Bergische Universität
Wuppertal

Sparse Matrices and Vectors in C-XSC

Michael Zimmer, Walter Krämer, Werner Hofschuster

Preprint
BUW-WRSWT 2009/7

Wissenschaftliches Rechnen/
Softwaretechnologie

Impressum

Herausgeber:	Prof. Dr. W. Krämer, Dr. W. Hofschuster Wissenschaftliches Rechnen/Softwaretechnologie Fachbereich C (Mathematik und Naturwissenschaften) Bergische Universität Wuppertal Gaußstr. 20 42097 Wuppertal, Germany

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

Michael Zimmer, Walter Krämer, Werner Hofschuster
Bergische Universität Wuppertal
Gaußstr. 20
42097 Wuppertal, Germany

E-mail:
`[zimmer,kraemer,hofschuster]@math.uni-wuppertal.de`

Sparse Matrices and Vectors in C-XSC

Michael Zimmer, Walter Krämer, Werner Hofschuster

December 4, 2009

Abstract

C-XSC is a C++ library for reliable scientific computing, which provides data types for dense vectors and matrices with real, complex, real interval and complex interval entries. These data types are easy to use and provide many helpful functionalities such as the ability to work with submatrices and subvectors. However, when dealing with sparse vectors, and especially with sparse matrices, these data types are inefficient. Since C-XSC version 2.4.0 also special types for sparse vectors and matrices that take advantage of the sparsity, both in terms of performance and memory consumption, are provided by the library. This paper explains the basic data structures for and the implementation of these new types. Many examples and some performance tests with sparse matrices from real world applications are included.

Key words: C-XSC, sparse data types, compressed column storage, sparse matrix algorithms, K-fold precision.

AMS subject classification's: 65-04, 65F05, 65F20, 65F50, 68N15, 68R10.

1 Introduction

Sparse matrices and algorithms working with such matrices are a very important part of numerical linear algebra. They have many applications, especially in engineering. For example, discretizing partial differential equations naturally leads to sparse matrices. A sparse matrix is generally defined as a matrix which has so many zero elements that it is beneficial to take advantage of this fact by writing special algorithms using special data structures to exploit this sparsity (with respect to memory consumption or addressing performance issues).

Writing efficient code for sparse matrices is in general very complicated. For example, coding the software packages for the sparse LU-, sparse Cholesky-, and sparse QR-decomposition in Matlab is done using in total more than 100,000 lines of code [4]. Even the basic operations, like matrix addition and matrix-matrix products, are a lot more complicated than the corresponding operations for dense matrices. However, exploiting the sparsity of a given problem can lead to tremendous performance gains and also allows to tackle problems of far greater dimensions due to drastically reduced memory requirements.

The C-XSC (eXtended Scientific Computing) library [7, 6] is a C++ class library for reliable scientific computing, which provides many basic data types for real, complex, and especially interval computations. These data types also include dense matrix and vector types. Due to the utilization of object oriented principles and the operator overloading capabilities of C++, using these types is very straight forward. C-XSC also provides many more useful features like the possibility to change the index ranges of matrices and vectors, and to work with subvectors and slices of matrices. Also many helpful functions are predefined. It also provides the ability to compute dot products or whole dot product expressions in (simulated) higher or even maximum precision. The included toolbox package and other additional software packages available online implement many useful algorithms from the field of reliable computing which allow to compute verified results for many common numerical problems. C-XSC is free open source software released under the LGPL and can be downloaded from [1].

Since version 2.4.0, C-XSC also contains data types for sparse matrices and vectors. This paper explains the implementation and usage of these new data types in detail and also gives some examples and performance tests. The algorithms in this paper are given in a C++ like pseudo code. For the sake of simplicity, matrices are assumed to be square in these algorithms (they can easily be adapted to the non-square case as it has been done for our C-XSC implementation).

This paper is organized in the following way. First, in Section 2, an overview over the dense matrix and vector types and their capabilities is given. Section 3 then explains the basic data structure used for the sparse types and also contains a few examples and algorithms demonstrating the use of these sparse structures. The next Section 4 then deals with the actual implementation of these data structures in C-XSC. It explains the general structure and the usage of the new types in detail. Section 5 gives some examples and test results for the new data types. Finally, Section 6 ends this paper with some concluding remarks and a short outlook to future work concerning sparse computations with C-XSC.

2 Dense Matrices and Vectors in C-XSC

In this section the basic C-XSC data types and especially the corresponding types for dense matrices and vectors are explained in detail. Since the sparse types explained in Section 4 were implemented with the goal of providing essentially the same functionality and the same interface as the dense types, all of the features and functions explained in this section also apply to the sparse matrices and vectors in C-XSC, except if explicitly stated otherwise.

C-XSC uses four basic data types:

- `real`: Real floating point numbers (`real` is a wrapper class for `double`)
- `complex`: Complex floating point numbers (the real and imaginary part are of type `real`)
- `interval`: Real floating point intervals (infimum and supremum of the interval are of type `real`)
- `cinterval`: Complex floating point intervals (infimum and supremum of the interval are of type `complex`)

C-XSC provides a unique data type for dense matrices and vectors based on these four basic data types (the elements of the matrix or vector are of type `real`, `complex`,

`interval` or `cinterval`). These matrix and vector types are:

- For basic type `real`: `rvector`, `rmatrix`
- For basic type `complex`: `cvector`, `cmatrix`
- For basic type `interval`: `ivector`, `imatrix`
- For basic type `cinterval`: `civector`, `cimatrix`

All these scalar, vector and matrix types feature overloaded arithmetic and relational operators which make it easy to use them in a program. Table 1 shows all predefined arithmetic operators, while Table 2 shows all predefined relational operators.

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

$|$: Convex Hull $\&$: Intersection ^{1}: Dot Product with choosable accuracy

Table 1: Predefined arithmetic operators

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

$\vee_{all} = \{==, !=, <=, <, >=, >\}$ $\quad \vee_{eq} = \{==, !=\}$

$\vee_{\subset} = \{==, !=, <=, <\}$ $\quad \vee_{\supset} = \{==, !=, >=, >\}$

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

Table 2: Predefined relational operators

Additional operators for the input, output and assignment of objects and copy constructors are also available. Elements of a vector or matrix can be accessed using the `[]` or `[][]` operator.

