Libmtx User Guide

for version 0.3.0, 24 November 2022

James D. Trotter (james@simula.no)

This manual describes how to install and use Libmtx, version 0.3.0, 24 November 2022, a C library and collection of utility programs for working with objects in the Matrix Market file format, including vectors, dense matrices and sparse matrices.

Table of Contents

Copying Conditions

1 Introduction to Libmtx

Libmtx is a C library and a collection of utility programs for sparse linear algebra, including some of the most common linear algebra operations, as well as various preprocessing tasks, such as sorting, reordering and partitioning of sparse matrices and vectors. Libmtx enables these operations to be carried out directly from the command line, while also allowing for the underlying data to be easily exchanged. To do so, these tools rely heavily on the Matrix Market file format, a commonly used, human-readable text file format for storing both dense and sparse matrices and vectors.

Matrices and vectors are basic objects of numerical linear algebra and thus appear in most fields of science. In particular, sparse matrices, where most of the entries are zero, are especially important for representing sparse graphs and for solving partial differential equations, to name just two examples. Sparse matrices require methods that are different from those used with dense matrices. But, in return, they often result in significantly fewer arithmetic operations to be carried out, as well as lower memory footprint and memory bandwidth usage.

1.1 Background

The *Matrix Market* ([National Institute of Standards and Technology [NIST] (2007)], page 65) is a repository of test data that has been used to study and compare various algorithms for numerical linear algebra. It was originally designed and developed by R. Boisvert, R. Pozo, K. Remington, R. Barrett and J.J. Dongarra, and first became available online in February 1996.

The Matrix Market data set was made available in the Matrix Market file format, which was initially described in the paper *The Matrix Market Formats: Initial Design* ([R.F. Boisvert, R. Pozo and K. Remington (1996)], page 65). There is also a reference software for reading and writing Matrix Market files, known as *mmio*, or the ANSI C library for Matrix Market I/O, ([National Institute of Standards and Technology [NIST] (2000)], page 65).

More recently, the *SuiteSparse Matrix Collection* (`https://sparse.tamu.edu/`) has become a large, widely used repository of sparse matrices from numerous application domains. These matrices are also distributed in the Matrix Market format.

1.2 How to Use This Manual

To familiarise yourself with the Matrix Market file format, read Chapter 3 [Matrix Market format], page 4. If you need to install Libmtx yourself, you should read Chapter 2 [Installing Libmtx], page 3. If you want to learn how to use the Libmtx C library to work with Matrix Market files, refer to Chapter 4 [Matrix Market files], page 9. To learn how to use matrices and vectors for basic linear algebra, refer to Chapter 6 [Matrices and vectors], page 41. If you are working in a distributed-memory setting and would like to use distributed matrices and vectors, see Chapter 5 [Distributed Matrix Market files], page 31. If you would like to get an overview of the command-line programs included in Libmtx, then you should read Chapter 7 [Commands], page 57.

2 Installing Libmtx

Libmtx uses GNU Autotools, which means that a basic build on a UNIX-like system can be done by running the commands

```
./configure
make
```

To run some tests, use the command

```
make check
```

Finally, install the programs and library with

```
make install
```

If you experience problems, please report them to `james@simula.no`. See Section 2.2 [Reporting bugs], page 3, for information on what to include in a bug report.

2.1 Optional dependencies

Some features of Libmtx are optional and will only be available if some of the following dependencies are provided:

- *MPI* is needed for distributed-memory computing.
- *BLAS* may be used to accelerate various linear algebra operations. Different implementations may be used, such as the netlib reference BLAS, OpenBLAS, BLIS, and so on.
- *zlib* (‘`https://www.zlib.net/`’) (version 1.2.7.1 or newer) is needed for reading and writing gzip-compressed Matrix Market files.
- *libpng* (‘`http://www.libpng.org/pub/png/libpng.html`’) is needed for writing PNG images of matrix sparsity patterns.

2.2 Reporting bugs

To report bugs, suggest enhancements or otherwise discuss Libmtx, please send electronic mail to `james@simula.no`.

For bug reports, please include enough information for the maintainers to reproduce the problem. Generally speaking, that means:

- The version numbers of Libmtx (which you can find by running ‘`mtxinfo --version`’) and any other program(s) or manual(s) involved.
- Hardware and operating system names and versions.
- The contents of any input files necessary to reproduce the bug.
- The expected behavior and/or output.
- A description of the problem and samples of any erroneous output.
- Options you gave to `configure` other than specifying installation directories.
- Anything else that you think would be helpful.

When in doubt whether something is needed or not, include it. It’s better to include too much than to leave out something important.

3 Matrix Market format

This chapter describes the Matrix Market file format. Matrix Market files, which are usually given a `.mtx` suffix, are plain text ASCII files consisting of the following four parts:

1. a *header line*,
2. an optional section containing *comment lines*,
3. a *size line*, and
4. *data lines* for nonzero matrix or vector entries.

