C++ bindings to external software libraries with examples from BLAS, LAPACK, UMFPACK, and MUMPS

KARL MEERBERGEN
K.U. Leuven (Computer Science)
and
KREŠIMIR FRESL
University of Zagreb (Civil Engineering)
and
TOON KNAPEN

FORTRAN and C software packages are often used in generic C++ software. Calling non-generic functions in generic code is not straightforward. The bindings in this paper help the C++ programmer using external software with a small effort. The bindings provide a mechanism to keep external software interfaces and specific vector and matrix containers orthogonal. We show examples using BLAS, LAPACK, UMFPACK, and MUMPS functions and subroutines.

Categories and Subject Descriptors: G.4 [**Mathematics of Computing**]: Mathematical software

General Terms: Documentation, Languages

Additional Key Words and Phrases: Bindings, BLAS, C++, LAPACK, traits

1. INTRODUCTION

Scientific software is more and more frequently written in C++. Improvements in the compiler have led to fairly efficient software. See for example Blitz++ [Veldhuizen 2001] and the Matrix Template Library [MTL4 2007] [Gottschling et al. 2007]. Some packages are a user friendly interface to the BLAS, see e.g. FLENS [Lehn 2008].

Generic programming has improved the reusability of software; see for example the Standard Template Library (STL) [Meyers 2001], the Boost project [Boost b] and the related Boost.Sandbox project [Boost c]. Improvements in the compiler have made generic programming a useful tool for numerical software. From the point of view of the user, generic programming can be seen as a form of overloading.

This paper presents research results of the Belgian Network DYSCO (Dynamical Systems, Control, and Optimization), funded by the Interuniversity Attraction Poles Programme, initiated by the Belgian State, Science Policy Office. The scientific responsibility rests with its author(s).

The difference with overloading is that function arguments are abstract types that satisfy some constraints, called concepts. We will discuss this later. An overloaded function is for example

```
void foo( float e ) { ... }
void foo( double e ) { ... }
void foo( std::complex<float> e ) { ... }
void foo( std::complex<double> e ) { ... }
```

while a generic function is

```
template <typename T>
void bar( T e ) {...}
```

where T is a generic type. The conditions on which types can be used are called concepts. For example, we could impose that T must satisfy the Float concept, i.e. a concept for floating point numbers having addition, subtraction, etc. Note that C++ does not require the definition of concepts, but it is considered good practice to do so.

Scientific programmers using C++ also want to use FORTRAN and C codes that have been available for a long time or are still under development. Such packages are LAPACK [Anderson et al. 1995], sparse direct linear system solvers, including SuperLU [Demmel et al. 1999] and UMFPACK [Davis 1995], both written in C, or MUMPS [Agullo et al. 2006] [Amestoy et al. 2006], written in Fortran 90/95. The programming effort for rewriting these codes in C++ is so high that it makes more sense to link them into C++ code. Another argument for linking with external software is performance: it is hard to beat the performance of hand-tuned software, such as the BLAS3 [Dongarra et al. 1990b], see e.g. ATLAS [ATLAS], the GOTO BLAS [Goto and van de Geijn 2008a] [Goto and van de Geijn 2008b], GEMM BLAS [Kågström et al. 1995], and the vendor tuned BLAS implementations, by compiled high-level languages, although significant steps are taken in developing efficient software using generic programming [Veldhuizen 2001] [Gottschling et al. 2007].

Scientific software written in C++ use one of the many packages available for vector and matrix operations, e.g. MTL[MTL4 2007], uBLAS[Boost a], Blitz[Veldhuizen 2001], FLENS [Lehn 2008], GLAS[GLAS 2005] to cite a few. The package FLENS is an interface to BLAS and LAPACK, whereas the other packages mainly use C++ code to implement the evaluation of mathematical expressions. Unfortunately, there is no standard vector and matrix package for C++ as there is the BLAS for FORTRAN, or the vector and matrix functionality in Fortran 90. Complex scientific codes make use of existing software such as LAPACK, BLAS, UMFPACK, MUMPS, and Metis. None of the mentioned C++ packages contains user friendly interfaces to all of these software packages. Linking generic C++ software with external packages can be tedious: C and FORTRAN software do not overload function names for different types of arguments; an overloaded function must be added in the C++ interface for the different argument types. Often, the number of arguments of FORTRAN routines is quite high. The arguments should be correctly computed, including the vector stride, the matrix leading dimensions and storage pointers. Also, FORTRAN names may require the addition of an underscore depending on the FORTRAN and C++ compilers used. These 'details' distract the

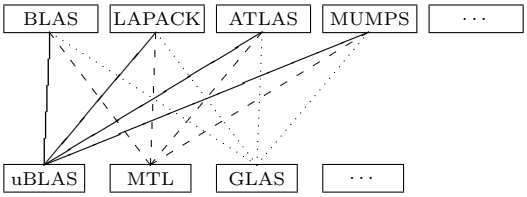

Fig. 1. Traditional interfaces between software

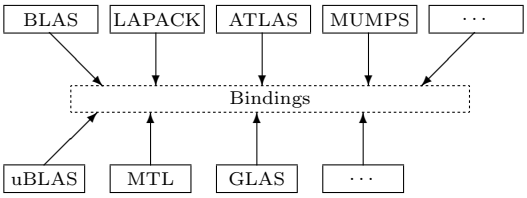

Fig. 2. Concept of bindings as a generic layer between linear algebra algorithms and vector and matrix software

programmer from his ultimate goal and may introduce programming errors. An important advantage of bindings software are additional tests on the input arguments, which makes the code more robust. These aspects will be illustrated in §§3.2–3.5.

In the traditional approach, an interface is developed for each C++ linear algebra package and for each external linear algebra package, which leads to a multiplicity of code interfaces. This is illustrated by Figure 1. In this paper, we adopt the approach of orthogonality between algorithms and data. In the Standard Template Library [Musser and Saini 1996], orthogonality is created by the introduction of iterators. We define a collection of traits classes [Myers 1995], that create a similar orthogonality. The traits classes provide all the necessary data to the external software. For example, the vector traits provide a pointer (or address), size and stride, which are used by e.g. the BLAS function `ddot`. Each traits class is specialized for C++ vector and matrix packages, i.e. the specialization is a specific implementation of obtaining size, stride and pointer for example. Figure 2 illustrates this philosophy. Note the difference with Figure 1.