When computing dot products or operations containing dot products (like a matrix-vector product) with C-XSC, the user can choose which precision should be used for these dot products. For dot products computed through operators, this can be done by setting the global variable `opdotprec`. Here, a value of 0 indicates maximum precision (this is the default), a value of 1 means pure floating point computations and a value ≤ 2 means using simulated higher precision (a value of k means k-fold double precision). Higher precision results in slower computing times but also more accurate results. The user should choose the precision depending on the application. More details can be found in [17]. Listing 1 shows a simple example program demonstrating the use of these basic functionalities.

Listing 1: Usage of basic matrix/vector functionalities

```
//Include matrix headers
//Matrix headers also include vector headers
#include <rmatrix.hpp>
#include <ivector.hpp>
#include <iostream>

using namespace cxsc;
using namespace std;

int main() {
  int n = 100; //Dimension
  //Real matrix of dimension nxn
  rmatrix A(n,n);
  //Interval vectors of dimension n
  ivector x(n),y(n);

  //Set all elements of A and x to given values
  A = 1.0; x = interval(1.0,2.0);

  //Set dot product precision to floating point
  opdotprec = 1;
```

```
  //Compute (real matrix)-(interval vector) product using an operator
  y = A*x;

  //Output of resulting interval vector
  cout << y << endl;

  return 0;
}
```

It is also possible to compute a whole dot product expression (a mathematical expression only containing $+, -$ as well as dot product computations), like, for example, $b - Ax$, where b and x are vectors of dimension n and A is a $n \times n$ matrix, in higher precision. When computing such an expression by using the operators, the intermediate result of each operation will be individually rounded into `double` precision, independent of the selected dot product precision. This, in general, introduces additional rounding errors. The whole computation can instead be performed using a `dotprecision` variable which can store intermediate results exactly in a fixed point format [10, 11]. The precision of the dot products must then be selected by using the `set_dotprec` function of `dotprecision`. Listing 2 gives a short example of how to use this variable. For more details we again refer to [17].

Listing 2: Computing dot product expressions

```
#include <rmatrix.hpp>
#include <dot.hpp>
#include <iostream>

using namespace cxsc;
using namespace std;

int main() {
  int n = 100; //Dimension
  //Real matrix of dimension nxn
  rmatrix A(n,n);
  //Real vectors of dimension n
  rvector x(n),b(n);

  //Set elements of A, x and b
  for(int i=1 ; i<=n ; i++) {
    x[i] = i*1e-16;
    b[i] = i;
    for(int j=1 ; j<=n ; j++) {
      A[i][j] = (i>j) = i : j;
    }
  }

  //Compute x=b-A*x using dotprecision data type
  dotprecision dot = 0.0;
  //Dot products computed in simulated 2-fold double precision
  dot.set_dotprec(2);
  for(int i=1 ; i<=n ; i++) {
    //Store b[i] in variable dot
    dot = b[i];
    //Compute dot product of i-th row of A and x in 2-fold double precision
    //and add intermediate result to variable dot (exactly!)
    accumulate(dot,-A[i],x);
    //Round final result to double precision
    x[i] = rnd(dot);
  }

  //Output of resulting vector x
  cout << x << endl;

  return 0;
}
```

C-XSC matrices and vectors also have two additional features which can be very useful in practical applications. This first is the possibility to freely change the index range of the matrix elements. By default, all matrices and vectors use a one based index range (usual indexing in mathematics). Depending on the algorithm or application which the user wants to implement, it can be beneficial to use a zero based or entirely different index range (negative indices are also possible). Listing 3 shows how to change the index range.

Listing 3: Changing the index range of matrices and vectors

```
#include <rmatrix.hpp>
#include <dot.hpp>
#include <iostream>

using namespace cxsc;
using namespace std;

int main() {
  int n = 100; //Dimension
  //Real matrix of dimension nxn
  rmatrix A(n,n);
  //Real vector of dimension n
  rvector x(n);

  //Set lower index bound of x to zero
  //Upper bound is adjusted automatically to n-1
  SetLb(x,0);

  //Index bounds can be read with Lb and Ub functions
  cout << "Index␣bounds␣of␣x:␣" << Lb(x) << "," << Ub(x) << endl;

  //Set upper bound of row indices of A to n-1
  //Lower bound will automatically be adjusted to 0
  SetUb(A,ROW,n-1);

  //Operators work correctly with different index ranges
  cout << A*x << endl;

  //When looping over elements new index bounds must be
  //taken into account!
  for(int i=Lb(x) ; i<=Ub(x) ; i++)
    x[i]++;

  return 0;
}
```

Another very helpful feature is the ability to cut slices out of vectors and matrices. These slices are not just copies of the original data, but actual references to the original memory location. That means that changes to these slices also affect the data of the original matrix. Slices can be accessed with the `()`-operator, as demonstrated in Listing 4.

Listing 4: Using slices of matrices and vectors

```
#include <rmatrix.hpp>
#include <iostream>

using namespace cxsc;
using namespace std;

int main() {
  int n = 100; //Dimension
  //Real matrix of dimension n by n
  rmatrix A(n,n);
  //Real vector of dimension n
  rvector x(n);

  x = 0.0; A = 0.0;

  //Output of left upper corner of A
  cout << A(1,10,1,10) << endl:

  //Set first ten elements of x to 1
  x(1,10) = 1.0;

  //Set right upper corner of A to 2.0
  A(1,10,91,100) = 2.0;

  //further computations ...

  return 0;
}
```

C-XSC also defines many different functions for matrices and vectors and provides some algorithms using the types in the C-XSC toolbox. For more details, we refer to [7, 6, 9].

3 Sparse Data Structures

To take advantage of the sparsity of a matrix or vector, an appropriate data structure has to be choosen. This section describes the most commonly used data structures, gives some examples for common algorithms using these data structures and explains which are used in the actual C-XSC implementation described in Section 4.

The basic idea of every sparse data structure is to not store any information about the zero entries, but only the information (value and indices) of the non zero entries of the matrix. For vectors, this can be done in a very straight-forward manner with two arrays, where one array stores the position of the non zero entry and another array stores the value of the respective entry, as seen in Example 3.1.