Here is an example of a Matrix Market file, which describes a rectangular, sparse matrix with 4 rows, 5 columns and 7 real-valued, nonzero entries:

```
%%MatrixMarket matrix coordinate real general
% Sparse matrix example
4 5 7
1 2 2.0
3 3 3.0
1 1 1.0
4 5 1.0
2 2 1.0
4 1 -1.0
4 4 2.0
```

The above example is equivalent to the following 4-by-5 matrix:

```
 1 2 0 0 0
 0 1 0 0 0
 0 0 3 0 0
-1 0 0 2 1
```

The different parts of a Matrix Market file are described in the following subsections.

3.1 Header line

The *header line* of a Matrix Market file is on the form:

`%%MatrixMarket` *`object format field symmetry`*

This line always begins with the identifier '`%%MatrixMarket`' to indicate that the file is in the Matrix Market format. The four fields that follow describe different properties of the Matrix Market object, such as whether it is a matrix or a vector, and if it is stored in sparse or dense form.

1. *`object`* is either '`matrix`' or '`vector`'.
2. *`format`* is '`array`' if the data is stored in a dense format, or '`coordinate`' if it is stored in a sparse format.
3. *`field`* indicates the field to which the matrix (or vector) values belong, and may be one of the following: '`real`', '`complex`', '`integer`' or '`pattern`'.
4. *`symmetry`* specifies the symmetry properties of a matrix, and may be one of the following: '`general`', '`symmetric`', '`skew-symmetric`' or '`hermitian`'.

The *field* value of the header is used as follows:

- For real-valued matrices or vectors, *field* is '`real`' and values are stored as decimal numbers.
- For complex matrices or vectors, *field* is '`complex`' and values are stored as pairs of decimal numbers, comprising the real and imaginary parts of a complex number.
- For integer-valued matrices or vectors, *field* is '`integer`' and values are stored as integers.
- For binary-valued sparse matrices or vectors, *field* is '`pattern`'. No values are stored for nonzero matrix or vector entries. Instead, only the locations of nonzeros are stored.

The *symmetry* value of the header is ignored if *object* is '`vector`'. However, if *object* is '`matrix`', then it is used as follows:

- For a general, unsymmetric matrix, *symmetry* is '`general`'. Every nonzero matrix entry is stored explicitly.
- For a symmetric matrix, (i.e., a matrix that is equal to its transpose), *symmetry* is '`symmetric`'. The matrix must be square. If *format* is '`array`', then only the lower or upper triangular part of the matrix is stored explicitly. (Note that there is no indication in the Matrix Market file regarding which part is stored, but Libmtx assumes by default that the lower triangular part is stored in row major order.) Otherwise, if *format* is '`coordinate`', then values may be explicitly represented from the lower or upper triangular part of the matrix, or both. For each off-diagonal entry '`(i,j)`' that is given explicitly, there is assumed to be an identical value present at location '`(j,i)`'.
- For a skew-symmetric matrix, (i.e., a matrix that is equal to the negative of its transpose), *symmetry* is '`skew-symmetric`'. The matrix must be square, but skew symmetry implies that diagonal entries are zero. Therefore, if *format* is '`array`', then only the strictly lower or upper triangular part of the matrix is stored explicitly. (Note that there is no indication in the Matrix Market file regarding which part is stored, but Libmtx assumes by default that the strictly lower triangular part is stored in row major order.) Otherwise, if *format* is '`coordinate`', then values may be explicitly represented from the strictly lower or upper triangular part of the matrix, or both. Nonzero diagonal entries are not allowed. For each off-diagonal entry '`(i,j)`' that is given explicitly, there is assumed to be an equal value with the opposite sign present at location '`(j,i)`'.
- For a Hermitian matrix, (i.e., a matrix that is equal to its conjugate transpose), *symmetry* is '`hermitian`'. The matrix must be square. If *format* is '`array`', then only the lower or upper triangular part of the matrix is stored explicitly. (Note that there is no indication in the Matrix Market file regarding which part is stored, but Libmtx assumes by default that the lower triangular part is stored in row major order.) Otherwise, if *format* is '`coordinate`', then values may be explicitly represented from the lower or upper triangular part of the matrix, or both. For each off-diagonal entry '`(i,j)`' that is given explicitly, there is assumed to be an equal, complex conjugated value present at location '`(j,i)`'.

3.2 Comment lines

Matrix Market files allow for an optional section of comments, which are ignored when processing the file. If present, comments must follow immediately after the header line and

right before the size line. Each comment line begins with the character '`%`' and continues until the end of the line.

3.3 Size line

The *size line* describes the size of the object stored in a Matrix Market file, and it depends both on the *`object`* and *`format`* values in the header.

- For *dense vectors* (i.e., *`object`* is '`vector`' and *`format`* is '`array`'), the size line is on the form

  ```
  M
  ```

  Here, `M` is an integer denoting the size or number of elements in the vector. In the case of a column vector, `M` is the number of rows. Alternatively, for a row vector, `M` is the number of columns. In any case, it is also the number of lines or lines in the data section of the Matrix Market file.

