Implementation of a C++ library for data exchange with matlab®

Bachelor Thesis

by
Andrea Arteaga

supervisor
Prof. Dr. R. Hiptmair*

advisor
Roman Andreev†

September 10, 2010

Abstract

MATLAB® stores matrices and others data structures into binary files which we call MAT-files. They represent an efficient and powerful way for data exchange between C++ programs and MATLAB® or other C++ programs. We propose an easy-to-use C++ library for input and output operations to share data in matrix form.

1 Description of the problem

In the field of numerical simulations the need to share data between a C++ algorithm and other programs like MATLAB®, Octave or even other C++ programs is a common problem. An efficient way to do this is through MAT-files, the MATLAB® standard for data saving. This standard has many advantages:

- It supports dense and sparse matrix storage.
- It supports other data structures, such as rectangular tensors, structs, objects and cell arrays.
- The data are stored in a binary format, i.e. data precision is preserved.
- The standard does not impose a limit on the number of data objects.
- The standard allows data compression using zlib.
- The standard is portable.

Nevertheless, there is a lack of *modern*, powerful, object oriented, simple to use C++ libraries for input-output operation with this type of files. MATLAB® does provide within *External interfaces* [2] a C/C++ API for that purpose, but its C based functional programming does not conform to the object-oriented

*Seminar for Applied Mathematics, Swiss Federal Institute of Technology, Rämistrasse 101, Zurich

†Seminar for Applied Mathematics, Swiss Federal Institute of Technology, Rämistrasse 101, Zurich

paradigm. On the web libraries for handling MAT-files are available, which, however, do not meet the requirements above.

Therefore, the library presented here (*the library* in the following) aims to be an object-oriented, portable, easy to learn and to use, fast, free and open-source C++ library. Moreover, it should provide support for linear algebra facilities, such as uBLAS [3] or Eigen [4]. The library has no dependencies other than zlib, but can be used with other linear algebra libraries.

2 The structure of a MAT-file

The document matfile-format [1] describes the format of a MAT-file. It is not our intention to analyze or explain this format, but only to consider the most important implications of its structure for the construction of the library.

Header Every MAT-file begins with a 128-byte header which includes a first 116-byte field of human-readable description, an 8-byte field with subsystem-specific data (which will not be part of our discussions), a 2-byte integer number specifying the version of the file and a 2-byte field named *endianess indicator*. The library allows the user to write a custom description; if the user does not write anything, the library fills the description with default information, like creation data and time, equivalent to the behavior of MATLAB®.

Elements The rest of the file is organized in *elements*: each element consists of an 8-byte tag and a set of bytes that correspond to the data. The tag contains two pieces of information: the first 4 bytes specify the *type*, i.e. how the data must be interpreted; the rest of the tag is a 32-bit integer containing the length of the data in bytes. If the data is 4 bytes or shorter, a short tag can be used which contains in the first 16 bytes the number of bytes (which can be a number between 0 and 4), in the next 16 bytes the type, after which the actual data (4 bytes) follows.

The type can be one of the following:

Value	Type name	Meaning
1	miINT8	signed 8-bit integer
2	miUINT8	unsigned 8-bit integer
3	miINT16	signed 16-bit integer
4	miUINT16	unsigned 16-bit integer
5	miINT32	signed 32-bit integer
6	miUINT32	unsigned 32-bit integer
7	miSINGLE	IEEE 754 single format
9	miDOUBLE	IEEE 754 double format
12	miINT64	signed 64-bit integer
13	miUINT64	unsigned 64-bit integer
14	miMATRIX	MATLAB® array
15	miCOMPRESSED	compressed data

Table 1: Numeric and compound types

Matrix elements The type miMATRIX represents MATLAB® arrays, i.e. dense/sparse matrices, structures, objects, cell arrays, etc. Arrays in MAT-files are composite elements, i.e. the data consists of many sub-elements, each with tag and data. Three sub-elements are common for all arrays, while other sub-elements can be specific for array type. We present only two examples which are the most important ones for the purpose of this work: dense matrices, whose treatment follows here, and sparse matrices, which are discussed on page 5.

2.1 Dense matrices

Dense matrices are MATLAB® arrays with two or more dimensions and a value for each entry. The storage of dense matrices is in column-major order. The following are the sub-elements of dense matrices.

Sub-element name	**Type**	**Number of bytes**
Flags	miUINT32	8
Dimensions	miINT32	$4 \cdot nDim$
Name	miINT8	$nChars$
Real part	Any numeric type	$typeSize \cdot \prod_{i=1}^{nDim} d_i$
Imaginary part (optional)	Any numeric type	$typeSize \cdot \prod_{i=1}^{nDim} d_i$

Table 2: Sub-elements for dense matrices

Here and in the equivalent table for sparse matrices on page 5 we used the following symbols:

nDim The number of dimensions of the matrix – for a common matrix this is equivalent to two.

nChars The number of character of which the matrix name consists.

typeSize Size in bytes of an element of the given numeric type. E.g. for double, this amounts to 8.

$\boldsymbol{d_i}$ The number of entries along the i-th dimension. E.g. for two-dimensions matrices $d_1 = \textit{number of columns}$, $d_2 = \textit{number of rows}$.

In the following we discuss more in-depth each sub-element that we listed:

Flags This sub-element is common to all arrays. The flags specify the class of the array and give some information about the matrix. In case of dense matrices, the class can be one of the following:

The class does not describe the type in which the data is stored but the class of the matrix when loaded into MATLAB®. One could save a matrix using tiny integer types (see **Real part** below), then load it into MATLAB® and treat it as a matrix of double type; so, for example, a result of a division will not be rounded to its integer part.

Further information provided by this sub-element is encoded in three bits:

- **complex**: The bit is set to one if the matrix contains complex elements. If this bit is set, then the matrix will also have the optional sub-element **Imaginary part** (see below).