Example 3.1 *The vector*

$$x = (\ 0 \quad 2 \quad 0 \quad 0 \quad 5 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0\ )^T$$

can be stored in two arrays exploiting its sparsity in the following way:

$$\begin{array}{rcl} p & = & [\ 1 \quad 4 \quad 6\ ] \\ x & = & [\ 2 \quad 5 \quad 1\ ] \end{array}$$

The array p in the above example stores the index of each non zero element (in this paper indices are always zero based, if not otherwise stated), while x stores its value. The i-th entries of both arrays refer to the same element, so that for each integer $i = 0, \ldots, nnz - 1$ (nnz denotes the number of non zero entries of the respective data structure throughout this paper) the element at position p_i of the vector has value x_i, and every element of the vector whose index is not an entry of p is zero.

Obviously, this data structure can be used independent of the type of the elements of the vector (real, complex, interval, ...). This data structure has a memory requirement of $2nnz$ opposed to n for storing the full vector. Algorithms using this data type can also be formulated in a straight forward way. The dot product of two sparse vectors a and b stored in the above manner can, for example, be formulated as seen in Algorithm 1 (the data structure must be sorted ascending by the indices of the elements).

```
Input: Two sparse vectors a and b
Output: The result res of the dot product a · b
res = 0; i = 0; j = 0
while i < a.nnz and j < b.nnz do
    if a.p[i] == b.p[j] then
        res += a.x[i] * b.x[j]
    else if a.p[i] < b.p[j] then
        i++
    else if a.p[i] > b.p[j] then
        j++
```

Algorithm 1: Dot product of sparse vectors

Data structures for sparse matrices are somewhat more complex and more sophisticated. There is also a straightforward approach called the triplet form, which is basically

an extension of the data structure for sparse vectors explained above to the two dimensional case, but this method is normally not optimal. The triplet form uses three arrays, where one stores the row index, one stores the column index and one stores the value of every non zero entry in the matrix, as seen in the following example:

Example 3.2 *The matrix*

$$\begin{pmatrix} 0 & 0 & 0 & 1.1 & 0 \\ 0 & 2.3 & 0 & 0 & 1.0 \\ 0 & 0 & 0 & 0 & 0 \\ 3.3 & 0 & 2.1 & 0 & 0 \\ 0 & 0 & 6.5 & 0 & 0 \end{pmatrix}$$

can be stored in triplet form in three arrays in the following way:

$$\begin{array}{lcl} row & = & [\ 0 \quad 1 \quad 1 \quad 3 \quad 3 \quad 4\ ] \\ col & = & [\ 3 \quad 1 \quad 4 \quad 0 \quad 2 \quad 2\ ] \\ x & = & [\ 1.1 \quad 2.3 \quad 1.0 \quad 3.3 \quad 2.1 \quad 6.5\ ] \end{array}$$

As for vectors, the i-th element of each array refers to the same entry of the matrix. So for each $i = 0, \ldots, nnz - 1$, there exists a non zero entry in the matrix with index pair row_i, col_i and value x_i. All other entries of the matrix are zero. The triplet form has a memory requirement of $3nnz$ instead of n^2 for the full matrix. In the above example, the data structure is sorted by rows, but in general the elements can be stored in any order. Although the triplet form is easy to create and understand, it is in general hard to write sparse algorithms using the triplet form in an efficient way.

A more sophisticated approach for storing a general sparse matrix is to use compressed row or compressed column storage (in the following only compressed column storage is used, compressed row storage works in the same way). Compressed column storage also uses three arrays. The array ind storing the row indices and the array x storing the values of the non zero entries work in much the same way as before. The third array p does not store the indices of the columns directly, but rather points to the start and end of each column in the ind and x arrays. The i-th column of the matrix has non zero entries at the row indices stored between ind_{p_i} and $ind_{p_{i+1}}$ with the values stored between x_{p_i} and $x_{p_{i+1}}$, as demonstrated in the following example:

Example 3.3 *The matrix*

$$\begin{pmatrix} 0 & 0 & 0 & 1.1 & 0 \\ 0 & 2.3 & 0 & 0 & 1.0 \\ 0 & 3.1 & 0 & 0 & 0 \\ 0 & 4.0 & 2.1 & 0 & 0 \\ 0 & 0 & 6.5 & 0 & 0 \end{pmatrix}$$

can be stored in compressed column form in three arrays in the following way:

$$\begin{array}{lcl} p & = & [\ 0 \quad 0 \quad\quad\quad 3 \quad\quad 5 \quad 6 \quad 7\ ] \\ ind & = & [\ \quad 1 \quad 2 \quad 3 \quad 3 \quad 4 \quad 0 \quad 1 \quad\ ] \\ x & = & [\ \quad 2.3 \quad 3.1 \quad 4.0 \quad 2.1 \quad 6.5 \quad 1.1 \quad 1.0 \quad\ ] \end{array}$$

The array p always has $n + 1$ elements, with the last element containing the number of non zero entries. Thus compressed column storage has a memory requirement of $(n+1)+2nnz$. Sparse algorithms can often be formulated in an efficient way with it. For the general case it is a good choice for a storage scheme for sparse matrices. Algorithm 2 shows how to compute the product of two matrices stored in compressed column form.

```
Input: Two sparse matrices A and B of dimension n × n stored in compressed column
       form
Output: A sparse matrix C = AB stored in compressed column form
real work(n) = 0.0; int w(m) = -1; nnz = 0
for i=0 to n-1 do
    for k=B.p[i] to B.p[i+1]-1 do
        for l=A.p[B.ind[k]] to A.p[B.ind[k]+1]-1 do
            if w[A.ind[l]] < i then
                w[A.ind[l]] = i
                C.ind.addElement(A.ind[l])
                work[A.ind[l]] = A.x[l] * B.x[k]
                nnz++
            else
                work[A.ind[l]] += A.x[l] * B.x[k]
    for j=C.p[i] to nnz do
        C.x.addElement(dot[C.ind[j]])
    C.p[i+1] = nnz
```

Algorithm 2: Product of sparse matrices in compressed column form

To get familiar with the compressed column storage format used to store sparse matrices in C-XSC, the reader should verify Algorithm 2.

Special algorithms also have to be formulated for operations between sparse and full matrices. The result of such an operation can in general be assumed to be full (this is also done in Matlab, for example [5]). Algorithm 3 shows how to compute the product of a full matrix and a sparse matrix stored in compressed column form. More sparse algorithms can be found in [4].

```
Input: A full matrix A and a sparse matrix B stored in compressed column form
Output: A full matrix C = AB
for i=0 to n-1 do
    for j=0 to n-1 do
        for k=B.p[j] to B.p[j+1]-1 do
            C(i,j) += A(i, B.ind[k]) * B.x[k]
```

Algorithm 3: Product of a full and a sparse matrix

In the C-XSC implementation of sparse matrices, explained in detail in Section 4, the compressed column storage is used. Another reason for this, beside the advantages already mentioned, is that this storage scheme is used in many other software packages for

sparse computations such as Matlab [14] and CSparse [4]. This makes it easier to write interfaces between C-XSC and these software packages if necessary, for example to use an LU-decomposition function as provided by CSparse..

