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

Karl Meerbergen   *Krešimir Fresl*   *Toon Knapen*

Report TW 506, October 2007

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

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

Karl Meerbergen *Krešimir Fresl* *Toon Knapen*

Report TW 506, October 2007

Department of Computer Science, K.U.Leuven

Abstract

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.

Keywords : Bindings, BLAS, C++, LAPACK, traits
MSC : Primary : 68N01, 65F99

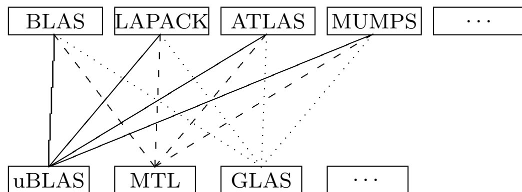

Figure 1: Traditional interfaces between software

1 Introduction

Scientific software is more and more frequently written in C++. It has also led to fairly efficient software thanks to recent improvements in the compiler. See for example the Matrix Template Library [19] [15]. Nevertheless, performance of compiled high-level languages is unable to match the performance of hand-tuned software, such as the BLAS3 [11], see e.g. ATLAS [5], the GOTO BLAS, GEMM BLAS [17], and the vendor tuned BLAS implementations.

Generic programming has improved the reusability of software ; see for example the Boost project [6] and the related Boost Sandbox project [7]. Improvements in the compiler have made generic programming a very useful tool for numerical software.

Scientific programmers using C++ also want to use FORTRAN and C codes that are available for a long time. The programming effort for rewriting these codes in C++ is very high. It therefore makes more sense to link the codes into C++ code. Such packages are LAPACK [4], sparse direct linear system solvers, including SuperLU [9] and UMFPACK [8], both written in C, or MUMPS [1] [2], written in Fortran 90/95. Another argument for linking with external software is performance : the vendor tuned BLAS functions are perhaps the most obvious example.

The context of the paper is the following. Suppose, we have a number of C++ packages for vectors and matrices, e.g. MTL, uBLAS, GLAS, Blitz, and a number of external C or Fortran packages for linear algebra algorithms, LAPACK, BLAS, UMFPACK, MUMPS, which we want to use with the C++ vector and matrix packages in a flexible way. Linking generic C++ software with external packages can be very tedious, since C and FORTRAN software do not overload function names for different types of arguments.

In the traditional approach, an interface is developed for each basic C++ linear algebra package and for each external linear algebra package. This is illustrated by Figure 1. In this paper, we adopt the approach of orthogonality between algorithms and data. In the Standard Template Library [21], orthogonality is created by the introduction of iterators. In the bindings, the orthogonality is created by traits classes that are used by the bindings for external software and specialized for user's vector and matrix classes. The traits classes provide to the external software all data : for example, the vector traits provides 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. In other words, for a new vector or matrix type, the development effort is limited to the specialization of the traits classes. Once the traits classes are specialized, BLAS and LAPACK can be used straightaway. For a new external software package, it is sufficient to provide a layer that uses the bindings. Figure 2 illustrates this philosophy. Note the difference with Figure 1.

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 subpro-

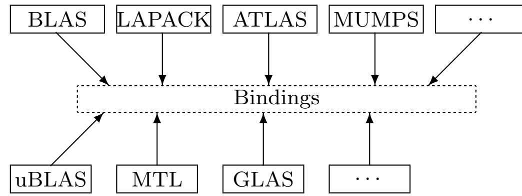

Figure 2: Concept of bindings as a generic layer between linear algebra algorithms and vector and matrix software

grams 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 [7]. They reside in the namespace boost::numeric::bindings. In the following, we use the namespace alias bindings for boost::numeric::bindings.

2 Bindings traits classes

The bindings traits classes are the link between the interface of external software to be linked with and the data containers. In this section, we give a detailed overview of the functionalities of the traits classes and their implementations.

2.1 Dense vector traits

The traits classes provide 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 has the following signature :

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

where V is the vector container type. We call a type V that has a valid specialization of vector_traits, a BindableVector. Sometimes it is easier using free functions 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.

The vector traits class is defined in boost/numeric/bindings/traits/vector_traits.hpp.

The following listing shows the default traits implementation. The meta function 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 int size (V& v) { return static_cast<int>(v.size()); }
  static int 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
: vector_default_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 [6] vector expressions. The package GLAS [14], which is still under development, contains the vector traits specializations for its vector expressions.

It is important to note that the result type of the functions vector_size and vector_stride is **int**, since most external codes use **int** for integers.

