Bergische Universität
GH Wuppertal

C–XSC 2.0

A C++ Class Library for Extended Scientific Computing

W. Hofschuster, W. Krämer, S. Wedner, A. Wiethoff

Preprint 2001/1

Wissenschaftliches Rechnen/
Softwaretechnologie

Impressum

Herausgeber:	Prof. Dr. W. Krämer, Dr. W. Hofschuster Wissenschaftliches Rechnen/Softwaretechnologie Fachbereich 7 (Mathematik) Bergische Universität GH Wuppertal Gaußstr. 20 D-42097 Wuppertal

Internet-Zugriff

Die Berichte sind in elektronischer Form erhältlich über die World Wide Web Seiten

`http://www.math.uni-wuppertal.de/wrswt/literatur.html`

Autoren-Kontaktadresse

Dr. W. Hofschuster
Prof. Dr. W. Krämer
Dr. S. Wedner
Dr. A. Wiethoff
Bergische Universität GH Wuppertal
Gaußstr. 20
D-42097 Wuppertal

e-mail: `xsc@math.uni-wuppertal.de`
WWW: `http://www.math.uni-wuppertal.de/~xsc/`

C – XSC 2.0 is freely available from
`http://www.math.uni-wuppertal.de/~xsc/xsc/download.html`

**Version 2.0 of the C++Toolbox for Verified Computing [3]
is freely available from**
`http://www.math.uni-wuppertal.de/~xsc/xsc/download.html`

Zusammenfassung

C–XSC 2.0: Eine C++ Klassenbibliothek für erweitertes Wissenschaftliches Rechnen: C – XSC ist ein Werkzeug zur Entwicklung numerischer Algorithmen, die hochgenaue und selbstverifizierende Resultate liefern. C – XSC stellt eine große Zahl vordefinierter Datentypen und Operatoren zur Verfügung. Diese Datentypen sind als Klassen in C++ implementiert. Damit ermöglicht C – XSC die komfortable Programmierung numerischer Anwendungen in C++. C – XSC ist für viele Rechnersysteme verfügbar. Die neue Version C – XSC 2.0 entspricht dem C++ Standard [12]. Die Quellen von C – XSC 2.0 sind im Netz frei verfügbar. Dies trifft auch auf eine an C – XSC 2.0 angepasste Version der Problemlöseroutinen [3] zu.

Abstract

C–XSC 2.0: A C++ Class Library for Extended Scientific Computing: 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++. The C – XSC package is available for many computers with a C++ compiler conforming to the C++ standard [12]. The sources of the new version C – XSC 2.0 as well as the problem solving routines [3] are freely available. C – XSC 2.0 now conforms ISO/IEC C++ standard [12].

1 Acknowledgements

The work on C – XSC started in 1990 at the Institute for Applied Mathematics (Prof. Kulisch), University of Karlsruhe. Many colleagues and scientists have directly and indirectly contributed to the realization of C – XSC. The authors would like to thank each of them for his or her cooperation. Special thanks go to U. Allendörfer, C. Baumhof, H. Berlejung, B. Bohl, G. Bohlender, D. Cordes, K. Grüner, R. Hammer, M. Hocks, W. Hofschuster, R. Klatte, W. Krämer, U. Kulisch, C. Lawo, M. Neaga, D. Ratz, M. Rauch, S. Ritterbusch, G. Schumacher, J. Suckfüll, F. Toussaint, W. Walter, S. Wedner, G. Werheit, A. Wiethoff, and J. Wolff von Gudenberg.

C – XSC 2.0 is an outcome of an ongoing collaboration of the Institute for Applied Mathematics (Prof. Kulisch), University of Karlsruhe and the Institute for Scientific Computing/Software Engineering (Prof. Krämer), University of Wuppertal. For the latest news and up to date software contact
`http://www.math.uni-wuppertal.de/~xsc/`

2 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, ANSI C 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 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 language 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
- Library of problem-solving routines (C++ Toolbox for Verified Computing [3])
- Numerical results with mathematical rigor

What is new in C – XSC 2.0?