The data structures used in C-XSC have some small restrictions and differences compared to, for example, the data structures used in CSparse, which have to be taken into account when directly accessing the arrays describing the sparse C-XSC data structures:

- The arrays of the sparse vector data structure have to be sorted ascending by the index of the element.
- Each column of the compressed column storage for matrices has to be sorted ascending by row indices (the array ind has to be sorted between index p_i and p_{i+1} for every $i \in \mathbb{N}_0 < n + 1$ and the elements of x must be arranged accordingly).
- Zero elements in sparse vectors and matrices (for example, due to numerical cancellation in computations) are allowed.

When writing an interface to another software package or directly manipulating the basic arrays of a sparse C-XSC type (see Section 4), these differences must be taken into account to avoid undefined behaviour when using these types.

4 Implementation of Sparse Data Types for C-XSC

This section describes the actual implementation of the sparse data types in C-XSC. As mentioned before, one of the major goals of the implementation is to provide the same functionality with the same interface (especially the same operators) as the dense matrix and vector types described in Section 2. It should thus be possible to write a template function, which can use dense and sparse types without further changes to the function itself (although it is often not a good idea to use code written for dense matrices for sparse matrices without adapting it).

Other major goals were to achieve good performance and low memory requirements (although clarity of code and easy usability have in some cases been given the priority over maximum performance) and to use an underlying data structure that is commonly used in other packages. The sparse data types were designed to work efficiently for general sparse matrices and do not take a special structure (for example symmetric matrices or banded matrices) into account. As described in Section 3, compressed column storage has been used as the underlying data structure.

The sparse data types have been implemented in eight different classes corresponding to the dense matrices and vectors, with one matrix and vector class for each basic data type as described in Section 2:

- `real`: `srvector` and `srmatrix`
- `complex`: `scvector` and `scmatrix`
- `interval`: `sivector` and `simatrix`
- `cinterval`: `scivector` and `scimatrix`

As in the case of dense data structures, the corresponding header file (for example `srmatrix.hpp`) has to be included. All sparse headers include the corresponding dense headers, the matrix headers automatically include the vector headers, and the headers for

the types with elements of higher complexity include those with lower complexity (for example `simatrix.hpp` includes `srmatrix.hpp` and `scimatrix.hpp` includes all other matrix headers).

The vector classes for each basic data type have essentially the same basic structure. Each vector class has the following data members:

- The array `p` as described in Section 3 (Example 3.1), implemented as a vector from the Standard Template Library (type `std::vector<int>`).
- The array `x` as described in Section 3 (Example 3.1), also implemented as a vector from the Standard Template Library (type `std::vector<T>`, where T is the corresponding basic data type).
- Integers for the lower and upper bound of the index.
- The dimension `n` of the vector as an integer.

The number of non zero elements is implicitly stored as the `size`-Element of each of the two STL-vectors. In the same way, all sparse matrix types use the same basic structure and have the following data members:

- The array `p` as described in Section 3 (Example 3.3), implemented as a vector from the Standard Template Library (type `std::vector<int>`).
- The array `ind` as described in Section 3 (Example 3.3), also implemented as a vector from the Standard Template Library (type `std::vector<int>`).
- The array `x` as described in Section 3 (Example 3.3), also implemented as a vector from the Standard Template Library (type `std::vector<T>`, where T is the corresponding basic data type).
- Four integers storing the lower and upper bound of the index for the rows and columns.
- Integers `m` and `n` for the dimension of the matrix.

As explained in Section 3, the number of non zeros is implicitly stored in the last element of the vector `p`.

Using an STL-vector instead of a standard array has several advantages. One is that a vector can dynamically allocate more memory if needed, which can be necessary if the number of non zero elements of the matrix increases during an operation. For arrays, this would have to be done manually. Another advantage is that all the functions from the STL are available if necessary, for example, if portions of the vector need to be sorted. A possible downside is that using a vector could, at least in theory, result in a performance hit for the program, since using an operator of the STL-vector, for example the `[]`-operator to access one element, results in a function call. However, when activating compiler optimizations (which is highly recommended) and especially inlining, using the STL-vector should result in only a slight or, in the best case, no performance hit at all.

As with the dense matrix/vector classes, it is also possible to access slices of sparse matrices and vectors using the `()`-operator (see Listing 4). Since data manipulations on these slices should also affect the original data, it is necessary to use some helper classes. For the real dense matrices and vectors these are the classes `rvector_slice` (slice of a vector), `rmatrix_slice` (slice of a matrix) and `rmatrix_subv` (one row or column of a matrix). The matrix classes for the other base types have corresponding names. For the sparse data types, there are corresponding classes `srvector_slice`, `srmatrix_slice`,

`srmatrix_subv` etc. These data types are normally not directly relevant for the user and are mainly used internally by C-XSC.

In the dense case, these helper classes basically point to the respective memory location of the original data, which works quite nicely, since the whole matrix or vector is stored in one continuous block of memory. For the sparse types, this is generally not the case. For slices of a sparse vector, the helper class can still be implemented in a relatively easy fashion. It stores a reference to the original `p` and `x` arrays and the start and end index of the slice according to these arrays. For sparse matrices however, the implementation is more difficult.

Since most operations do not modify the original data (for example a simple addition does not change anything, only assignment operators are of importance), C-XSC stores a copy of the original data of the slice in the helper class, which is used for all computations that do not manipulate the original data, and also a pointer to the original sparse matrix from which the slice is cut out. If an operation changes the original data, the underlying data structure of the original sparse matrix is accessed and modified in specialized versions of the respective operators. Such operations are fairly expensive, because they generally affect the whole data structure (introducing new elements requires to shift all elements stored afterwards in the data structure). Due to the use of a copy of the original data for non-manipulating operations however, most operations with slices can still be executed with good performance.