Value	Class name	Meaning
6	mxDOUBLE_CLASS	Double precision matrix
7	mxSINGLE_CLASS	Single precision matrix
8	mxINT8_CLASS	Signed 8-bit integer matrix
9	mxUINT8_CLASS	Unsigned 8-bit integer matrix
10	mxINT16_CLASS	Signed 16-bit integer matrix
11	mxUINT16_CLASS	Unsigned 16-bit integer matrix
12	mxINT32_CLASS	Signed 32-bit integer matrix
13	mxUINT32_CLASS	Unsigned 32-bit integer matrix

Table 3: Matrix classes

- **global**: The bit is set to one if this matrix has to be saved in the global workspace when loaded into MATLAB®. This flag is ignored by the library, which sets this bit to zero: every outputted matrix will be a local object in MATLAB®.
- **logical**: The bit is set to one if this matrix has to be interpreted as containing logical (boolean) values when loaded into MATLAB®. The library ignores this flag, setting it to zero. No logical matrices can be outputted with the library, and imported matrices will be considered as matrices containing only 0's and 1's – but interpreted as numbers.

Dimensions This sub-element is common to all arrays. It is a sequence of miINT32[1]. In the most common case it contains exactly two integers which specify the number of rows and the number of columns, respectively. Although MAT-files allow to multi-dimensional tensorsthe library currently only provides support for 2-dimensional matrices. Note that the number of dimensions is an implicit piece of information which can be extracted from the size of this sub-element: since every dimension accounts for exactly 4 bytes, the size of this sub-element divided by 4 gives the number of dimensions.

Name This sub-element is common to all arrays. The name of the matrix is the name of the variable which will be used to reference the matrix when loaded into MATLAB®. The data is to be interpreted as ASCII text.

Real part This sub-element contains the real part of the entries. The order is column-major. Any numeric type can be used for storing this data, independently of the class of the matrix. For instance, one could store the entries of a mxDOUBLE_CLASS matrix as integers in order to save space, if all of them are representable as integers without loss of precision.

Imaginary part This optional sub-element contains the imaginary part of the entries. This sub-element is present if and only if the matrix is declared to be complex trough the *complex* flag. The type of the entries of this sub-element may differ from the type of the real part. So one can write a mxDOUBLE_CLASS matrix with real part stored as miINT32 and imaginary part stored as miUINT16; when loaded inside MATLAB®, that matrix is interpreted as an array containing complex numbers in double precision.

[1]See table 1 on page 2

Sub-element name	**Type**	**Number of bytes**
Flags	miUINT32	8
Dimensions	miINT32	$4 \cdot nDim$
Name	miINT8	$nChars$
Row index	miINT32	$4 \cdot nnz$
Column index	miINT32	$4 \cdot (cols + 1)$
Real part	Any numeric type	$typeSize \cdot nnz$
Imaginary part (optional)	Any numeric type	$typeSize \cdot nnz$

Table 4: Sub-elements for sparse matrices

2.2 Sparse matrices

Sparse matrices are stored as MATLAB® arrays that contain only the non-zero elements using CCS (aka CSC) [5] format. The indexing always starts at 0. Table 4 lists the sub-elements of sparse matrices, which are now discussed.

Flags Contains the same information as the flags for dense matrices. The class is always mxSPARSE_CLASS. Sparse matrices are interpreted as containers for doubles. Additionally, the last 4 bytes, which were unused inside the flags sub-element of dense matrices, contain the number of non-zero values stored as an miUINT32 integer that we call *nnz*.

Dimensions Same as the correspinding sub-element for dense matrices. *Sparse matrices have exactly 2 dimensions.*

Name Same as the corresponding sub-element for dense matrices.

Row index The row index of every non-zero entry, ordered according to the column-major strategy.

Column index For each column holds the index of the first element, i.e. the number of elements before the column; at the end the value *nnz* is appended.

Real part The real part of each non-zero entry. Any numeric type can be used here.

Imaginary part The imaginary part of each non-zero entry. Like dense matrices, sparse matrices have this sub-element if and only if the complex flag is set to one. Any numeric type can be used, independently of the type used for the real part, cf. the corresponding sub-element of **Dense matrices**, above.

2.3 Remarks

Types MAT-files do not only contain the data, but also the information on the storage form. Naturally, this is all dynamic data which is parsed at run-time. Therefore C++ has extract dynamically typed data into statically typed arrays. For example, the library could read a MAT-file and save the data in an array; but the type of the data contained in the MAT-file is only known at run-time, while the type of the array is decided at compile-time. We adress this issue by treating the data in MAT-files as raw bytes and apply type casts.

Dimensions In MAT-files, the minimum number of dimensions of a matrix is 2, i.e. vectors are treated as matrices with either only one row or only one column. There is no maximum number of dimensions and, for simplicity, the library only supports the case of 2 dimensions. An exception is throw if the user tries to load a matrix with a higher number of dimensions.

Complex numbers Common linear algebra libraries provide support for complex-valued matrices, treating them as matrices of std::complex objects. In MAT-files real and imaginary part of a matrix are stored separately. Consequently, the library separates the two parts while outputting complex matrices and constructs objects of type std::complex while inputting them.

3 Wrappers

C-arrays, std::vector, Boost::uBLAS, Eigen, Armadillo [6], MKL [7] are only a few ways to handle dense or sparse matrices. The library allows any of these to be used by means of Wrappers. A Wrapper is an interface that handles a matrix, reads or writes data on it, provides a uniform and standardized access from external objects to the matrix object without the need – from the external point of view – to know the methods of that object or its properties.

A Wrapper is a class, which respect some *structural rule*, such as a set of public methods and public typedefs[2]. These methods and typedefs serve as *unified interface* for accessing the matrix's properties. Since different matrix classes can also be designed with different patterns in mind, we define several types (designs) of wrappers, each addressing a certain access and storage strategy. So, for example, `ArrayDenseWrapper` is a wrapper design for matrix objects that store, or can easily return, their elements in column-major order, while `CWiseDenseWrapper` is a wrapper design for matrix objects that store the entries otherwise and for which it is preferrable to request them component by component. At this point, for sparse matrix objects, the library only provides the wrapper design `ArraySparseWrapper` for matrix objects which return the data in the CCS format.

Let us present some examples of wrapper implementing certain designs.

3.1 Boost wrappers

A relevant part of this work was to develop wrappers for matrices provided in the Boost uBLAS library. Those matrices are STL-like objects which customizable through template parameters. For example, for dense matrices, the user can specify the scalar type, the underlying storage strategy (row-major is default and column-major is available) and the storage structure (`std::vector` or a similar class).