- For *sparse vectors* (i.e., *`object`* is '`vector`' and *`format`* is '`coordinate`'), the size line is on the form

  ```
  M NNZ
  ```

  As above, `M` is an integer denoting the number of rows in a column vector or the number of columns in a row vector. In addition, `NNZ` is an integer denoting the number of lines or (nonzero) entries that are explicitly stored in the data section of the Matrix Market file.

- For *dense matrices* (i.e., *`object`* is '`matrix`' and *`format`* is '`array`'), the size line is on the form

  ```
  M N
  ```

  The integers `M` and `N` denote the number of matrix rows and columns, respectively. In the case of a general, unsymmetric matrix (*`symmetry`* is '`general`'), there are $\texttt{M} \times \texttt{N}$ matrix lines or entries in the data section of the Matrix Market file. However, if *`symmetry`* is '`symmetric`' or '`hermitian`', then `M` and `N` must be equal, and there are $\texttt{N} \times (\texttt{N}+1)/2$ lines or entries in the data section. Finally, if *`symmetry`* is '`skew-symmetric`', then `M` and `N` must again be equal, and there are $\texttt{N} \times (\texttt{N}-1)/2$ lines or entries in the data section.

- For *sparse matrices* (i.e., *`object`* is '`matrix`' and *`format`* is '`coordinate`'), the size line is on the form

  ```
  M N NNZ
  ```

  Here, `M` and `N` denote the number of rows and columns in the matrix, whereas `NNZ` denotes the number of (nonzero) matrix entries that are explicitly stored in the data section of the Matrix Market file.

3.4 Data lines

The final section of a Matrix Market file contains *data lines* for each (nonzero) matrix or vector entry. The number of data lines depends on the matrix or vector format and size, as described in the previous section.

It is assumed that dense matrix entries are stored in row major order. However, for sparse matrices and vectors, the nonzero entries may appear in any order. Moreover, for

any particular location in a sparse matrix or vector, there may be more than one data line or entry. In this case, the value associated with a particular matrix or vector location is obtained as the sum of all nonzero values for that location. The procedure of adding together values for duplicate entries is sometimes referred to as *assembly* (see also Section 4.5.2 [Sort compact and assemble], page 20).

The format of data lines in a Matrix Market file depends on the *object*, *format* and *field* values in the header (see Section 3.1 [Header line], page 4). The different data line formats are described in detail below.

First, dense matrices or vectors with real or integer values (that is, *format* is 'array' and *field* is 'real' or 'integer'), have data lines on the form

```
a
```

where a is a decimal number if *field* is 'real', or an integer if *field* is 'integer'.

Dense, complex-valued matrices or vectors (i.e., *format* is 'array' and *field* is 'complex'), have data lines on the form

```
a b
```

Here a and b are decimal numbers denoting the real and imaginary parts of the complex number $z = \mathtt{a} + \mathtt{b}i$, where i is the imaginary unit.

A sparse matrix, (that is, *object* is 'matrix' and *format* is 'coordinate'), has data lines on one of three different forms depending on the *field* value. Note that indexing of sparse matrix and vector entries is 1-based.

- If *field* is 'real' or 'integer', then data lines are written as:

  ```
  i j a
  ```

  where i and j are integers denoting the row and column index of a nonzero entry and a is a decimal number denoting its value.
- If *field* is 'complex', then data lines are on the form

  ```
  i j a b
  ```

  where i and j again denote the row and column index. Here, the decimal numbers a and b are the real and imaginary parts, respectively, of the complex number $z = \mathtt{a} + \mathtt{b}i$, where i is the imaginary unit.
- If *field* is 'pattern', then each data line consists of two integers

  ```
  i j
  ```

  denoting the row and column index of a nonzero entry.

For a sparse vector, (that is, *object* is 'vector' and *format* is 'coordinate'), the data lines are similar to those of a sparse matrix, but the column index is omitted. More specifically, data lines are on one of three different forms depending on the *field* value.

- If *field* is 'real' or 'integer', then data lines are written as:

  ```
  i a
  ```

  where i is an integer denoting the index of a nonzero entry and a is a decimal number denoting its value.
- If *field* is 'complex', then data lines are on the form

  ```
  i a b
  ```

The integer `i` is the nonzero index, whereas `a` and `b` are decimal numbers representing the real and imaginary part, respectively, of the complex number $z = \mathtt{a} + \mathtt{b}i$, where i is the imaginary unit.

- If *`field`* is '`pattern`', then each data line consists of a single integer

  ```
  i
  ```

  denoting the index of a nonzero entry.

4 Matrix Market files

This chapter explains in detail how to use the Libmtx C library to work with Matrix Market files. For now, we are concerned with using a single, shared-memory machine or node. To distribute Matrix Market files across multiple processes using MPI for large-scale, parallel computations, see Chapter 5 [Distributed Matrix Market files], page 31.

For any user-facing types and functions, Libmtx uses the convention of prefixing names with `mtx` or `mtxfile`. This makes it easier to avoid possible name clashes with other code when using Libmtx.

