Precision Control and Exception Handling in Scientific Computing

by

Nedialko Stoyanov Nedialkov

A thesis submitted in conformity
with the requirements for the degree of
Master of Science
Department of Computer Science
University of Toronto

Abstract

Precision Control and Exception Handling in Scientific Computing

Nedialko Stoyanov Nedialkov
Master of Science, 1994

Department of Computer Science
University of Toronto

We describe language facilities for precision control and exception handling and show how they can help to construct better numerical algorithms.

Precision of computations can be changed during program execution. The first two precisions are IEEE single and double. Increasing the precision provides at least two times more significant digits while the exponent range is significantly larger than double the exponent range of the previous precision.

The exception handling mechanism treats only numerical exceptions and does not distinguish between different types of numerical exceptions.

A variable-precision and exception handling library, SciLib, has been implemented in C++. A new scalar data type *real* is introduced consisting of variable-precision floating-point numbers. Arithmetic, relational, input and output operators of the language are overloaded for the *real* data type, so it can be used like any other scalar data type of C++.

The proposed precision control and exception handling are illustrated using SciLib. The examples show the benefits of using precision control to handle exceptions and to avoid catastrophic cancelations.

Acknowledgments

I am very grateful to my supervisors Prof. T.E. Hull and Prof. K.R. Jackson for their invaluable advice, trust, and patience. Prof. Hull inspired this work, and his experience helped me to avoid many pitfalls. He is also the person who did the most for me in Canada.

My special thanks to Tom Fairgrieve who was always willing to help. Thanks also to my other colleagues for providing an enjoyable working environment and to the department for its financial support.

I am extremely grateful to my wife Emilia and our parents for taking care of our son Stoyan.

This work is dedicated to my son Stoyan

Contents

Chapter 1

Introduction

Most scientific computing environments provide only very limited opportunities 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 — for example, this is the case with Fortran.

The purpose of this thesis is to describe convenient language facilities for precision control and exception handling, and to show how they help to construct better numerical algorithms.

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 are IEEE single and double respectively, and higher precisions are specified.

Our exception handling mechanism deals only with exceptions arising in floating-point computations. It can be easily implemented in all high-level languages, as well as on all modern machines, including parallel ones.

Using a computing environment equipped with these two features would have the following benefits:

1. more powerful algorithms can be implemented;
2. the algorithms can be easily analyzed and proved correct;
3. programs are easier to understand.

In addition, precision control introduces 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 as well as in combination. The uses of the following paradigms are illustrated in this thesis.

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

A variable-precision and exception handling library, SciLib, has been implemented in C++, so the notation and examples are based on this language.

Chapter 2 describes our floating-point format, states the relations between precisions in a variable-precision environment, and describes the tools for precision control. Chapter 3 considers the exception handling mechanism and language facilities for supporting it. The material in each chapter is illustrated by a simple example. Chapter 4 presents more complex examples where using precision control is essential, and the algorithms are simplified if higher precision is used in handling exceptions. The final chapter, chapter 5, contains some implementation issues and concluding remarks. A self-contained user's guide for SciLib is supplied in Appendix A.

Chapter 2

Precision Control

In this chapter, we describe a floating-point format and precision, and specify the relations between precisions in terms of significant digits and exponent ranges that we believe should hold in an implementation of variable-precision arithmetic. Language facilities for precision control are also described and illustrated at the end of the chapter.

2.1 Floating-Point Formats and Precisions

A floating-point format is specified by

$$(-1)^s \times d_0.d_1d_2 \dots d_m \times b^e \tag{2.1}$$

where $s \in \{0, 1\}$, $(-1)^s$ is the *sign*; b is the *base* of the floating-point system; $d_0.d_1d_2 \dots d_m$ is the *significand*[1], $0 \leq d_i \leq b-1$, $d_0 \neq 0$; the integer e is the *exponent*, satisfying $E_{min} \leq e \leq E_{max}$. The floating-point format (2.1) is characterized by four integer parameters: b, m, $E_{\min}$ and $E_{\max}$.

The IEEE Standard [1] uses the terms *single*, *double* and *extended* to denote different floating-point formats and the corresponding precisions. Since our purpose

[1]Note that *significand* denotes the part of the floating-point format (2.1) which contains (m+1) *significant* digits, $d_0, d_1, \dots d_m$.

is to use more precisions than specified by the IEEE Standard [1] and to provide a simple method of precision control, we denote the precisions by the powers of two: $1, 2, 4, 8, \ldots$.

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

1. the base of the floating-point system is 2;

2. precision 1 is the same as normalized *single* precision in the IEEE Standard [1] and precision 2 is the same as normalized *double*;

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 implementation of SciLib, we have $E_{max}(2p)$ and $E_{min}(2p)$ approximately 4 times $E_{max}(p)$ and $E_{min}(p)$, except that when $p = 1$ the factor is 8 times, to conform to the IEEE Standard [1]. See Section A.2.1 for details.

2.2 Language Facilities for Precision Control

An environment for scientific computing should have the capability to change precision during program execution. In our system, SciLib, this is accomplished by changing the precision value (power of two). We shall use *double precision*[2] to denote the precision when its value is increased from 2^i to 2^{i+1}. 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:

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

p = precsav();

The largest value of the precision is implementation dependent and is stored in a constant, $MAXPREC$, which can be used by the software. If the precision is not explicitly set, its value is 1 by default. If the current precision is p, then the result of a floating-point operation will be stored, if possible, in the format for precision p floating-point values. If the absolute value of the 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, then it *underflows*.

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 was 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 including *real* objects are expressed in the usual C++ notation without explicit function calls. 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.

By default, the arithmetic operations are rounded to the nearest floating-point number in the current precision and to the nearest even number in the case of a tie. The rounding mode can be set to round arithmetic operations toward minus infinity or toward plus infinity. The rounding mode value is global, as is the precision. More details about the rounding mode control can be found in Sections A.2.2, A.3.5, and A.6.4.

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.