The novelty of the bindings mainly lies in the orthogonality principle. The concept of bindings and traits classes is similar to the orthogonality between containers and algorithms. We therefore call packages such as MTL, uBLAS, FLENS, Blitz++ and GLAS container packages and we call BLAS, LAPACK, MUMPS, and UMFPACK algorithm packages. In addition, the traits classes are minimal in that they just contain the minimum information to interface software packages. A minimum effort to use the traits consists of specializing traits classes for the container packages and writing bindings for the algorithm packages. In other words, with a bindings layer for MUMPS we can use MUMPS for uBLAS, and GLAS, e.g. without having to write any code based on these container packages. Similarly, specializing the traits for MTL allows us to use LAPACK, BLAS, etc. without any additional effort. The design of the traits classes is kept minimal in order to reduce

the implementation effort.

An alternative approach would be to only provide interfaces between a reference container package, e.g. MTL, with all algorithm packages and then write a layer between the other container packages and the reference. Usually, container packages have too many functionalities, just for binding algorithms. More importantly, many container packages are still under development, which makes bindings potentially unstable. The uBLAS traits specializations, for example, have been modified a couple of times since the creation of the traits a couple of years ago, but the traits interface remained stable. The concepts of one of the more stable and better designed packages, MTL4, could be used as a basis to develop bindings. However, MTL4 was still under development when the Boost.Bindings were developed. In addition, there is no guarantee that concepts will not change in MTL, which could lead to potentially unstable bindings.

In this way, one could e.g. write a generic function

```
template <typename T, typename X, typename Y>
void axpy(T alpha, const X& x, Y& y) ;
```

which computes $y = y+\alpha x$ where x and y are vectors and α is a scalar using a BLAS function. This is a generic function, where we impose conceptual conditions on the arguments. (In this case, we require X and Y to satisfy the concept BindableVector, which will be defined below.) In order to implement the generic axpy function, an overloaded function axpy will be defined for the different flavours of the BLAS routines, e.g. one function for **float**, **double**, complex<**float**> and complex<**double**> respectively. We will give more detailed examples below.

The storage of (sparse) matrices is not unique. Algorithm packages requires matrices to be available in a specific format. Transformation from the one to the other format are not provided by the bindings themselves. Many container packages provide such tools. We would also like to stress that we cannot use bindings or even specialize the traits if the container's data structure does not match the data structure that linear algebra routines expect — e.g. row major vs. column major matrix layout with bindings to FORTRAN 77 packages, or restrictions to sparse matrix formats.

Here is the plan of the paper. In §2, we introduce the traits classes and we give examples for `std::vector` and `ublas::matrix`. We give examples of bindings for some BLAS, LAPACK, MUMPS, and UMFPACK subprograms and functions in §3.2, §3.3, §3.4, and §3.5 respectively. The traits and BLAS, LAPACK, UMFPACK, and MUMPS bindings are part of the Boost.Sandbox repository [Boost c]. They reside in the namespace boost::numeric::bindings. For the remainder of the paper, we will use the namespace alias bindings for boost::numeric::bindings.

2. CONTAINER TRAITS CLASSES

The bindings traits classes are the link between the interface of algorithm packages with container packages. In this section, we give a detailed overview of the functionalities of the traits classes and their implementations.

2.1 Dense vector traits

The vector traits class is defined in boost/numeric/bindings/traits/vector_traits.hpp. It provides a common interface for extracting the following information from a vector container:

—the data pointer (or address),
—the vector size (i.e. the number of elements in the vector),
—the vector stride (i.e. the distance of the pointers of two consecutive elements),
—the value type of the data and
—the pointer type of the data (which is normally, value_type* or **const** value_type*).

The traits class itself is valid when it has the following signature:

```
template <typename V>
struct vector_traits {
  typedef ... value_type ;
  typedef ... pointer ;
  static std::ptrdiff_t size( V& v ) { ... }
  static std::ptrdiff_t stride( V& v ) { ... }
  static pointer storage( V& v ) { ... }
} ;
```

where V is the vector container type. Strictly speaking, value_type is not required, since it can be derived from pointer. However, it is a useful type for selecting the appropriate function in bindings of FORTRAN subprograms and C functions where value type overloading is not possible. We now define the concept BindableVector: a type V that has a valid specialization of vector_traits, is a model of the concept BindableVector.

Sometimes code is cleaner when free functions are used that return the value with the appropriate type. The following free functions are provided:

—bindings::traits::vector_size(v) is a short cut for bindings::traits::vector_traits<V>::size(v),
—bindings::traits::vector_stride(v) is a short cut for bindings::traits::vector_traits<V>::stride(v),
—bindings::traits::vector_storage(v) is a short cut for bindings::traits::vector_traits<V>::storage(v),

For the data types (value_type and pointer), we have to use the traits class. For each type of vector V, a specialization of vector_traits is required, except, when the default implementation applies.

It is important to note that the result type of the functions vector_size and vector_stride is std::ptrdiff_t. We could have provided a size_type, stride_type, etc. for all functions, but this would only increase the number of types in the traits classes. The type std::ptrdiff_t is an integral type that may be compiler and platform dependent. The integral type std::ptrdiff_t covers the entire address space of the machine, and thus sizes of any container type are covered. FORTRAN based vector and matrix algorithms always use **int** as size type in contrast to STL and other C++ packages where **unsigned int** or std::size_t is used. These days, **long int** is also used for supporting large containers on 64-bit machines. As a result, integral casts are unavoidable, but they should never pose a problem in practice: suppose that we call the LAPACK routine DGESV using the bindings. LAPACK assumes **int** as size

type. If the range of **int** would be too small for the container, the corresponding LAPACK routine cannot be used anyway.