- All routines are now in the namespace `cxsc`
- Explicit typecast constructors
- Constant values passed by reference are now passed by const reference

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

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

[5] Krämer, W.; Bantle, A.: *Automatic Forward Error Analysis for Floating Point Algorithms.* Reliable Computing, Vol. 7, No. 4, pp. 321-340, 2001.

[6] Krämer, W.; Wolff von Gudenberg, J. (eds.): *Scientific Computing, Validated Numerics, Interval Methods.* Kluwer Academic Publishers, Boston, Dordrecht, London, 2001.

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

[8] Kulisch, U.; Lohner, R.; Facius, A. (eds): *Perspectives on Enclosure Methods.* Springer Verlag, Wien, New York, 2001.

[9] Neher, M.: *Validated Bounds for Taylor Coefficients of Analytic Functions.* Reliable Computing, Vol. 7, No. 4, pp. 307-319, 2001.

[10] Stroustrup, B.: *The C++ Programming Language.* Special Edition, Addison-Wesley, Reading, Mass., 2000.
Deutsche Übersetzung: *Die C++ Programmiersprache*, 4. Auflage, Addison-Wesley, München, 2000.

[11] Wedner, S.: *Verifizierte Bestimmung singulärer Integrale - Quadratur und Kubatur.* Dissertation, Universität Karlsruhe, 2000.

[12] ISO/IEC 14882: *Standard for the C++ Programming Language*, 1998.

11 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. The huge set of predefined operators and functions can be called by their usual symbols and names. Thus, arithmetic expressions and numerical algorithms are expressed in a notation that is very close to the usual mathematical notation. Using C–XSC many programs can be read like a technical report. Programs are much easier to read, to write, and to debug.

C–XSC is particularly suited for the the development of numerical algorithms that deliver highly accurate and automatically verified results, which are essential, for example, in simulation runs where the user has to distinguish between computational artifacts and genuine reactions of the model. C–XSC allows the numerical computations to carry their own accuracy control.

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.

Meanwhile a lot of problem-solving functions with automatic result verification have been developed in C–XSC for several standard problems of numerical analysis. This is still an ongoing process (e. g. numerical quadrature and cubature [11], validated bounds for taylor coefficients [9], automatic forward error analysis [5]). A lot of actual material as well as a lot of references in the field of validated numerics and verified computing may be found in [6] and [8].

- The error handling is done according to the C++ error handling using exception classes
- Modification in the field for subvectors and submatrices
- The library uses templates extensively
- The source code of C-XSC 2.0 is freely available from `http://www.math.uni-wuppertal.de/~xsc/xsc/download.html`
- Older C–XSC programs have to be modified slightly to run with C–XSC 2.0.
- The source code of a new version of the C++ Toolbox for Verified Computing [3] which works with C-XSC 2.0 is freely available from `http://www.math.uni-wuppertal.de/~xsc/xsc/download.html`

3 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 elementary mathematical functions for the data type *interval* provide range inclusions which are sharp bounds. Elementary functions for the data type *cinterval* are also available.

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.

left operand \ right 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	$*,/$	$*,/$	$*^1$	$*^1$	$+,-,*^1, \vert$	$+,-,*^1, \vert, \&$

$\vert$: Convex Hull   &: Intersection ^{1}: Dot Product with Maximum Accuracy

Table 1: Predefined Arithmetic Operators

Function	Generic Name
Sine	sin
Cosine	cos
Tangent	tan
Cotangent	cot
Hyperbolic Sine	sinh
Hyperbolic Cosine	cosh
Hyperbolic Tangent	tanh
Hyperbolic Cotangent	coth
Square	sqr
Integer Power Function	power
Exponential Function	exp
Power Function	pow
Absolute Value	abs

