C–XSC

A C++ Class Library for Extended Scientific Computing

A. Wiethoff

Institute for Applied Mathematics
Prof. Dr. U. Kulisch
University of Karlsruhe
P. O. Box 6980
D-76128 Karlsruhe
Germany
http://www.uni-karlsruhe.de/~iam/lehrstuhl2-e.html

July 1996

XSC Languages

Language eXtensions for Scientific Computation provide all features indispensible for modern numerical software development, such as

- Operator concept (user-defined operators)
- Overloading concept
- Module concept
- Dynamic arrays
- Controlled rounding
- Predefined arithmetic data types *real*, (*extended real*), *complex*, *interval*, *complex interval*, and corresponding vector and matrix types
- Predefined arithmetic operators of highest accuracy for the arithmetic data types
- Predefined elementary functions of highest accuracy for the arithmetic data types
- Data type dotprecision for the exact representation of dot products
- Library of mathematical problem-solving routines with automatic result verification and high accuracy

The following XSC languages are available:

Language (Type of extension)	Available from
PASCAL–XSC (Compiler)	Numerik Software GmbH
ACRITH–XSC (FORTRAN 77 extension)	IBM
FORTRAN 90–XSC (FORTRAN 90 modules)	(in preparation)
C–XSC (C++ class library)	Numerik Software GmbH

C – XSC

A C++ Class Library for Extended Scientific Computing

ANDREAS WIETHOFF*

Abstract

C – XSC is a tool for the development of numerical algorithms delivering highly accurate and automatically verified results. It provides a large number of predefined numerical data types and operators. These types are implemented as C++ classes. Thus, C – XSC allows high-level programming of numerical applications in C and C++. The C – XSC package is available for many computers with a C++ compiler translating the AT&T language standard 2.0.

1 Introduction

Some deficiencies in the programming language C make it seem rather inappropriate for the programming of numerical algorithms. C does not provide the basic numerical data structures such as vectors and matrices and does not perform index range checking for arrays. This results in unpredictable errors which are difficult to locate within numerical algorithms. Additionally, pointer handling and the lack of overloadable operators in C reduce the readability of programs and make program development more difficult. Furthermore, C (even the ANSI C standard) does not specify the accuracy or the rounding direction of the arithmetic operators. The same applies to input and output library functions of C. The ANSI C standard does not prescribe the conversion error of input or output.

The programming language C++, an object-oriented C extension, has become more and more popular over the past few years. It does not provide

*Institute for Applied Mathematics, University of Karlsruhe, P.O. Box 6980, D-76128 Karlsruhe, Germany, e-mail: `c-xsc@math.uni-karlsruhe.de`

better facilities for the given problems, but its new concept of abstract data structures (classes) and the concept of overloaded operators and functions provide the possibility to create a programming tool eliminating the disadvantages of C mentioned above: C–XSC (C for eXtended Scientific Computing). It provides the C and C++ programmer with a tool to write numerical algorithms producing reliable results in a comfortable programming environment without having to give up the intrinsic language with its special qualities. The object-oriented aspects of C++ provide additional powerful language features that reduce the programming effort and enhance the readability and reliability of programs.

With its abstract data structures, predefined operators and functions, C–XSC provides an interface between scientific computing and the programming languages C and C++. Besides, C–XSC supports the programming of algorithms which automatically enclose the solution of a given mathematical problem in verified bounds. Such algorithms deliver a precise mathematical statement about the true solution.

The most important features of C–XSC are:

- Real, complex, interval, and complex interval arithmetic with mathematically defined properties
- Dynamic vectors and matrices
- Subarrays of vectors and matrices
- Dotprecision data types
- Predefined arithmetic operators with highest accuracy
- Standard functions of high accuracy
- Dynamic multiple-precision arithmetic and standard functions
- Rounding control for I/O data
- Error handling
- Library of problem-solving routines