The following listing shows the default traits implementation. The meta-function [Abrahams and Gurtovoy 2004] generate_const copies the keyword **const** from V to pointer. The idea is that if V is a const type, the pointer is **const** V::value_type* instead of V::value_type*.

```
template <typename V, typename T = typename V::value_type >
struct default_vector_traits {
  typedef T value_type;
  typedef typename detail::generate_const<V,value_type>::type* pointer;

  static pointer storage (V& v) { return &v[0]; }
  static std::ptrdiff_t size (V& v) { return static_cast<std::ptrdiff_t>(v.size()); }
  static std::ptrdiff_t stride (V&) { return 1; }
};
```

The traits class has to be specialized for any V and **const** V. Usually, the implementation for both cases is the same, and therefore, we have provided a mechanism to do the specialization at once. Instead of specializing vector_traits, we can specialize the following class:

```
template <typename VR, typename V>
struct vector_detail_traits
: default_vector_traits< V >
{} ;
```

where V is the vector container and VR is an identifier: V is VR or **const** VR. The class vector_detail_traits is specialized for VR only. The vector_traits class is defined as follows:

```
template <typename V>
struct vector_traits
: vector_detail_traits< typename boost::remove_const<V>::type, V >
{};
```

As an illustration, we give the specialization for std::vector:

```
template <typename T, typename Alloc, typename V>
struct vector_detail_traits<std::vector<T, Alloc>, V>
: default_vector_traits< V, T >
{
  typedef V vector_type;
  typedef typename default_vector_traits< V, T >::pointer pointer;

  static pointer storage (vector_type& v) { return &v.front(); }
};
```

The repository also has specializations for uBLAS [Boost b] vector expressions. The package GLAS [GLAS 2005], which is still under development, contains the vector traits specializations for its vector expressions.

Note that the introduction of the concept BindableVector is strictly speaking not required by the C++ compiler. The container packages themselves do not have

to support concepts. The only condition is that the traits classes are correctly specialized.

2.2 Dense and banded matrix traits

Whereas the storage of vectors is very natural in most languages since they are just arrays, this is not so for matrices. In many software packages, matrices are also stored in an array, column by column (column_major) or row by row (row_major). Usually, FORTRAN codes assume column-wise storage where C codes often adopt a row-wise storage.

Banded matrices are matrices that only have nonzero elements in a band along the main diagonal. The lower half bandwidth l is the number of diagonals below the main diagonal and the upper half bandwidth u is the number of diagonals above the main diagonal. Hence, a diagonal matrix has $u = l = 0$. Banded matrices only store the nonzero bands, i.e. store $(l + u + 1)m$ elements where m is the minimum of the number of rows and the number of columns. This packed format reduces the storage cost.

Symmetric and Hermitian matrices store their data only in the upper or lower triangular parts. The symmetry permits us to compute the other part. If only the lower part is filled, the upper part is usually untouched and not used by the code. This is a waste of memory. The packed format compresses rows or columns so that the unused part is not allocated. Therefore, we distinguish between symmetric and symmetric packed matrices.

The matrix bindings are organized in a similar way as the vector bindings.

```
template <typename M>
struct matrix_traits : matrix_detail_traits< typename boost::remove_const<M>::type, M>
```

The following information can be retrieved from valid matrix_traits:

—matrix_structure: this is a type that can take the instances general_t, symmetric_t, symmetric_packed_t, hermitian_t, hermitian_packed_t, banded_t, and unknown_structure_t.

—ordering_type: the orientation of the storage, i.e. row_major_t (typically for C-codes), column_major_t (typically for FORTRAN codes).

—the value_type and pointer types.

—the uplo_type is only used for Hermitian or symmetric matrices to indicate whether the upper or lower triangular part of the matrix is stored; the following values can be used: upper_t and lower_t;

When matrix_traits is valid, we also have free functions for computing the following data:

—matrix_storage(m) is a short cut for matrix_traits<M>::storage(m) with return type matrix_traits<M>::pointer; the function returns the storage pointer (or address) to the matrix values;

—matrix_num_rows(m) is a short cut for matrix_traits<M>::num_rows(m) with return type std::ptrdiff_t; the function returns the number of rows of m;

—matrix_num_columns(m) is a short cut for matrix_traits<M>::num_columns(m) with return type std::ptrdiff_t; the function returns the number of columns of m;

—leading_dimension(m) is a short cut for matrix_traits<M>::leading_dimension(m) with return type std::ptrdiff_t; the function returns the leading row dimension (column major) or column dimension (row major);

For banded matrices, we also have the following functions:

—matrix_upper_bandwidth(m) is a short cut for matrix_traits<M>::upper_bandwidth(m) with return type std::ptrdiff_t; the function computes the upper bandwidth without the main diagonal.

—matrix_lower_bandwidth(m) is a short cut for matrix_traits<M>::lower_bandwidth(m) with return type std::ptrdiff_t; the function computes the lower bandwidth without the main diagonal.

The matrix traits class is defined in boost/numeric/bindings/traits/matrix_traits.hpp. Similar to the vector case, we introduce the concept BindableMatrix: we say that M is a model of BindableMatrix if and only if the corresponding matrix_traits<M> is valid.

The repository also has specializations for uBLAS [Boost b] matrix expressions. The package GLAS [GLAS 2005], which is still under development, contains the matrix traits specializations for its matrix expressions.

2.3 Sparse matrix traits

Sparse matrices often take a sparse format, i.e. only the structurally non-zero elements are stored. This implies that next to the values, also the associated row and column indices need to be stored. We consider two storage formats. The coordinate format stores for each non-zero entry, the row index, the column index and the associated numerical value. This data is stored in three separate arrays. The first index array contains the row indices, and the second array the column indices.

The compressed sparse column (row) storage stores a sparse matrix column by column (row by row). For a column major matrix, the first index array contains the start of the columns, while the second index array contains the row indices. For a row major matrix, the first index array contains the start of the rows, while the second index array contains the column indices. The third array contains the numerical values. The advantage of the compressed storage is a reduction of the storage cost of the indices. In addition, it is easy to identify the beginning and the end of each column (row).

For example, the sparse matrix

$$\begin{bmatrix} 1. & 3. & & & \\ & 2. & & 4. & \\ 1. & 7. & & & 5. \\ 1. & 3. & & 4. & \\ & & 2. & & \end{bmatrix}$$

has the following row major coordinate data:

row (index 1) :	0	0	1	1	2	2	2	3	3	3	4
column (index 2) :	0	1	1	3	0	1	4	0	1	3	2
values :	1.	3.	2.	4.	1.	7.	5.	1.	3.	4.	2.

The column major compressed data are as follows:

column pointers (index 1) :	0	3	7	8	10	11					
row (index 2) :	0	2	3	0	1	2	3	4	1	3	2
values :	1.	1.	1.	3.	2.	7.	3.	2.	4.	4.	5.

Note that the first index array contains 6 elements, while the second array contains 11 elements. The elements i and $i+1$ in the first index array point to the beginning and end of the ith column.

Sparse storage is often experienced as being complicated. The fact that different formats are used creates confusion and makes programming hard. The bindings do not contain algorithms that transform one data format to another. It is assumed that the user of external software packages uses the containers that provide the right storage format.