As stated above, all sparse matrix and vector classes have basically the same structure, except for the respective basic data type used. This implies that using one template class for all sparse matrices and one template class for all sparse vectors would be a good approach for the implementation. This however was *not* done in the actual C-XSC implementation for the following reasons:

- Some special features of C-XSC (dot products in K-fold precision) would require many additional template parameters instead of only one parameter for the basic type of the elements of the vector or matrix, resulting in more complicated code.
- Compile times would increase (definitions are included in the header files, see below).
- Most operators are not class members and would thus also have to be template functions, which would result in unclear error messages in user programs (the compiler would try to use these templates for every undefined operator).
- The dense types are also not template types.

For the above reasons and to keep the basic structure of the implementation close to the structure of the dense types, all sparse matrices and vectors were implemented as simple classes. Instead, to keep redundant code to a minimum, the actual computations performed in the operators were implemented as template functions, which are called from the operators. For example, the code for the addition of two sparse matrices was implemented in a template function `spsp_mm_add`. Here, `spsp` indicates that the first and second operand are sparse, `mm` indicates that both operands are matrices, and `add` indicates that an addition is performed. The other template computation functions are named in a similar way corresponding to this pattern. Listing 5 shows the code for this function.

Listing 5: Template function for the addition of two sparse matrices

```
template<class TA, class TB, class Tres, class TElement>
inline Tres spsp_mm_add(const TA& A, const TB& B) {
  int m = ColLen(A);
```

```
  int n = RowLen(A);
  int nnz = 0;

  Tres C(m, n, A.get_nnz()+B.get_nnz());

  for(int j=0 ; j<n ; j++) {

    int k = A.p[j];
    int l = B.p[j];

    while(k<A.p[j+1] && l<B.p[j+1]) {
      if(A.ind[k] == B.ind[l]) {
        C.ind.push_back(A.ind[k]);
        C.x.push_back(A.x[k] + B.x[l]);
        k++; l++;
      } else if(A.ind[k] < B.ind[l]) {
        C.ind.push_back(A.ind[k]);
        C.x.push_back(TElement(A.x[k]));
        k++;
      } else {
        C.ind.push_back(B.ind[l]);
        C.x.push_back(TElement(B.x[l]));
        l++;
      }
      nnz++;
    }

    for( ; k<A.p[j+1] ; k++) {
      C.ind.push_back(A.ind[k]);
      C.x.push_back(TElement(A.x[k]));
      nnz++;
    }

    for( ; l<B.p[j+1] ; l++) {
      C.ind.push_back(B.ind[l]);
      C.x.push_back(TElement(B.x[l]));
      nnz++;
    }

    C.p[j+1] = nnz;

  }

  return C;
}
```

The types of both operands, of the result matrix and of the elements of the result matrix are given as template parameters. This function can now be called from each operand which adds two sparse matrices, independent of the basic data type of each matrix. For example, the `operator+` for the addition of a sparse real matrix and a sparse interval matrix can now be defined as depicted in Listing 6.

Listing 6: Operator for the addition of a real sparse matrix and a sparse interval matrix

```
inline simatrix operator+(const srmatrix& A, const simatrix& B) {
  return spsp_mm_add<srmatrix,simatrix,simatrix,interval>(A,B);
}
```

The operator simply calls the template function. Due to the use of inlining, this additional function call should normally come at no cost when using compiler optimizations. This method is used for nearly all operators defined for the sparse matrix types, which correspond to the dense operators shown in Table 1 and Table 2.

As already mentioned, the definitions of the sparse classes reside in the header files. This method is used for a lot of the C-XSC data types (also for the dense vectors and matrices), since it is necessary to allow the compiler further optimizations, especially inlining. Since every use of an operator in C++ results in a function call, precompiling the code for all data types into the library itself would make it impossible for the compiler to use inlining, which would avoid these calls by directly copying the code of the respective function to the location of the call. This leads to significant performance gains for programs using C-XSC. The downside is that it also increases the compile time of C-XSC programs and increases the code size of the final program somewhat. We feel, however, that the performance increases are well worth this relatively minor disadvantage. For more information

on this topic and on how to compile the C-XSC library and C-XSC programs in a way that leads to well performing programs we refer to [16].

As stated before, the sparse data types offer the same features as the dense types explained in Section 2. We shortly repeat those features before explaining some important differences between the sparse and the dense matrix/vector types:

- Operators for all basic operations according to the overviews from Table 1 and Table 2.
- Choosable precision for dot products (tune for speed or accuracy).
- Computation of dot product expressions in high accuracy using the `dotprecision` types.
- Working with slices and subvectors (both reading and writing data).
- Variable index range for row and column access.
- Many useful predefined functions.

While one goal for the implementation of the sparse types was to keep the interfaces and set of features close to the dense types, there are still some differences due to the nature of sparse data structures. One main difference is the availability of a set of constructors: There are basically three different ways to construct a sparse matrix in C-XSC. In the following we show how to construct a sparse $n \times n$ matrix

$$A = (a_{ij}) = \begin{cases} i, & i = j \\ 0, & else \end{cases}$$

The first, and in general the worst in terms of runtime performance, possibility is to construct the matrix in the same way as in the dense case, as shown in Listing 7.

Listing 7: Creating a sparse matrix (option 1)

```
//Dimension
int n = 10;

//Empty nxn matrix
srmatrix A(n,n);

for(int i=1 ; i<=n ; i++)
  A[i][i] = i;
```

This option is very clear and easy, the problem, however, is that it requires to access all non zero elements directly using the `[][]`-operator. This tends to be very slow in general, since adding a new element into a compressed column structure is quite expensive, as explained later in this section. The next option is to first construct a dense matrix and create the sparse matrix out of the dense matrix as shown in Listing 8.

Listing 8: Creating a sparse matrix (option 2)

```
//Dimension
int n = 10;

//Dense nxn matrix
rmatrix Tmp(n,n);
//Set dense matrix to 0
Tmp = 0.0;

//Fill dense matrix diagonal
for(int i=1 ; i<=n ; i++)
  Tmp[i][i] = i;

//Create sparse matrix
srmatrix A(Tmp);

//Delete dense matrix
Resize(Tmp);
```

This version has the downside that first a dense matrix has to be created which, depending on the dimension, might require a lot of memory. Many sparse matrix applications require huge matrices with dimensions of $100,000$ or more, which are far to big to fit into the main memory of a usual PC. The third option shown in Listing 9 is in general the best one, both in terms of performance and memory footprint.