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 allows 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. Somehow, 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 the matrix bindings :

- 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 ;

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_size1(m) is a short cut for matrix_traits<M>::size1(m) with return type **int** ; the function returns the number of rows of m ;
- matrix_size2(m) is a short cut for matrix_traits<M>::size2(m) with return type **int** ; the function returns the number of columns of m ;
- matrix_storage_size(m) is a short cut for matrix_traits<M>::storage_size(m) with return type **int** ; the function returns the size of the storage of the matrix taking into account bandedness, symmetry and leading dimension ;
- leading_dimension(m) is a short cut for matrix_traits<M>::leading_dimension(m) with return type **int** ; 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 **int** ; 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 **int** ; 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 call a type BindableMatrix when the matrix_traits provides us the data members and member functions that we need for bindings.

The repository also has specializations for uBLAS [6] matrix expressions. The package GLAS [14], which is still under development, contains the matrix traits specializations for its matrix expressions.

2.3 Sparse matrix traits

Sparse matrices often take the 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. These three data are 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), where one array indicates the beginning of each column (row), and the second index array indicates the row (column) index. The third array are 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 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.

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 :	0	0	1	1	2	2	2	3	3	3	4
column :	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 :

index 1 :	0	3	7	8	10	11					
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 ith and $i+1$st elements in the first index array show the start and the end of the ith column.

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 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.

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_size1(m) returns the number of rows of m,
- spmatrix_size2(m) returns the number of columns of m,
- spmatrix_num_nonzeros(m) returns the number of nonzero elements.

Similar to the vector case, we call a type BindableSparseMatrix when the sparse_matrix_traits provides us the data members and member functions that we need for binding.

The repository also has specializations for uBLAS [6] sparse matrix expressions. The package GLAS [14], which is still under development, contains the sparse matrix traits specializations for its sparse matrix expressions.

3 Software 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 std::complex<**float**>* to fcomplex_t* and from std::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. Since they have the same memory layout, this should not pose a problem.

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 allows to transform 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 [18] [13] [12] [11] [10], 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 operations. Platform specific optimized BLAS libraries are available, see e.g. the ATLAS library [5], the GOTO BLAS, GEMM BLAS [17].

The BLAS bindings in Boost Sandbox contain interfaces to some BLAS functions. Functions are added on request. The interfaces check the input arguments using the assert command, which is only compiled when the NDEBUG compile flag is not set. 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.

[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 ;
}
```

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

The BLAS provide functions for vectors and matrices with value_type **float**, **double**, std::complex<**float**>, and std::complex<**double**>. All matrix containers 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 include files for the bindings of the BLAS-1 subprograms and the include file that contains the specialization of vector_traits for std::vector.

We now discuss the implementation of the binding for axpy. This is useful information for those who want to develop bindings to FORTRAN subprograms. 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. 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 ) ;
}
```

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 [4]. It is a collection of FORTRAN routines mainly for solving linear systems, and eigenvalue problems, including the singular value decomposition. As for the BLAS, the Boost Sandbox does not contain a full set of interfaces to LAPACK routines, but only very commonly used subprograms. On request, more functions are 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 does not wish to allocate for reasons of comfort. The interface allows the user 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 syev for computing the eigenvalues of a symmetric matrix.

- syev ('N', 'U', a, w, optimal_workspace()) creates its own workspace whose size is determined by the allocation of dense buffers that can use BLAS3 kernels ;

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

  ~array() { delete[] stg; }

  int size() const { return sz; }

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

  void resize (int 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[] (int i) { return stg[i]; }
  T const& operator[] (int i) const { return stg[i]; }

private:
  int sz;
  T* stg;
};
```

Figure 4: Auxiliary array for LAPACK subroutines

```
#include <iostream>
#include <boost/numeric/bindings/lapack/gesv.hpp>
#include <boost/numeric/bindings/traits/ublas_matrix.hpp>
#include <boost/numeric/ublas/io.hpp>

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

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.;
  std::cout << "A: " << A << std::endl;

  // 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.;
  std::cout << "B: " << B << std::endl;

  // solve system:
  lapack::gesv (A, B);
  // B now contains solution:
  std::cout << "X: " << B << std::endl;
}
```

Figure 5: Example for LAPACK bindings and matrix bindings traits

- syev ('N', 'U', a, w, minimal_workspace()) creates its own workspace with the minimal size, not taking advantage of the possibility to use BLAS3 kernels ;
- syev ('N', 'U', a, w, workspace(my_real_workspace)) uses the user created array my_real_workspace as workarray. my_real_workspace should be a BindableVector.
- heev ('N', 'U', a, w, workspace(my_real_workspace,my_complex_workspace)) uses the user created real array my_real_workspace and the complex array my_complex_workspace as workarrays. Both arrays should be BindableVector.

The LAPACK bindings verify the 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 on a vector argument to a LAPACK function that is supposed to be a matrix with only one column. This is e.g. 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 where the pivot array is created internally in the function gesv. The matrix a is overwritten. Figure 6 shows another example using GLAS.