The sparse matrix traits class is defined in boost/numeric/bindings/traits/sparse_traits.hpp. It is organized in a similar way as the dense matrix traits. The following types are defined in a valid sparse_matrix_traits:

—matrix_structure: this can take the values general_t, symmetric_t, symmetric_packed_t, hermitian_t, hermitian_packed_t, banded_t, and unknown_structure_t as for the dense case.

—storage_format: this can take the values compressed_t for matrices in Compressed Sparse Storage format, and coordinate_t for matrices in Coordinate format.

—value_type,

—value_pointer: this is usually, value_type* or **const** value_type*.

—index_pointer: this is usually **int*** or **const int***.

—ordering_type: this can take the values row_major_t and column_major_t.

The traits class also has the static constant index_base that indicates whether the indices are stored in base 0 or 1. In FORTRAN codes, we typically have base 1, since indices start counting from 1. In C-codes, the index base usually is 0.

When sparse_matrix_traits is valid, the following free functions extract the data for sparse matrix routines:

—spmatrix_index1_storage(m) returns the pointer to the first index array;

—spmatrix_index2_storage(m) returns the pointer to the second index array;

—spmatrix_value_storage(m) returns the pointer to the numerical values array;

—spmatrix_num_rows(m) returns the number of rows of m,

—spmatrix_num_columns(m) returns the number of columns of m,

—spmatrix_num_nonzeros(m) returns the number of nonzero elements.

Similar to the vector case, the concept BindableSparseMatrix is satisfied when the type's specialization of the sparse_matrix_traits is valid.

The repository also has specializations for uBLAS [Boost b] sparse matrix expressions. The package GLAS [GLAS 2005], which is still under development, contains the sparse matrix traits specializations for its sparse matrix expressions.

3. ALGORITHM BINDINGS

In this section, we explain how the bindings can be used to interface external software. First, we describe a number of tools to implement bindings. Then we give two dense matrix/vector examples (BLAS and LAPACK) and two sparse matrix examples (UMFPACK and MUMPS).

3.1 Tools

The file boost/numeric/bindings/traits/matrix_traits.hpp contains meta-functions that map the types from the matrix_traits to characters that are used by BLAS and LAPACK subprograms. For example, the function

```
template <typename SymmM>
inline char matrix_uplo_tag (SymmM&) {
  return 'U' or 'L' ;
}
```

returns a character that indicates whether a symmetric matrix is stored in the upper or lower triangular part. The choice of 'U' and 'L' is based on the matrix uplo_type.

Complex numbers are a built-in type in FORTRAN. We have defined equivalent types fcomplex_t for the FORTRAN type COMPLEX and dcomplex_t for the (non standard) type DOUBLE COMPLEX. The definitions can be found in the file boost/numeric/bindings/traits/type.h. We provide the function complex_ptr that transforms a pointer of type complex<**float**>* to fcomplex_t* and from complex<**double**>* to dcomplex_t* without error or warning messages. It is a *clean* way to map the C++ complex types to FORTRAN complex types. It is assumed, however, that complex numbers have the same layout in FORTRAN and C/C++, more precisely as follows:

```
typedef
union {
  float cmplx[2] ;
  double align_struct_ ;
} fcomplex_t ;

typedef
struct {
  double cmplx[2] ;
} dcomplex_t ;
```

If this is not the case, FORTRAN subprograms cannot be used in C++ code using complex<**float**> and complex<**double**>. To our knowledge, all FORTRAN and C++ compilers have compatible complex number representations.

Symbol names are made in a different way by the FORTRAN and C (C++) compilers. The naming convention is platform dependent, so we must map C/C++ names to FORTRAN symbol names using a platform dependent mechanism. The file boost/numeric/bindings/traits/fortran.h contains the macro FORTRAN_ID which transforms a FORTRAN subprogram name to the symbol name in the object file. Typically, the names are lowercase and sometimes an underscore is added to the end. This macro decides whether the underscore is added or not. Note that

this does not work when the FORTRAN symbol contains an underscore. In this case, it is possible two underscores have to be added. This usually does not pose any problems for FORTRAN 77 software since underscores are not used in names of subprograms. For Fortran 90 software, underscores are often used, which might pose a problem in this case. The user can define the preprocessor symbol BIND_FORTRAN_LOWERCASE_UNDERSCORE to indicate that an underscore should be added or BIND_FORTRAN_LOWERCASE to indicate the underscore should not be added. If the user does not define any of those, default settings based on the compiler are used. Older Windows compilers mapped FORTRAN symbols to upper case, but the more recent Visual Studio compilers no longer do this.

3.2 BLAS bindings

The BLAS are the Basic Linear Algebra Subroutines [Lawson et al. 1979] [Dongarra et al. 1988b] [Dongarra et al. 1988a] [Dongarra et al. 1990b] [Dongarra et al. 1990a], whose reference implementation is available through Netlib[1]. The BLAS are subdivided in three levels: level one contains vector operations, level two matrix-vector operations and level three, matrix-matrix operations. Platform specific optimized BLAS libraries are available, see e.g. the ATLAS library [ATLAS], the GOTO BLAS [Goto and van de Geijn 2008a] [Goto and van de Geijn 2008b], GEMM BLAS [Kågström et al. 1995].

The BLAS bindings in Boost.Sandbox contain interfaces to some BLAS functions. The community still contributes to the bindings. The interfaces check the sizes of the input arguments using the assert command, which is only compiled when the NDEBUG compile flag is not set. If the sizes do not match, there is a bug in the function call. Note that exceptions could be used instead of assert, but this decreases performance of codes. As such, assert is the more accepted way to detect interface violations in scientific C++ codes. The interfaces are contained in three files: blas1.hpp, blas2.hpp, and blas3.hpp in the directory boost/numeric/bindings/blas. The BLAS bindings reside in the namespace boost::numeric::bindings::blas.

The BLAS provide functions for vectors and matrices with value_type **float**, **double**, complex<**float**>, and complex<**double**>. All matrix containers must have ordering_type column_major_t, since the (FORTRAN) BLAS assume column major matrices.

The bindings are illustrated in Figure 3 for the BLAS subprograms DCOPY, DSCAL, and DAXPY for objects of type std::vector<**double**>. Note the inclusion of the header files for the bindings of the BLAS-1 subprograms and specialization of vector_traits for std::vector.

