer;
}
```

Figure 3: The hypotenuse function with precision control.

The same expression is computed in Fig. 4 (see also [8]), but the exceptions are handled without using precision control. The handler first tests if x or y is zero. If so, the answer is $|x|+|y|$. Otherwise, if x and y differ greatly in magnitude, the variable having the smaller absolute value can be neglected. If neither of the simple cases described above holds, then x and y are scaled so

```
real hypot2(const real &x, const real &y)
{
real answer;

enable
   answer = sqrt(x * x + y * y);
handle /* an overflow or underflow
            has occurred */
   if (x == 0 || y == 0)
      answer = fabs(x)+fabs(y);
   else
      {
      int logbx = ilogb(x);
      int logby = ilogb(y);

      if (2*fabs(logbx−logby) > precdig() + 1)
         answer = max(fabs(x), fabs(y));
      else
         {
         real scaledx = scalb(x, −logbx);
         real scaledy = scalb(y, −logbx);
         real scaledanswer =
           sqrt(scaledx*scaledx+scaledy*scaledy);
         answer = scalb(scaledanswer, logbx);
         }
      }
end

return answer;
}
```

Figure 4: Hypotenuse function without precision control. Here fabs(x) returns the absolute value of x, ilogb(x) returns the exponent of x, scalb(x,n) returns $x \times 2^n$ and precdig() returns the number of significant binary digits in the current precision.

that the result can be computed without exceptions. When the result is unscaled, it can overflow, but again note that, unlike the exceptions that might occur in the *enable* block, the exception in the *handle* block arises only if the answer is outside the range of precision p floating-point numbers. The calling function can detect this exception by using another *enable-handle-end* construct.

These two examples illustrate the commonly arising case that handling exceptions by repeating the computation in double precision makes an algorithm easier to implement, analyze and understand. Note also that both functions produce results in the precision set when the function is called.

4 Other Applications of Precision Control and Exception Handling

Like the dot product example given in §2, many numerical algorithms benefit from the use of higher precision to compute intermediate results. Iterative improvement for the solution of linear algebraic equations is a case in point. See [9] for an SciLib implementation of this scheme. Range reduction in the evaluation of trigonometric functions is another example of an application in which higher precision is needed temporarily.

Some of the complex elementary functions in [8] can be more easily implemented and analyzed if precision control and exception handling are available. See [9] for an example of a complex natural logarithm function (based on the algorithm described in [8]) where higher precision is used in a *handle* block to avoid overflows, underflows and catastrophic cancelation.

Users may wish to compute results to a predefined accuracy. Consider for example the problem of calculating a numerical solution $\bar{x}$ of a linear system $Ax = b$ such that $\|x - \bar{x}\| \leq \epsilon$, where ϵ is a user specified error tolerance. In this example, as is often the case, the required precision of the intermediate results is not know beforehand, and it might be necessary to increase the precision of the computation significantly to achieve the desired accuracy. See [9] for an example of a SciLib program that solves such a problem.

To illustrate the correctness of theoretical error bounds, it can be helpful to have a way of performing higher precision calculations. The tests may be more reliable if the higher precision results are computed by a different algorithm from the one being tested.

Bailey [3] briefly surveys other applications of variable-precision computations, including to support or to reject conjectures in mathematics, to study mathematical constants, to explore public-key cryptography and to test the integrity of computer systems.

5 Conclusion

The main advantage of using precision control is that results with desired accuracy can be obtained with less effort than by utilizing only single and/or double precision, since common difficulties in numerical algorithms design, such as dealing with cancelations, overflows and underflows, can often be eliminated by increasing the precision in critical sections. As a consequence, programs are usually more concise and accurate, easier to understand and analyze, and it is often possible to prove tighter error bounds for them.

There are many questions concerning what precisions to incorporate in a scientific computing environment. We have chosen to use binary floating-point numbers, but decimal numbers are an attractive alternative. Moreover, our precisions are based on a 32 bit computer word and we double the number of words used to represent a floating-point number as the precision is increased from 2^i to 2^{i+1}. Some may argue that it is useful to increase the precision in smaller increments. One convention that we have found quite useful is to more than double the exponent range when doubling the precision — but by how much more is a question. Similarly, we have found that to more than double the length of the significand when doubling the precision works well in practice — but again by how much more is a question.

In contrast to precision control, the proposed exception handling construct is much easier to implement, and it can work efficiently on pipelined and parallel machines.

We chose to implement our prototype system in C++ because it provides convenient facilities for implementing new data types, and the operators of the language can be overloaded for such types. A disadvantage of C++ is that new operator symbols cannot be introduced, which restricts our ability to express some useful operations such as directed roundings for the standard arithmetic operators $+$, $-$, $*$, $/$. This problem can be overcome by building a precompiler which translates a high-level C++ like language into C or C++. There are, though, two disadvantages with this approach: the translated code is not always optimal and existing debuggers cannot be used on the high-level language.

Acknowledgments. The authors would like to thank Prof. T. E. Hull, who inspired this work, for his helpful advice and many extensive discussions.

References

[1] O. Aberth and M. J. Schaefer. Precise computation using range arithmetic, via C++. *ACM Trans. Math. Softw.*, 18(4):481–491, 1992.

[2] American National Standards Institute, New York. *IEEE Standard for Binary Floating-Point Arithmetic*, 1985. ANSI/IEEE Std. 754–1985.

[3] D. H. Bailey. Algorithm 719: Multiprecision translation and execution of Fortran programs. *ACM Trans. Math. Softw.*, 19(3):288–319, 1993.

[4] R. P. Brent. A Fortran multiple precision arithmetic package. *ACM Trans. Math. Softw.*, 4(1):57–81, 1978.

[5] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, MA, 1990.

[6] J. Ely. *Prospects for Using Variable Precision Interval Software in C++ for Solving Some Contemporary Scientific Problems.* PhD thesis, Computer Science Dept., Ohio State University, Columbus, Ohio, 1990.

[7] T. E. Hull and M. S. Cohen. Toward an ideal computer arithmetic. In M. J. Irwin and R. Stafanelli, editors, *Proceedings of the 8th Symposium on Computer Arithmetic*, pages 131–138. IEEE Computer Society, Los Angeles, 1987.

[8] T. E. Hull, T. F. Fairgrieve, and P. T. P. Tang. Implementing complex elementary functions using exception handling. *ACM Trans. Math. Softw.*, 20(2):215–244, 1994.

[9] N. S. Nedialkov. Precision control and exception handling in scientific computing. Master's thesis, Computer Science Dept., University of Toronto, Toronto, Canada, 1994.