Function	Generic Name
Arc Sine	asin
Arc Cosine	acos
Arc Tangent	atan
Arc Cotangent	acot
Inverse Hyperbolic Sine	asinh
Inverse Hyperbolic Cosine	acosh
Inverse Hyperbolic Tangent	atanh
Inverse Hyperbolic Cotangent	acoth
Square Root	sqrt
nth Root	sqrt(x, n)
Natural Logarithm	ln

Table 2: Mathematical Standard Functions


```
    cout << SetPrecision(23,15) << Scientific;   // Output format

    cout << "Search interval     : ";
    cin  >> SearchInterval;
    cout << "Tolerance (relative): ";
    cin  >> Tolerance;
    cout << endl;

    // Call the function 'AllZeros()' from the C++ Toolbox
    AllZeros(f,SearchInterval,Tolerance,Zero,Unique,NumberOfZeros,Error);

    for ( i = 1; i <= NumberOfZeros; i++) {
      cout << Zero[i] << endl;
      if (Unique[i])
        cout << "encloses a locally unique zero!" << endl;
      else
        cout << "may contain a zero (not verified unique)!" << endl;
    }
    cout << endl << NumberOfZeros << " interval enclosure(s)" << endl;
    if (Error) cout << endl << AllZerosErrMsg(Error) << endl;
    return 0;
  }
```

Run Time Output

```
  Search interval      : [-1,15]
  Tolerance (relative) : 1e-10

  [ 5.885327439818601E-001, 5.885327439818619E-001]
  encloses a locally unique zero!
  [ 9.999999999999998E-001, 1.000000000000001E+000]
  encloses a locally unique zero!
  [ 3.096363932404308E+000, 3.096363932416931E+000]
  encloses a locally unique zero!
  [ 6.285049273371415E+000, 6.285049273396501E+000]
  encloses a locally unique zero!
  [ 9.424697254738511E+000, 9.424697254738533E+000]
  encloses a locally unique zero!
  [ 1.256637410119757E+001, 1.256637410231546E+001]
  encloses a locally unique zero!

  6 interval enclosure(s)
```

Let us again emphasize that these results are verified! The problem-solving routines apply interval arithmetic and mathematical fixed point theorems to guarantee the existence and uniqueness of the zeros. It is also verified that f has no other zeros in the interval $[-1, 15]$.

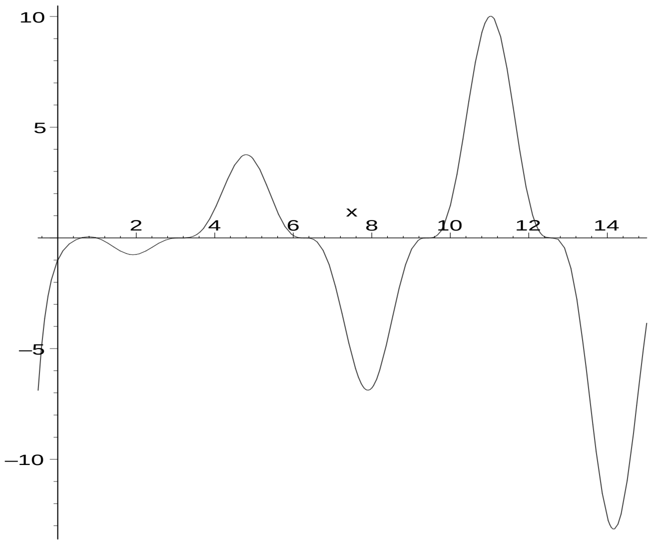

Figure 1: $f(x) = (x-1) * (\exp(-3 * x) - \mathrm{power}(\sin(x), 3))$

```
//
#include "nlfzero.hpp"     // Nonlinear equations module
#include "stacksz.hpp"     // To increase stack size for some
                           // special C++ compiler
using namespace cxsc;
using namespace std;


DerivType f ( const DerivType& x )                    // Sample function
{
   return (x-1)*( exp(-3*x) - power(sin(x),3) );
}

// The class DerivType allows the computation of f, f', and f'' using
// automatic differentiation; see "C++ Toolbox for Verified Computing"

int main()
{
  interval  SearchInterval;
  real      Tolerance;
  ivector   Zero;
  intvector Unique;
  int       NumberOfZeros, i, Error;
```


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.