An Example: Dot Product. To illustrate the use of precision control, consider the double precision dot product of two vectors a and b, as shown in Fig. 2.1. Two arrays of real objects, a and b, are declared in the current precision, p. Then, the precision is saved and set to $2 * p$. The real variable tmp is declared in precision $2 * p$ and initialized to 0. The dot product is computed in precision $2 * p$: that is, the multiplications and additions are carried out in double precision. In the last statement, the variable tmp, containing the accumulated products, is assigned to the variable dot, which is in precision p. A rounding to the format of precision p normally takes place here.

```
real a[n], b[n];
// determine values of 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 2.1: Dot Product.

Chapter 3

Exception Handling

In this chapter, the exception handling mechanism and a language construct for its support are described. The language construct is illustrated by the example of a hypotenuse function which computes $\sqrt{x^2 + y^2}$. Two variants of the function are presented: the first uses higher precision to calculate $\sqrt{x^2 + y^2}$ if an exception arises in the current precision, and the second uses the current precision throughout.

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

setexc(); // raise the exception flag

The exception handler is implemented as an *enable-handle-end* construct, Fig. 3.1. 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 constructs is allowed. When the program control is transferred to the *handle* block is implementation dependent. The program control can be transferred immediately after the exception arises in trapping mode or after the execution of all

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

Figure 3.1: The exception handling construct

statements in the *enable* block in propagate mode. The mode does not affect the examples described in this thesis.

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 code, 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.

Since the exception handling only detects if an exception has occurred, it can work efficiently on pipelined machines or machines with multiple arithmetic units.

An Example: Hypotenuse Function. Consider the function *hypot* in Fig. 3.2 that computes $\sqrt{x^2 + y^2}$. It first tries to compute the result in the current precision in the *enable* block; if an overflow or underflow exception occurs, the computations are repeated in the *handle* block in double precision. Although *sqrt*(x*x+y*y) cannot overflow or underflow in precision $2p$, there is still a possibility of overflow in the *handle* block when the result is assigned to *answer*; the calling function can detect the exception by using another *enable-handle-end* construct.

The same expression is computed in Fig. 3.3 [4], but the exceptions are handled without using precision control. The handler first tests if x or y is zero. Otherwise, if x and y differ greatly in magnitude, the variable that has the smaller absolute value can be neglected. Otherwise x and y are scaled so that the result can be computed

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

Figure 3.2: Hypotenuse function.

without exceptions. When the result is unscaled, it can overflow, in which case the calling function can catch the exception.

Clearly, in this example, handling the exceptions by repeating the computations in double precision makes the algorithm easier to implement, analyze and understand. Note also that both functions produce results in the precision set when the function is called.

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

real answer;

enable
   answer = sqrt(x*x+y*y);
handle                  // 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 3.3: Hypotenuse function without using 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.

Chapter 4

Some Applications of Precision Control and Exception Handling

In the two previous chapters, we described how the proposed precision control and exception handling facilities can be used and showed the advantages of using precision control in handling numerical exceptions.

Many other applications require using higher precision and sometimes very high precision. For example, the range reduction in the evaluation of a trigonometric function has to be done in higher precision; constants such as $\sqrt{2}$ or multiples of π often need to be stored in higher precision; in testing theoretical error bounds it can be useful to have a way of performing higher-precision calculations.

In this chapter, we illustrate further the benefits of using precision control in critical parts of numerical algorithms to handle exceptions and to avoid catastrophic cancellations. Section 4.1 gives an example of iterative improvement, where using higher precision temporarily is crucial to the quality of the result. Section 4.2 describes a function that computes the inverse of a matrix using iterative improvement. The numerical results show that using higher precision(s) might be necessary to obtain results with a specified accuracy. Section 4.3 gives an example of a complex natural logarithm function where precision control is used to handle overflows and underflows and to avoid catastrophic cancellation. As a result, the implementation and error analysis are simplified.

4.1 Iterative Improvement

Consider the linear system

$$Ax = b \tag{4.1}$$

where A is an n-by-n matrix and b is an n-dimensional vector.

Let x_1 be an approximate solution of (4.1) obtained in precision p. If the matrix A is not too ill-conditioned, the round-off error in the computed solution can be reduced by *iterative improvement* [6]:

For $m = 1, 2 \ldots$

1. Compute the residual $r_m = b - Ax_m$.
2. Solve $Ay_m = r_m$
3. $x_{m+1} := x_m + y_m$.

The most critical step is (1). If it is carried out in precision p, serious cancellations normally occur in the evaluation of $b - Ax_m$. However, it can be more safely computed in precision $2 * p$. Steps (2) and (3) are performed in precision p. With precision control, (1) is easily implemented as shown in Fig. 4.1. The function *residual* computes $b - Ax$ in precision $2*p$ and assigns the result to r. In the first assignment, the vector b is copied into r; then the temporary vector tmp is declared in precision $2 * p$; the multiplication Ax is performed in precision $2 * p$ by the *matvec* function, and the result is assigned to tmp; the subtraction $r - tmp$ (r $-=$ tmp) is also performed in precision $2 * p$, and the result is rounded to precision p when assigned to r.

The above algorithm discussed in [2] is implemented in the function *improve* shown in Fig. 4.2. The parameters of *improve* are: A, the coefficient matrix; LU, the L-U decomposition of A obtained by partial pivoting; b, the right side of (4.1); *pivot*, the integer vector containing information about the row interchanges performed during the decomposition of A [3]; x, the vector containing the first solution as input and the corrected solution as output; *tol*, is a user-specified tolerance; and *converged* is an output parameter indicating if the tolerance is achieved, *converged* $= 1$, or if

```
void residual(vector & r, const vector & b, const matrix &A, const vector &x)
{
r = b;
int p = precsav();
precset(2*p);

vector tmp(size(b));
matvect(tmp, A, x);
r -= tmp;

precset(p);
}
```

Figure 4.1: The *residual* function.

the iterations do not converge, $converged = 0$. The *solve* function uses the L-U decomposition of A and the pivot vector for solving the linear system $Ay = res$ by a forward elimination and backward substitution; the right side of the system and the computed solution are contained in the input-output parameter res of *solve*. The number of iterations and the stopping criterion are determined as considered in [2]. The maximum number of iterations is taken to be $2 * \log_{10}(1/tol) = -2 * \log_{10}(tol)$, that is, twice the number of decimal digits specified by tol. The iterations stop if $normres \leq tol * normx$, where $normx$ is the infinity norm of the first solution, and $normres$ is the infinity norm of the correction res added to x. If $normres \leq tol * normx$, we assume that the iterations have converged. Note that, if the iterations do not converge, then the decomposition and *solve* must be carried out in higher precision.

4.2 Computing the Inverse of a Matrix

We describe a function to compute the inverse of a matrix and some numerical results obtained by applying this function to Hilbert matrices. Numerical results show that higher and higher precision is needed to compute the inverse of an n-by-n Hilbert matrix as n increases.

```
void improve(const matrix & A, const matrix & LU, const vector & b,
             vector & x, const intvector & pivot,
             const real & tol, int & converged)
{
converged = 0;
real normx = inf_norm(x);
if ( normx == 0 )     // x = 0 and the iterations will not improve it
   converged = 1;     // the norm of the error is ≈ 0
else
  {
  int itmax = real2int( -2*log10(tol));
  vector res(size(x));
  real normres;
  for( int i = 1; i <= itmax; i++)
     {
     residual(res, b, A, x);
     solve (LU, res, pivot);
     x += res;
     normres = inf_norm(res);
     if (normres <= tol * normx )
       {
       converged = 1;
       break;
       }
     }
  }
}
```

Figure 4.2: The *improve* function.

4.2.1 An Algorithm to Compute the Inverse of a Matrix

Let A be an n-by-n matrix. Denote the inverse matrix of A by A^{-1}. The columns of A^{-1} are the solutions of the linear systems

$$Ax_i = e_i, \ i = 1, 2, \ldots, n, \tag{4.2}$$

where e_i is the i-th unit vector whose coordinates are all zero except for the i-th, which is one. Therefore, A^{-1} can be computed by solving the linear systems (4.2).

The function *inverse*, Fig. 4.3, computes the inverse of a matrix. It uses the *improve* function, Section 4.1, to obtain more accurate solutions of (4.2). The parameters of *inverse* are: A, the input matrix to be inverted; *tol*, a user-specified tolerance which is used in *improve*; *converged*, an output integer parameter indicating if the tolerance is achieved, *converged* $= 1$, or if it cannot be attained, *converged* $= 0$; and *Ainv*, the computed inverse of A. At the beginning of *inverse*, the precision is saved, and the order of A is returned by the *dim* function and assigned to an integer variable n. The integer vector *pivot* is computed during the decomposition of A and is used in solving the linear systems (4.2). It contains information about row interchanges performed during the *L-U* decomposition carried out with partial pivoting.

Consider the code inside the *while* loop. The original matrix A is copied into the matrix LU. The *decomp* function produces an L-U decomposition of A such that the result is stored in LU. If zero pivots occur during the factorization by *decomp*, the *solve* function should not be used — the *quit* function terminates the program with a message, which is a parameter of *quit*[1]. Otherwise each of the n systems (4.2) is solved. In the innermost *for* loop, vectors e, corresponding to e_i in (4.2), are generated[2]. For each such vector, the linear system $LUx_i = e_i$ is solved by the *solve* function.

The right side of the system is stored in the vector x, and the result of *solve* is

[1]Alternatively, we could do the decomposition in higher precision, but we have not done so since the point of this example is to emphasize iterative improvement.

[2]The operator "()" of C++ is overloaded for the vector and matrix classes, and the indexing begins from 1 instead of 0.

