MATIO

User Manual
for version 1.5.26
November 2023

Christopher C. Hulbert

Copyright (c) *2015-2023*, *The matio contributors*.
Copyright (c) *2011-2014*, *Christopher C. Hulbert*.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this
   list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice,
   this list of conditions and the following disclaimer in the documentation
   and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Table of Contents

1 Introduction 1

1.1 About and Licensing 1
1.2 Incompatible Changes from 1.3 1
1.2.1 Type Change for Dimensions Array 2
1.2.2 Removed Preprocessor Flag to Conserve Memory 2
1.2.3 Renamed Structure Field Lookup Enumerations 2
1.2.4 Memory Conservation with Cells and Structures 2

2 Quick Start 3

2.1 Opening and Creating MAT Files 3
2.2 Reading Variables in a MAT File 4
2.2.1 Reading a Variable by Name 4
2.2.2 Iterating Over Variables in a MAT File 5
2.3 Writing Variables 6
2.3.1 Writing Numeric Arrays 6
2.3.2 Writing Cell Arrays 7
2.3.3 Writing Structure Arrays 8

3 Building matio 11

3.1 Quick Build Guide 11
3.2 Configure Options 11
3.3 CMake build system 11
3.4 Visual Studio 12
3.5 Testsuite 12

4 MATLAB Variable Structure 13

4.1 Variable Information 13
4.1.1 Sparse Matrix Variables 13
4.1.2 Structure Variables 13
4.1.3 Cell Variables 13

1 Introduction

1.1 About and Licensing

The *matio* software contains a library for reading and writing MATLAB MAT files. The *matio* library (*libmatio*) is the primary interface for creating/opening MAT files, and writing/ reading variables.

This *matio* software is provided with the Simplified BSD License reproduced below. The license allows for commercial, proprietary, and open source derivative works.


```
Copyright (c) 2015-2023, The matio contributors
Copyright (c) 2011-2014, Christopher C. Hulbert
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice,
   this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice,
   this list of conditions and the following disclaimer in the documentation
   and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.
```


1.2 Incompatible Changes from 1.3

This version has changes that break compatibility with the 1.3 versions of the *matio* software. This section lists these changes and how existing code should be modified to handle these changes.

1. dims field of matvar_t structure changed to size_t *
2. MEM_CONSERVE preprocessor definition removed
3. BY_NAME and BY_INDEX renamed
4. Added `MAT_` prefix to enumerations of `matio_compression`
5. Changed name of structure for complex split-format data from `struct ComplexSplit` to `struct mat_complex_split_t`
6. Changed name of sparse data structure from `sparse_t` to `mat_sparse_t`.
7. Changed meaning of memory conservation for cell arrays and structures

Each of these changes are described in the remaining sections, and as necessary include recommendations to upgrade existing code for compatibility with this version.

1.2.1 Type Change for Dimensions Array

The existing dims field of the `matvar_t` structure was an `int *` which limited the maximum size of a dimension to 2^{31}. In version 1.5, the type was changed to `size_t *` which allows a variable of length 2^{31} on 32-bit systems, but $2^{64} - 1$ on 64-bit system. To upgrade to version 1.5, all existing code should ensure the use of `dims` allows for `size_t`, and that any use of the `Mat_VarCreate` function passes an array of type `size_t` and not `int`. Not upgrading to `size_t` is likely to produce segmentation faults on systems where `sizeof(size_t) != sizeof(int)`.

1.2.2 Removed Preprocessor Flag to Conserve Memory

Previous versions of the *matio* library had a preprocessor macro `MEM_CONSERVE` that was passed as an option to `Mat_VarCreate` to tell the library to only store a pointer to the data variable instead of creating a copy of the data. Copies of scalars or small arrays are not critical, but for large arrays is necessary. In version 1.5, this macro has been changed to the enumeration value `MAT_F_DONT_COPY_DATA`. A quick search/replace can quickly upgrade any references to `MEM_CONSERVE`. Alternatively, since `MAT_F_DONT_COPY_DATA` has the same value as `MEM_CONSERVE`, software using *matio* can simply define `MEM_CONSERVE` to 1.