4 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}	$V_{\subset}$	V_{eq}	$V_{\subset}$								
interval	$V_{\supset}$	V^1_{all}		V^1_{all}								
complex	V_{eq}		V_{eq}	$V_{\subset}$								
cinterval	$V_{\supset}$	V^1_{all}	$V_{\supset}$	V^1_{all}								
rvector					V_{all}	$V_{\subset}$	V_{eq}	$V_{\subset}$				
ivector					$V_{\supset}$	V^1_{all}		V^1_{all}				
cvector					V_{eq}		V_{eq}	$V_{\subset}$				
civector					$V_{\supset}$	V^1_{all}	$V_{\supset}$	V^1_{all}				
rmatrix									V_{all}	$V_{\subset}$	V_{eq}	$V_{\subset}$
imatrix									$V_{\supset}$	V^1_{all}		V^1_{all}
cmatrix									V_{eq}		V_{eq}	$V_{\subset}$
cimatrix									$V_{\supset}$	V^1_{all}	$V_{\supset}$	V^1_{all}

$V_{all} = \{==, !=, <=, <, >=, >\}$ $V_{eq} = \{==, !=\}$

$V_{\subset} = \{==, !=, <=, <\}$ $V_{\supset} = \{==, !=, >=, >\}$

[1] <=: "subset of", <: "proper subset of", >=: "superset of", >: "proper superset of"

Table 3: Predefined Relational Operators

5 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


```
    // The exact result stored in the complex dotprecision variable accu
    // is rounded to the nearest complex floating point number:
    complex result = rnd(accu);
    cout << SetPrecision(12,6) << RndNext << Dec;
    cout << "Trace of tye product matrix:" << result << endl;
    return 0;
}
```

Run Time Output

```
Please enter the matrix dimension n: 3
Please enter the matrix A:
(1,0) (2,0) (3,0)
(4,0) (5,0) (6,0)
(7,0) (8,0) (9,0)
Please enter the matrix B:
(10,0) (11,0) (12,0)
(13,0) (14,0) (15,0)
(16,0) (17,0) (18,0)
Trace of product matrix: (  666.000000,    0.000000)
```

10.5 Get All Zeros of a One-Dimensional Function

The interval Newton method as described in 10.1 can be generalized in several ways. The following program uses such a modification which allows the computation of all zeros of a given one-dimensional function f in a specified starting interval. Because automatic differentiation is used to compute the derivative f' automatically, f has to be defined for the type DerivType. Using extended interval divisions in the Newton steps allows the treatment of functions with horizontal tangents in the search interval (a typical situation for functions with several zeros).

Notice: if no error is indicated by the routine `AllZeros` the computed enclosures for the zeros are verified in a rigorous mathematical sense.

We use modules of the C++ Toolbox for verified computing [3]. So the program below can not be run without installing this toolbox. The sources as well as an installation guide can be found on the web:

`http://www.math.uni-wuppertal.de/~xsc/xsc/download.html`

Figure 1 shows the graph of the function

$$f(x) = (x-1) * (\exp(-3 * x) - \text{power}(\sin(x), 3))$$

for which the enclosures of all zeros in the interval $[-1, 15]$ are to be computed. Notice the mathematical notation of the sample function f in the program code. To consider any other function it suffices to modify the definition of f.

```
// Compute all zeros of the function
//
//              (x-1)*(exp(-3*x) - power(sin(x), 3))
```

```
    0.980203
    0.989716

Step: 3,  x =     0.300000
Y =
    0.293320
    0.956014
    0.977286
```