```
void inverse(matrix & Ainv, const matrix & A, const real & tol, int & converged)
{
int p = precsav();

int n = dim(A);
intvector pivot(n);

converged = 0;
while (!converged)
    {
    matrix LU = A;
    int zero_pivot;
    decomp(LU, pivot, zero_pivot);
    if (zero_pivot)
      quit("inverse function : zero pivot in the factorization");

    vector x(n), e(n);
    for( int i = 1; i <= n; i++)
       {
       for( int j = 1; j <= n; j++)
          if(i==j)
            e(j) = 1.0;
          else
            e(j) = 0.0;

       x = e;
       solve(LU, x, pivot);
       improve(A, LU, e, x, pivot, tol, converged);

       if (!converged && precsav() < MAXPREC)
          {
          precset(2*precsav());
          break;
          }
       else
          for ( int k = 1; k <= n; k++)
              Ainv(k,i) = x(k);
       }
    }
precset(p);
}
```

Figure 4.3: The *inverse* function

also contained in x. Then the *improve* function tries to improve the solution obtained by *solve*. As was pointed out, *improve* uses precision $2 * p$ to compute residuals. If the iterations do not converge in *improve*, the variable *converge* has the value 0. In this case, *decomp* and *solve* must be called in higher precision, which is double the current precision; otherwise the solution of the i-th system is copied into the i-th column of *Ainv*. If the iterations do not converge in the highest precision, then there is no guarantee that the result is accurate. The last statement of *inverse* restores the current precision.

4.2.2 Numerical Results

The purpose of our experiments is to study the effect of using higher precision(s) in computing the inverse of a matrix by solving linear systems (4.2). We use the $n \times n$ Hilbert matrix $H_n = (h_{ij})$ given by

$$h_{ij} = 1/(i + j - 1), \ i, j = 1, 2, \ldots, n$$

in these examples. Hilbert matrices are badly conditioned, even for small values of n, and their condition number grows rapidly with increasing n — as Table 4.3 and Table 4.4 show.

Experiment 1. Our test program generates Hilbert matrices of odd orders $3, 5, \ldots, 31$ in precision 16. Then, for each matrix, the *inverse* function is called in precision 2 and tolerance $tol = 1e{-}10$. As a result, the maximum number of iterations computed in *improve* is 20. The values recorded in Table 4.1 are:

1. n — the dimension of the Hilbert matrix;

2. precision — the precision at which the tolerance is satisfied, that is, the precision at which *decomp* and *solve* are called such that the iterations in *improve* converge;

3. correct dec. digits — approximate number of correct decimal digits in x calculated by *solve* computed in the first iteration of *improve* by the formula $digits = -log10(normres/normx)$ if $normres > 0$ and $digits = -log10(tol)$ otherwise [2]. This is also approximately how many correct decimal digits are gained at each iteration;

4. iterations — the number of iterations performed in *improve* at the final precision. Note that if *solve* has produced the desired accuracy, one iteration is used to satisfy the stopping criterion.

Matrices of order 3,5,7,9, and 11 are successfully inverted in precision 2 without using precision 4 except for computing residuals. For $n = 13$, however, iterative improvement does not converge in 20 iterations if *decomp* and *solve* are called in precision 2; but, in precision 4, *solve* produces solutions approximately correct to 19 decimal digits. The next interesting case occurs when $n = 27$. The iterative improvement does not converge in precisions 2 and 4 in 20 iterations, but no iterations are needed in precision 8.

Since for $15 \leq n \leq 25$, iterative improvement does not converge in precision 2, we give only the results for precision 4. For the same reason, we do not supply results for $29 \leq n \leq 31$ in precisions 2 and 4 and for $n = 27$ in precision 2.

Experiment 2. Tables 4.3 and 4.4 show that iterative improvement improves the solution of a linear system if the residual is computed in double precision, and, in general, it does not improve the solution significantly if the residual is computed in the same precision as the L-U factorization and *solve*.

We performed two similar experiments with approximate Hilbert matrices of order $n = 2, 3, \ldots, 8$ generated in precision 1 and of order $n = 11, 12, 13$ generated in precision 2. We describe the first experiment and comment on the results of both afterwards. For each of $n = 2, 3, \ldots, 8$, we computed five matrices by calling the *inverse* function and variants of it in precision 1 with tolerance the machine epsilon of precision 1, $tol = 1.192093e-07$. The first matrix H_n^{-1} in Table 4.2 is computed by the *inverse* function as described earlier in this section. The other matrices are

n	precision	correct dec. digits	iterations
3	2	14.4	1
5	2	12.4	1
7	2	8.8	2
9	2	5.5	2
11	2	2.6	4
13	2	0.2	20
	4	19.0	1
15	4	15.1	1
17	4	12.1	1
19	4	9.3	2
21	4	6.6	2
23	4	3.7	3
25	4	1.2	9
27	4	0.4	20
	8	36.5	1
29	8	33.3	1
31	8	29.8	1

Table 4.1: Numerical results for Experiment 1.

computed by variants of *inverse* as shown in Table 4.2. In the second experiment, *improve* is called in precision 2 with tolerance the machine epsilon of precision 2, $tol = 2.220446049250313e - 16$.

We assume that H_n^{-1} is accurate to full precision (1 or 2), and we approximate the error in the other matrices by

$$\frac{\|Hi_n^{-1} - H_n^{-1}\|}{\|H_n^{-1}\|},$$

where $i = 1, 2, 3, 4$ and $\|\cdot\|$ is the infinity norm.

We also compute the condition number

$$cond(H_n) = \|H_n\| \times \|H_n^{-1}\|$$

for each n and the number of iterations used in *improve* to compute H_n^{-1}. Note that our estimate of the condition number uses an approximate Hilbert matrix and its inverse in the corresponding precision.

matrix	iterations of *improve*	residual precision	increasing the precision in *inverse*
H_n^{-1}	until the solution converges or *itmax*	double	yes
$H1_n^{-1}$	none	—	no
$H2_n^{-1}$	1	double	no
$H3_n^{-1}$	1	same	no
$H4_n^{-1}$	until the solution converges or *itmax*	same	no

Table 4.2: The first matrix is computed by the *improve* function as described earlier in this section, the other matrices are computed by modified *inverse* functions.

n	condition	iter	$\frac{\|\|H1_n^{-1}-H_n^{-1}\|\|}{\|\|H_n^{-1}\|\|}$	$\frac{\|\|H2_n^{-1}-H_n^{-1}\|\|}{\|\|H_n^{-1}\|\|}$	$\frac{\|\|H3_n^{-1}-H_n^{-1}\|\|}{\|\|H_n^{-1}\|\|}$	$\frac{\|\|H4_n^{-1}-H_n^{-1}\|\|}{\|\|H_n^{-1}\|\|}$
2	$2.70e+01$	1	0	0	0	0
3	$7.48e+02$	2	$3.18e-07$	0	$2.34e-06$	$2.34e-06_{02}$
4	$2.84e+04$	2	$1.58e-05$	0	$2.90e-05$	$2.40e-05_{14}$
5	$9.43e+05$	2	$3.82e-04$	$1.42e-07$	$8.20e-04$	$4.55e-04_{14}$
6	$2.81e+07$	4	$1.18e-02$	$1.43e-04$	$1.06e-02$	$3.03e-02_{14}$
7	$6.29e+08$	1	$5.79e-01$	$3.37e-01$	$3.37e-01$	$2.65e-01_{14}$
8	$3.30e+09$	1	$8.62e-01$	$7.55e-01$	$7.51e-01$	$3.68e-01_{14}$

Table 4.3: Precision 1, tolerance $1.192093e-07$.

Consider Table 4.3 first. For $n = 2$ the five matrices are accurate to full precision. In the third column, we have one iteration since it is used to satisfy the stopping criterion in *improve*. For $n = 3, 4$ and 5, one additional iteration of the iterative improvement with residuals computed in double precision is sufficient to obtain accuracy of order $1e-7$, but it is not sufficient for $n = 6$. For $n = 7, 8$, the iterative improvement does not converge in precision 1, but, if *decomp* and *solve* are executed in precision 2, one iteration gives the desired accuracy. Iterations with residuals computed in precision 1 can improve the results slightly, but can also produce worse results. The subscript in the last columns denotes the number of iterations used in *improve*.

Table 4.4 lists similar results for $n = 11, 12,$ and 13. One iteration of *improve* with residuals computed in double precision does not give accurate enough results, and iterations with residuals computed in precision 2 improve the solutions slightly. For $n = 13$ the iterative improvement does not converge in precision 2, but one

n	condition	iter	$\frac{\|\|H1_n^{-1}-H_n^{-1}\|\|}{\|\|H_n^{-1}\|\|}$	$\frac{\|\|H2_n^{-1}-H_n^{-1}\|\|}{\|\|H_n^{-1}\|\|}$	$\frac{\|\|H3_n^{-1}-H_n^{-1}\|\|}{\|\|H_n^{-1}\|\|}$	$\frac{\|\|H4_n^{-1}-H_n^{-1}\|\|}{\|\|H_n^{-1}\|\|}$
11	$1.23e+15$	5	$7.01e-04$	$4.93e-07$	$6.58e-04$	$5.32e-04_{32}$
12	$4.04e+16$	13	$5.16e-02$	$2.67e-03$	$1.96e-02$	$2.04e-02_{32}$
13	$5.12e+18$	1	$9.10e-01$	$8.27e-01$	$8.38e-01$	$6.12e-01_{32}$

Table 4.4: Precision 2, tolerance $2.220446049250313e-16$.

iteration is sufficient in precision 4.

4.3 The Complex Natural Logarithm

The complex natural logarithm of $z = x + iy = re^{i\theta}$ can be expressed as

$$\begin{aligned} log(z) &= log(r) + i\theta \\ &= \frac{1}{2}log(x^2 + y^2) + iarg(z), \end{aligned} \tag{4.3}$$

where $-\pi < \theta \leq \pi$, and the logarithm on the right side is the real natural algorithm.

An implementation of the complex natural logarithm and detailed error analysis can be found in [4]. It uses the exception handling construct described in Chapter 3, but does not employ precision control.

In this section, we present two functions to compute the complex natural logarithm: *clog*, Fig. 4.4, as given in [4] and *clog2*, Fig. 4.5, which is similar to *clog*, except that *clog2* uses double precision in critical parts to avoid exceptions and loss of accuracy.

The functions compute the imaginary part in a similar way. The variables M and m are maximum and minimum absolute values of the real and imaginary parts of the complex argument. If $M = 0$, the exception flag is raised, and the functions set the real part of $answer$ to $-\infty$ (see Section A.2.1 for a description of minus infinity) and its imaginary part to 0. If $M \neq 0$, both *clog* and *clog2* use *atan2* to compute the argument of the complex number (*atan2*(x,y) computes the arc tangent of y/x). If *atan2* underflows, the *handle* block is carried out, setting the imaginary part to 0 and the real part to $\log(M)$.

If $m = 0$, both *clog* and *clog2* set the real part of the result to $\log(M)$. Our implementation, Fig. 4.5, differs from that of [4] mainly in how it computes the real part when both M and m are distinct from zero.

If $M \notin (1/2, \sqrt{2})$, which ensures $M^2 + m^2 \notin (1/2, 2)$, overflows or underflows can arise in the evaluation of $M^2 + m^2$, but $log(M^2 + m^2)$ is a representable floating-point number in the working precision. The technique suggested in [4] for handling these exceptions is similar to that used in the *hypot2* function, Fig. 3.3: in *clog*, we scale the arguments, compute the logarithm and then unscale the result. The solution used in *clog2* is more straightforward: if an exception arises in computing $M^2 + m^2$ (the second enable block), handle the exception by repeating the computation in double precision.

If $M \in (1/2, \sqrt{2})$, then $M^2 + m^2 \in (1/4, 4)$. The function log1p(x) computes log(1+x), but more accurately than could be obtained from $log(1+x)$ when x is small. In both *clog* and *clog2*, we compute the expression $M^2+m^2-1 = (M-1)(M+1)+m^2$. In *clog*, we use the same precision, so there is a possibility of serious cancellation, and the error analysis is complicated [4]. In *clog2*, this expression is safely evaluated in double precision, and the result is assigned to the variable *tmp* which is in the current precision. Underflow can arise only in the assignment to *tmp* and only if $M = 1$ and m^2 underflows in the current precision, in which case the real part of the result is set to 0 in the handle block.

The *atan2* and *log1p* functions are not yet implemented, but double precision versions exist in most C math libraries.

By using precision control, we obtain a complex natural logarithm function that is easier to implement and understand. The error analysis is simplified as well since the main difficulty is to analyze the case in which cancellation arises in the evaluation of $M^2 + m^2 - 1$. Moreover, the error bound in the final result is smaller when double precision is used.