1.2.3 Renamed Structure Field Lookup Enumerations

The `BY_NAME` and `BY_INDEX` enumerations are used by `Mat_VarGetStructField` to indicate if the field is retrieved by its name, or by its index in the list of fields. To bring these into a *matio* namespace and hopefully avoid conflicts, these have been renamed to `MAT_BY_NAME` and `MAT_BY_INDEX`. A quick search/replace operation should be able to correct existing code that uses the old names.

1.2.4 Memory Conservation with Cells and Structures

Previous versions of *matio* would still free fields of structures and elements of cell arrays even if created with memory conservation flag set. In the latest version of *matio*, the fields/cell elements are not free'd if the structure was created with the `MAT_F_DONT_COPY_DATA` flag. This is useful if the fields/elements are referenced by another variable such as the case when they are indices of a larger array (i.e. `Mat_VarGetStructs`, `Mat_VarGetStructsLinear`).

2 Quick Start

2.1 Opening and Creating MAT Files

This section will show how to create a new MAT file, open an existing MAT file for read and read/write access, and close the MAT file.

The key functions in working with MAT files include:

- Mat_Open,
- Mat_CreateVer, and
- Mat_Close.

The following example program shows how to open a MAT file where the filename is the first argument to the program.

```
#include <stdlib.h>
#include <stdio.h>
#include "matio.h"

int
main(int argc,char **argv)
{
    mat_t *matfp;

    matfp = Mat_Open(argv[1],MAT_ACC_RDONLY);
    if ( NULL == matfp ) {
        fprintf(stderr,"Error opening MAT file \"%s\"!\n",argv[1]);
        return EXIT_FAILURE;
    }

    Mat_Close(matfp);
    return EXIT_SUCCESS;
}
```

The `Mat_CreateVer` creates a new MAT file (or overwrites an existing file) with a specific version. The *matio* library can write version 4 MAT files, version 5 MAT files, version 5 MAT files with variable compression (if built with zlib), and HDF5 format MAT files introduced in MATLAB version 7.3. The format of the MAT file is specified by the third argument. The short example below creates a version 4 MAT file named *matfile4.mat*, a version 5 MAT file named *matfile5.mat* and an HDF5 format MAT file named *matfile73.mat*.

```
#include <stdlib.h>
#include <stdio.h>
#include "matio.h"

int
main(int argc,char **argv)
{
    mat_t *matfp;

    matfp = Mat_CreateVer("matfile4.mat",NULL,MAT_FT_MAT4);
    if ( NULL == matfp ) {
        fprintf(stderr,"Error creating MAT file \"matfile4.mat\"!\n");
        return EXIT_FAILURE;
```

```
    }
    Mat_Close(matfp);

    matfp = Mat_CreateVer("matfile5.mat",NULL,MAT_FT_MAT5);
    if ( NULL == matfp ) {
        fprintf(stderr,"Error creating MAT file \"matfile5.mat\"!\n");
        return EXIT_FAILURE;
    }
    Mat_Close(matfp);

    matfp = Mat_CreateVer("matfile73.mat",NULL,MAT_FT_MAT73);
    if ( NULL == matfp ) {
        fprintf(stderr,"Error creating MAT file \"matfile73.mat\"!\n");
        return EXIT_FAILURE;
    }
    Mat_Close(matfp);

    return EXIT_SUCCESS;
}
```

2.2 Reading Variables in a MAT File

This section introduces the functions used to read variables from a MAT file. The *matio* library has functions for reading variable information only (e.g. name, rank, dimensions, type, etc.), reading information and data, and reading data from previously obtained information. Reading information and data in separate function calls provides several conveniences including:

- Querying the names of variables in a file without reading data,
- Reading only some fields of a structure or elements of a cell array, and
- other actions where the variable data is not needed.