2 Standard Data Types, Predefined Operators, and Functions

C–XSC provides the simple numerical data types

real, *interval*, *complex*, and *cinterval* (complex interval)

with their appropriate arithmetic and relational operators and mathematical standard functions. All predefined arithmetic operators deliver results with an accuracy of at least 1 ulp (unit in the last place). Thus, they are of maximum accuracy in the sense of scientific computing. The rounding of the arithmetic operators may be controlled using the data types *interval* and *cinterval*. Type casting functions are available for all mathematically useful combinations. Literal constants may be converted with maximum accuracy. All mathematical standard functions for the simple numerical data types may be called by their generic names and deliver results with guaranteed high accuracy for arbitrary permissible arguments. The standard functions for the data types *interval* and *cinterval* provide range inclusions which are sharp bounds.

For the scalar data types presented above, vector and matrix types are available:

rvector, *ivector*, *cvector*, and *civector*,
rmatrix, *imatrix*, *cmatrix*, and *cimatrix*.

The user can allocate or deallocate storage space for a dynamic array (vector or matrix) *at run time*. Thus, without recompilation, the same program may use arrays of size restricted only by the storage of the computer. Furthermore, the memory is used efficiently, since the arrays are stored only in their required sizes. When accessing components of the array types, the index range is checked at run time to provide increased security during programming by avoiding invalid memory accesses.

Function	Generic Name
Sine	sin
Cosine	cos
Tangent	tan
Cotangent	cot
Arc Sine	asin
Arc Cosine	acos
Arc Tangent	atan
Arc Cotangent	acot
Hyperbolic Sine	sinh
Hyperbolic Cosine	cosh
Hyperbolic Tangent	tanh
Hyperbolic Cotangent	coth
Inverse Hyperbolic Sine	asinh
Inverse Hyperbolic Cosine	acosh
Inverse Hyperbolic Tangent	atanh
Inverse Hyperbolic Cotangent	acoth
Square	sqr
Integer Power Function	power
Exponential Function	exp
Power Function	pow
Square Root	sqrt
nth Root	sqrt
Natural Logarithm	ln
Absolute Value	abs

Table 1: Mathematical Standard Functions

right operand / left operand	integer real complex	interval cinterval	rvector cvector	ivector civector	rmatrix cmatrix	imatrix cimatrix
monadic	$-$	$-$	$-$	$-$	$-$	$-$
integer real complex	$+, -, *, /$ $\vert$	$+, -, *, /,$ $\vert$	$*$	$*$	$*$	$*$
interval cinterval	$+, -, *, /,$ $\vert$	$+, -, *, /,$ $\vert$, &	$*$	$*$	$*$	$*$
rvector cvector	$*, /$	$*, /$	$+, -, *^1,$ $\vert$	$+, -, *^1,$ $\vert$		
ivector civector	$*, /$	$*, /$	$+, -, *^1,$ $\vert$	$+, -, *^1,$ $\vert$, &		
rmatrix cmatrix	$*, /$	$*, /$	$*^1$	$*^1$	$+, -, *^1,$ $\vert$	$+, -, *^1,$ $\vert$
imatrix cimatrix	$*, /$	$*, /$	$*^1$	$*^1$	$+, -, *^1,$ $\vert$	$+, -, *^1,$ $\vert$, &

|: Convex Hull &: Intersection [1]: Dot Product with Maximum Accuracy

Table 2: Predefined Arithmetic Operators

Example: Allocation and resizing of dynamic matrices:

```
...
 int n, m;
 cout << "Enter the dimensions n, m:";
 cin  >> n >> m;

 imatrix B, C, A(n, m);     /* A[1][1] ... A[n][m] */
 Resize(B, m, n);           /* B[1][1] ... B[m][n] */
...
 C = A * B;                 /* C[1][1] ... C[n][n] */
```

Defining a vector or a matrix without explicitly indicating the index bounds results in a vector of length 1 or in a 1×1 matrix. The storage for the object is not allocated until run time. Here, we use the *Resize* statement (see example above) to allocate an object of the desired size. Alternatively, the index bounds may be determined when defining the vector or matrix as we did in the example above with matrix A.