```
complex clog(const complex &z)
{
real M = max( fabs(z.re), fabs(z.im));
real m = min( fabs(z.re), fabs(z.im));
real sqrt2 = sqrt(2.0);
real log2 = log(2.0);
complex answer;

if (M == 0)
   {
   setexc();
   answer.re = −Infinity();
   answer.im = 0;
   return answer;
   }

enable
    answer.im = atan2(z.im, z.re);
handle                  // must be underflow in atan2
    answer.im = 0;
    answer.re = log(M);
    return answer;
end

if (m == 0)
    answer.re = log(M);
else
   if (M <= 0.5 || M >=sqrt2)
      enable
          answer.re = 0.5*log(M*M+m*m);
      handle             // must be underflow or overflow in M*M+m*m
          if ( 2*(ilogb(M)−ilogb(m)) > precdig()+1 )
              answer.re = log(M);
          else // scale so that the exceptions are avoided,
               //and the two terms in answer.re are of the same sign
               {
               int scale;
               if ( M > 1.0) // must have been overflow
                  scale = ilogb(M);
               else             // must have been underflow
                  scale = ilogb(M) + 2;
```

```
                real scaledM = scalb(M, −scale);
                real scaledm = scalb(m, −scale);
                real scaledr = scaledM∗scaledM+scaledm∗scaledm;
                answer.re = scale∗log2 + 0.5∗log(scaledr);
                }
            end
        else    // 1/2 < M < sqrt2
            enable
              answer.re = 0.5∗log1p( (M−1)∗(M+1) + m∗m);
            handle // must be underflow in m∗m
              answer.re = log(M);
            end

return answer;
}
```

Figure 4.4: Complex natural logarithm without precision control