3.1.1 `BoostDenseOWrapper`

An example of a wrapper class implementing the design `ArrayDenseWrapper` is `BoostDenseOWrapper`, which works with dense uBLAS matrices, i.e. `matrix`

[2] In C++, a typdef is a programming clause that assigns an alternative name to an existing type or class. It is often used for handling of statical informations through templates.

objects. Any such matrix object can be used with this wrapper, but some tricks – which we discuss below – allow the user to do an optimal usage of these objects.

Let us now focus on the requirements of en `ArrayDenseWrapper`. A class is considered a valid `ArrayDenseWrapper` if it has:

- A public typedef named `Type` for the class `ArrayDenseWrapper` which is expressed by the following clause: `typedef ArrayDenseWrapper Type;`.
- A public method with prototype `void size(miINT32_t&, miINT32_t&)`, which provides the number of rows (first argument) and of cols (second argument) through references.
- A public method with prototype `bool isComplex()` which returns true iff the scalar type is complex (i.e. is of type `std::complex<T>`).
- A public method with prototype `MATtype realType()` which returns the type (`MATtype` is a typedef for the signed 32-bit integer type, that represent a type – see Section 2) of the real part of the entries. In case of Boost complex matrices, the type of the real part is the same as the type of the complex part.
- A public method with prototype `MATtype imagType()` which returns the type of the complex part. In this case, it simply returns the same as `realType()`.
- A public method with prototype `const char *realData()` which returns a pointer to the array containing the real part of the data (in column-major order).
- A public method with prototype `const char *imagData()` which returns a pointer to the arrays containing the imaginary part of the data (in column-major order).

Besides, a constructor (which is not mandatory for `ArrayDenseWrappers`) is added in order to organize the data. The constructor is the only template part of this class: it accepts any matrix type, extracts the data, making use of traits and other tools for constructing dynamical information from statical one, copies and formats the data in case of complex types or row-major-ordered elements, stores pointers to the internal storage in case of real, column-major – or symmetric, if the user specifies this information – matrices. An important remark for this wrappers we point out is that, once the wrapper is constructed with a matrix, the matrix should not be modified until the wrapper is used – i.e. the data has been written to a MAT-file – or the result is undefined because the pointers used for internal storage inside the matrix can be invalidated, which results in a corruption of the information inside the wrapper.

3.1.2 `BoostCompressedOWrapper`

The ideas behind this wrapper for outputting sparse matrices – provided as objects of type `compressed_matrix` from the Boost uBLAS library – are the same as for the already described `BoostDenseOWrapper`. This wrapper, like every wrapper implementing the design `ArraySparseWrapper`, has the following entities:

- A public typedef named Type for the class ArraySparseWrapper which is expressed by the following clause: typedef ArraySparseWrapper Type;.
- A public method with prototype void size(miINT32_t&, miINT32_t&) which provides the number of rows (first argument) and of cols (second argument) through references.
- A public method with prototype bool isComplex() which returns true iff the scalar type is complex (i.e. is of type std::complex<T>).
- A public method with prototype miINT32_t nnz() that returns the number of non-zero values.
- A public method with prototype const char *ir() that returns a pointer to the begin of the array containing the row indices formatted as 32-bit signed integers and ordered according to column-major storage strategy.
- A public method with prototype const char *jc() that returns a pointer to the begin of the array containing the column indices formatted as 32-bit signed integers and ordered according to column-major storage strategy.
- A public method with prototype MATtype realType() which returns the type (MATtype is a typedef for the signed 32-bit integer type, that represent a type – see Section 2) of the real entries. In case of Boost complex matrices, the type of the real part is the same as the type of the complex part.
- A public method MATtype imagType() which returns the type of the complex part. In this case, it simply returns the same as realType().
- A public method const char *pr() which returns a pointer to the array containing the real part of the data (in column-major order).
- A public method const char *pi() which returns a pointer to the arrays containing the imaginary part of the data (in column-major order).

Like for BoostDenseOWrapper, this wrapper has a constructor that checks the types, extracts the data and formats it if needed, or just stores the pointers if no modification is necessary. If the matrix is real and stored according to the CCS schema, then the entries can be written directly to the file from their actual position in the matrix object; otherwise they are copied and reorganized in order to be ready to be written, which is more expensive and requires more memory. The indices have to be stored as 32-bit integers and according to the CCS schema in order to be written directly to the file without reordering. Indices and entries are treated separately, so that the operation also benefit by speed-up if the matrix storage is not optimal, e.g. in case of complex entries – data is reorganized – with 32-bit indices and CCS storage – no indices reorganization needed.

4 IO operations

Having standardized wrapper objects, which provide a blackbox access to matrix objects' data, we can construct classes that use them to perform input and

output operations on MAT-files. We thus introduce two concepts: devices and writers/readers. For writers and readers, see the section 4.2 on page 9.

4.1 Devices

The library introduces two classes that manage input and output operations. These classes are *devices*. A device reads or writes the data to a stream – commonly a file, but in practice also other types of streams –, recognizes and organizes the structure of the MAT-file, handles the metadata, provides access to the matrices and their informations. They do not read or write directly to the stream but delegate these processes to the template classes `Reader` and `Writer`, respectively. The interface is designed to be as simple as possible.

OutputDevice The behavior of `OutputDevice` is very simple: it contains a method for adding a human-readable description to the MAT-file and a method for writing a matrix. If the method for the description is never called, a standard description is written. The method for writing a matrix – `writeArray (const std::string& name, const WrapperT& wrapper)` – takes as first argument the name of the matrix and a wrapper as second argument; this is a template method that accepts any class as parameter: this parameter is then passed to the Writer that will decide which actual specialization use depending on the design of the Wrapper class.

InputDevice InputDevice is slightly more complex. When a MAT-file is open, the device reads the file header, then reads every matrix present in the file. If a matrix is compressed, then it inflates the data until it can read each item of relevant information, i.e. class, flags, dimensions and name; this information is written in the approximately first 60 bytes of the deflated string: this process is very fast. Information about class, flags, dimension and name is then stored together with the whole string (deflated or inflated) in an object named `MatrixInfo`. The whole content of the MAT-file - without metadata – is then represented as a vector of such objects.

The user can make use of iterators of that vector in order to get information about the matrices contained in the MAT-file and request every single matrix through its name using the method `void readArray(const std::string& name, WrapperClass& wrapper)`. Here, an input wrapper has to be created before this call and its type will be passed as template parameter to the `Reader`.