An implicit resizing of a vector or a matrix is also possible during an assignment: If the index bounds of the object on the right-hand side of an assignment do not correspond to those of the left-hand side, the object is changed correspondingly on the left side as shown in the example above with the assignment $C = A * B$.

The storage space of a dynamic array that is local to a subprogram is automatically released before control returns to the calling routine.

The size of a vector or a matrix may be determined at any time by calling the functions *Lb()* and *Ub()* for the lower and upper index bounds, respectively.

3 Subarrays of Vectors and Matrices

C–XSC provides a special notation to manipulate subarrays of vectors and matrices. Subarrays are arbitrary rectangular parts of arrays. All predefined operators may also use subarrays as operands. A subarray of a matrix or vector is accessed using the ( )-operator or the [ ]-operator. The ( )-operator specifies a subarray of an object of the same type as the original object. For example, if A is a real $n \times n$-matrix, then $A(i, i)$ is the left upper $i \times i$ submatrix. Note that parentheses in the declaration of a dynamic vector or matrix do not specify a subarray, but define the index ranges of the object to be allocated. The [ ]-operator generates a subarray of a "lower" type. For example, if A is a $n \times n$ *rmatrix*, then $A[i]$ is the i-th row of A of type *rvector* and $A[i][j]$ is the (i, j)-th element of A of type *real*.

Both types of subarray access may also be combined, for example:
$A[k](i, j)$ is a subvector from index i to index j of the k-th row vector of the matrix A.

The use of subarrays is illustrated in the following example describing the LU-factorization of a $n \times n$-matrix A:

```
for (j=1; j<=n-1; j++) {
  for (k=j+1; k<=n; k++) {
    A[k][j]      = A[k][j] / A[j][j];
    A[k](j+1,n)  = A[k](j+1,n) - A[k][j] * A[j](j+1,n);
  }
}
```

This example demonstrates two important features of C–XSC. First, we save one loop by using the subarray notation. This reduces program complexity. Second, the program fragment above is independent of the type of matrix A (either *rmatrix*, *imatrix*, *cmatrix* or *cimatrix*), since all arithmetic operators are suitably predefined in the mathematical sense.

left operand \ right operand	real	interval	complex	cinterval	rvector	ivector	cvector	civector	rmatrix	imatrix	cmatrix	cimatrix
monadic	!	!	!	!	!	!	!	!	!	!	!	!
real	V_{all}	$\mathsf{V}_{\subset}$	V_{eq}	$\mathsf{V}_{\subset}$								
interval	$\mathsf{V}_{\supset}$	V^1_{all}		V^1_{all}								
complex	V_{eq}		V_{eq}	$\mathsf{V}_{\subset}$								
cinterval	$\mathsf{V}_{\supset}$	V^1_{all}	$\mathsf{V}_{\supset}$	V^1_{all}								
rvector					V_{all}	$\mathsf{V}_{\subset}$	V_{eq}	$\mathsf{V}_{\subset}$				
ivector					$\mathsf{V}_{\supset}$	V^1_{all}		V^1_{all}				
cvector					V_{eq}		V_{eq}	$\mathsf{V}_{\subset}$				
civector					$\mathsf{V}_{\supset}$	V^1_{all}	$\mathsf{V}_{\supset}$	V^1_{all}				
rmatrix									V_{all}	$\mathsf{V}_{\subset}$	V_{eq}	$\mathsf{V}_{\subset}$
imatrix									$\mathsf{V}_{\supset}$	V^1_{all}		V^1_{all}
cmatrix									V_{eq}		V_{eq}	$\mathsf{V}_{\subset}$
cimatrix									$\mathsf{V}_{\supset}$	V^1_{all}	$\mathsf{V}_{\supset}$	V^1_{all}

$\mathsf{V}_{all} = \{==, !=, <=, <, >=, >\}$ $\qquad$ $\mathsf{V}_{eq} = \{==, !=\}$

$\mathsf{V}_{\subset} = \{==, !=, <=, <\}$ $\qquad$ $\mathsf{V}_{\supset} = \{==, !=, >=, >\}$

Table 3: Predefined Relational Operators

4 Evaluation of Expressions with High Accuracy

When evaluating arithmetic expressions, accuracy plays a decisive role in many numerical algorithms. Even if all arithmetic operators and standard functions are of maximum accuracy, expressions composed of several operators and functions do not necessarily deliver results with maximum accuracy (see [7]). Therefore, methods have been developed for evaluating numerical expressions with high and mathematically guaranteed accuracy.

