Mecánica Computacional Vol. XXII
M. B. Rosales, V. H. Cortínez y D. V. Bambill (Editores)
Bahía Blanca, Argentina, Noviembre 2003.

ACCURATE EXPRESSIONS IN C-XSC

Mariana L. Kolberg[*], Gerd Bohlender[†] and Dalcidio M. Claudio[*]

[*]Programa de Pós-graduação em Ciência da Computação
Faculdade de Informática, Pontifícia Universidade Católica do Rio Grande do Sul
Av Ipiranga, 6681 Prédio 16, sala 106 – 90619-900 – Porto Alegre – RS – Brasil
e-mail: mkolberg@inf.pucrs.br

[†] Institut für Angewandte Mathematik, Universität Karlsruhe
Kaiserstraße 12 – 76131 Karlsruhe - Deutschland
e-mail: gerd.bohlender@math.uka.de

Key words: Accurate expressions, C-XSC, Scalar products.

Abstract. *C-XSC is a programming library for Extended Scientific Compu-ing which support many data types with high accuracy. However it does not provide the accurate expressions which are already available in the older language PASCAL-XSC. These expressions have to be replaced by long loops which are more difficult to understand. Therefore we studied possibilities how this accurate expressions could be implemented in C-XSC. In this paper we present possibilities like operator overloading with special result types avoiding rounding errors, and a small pre-processor which converts accurate expressions to functions calls. We present results about execution times in comparison with C-XSC. Finally we present some perspectives for future work.*

1 INTRODUCTION

This section describes some concepts on Computational Arithmetic (sub-section 1.1), PASCAL-XSC (subsection 1.2) and C-XSC (subsection 1.3). A small discussion about the differences in the evaluation of expressions with high accuracy between PASCAL-XSC and C-XSC is done in the end of the section (subsection 1.4).

1.1 Computational Arithmetic

PASCAL-XSC and C-XSC are programming tools intended for the development of numerical algorithms with interval arithmetic, providing highly accurate and automatically verified results. These languages provide many predefined numerical data types and operators.

PASCAL-XSC and C-XSC provide the following data types: real, complex, interval, cinterval with their appropriate arithmetic, relational operators, and mathematical standard functions. All predefined operators deliver results with high accuracy of at most l ulp (unit of last place). Thus they are of maximum accuracy in the sense of scientific computing. 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 the scalar data types presented above, vector and matrix types are available in the following data types: rvector, rmatrix, cvector, cmatrix, ivector, imatrix, civector and cimatrix. Arithmetic operators are provided only in mathematically meaningful cases. Thus, it is not permitted to add a vector and a matrix.

In contrast to PASCAL and C/C++, PASCAL-XSC and C-XSC satisfies all demands for a precisely defined computer arithmetic:

- Access to direct roundings up and down;
- Realization of the optimal scalar product for vectors of all relevant lenghts;
- Interval types with corresponding operators;
- Dynamic and structured numerical data types;
- Predefined standard functions with very high accuracy for numerical base types;
- Rounding control for I/O data;

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[5]. Therefore, PASCAL-XSC and C-XSC have methods for evaluating numerical expressions with high and mathematically guaranteed accuracy. The accurate expressions that are handled by PASCAL-XSC and C-XSC have the form of scalar product expressions, which are defined as sums of simple expressions like variables, constants or single products of two such objects. This kind of evaluation can be done both in PASCAL-XSC and C-XSC, but in each of these programming languages it is formulated in a different way. In sections 1.2 and 1.3 we describe some characteristics of PASCAL-XSC and C-XSC respectively.

1.2 PASCAL-XSC

The programming language PASCAL-XSC[1,4], PASCAL eXtension for Scientific Computation, was developed to supply a tool for the numerical solution of scientific problems. The mathematical definition of the arithmetic is an intrinsic part of the language, including optimal arithmetic operations with direct roundings that are directly accessible in the language. Further arithmetic operations for interval, complex and complex interval numbers and even for vector/matrix operations provided by precompiled arithmetical modules are defined with maximum accuracy.