```
complex clog2(const complex &z)
{
real M = max( fabs(z.re), fabs(z.im));
real m = min( fabs(z.re), fabs(z.im));
real sqrt2 = sqrt(2.0);
complex answer;

if (M == 0)
   {
   setexc();
   answer.re = -Infinity();
   answer.im = 0;
   return answer;
   }

enable
    answer.im = atan2(z.im, z.re);
handle                  // must be underflow in atan2
    answer.im = 0;
    answer.re = log(M);
    return answer;
end

if (m == 0)
    answer.re = log(M);
else
   if (M <= 0.5 || M >=sqrt2)
     enable
         answer.re = 0.5*log(M*M+m*m);
     handle             // must be underflow or overflow in M*M+m*m
         int p = precsav();
         precset(2*p);
         answer.re = 0.5*log(M*M+m*m);
         precset(p);
     end
```

```
      else      // 1/2 < M < sqrt2
         {
         enable
            real tmp;
            int p = precsav();
            precset(2*p);
            tmp = (M-1)*(M+1) + m*m;
            precset(p);
            answer.re = 0.5*log1p(tmp);
         handle              // M = 1 and m*m underflows in precision p
            answer.re = 0;
         end
         }
return answer;
}
```

Figure 4.5: Complex natural logarithm using precision control

Chapter 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 precisions, since common difficulties in numerical algorithms design, such as dealing with cancellations, overflows and underflows, can usually 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.

An important problem is to specify precisions $4, 8, 16, \ldots$, that is, how much to increase the exponent range and the significand length when increasing precision. The exponent range must be more than double when doubling precision, but how much more? Doubling the length of the significand works well in practice, but should we more than double it?

Since an implementation of variable-precision arithmetic requires considerable system support, another problem is to determine the largest precision which is sufficient for practical purposes. Precision control also demands dynamic allocation and deallocation of data, so a nonnegligible part of program execution can be spent on memory management; therefore these operations should be carried out efficiently.

In contrast to precision control, the proposed exception handling is much easier to implement, and it can work efficiently on serial 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 with 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.

Appendix A

SciLib User's Guide

A.1 Introduction

In this user's guide, we describe a C++ variable-precision and exception handling library, named *SciLib*, and explain how to use it. The guide is self-contained, so some material from the previous chapters is repeated.

SciLib is 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. In general, using the new data type does not differ from using any standard scalar data type of C++.

In contrast to most programming languages precision of computation involving objects of type *real* can be easily altered during program execution. As a consequence, results of accuracy as high as the implementation allows can be obtained.

The exception handling is simple and easy to use. The exception handler treats only numerical exceptions and does not distinguish between different types of numerical exceptions. The only information available to the user is whether any numerical exception has occurred.

Since real objects are created in the precision at the time of declaration, and the computations are carried out in the same precision, functions producing results in different precisions can be easily implemented. An advantage is that only one numerical library is needed, rather than single and double versions of the same library.

The library is compiled with the Gnu C++ compiler, version 2.3.3 The only machine dependence is related to the representation of *float* and *double* data types of C++ since their values can be assigned to floating-point numbers of SciLib and vice versa.

This guide is organized as follows. Section A.2 considers floating-point numbers in different precisions, their representation, and conversion and arithmetic operations involving them. Section A.3 describes how real objects can be declared and initialized, and operations with real objects. Precision control is considered in Section A.4. Exception handling is explained in Section A.5. Section A.6 contains a listing of the available functions of SciLib with short descriptions.

A.2 Floating-Point Numbers and Operations in Variable Precision

In this section, we describe floating-point numbers in different precisions, used in SciLib, and specify conversion and arithmetic operations between them.

A.2.1 Floating-Point Numbers

The SciLib package uses a binary representation of nonzero floating-point numbers of the form

$$(-1)^s \times d_0.d_1d_2\ldots d_m \times 2^e \tag{A.1}$$

where $s \in \{0,1\}$, $(-1)^s$ is the *sign*; $d_0.d_1d_2\ldots d_m$ is the *significand*, m is the number of binary digits after the binary point, $d_0 = 1$, $d_i \in \{0,1\}$, $i \in 1\ldots m$; and the integer e is an *exponent* satisfying $E_{\min} \leq e \leq E_{\max}$. We refer to digits $d_1, d_2, \ldots, d_m$ to the right of the binary point in the significand as the *fraction*.

s	*exponent*	*fraction*

Figure A.1: The floating-point format.

If the significand of a floating-point number is as in (A.1) but the exponent $e > E_{\max}$, we say that the floating-point number *overflows* in the format (A.1). If the exponent $e < E_{\min}$, the floating-point number *underflows* in (A.1). Note immediately that if the number *overflows*, *underflows*, or its value is *zero*, it cannot be expressed in the form (A.1). We consider later in this section how such values are represented.

Floating-point numbers (A.1) are stored as shown in Fig. A.1. The sign, $(-1)^s$, is coded by storing s in the leading bit; next to it is stored the exponent, and after that the fraction part of the significand. Since $d_0 = 1$, it need not be stored explicitly.

We refer to the floating-point format, Fig. A.1, having 8 exponent and 23 fraction bits, as precision 1 or *single* precision. If the exponent and fraction bits are 11 and 52 respectively, we say that the precision is 2 or *double*. The current implementation uses 12 precisions denoted by $1, 2, 4, \ldots, 2048$, that is, by the powers of 2 from 0 to 11. The precisions and corresponding exponent and fraction bits, and exponent ranges in binary and decimal are given in Table A.1. Since different precisions have different characteristics, we use the notation $parameter(p)$ to denote the parameter corresponding to precision p, for example, $m(p)$, $E_{\max}(p)$, $E_{\min}(p)$, $E_{min_{10}}(p)$, $E_{max_{10}}(p)$, and $exp_{bits}(p)$ listed in Table A.1.