A special kind of such expressions are called *dot product expressions*, which are defined as sums of simple expressions. A simple expression is either a variable, a constant, or a single product of two such objects. The variables may be of scalar, vector, or matrix type. Only the mathematically relevant operations are permitted for addition and multiplication. The result of such an expression is either a scalar, a vector, or a matrix. In numerical analysis, dot product expressions are of decisive importance. For example, methods for defect correction or iterative refinement for linear or nonlinear problems are based on dot product expressions. An evaluation of these expressions with maximum accuracy avoids cancellation. To obtain an evaluation with 1 ulp accuracy, C–XSC provides the dotprecision data types

dotprecision, *cdotprecision*, *idotprecision*, and *cidotprecision*.

Intermediate results of a dot product expression can be computed and stored in a dotprecision variable without any rounding error. The following example computes an optimal inclusion of the defect $b - Ax$ of a linear system $Ax = b$:

```
ivector defect(rvector b, rmatrix A, rvector x)
{
  idotprecision accu;
  ivector incl(Lb(x),Ub(x));
  for (int i=Lb(x); i<=Ub(x); i++) {
     accu = b[i];
     accumulate(accu, -A[i], x);
     incl[i] = rnd(accu);
  }
  return incl;
}
```

In the example above, the function *accumulate()* computes the sum:

$$\sum_{j=1}^{n} -A_{ij} \cdot x_j$$

and adds the result to the accumulator *accu* without rounding error. The *idotprecision* variable *accu* is initially assigned $b[i]$. Finally, the accumulator is rounded to the optimal standard interval *incl*$[i]$. Thus, the bounds of *incl*$[i]$ will either be the same or two adjacent floating-point numbers.

For all dotprecision data types, a reduced set of predefined operators is available to compute results without any error. The overloaded dot product routine *accumulate()* and the rounding function *rnd()* are available for all reasonable type combinations.

left operand \ right operand	real complex	interval cinterval	dotprecision cdotprecision	idotprecision cidotprecision
monadic	$-$	$-$	$-$	$-$
real complex	$+, -,$ $*, /, \vert$	$+, -,$ $*, /, \vert$	$+, -,$ $\vert$	$+, -,$ $\vert$
interval cinterval	$+, -,$ $*, /, \vert$	$+, -,$ $*, /, \vert,$ &	$+, -,$ $\vert$	$+, -,$ $\vert,$ &
dotprecision cdotprecision	$+, -,$ $\vert$	$+, -,$ $\vert$	$+, -,$ $\vert$	$+, -,$ $\vert$
idotprecision cidotprecision	$+, -,$ $\vert$	$+, -,$ $\vert,$ &	$+, -,$ $\vert$	$+, -,$ $\vert,$ &

| : Convex hull   & : Intersection

Table 4: Predefined Dotprecision Operators

5 Dynamic Multiple-Precision Arithmetic

Besides the classes *real* and *interval*, the dynamic classes long real (*l_real*) and long interval (*l_interval*) as well as the corresponding dynamic vectors and matrices are implemented including all arithmetic and relational operators and multiple-precision standard functions. The computing precision may be controlled by the user at run time. By replacing the *real* and *interval* declarations by *l_real* and *l_interval*, the user's application program turns into a multiple-precision program. This concept provides the user with a powerful and easy-to-use tool for error analysis. Furthermore, it is possible to write programs delivering numerical results with a user-specified accuracy by internally modifying the computing precision at run time in response to the error bounds for intermediate results within the algorithm.