4.1 Error handling

Functions in the Libmtx C library generally return a status code that either indicates success (represented by the status code '`MTX_SUCCESS`') or a specific error. Valid error codes correspond to values of the type `enum mtxerror`, which is defined in the file `libmtx/error.h` along with some error handling functions.

The function `mtxstrerror` can be used to give a string containing a description of a given error code.

```
const char * mtxstrerror(int err);
```

Here, the integer `err` should correspond to one of the error codes from the `mtxerror` enum type.

`mtxstrerror` is typically used as shown in the example below.

```
struct mtxfile mtx;
int64_t lines_read = 0;
FILE * f = fopen("test.mtx", "r");
int err = mtxfile_fread(
    &mtx, mtx_double, f,
    &lines_read, NULL, 0, NULL);
if (err) {
    fprintf(stderr, "test.mtx:%d: %s\n",
            lines_read+1, mtxstrerror(err));
    fclose(f);
}
```

If `mtxfile_fread` returns an error, for example, '`MTX_ERR_INVALID_MTX_OBJECT`', then the following message will be printed:

```
test.mtx:1: invalid Matrix Market object
```

4.2 Data structures

This section describes the basic data types used to represent objects in the Matrix Market file format.

The file `libmtx/mtxfile/mtxfile.h` defines the `struct mtxfile` type. The main purpose of `struct mtxfile` is to represent objects in Matrix Market format, including dense and sparse matrices and vectors with real, complex, integer or binary values.

The `mtxfile` struct is very close to the ASCII representation of a Matrix Market file. However, instead of ASCII strings, the header values are converted to appropriate enum

types, row and column offsets are represented as 64-bit signed integers, and (depending on the underlying field and desired precision) matrix or vector values are converted to 32- or 64-bit integers or floating point numbers.

The definition of the `mtxfile` struct is shown below.

```
struct mtxfile {
    struct mtxfileheader header;
    struct mtxfilecomments comments;
    struct mtxfilesize size;
    enum mtxprecision precision;
    int64_t datasize;
    union mtxfiledata data;
};
```

Roughly speaking, the `mtxfile` struct consists of four parts: header information, comment lines, size information and data. In addition, the `precision` struct member describes the precision used to store matrix or vector values, and `datasize` stores the number of entries in the array of matrix or vector values.

The following sections provide detailed explanations of the `mtxfile` struct members.

4.2.1 Header

The `mtxfileheader` data type is used to represent the header line of a Matrix Market file.

```
struct mtxfileheader {
    enum mtxfileobject object;
    enum mtxfileformat format;
    enum mtxfilefield field;
    enum mtxfilesymmetry symmetry;
};
```

The four enum types, `mtxfileobject`, `mtxfileformat`, `mtxfilefield` and `mtxfilesymmetry` are used to represent values that appear in the Matrix Market header (see Section 3.1 [Header line], page 4). The meaning of the values associated with these types is described in detail in Chapter 3 [Matrix Market format], page 4.

```
enum mtxfileobject {
    mtxfile_matrix,    /* matrix */
    mtxfile_vector     /* vector */
};

enum mtxfileformat {
    mtxfile_array,     /* dense matrix or vector */
    mtxfile_coordinate /* sparse matrix or vector */
};

enum mtxfilefield {
    mtxfile_real,    /* real coefficients */
    mtxfile_complex, /* complex coefficients */
    mtxfile_integer, /* integer coefficients */
    mtxfile_pattern  /* boolean coefficients (sparsity pattern) */
```

```
};

enum mtxfilesymmetry {
    mtxfile_general,        /* general, unsymmetric matrix or vector */
    mtxfile_symmetric,      /* symmetric matrix */
    mtxfile_skew_symmetric, /* skew-symmetric matrix */
    mtxfile_hermitian       /* Hermitian matrix */
};
```

4.2.2 Comments

Comment lines are stored in a doubly linked list data structure, `struct mtxfilecomments`.

```
struct mtxfilecomments {
    struct mtxfilecomment * root;
};
```

Each comment line in a list is represented with `struct mtxfilecomment`.

```
struct mtxfilecomment {
    struct mtxfilecomment * prev;
    struct mtxfilecomment * next;
    char * comment_line;
};
```

Here, `comment_line` is a non-empty, null-terminated string that must begin with the character '`%`'.

4.2.3 Size

The size information in `struct mtxfilesize` includes the number of rows, columns and nonzeros in the underlying matrix or vector.

```
struct mtxfilesize {
    int64_t num_rows;
    int64_t num_columns;
    int64_t num_nonzeros;
};
```

In the case of a matrix, `num_rows` and `num_columns` are non-negative integers representing the number of rows and columns in the matrix, respectively. By convention, vectors are represented as column vectors. As a result, `num_rows` is equal to the number of vector elements, whereas `num_columns` is not used and is therefore set to '`−1`'.

For matrices and vectors in coordinate format, `num_nonzeros` is the number of entries explicitly stored in the data section of the Matrix Market file. For matrices and vectors in array format, `num_nonzeros` is not used and is therefore set to '`−1`'.