10.4 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

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

i. e. the sum of the diagonal entries of the product matrix.

```
// Trace of a (complex) matrix product
// Let C denote the matrix product A*B.
// Then the diagonal entries of C are added to get the trace of C.

#include <iostream>
#include "cmatrix.hpp"       // Use the complex matrix package
using namespace std;
using namespace cxsc;

int main()
{
   int n;
   cout << "Please enter the matrix dimension n: ";  cin  >> n;
   cmatrix A(n,n), B(n,n);   // Dynamic allocation of A, B
   cdotprecision accu;       // Allows exact computation of dotproducts
   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 rvector.
```


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.

6 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

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	$+,-,$ $\vert$	$+,-,$ $\vert,$ &	$+,-,$ $\vert$	$+,-,$ $\vert,$ &

$\vert$: Convex hull & : Intersection

Table 4: Predefined Dotprecision Operators

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"  // predefined interval arithmetic
#include <iostream>
using namespace cxsc;
using namespace std;

int 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 << endl;
  return 0;
}
```



```
rvector F(const real x, const rvector& Y) // Function definition
{
   rvector Z(3);                   // Constructor call

   Z[1] =  Y[2]  * Y[3];           // F is independent of x
   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
   x    = 0.0;   h    = 0.1;
   Y[1] = 0.0;   Y[2] = 1.0;  Y[3] = 1.0;
}

int main(void)
{
   real x, h;                      // Declarations and dynamic
   rvector Y(3), K1, K2, K3, K4;   // memory allocation
                                   // Automatic resize of Ki below
   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(10,6) << Dec;  // I/O modification
      cout << "Step: " << i << ",  "
           << "x =   " << x << endl;
      cout << "Y =   " << endl << Y << endl;
   }
   return 0;
}
```

Run Time Output

```
Step: 1,  x =     0.100000
Y =
  0.099747
  0.995013
  0.997400

Step: 2,  x =     0.200000
Y =
  0.197993
```

```
Diameter of actual enclosure: 2.625899E-012
Diameter of actual enclosure: 4.708711E-027
Diameter of actual enclosure: 5.473822E-048
Final enclosure of the zero:
   [ 2.059045253415143788680636155343254522623083897,
     2.059045253415143788680636155343254522623083898 ]
```

As can be seen the method converges quadratically.

10.3 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: $x_0 = 0, h = 0.1$).

```
// Runge-Kutta Method

#include <iostream>
#include "rvector.hpp"              // Include dynamic arrays (real vectors)
using namespace std;
using namespace cxsc;
```


Run Time Output

```
a/b = [  0.333333,  0.333334]
```

The corresponding multi-precision version using the staggered interval arithmetic is very similar:

```
#include "l_interval.hpp"  // interval staggered arithmetic
#include <iostream>
using namespace cxsc;
using namespace std;

int main()
{
  l_interval a, b;         // Multiple-precision intervals
  a = 1.0;
  b = 3.0;
  stagprec = 2;            // global integer variable
  cout << SetDotPrecision(16*stagprec, 16*stagprec-3) << RndNext;
  // I/O for variables of type l_interval is done using
  // the long accumulator (i.e. a dotprecision variable)

  cout << "a/b = " << a/b << endl;
  return(0);
}
```

Run Time Output

```
a/b = [ 0.333333333333333333333333333333, 0.333333333333333333333333333334]
```

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.

7 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:

```
#include <iostream>
#include "interval.hpp"
using namespace cxsc;
using namespace std;

int main()
{
  real a, b;

  cout << "Please enter real a: ";
  cout << RndDown;             // set rounding mode
  cin  >> a;                   // read a rounded downwards
  cout << SetPrecision(7,4);   // set field width and number of digits
                               // for output
  cout << a << endl;           // print a rounded downwards to 4 digits
  cout << RndUp;
  cout << a << endl;           // print a rounded upwards to 4 digits