Listing 9: Creating a sparse matrix (option 3)

```
//Dimension
int n = 10;
//Number of non zeros of new matrix
int nnz = 10;

//Vectors storing matrix elements in triplet form
intvector rows(nnz);
intvector cols(nnz);
rvector vals(nnz);

//Fill triplet form
for(int i=1 ; i<=n ; i++) {
  rows[i] = i;
  cols[i] = i;
  vals[i] = i;
}

//Create sparse matrix
srmatrix A(n,n,nnz,rows,cols,vals);
```

Here the matrix is constructed by passing the elements of the matrix in triplet form. Instead of the version above using the C-XSC type `intvector`, it is also possible to use standard arrays with elements of type `int` and `real` (or another appropriate base type). It is also possible to pass three arrays containing the matrix in compressed column form by adding the constant `compressed_column` to the end of the parameter list.

Another difference between the sparse and the dense types has to be taken into account when accessing single elements of a matrix or vector. In the dense case, one would use the `[]`-Operator, both for read and write access (for example `A[2][1]=3.0` for matrices and `x[1]=1.0` for vectors). In the sparse case, there needs to be a clear distinction between read and write access, since write access requires to return a reference to the element.

When accessing a zero element of the matrix, the element is not really stored in memory and thus a reference to this element is only possible by first adding the element to the matrix. If the same operator is used for both reading and writing, and zero elements of a matrix are read for example in a loop, then each read operation adds, unnecessarily, a new element, increasing the memory requirement of the matrix and also the computational cost of later operations with this matrix. Furthermore, all elements that are stored after the new element in the data structure have to be shifted. So if the element is added somewhere in the middle or at the beginning of the matrix, all following elements need to be moved and nearly the whole data structure must be updated. Example 4.1 demonstrates this effect with the matrix from Example 3.3.

Example 4.1 *The matrix*

$$\begin{pmatrix} 0 & 0 & 0 & 1.1 & 0 \\ 0 & 2.3 & 0 & 0 & 1.0 \\ 0 & 3.1 & 0 & 0 & 0 \\ 0 & 4.0 & 2.1 & 0 & 0 \\ 0 & 0 & 6.5 & 0 & 0 \end{pmatrix}$$

is stored in compressed column form in the following way:

$$\begin{array}{lcl} p & = & [\ 0 \quad 0 \quad\quad\quad\quad 3 \quad\quad 5 \quad 6 \quad 7\ ] \\ ind & = & [\ \quad 1 \quad 2 \quad 3 \quad 3 \quad 4 \quad 0 \quad 1 \quad\ ] \\ x & = & [\ \quad 2.3 \quad 3.1 \quad 4.0 \quad 2.1 \quad 6.5 \quad 1.1 \quad 1.0 \quad\ ] \end{array}$$

After setting the top left element of the matrix to 1.0*, the compressed column form of this matrix changes to:*

$$\begin{array}{lcl} p & = & [\ 0 \quad 1 \quad\quad\quad\quad 4 \quad\quad 6 \quad 7 \quad 8\ ] \\ ind & = & [\ 0 \quad 1 \quad 2 \quad 3 \quad 3 \quad 4 \quad 0 \quad 1 \quad\ ] \\ x & = & [\ 1.0 \quad 2.3 \quad 3.1 \quad 4.0 \quad 2.1 \quad 6.5 \quad 1.1 \quad 1.0 \quad\ ] \end{array}$$

For this change, all elements of the arrays ind *and* x *need to be shifted by one position in memory, and every element of the* p *array needs to be updated. So in this example, the whole data structure is affected by adding just one element to the matrix.*

Because of this, the sparse types make a clear distinction between write access, using the `[]`-operator, and read access using the `()`-operator. Listing 10 gives an example for the usage of these operators.

Listing 10: Accessing the elements of a sparse matrix

```
srmatrix A(10,10);

//...diagonal of A is filled...

cout << A[1][1] << endl; //Element exists,
                         //simple output
cout << A[2][1] << endl; //Element doesn't exist,
                         //gets created
cout << A(3,1) << endl;  //Element doesn't exit,
                         //doesn't get created
A[4][2] = 2.0;           //Element gets created and set
A(5,2) = 3.0;            //Error, ()-operator is read only
```

A final major difference between the sparse and dense types is the in- and output using the respective operators. By default, the in- and output for sparse matrices works similar to the dense matrices, that is, when using the output operator, the zero elements are also written to the stream, and when using the input operator, the zero elements must be written to the stream. This might be inconvenient for very large matrices. It is thus possible to switch to a sparse in-/output mode which will look like the following example:

Example 4.2 *Using sparse output, the matrix*

$$\begin{pmatrix} 1.1 & 0.0 & 3.0 & 0.0 & 0.0 \\ 0.0 & 2.4 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 2.2 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.6 & 0.0 \\ 0.0 & 0.0 & 0.0 & 6.4 & 3.2 \end{pmatrix}$$

will be written to the output stream as

```
5 5
7
0 1 2 4 5 7
0 1 0 2 3 3 4
1.1 2.4 3.0 2.2 1.6 6.4 3.2
```

In the sparse output the first two numbers depict the dimension of the matrix, the next number is nnz, and then the three arrays of the compressed column form are given. Switching the mode of the in- and output is possible with an IO-manipulator as shown in Listing 11.

Listing 11: Switching the in- and output mode for sparse matrices and vectors

```
srmatrix A(10,10);

//...

cout << A << endl; //Full (default)

cout << SparseInOut;
cout << A << endl; //Sparse

cout << FullInOut;
cout << A << endl; //Full
```

Currently there are no special algorithms, such as a LU-decomposition, for the sparse matrices implemented in C-XSC (this is planned for the future, see Section 6). Just using the possibilities offered by the classes, such as the overloaded operators, is often not sufficient to write truly efficient sparse algorithms. For this cause, C-XSC provides functions to directly access the three STL-vectors of the sparse matrices and the two STL-vectors of the sparse vectors with the following functions:

- For matrices: Functions `column_pointers()`, `row_indices()` and `values()`
- For vectors: Function `row_indices()` and `values()`

These functions return references to the vectors containing the data of the sparse matrix or vector. When manipulating these arrays it is of course very important to keep the data structure consistent to the used format. For vectors this means that the row indices have to be sorted ascending, and for matrices that the compressed column storage format with sorted row indices for each column is used. If these requirements are not met, working with the sparse object will lead to undefined behaviour and maybe even to segmentation faults when executing the respective program.