We now discuss the implementation of the binding for axpy. First, we define the FORTRAN symbol depending on whether an underscore is required or not. For axpy, these are the following lines from the file blas_names.h

```
#define BLAS_SAXPY FORTRAN_ID( saxpy )
#define BLAS_DAXPY FORTRAN_ID( daxpy )
#define BLAS_CAXPY FORTRAN_ID( caxpy )
#define BLAS_ZAXPY FORTRAN_ID( zaxpy )
```

Next, we define generic function names in the file blas1_overloads.hpp by overloading.

[1] http://www.netlib.org

```
#include <boost/numeric/bindings/traits/std_vector.hpp>
#include <boost/numeric/bindings/blas/blas1.hpp>

int main() {
  std::vector< double > x( 10 ), y( 10 ) ;

  // Fill the vector x
  ...

  bindings::blas::copy( x, y ) ;
  bindings::blas::scal( 2.0, y ) ;
  bindings::blas::axpy( -3.0, x, y ) ;

  return 0 ;
}
```

Fig. 3. Example for BLAS-1 bindings and std::vector bindings traits

These functions are contained in the namespace boost::numeric::bindings::blas::detail.

```
inline void axpy( const int& n, const float& alpha, const float* x
                , const int& incx, float* y, const int& incy)
{
  BLAS_SAXPY( &n, &alpha, x, &incx, y, &incy ) ;
}
inline void axpy( const int& n, const double& alpha, const double* x
                , const int& incx, double* y, const int& incy)
{
  BLAS_DAXPY( &n, &alpha, x, &incx, y, &incy ) ;
}
inline void axpy( const int& n, const complex_f& alpha, const complex_f* x
                , const int& incx, complex_f* y, const int& incy)
{
  BLAS_CAXPY( &n, complex_ptr( &alpha ), complex_ptr( x ), &incx
            , complex_ptr( y ), &incy ) ;
}
inline void axpy(const int& n, const complex_d& alpha, const complex_d* x
                , const int& incx, complex_d* y, const int& incy)
{
  BLAS_ZAXPY( &n, complex_ptr( &alpha ), complex_ptr( x ), &incx
            , complex_ptr( y ), &incy ) ;
}
```

It is important to note that the overloaded function axpy only exists in the four flavours (or value types) for which BLAS routines are developed. For other value types, BLAS calls cannot be used. Finally, the vector_traits in blas1.hpp are used for a nicer interface:

```
template < typename value_type, typename vector_type_x, typename vector_type_y >
void axpy(const value_type& alpha, const vector_type_x &x, vector_type_y &y )
{
    BOOST_STATIC_ASSERT( ( is_same< value_type,
```

```
        typename traits::vector_traits< vector_type_x >::value_type >::value ) ) ;
    BOOST_STATIC_ASSERT( ( is_same< value_type,
        typename traits::vector_traits< vector_type_y >::value_type >::value ) ) ;

    assert( traits::vector_size( x ) == traits::vector_size( y ) ) ;

    const int n = traits::vector_size( x ) ;
    const int stride_x = traits::vector_stride( x ) ;
    const int stride_y = traits::vector_stride( y ) ;
    const value_type *x_ptr = traits::vector_storage( x ) ;
    value_type *y_ptr = traits::vector_storage( y ) ;

    detail::axpy( n, alpha, x_ptr, stride_x, y_ptr, stride_y ) ;
}
```

3.3 LAPACK bindings

Software for dense and banded matrices is collected in LAPACK [Anderson et al. 1995]. It is a collection of FORTRAN routines mainly for solving linear systems, linear least squares problems, and eigenvalue problems, including the singular value decomposition. As for the BLAS, the Boost.Sandbox does currently not contain a full set of interfaces to LAPACK routines, but only very commonly used subprograms. Functions are constantly being added to the library. The LAPACK bindings reside in the namespace boost::numeric::bindings::lapack.

The goal of this section is not to repeat the LAPACK manual: it is assumed that users are familiar with LAPACK. This section only illustrates the philosophy of the interfaces.

Many LAPACK subroutines require auxiliary arrays, which a non-expert user may not wish to allocate for reasons of familiarity. The interface gives the user the possibility to allocate auxiliary vectors using the templated class array that can be found in boost/numeric/bindings/traits/detail/array_impl.hpp, see Figure 4. The class array<T> is a BindableVector. The corresponding vector_traits specialization is in boost/numeric/bindings/traits/detail/array.hpp.

The workspace in many LAPACK functions is pretty clearly defined. In some functions, the user can specify how much workspace is available, allowing for LAPACK to optimize the computations by using blocking. Here is an example to illustrate the different possibilities to handle auxiliary space. We illustrate this using the function gees for computing the Schur decomposition of a matrix. In the following, the variable a contains the matrix, e is the complex vector of eigenvalues and vs the complex matrix of Schur vectors.

—gees(a, e, vs, optimal_workspace()) creates its own workspace whose size is determined by the allocation of dense buffers that can use BLAS3 kernels;

—gees (a, e, vs, minimal_workspace()) creates its own workspace with the minimal size, not taking advantage of the possibility to use BLAS3 kernels;

—gees (a, e, vs, workspace(my_real_workspace)) uses the user created array my_real_workspace as workarray. my_real_workspace should be a BindableVector.

—gees (a, e, vs, workspace(my_real_workspace,my_complex_workspace)) uses the user created real array my_real_workspace and the complex array my_complex_workspace as

```
template <typename T>
class array : private noncopyable {
public:
  array (std::ptrdiff_t n) {
    stg = new (std::nothrow) T[n];
    sz = (stg != 0) ? n : 0;
  }

  ~array() { delete[] stg; }

  std::ptrdiff_t size() const { return sz; }

  bool valid() const { return stg != 0; }

  void resize (std::ptrdiff_t n) {
    delete[] stg;
    stg = new (std::nothrow) T[n];
    sz = (stg != 0) ? n : 0;
  }

  T* storage() { return stg; }
  T const* storage() const { return stg; }

  T& operator[] (std::ptrdiff_t i) { return stg[i]; }
  T const& operator[] (std::ptrdiff_t i) const { return stg[i]; }

private:
  std::ptrdiff_t sz;
  T* stg;
};
```

Fig. 4. Auxiliary array for LAPACK subroutines

workarrays. Both arrays should be BindableVector.

The LAPACK bindings consult matrix_structure to see whether the routine is the right choice. It is also checked whether the matrix arguments are column major. All functions return type is **int**. The return value is the return value of the INFO argument of the corresponding LAPACK subprogram.