Given a valid `struct mtxfilesize`, the number of lines in the data section of a Matrix Market file can be obtained by calling `mtxfilesize_num_data_lines`.

```
int mtxfilesize_num_data_lines(
    const struct mtxfilesize * size,
    enum mtxfile_symmetry symmetry,
    int64_t * num_data_lines);
```

The number of data lines in a Matrix Market file with the given size line and symmetry is stored in the integer pointed to by `num_data_lines`. More specifically, it is set to

- `num_nonzeros`, if `num_nonzeros` is non-negative; or
- `num_rows*num_columns`, if `num_rows` and `num_columns` are both non-negative and `symmetry` is '`mtxfile_general`'; or
- `num_rows*(num_rows+1)/2`, if `num_rows` is non-negative and equal to `num_columns`, and `symmetry` is '`mtxfile_symmetric`' or '`mtxfile_hermitian`'; or
- `num_rows*(num_rows-1)/2`, if `num_rows` is non-negative and equal to `num_columns`, and `symmetry` is '`mtxfile_skew_symmetric`'; or
- `num_rows`, if `num_rows` is non-negative.

4.2.4 Precision

Matrix Market files represent matrix or vector values as integers or decimal numbers in ASCII text. In this form, there is no limit or prescribed precision associated with the values that are stored, making the format is quite flexible. In practice, however, it is necessary to convert matrix and vector values to fixed-precision integer or floating point types. For this purpose, the `mtxprecision` enum type can be used to choose between single (32-bit) and double (64-bit) precision.

```
enum mtxprecision {
    mtx_single,       /* single (32-bit) precision */
    mtx_double,       /* double (64-bit) precision */
};
```

4.2.5 Data

Matrix or vector values are stored in an array whose type depends on the object, format and field of the Matrix Market file, as well as the chosen precision. The appropriate array can therefore be accessed through the `mtxfiledata` union type, which is shown below. The length of the array (which depends on the size and symmetry of the matrix or vector, see Section 4.2.3 [Size], page 11) is given by the `datasize` member of `struct mtxfile`.

```
union mtxfiledata {
    /* Array formats */
    float * array_real_single;
    double * array_real_double;
    float (* array_complex_single)[2];
    double (* array_complex_double)[2];
    int32_t * array_integer_single;
    int64_t * array_integer_double;

    /* Matrix coordinate formats */
    struct mtxfile_matrix_coordinate_real_single *
        matrix_coordinate_real_single;
    struct mtxfile_matrix_coordinate_real_double *
        matrix_coordinate_real_double;
    struct mtxfile_matrix_coordinate_complex_single *
        matrix_coordinate_complex_single;
```

```
        struct mtxfile_matrix_coordinate_complex_double *
            matrix_coordinate_complex_double;
        struct mtxfile_matrix_coordinate_integer_single *
            matrix_coordinate_integer_single;
        struct mtxfile_matrix_coordinate_integer_double *
            matrix_coordinate_integer_double;
        struct mtxfile_matrix_coordinate_pattern *
            matrix_coordinate_pattern;

        /* Vector coordinate formats */
        struct mtxfile_vector_coordinate_real_single *
            vector_coordinate_real_single;
        struct mtxfile_vector_coordinate_real_double *
            vector_coordinate_real_double;
        struct mtxfile_vector_coordinate_complex_single *
            vector_coordinate_complex_single;
        struct mtxfile_vector_coordinate_complex_double *
            vector_coordinate_complex_double;
        struct mtxfile_vector_coordinate_integer_single *
            vector_coordinate_integer_single;
        struct mtxfile_vector_coordinate_integer_double *
            vector_coordinate_integer_double;
        struct mtxfile_vector_coordinate_pattern *
            vector_coordinate_pattern;
    };
```

For a matrix or vector in array format, values are stored in the union member `array_`*`field_precision`*, which is an array of type

- `float` or `double` if `field` is '`mtxfile_real`' and `precision` is '`mtx_single`' or '`mtx_double`', respectively; or
- `float (*)[2]` or `double (*)[2]` if `field` is '`mtxfile_complex`' and `precision` is '`mtx_single`' or '`mtx_double`', respectively; or
- `int32_t` or `int64_t` if `field` is '`mtxfile_integer`' and `precision` is '`mtx_single`' or '`mtx_double`', respectively.

Note that the type used for complex values, e.g., `float (*)[2]`, denotes a pointer to an array of size `2`. Thus, complex values are accessed using two-dimensional array indexing, e.g., `array_complex_single[i][j]`. The first index, `i`, indicates the position in array of complex vector or matrix values, while the second index, `j`, is `0` for the real part and `1` for the imaginary part of the complex number. Also, note that matrices or vectors in array format must not have `field` set to '`mtxfile_pattern`'.