2.2.1 Reading a Variable by Name

If the name of the variable is known, the `Mat_VarRead` and `Mat_VarReadInfo` functions can be used. The `Mat_VarRead` function reads both the information and data for a variable, and the `Mat_VarReadInfo` reads information only. The short example below reads a named variable from a MAT file, and checks that the variable is a complex double-precision vector.

```
#include <stdlib.h>
#include <stdio.h>
#include "matio.h"

int
main(int argc,char **argv)
{
    mat_t    *matfp;
    matvar_t *matvar;

    matfp = Mat_Open(argv[1],MAT_ACC_RDONLY);
    if ( NULL == matfp ) {
        fprintf(stderr,"Error opening MAT file \"%s\"!\n",argv[1]);
        return EXIT_FAILURE;
    }
```

```
    matvar = Mat_VarReadInfo(matfp,"x");
    if ( NULL == matvar ) {
        fprintf(stderr,"Variable 'x' not found, or error "
                       "reading MAT file\n");
    } else {
        if ( !matvar->isComplex )
            fprintf(stderr,"Variable 'x' is not complex!\n");
        if ( matvar->rank != 2 ||
             (matvar->dims[0] > 1 && matvar->dims[1] > 1) )
            fprintf(stderr,"Variable 'x' is not a vector!\n");
        Mat_VarFree(matvar);
    }

    Mat_Close(matfp);
    return EXIT_SUCCESS;
}
```

2.2.2 Iterating Over Variables in a MAT File

For some applications, the name of the variable may not be known ahead of time. For example, if the user needs to select a variable of interest, a list of variables should be obtained. Like reading a variable by name, there are two functions that will read the next variable in the MAT file: `Mat_VarReadNext` and `Mat_VarReadNextInfo`. The short example shown below opens a MAT file, and iterates over the variables in the file printing the variable name.

```
#include <stdlib.h>
#include <stdio.h>
#include "matio.h"

int
main(int argc,char **argv)
{
    mat_t    *matfp;
    matvar_t *matvar;

    matfp = Mat_Open(argv[1],MAT_ACC_RDONLY);
    if ( NULL == matfp ) {
        fprintf(stderr,"Error opening MAT file \"%s\"!\n",argv[1]);
        return EXIT_FAILURE;
    }

    while ( (matvar = Mat_VarReadNextInfo(matfp)) != NULL ) {
        printf("%s\n",matvar->name);
        Mat_VarFree(matvar);
        matvar = NULL;
    }

    Mat_Close(matfp);
    return EXIT_SUCCESS;
}
```

2.3 Writing Variables

A variable can be saved in a MAT file using the `Mat_VarWrite` function which has three arguments: the MAT file to write the variable to, a MATLAB variable structure, and a third option used to control compression options. The variable structure can be filled in manually, or created from helper routines such as `Mat_VarCreate`. Note that MATLAB, and thus *matio*, has no concept of a rank 1 array (i.e. vector). The minimum rank of an array is 2 (i.e. matrix). A vector is simply a matrix with one dimension length of 1.

Optionally, a variable can be appended to an existing variable of an HDF5 format MAT file by the `Mat_VarWriteAppend` function, which takes the same arguments as `Mat_VarWrite` as the first three arguments and the dimension as fourth argument. The dimension argument is index 1 based, i.e., if it is set to `d`, the variable is appended along the d-th dimension. If a variable is to be created for later appending, it always must be written by the `Mat_VarWriteAppend` function and `Mat_VarWrite` must not be called.