PASCAL-XSC is an extension of ISO Standard PASCAL with many useful concepts for scientific computation. However, an extended compiler is required. The main concepts of PASCAL-XSC are:

- Universal operator concept (user-defined operators);
- Functions and operators with arbitrary result type;
- Overloading of procedures, functions, and operators;
- Access to subarrays;
- String concept;
- Controlled rounding;
- Optimal (exact) scalar product;
- Predefined type dotprecision (a fixed-point data type to cover the whole range of floating-point products);
- Additional arithmetic built-in types such as complex, interval, rvector, rmatrix, etc;
- Highly accurate arithmetic for built-in types;
- Highly accurate elementary functions;
- Exact evaluation of expressions within accurate expressions (#-expressions);

1.2.1 Accurate Expressions

In many numerical algorithms the exact evaluation of scalar products are extremely important for the overvall accuracy of results. For this purpose, the new type dotprecision was introduced into PASCAL-XSC representing a fixed-point format covering the whole range of floating-point products. This format allows scalar results - especially sums of floating-point products - to be stored exactly. Furthermore, scalar product expressions (dot product expressions) with vector and matrix operands with only one rounding per component can be computed via exact evaluation within accurate expressions (#-expressions).

Accurate expressions (#-expressions) can be formed based on dotprecision data type by an accurate symbol (#,#*,#<,#>, or ##) followed by an exact expression enclosed in parentheses. The exact expression must have the form of a dot product expression in scalar, vector or matrix structure and is evaluated without any rounding error. The final result can be stored without rounding error or rounded to a floating-point result (scalar, vector or matrix, respectively). Because of this, the result of an accurate expression is of maximum accuracy in the sense that in every component of the result there is no floating-point number between the exact value and the computed one. That is, the rounded and the exact result differ at most by 1 unit in the last place of the mantissa.

To obtain the unrounded or correctly rounded result of a dot product expression, you need to parenthesize the expression and precede it by the symbol #, optionally followed by a symbol for the rounding mode.

Table 1: Symbols for rounding mode.

Symbol	Expression Form	Rounding Mode
#*	scalar, vector, or matrix	nearest
#<	scalar, vector, or matrix	downwards
#>	scalar, vector, or matrix	upwards
##	scalar, vector, or matrix	smallest enclosing interval
#	scalar only	exact, no rounding

1.3 C-XSC

C-XSC[2,3], C for eXtended Scientific Computation, is a programming tool for the development of numerical algorithms providing highly accurate and automatically verified results.

The programming language C++, an object-oriented extension of the programming language C, does not provide better facilities than C to programming numerical algorithms, but its new concept of abstract data structures (classes) and the concept of overloading operators and functions provide the possibility to create a programming tool.

In contrast to PASCAL-XSC, C-XSC is not an extension of the standard language, but a class library which is written in C++. Therefore, no special compiler is needed. 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 given mathematical problem in verified bounds. Such algorithms deliver a precise mathematical statement about the true solution. The main concepts of C-XSC are:

- Real, complex, interval, and complex interval arithmetic with mathematically defined proprierties;
- Dynamic vectors and matrices;
- Subarrays of vector and matrices;
- Dot precision data type;
- Predefmed arithmetic operators with highest accuracy;
- Standard functions of high accuracy;
- Dynamic multiple-precision arithmetic and standard functions;
- Rounding control for I/O data;
- Numerical results with mathematical rigor;

1.3.1 Evaluation of Expression 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. 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. To obtain an evaluation with l ulp accuracy, C-XSC provides the dotprecision data types dotprecision, cdotprecision, idotprecision and cidotprecision.

In C-XSC the user has to transform vector products in one loop and matrix products in three loops to compute each component as simple scalar products. Intermediate results of a dot product expression can be computed and stored in a dotprecision variable without any rounding error. For all dotprecision 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.

1.4 Comparison of PASCAL-XSC and C-XSC

PASCAL-XSC and C-XSC both provide the possibility of evaluation of dot products expressions with high accuracy, but the way it can be calculated using PASCAL-XSC and C-XSC is very different.

Using PASCAL-XSC to obtain the unrounded or correctly rounded re-sult of a dot product expression, you need to parenthesize the expression and precede it by the symbol #, optionally followed by a symbol for the rounding mode. The expression must have the form of a dot product expression in scalar, vector or matrix structure and is evaluated without any rounding error.

Using C-XSC to obtain the solution for the same dot product expression, you need to write loops, use the function accumulate, use dotprecision variables for intermediate results, and in the end of the calculation of each element, you should round it.

Supposing that we want to calculate the following expression with high accuracy: y0 = b - A * x1 - A * x0 , and y0, b, x1, x0 are rvectors and A is an rmatrix. The solutions using PASCAL-XSC and C-XSC are shown in algorithms l and 2 respectively.