4.2 Writers and readers

For each wrapper type design a writer and a reader is provided. These objects make use of the unified access methods to the matrices provided by wrppaer in order to writeor to read MAT-files.

For that purpose, unformatted IO-methods of C++ standard streams are used. For example, the code for writing the elements of a dense matrix to a MAT-file – let elements be of type double and the matrix have dimensions 4×4; let `out` be the file device; let `wrapper` be a wrapper of type `ArrayDenseWrapper` – will be the following: `out.write(wrapper.realData(), 8*16);`. For input operations the equivalent method `ifstream::read` is used.

Notice that writers and readers are internal classes: the user will never need to use them directly, at least until he writes a custom wrapper type and therefore needs a custom reader or writer. These classes are thus included (hidden) in the namespace `matfile::detail`.

5 Usage examples

We will show here some usage examples with the library and the uBLAS matrices. Please remember that you have to make an aware use of Boost uBLAS classes, in particular of their template parameter, in order to efficiently work with MAT-files.

5.1 Dense matrices

The uBLAS class for dense matrices is `matrix`. The default storage strategy is row-major, which is the less efficient one for our purposes. We can either use column-major storage by adding a template parameter, or tell the `BoostDenseOWrapper` not to transpose the data. Here we present the example with the first solution. For the latter we refer to the online documentation.

```
#include <boost/numeric/matrix.hpp>
#include <matfile/Output>
#include <matfile/Wrappers/BoostDenseOWRapper.hpp>

using namespace boost::numeric::ublas;
using namespace matfile;

int main()
{
  //Declare the first matrix: real, column-major
  matrix<double, column_major> my_real_matrix(8, 8);

  // Fill the matrix with some value...
  /*
    ...
  */


  // Save the matrix inside a MAT-file:

  // 1. Open the device
  OutputDevice out("my_mat_file.mat");

  // 2. Put the matrix into a wrapper
  BoostDenseOWrapper wrapper(my_real_matrix);

  // 3. Write the matrix
  out.writeArray("my_real_matrix", wrapper);
```

```
  // 4. Close the device
  out.close();

}
```

5.2 Sparse matrices

uBLAS provides more than one class for handling sparse matrices. The two most common classes are:

`mapped_matrix` Each element is saved as 3-tuple with row-index, column-index and value. Insertions are very fast (constant complexity), since they only consist of three additions to vectors, while linear algebra operations are slow due to the large number of iterations to be performed.

`compressed_matrix` CRS (compressed row storage) or CCS (compressed column storage) are used: the storage strategy allows optimized loops for typical linear algebra operations, resulting in significantly faster computations. The main disadvantage of this strategy is the linear complexity of insertions.

The library only supports `compressed_matrix` objects. The user can also specify other template parameters that affect the behavior of the matrix and consequently of the wrapper. We discuss here the five template parameters and we encourage the user to be carefully while deciding the values to assign to these parameters.

T The scalar type: can be of any numeric type (`int`, `float`, `double`,...) or a `std::complex` type.

F The storage strategy: can be either `row_major` (default) or `column_major`. `row_major` means CRS storage, while `column_major` means CCS. At this point, only `column_major` is supported by the library.

IB The index base. Common values are 0 (default) and 1. At this point, only 0 is supported by MATfile.

IA The index array type. Can be any STL vector-like class (the default is `unbounded_array<std::size_t>`). **Warning**: this template parameter also defines the index type, which has to be an integer type. Every type is supported, but only 32-bit types allow the library to be very efficient. Therefore the preferred index array class should be of the form `some_array_type<miINT32_t>`.

TA The value array type. Can be any vector-like class, with the same elemnt type specified in the first parameter.

We present an usage example with a `BoostSparseOWrapper` handling a `compressed_matrix`. Note that we set four of the five template parameters of the class, resulting in a quite complex type definition. We put it in a typedef and splitted the typdef among many lines, which makes the code more readable.

```
#include <boost/numeric/matrix_sparse.hpp>
#include <matfile/Output>
#include <matfile/Wrappers/BoostSparseOWRapper.hpp>

using namespace boost::numeric::ublas;
using namespace matfile;

//Useful typedef for the matrix
typedef compressed_matrx<
  double,
  column_major,
  0,
  unbounded_array<miINT32_t>
> RealMartixT;

int main()
{
  // Declare the matrix
  // 8 x 8 real matrix with 12 non-zero values
  RealMartixT my_real_matrix(8, 8, 12);

  // Fill the matrix
  /*
    ...
  */


  // Save the matrix inside a MAT-file:

  // 1. Open the device
  OutputDevice out("my_mat_file.mat");

  // 2. Put the matrix into a wrapper
  BoostSparseOWrapper wrapper(my_real_matrix);

  // 3. Write the matrix
  out.writeArray("my_real_matrix", wrapper);

  // 4. Close the device
  out.close();

}
```

6 Tests and performance

We ran some tests in order to give an approximation for the library performance. The tests involved dense and sparse, real and complex matrices. The typical test consists of the following steps:

1. A set of matrices is created and saved whithin MAT-files with MAT-

LAB®. Each MAT-file contains only one matrix and the sizes of the matrices are determined by the requirement that the number of elements scales exponentially and covers a relatively wide range of common values (from 300×300 up to 8000×8000 for dense matrices and from 10000 up to 12000000 elements for sparse ones). Dense matrices are generated with the `rand` function of MATLAB®, resulting in square random matrices with values between 0 and 1. Sparse matrices are Poisson-matrices, generated with `gallery('poisson', size)`.

2. The matrices are loaded whithin the C++ program and read into uBLAS objects. Each step of the input operation is measured separately.

3. The matrices are saved into other MAT-files. Each step of the output operation is measured separately.

4. The timings are saved (into MAT-files) for analysis with MATLAB®.

5. The outputted matrices are verified to agree exactly.

We used the class `boost::ublas::matrix` for dense matrices and the class `boost::ublas::compressed_matrix` for sparse matrices; we adjusted the template parameters in order to comply with column-major storage and optimal (i.e. 32-bit) indices for sparse matrices (see the above examples). The C function `gettimeofday` was used for the timings.