All predefined operators for *real* and *interval* types are also available for *l_real* and *l_interval*. Additionally, all possible operator combinations between single and multiple-precision types are included. The following example shows a single-precision program and its multiple-precision version:

```
#include <interval.hpp>

main()
{
  interval a, b;            /* Standard intervals     */
  a = 1.0;                  /* a   = [1.0,1.0]        */
  b = 3.0;                  /* b   = [3.0,3.0]        */
  cout << "a/b = " << a/b;  /* a/b = [0.333333333333,
                                      0.333333333334] */
}
```

```
#include <l_interval.hpp>

main()
{
  l_interval a, b;    /* Multiple-precision intervals */
  a = 1.0;
  b = 3.0;
  stagprec = 2;       /* global integer variable      */
  cout << "a/b = " << a/b;
/* a/b = [0.333333333333333333333333333,
          0.333333333333333333333333334]              */
}
```

At run time, the predefined global integer variable *stagprec* (staggered precision) controls the computing precision of the multiprecision arithmetic in steps of a single *real* (64 bit words). The precision of a multiple-precision number is defined as the number of *real*s used to store the long number's value. An object of type *l_real* or *l_interval* may change its precision at run time. Components of a vector or a matrix may be of different precision. All multiple-precision arithmetic routines and standard functions compute a numerical result possessing a precision specified by the actual value of *stagprec*. Allocation, resize, and subarray access of multiple-precision vectors and matrices are similar to the corresponding single-precision data types.

6 Input and Output in C–XSC

Using the stream concept and the overloadable operators << and >> of C++, C–XSC provides rounding and formatting control during I/O (input/output) for all new data types, even for the dotprecision and multiple-precision data types. I/O parameters such as rounding direction, field width, etc. also use the overloaded I/O operators to manipulate I/O data. If a new set of I/O parameters is to be used, the old parameter settings can be saved on an internal stack. New parameter values can then be defined. After the use of the new settings, the old ones can be restored from stack. The following example illustrates the use of the C–XSC input and output facilities:

```
main()
{
  real a, b;
  interval c;

  cout << "Please enter real a, b: ";
  cout << RndDown;
  cin  >> a;             /* read a rounded downwards      */
  cout << RndUp;
  cin  >> b;             /* read b rounded upwards        */
  "[0.11, 0.22]" >> c;   /* string to interval conversion */
  cout << SaveOpt;       /* push I/O parameters to stack  */
  cout << SetPrecision(20,16); /* set field width, digits */
  cout << Hex;           /* hexadecimal output format     */
  cout << c << RestoreOpt; /* pop parameters from stack   */
}
```

7 Error Handling in C–XSC

C++ provides intrinsic safety features such as type checking, type-safe linking of programs, and function prototypes. C–XSC supports additional features for safe programming such as index range checking for vectors and matrices and checking for numerical errors such as overflow, underflow, loss of accuracy, illegal arguments, etc. C–XSC provides the user with various modification possibilities to manipulate the reactions of the error handler.

8 Library of Problem Solving Routines

The C–XSC problem solving library is a collection of routines for standard problems of numerical analysis producing guaranteed results of high accuracy. The following areas are covered:

- Evaluation and zeros of polynomials
- Matrix inversion, linear systems
- Eigenvalues, eigenvectors
- Fast Fourier Transform

- Zeros of a nonlinear equation
- Systems of nonlinear equations
- Initial value problems in ordinary differential equations

9 Conclusions

In contrast to C and C++, all predefined arithmetic operators, especially the vector and matrix operations, deliver a result of at least 1 ulp accuracy in C–XSC. There is no need to learn the new features of C++ in order to be able to use the C–XSC programming environment for numerical applications. In most cases, knowledge of the language C is sufficient to work with C–XSC.

The advanced user can extend C–XSC using object-oriented programming features of C++. Programs written in C–XSC can be combined with any other C++ software. If some elementary programming rules are respected, C–XSC programs always deliver compatible numerical results even on different computers with different C++ compilers. That is, C–XSC is a tool to achieve full numerical result compatibility in the sense of interval mathematics.

References

[1] Adams, E.; Kulisch, U.: *Scientific Computing with Automatic Result Verification.* Academic Press, New York, 1993.