Algorithm l Solution using PASCAL-XSC

```
y0:= #*(b-A*xl-A*x0);
```

Algorithm 2 Solution using C-XSC

```
for (i=Lb(A,l);i<=Ub(A,l);i++)
{
accu = b [i];
accumulate(accu,-A[Row(i)],xl); accumulate(accu,-A[Row(i)],x0); y0[i]=rnd(accu);
}
```

The accurate symbol # could not be implemented in C-XSC due to several reasons. Some of them are:

- The symbol # has a different meaning in C++. It is used to include libraries, files and for other purposes in the pre-processor.
- In C++, no symbols are available which could be used for accurate expressions. In contrast in PASCAL-XSC new symbols were introduced for this purpose.
- It is not possible to change the semantics of expressions like in PASCAL.
- C-XSC was implemented as a C++ class library, different from the implementation of the PASCAL-XSC where the programmers could modify the compiler.

2 AIMS AND IDEAS

As it is shown in section 1.4, it is not so easy to implement dot products in C-XSC as in PASCAL-XSC. The aim of our work was to simplify the way the user could use dot products in C-XSC. The idea was to implement a similar concept like # expressions used in PASCAL-XSC.

The objective is to implement a library which contains the routines to evaluate an expression with high accuracy. The user should only write it as a mathematical expression, and our library should be called and execute the defined loops, the use of accumulate functions, etc, for that case. Then, the user should only write something like in PASCAL-XSC for calculating dot products.

We discussed many ways how it could be done. Some of these ideas are:

- Change GNU compiler;
- Construct a pre-compiler;
- Use Templates;
- Use Operator Overloading;

Discussing about the first idea, change the GNU compiler, we thought it would be too error prone and complicated. If we do a small modification in a wrong place, it can make all the compiler useless, or at least make it work wrong. It would be complicated as well, because the GNU compiler is not small, and it would take some time to understand how it works and think what could be changed.

The second idea, construct a pre-compiler, was consider a good idea, because it is easier than write a full compiler. But after some discussions we realised that the compilation could not be local to the expression, because we need the datatypes to evaluate the expression. We would need to read all the program because the variable definitions could be everywhere in the program, só we would need to write a full compiler.

The concept of templates[6] in C++ may be useful for this implementation. Using templates does not make programs slower. The time is spent in compilation time, not in execution time, what makes the program faster. We would need to use specialization with templates. After some tests we knew that specialization using C-XSC data types worked correctly. The problem would be the number of arguments and the generalization for any expression. This idea should be analyzed more carefully in the future.

Operator overloading seemed to be the best idea considering the short time we had. The idea is to redefine the result type of all matrix and vector operators of C-XSC as a dotprecision type. Then the user can write scalar product expressions in a mathematical way. The redefined operators are called and the expression is solved with high accuracy with no extra work for the user. This would imply the use of many temporary variables of type dot precision, what would create an overhead for the construction and destruction of that variables and would make the execution time of the program worse than if the user prefers to write the loops by himself. In the other hand it would be easier and safer for the user to use the new implementation.

3 THE FIRST IMPLEMENTATION - OPERATOR OVERLOADING

The idea was to implement the new dot precision operators with operator overloading. The results of all operations are dot precision values, vectors or matrices. Only the final result is rounded to the regular floating-point value, vector or matrix. This means that instead of one dot precision variable we use many of these, and it involves an overhead for the creation and the destruction of all this variables.

We implemented the overloaded operators using reference arguments (&) in order to keep the overhead as small as possible. This version worked in most cases, but we had some problems related to ambiguities between the C-XSC version and the new version. We noticed that the compiler used the new implementation for simple operations, but for expressions it used the C-XSC existing operator. As the results were constants, the compiler prefered to call an operator defined with constants, and this was the C-XSC operator. When we took out the reference (or added a const before the data type), we got an error from the compiler because it was ambiguous between the new operator and the C-XSC existing operator. The compiler could not decide which operator it should use.

We thought about using namespaces, but it did not solve the problem. The namespace concept cannot avoid the ambiguities. Namespaces can be concatenated, but in this case, the both spaces are being used. One namespace do not annulate the other. Another problem with namespaces is that it cannot be used easily with operators. An operator symbol cannot be qualified with a namespace, instead of *a+b* we would have to use function style operators like *a.name::operator+(b).*

Another point is that the user should be able to choose between using the accurate expression or not.