For a matrix in coordinate format, values are stored in the union member `matrix_coordinate_`*`field_precision`*, which is an array of type `struct mtxfile_matrix_`*`field_precision`*, where *`field`* is the field associated with the matrix and *`precision`* is the chosen precision. The struct data types for each combination of field and precision are shown below.

```
    struct mtxfile_matrix_coordinate_real_single {
        int64_t i, j; /* row and column index */
```

```
    float a;      /* nonzero value */
};

struct mtxfile_matrix_coordinate_real_double {
    int64_t i, j; /* row and column index */
    double a;     /* nonzero value */
};

struct mtxfile_matrix_coordinate_complex_single {
    int64_t i, j; /* row and column index */
    float a[2];   /* real and imaginary part of nonzero value */
};

struct mtxfile_matrix_coordinate_complex_double {
    int64_t i, j; /* row and column index */
    double a[2];  /* real and imaginary part of nonzero value */
};

struct mtxfile_matrix_coordinate_integer_single {
    int64_t i, j; /* row and column index */
    int32_t a;    /* nonzero value */
};

struct mtxfile_matrix_coordinate_integer_double {
    int64_t i, j; /* row and column index */
    int64_t a;    /* nonzero value */
};

struct mtxfile_matrix_coordinate_pattern {
    int64_t i, j; /* row and column index */
};
```

Note that there is no precision associated with matrices whose `field` is '`mtxfile_pattern`'. Instead, the presence of a nonzero value with row index `i` and column index `j` indicates that the matrix has a value of `1` at position `(i,j)`.

Vectors in coordinate format are treated similarly to matrices, except that the column index is omitted. More specifically, vector values are stored in the union member `vector_coordinate_`*`field_precision`*, which is an array of type `struct mtxfile_vector_`*`field_precision`*, where *`field`* is the field associated with the vector and *`precision`* is the chosen precision. The struct data types for each combination of field and precision are shown below.

```
struct mtxfile_vector_coordinate_real_single {
    int64_t i;   /* row index */
    float a;     /* nonzero value */
};

struct mtxfile_vector_coordinate_real_double {
    int64_t i;   /* row index */
```

```
    double a;    /* nonzero value */
};

struct mtxfile_vector_coordinate_complex_single {
    int64_t i;   /* row index */
    float a[2];  /* real and imaginary part of nonzero value */
};

struct mtxfile_vector_coordinate_complex_double {
    int64_t i;   /* row index */
    double a[2]; /* real and imaginary part of nonzero value */
};

struct mtxfile_vector_coordinate_integer_single {
    int64_t i;   /* row index */
    int32_t a;   /* nonzero value */
};

struct mtxfile_vector_coordinate_integer_double {
    int64_t i;   /* row index */
    int64_t a;   /* nonzero value */
};

struct mtxfile_vector_coordinate_pattern {
    int64_t i;   /* row index */
};
```

Note that there is no precision associated with vectors whose `field` is '`mtxfile_pattern`'. Instead, the presence of a nonzero value with index `i` indicates that the vector has a value of `1` at position `i`.

4.3 Reading and writing Matrix Market files

In most cases, matrices and vectors are obtained by reading from a file in Matrix Market format. These files are typically named with a `.mtx` extension, so we refer to them here as `mtx` files. This section describes how to use Libmtx to read or write matrices and vectors to and from files in Matrix Market format.

4.3.1 Reading Matrix Market files

To read an `mtx` file from a `FILE` stream, use the function `mtxfile_fread`:

```
int mtxfile_fread(
    struct mtxfile * mtxfile,
    enum mtxprecision precision,
    FILE * f,
    int64_t * lines_read,
    int64_t * bytes_read,
    size_t line_max,
    char * linebuf);
```

If successful, '`MTX_SUCCESS`' is returned, and `mtxfile` will contain the matrix or vector. The user is responsible for calling `mtxfile_free` to free any storage allocated by `mtxfile_fread`. Otherwise, if `mtxfile_fread` fails, an error code is returned and `lines_read` and `bytes_read` are used to indicate the line number and byte of the Matrix Market file at which the error was encountered. `lines_read` and `bytes_read` are ignored if they are set to '`NULL`'.

Moreover, `precision` is used to choose the precision for storing the values of matrix or vector entries, as described in Section 4.2.4 [Precision], page 12. If `linebuf` is not '`NULL`', then it must point to an array that can hold a null-terminated string whose length (including the terminating null-character) is at most `line_max`. This buffer is used for reading lines from the stream. Otherwise, if `linebuf` is '`NULL`', a temporary buffer is allocated and used, and the maximum line length is determined by calling `sysconf()` with `_SC_LINE_MAX`.

If Libmtx is built with zlib support, then `mtxfile_gzread` can be used to read gzip-compressed `mtx` files.

```
int mtxfile_gzread(
    struct mtxfile * mtxfile,
    enum mtxprecision precision,
    gzFile f,
    int64_t * lines_read,
    int64_t * bytes_read,
    size_t line_max,
    char * linebuf);
```

For convenience, the function `mtxfile_read` can be used to read an `mtx` file from a given path.

```
int mtxfile_read(
    struct mtxfile * mtxfile,
    enum mtxprecision precision,
    const char * path,
    bool gzip,
    int64_t * lines_read,
    int64_t * bytes_read);
```

The file is assumed to be gzip-compressed if `gzip` is '`true`', and uncompressed otherwise. If `path` is '`-`', then the standard input stream is used.