[2] Alefeld, G.; Herzberger, J.: *Introduction to Interval Analysis.* Academic Press, New York, 1983.

[3] Ellis, M. A.; Stroustrup, B.: *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, Mass., 1990.

[4] Hammer, R.; Hocks, M.; Kulisch, U.; Ratz, D.: *C++ Toolbox for Verified Computing*. Basic Numerical Problems. Springer-Verlag, Berlin, 1995.

[5] Kernighan, B. W.; Ritchie, D. M.: *The C Programming Language.* Second Edition, ANSI C, Prentice Hall, 1989.

[6] Klatte, R.; Kulisch, U.; Lawo, C.; Rauch, M.; Wiethoff, A.: *C–XSC – A C++ Class Library for Scientific Computing.* Springer-Verlag, Berlin, 1993.

[7] Kulisch, U.: *Computer Arithmetic in Theory and Practice.* Academic Press, New York, 1983.

[8] Stroustrup, B.: *The C++ Programming Language.* Second Edition, Addison-Wesley, Reading, Mass., 1991.

A C–XSC Sample Programs

The examples demonstrate various concepts of C–XSC

- Interval Newton Method
  - Data type *interval*
  - Interval operators
  - Interval standard functions
- Runge-Kutta Method
  - Dynamic arrays
  - Array operators
  - Overloading of operators
  - Mathematical notation
- Trace of a Product Matrix
  - Dynamic arrays
  - Subarrays
  - Dotproduct expressions

Well-known algorithms were intentionally chosen so that a brief explanation of the mathematical background is sufficient. Since the programs are largely self-explanantory, comments are kept to a minimum.

A.1 Interval Newton Method

Compute an enclosure of a zero of a real function $f(x)$. It is assumed that the derivative $f'(x)$ is continuous in $[a, b]$, and that

$$0 \notin \{f'(x), x \in [a, b]\}, \text{ and } \; f(a) \cdot f(b) < 0.$$

If X_n is an inclusion of the zero, then an improved inclusion X_{n+1} may be computed by

$$X_{n+1} := \left(m(X_n) - \frac{f(m(X_n))}{f'(X_n)} \right) \cap X_n,$$

where $m(X)$ is a point within the interval X, usually the midpoint. The mathematical theory of the Interval Newton method appears in [1].

In this example, we apply Newton's method to the function

$$f(x) = \sqrt{x} + (x+1) \cdot \cos(x).$$

Generic function names are used for interval square root, interval sine, and interval cosine so that f may be written in a mathematical notation.

```
#include <interval.hpp>         // Interval arithmetic package
#include <imath.hpp>            // Interval standard functions

interval f(real& x)
{                               // Function f
  interval y;
  y = x;                        // Use interval arithmetic
  return sqrt(y) + (y+1.0) * cos(y);
}

interval deriv(interval& x)
{                               // Derivative function f'
  return (1.0 / (2.0 * sqrt(x)) + cos(x) - (x+1.0) * sin(x));
}

int criter(interval& x)         // f(a)*f(b) < 0   and
{                               // not 0 in f'([x])?
  return (Sup( f(Inf(x))*f(Sup(x)) ) < 0.0  &&
          !(0.0 <= deriv(x))); // '<=': `element of'
}

main()
{
  interval y, y_old;

  cout << "Please enter starting interval: ";  cin >> y;
  cout << SetPrecision(20,12);
  if (criter(y))
    do {
      y_old = y;
      cout << "y = " << y << endl;
      y = (mid(y)-f(mid(y))/deriv(y)) & y; // Iter. formula
    } while (y != y_old);                 // &: intersection
```

```
    else
      cout << "Criterion not satisfied!" << endl;
  }
```

Run Time Output

```
Please enter starting interval: [2.0,3.0]
y = [      2.000000000000,      3.000000000000]
y = [      2.000000000000,      2.218137182954]
y = [      2.051401462380,      2.064726329908]
y = [      2.059037791936,      2.059053011234]
y = [      2.059045253413,      2.059045253417]
y = [      2.059045253415,      2.059045253416]
```

A.2 Runge-Kutta Method

The initial value problem for a system of differential equations is to be solved by the well known Runge-Kutta method. The C–XSC program is very similar to the mathematical notation. Dynamic vectors are used to make the program independent of the size of the system of differential equations to be solved.