The exponent is stored using a biased notation. In fact, the stored value of the exponent of precision p is $e + 1 - E_{\min}(p)$, where $1 - E_{\min}(p)$ is the bias value for precision p. The length of the format in bits, Fig. A.1, is given by the formula $1 + m(p) + exp_{bits}(p)$.

A floating-point number in SciLib is characterized by its *precision*, *value*, and *type*.

p	m	*exp. bits*	Exponent range			
			binary		decimal $\approx$	
			$E_{\min}$	$E_{\max}$	$E_{\min_{10}}$	$E_{\max_{10}}$
1	23	8	-2^7+2	2^7-1	$-3.7930E+01$	$3.8231E+01$
2	52	11	$-2^{10}+2$	$2^{10}-1$	$-3.0765E+02$	$3.0795E+02$
4	114	13	$-2^{12}+2$	2^{12}	$-1.2324E+03$	$1.2330E+03$
8	240	15	$-2^{14}+2$	2^{14}	$-4.9315E+03$	$4.9321E+03$
16	494	17	$-2^{16}+2$	2^{16}	$-1.9728E+04$	$1.9728E+04$
32	1004	19	$-2^{18}+2$	2^{18}	$-7.8913E+04$	$7.8913E+04$
64	2026	21	$-2^{20}+2$	2^{20}	$-3.1565E+05$	$3.1565E+05$
128	4072	23	$-2^{22}+2$	2^{22}	$-1.2626E+06$	$1.2626E+06$
256	8166	25	$-2^{24}+2$	2^{24}	$-5.0504E+06$	$5.0504E+06$
512	16356	27	$-2^{26}+2$	2^{26}	$-2.0202E+07$	$2.0202E+07$
1024	32738	29	$-2^{28}+2$	2^{28}	$-8.0807E+07$	$8.0807E+07$
2048	65504	31	$-2^{30}+2$	2^{30}	$-3.2323E+08$	$3.2323E+08$

Table A.1: Fraction bits, exponent bits, and exponent ranges of different precisions.

The *type* can be

$$type \equiv \begin{cases} \textit{zero}, & (0) \\ \textit{plus infinity}, & (+\infty) \\ \textit{minus infinity}, & (-\infty) \\ \textit{not a number}, & (NaN) \\ \textit{number} & \end{cases}$$

If the type is different from *number*, the value is encoded by its type. The following cases are possible:

1. the value is 0. Then $type \equiv 0$;
2. $d_0 = 1$ but the exponent e is
   (a) $e > E_{\max}(p)$. The number overflows in that precision and $type \equiv -\infty$ or $type \equiv +\infty$ depending on the sign
   (b) $e < E_{\min}(p)$. The number underflows in that precision and it is set to zero. The type is $type \equiv 0$;
3. if the operation is invalid, such as $(+\infty)+(-\infty)$, $0 \times \infty$, $0/0$, and ∞/∞, square root of a negative number, etc., the result is a NaN. The type is $type \equiv NaN$.

We note the following:

- denormalized numbers, $e = E_{\min}(p)$, $d_0 = 0$, $d_i \neq 0$ for some i, are not supported in SciLib. If they occur during computations, they underflow to zero;
- the zero is unsigned;
- the sets of representable numbers in precisions 1 and 2, excluding denormalized numbers and signed zeros, are the same as those in *single* and *double* of the IEEE Standard [1];
- the following relation between $m(2p)$ and $m(p)$, $p < 2048$, holds

$$m(2p) \geq 2m(p) + t,\ \ t \geq 6,$$

  so doubling the precision ensures more than double the fraction bits. The parameter t increases slightly as p increases. For $1 < p < 2048$ the exponent range of precision $2p$ is nearly four times larger than the exponent range of precision p.

- The range of the floating-point numbers in each precision can be easily determined from Table A.1 and (A.1). For example, the maximum value of a floating-point number in precision p is $1.\underbrace{11\ldots1}_{m(p)}\times 2^{E_{\max}(p)} = (2 - 2^{-m(p)}) \times 2^{E_{\max}(p)}$. The largest positive number in precision p in decimal is $\approx 2 \times 10^{E_{\max_{10}}(p)}$. The absolute value of the smallest positive floating-point number is given by $1.\underbrace{00\ldots0}_{m(p)}\times 2^{E_{\min}(p)}$ which is $\approx 10^{E_{\min_{10}}(p)}$.

A.2.2 Roundings

Denote the set of real numbers by R and the set of floating-point numbers in precision p by R_p. The mapping, $round_p : R \to R_p$ is a rounding if $round_p$ satisfies the following properties:

$$round_p(x) \ \ = \ \ x,\ \text{for all } x \in R_p \tag{A.2}$$

$$x \leq y \Rightarrow round_p(x) \leq round_p(y) \tag{A.3}$$

By the definition, there are no representable floating-point numbers in precision p between x and $round_p(x)$. Consider the following roundings, denoted by *near*, *up*, and *down*, $x \in R$ (assuming no exceptions arise).

1. *To the nearest*: $near_p(x)$ is the nearest floating-point number in precision p to x, where in the case of a tie, $near_p(x)$ is the nearest even number.

2. *Toward plus infinity*: $up_p(x)$ is the smallest representable number in precision p such that $x \leq up_p(x)$.

3. *Toward minus infinity*: $down_p(x)$ is the largest representable number in precision p such that $x \geq down_p(x)$.

A.2.3 Conversions Between Floating-Point Numbers

Let x be a floating-point number in precision p whose type is *number*. Its value is converted to a floating-point number in precision q if x can be represented in the format of precision q.

The following cases are possible.

1. $q = p$: the value of x in precision q is the same as in precision p.

2. $q > p$: the binary representation of x is expanded with trailing zeros.

3. $q < p$: a rounding can take place. Note that the result of the conversion can overflow or underflow.

A.2.4 Arithmetic Operations in Variable Precision

Let x and y be floating-point numbers in precisions p and q. A floating-point operation in variable precision is defined by

$$x \circ_l y = round_l(x \circ y) \tag{A.4}$$

where $\circ \in \{+, -, *, /\}$, l is a particular precision, $x \circ y$ is the exact result of the operation, and $round \in \{near, up, down\}$. The definition ensures maximum accuracy in the sense that there are no representable floating-point numbers in precision l between the mathematically exact and the rounded results.

A.3 Using Objects of Class Real

The representation of the floating-point numbers, as described in section A.2, and the operations including them are implemented by a C++ class named *real*. The *real* data type can be used like the types *float* and *double* of C++. In this section, we consider how real objects can be declared, initialized, and used in arithmetic, relational, and input and output operations. Simple examples are given as illustrations. In this implementation of SciLib, we assume that *float* and *double* data types agree with the IEEE Standard[1], to simplify the conversion of *float* and *double* to *real*. The arithmetic operations are carried out in the specified precision. Its value is 1 by default, and it can be changed by the function *precset*(*int*), Section A.4, where the integer parameter is as specified in Table A.1. The value of the precision at the time of computation will be denoted as the *current precision*.

In order to use the *real* data type, the user must include the file *real.hpp* at the beginning of the program files.

A.3.1 Declaration of Real Objects

A real object can be declared like any other object. For example, the following statement

real a1, a2;

declares two variables $a1$ and $a2$ of type *real* in the current precision. In the following example, the precision of $b1$ is 1 and the precision of $b2$ is 4.