2.3.1 Writing Numeric Arrays

Numeric arrays can be either real or complex. Complex arrays are encapsulated in the `struct mat_complex_split_t` data structure that contains a pointer to the real part of the data, and a pointer to the imaginary part of the data. The example program below writes two real variables x and y, and one complex variable z whose real and imaginary parts are the x and y variables respectively. Note the `MAT_F_COMPLEX` argument passed to `Mat_VarCreate` for z to indicate a complex variable.

```
#include <stdlib.h>
#include <stdio.h>
#include "matio.h"

int
main(int argc,char **argv)
{
    mat_t    *matfp;
    matvar_t *matvar;
    size_t    dims[2] = {10,1};
    double    x[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9,10},
              y[10] = {11,12,13,14,15,16,17,18,19,20};
    struct mat_complex_split_t z = {x,y};

    matfp = Mat_CreateVer("test.mat",NULL,MAT_FT_DEFAULT);
    if ( NULL == matfp ) {
        fprintf(stderr,"Error creating MAT file \"test.mat\"\n");
        return EXIT_FAILURE;
    }

    matvar = Mat_VarCreate("x",MAT_C_DOUBLE,MAT_T_DOUBLE,2,dims,x,0);
    if ( NULL == matvar ) {
        fprintf(stderr,"Error creating variable for 'x'\n");
    } else {
        Mat_VarWrite(matfp,matvar,MAT_COMPRESSION_NONE);
        Mat_VarFree(matvar);
    }

    matvar = Mat_VarCreate("y",MAT_C_DOUBLE,MAT_T_DOUBLE,2,dims,y,0);
    if ( NULL == matvar ) {
```

```
        fprintf(stderr,"Error creating variable for 'y'\n");
    } else {
        Mat_VarWrite(matfp,matvar,MAT_COMPRESSION_NONE);
        Mat_VarFree(matvar);
    }

    matvar = Mat_VarCreate("z",MAT_C_DOUBLE,MAT_T_DOUBLE,2,dims,&z,
                MAT_F_COMPLEX);
    if ( NULL == matvar ) {
        fprintf(stderr,"Error creating variable for 'z'\n");
    } else {
        Mat_VarWrite(matfp,matvar,MAT_COMPRESSION_NONE);
        Mat_VarFree(matvar);
    }

    Mat_Close(matfp);
    return EXIT_SUCCESS;
}
```

2.3.2 Writing Cell Arrays

Cell arrays are multidimensional arrays whose elements can be any class of variables (e.g. numeric, structure, cell arrays, etc.). To create a cell array, pass an array of `matvar_t *`. Detailed information on the MATLAB variable structure for cell-arrays is given in Section 4.1.3 [Cell Variables], page 13. The following example shows how to create a 3x1 cell array.

```
#include <stdlib.h>
#include <stdio.h>
#include "matio.h"

int
main(int argc,char **argv)
{
    mat_t    *matfp;
    matvar_t *cell_array, *cell_element;
    size_t    dims[2] = {10,1};
    double    x[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9,10},
              y[10] = {11,12,13,14,15,16,17,18,19,20};
    struct mat_complex_split_t z = {x,y};

    matfp = Mat_CreateVer("test.mat",NULL,MAT_FT_DEFAULT);
    if ( NULL == matfp ) {
        fprintf(stderr,"Error creating MAT file \"test.mat\"\n");
        return EXIT_FAILURE;
    }

    dims[0] = 3;
    dims[1] = 1;
    cell_array = Mat_VarCreate("a",MAT_C_CELL,MAT_T_CELL,2,dims,NULL,0);
    if ( NULL == cell_array ) {
        fprintf(stderr,"Error creating variable for 'a'\n");
        Mat_Close(matfp);
        return EXIT_FAILURE;
```

```
    }

    dims[0] = 10;
    dims[1] = 1;
    cell_element = Mat_VarCreate(NULL,MAT_C_DOUBLE,MAT_T_DOUBLE,2,dims,x,0);
    if ( NULL == cell_element ) {
        fprintf(stderr,"Error creating cell element variable\n");
        Mat_VarFree(cell_array);
        Mat_Close(matfp);
        return EXIT_FAILURE;
    }
    Mat_VarSetCell(cell_array,0,cell_element);

    cell_element = Mat_VarCreate(NULL,MAT_C_DOUBLE,MAT_T_DOUBLE,2,dims,y,0);
    if ( NULL == cell_element ) {
        fprintf(stderr,"Error creating cell element variable\n");
        Mat_VarFree(cell_array);
        Mat_Close(matfp);
        return EXIT_FAILURE;
    }
    Mat_VarSetCell(cell_array,1,cell_element);

    cell_element = Mat_VarCreate(NULL,MAT_C_DOUBLE,MAT_T_DOUBLE,2,dims,&z,
                        MAT_F_COMPLEX);
    if ( NULL == cell_element ) {
        fprintf(stderr,"Error creating cell element variable\n");
        Mat_VarFree(cell_array);
        Mat_Close(matfp);
        return EXIT_FAILURE;
    }
    Mat_VarSetCell(cell_array,2,cell_element);

    Mat_VarWrite(matfp,cell_array,MAT_COMPRESSION_NONE);
    Mat_VarFree(cell_array);

    Mat_Close(matfp);

    return EXIT_SUCCESS;
}
```