6.1 Input operations

The input process consists of the following sequence of operations:

Open matrix The MAT-file device is opened; the header of the MAT-file is read and interpreted; the first bytes of the matrix are inflated and read. The relevant pieces of information – matrix flags, size, name – are stored, while the rest is copied into the RAM.

Read matrix Matrix data is inflated and given to the wrapper, which decides eighter to store it in a temporary memory location or to write it directly to the assigned uBLAS object of type `matrix`.

Return matrix The wrapper reorders and casts the data in order to fill the matrix if the operation **Read matrix** did not do that.

Note: these benchmarks refer to MAT-file compressed with zlib, which is probably the most common case if the file is created with MATLAB®. Non-compressed input operation will be significantly faster as no inflation has to be performed.

6.1.1 Dense real matrices

The Figure 1 shows that a dense matrix with about 6.5×10^7 elements is read from the hard disk in less than 7 seconds. This value obvously depends on the hard disk's performance, fragmentation, location, and other parameters.

Note that the **Return matrix** operation has in this case constant complexity, since everything is already done by **Read matrix**.

6.1.2 Dense complex matrices

Figure 2 documents the detailed time consumption for an input operation with complex numbers. As expected, the **Open file** and **Read matrix** operations take about double the time needed by the same operations with real matrices: this is clear, since the same number of complex elements versus real ones involves twice the number of entries (which are in double precision here). The last operation exhibits however a different behaviour than for real matrices: the **Return matrix** operation has linear complexity with respect to the number of elements and requires for a 8000×8000 matrix about 3 seconds to create 8000^2 objects of type std::complex<double>. This is however a relatively small task when compared with the most expensive operation, which is **Read matrix**.

6.1.3 Sparse real matrices

See Figure 3 for the results of this test. As a remark, we point out that the only significantly expensive operation here is **Read matrix**: for a sparse matrix with about 1.2×10^7 nonozero elements this operation takes about 1.2 seconds, while the other two take jointly less than 0.05 seconds. We expect the reason to be the following: the deflate operation on sparse matrices is significantly more efficient than on dense matrices, since random data are unlikely to be compressed; dense matrices contain almost only random data (two double precision numbers between 0 and 1 can differ on 52 or more of 64 bits), while sparse Poisson-matrices contain only two different real values, 4 and -1, and the indices are integer numbers which exhibit a relatively small amount of information entropy. That means, MAT-files containing sparse matrices are smaller than ones with dense matrices and the loading operation (**Open file**) is less expensive. As for dense real matrices, no operation is needed whithin **Return matrix**, resulting in a constant complexity for that step.

6.1.4 Sparse complex matrices

See Figure 4 for the results of this test. Equivalent remarks as for complex dense matrices apply: the **Open file** and **Read matrix** operation are computationally about twice as expensive as for real matrices due to the bigger amount of data – precisely **Read matrix** is only 40% more expensive than before, since the indices read operation has to be performed only once –, while **Return matrix** has now linear complexity due to the need to fill the std::complex<double> objects.

6.2 Output operations

The output process in our tests consists of the following sequence of steps:

Create wrapper A wrapper object is created with a given matrix (in our tests from a matrix returned by the input process described above). If the matrix object obeys to a set of conditions – for these conditions we refer to section 3.1.1, section 3.1.2 and to the online documentation – this step has constant complexity, since the data is just referenced through pointers, but not reorganized or copied. Otherwise the step has linear complexity with respect to the number of nonzero values.