Consider the first-order system of differential equations

$$Y' = F(x, Y), \quad Y(x_0) = Y_0.$$

If the solution Y is known at a point x, the approximation $Y(x + h)$ may be determined by the Runge-Kutta method:

$$\begin{aligned}
K_1 &= h \cdot F(x, Y) \\
K_2 &= h \cdot F(x + h/2, Y + K_1/2) \\
K_3 &= h \cdot F(x + h/2, Y + K_2/2) \\
K_4 &= h \cdot F(x + h, Y + K_3) \\
Y(x + h) &= Y + (K_1 + 2 \cdot K_2 + 2 \cdot K_3 + K_4)/6.
\end{aligned}$$

For example, we solve the system

$$\begin{array}{rclrcl}
Y_1' &=& Y_2 Y_3, & Y_1(0) &=& 0 \\
Y_2' &=& -Y_1 Y_3, & Y_2(0) &=& 1 \\
Y_3' &=& -0.522 Y_1 Y_2, & Y_3(0) &=& 1.
\end{array}$$

The program computes an approximation of the solution at the points

$$x_i = x_0 + i \cdot h, \ i = 1, 2, 3$$

starting at given x_0 (here: $x_0 = 0, h = 0.1$).

```
#include <rvector.hpp>

rvector F(real x, rvector Y)
{ // Function definition
  rvector Z(3);                // Constructor call
  x    =  0.0;                 // F is independent of x
  Z[1] =  Y[2]  * Y[3];
  Z[2] = -Y[1]  * Y[3];
  Z[3] = -0.522 * Y[1] * Y[2];
  return Z;
}

void Init(real& x, real& h, rvector& Y)
{                              // Initialization
  Resize(Y,3);                 // Resize dynamic array
  x    = 0.0;   h    = 0.1;
  Y[1] = 0.0;   Y[2] = 1.0;  Y[3] = 1.0;
}

main()
{
  real x, h;
  rvector Y(3), K1(3), K2(3), K3(3), K4(3);

  Init(x, h, Y);
  for (int i=1; i<=3; i++) {  // 3 Runge-Kutta steps
    K1 = h * F(x, Y);          // with array result
    K2 = h * F(x + h / 2, Y + K1 / 2);
    K3 = h * F(x + h / 2, Y + K2 / 2);
    K4 = h * F(x + h, Y + K3);
    Y  = Y + (K1 + 2 * K2 + 2 * K3 + K4) / 6;
    x += h;
    cout << SetPrecision(18,16) << Dec;  // I/O modification
    cout << "Step:  " << i << ",  "
         << "x =    " << x << endl;
    cout << "Y =    " << endl << Y << endl;
  }
}
```

Run Time Output

```
Step:  1,  x =    0.1000000000000000
Y =
0.0997468762564554
0.9950128338579525
0.9973998103188983

Step:  2,  x =    0.2000000000000000
Y =
0.1979929459605080
0.9802034181615258
0.9897155789871044

Step:  3,  x =    0.3000000000000000
Y =
0.2933197040249942
0.9560143619620973
0.9772864889519136
```

A.3 Trace of a Product Matrix

Dot product expressions are sums of *real*, *interval*, *complex*, or *cinterval* constants, variables, vectors, matrices, or simple products of them. Dotprecision variables are used to store intermediate results of a dot product expression without any rounding error. The contents of a dotprecision variable may be rounded into a floating-point number using the rounding direction specified by the user.