4.3.2 Writing Matrix Market files

To write an `mtx` file to a `FILE` stream, use the function `mtxfile_fwrite`:

```
int mtxfile_fwrite(
    const struct mtxfile * mtxfile,
    FILE * f,
    const char * fmt,
    int64_t * bytes_written);
```

If successful, '`MTX_SUCCESS`' is returned, and the matrix or vector was written to the `FILE` stream. Moreover, if it is not '`NULL`', then the number of bytes written to the stream is returned in `bytes_written`.

The `fmt` argument may optionally be used to specify a format string for outputting of numerical values. If `fmt` is '`NULL`', then the format specifier '`%g`' is used to print floating point numbers with enough digits to ensure correct round-trip conversion from decimal text and back. Otherwise, the given format string is used to print numerical values. The format string follows the conventions of `printf`. If the field of `mtxfile` is '`mtxfile_real`' or '`mtxfile_complex`', then the format specifiers '`%e`', '`%E`', '`%f`', '`%F`', '`%g`' or '`%G`' may be used. If the field is '`mtxfile_integer`', then the format specifier must be '`%d`'. The format string is ignored if the field is '`mtxfile_pattern`'. Field width and precision may be specified (e.g., '`%3.1f`'), but variable field width and precision (e.g., '`%*.*f`') or length modifiers (e.g., '`%Lf`') are not allowed.

Note that the locale is temporarily changed to `"C"` to ensure that locale-specific settings, such as the type of decimal point, do not affect output.

If Libmtx is built with zlib support, then `mtxfile_gzwrite` can be used to write gzip-compressed `mtx` files.

```
int mtxfile_gzwrite(
    const struct mtxfile * mtxfile,
    gzFile f,
    const char * fmt,
    int64_t * bytes_written);
```

For convenience, the function `mtxfile_write` can be used to write an `mtx` file to a given path.

```
int mtxfile_write(
    const struct mtxfile * mtxfile,
    const char * path,
    bool gzip,
    const char * fmt,
    int64_t * bytes_written);
```

The file is written as a gzip-compressed stream if `gzip` is '`true`', and uncompressed otherwise. If `path` is '`-`', then the standard output stream is used.

4.4 Creating Matrix Market files

This section covers a number of functions that are provided to construct matrices and vectors in Matrix Market format.

Routines for constructing matrices and vectors typically allocate their own storage for matrix or vector data. Therefore, once a user is finished with an object of type `struct mtxfile`, they should free any allocated storage by calling `mtxfile_free`:

```
void mtxfile_free(struct mtxfile * mtxfile);
```

In the following subsections, we describe functions for allocating matrices and vectors when the size is known, but the values of the matrix or vector entries are not given. In this case, storage is allocated for data, but initialising the data is left to the user. (See, for example, Section 4.4.3 [Setting matrix and vector values], page 20.) In addition, Libmtx provides functions for when both the size and the matrix or vector entries are provided directly by the user. In this case, storage is allocated and the provided data is copied to the newly allocated storage.

The function `mtxfile_alloc` can be used to allocate storage for a Matrix Market file with specified header line, comment lines, size line and precision.

```
int mtxfile_alloc(
    struct mtxfile * mtxfile,
    const struct mtxfileheader * header,
    const struct mtxfilecomments * comments,
    const struct mtxfilesize * size,
    enum mtxprecision precision);
```

The underlying matrix or vector values are not initialised, and it is therefore up to the user to initialise them.

To allocate storage for a copy of an existing Matrix Market file, the function `mtxfile_alloc_copy` may be used.

```
int mtxfile_alloc_copy(
    struct mtxfile * dst,
    const struct mtxfile * src);
```

Although storage is allocated for the underlying matrix or vector values, the data is not initialised. It is therefore up to the user to initialise the matrix or vector values.

If, on the other hand, an exact copy of an existing Matrix Market file is needed, including the matrix or vector values, then the function `mtxfile_init_copy` can be used.

```
int mtxfile_init_copy(
    struct mtxfile * dst,
    const struct mtxfile * src);
```

4.4.1 Creating Matrix Market files in array format

The functions `mtxfile_alloc_matrix_array` and `mtxfile_alloc_vector_array` can be used to allocate storage for matrices and vectors in array format.

```
int mtxfile_alloc_matrix_array(
    struct mtxfile * mtxfile,
    enum mtxfilefield field,
    enum mtxfilesymmetry symmetry,
    enum mtxprecision precision,
    int64_t num_rows,
    int64_t num_columns);

int mtxfile_alloc_vector_array(
    struct mtxfile * mtxfile,
    enum mtxfilefield field,
    enum mtxprecision precision,
    int64_t num_rows);
```

The field and precision must be specified, and storage is allocated appropriately. (For matrices, the symmetry must also be specified.) The matrix or vector values are not initialised, and it is therefore up to the user to initialise them.