Figure 1: Input of real matrices

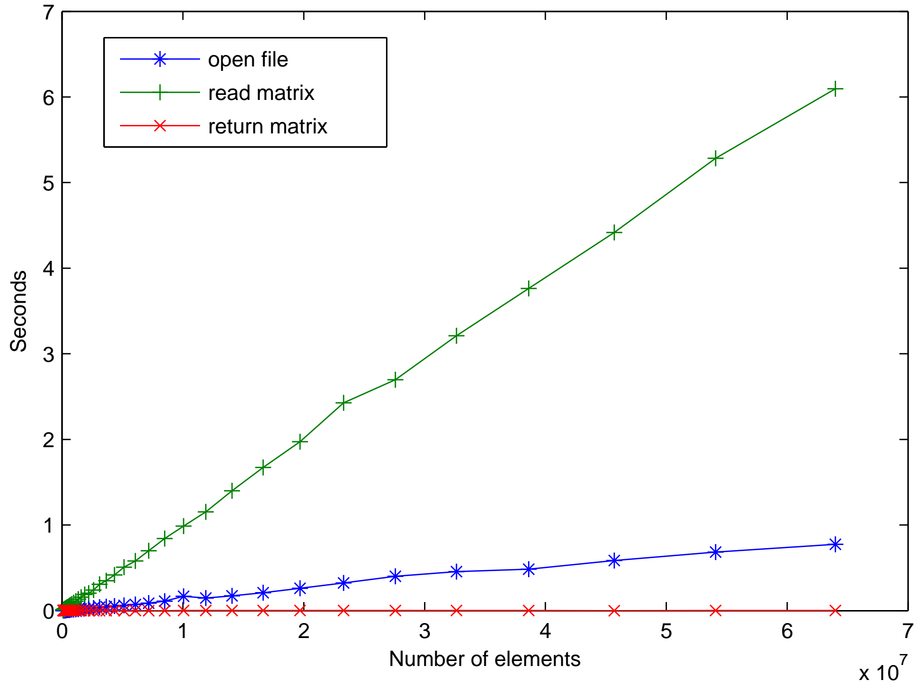


Figure 2: Input of complex matrices

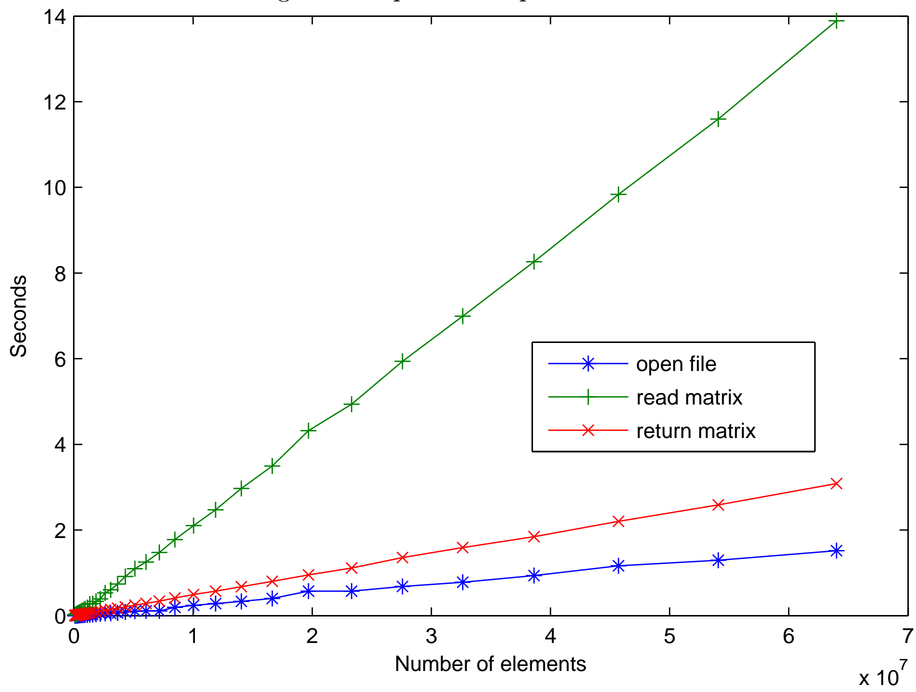


Figure 3: Input of sparse real matrices

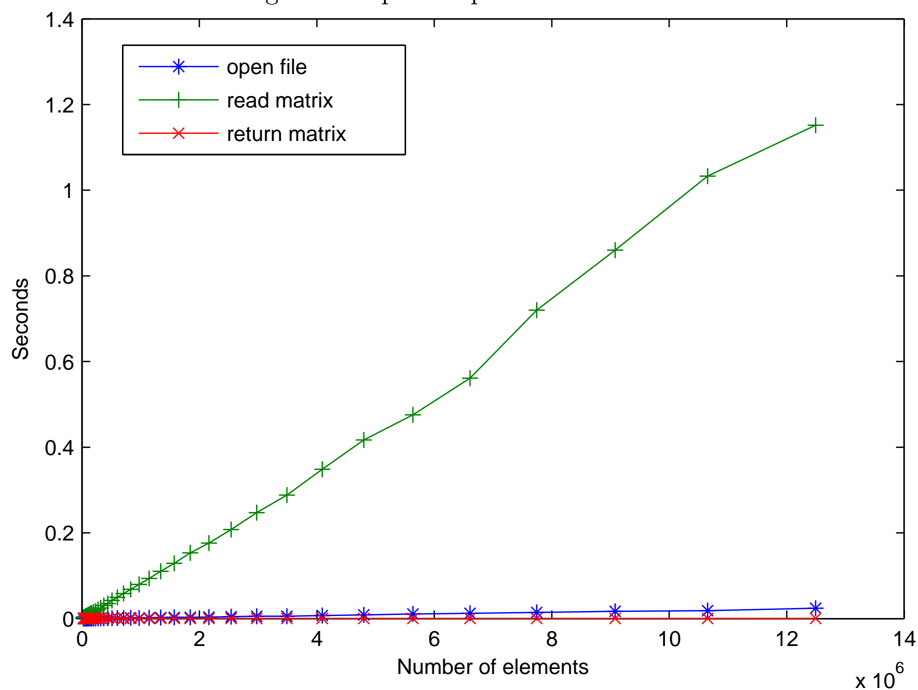
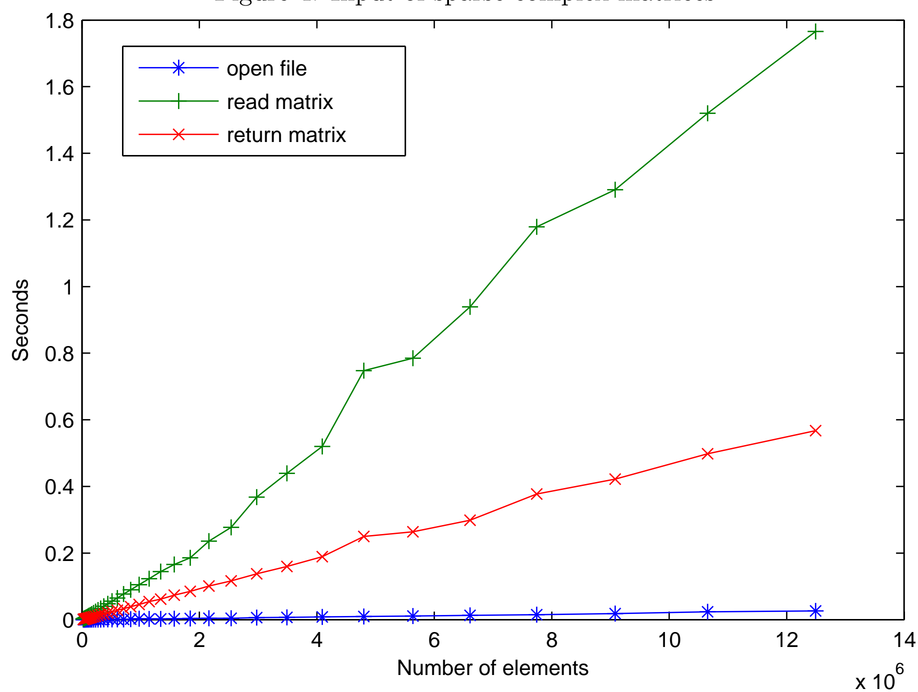


Figure 4: Input of sparse complex matrices



Write matrix The data is written to a MAT-file. This step has linear complexity with respect to the number of nonzero values.

Note: no compression is performed for these benchmarks. Actually, the library does not provide support for compressed output yet.

6.2.1 Dense real matrices

The Figure 5 shows quite clearly that this output operation is very simple and fast: the **Create wrapper** operation runs in constant time, as the organization of the matrix is a trivial task: the data as provided by the mytrix object is ready to be written to the MAT-file. The performance of the second operation depends on hardware properties.

6.2.2 Dense complex matrices

Figure 6 looks very different: actually, the most expensive operation while outputting complex matrices is not the **Write matrix**, which as fast as before, but the **Create wrapper**; the data has to be reorganized: the real part has to be separated from complex one. As a consequence, out measurements show that writing a complex matrix to a MAT-file is twenty times more time-consuming that the same task with a real matrix of same size.