Very often, we pass a vector argument to a LAPACK function that is supposed to be a matrix with only one column. This is the case for the solution of linear systems, where the right-hand side of the LAPACK subprogram is a matrix. This suggests that vector containers should also have a specialization of the matrix_traits. For example, the include file ublas_vector2.hpp specializes the matrix_traits for a uBLAS vector.

Figure 5 shows an example of a LAPACK bindings using boost::numeric::ublas::matrix. It is code for the solution of a dense linear system using the LAPACK subprogram DGESV. The matrix a is overwritten. Figure 6 shows another example using GLAS.

```
#include <boost/numeric/bindings/lapack/gesv.hpp>
#include <boost/numeric/bindings/traits/ublas_matrix.hpp>
#include <boost/numeric/bindings/traits/std_vector.hpp>

namespace ublas = boost::numeric::ublas;
namespace lapack = boost::numeric::bindings::lapack;

int main() {
  // system matrix A:
  ublas::matrix<double, ublas::column_major> A(3,3);
  A(0,0) = 1.; A(0,1) = 1.; A(0,2) = 1.;
  A(1,0) = 2.; A(1,1) = 3.; A(1,2) = 1.;
  A(2,0) = 1.; A(2,1) = -1.; A(2,2) = -1.;

  // right-hand side matrix B:
  ublas::matrix<double, ublas::column_major> B(3,1);
  B(0,0) = 4.; B(1,0) = 9.; B(2,0) = -2.;

  // Pivot array
  std::vector<int> pivots( 3 ) ;

  // solve system:
  lapack::gesv (A, pivots, B); // B now contains solution:
}
```

Fig. 5. Example for LAPACK bindings and matrix bindings traits

```
#include <glas/toolbox/bindings/dense_matrix_bindings.hpp>
#include <glas/toolbox/bindings/dense_vector_bindings.hpp>
#include <glas/glas.hpp>
#include <boost/numeric/bindings/lapack/gees.hpp>

int main () {
  int n=100;

  // Define a real n x n matrix
  glas::dense_matrix< double > matrix( n, n ) ;

  // Define a complex n vector
  glas::dense_vector< std::complex<double> > eigval( n ) ;

  // Fill the matrix
  ...

  // Call LAPACK routine DGEES for computing the eigenvalue Schur form.
  // We create workspace for best performance.
  bindings::lapack::gees( matrix, eigval, bindings::lapack::optimal_workspace() ) ;
}
```

Fig. 6. Example for LAPACK bindings and matrix bindings traits

As an illustration of the mechanism of the bindings, we add the bindings for the function gesv for the solution of a linear system. We first define the FORTRAN names and interfaces:

```
#define LAPACK_SGESV FORTRAN_ID( sgesv )
#define LAPACK_DGESV FORTRAN_ID( dgesv )
#define LAPACK_CGESV FORTRAN_ID( cgesv )
#define LAPACK_ZGESV FORTRAN_ID( zgesv )

void LAPACK_SGESV( int const* n, int const* nrhs, float* a, int const* lda
                 , int* ipiv, float* b, int const* ldb, int* info);
void LAPACK_DGESV( int const* n, int const* nrhs, double* a, int const* lda
                 , int* ipiv, double* b, int const* ldb, int* info);
void LAPACK_CGESV( int const* n, int const* nrhs, fcomplex_t* a, int const* lda
                 , int* ipiv, fcomplex_t* b, int const* ldb, int* info);
void LAPACK_ZGESV( int const* n, int const* nrhs, dcomplex_t* a, int const* lda
                 , int* ipiv, dcomplex_t* b, int const* ldb, int* info);
```

Then we write a generic layer independent of the value type but restricted to **float**, **double**, complex<**float**> and complex<**double**>:

```
inline void gesv( int const n, int const nrhs, float* a, int const lda
                , int* ipiv, float* b, int const ldb, int* info) {
  LAPACK_SGESV (&n, &nrhs, a, &lda, ipiv, b, &ldb, info);
}

inline void gesv( int const n, int const nrhs, double* a, int const lda
                , int* ipiv, double* b, int const ldb, int* info) {
  LAPACK_DGESV (&n, &nrhs, a, &lda, ipiv, b, &ldb, info);
}

inline void gesv( int const n, int const nrhs, traits::complex_f* a, int const lda
                , int* ipiv, traits::complex_f* b, int const ldb, int* info) {
  LAPACK_CGESV (&n, &nrhs, traits::complex_ptr (a), &lda, ipiv,
                traits::complex_ptr (b), &ldb, info);
}

inline void gesv( int const n, int const nrhs, traits::complex_d* a, int const lda
                , int* ipiv, traits::complex_d* b, int const ldb, int* info) {
  LAPACK_ZGESV (&n, &nrhs, traits::complex_ptr (a), &lda, ipiv,
                traits::complex_ptr (b), &ldb, info);
}
```

Finally, we write function that take matrix arguments:

```
template <typename MatrA, typename MatrB, typename IVec>
inline int gesv (MatrA& a, IVec& ipiv, MatrB& b) {
  BOOST_STATIC_ASSERT((boost::is_same<
        typename traits::matrix_traits<MatrA>::matrix_structure,
        traits::general_t
      >::value));
  BOOST_STATIC_ASSERT((boost::is_same<
        typename traits::matrix_traits<MatrB>::matrix_structure,
```

```
        traits::general_t
      >::value));

  int const n = traits::matrix_num_rows (a);
  assert (n == traits::matrix_num_colums (a));
  assert (n == traits::matrix_num_rows (b));
  assert (n == traits::vector_size (ipiv));

  int info;
  gesv (n, traits::matrix_num_colums (b),
        traits::matrix_storage (a), traits::leading_dimension (a),
        traits::vector_storage (ipiv), traits::matrix_storage (b),
        traits::leading_dimension (b), &info);
  return info;
}
```

3.4 MUMPS bindings

MUMPS stands for Multifrontal Massively Parallel Solver. The first version was a result from the EU project PARASOL [Amestoy P.R. et al. 2001; Amestoy et al. 2006; Agullo et al. 2006]. The software is developed in Fortran 90 and contains a C interface. The input matrices should be given in coordinate format, i.e. storage_format=coordinate_t and the index numbering should start from one, i.e. sparse_matrix_traits<M>::index_base==1. We refer to the MUMPS Users Guide, distributed with the software [MUMPS 2001].