2.3.3 Writing Structure Arrays

Structure arrays are multidimensional arrays where each element of the array contains multiple data items as named fields. The fields of a structure can be accessed by name or index. A field can be a variable of any type (e.g. numeric, structure, cell arrays, etc.). The preferred method to create a structure array is using the `Mat_VarCreateStruct` function. After creating the structure array, the `Mat_VarSetStructFieldByName` and `Mat_VarSetStructFieldByIndex` functions can be used to set the fields of the structure array to a variable. The example below shows how to create a 2 x 1 structure array with the fields x, y, and z.

```
#include <stdlib.h>
#include <stdio.h>
#include "matio.h"
```

```
int
main(int argc,char **argv)
{
    mat_t    *matfp;
    matvar_t *matvar, *field;
    size_t    dims[2] = {10,1}, struct_dims[2] = {2,1};
    double    x1[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9,10},
              x2[10] = {11,12,13,14,15,16,17,18,19,20},
              y1[10] = {21,22,23,24,25,26,27,28,29,30},
              y2[10] = {31,32,33,34,35,36,37,38,39,40};
    struct mat_complex_split_t z1 = {x1,y1}, z2 = {x2,y2};
    const char *fieldnames[3] = {"x","y","z"};
    unsigned nfields = 3;

    matfp = Mat_CreateVer("test.mat",NULL,MAT_FT_DEFAULT);
    if ( NULL == matfp ) {
        fprintf(stderr,"Error creating MAT file \"test.mat\"\n");
        return EXIT_FAILURE;
    }

    matvar = Mat_VarCreateStruct("a", 2,struct_dims,fieldnames,nfields);
    if ( NULL == matvar ) {
        fprintf(stderr,"Error creating variable for 'a'\n");
        Mat_Close(matfp);
        return EXIT_FAILURE;
    }

    /* structure index 0 */
    field = Mat_VarCreate(NULL,MAT_C_DOUBLE,MAT_T_DOUBLE,2,dims,x1,0);
    Mat_VarSetStructFieldByName(matvar,"x",0,field);
    field = Mat_VarCreate(NULL,MAT_C_DOUBLE,MAT_T_DOUBLE,2,dims,y1,0);
    Mat_VarSetStructFieldByName(matvar,"y",0,field);
    field = Mat_VarCreate(NULL,MAT_C_DOUBLE,MAT_T_DOUBLE,2,dims,&z1,
                MAT_F_COMPLEX);
    Mat_VarSetStructFieldByName(matvar,"z",0,field);

    /* structure index 1 */
    field = Mat_VarCreate(NULL,MAT_C_DOUBLE,MAT_T_DOUBLE,2,dims,x2,0);
    Mat_VarSetStructFieldByName(matvar,"x",1,field);
    field = Mat_VarCreate(NULL,MAT_C_DOUBLE,MAT_T_DOUBLE,2,dims,y2,0);
    Mat_VarSetStructFieldByName(matvar,"y",1,field);
    field = Mat_VarCreate(NULL,MAT_C_DOUBLE,MAT_T_DOUBLE,2,dims,&z2,
                MAT_F_COMPLEX);
    Mat_VarSetStructFieldByName(matvar,"z",1,field);

    Mat_VarWrite(matfp,matvar,MAT_COMPRESSION_NONE);
    Mat_VarFree(matvar);

    Mat_Close(matfp);
    return EXIT_SUCCESS;
}
```