6.2.3 Sparse real matrices

In Figure 7 we have again the situation where the **Create wrapper** operation has constant complexity, while the **Write matrix** runs with constant complexity.

6.2.4 Sparse complex matrices

An intresting phenomenom appears in Figure 8: the two operations are nearly equally expensive and the **Create wrapper** requires signifantly less time than on dense matrices. This is due to the fact that the CCS indices – see Section 2.2 and 3.1.2 – are stored in the matrix object in the same format as they will be written onto the MAT-file and consequently only the complex entries have to be reorganized.

7 Conclusions and outlook

The presented library for reading and writing MAT-files (the MATLAB®'s standard for data storage to disk) is already quite stable when working with matrix and vector objects provided in the Boost uBLAS library. We have presented timings involving output and input processes acting on dense and sparse, real and complex matrices. The results show that every process has linear complexity with respect to the number of nonzero values. The optimizations which are accomplished when the matrix object properties allow them make the library fast and memory-preserving. The aim of easyness was in our opinion archieved, since only few lines of code allow the user to realize every operation. The design pattern we implemented through *wrappers* enable the use of this library with a

Figure 5: Output of real matrices

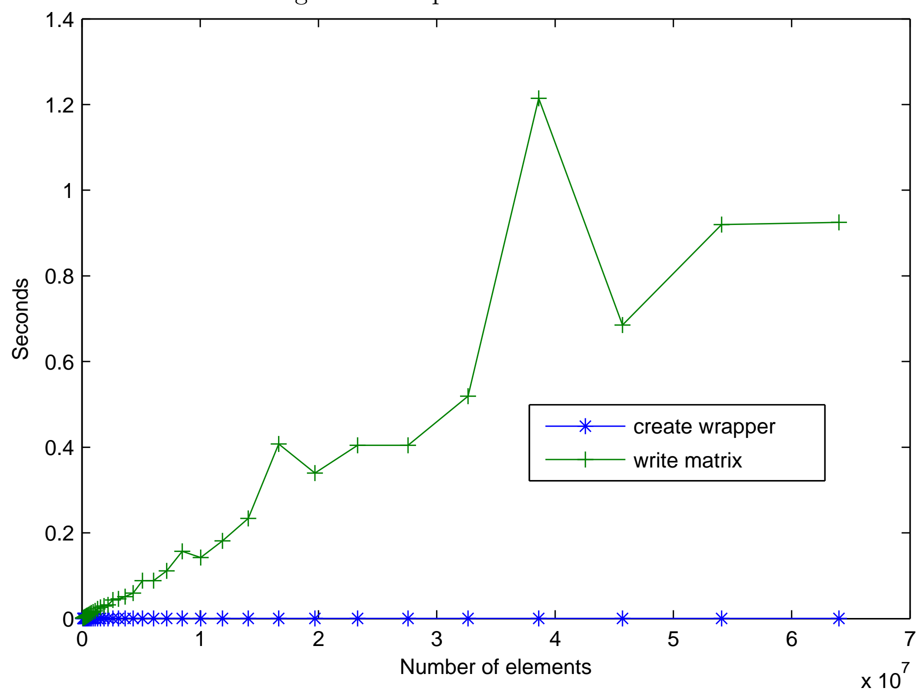


Figure 6: Output of complex matrices

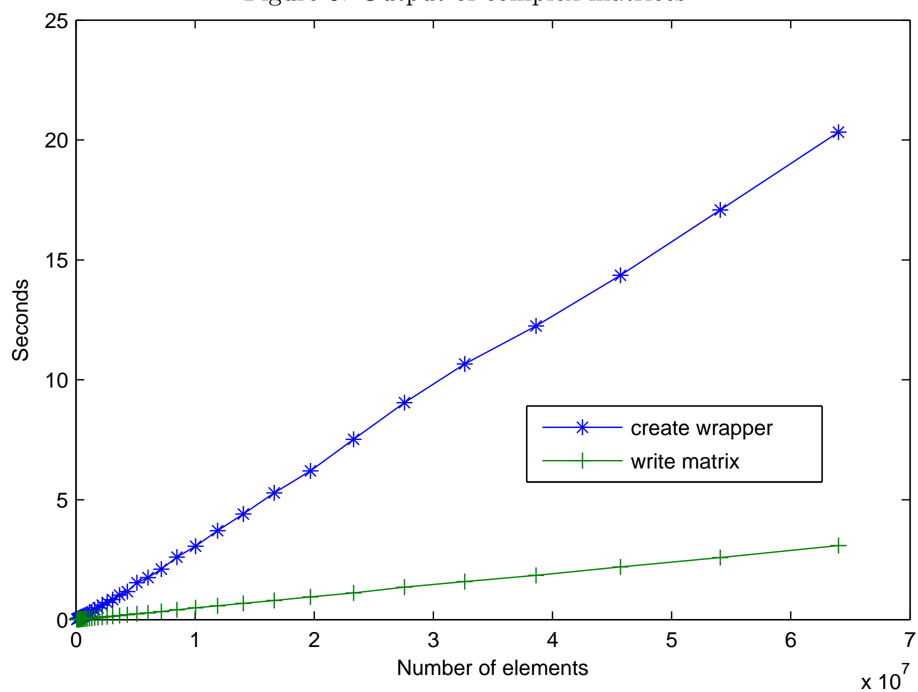


Figure 7: Output of sparse real matrices

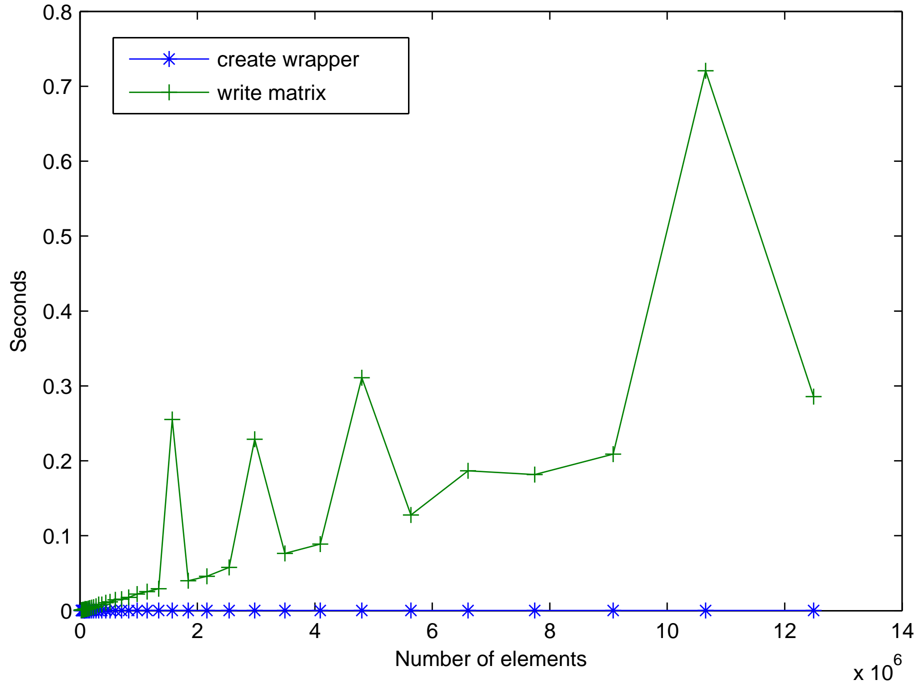
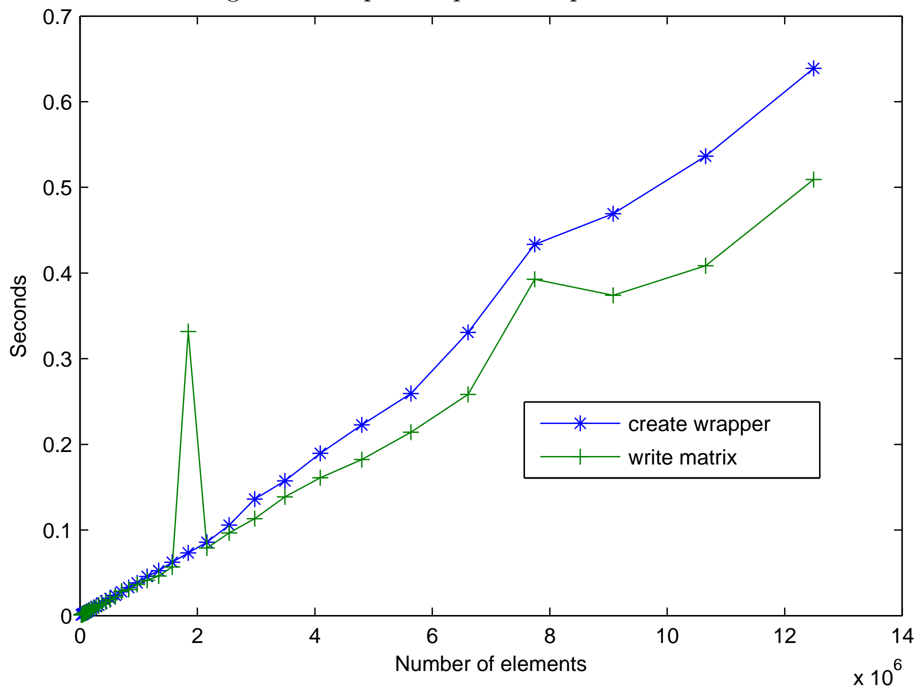


Figure 8: Output of sparse complex matrices



potentially infinite set of linear algebra libraries with the only need to construct special classes for standard access to the matrix objects. At this point only Boost uBLAS matrix objects are supported.

However, we consider our work not to be ended. The library will be actively developed in the future. Here we list some interesting points for future improvement:

- Construction of some data structure. Often one uses uBLAS or other matrix libraries just to store the results of experiments and to subsequently save them to a file for postprocessing. With `miMATRIX` we tried to reproduce the MAT-file usual data structures, in order to let the user also choose those objects for multi-dimensional matrices, sparse matrices, MATLAB® structures, objects and cell arrays. No linear algebra operations are provided, only simple but powerful storage of data. The construction of this library is in progress. We do not discuss this library in this document, as not directly related with the purpose of our work.