```
#define GLAS_COMPLEX
#include <glas/toolbox/bindings/dense_matrix_bindings.hpp>
#include <glas/toolbox/bindings/dense_vector_bindings.hpp>
#include <glas/container/dense_matrix.hpp>
#include <glas/container/dense_vector.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() ) ;

  ...
}
```

Figure 6: Example for LAPACK bindings and matrix bindings traits

3.4 MUMPS bindings

MUMPS stands for Multifrontal Massively Parallel Solver. The first version was a result from the EU project PARASOL [3, 2, 1]. 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 [20].

The C++ interface is a generic interface to the respective C structs for the different value types that are available from the MUMPS distribution: **float**, **double**, std::complex<**float**>, and std::complex<**double**>. The C++ bindings also contain functions to set the pointers and sizes of the parameters in the C struct using the bindings traits classes. An example is given in Figure 7. 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' 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 a multifrontal method. The input matrices are given in compressed sparse column format (CSC). UMFPACK 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 include files contain the traits specialization for uBLAS sparse matrices and dense vectors, and the bindings for UMFPACK. Figure 8 shows an example.

4 Organization of the software

The software is part of the Boost Sandbox [7]. 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.

5 Conclusions

We presented the bindings software from Boost.Sandbox as a generic layer between C++ vector and matrix libraries and external linear algebra software. We have shown the orthogonality between C++ vector and matrix libraries and external linear algebra software with examples.

There is room for further improvements to the traits classes : return type of size(), stride(), etc. should perhaps depend on the traits, rather than being **int**.

The new Concept C++ compiler [16] 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.

```
#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 ;

  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 ;
  driver( mumps_solver ) ;

  // Factorization (Set the pointer for the values of the matrix)
  matrix_value_data( mumps_solver, matrix ) ;
  mumps_solver.job = 2 ;
  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_sol_value_data( mumps_solver, v ) ;
  mumps_solver.job = 3 ;
  mumps::driver( mumps_solver ) ;

  return 0 ;
}
```

Figure 7: Example of the use of the MUMPS bindings

```
#include <iostream>
#include <boost/numeric/bindings/traits/ublas_vector.hpp>
#include <boost/numeric/bindings/traits/ublas_sparse.hpp>
#include <boost/numeric/bindings/umfpack/umfpack.hpp>
#include <boost/numeric/ublas/io.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);

  std::cout << X << std::endl ;
}
```

Figure 8: Example for UMFPACK bindings and matrix bindings traits

Acknowledgement

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).

References

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

[2] P. Amestoy, I.S. Duff, A. Guermouche, and Tz. Slavova. A preliminary analysis of the out-of-core solution phase of a parallel multifrontal approach. September 2006. Presentation on 4th International Workshop on Parallel Matrix Algorithms and Applications (PMAA'06),September 7-9, 2006, IRISA, Rennes, France.

[3] I.S. Amestoy P.R., Duff, J.-Y. L'Excellent, and J. Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM J. Matrix Anal. Applic.*, 23(1):15–41, 2001.

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

[5] ATLAS. Automatically tuned linear algebra software. http://math-atlas.sourceforge.net/.

[6] Boost. Boost C++ libraries. version 1.34.0. http://www.boost.org.

[7] Boost. Boost sandbox C++ libraries. http://svn.boost.org/trac/boost/wiki/BoostSandbox.

[8] T.A. Davis. 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, 1995.

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

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

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

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

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

[14] GLAS. Generic linear algebra software, 2005. `http://www.sourceforge.net/glas`.

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

[16] Douglas Gregor, Jaakko Järv, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: linguistic support for generic programming in C++. In *ACM SIGPLAN Notices*, volume 41, pages 291–310, New York, NY, USA, 2006. ACM Press.

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

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

[19] MTL. The matrix template library, version 2.1.2-22, 2005. http://www.osl.iu.edu/research/mtl/.

[20] MUMPS. Mufrontal massively parallel solver, 2001. http://graal.ens-lyon.fr/MUMPS/.

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