3 Building matio

3.1 Quick Build Guide

The primary method for building the software is using `configure` followed by `make`. After building, the testsuite can be executed to test the software using `make check` (See Section 3.5 [Testsuite], page 12. The software can be installed using 'make install'. For example,

```
$ tar zxf matio-X.Y.Z.tar.gz
$ cd matio-X.Y.Z
$ ./configure
$ make
$ make check
$ make install
```

3.2 Configure Options

`--enable-mat73=[yes|no]`
: This flag enables/disables the support for version 7.3 MAT files. The option only makes sense if built with HDF5 as support for version 7.3 files will be disabled if HDF5 is not available.

`--enable-extended-sparse=yes`
: Enable extended sparse matrix data types not supported in MATLAB. MATLAB only supports double-precision sparse data. With this flag, matio will read sparse data with other types (i.e. single-precision and integer types).

`--with-matlab=DIR`
: This option specifies the directory (DIR) with the `matlab` program. With this option, the testsuite will check that the MAT files written by matio can be read into MATLAB (see Section Section 3.5 [Testsuite], page 12, for more information about the testsuite).

`--with-zlib=DIR`
: This option specifies the prefix where zlib is installed.

`--with-hdf5=DIR`
: This option specifies the prefix where the HDF5 software is installed.

`--with-default-file-ver=[4|5|7.3]`
: This option sets the default MAT file version that will be used when writing. The default file version is used by the Mat_Create macro and the Mat_CreateVer function when MAT_FT_DEFAULT is used for the version argument.

`--with-libdir-suffix=suffix`
: This option specifies a suffix to apply to library directories when installing and looking for dependent libraries (i.e. HDF5 and zlib). For example, some multi-arch Linux distributions install 64-bit libraries into lib64 and 32-bit libraries into lib.

3.3 CMake build system

The CMake build system is supported as an alternative build system, which usually consists of three steps for configuration, build and installation, for example,

```
$ tar zxf matio-X.Y.Z.tar.gz
$ cd matio-X.Y.Z
$ cmake .
```

```
$ cmake --build .
$ cmake --install .
```

The following matio specific options for building with CMake are available.

`MATIO_USE_CONAN:BOOL=OFF`
: This option enables the Conan package manager to resolve the library dependencies.

`MATIO_DEFAULT_FILE_VERSION:STRING=5`
: This option sets the default MAT file version (4,5,7.3) that will be used when writing.

`MATIO_EXTENDED_SPARSE:BOOL=ON`
: This option enables extended sparse matrix data types not supported in MATLAB.

`MATIO_MAT73:BOOL=ON`
: This flag enables the support for version 7.3 MAT files.

`MATIO_PIC:BOOL=ON`
: This option enables position-independent code (PIC), i.e., compilation with the `-fPIC` flag. It is ignored for Visual Studio builds.

`MATIO_SHARED:BOOL=ON`
: This option builds the matio library as shared object (i.e., a dynamic link library on Windows).

`MATIO_WITH_HDF5:BOOL=ON`
: This option enables CMake to check for availability of the HDF5 library (see section 2.1.2 for information about HDF5).

`MATIO_WITH_ZLIB:BOOL=ON`
: This option enables CMake to check for availability of the zlib library (see section 2.1.1 for information about zlib).