The C++ bindings contain a generic interface to the respective C structs for the different value types that are available from the MUMPS distribution: **float**, **double**, complex<**float**>, and complex<**double**>. The bindings also contain functions to set the pointers and sizes of the parameters in the C struct using the bindings traits classes. The binding mechanism is similar to BLAS and LAPACK. The templated class mumps<BindableSparseMatrix> inherits from the MUMPS C-struct associated with the value_type of the matrix, i.e;

```
template <typename M>
struct mumps
: detail::mumps_type< typename traits::sparse_matrix_traits<M>::value_type >::type
{} ;
```

where the appropriate C-struct is selected by mumps_type, e.g.

```
template <>
struct mumps_type< float > {
  typedef SMUMPS_STRUC_C type ;
} ;
```

An example is given in Figure 7. The functions mumps::matrix_integer_data, mumps::matrix_value_data, and mumps::rhs_sol_value_data pass the pointers for the integer, value data, and right-hand side and solution vectors respectively before calling the MUMPS driver using mumps::driver. The sparse matrix is the uBLAS coordinate_matrix, which is a sparse matrix in coordinate format. The matrix is stored column-wise. The template argument 1 indicates that row and column numbers start from one, which is required for the Fortran 90 code MUMPS. Finally,

the last argument indicates that the row and column indices are stored in type **int**, which is also a requirement for the Fortran 90 interface. The solve consists of three phases: (1) the analysis phase, which only needs the matrix's integer data, (2) the factorization phase, where also the numerical values are required and (3) the solution phase (or backtransformation), where the right-hand side vector is passed on. The included files contain the specializations of the dense matrix and sparse matrix traits for uBLAS and the MUMPS bindings.

3.5 UMFPACK bindings

UMFPACK is a C implementation of the unsymmetric–pattern multifrontal method for direct solution of systems of linear equations with sparse unsymmetric coefficient matrices. Figure 8 shows an example of use based on the introductory example from [Davis 1995].

Include files contain the traits specialization for uBLAS sparse matrices and dense vectors, and the bindings for UMFPACK. UMFPACK requires that the input matrices are given in compressed sparse column format (CSC). Besides, it assumes that the indices have type **int**, so we must make sure that the sparse matrix container also uses this type. This is the reason why we use ublas::unbounded_array<**int**> as template argument for ublas::compressed_matrix. The template argument 0 indicates that row and column indices begin from zero, as is usual in C. Note that there also is a **long int** version of UMFPACK, but currently bindings provide only **int** interface.

Solution of a linear system in UMFPACK consists of three steps: symbolic analysis of the coefficient matrix, numerical factorization and backward substitution. In more details, function symbolic() pre-orders matrix columns to reduce fill-in, finds the so-called supernodal column elimination tree and post-orders the tree. Symbolic analysis details are returned in object Symbolic of type symbolic_type<> which is passed by reference. In the second step, the function numeric() performs the numerical LU factorization of the coefficient matrix using the symbolic analysis previously computed by symbolic(). The numerical factorization is returned in the object Numeric of type numeric_type<> which is again passed by reference. Finally, the function solve() solves a linear system using the numerical factorization stored in Numeric. In this last step only square systems are handled (if the system matrix is singular, a division by zero will occur and the solution will contain Infs and/or NaNs), but it should be noted that first two steps—symbolic analysis and numerical factorization—can also be performed on rectangular matrices. As short hands, the bindings provide two additional functions: factor() combines symbolic analysis and numerical factorization (that is, the symbolic_type<> object is used only internally), while umf_solve() combines all three solution steps (i.e., both the symbolic_type<> and numeric_type<> objects are used internally). The symbolic_type<> and numeric_type<>'s wrap void pointers which UMFPACK uses to pass around symbolic analysis and numerical factorisation information.

Bindings also provide classes control_type<> and info_type<> which wrap UMFPACK's control and info arrays. Constructor of control_type<> sets UMFPACK's default control parameters, but individual entries can be accessed and changed using **operator**[].

```
#include <boost/numeric/bindings/traits/ublas_sparse.hpp>
#include <boost/numeric/bindings/traits/ublas_vector2.hpp>
#include <boost/numeric/bindings/mumps/mumps_driver.hpp>

int main() {
  namespace ublas = boost::numeric::ublas ;
  namespace mumps = boost::numeric::bindings::mumps ;

  typedef ublas::coordinate_matrix< double, ublas::column_major
                                    , 1, ublas::unbounded_array<int>
                                    > sparse_matrix_type ;

  int n = 5; nnz = 10 ;
  sparse_matrix_type matrix( n, n, nnz ) ;

  // Fill the sparse matrix
  ...

  mumps::mumps< sparse_matrix_type > mumps_solver ;

  // Analysis (Set the pointer and sizes of the integer data of the matrix)
  matrix_integer_data( mumps_solver, matrix ) ;
  mumps_solver.job = 1 ;
  mumps::driver( mumps_solver ) ;

  // Factorization (Set the pointer for the values of the matrix)
  mumps::matrix_value_data( mumps_solver, matrix ) ;
  mumps_solver.job = 2 ;
  mumps::driver( mumps_solver ) ;

  // Set the right-hand side
  ublas::vector<double> v( 10 ) ;

  // Solve (set pointer and size for the right-hand side vector)
  rhs_mumps::sol_value_data( mumps_solver, v ) ;
  mumps_solver.job = 3 ;
  mumps::driver( mumps_solver ) ;

  return 0 ;
}
```

Fig. 7. Example of the use of the MUMPS bindings

4. ORGANIZATION OF THE SOFTWARE

The software is part of the Boost.Sandbox [Boost c]. The software resides in the subdirectory boost/numeric/bindings with the subdirectory traits for the vector, and matrix traits classes, the directory lapack for LAPACK bindings, blas for BLAS bindings, umfpack for UMFPACK bindings, and mumps for MUMPS bindings.

```
#include <boost/numeric/bindings/traits/ublas_vector.hpp>
#include <boost/numeric/bindings/traits/ublas_sparse.hpp>
#include <boost/numeric/bindings/umfpack/umfpack.hpp>

namespace ublas = boost::numeric::ublas;
namespace umf = boost::numeric::bindings::umfpack;

int main() {
  ublas::compressed_matrix<double, ublas::column_major, 0,
    ublas::unbounded_array<int>, ublas::unbounded_array<double> > A (5,5,12);
  ublas::vector<double> B (5), X (5);

  A(0,0) = 2.; A(0,1) = 3;
  A(1,0) = 3.; A(1,2) = 4.; A(1,4) = 6;
  A(2,1) = -1.; A(2,2) = -3.; A(2,3) = 2.;
  A(3,2) = 1.;
  A(4,1) = 4.; A(4,2) = 2.; A(4,4) = 1.;

  B(0) = 8.; B(1) = 45.; B(2) = -3.; B(3) = 3.; B(4) = 19.;

  umf::symbolic_type<double> Symbolic;
  umf::numeric_type<double> Numeric;

  umf::symbolic (A, Symbolic);
  umf::numeric (A, Symbolic, Numeric);
  umf::solve (A, X, B, Numeric);
}
```