  "0.3" >> b;                  // convert the string "0.3" to a floating
                               // point value b using rounding down

  cout << SetPrecision(18,15); // from now on print 15 digits

  cout << b << endl;           // print b rounded upwards to 15 digits
  cout << RndDown;
  cout << b << endl;           // print b rounded downwards to 15 digits
  interval x;
  "[1.5, 2]" >> x;             // string to interval conversion
  cout << x << endl;           // print interval x using default setting

  cout << SaveOpt;             // push I/O parameters to internal stack
  cout << SetPrecision(10,7);  // modifies output field width and
                               // number of digits to be print
  cout << x << endl;           // print x in the modified format
  cout << RestoreOpt;          // pop parameters from internal stack
  cout << x << endl;           // again, print x using the former format
  return 0;
}
```

Run Time Output

```
Please enter real a: 0.3
 0.2999
 0.3000
```



```
using namespace cxsc;

l_interval f(const l_real& x)     // Function f
{
   l_interval y(x);               // y is a thin interval initialized by x
   return sqrt(y) + (y+1)*cos(y); // Use multi-precision interval arithmetic
}

l_interval deriv(const l_interval& x) // Derivative function f'
{
   return 1/(2*sqrt(x)) + cos(x) - (x+1)*sin(x);
}

bool criter(const l_interval& x)  // Computing: 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));
}                                 // '<=' means 'element of'

int main(void)
{
   l_interval x, xOld;
   stagprec= 3; // Set precision of the staggered correction arithmetic
   x= l_interval(2,3);
   cout << "Starting interval is [2,3]" << endl;
   cout << SetDotPrecision(16*stagprec, 16*stagprec-3) << RndNext;
   // I/O for variables of type l_interval is done using
   // the long accumulator (i.e. a dotprecision variable)
   if (criter(x))
   {  // There is exactly one zero of f in the interval x
      do {
         xOld = x;
         cout << "Diameter of actual enclosure: " << real(diam(x)) << endl;
         x = (mid(x)-f(mid(x))/deriv(x)) & x; // Iteration formula
      } while (x != xOld);                   // &  means  intersection
      cout << "Final enclosure of the zero: " << x << endl;
   }
   else
      cout << "Criterion not satisfied!" << endl;
   return 0;
}
```

Run Time Output

```
Starting interval is [2,3])
Diameter of actual enclosure:   1.000000
Diameter of actual enclosure:   0.218137
Diameter of actual enclosure:   0.013325
Diameter of actual enclosure: 1.521930E-005
```

```
   interval x, xOld;
   cout << SetPrecision(20,15); // Number of mantissa digits in I/O
   x= interval(2,3);
   cout << "Starting interval is [2,3]" << endl;
   if (criter(x))
   {  // There is exactly one zero of f in the interval x
      do {
         xOld = x;
         cout << "Actual enclosure is " << x << endl;
         x = (mid(x)-f(mid(x))/deriv(x)) & x; // Iteration formula
      } while (x != xOld);
      cout << "Final enclosure of the zero: " << x << endl;
   }
   else
      cout << "Criterion not satisfied!" << endl;
   return 0;
}
```

Run Time Output

```
Starting interval is [2,3]
Actual enclosure is [   2.000000000000000,   3.000000000000000]
Actual enclosure is [   2.000000000000000,   2.218137182953809]
Actual enclosure is [   2.051401462380920,   2.064726329907714]
Actual enclosure is [   2.059037791936965,   2.059053011233253]
Actual enclosure is [   2.059045253413831,   2.059045253416460]
Actual enclosure is [   2.059045253415142,   2.059045253415145]
Final enclosure of the zero: [ 2.059045253415142,
                               2.059045253415145]
```

10.2 Interval Newton Method Using Staggered Arithmetic

Again, we apply Newton's method to the function

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

But now we search for an enclosure of the zero in higher precision using multi-precision staggered intervals. The code is very similar to the code given in section 10.1 using ordinary interval arithmetic.