To help CMake find the HDF5 libraries, set environment variable HDF5_DIR to the *cmake/hdf5* directory (containing *hdf5-config.cmake*) inside the HDF5 build or installation directory, or call cmake with `-DHDF5_DIR="dir/to/hdf5/cmake/hdf5`. Alternatively call CMake with `-DCMAKE_PREFIX_PATH="dir/to/hdf5/cmake"`. See the HDF5 instructions for more information. Using *hdf5-config.cmake* is recommended over using CMake's built-in *FindHDF5*, especially for static builds. CMake 3.10 or later is recommended.

3.4 Visual Studio

A visual studio solution is provided as visual_studio/matio.sln. The solution is set up to build a DLL of the matio library (*libmatio.dll*) and *matdump* tool in release mode and assumes HDF5 is available in the directory specified by the HDF5_DIR environment variable. The build was tested with the HDF5 visual studio pre-built Windows binaries including zlib.

3.5 Testsuite

A testsuite is available when building with the GNU autotools. To run the testsuite, First configure and build matio. After building run `make check` to run the testsuite. If matio was built without zlib, the compressed variable tests will be skipped. If built without HDF5, the tests for version 7.3 MAT files will be skipped. If the path to the MATLAB application was not specified (`--with-matlab`), the write tests will fail if matio cannot read the file and skip if matio can read the file. The write tests will pass if MATLAB is available and can also read the file.

To report matio testsuite failures, compress the testsuite.log file in the test sub-directory of the build directory. Upload the compressed log file along with a bug report (see Section 1.4 for information on reporting bugs).

4 MATLAB Variable Structure

4.1 Variable Information

When a MATLAB variable is read or created, all of the information about the variable (e.g. name, dimensions, etc.) are stored in the MATLAB variable structure type `matvar_t`.

`name` Nul-terminated string that is the name of the variable. The name may be NULL (e.g. for elements of a cell-array), so the field should be checked prior to use.

`rank` The number of dimensions of the variable. The minimum rank is 2.

`dims` An array of the number of elements in each dimensions of the variable.

`class_type`
Indicates the class of the variable (e.g. double-precision, structure, cell, etc.).

`data_type`
Indicates the type of the data stored in the `data` field of the MATLAB variable structure.

`isComplex`
is non-zero if the variable is a complex-valued numeric array.

`isLogical`
is non-zero of the variable should be interpreted as logical (i.e. zero for false, non-zero for true).

`isGlobal` is non-zero if the variable should be a global variable. In MATLAB a global variable is available in all scopes (e.g. base workspace, function, etc.)

4.1.1 Sparse Matrix Variables

If a variable's class type is sparse, the `data` field of the MATLAB variable structure is a pointer to the sparse matrix structure `mat_sparse_t`. The sparse matrix structure stores the non-zero elements of the matrix in compressed column format.

4.1.2 Structure Variables

If the MATLAB variable structure's `class_type` is `MAT_C_STRUCT`, the `data_type` field should be `MAT_T_STRUCT`. The `data` field of the variable structure is a pointer to an array of `matvar_t *`. The length of the array is $numel \times nfields$ where `numel` is the number of elements in the structure array (product of dimensions array), and $nfields$ is the number of fields in the structure. The order of the variables in the array is first by field, and then by structure index. For example, for a 2×1 structure array with 3 fields *field1*, *field2*, and *field3*, `data` field of the structure variable is ordered as:

```
s(1).field1
s(1).field2
s(1).field3
s(2).field1
s(2).field2
s(2).field3
```

4.1.3 Cell Variables

If the MATLAB variable structure's `class_type` is `MAT_C_CELL`, the `data_type` field should be `MAT_T_CELL`. The `data` field of the variable structure is a pointer to an array of `matvar_t *`. The length of the array is the product of the dimensions array. Each element of the cell array can be a different type.