4 THE SECOND IMPLEMENTATION

4.1 Function

The second implementation uses normal functions to solve the expressions. These functions are defined for scalars, vectors and matrices of real, complex, interval and complex interval data types of C-XSC. Using function overloading ali functions for addition can be named identically, and the same for subtraction, multiplication, rounding, etc.

The user may call the nested functions to solve the mathematical expression in just one line. Using the same example that was used in section 2, supposing that we want to calculate the following expression with high accuracy: *yo = b — A * xi — A * XQ* , and *yo, b, xi, XQ* are rvectors and *A* is rmatrix. Solving this problem with the second implementation, it would be done by the user as shown in algorithm 3:

Algorithm 3 Solution using the second implementation

```
y0=dprnd(dpadd(dpadd(b,dpmul(dpminus(A),x1)),dpmul(dpminus(A),x0)));
or
y0=dprnd(dpsub(dpsub(b,dpmul(A,x1)),dpmul(A,x0)));
```

As it can be seen in the example, it is not easy to read the expression. This is in contrast with our original aim that was to make it simpler for the user. Because of these disadvantages, we decided to write a small pre-compiler to make it possible to use an operator notation.

4.2 Pre-Compiler

A pre-compiler reads a text file containing an extended C++ program with some special symbols and transforms it in a text file which is afterwards passed to a normal C++ compiler. This concept seemed to be very complicated for us, because we believed that we would need to analyze all the program in order to determine the data types of the operands. In this case we would need to write nearly a full compiler.

As described in the previous section, all the functions are named identically for all data types with function overloading. Thus, we recognize that we do not need to know the data types and therefore we can just make a mechanical transformation from operator symbols in accurate expressions (#-expressions) to function calls.

Supposing that we want to calculate the following expression with high accuracy: *yo = b- A * x1 - A * x0,* and *y0, b, x1, x0* are rvectors and *A* is rmatrix. The solution using C-XSC and the pre-compiler is:

Algorithm 4 User solution using C-XSC and pre-compiler

```
y0 = #*(b-A*x1-A*x0)
```

Algorithm 5 Solution generated by the pre-compiler

```
y0 = dprnd(dpsub(dpsub(b,dpmul(A,x1)),dpmul(A,x0)));
```

This pre-compiler is powerful enough for most of the dot expressions but does NOT analyze the syntax/semantics of a C++ program. There are the following limitations:

- there must be no whitespace between # and (
- the syntax for...sum for summation loops is not implemented
- ( ) are not allowed in #-expressions
- no syntax analysis is performed for the operands, therefore one should not use arithmetic operators in index expressions;

- comments, string literais, char literais etc. are not handled correctly.

The program will e.g. be confused by the following lines:

Table 2: Expressions that are not handled by the program.

Expression	The pre-processor thinks it should
x=#(a+b/*+c*/);	add a, b/*, and */
x = #(a+b['*']);	Multiply b[' with ']
x = #(x[i+l]);	add x[i and 1]

5 CHARACTERISTICS OF THE LIBRARY

The implemented library follows the philosophy of C-XSC. We defined a namespace (accuexp) that contains all the function defmitions for all data types defined in C-XSC. As in C-XSC, we separated the matrices and vectors of real, complex, interval and complex interval in different files (rmatrix, rvector, cmatrix, cvector, imatrix, ivector, cimatrix and civector).

We defined a class for each data type that defines some functions and operators for matrices and vectors composed by real, complex, interval or complex interval dot precision elements. The result of the implemented functions are always dot precision. The following functions, ect are defined:

- a constructor, a copy constructor and a destructor;
- an assignment operator;
- addition and subtraction of different combinations of C-XSC data types and dot precision types;
- multiplication, for the C-XSC data types;
- the monadic subtraction for the C-XSC data types;
- different roundings to the basic C-XSC data types;

The implemented library have some restrictions:

- No check index ranges: we use always from O to n-1. But C-XSC uses normally from l to n.
- We assume that the dimension of matrices and vectors are the same.

6 EXECUTION TIME

The execution time when using C-XSC are worse than using just C++. We decided to make some measurements of the time to analyze if this implementation to make it easy to the user is faster or slower than the version the user would implement by himself.

In the tables 3,4,5 and 6 we show the quotient between the execution time of the new version and the C-XSC version. The measurement was made using a repetition factor 100 to 100.000 depending on the dimension, the operation and the data type.