```
// Interval Newton method using a multi-precision interval data type
// Verified computation of a zero of the function f() to high accuracy

#include <iostream>
#include "l_interval.hpp"         // Include multi-precision intervals
#include "l_imath.hpp"            // Include multi-precision math functions
using namespace std;
```



```
 0.300000000000001
 0.300000000000000
[ 1.500000000000000, 2.000000000000000]
[ 1.5000000, 2.0000000]
[ 1.500000000000000, 2.000000000000000]
```

8 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. C – XSC 2.0 supports C++ error handling using exception classes.

9 Library of Problem Solving Routines

The C – XSC problem solving library (C++ Toolbox for verified computing [3], also freely available) is a collection of routines for standard problems of numerical analysis producing guaranteed results of high accuracy. The following areas are covered:

- One-dimensional problems
  - Evaluation of polynomials
  - Automatic differentiation
  - Nonlinear equations in one variable
  - Global optimization
  - Accurate evaluation of arithmetic expressions
  - Zeros of complex polynomials
- Multi-dimensional problems
  - Linear systems of equations
  - Linear optimization
  - Automatic differentiation for gradients, Jacobians, and Hessian
  - Nonlinear systems of Equations
  - Global optimization
  - Initial value problems in ordinary differential equations[1]

Slightly modified sources of the programs published in the book *C++ Toolbox for verified computing* [3] are available for the use in connection with C – XSC 2.0. You can freely download these files from
`http://www.math.uni-wuppertal.de/˜xsc/xsc/download.html`

[1] Available from R. Lohner (see `http://www.uni-karlsruhe.de/~Rudolf.Lohner/`)

10 C – XSC 2.0 Sample Programs

The following C – XSC sample programs demonstrate various concepts of C – XSC. The sources are all available in the examples directory of your C – XSC 2.0 installation.

- Interval Newton Method
  - Data type *interval*
  - Interval operators
  - Interval standard functions
- Interval Newton Method using Staggered Intervals
  - Data type *l_interval* for multi-precision computations
  - Overloaded operators and function for staggered intervals
  - Illustration of output facilities
- Runge-Kutta Method
  - Dynamic arrays
  - Array operators
  - Overloading of operators
  - Mathematical notation
- Trace of a Product Matrix
  - Dynamic arrays
  - Subarrays
  - Dotproduct expressions
- Get All Zeros of a One-Dimensional Function
  - Calling a problem-solving routine from the C++ Toolbox for verified computing [3]
  - Reliability of computed results: Verified(!) enclosures of all(!) zeros are computed
  - Simple usage: Only the function expression has to be supplied by the user; derivatives are computed automatically using automatic differentiation

Well-known algorithms were intentionally chosen so that a brief explanation of the mathematical background is sufficient. We hope that the programs are largely self-explanatory. For a very readable introduction to interval mathematics and the construction of algorithms with numerical result verification we refer to [3].

10.1 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 [3].

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. `mid(x)` computes any floating point number out of `x`.

```
// Interval Newton method using ordinary interval arithmetic
// Verified computation of a zero of the function f()

#include <iostream>
#include "interval.hpp"          // Include interval arithmetic package
#include "imath.hpp"             // Include interval standard functions
using namespace std;
using namespace cxsc;

interval f(const real x)
{                                // Function f
  interval y(x);                 // y is a thin interval initialized by x
  return sqrt(y) + (y+1)*cos(y); // Interval arithmetic is used
}

interval deriv(const interval& x)
{                                // Derivative function f'
   return 1/(2*sqrt(x)) + cos(x) - (x+1)*sin(x);
}

bool criter(const interval& x)    // Computing: 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));
}                                 // '<=' means 'element of'

int main(void)
{
```