- Definition of new wrapper designs supporting objects like tensors, structures, objects and cell arrays.
- Construction of wrappers. So far, only the matrices provided by the Boost uBLAS library and `std::vector` are supported.
- Improvement of devices. OutputDevice and InputDevice have so far a very limited set of possible operations. For example, it is impossible to delete a matrix written to an OutputDevice or to append matrices to a MAT-file. It would not be not so difficult to extend the functionalities of these devices and maybe to introduce a new IODevice for edit of existing MAT-files.

Aknowledgements

The author thanks the Seminar for Applied Mathematics and in particular Prof. Dr. Ralf Hiptmair for the proposal of this bachelor thesis. Many thanks go to my advisor Roman Andreev who helped me during the implementation of the library and the drafting of this document. Concerning this, also thanks to Julia Schweitzer (SAM) and Eliana Pusterla for helping with the proof-reading of the thesis.

I would like to mention my sources of information:

- The MathWorks® Company that places a large quantity of free documentation concerning its products at disposal.
- The Eigen community and its mailing list. The members are very kind and competent.
- The Boost community, in particular the mailing list of uBLAS that is often very helpful for me.
- In general the open-source community. I received many helps on the international Gentoo forum[3] and the italian Ubuntu forum[4].

[3] http://forums.gentoo.org
[4] http://forum.ubuntu-it.org

References

[1] MathWorks® Company,
http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/matfile_format.pdf,
Revision September 2010

[2] MathWorks® Company,
http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/apiext.pdf,
Revision September 2010

[3] Boost uBLAS library,
http://www.boost.org/doc/libs/release/libs/numeric/ublas/doc/index.htm

[4] Eigen library,
http://eigen.tuxfamily.org

[5] Netlib, Compressed Column Storage,
http://netlib.org/linalg/html_templates/node92.html

[6] Armadillo library,
http://arma.sourceforge.net

[7] Intel® Math Kernel Library,
Intel® Company,
http://software.intel.com/en-us/intel-mkl

[8] Andrea Arteaga,
matfile library,
n.ethz.ch/~arteagaa/matfile