At the end of this section we finally want to give a small list of advices for working with the sparse data types, which should lead to better performing programs:

- Accessing single elements of a sparse matrix or vector should be avoided, especially repeated access in a loop. Use the predefined operators instead.
- If accessing single elements, differentiate between read and write access.
- Do not use slices or subvectors of sparse matrices excessively.
- If subvectors of a sparse matrix have to be accessed (for example when computing dot product expressions using a `dotprecision` variable), prefer column access to row access. If row access is necessary, it might be a good idea to first transpose the matrix using the `transp` function, and then to access the columns of the transposed matrix.
- When implementing special algorithms for sparse data structures, directly accessing and manipulating the underlying STL-vectors will often lead to much faster code.

5 Test Results

In this section, we present a few performance tests of the new sparse data types with sparse matrices from real world applications from the Matrix Market [2] website. We give timing results for the basic operation of a sparse matrix-matrix product with different dot product precisions and for each of the four basic data types. As a reference, some of the results are compared to CSparse [4] and Intlab [14].

The matrices from the Matrix Market come as text files in a special format. Listing 12 shows a simple example program that demonstrates how the matrices from these text files can be read out and stored as sparse C-XSC matrices.

Listing 12: Example for reading matrix market text files with C-XSC

```
/*
 * Call program this way:
 * progname < matmarketfile.mtx
 */
#include <iostream>
#include <string>
#include <srmatrix.hpp>

using namespace std;
using namespace cxsc;

int main() {
    int m,n,nnz;
    string tmp;

    //Read head of file (not important here)
    getline(cin,tmp);

    //Read dimension and number of entries
    cin >> m >> n >> nnz;

    //Triplet form
    intvector rows(nnz);
    intvector cols(nnz);
    rvector vals(nnz);

    //Read text file, fill triplet form
    for(int i=1 ; i<=nnz ; i++) {
        int x,y;
        cin >> x >> y >> vals[i];
        rows[i] = x-1;
        cols[i] = y-1;
    }

    //create matrix
    srmatrix A(m,n,nnz,rows,cols,vals);

    //Drop possible zero entries
    A.dropzeros();

    return 0;
}
```

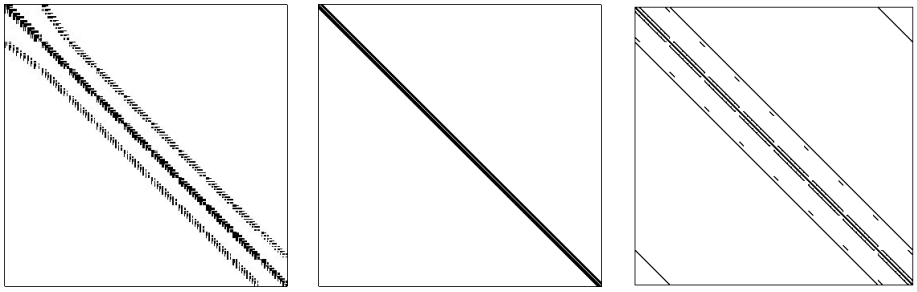
For the purpose of our tests, we look at the following three different matrices, listed with their matrix market names:

- cavity09: Real unsymmetric 1182×1182 matrix with 32702 non zero entries
- af23560: Real unsymmetric 23560×23560 matrix with 460598 non zero entries
- conf5.4-00l8x8-0500: Complex unsymmetric 49152×49152 matrix with 1916928 non zero entries

The sparsity pattern of each of these three matrices is given in Figure 1.

Storage format	Computed with	point	interval
Sparse	$K = 0$	0.138	0.284
	$K = 1$	0.009	0.047
	$K = 2$	0.032	0.088
Dense	$K = 0$	9.08	125.4
	$K = 1$	3.03	51.7
	$K = 2$	51.8	83.5
	BLAS using 8 cores	0.081	0.308

Table 3: Time measurements using one core for matrix-matrix product of *cavity09* matrix in seconds



cavity09 af23560 conf5.4-00l8x8-0500

Figure 1: Sparsity pattern of the test matrices

All tests were performed on a dual Intel Xeon 2.26 GHz machine (Nehalem architecture, each processor has four cores) with 24 GB Ram running OpenSUSE Linux 11.1. The sparse operations tested here are not parallelized, meaning that only one of the eight available cores on this machine will be used. As a compiler the GCC 4.4.2 was used with the highest optimization level.

First we measure the performance of the sparse matrix-matrix product $C = AA$ for the test matrix *cavity09* for the dot product precision setting $K = 0$ (maximum accuracy), $K = 1$ (normal floating point computation) and $K = 2$ (two-fold double precision). Furthermore, each test is also performed for the interval case, for which the matrix

$$A = T + T \cdot [-10^{-10}, 10^{-10}]$$

with T being the respective test matrix, is used. As a reference, these tests are also performed with dense matrices (with and without BLAS support, see [16]). The results of these tests are given in the Table 3.

The numerical results of the sparse and dense computations were always identical in our tests. The results in Table 3 show that the new sparse data types handle a truly sparse matrix (like the test matrix used) a lot more efficiently, just as expected. In the dense case, there is also a larger difference between the respective precision settings. This is due to the different memory access patterns necessary for the computations with higher precision,

which have a huge effect on the dense matrix-matrix product. For matrix-vector products and single dot products the difference will be much smaller and comparable to the sparse results in these tests.

Another interesting point in the results is the comparison between the highly optimized product using BLAS, which makes use of all eight cores of the test system and is still about a factor 10 slower than using the sparse computations on just one core. This again emphasizes the advantages of using special data structures for sparse matrices.

Next we compare the results and performance of the sparse C-XSC types to the C library CSparse and to the Matlab extension Intlab. CSparse is a deliberately simple (in terms of features) C library, which allows for very fast, straight-forward code. CSparse does not support interval matrices and higher precision dot products. Complex matrices are also not supported in the standard version, but there exists an extended version (CXSparse [3]) which supports complex matrices, which is used for the complex point product in the following test.

Intlab is a very popular extension to Matlab for interval computations. Since it is based on Matlab, it can make use of the very mature sparse algorithms provided by Matlab itself. Intlab does support the same basic data types as C-XSC (although (complex) intervals are implemented in midpoint-radius and not in infimum-supremum representation, allowing faster computations but introducing additional overestimation of the results) but has no support for higher precision dot products for the basic sparse matrix operations tested here. This time, no dense tests are used, since the other two test matrices beside *cavity09* are too large to be stored in dense format. The results of the comparison are presented in Table 4. An "-" entry in this Table means that this operation is not supported by the respective library.