```
real b1;
precset(4); // first call of precset during program execution
real b2;
```

Similarly, if real objects appear in aggregate types, such as arrays, structures, and classes, their precision is the current precision when they are declared. For example, an array of 10 elements of type *real* in precision 4 can be declared by

```
precset(4);
real vector[10];
```

Real objects are created during program execution. For example, when the following code is executed, it reads an integer number, sets the precision, and creates a real object a in precision p.

```
int p;
cin >> p;
precset(p);
real a;
```

Since C++ allows declarations to appear anywhere in the program, real objects can be declared where they are needed.

A.3.2 Conversions to *real* in Arithmetic and Relational Operations

If in a binary arithmetic or relational operation one of the operands is of type *real* and the other one is of type *float*, *double*, *int*, or *decimal string*, it is implicitly converted to *real* in the current precision, and then the operation is carried out. The order of the evaluation of an arithmetic expression depends on the compiler.

We note some details related to the conversions. In an arithmetic operation, where one of the operands is of type *real*, denote the other by *op*. Let the precision of computation be p.

1. If *op* is of type *float*, $p = 1$, and *op* is a denormalized number, then the converted value of *op* will underflow to zero.

2. If *op* is of type *double*, $p = 2$, and *op* is a denormalized number, the result of the conversion will underflow to zero.

3. If *op* is of type *int* it is first converted to *double*, and then a conversion to *real* is performed.

4. If *op* is a *decimal* string it is first converted to double, and then the double value is converted to *real*. We have not yet implemented conversions between decimal strings and real objects in precisions higher than two. A decimal string, representing a floating-point number whose value cannot be represented in the double format will cause overflow or underflow when converted to double. The double value will be zero or signed infinity.

A.3.3 Assignments to Real Objects

In this version of SciLib, the left side of an assignment operator cannot be of type *float*, *double*, or *int* if the right side is of type *real*. In such cases, conversion functions must be used explicitly, see section A.6.

Let the left side of an assignment operator be of type *real* in precision p. If the precision of the operands differ, the value of the right side is converted to a floating-point number in precision p and assigned to the real object. The right side can be

1. a float or double variable or constant;

2. an integer variable or constant;

3. a decimal string;

4. a real variable or constant;

5. an expression that evaluates to float, double, integer, or real result, and the result is rounded to a floating-point number in precision p and assigned to the real object.

The conversions are as specified in subsections A.2.3 and A.3.2. If the rounding mode is not explicitly set, it is by default to the nearest, see subsections A.2.2, A.3.5.

Since C++ objects can be initialized when declared, statements such as

real a = b;

can be used. The right side in the above assignment, *b*, can be of any scalar data type, decimal string, or an expression that evaluates to a scalar data type. Real constants can be declared and initialized by using the keyword *const*. For example, the following statement

const real pi = 3.141592653589793;

declares and initializes the constant *pi* in the current precision. The right side of the assignment to a constant object can be of any scalar data type, decimal string, or an expression that evaluates to a scalar data type. The value of a constant cannot be changed.

A.3.4 Arithmetic and Relational Operations

Arithmetic and relational operator symbols of C++ are overloaded, and arithmetic expressions involving real objects can be written without function calls.

> *Arithmetic operations, when at least one of the operands is of type real, are carried out in the current precision.*

Each intermediate result of an arithmetic operation is rounded to the format of the current precision, where the rounding is to the nearest if it is not otherwise specified, see subsection A.3.5. The overloaded operators for the class *real*, except the input and output operators, are listed in Table A.2.

A.3.5 Rounding Mode Control

The rounding mode is set by default to be round-to-nearest or nearest even in case of a tie. It can be changed to round toward plus infinity or minus infinity. The value of the rounding mode is global, that is, its value is valid in all called functions.

Operator	Type of the operands	
	left side	right side
$=$	real	real float double int decimal string
$==$ $!=$ $<=$ $<$ $>$ $>=$	real, float, double, int, decimal string	real, float, double, int, decimal string
$+$ $-$ $*$ $/$		

Table A.2: Overloaded operator symbols for the type real.

The following functions can be used to test and set the rounding mode:

rmode rmget(); return the rounding mode;

void rmset(rmode); set the rounding mode.

The type *rmode* is declared as an enumerate data type with possible values *up, down*, and *near* corresponding to rounding modes to plus infinity, minus infinity, and to the nearest.

A.3.6 Input and Output of Real Objects

The input and output operators of C++, “>>” and “<<”, are overloaded for *real* object. A floating-point number can be entered from the standard input and assigned to a *real* object by the operator “>>”. For example,

```
real a;
cin >> a;
```

A real object can be sent to the standard output by the operator "<<"

cout << a;

As mentioned earlier, conversion between decimal strings and floating-point numbers in precisions greater than two is not yet implemented. As a consequence, when a floating-point number is entered as a decimal string, it is first converted to *double* and then the double value is assigned to a *real* object. The reverse occurs on output: the floating-point number is converted to *double*, and then the double value is output as a decimal string.

Example. Consider the evaluation of the following arithmetic expression [5]

$$f = 333.75b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2) + 5.5b^8 + \frac{a}{2b} \tag{A.5}$$

for $a = 77617.0$ and $b = 33096.0$.

The function *rump_example*, Fig. A.2, computes (A.5) in variable precision. Note that the precision is not specified in the function. The variables a, b, $a2$, $b2$, $b3$ are declared in the current precision at the time of calling. The expression in the return statement is also evaluated in the current precision.

```
// file rump.c
#include <real.hpp>

real rump_example()
{
real a = 77617.0;
real b = 33096.0;
real a2 = a*a;
real b2 = b*b;
real b3 = b2*b;
return 333.75*b3*b3 + a2*(11.0*a2*b2 - b3*b3 - 121.0*b2*b2 - 2.0) +
        5.5*b3*b3*b2 + a/(2.0*b);
}
```

Figure A.2: The function *rump_example* in variable precision.

The function is called in the main program, Fig A.3, in different precisions, and the result is computed in the current precision. $MAXPREC$ contains the value of the largest precision, currently 2048.

```
// file main.c
#include <real.hpp>

extern real rump_example();

main()
{
for (int i = 1; i <= MAXPREC; i * = 2)
    {
    precset(i);
    cout << "Precision: " << precsav() << " " << rump_example() << "\n";
    }
}
```

Figure A.3: Computing the function *rump_example* in precisions $1, 2, \ldots, MAXPREC$.

The output of the main program is shown in Fig. A.4.

Precision: 1	$-1.267651e+30$
Precision: 2	$-2.361183241434823e+21$
Precision: 4	1.172603940053179
Precision: 8	-0.8273960599468214
Precision: 16	-0.8273960599468214
Precision: 32	-0.8273960599468214
Precision: 64	-0.8273960599468214
Precision: 128	-0.8273960599468214
Precision: 256	-0.8273960599468214
Precision: 512	-0.8273960599468214
Precision: 1024	-0.8273960599468214
Precision: 2048	-0.8273960599468214

Figure A.4: The expression (A.5) computed in precisions $1, 2 \ldots, MAXPREC$.

A.4 Precision Control

In this section, we give the information necessary for using precision control in SciLib.

The result of an arithmetic operation is evaluated in the current precision if at least one of the operands is of type *real*. If none of the operands is of type *real*, then the precision of the result cannot be controlled, and it depends on the compiler used.

The precision value is global and is accessible in all files used. Initially it is set to 1. Its maximum value is given by the constant $MAXPREC$, currently 2048. The following functions can be used for precision control.

int precsav(); save the value of the current precision;

void precset(int); set the value to the current precision. The value has to be one of those listed in Table A.1. If it is not, the program terminates with a message (see Section A.7).

An Example: Dot Product. To illustrate the use of precision control, consider the double precision dot product of two vectors a and b, as shown in Fig. A.5. Two arrays of real objects, a and b, are declared in the current precision, p. Then, the precision is saved and set to $2 * p$. The real variable tmp is declared in precision $2 * p$ and initialized to 0. The dot product is computed in precision $2 * p$: that is, the multiplications and additions are carried out in double precision. In the last statement, the variable tmp, containing the accumulated products, is assigned to the variable dot, which is in precision p. A rounding to the format of precision p normally takes place here.

A.5 Exception handling

The exception handling in SciLib can be used only for handling numerical exceptions arising in operations involving real objects or when a numerical exception is explicitly set as shown below. A numerical exception arises in the following situations:

```
real a[n], b[n];
// determine values of 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 A.5: Dot Product.

- divide-by-zero — the result of the division operation is signed infinity depending on the sign of the numerator, provided that the numerator is different from zero;
- overflow — the result of a floating-point or assignment operation overflows in the current precision, the result is signed infinity;
- underflow — the result of a floating-point or assignment operation underflows in the current precision, the result is zero;
- invalid operation — for example, $(+\infty) + (-\infty)$, $0 \times \infty$, $0/0$, and ∞/∞, etc., the result of the operation is NaN.

If any of the above situations occurs an exception flag is raised. The value of the flag is global. Initially the flag is cleared. The user can explicitly raise it by the function

void setexc() raises the exception flag.

The exception handling uses the *enable–handle–end* construct shown in Fig. A.6. The *enable* block is entered if no exceptions have occurred before it; otherwise the program terminates. If an exception occurs in the *enable* block, the exception flag is cleared, and the *handle* block is executed. If no exception arises in the *enable* block, the *handle* is not executed. Both *enable* and *handle* constructs can be nested.

If a *real* variable is uninitialized and takes part in any operation, then execution terminates with a message, see Section A.7.

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

Figure A.6: The exception handling construct.

In order to use the exception handling mechanism, the user must include the file *exc.h* at the beginning of the program files.