Table 3: Matrix addition

+	rmatrix	cmatrix	imatrix	cimatrix
2x2	2.6875	2.6966	2.6889	2.5481
4x4	2.4788	2.5903	2.3525	2.2583
8x8	2.4639	4.2946	4.1053	4.2453
16x16	4.3992	5.0373	4.9761	4.2702

Table 4: Matrix multiplication

*	rmatrix	cmatrix	imatrix	cimatrix
2x2	2.3418	1.9667	2.4114	1.9283
4x4	1.5622	1.5347	1.7797	1.5435
8x8	1.3102	1.5557	1.8883	1.6933
16x16	1.3977	1.3738	1.6293	1.4156

Table 5: Vector addition

+	rvector	cvector	ivector	civector
2	3.5443	3.3144	3.1813	2.4735
4	2.8305	2.7126	2.6118	2.2573
8	2.5786	2.5328	2.4325	2.2252
16	2.4834	2.5006	2.5308	2.3133

Table 6: Vector multiplication

*	rvector	cvector	ivector	civector
2	1.9455	1.7216	2.4317	1.5032
4	1.6129	1.3583	1.9120	1.3058
8	1.3196	1.2173	1.4533	1.2198
16	1.1680	1.1700	1.3297	1.1390

Comparing the new implementation with the old way of programing we could observate the following efects:

- in big matrices/vectors the diference is not so high as in small matrices/vector. It is caused because in bigger dimensions we have less overhead for management and construction/destruction of temporary variables than in smaller dimensions.
- in big dimensions of matrices we can observe a considerable increase in the execution time in the new implementation. It could be explained as a cache efect. The high size matrices do not fit in the cache memory, what make the execution time worse.

As we predicted, the new implementation is slower than the version that can be implemented by the user because of some reasons:

- we need to generate one or several dot precision matrix/vector.
- sometimes we need to add dot precision matrices/vectors, otherwise we would only need accumulate floating-point value to a dot precision variable.
- objects need to be copied when they are returned as a result of an operation.

This means that copy constructor should be called and some objects should be generated.

In the other hand, using the new version, the user do not need to worry about implement loops, and create dot precision variables. The user should only write as a mathematical expression, enclosed by parentheses and select the rounding mode. All the extra work is done by the preprocessor and our library.

7 CONCLUSIONS

C-XSC is a programming library for Extended Scientific Computing which support many data types with high accuracy. However it does not provide the accurate expressions which are already available in the older language PASCAL-XSC. These expressions have to be replaced by long loops which are more difficult to understand.

We studied possibilities how this accurate expressions could be implemented in C-XSC. We considered alternatives like operator overloading with special result types avoiding rounding errors, and a small pre-processor which converts accurate expressions to functions calls.

The use of the pre-compiler and the implemented functions solved the problem we want to solve: the user can just write the expression in a mathematical way, like in PASCAL-XSC, and this expression will be solved with high accuracy.

This version is basically working, but is not optimized. The performance using the new implementation was not so good compared to the use of C-XSC only. There are somethings that should be done in a future version:

- index ranges and checks;
- improvements in the performance;
- more sophisticated pre-compiler.

ACKNOWLEDGEMENT

We would like to express our gratitude to Prof. Walter Krämer of Wuppertal University and to Dr. Rudi Klatte of Karlsruhe University for many suggestions and fruitful discussions.

8 REFERENCES

[1] Hammer, R.; Hocks, M.; Kulisch, U.; Ratz, D. : *Numerical Toolbox for Verified Computing I- Basic Numerical Problems,* Springer Verlag, Berlin, 1993.

[2] Hofschuster, W.; Kraemer, W.; Wedner, S.; Wiethoff, A.: *C-XSC 2.0 A C++ Class Library for Extended Scientific Computing,* Preprint, Wup-pertal Universitaet, 2001/1.

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

[4] Klatte, R.; Kulisch, U.; Neaga, M.; Ratz, D.; Ullrich, Ch.: *PASCAL-XSC - Language Reference With Examples,* Springer Verlag, Berlin, 1992.

[5] Kulisch, U.; Miranker, W.L.: *Computer Arithmetic in Theory and Practice,* Academic Press, New York, 1981.

[6] Stroustrup, B.: *The C++ Programming Language,* Special Edition, Addison-Wesley, 2000.