CSparse/CXSparse is clearly the fastest library for the real and complex point products. This is due to its lightweight nature, which makes it easier for the compiler to optimize the resulting code. On the other hand, CSparse does not support higher precision dot products. Furthermore, CSparse is the most complicated from the tools compared here to use, since it is a pure C library and thus does not support object oriented concepts like operator overloading. Intlab shows about the same performance level for the real point and interval case. In the complex point case, it performs slightly better than C-XSC but is nearly on the same level, while in the complex interval case it performs about 20% to 30% better than the current C-XSC version. However, it also does not provide a sparse matrix-matrix product in higher accuracy.

6 Final Remarks

The new sparse data types further increase the scope of C-XSC, providing specialised types for sparse matrices and vectors with real and complex entries, both for the point and the interval case. Working with these types is easy, due to the concept of operator overloading and through using many of the features also being used in the case of dense matrices and vectors in C-XSC, such as slices and index range manipulation. Furthermore, they also allow to choose the precision for dot product computations at run time, which allows the user to adapt his program towards speed or accuracy.

The measured performance is comparable to Matlab/Intlab, although since this is the first version of these types in C-XSC there is still room for optimizations, which will be one field of work for the future. Additionally, there are currently no special sparse algorithms

Test matrix	Library	Test case	point	interval
cavity09	*C-XSC*	Sparse, $K = 0$	0.138	0.284
		Sparse, $K = 1$	0.009	0.047
		Sparse, $K = 2$	0.032	0.088
	CSparse	Sparse, $K = 0$	-	-
		Sparse, $K = 1$	0.005	-
		Sparse, $K = 2$	-	-
	Intlab	Sparse, $K = 0$	-	-
		Sparse, $K = 1$	0.008	0.044
		Sparse, $K = 2$	-	-
af23560	*C-XSC*	Sparse, $K = 0$	1.33	2.69
		Sparse, $K = 1$	0.13	0.59
		Sparse, $K = 2$	0.35	0.92
	CSparse	Sparse, $K = 0$	-	-
		Sparse, $K = 1$	0.09	-
		Sparse, $K = 2$	-	-
	Intlab	Sparse, $K = 0$	-	-
		Sparse, $K = 1$	0.13	0.64
		Sparse, $K = 2$	-	-
conf5.4-00l8x8-0500	*C-XSC*	Sparse, $K = 0$	31.04	72.77
		Sparse, $K = 1$	1.14	14.47
		Sparse, $K = 2$	7.18	22.13
	CXSparse	Sparse, $K = 0$	-	-
		Sparse, $K = 1$	0.92	-
		Sparse, $K = 2$	-	-
	Intlab	Sparse, $K = 0$	-	-
		Sparse, $K = 1$	1.11	10.80
		Sparse, $K = 2$	-	-

Table 4: Comparison of sparse matrix-matrix products, time in seconds

such as a LU-decomposition realised in the C-XSC library. Such specialised implementations will be added in the near future as an interface to the fast algorithms of the CXSparse library, allowing to utilize its functionality for point problems. Until then, the user can implement his own efficient sparse algorithms through directly accessing the basic data structure of the new types.

References

[1] Weblink to C-XSC

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

[2] Weblink to the Matrix Market

`http://math.nist.gov/MatrixMarket/`

[3] Weblink to CXSparse

`http://www.cise.ufl.edu/research/sparse/CXSparse/`

[4] T. A. Davis: Direct Methods for Sparse Linear Systems, SIAM, Philadelphia, Sept. 2006. Part of the SIAM Book Series on the Fundamentals of Algorithms.

[5] J.R. Gilbert, C. Moler, R. Schreiber: Sparse matrices in Matlab: Design and implementation, 2004:

`http://www.mathworks.com/access/helpdesk/help/pdf_doc/otherdocs/simax.pdf`

[6] Hofschuster, W.; Krämer, W.: C-XSC 2.0: A C++ Library for Extended Scientific Computing. Numerical Software with Result Verification, Lecture Notes in Computer Science, Volume 2991/2004, Springer-Verlag, Heidelberg, pp. 15 - 35, 2004.

[7] Klatte, Kulisch, Wiethoff, Lawo, Rauch: C-XSC - A C++ Class Library for Extended Scientific Computing. Springer-Verlag, Heidelberg, 1993.

[8] Krämer, W., Zimmer, M.: Fast (Parallel) Dense Linear System Solvers in C-XSC Using Error Free Transformations and BLAS. Lecture Notes in Computer Science LNCS 5492, pp. 230–249, Springer, 2009.

[9] R. Hammer and M. Hocks and U. Kulisch and D. Ratz: Numerical Toolbox for Verified Computing I: Basic Numerical Problems. Springer Verlag, 1993

[10] Kulisch, U.; Miranker, W.: The arithmetic of the digital computer: A new approach. SIAM Rev., 28(1):1-40, 1986.

[11] Kulisch, U.: Die fünfte Gleitkommaoperation für Top-Performance Computer. Berichte aus dem Forschungsschwerpunkt Computerarithmetik, Intervallrechnung und numerische Algorithmen mit Ergebnisverifikation, 1997.

[12] Ogita, T., Rump, S.M., Oishi, S.: Accurate sum and dot product. SIAM Journal on Scientific Computing, 26:6, 2005.

[13] Oishi, S., Tanabe, K., Ogita, T., Rump, S.M., Yamanaka, N.: A Parallel Algorithm of Accurate Dot Product. Submitted for publication, 2007.

[14] Rump, S.M.: Intlab - Interval Laboratory. Developments in Reliable Computing, pp. 77-104, 1999.

[15] Zimmer, Michael: Laufzeiteffiziente, parallele Löser für lineare Intervallgleichungssysteme in C-XSC. Master Thesis, University of Wuppertal, 2007.

[16] Zimmer, M., Krämer, W., Hofschuster, W.: Using C-XSC in High Performance Computing. Preprint BUW/WRSWT 2009/?, University of Wuppertal, 2009

[17] Zimmer, M., Krämer, W., Bohlender, G., Hofschuster, W.: Extension of the C-XSC Library With Scalar Products With Selectable Accuracy. ?, 2009