An Example: Hypotenuse Function. Consider the function *hypot* which computes $\sqrt{x^2+y^2}$, Fig. A.7. It first tries to compute the result in the current precision in the *enable* block. If an overflow or underflow exception occurs, the computations are repeated in double precision. There is still a possibility of overflow in the *handle* block since *answer* is in a lower precision than the result of the *sqrt* function. The calling function can detect the 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                    // overflow or underflow has occurred
  int p = precsav();
  precset(2 * p);
  answer = sqrt(x * x + y * y);
  precset(p);
end
return answer;
}
```

Figure A.7: Hypotenuse function.

A.6 SciLib Functions

We list the available functions in the library.

A.6.1 Functions Recommended in the Appendix to the IEEE Standard.

real scalb(const real &, const real &); scalb(x,n) returns $x \times 2^n$ by modifying the exponent of the floating-point number rather than explicitly computing 2^n and multiplying by x. The function raises an exception if n is not an integral value and returns NaN. The result for $x = \pm\infty$ or NaN is not specified in the IEEE Standard [1]. In SciLib, scalb(NaN,n) = NaN and scalb($\pm\infty$,n) = $\pm\infty$, and raises the exception flag.

real logb(const real &); logb(x) returns the unbiased exponent of x in the format of x, except that logb(NaN) = NaN, logb($\pm\infty$) = $+\infty$, and logb(0) = $-\infty$ and raises the exception flag for these arguments.

real nextafter(const real &, const real &); nextafter(x,y) returns the next representable floating-point number after x in the precision of x and in the direction toward y. If $x = y$ then the result is x. If one of the operands is NaN, then the result is NaN. An exception is raised if x is finite but the result is infinite or $x \neq 0$ but the result is zero. The largest positive floating-point number in the precision of x and $+\infty$ are neighbors, as are the smallest negative number in that precision and $-\infty$.

real copysign(const real &, const real &); copysign(x,y) returns $|x|$ with the sign of y. If one of the arguments is NaN, then the result is NaN.

int finite(const real &); finite(x) returns 1 (true) if $-\infty < x < +\infty$ and returns 0 (false) otherwise.

int isnan(const real &); isnan(x) returns 1 if x is NaN and 0 otherwise.

A.6.2 Additional Functions

real scalbn(const real &, int); scalbn(x,n) returns $x \times 2^n$ as in scalb except that n must be an integer value; scalbn(NaN,n) = NaN and scalb($\pm\infty$,n) = $\pm\infty$, and raises the exception flag.

int ilogb(const real &); ilogb(x) returns the unbiased exponent of x in integer format; ilogb(NaN) = $MAXINT$, ilogb($\pm\infty$) = $MAXINT$ and ilogb(0) = $-MAXINT$, where $MAXINT$ is the largest positive integer number. The function never generates an exception.

int signbit(const &); signbit(x) returns 0 if $x \geq 0$ and 1 if $x < 0$; signbit(NaN) = 0;

real significant(const real &); significant(x) returns scalb(x, −ilogb(x)) if x is finite, and x otherwise.

int ispinf(const real &); ispinf(x) returns 1 if x is $+\infty$ and 0 otherwise.

int isminf (const real &); isminf(x) returns 1 if x is $-\infty$ and 0 otherwise.

int iszero (const real &); iszero(x) returns 1 if x is 0 and 0 otherwise.

A.6.3 Conversion Functions

double real2double(const real &); real2double(x) returns x in the *double* format of the C++ compiler. An exception is raised if the result overflows or underflows.

int real2int(const real &); real2int(x) computes int(real2double(x)).

A.6.4 Rounding Mode Control Functions

void rmset(rmode); rmset(x) sets the rounding mode. The parameter x is of an enumerate data type *rmode* with values *near*, *up*, and *down* corresponding to roundings to the nearest, plus infinity, and minus infinity.

rmode rmget(); rmget() returns the current rounding mode: one of *near*, *up*, or *down*.

The following functions can be used for arithmetic operations with rounding toward minus infinity:

real plusdn(const real &, const real &)

real minusdn(const real &, const real &)

real timesdn(const real &, const real &)

real divdn(const real &, const real &)

The following functions can be used for arithmetic operations with rounding toward plus infinity:

real plusup(const real &, const real &)

real minusup(const real &, const real &)

real timesup(const real &, const real &)

real divup(const real &, const real &)

The following functions can be used for arithmetic operations with rounding to the nearest:

real plusnr(const real &, const real &)

real minusnr(const real &, const real &)

real timesnr(const real &, const real &)

real divnr(const real &, const real &)

A.6.5 Precision Control Functions

If the precision value is not one of $1, 2, 4, \ldots, MAXPREC$ in any of the functions described in this subsection which require a precision value, then the program terminates with a message (see Section A.7).

void precset(int); precset(n) sets the current precision to n.

int precsav(); returns the current precision.

real precof(const real &); precof(x) returns the precision of x.

int precdig (int); If called without an argument, precdig() returns the number of significant digits in a floating-point number in the current precision. Otherwise, precdig(n) returns the number of significant digits in a floating-point number in precision n.

void changeprec(const real &, int); changeprec(x,n) changes the precision of x to n. If x is finite and $precof(x) < n$, then a is expanded with zeros; if $precof(x) > n$, then a is rounded to precision n.

A.6.6 Exception Set Function

void setexc(); raises the exception flag.

A.6.7 Output Functions

void printx(const real &); printx(a) prints the binary representation of a in hexadecimal notation.

A.6.8 Mathematical Functions

real max(const real &, const real &); max(x,y) returns the maximum value of x and y.

real min(const real &, const real &); min(x,y) returns the minimum value of x and y.

real fabs(const real &); fabs(x) returns the absolute value of x.

real sqrt(const real &); sqrt(x) returns correctly rounded value of $\sqrt{x}$ in the current precision.

int expMin(int); expMin() and expMin(n) return respectively the minimum value of the exponent of a floating-point number in the binary format of the current precision and in precision n.

int expMax(int); expMax() and expMax(n) return respectively the maximum value of the exponent of a floating-point number in the binary format of the current precision and in precision n.

real epsilon(int); epsilon() returns machine epsilon of the current precision, and epsilon(n) returns machine epsilon of precision n, where machine epsilon is the distance from 1 to the next larger floating-point number.

real rmax_normal(); returns the largest positive floating-point number in the current precision.

real rmin_normal(); returns the smallest positive floating-point number in the current precision.

real NaN(); returns NaN in the current precision and raises the exception flag.

real Infinity(); returns $+\infty$ in the current precision and raises the exception flag.

A.7 Error messages

SciLib prints the following messages and terminates in the situations described below.

- - SciLib Error - - Uninitialized variable. An uninitialized *real* variable is an operand in an arithmetic, relational, or output operation, or it is a right side in an assignment statement.

- - SciLib Error - - Exception before enable. An exception has occurred before an enable block. In order to continue the execution, it must be handled.

- - SciLib Error - - Invalid precision value. The precision value is not one of $1, 2, 4 \ldots, MAXPREC$.

Bibliography

[1] American National Standards Institute, New York. *IEEE standard for binary floating-point arithmetic*, 1985. ANSI/IEEE Std. 754–1985.

[2] George Forsythe and Cleve B. Moler. *Computer Solution of Linear Algebraic Systems*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1967.

[3] George E. Forsythe, Michael A. Malcolm, and Cleve B. Moler. *Computer Methods for Mathematical Computations*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1977.

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

[5] S.M. Rump. Algorithms for verified inclusions–theory and practice. In R.E. Moore, editor, *Reliability in Computing*, pages 109–126. Academic Press, San Diego, Calif, 1988.

[6] J.H. Wilkinson. *Rounding Errors in Algebraic Processes*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1963.