The following C–XSC program demonstrates the use of this concept. The trace of a complex matrix $A \cdot B$ is evaluated without calculating the actual product. The result is of maximum accuracy. That is, it is the best possible approximation of the exact result. The trace of the product matrix is

$$\mathrm{Trace}(A \cdot B) := \sum_{i=1}^{n} \sum_{j=1}^{n} A_{ij} \cdot B_{ji}.$$

```
#include <cmatrix.hpp>     // Use the complex matrix package
main()
{
  int n;
  cout << "Please enter the matrix dimension n: ";
  cin  >> n;
  cmatrix A(n,n), B(n,n);  // Dynamic allocation of A, B
  complex result;  cdotprecision accu;
  cout << "Please enter the matrix A:" << endl;  cin  >> A;
  cout << "Please enter the matrix B:" << endl;  cin  >> B;
  accu = 0.0;              // Clear accumulator
  for (int i=1; i<=n; i++)
    accumulate(accu, A[i], B[Col(i)]);
// A[i] and B[Col(i)] are subarrays of type cvector.
// Rounding the exact result to the nearest complex number:
  result = rnd(accu);
  cout << SetPrecision(15,8) << RndNext << Dec;
  cout << "The trace of the product matrix is: " << endl
       << result << endl;
}
```

Run Time Output

```
Please enter the matrix dimension n: 3
Please enter the matrix A:
(2.2,3.3) (-4.1,0.0) (7.2,-1.1)
(1.2,3.4) (4.1,-5.0) (2.2,1.3)
(2.3,4.5) (-5.2,0.0) (5.2,1.4)
Please enter the matrix B:
(5.2,4.3) (-9.1,0.2) (4.2,-1.1)
(1.3,6.8) (4.1,5.9)  (-2.2,1.3)
(4.5,6.5) (5.7,0.3)  (4.2,-1.4)
The trace of the product matrix is:
(    128.18000000,     29.86000000)
```

R. Klatte, U. Kulisch, C. Lawo, M. Rauch, A. Wiethoff

C – XSC

A C++ Class Library for Extended Scientific Computing

Springer-Verlag, Berlin, 1993,
269 pp., ISBN 3-540-56328-8

C – XSC is a tool for the development of numerical algorithms delivering highly accurate and automatically verified results. It provides a large number of predefined numerical data types and operators. These types are implemented as C++ classes. Thus, C – XSC allows high-level programming of numerical applications in C and C++.

The most important features of C – XSC are:

Real, complex, interval, and complex interval arithmetic, dynamic vectors and matrices, subarrays of vectors and matrices, dot-precision data types, predefined arithmetic operators with maximum accuracy, standard functions of high accuracy, multiple-precision arithmetic and standard functions, rounding control for I/O data, error handling, and a library of problem solving routines with automatic result verification.

Therefore, C – XSC makes the computer more powerful concerning the arithmetic. C – XSC is immediately usable by C and C++ programmers, easy to learn, user-extendable, and may also be combined with other tools.

The book can be used as a textbook and as a reference manual. It consists of an introduction to advanced computer arithmetic, a chapter describing the programming languages C and C++, the major chapter "C – XSC Reference", sample programs, and indices.

R. Hammer, M. Hocks, U. Kulisch, D. Ratz

C++ Toolbox for Verified Computing I

Basic Numerical Problems

Springer-Verlag, Berlin, 1995,
377 pp., ISBN 3-540-59110-9

The C++ Toolbox for Verified Computing presents an extensive set of sophisticated tools to solve numerical problems with a verification of the results using the features of the scientific computer language C – XSC. It is the C++ edition of the Numerical Toolbox for Verified Computing which is based on the computer language PASCAL-XSC. The sources of this book are freely available via anonymous ftp.

The overriding concern of this book is reliability – the automatic verification of the result a computer returns for a given problem. The book offers a general discussion on arithmetic and computational reliability, analytical mathematics and verification techniques, algorithms, and (most importantly) actual C++ implementations in the form of working computer routines. In each chapter, examples, exercises, and numerical results demonstrate the application of the routines presented. It is not assumed that the reader has any prior formal knowledge of numerical verification or any familiarity with interval analysis. Some of the subjects that the book covers in detail are not usually found in standard numerical analysis texts.

This book is intended primarily as a reference text, however, it can also be used as a textbook for an advanced course in scientific computation with automatic result verification.