Fig. 8. Example for UMFPACK bindings and matrix bindings traits

5. CONCLUSIONS

We presented the traits software from Boost.Sandbox as a generic *minimum* layer between C++ vector and matrix container libraries and external linear algebra algorithms. The layer is minimal in that it only contains those types and functions that are needed to bind existing FORTRAN and C software. We have shown the *orthogonality* between C++ vector and matrix libraries and external linear algebra software with examples. This orthogonality supports independent developments of algorithm bindings and container traits classes. In addition, the traits classes have proven to be stable for many years, helping scientists to link essentially LAPACK and sparse direct solvers in scientific codes.

The new Concept C++ compiler [Gregor et al. 2006] would provide a much cleaner interface to external programs where concepts and concept maps instead of traits and traits specializations are used. This is future work.

ACKNOWLEDGMENTS

The authors are grateful to Peter Gottschling for helpful remarks on the development of MTL.

REFERENCES

Abrahams, D. and Gurtovoy, A. 2004. C++ template metaprogramming: Concepts, tools and techniques from Boost and beyond.

Agullo, E., Guermouche, A., and L'Excellent, J.-Y. 2006. A preliminary out-of-core extension of a parallel multifrontal solver. In *EuroPar'06 Parallel Processing*. 1053–1063.

Amestoy, P., Duff, I., Guermouche, A., and Slavova, T. 2006. A preliminary analysis of the out-of-core solution phase of a parallel multifrontal approach. Presentation on 4th International Workshop on Parallel Matrix Algorithms and Applications (PMAA'06),September 7-9, 2006, IRISA, Rennes, France.

Amestoy P.R., Duff, I., L'Excellent, J.-Y., and Koster, J. 2001. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications 23,* 1, 15–41.

Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Ostrouchov, S., and Sorensen, D. 1995. *LAPACK users' guide*. SIAM, Philadelphia, PA, USA.

ATLAS. Automatically tuned linear algebra software. `http://math-atlas.sourceforge.net/`.

Boost. Basic linear algebra. `http://www.boost.org/doc/libs/1_35_0/libs/numeric/ublas/doc/index.htm`.

Boost. Boost C++ libraries. version 1.34.0. `http://www.boost.org`.

Boost. Boost sandbox C++ libraries. `http://svn.boost.org/trac/boost/wiki/BoostSandbox`.

Davis, T. 1995. Users' guide for the Unsymmetric-pattern MultiFrontal Package (UMFPACK). Technical Report TR-95-004, Computer and Information Sciences Department, University of Florida, Gainesville, FL, USA.

Demmel, J. W., Gilbert, J. R., and Li, X. S. 1999. SuperLU users' guide. Available through `http://www.cs.berkeley.edu/~demmel/SuperLU.html` or `http://www.nersc.gov/~xiaoye/SuperLU/`.

Dongarra, J. J., Du Croz, J., Duff, I. S., and Hammarling, S. 1990a. Algorithm 679: A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw. 16*, 18–28.

Dongarra, J. J., Du Croz, J., Duff, I. S., and Hammarling, S. 1990b. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw. 16*, 1–17.

Dongarra, J. J., Du Croz, J., Hammarling, S., and Hanson, R. J. 1988a. Algorithm 656: An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Softw. 14*, 18–32.

Dongarra, J. J., Du Croz, J., Hammarling, S., and Hanson, R. J. 1988b. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Softw. 14*, 1–17.

GLAS. 2005. Generic linear algebra software. `http://www.cs.kuleuven.be/~karlm/glas`.

Goto, K. and van de Geijn, R. 2008a. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw. 34,* 3, 25.

Goto, K. and van de Geijn, R. 2008b. High-performance implementation of the level-3 blas. *ACM Trans. Math. Softw.*. To appear.

Gottschling, P., Wise, D. S., and Adams, M. D. 2007. Representation-transparent matrix algorithms with scalable performance. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*. ACM Press, New York, NY, USA, 116–125.

Gregor, D., Järv, J., Siek, J., Stroustrup, B., Dos Reis, G., and Lumsdaine, A. 2006. Concepts: linguistic support for generic programming in C++. In *Proceedings of the 2006 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '06)*. ACM Press, New York, NY, USA, 291–310.

Kågström, B., Ling, P., and Van Loan, C. 1995. GEMM-based level 3 BLAS : high-performance model implementations and performance evaluation benchmark. In *Parallel Programming and Applications*. IOS Press, 184–188.

Lawson, C. L., Hanson, R. J., Kincaid, D., and Krogh, F. T. 1979. Basic linear algebra subprograms for FORTRAN usage. *ACM Trans. Math. Softw. 5*, 308–323.

Lehn, M. 2008. Everything you always wanted to know about FLENS but were afraid to ask. http://flens.sourceforge.net/.

Meyers, S. 2001. *Effective STL: 50 specific ways to improve your use of the standard template library.* Addison-Wesley Professional. SBN-13: 9780201749625.

MTL4. 2007. The Matrix Template Library, version 4. `http://www.osl.iu.edu/research/mtl/mtl/4`.

MUMPS. 2001. MUlfrontal Massively Parallel Solver. `http://graal.ens-lyon.fr/MUMPS/`.

Musser, D. R. and Saini, A. 1996. *STL Tutorial and Reference Guide. C++ Programming with the Standard Template Library.* Addison-Wesley, Reading, MA.

Myers, N. 1995. Traits: a new and useful template technique. C++ report. `http://www.cantrip.org/traits.html`.

Veldhuizen, T. 2001. Blitz++ user's guide. `http://www.oonumerics.org/blitz/`.

Received Month Year; revised Month Year; accepted Month Year