If the matrix or vector values are already known, the functions `mtxfile_init_`*`object`*`_array_`*`field`*`_`*`precision`* can be used, where *`object`*, *`field`* and *`precision`* are the appropriate object type (i.e., '`matrix`' or '`vector`'), field (i.e., '`real`', '`complex`' or '`integer`')

and precision (i.e., '`single`' or '`double`'). For example, a matrix in array format with real, double precision floating point coefficients is initialised with `mtxfile_init_matrix_array_real_double`:

```
int mtxfile_init_matrix_array_real_double(
    struct mtxfile * mtxfile,
    enum mtxfilesymmetry symmetry,
    int64_t num_rows, int64_t num_columns,
    const double * data);
```

Similarly, a vector in array format with 32-bit integer values is initialised with `mtxfile_init_vector_array_integer_single`:

```
int mtxfile_init_vector_array_integer_single(
    struct mtxfile * mtxfile,
    int64_t num_rows,
    const int32_t * data);
```

4.4.2 Creating Matrix Market files in coordinate format

To allocate storage for a matrix in coordinate format, the function `mtxfile_alloc_matrix_coordinate` is used.

```
int mtxfile_alloc_matrix_coordinate(
    struct mtxfile * mtxfile,
    enum mtxfilefield field,
    enum mtxfilesymmetry symmetry,
    enum mtxprecision precision,
    int64_t num_rows, int64_t num_columns,
    int64_t num_nonzeros);
```

The field, symmetry and precision must be specified, along with the matrix dimensions and the number of nonzero matrix entries to allocate storage for.

Similarly, a vector in coordinate format can be allocated with `mtxfile_alloc_vector_coordinate`.

```
int mtxfile_alloc_vector_coordinate(
    struct mtxfile * mtxfile,
    enum mtxfilefield field,
    enum mtxprecision precision,
    int64_t num_rows, int64_t num_nonzeros);
```

The field and precision must be given, along with the number of rows in the vector and the number of nonzero vector entries to allocate storage for.

To allocate a matrix or vector and at the same time initialise the nonzero matrix entries, the function `mtxfile_init_`*`object`*`_coordinate_`*`field_precision`* can be used, where *`object`*, *`field`* and *`precision`* are the desired object (i.e., '`matrix`' or '`vector`'), field (i.e., '`real`', '`complex`', '`integer`' or '`pattern`') and precision (i.e., '`single`' or '`double`'). For example, a matrix in coordinate format with real, double precision floating point coefficients is allocated with `mtxfile_init_matrix_coordinate_real_double`:

```
int mtxfile_init_matrix_coordinate_real_double(
    struct mtxfile * mtxfile,
```

```
    enum mtxfilesymmetry symmetry,
    int64_t num_rows, int64_t num_columns,
    int64_t num_nonzeros,
    const struct mtxfile_matrix_coordinate_real_double * data);
```

The matrix values are copied from the `data` array.

To give another example, a vector in coordinate format with 32-bit integer values is created and initialised with the function `mtxfile_init_matrix_coordinate_integer_single`:

```
int mtxfile_init_vector_coordinate_integer_single(
    struct mtxfile * mtxfile,
    int64_t num_rows, int64_t num_nonzeros,
    const struct mtxfile_vector_coordinate_integer_single * data);
```

4.4.3 Setting matrix and vector values

For convenience, the functions `mtxfile_set_constant_`*`field_precision`* are provided to initialise the values of a matrix or vector to a constant, where *`field`* and *`precision`* match the field and precision of the specified `mtxfile` struct.

```
int mtxfile_set_constant_real_single(struct mtxfile *, float a);
int mtxfile_set_constant_real_double(struct mtxfile *, double a);
int mtxfile_set_constant_complex_single(struct mtxfile *, float a[2]);
int mtxfile_set_constant_complex_double(struct mtxfile *, double a[2]);
int mtxfile_set_constant_integer_single(struct mtxfile *, int32_t a);
int mtxfile_set_constant_integer_double(struct mtxfile *, int64_t a);
```

4.5 Other operations on Matrix Market files

This section describes various operations that may be performed on Matrix Market files, including transposing, sorting, permuting and partitioning matrices and vectors, as well as reordering the rows and columns of sparse matrices.

4.5.1 Transpose

The function `mtxfile_transpose` can be used to transpose a matrix.

```
int mtxfile_transpose(struct mtxfile * mtxfile);
```

If `mtxfile` is a vector, nothing is done.

4.5.2 Sort, compact and assemble

Sometimes, it is useful to sort the values of a matrix or vector in some particular order. For this purpose, the enum type `mtxfilesorting` is used to enumerate different ways of sorting Matrix Market files.

```
enum mtxfilesorting {
    mtxfile_unsorted,      /* unsorted (default ordering) */
    mtxfile_row_major,     /* row major ordering */
    mtxfile_column_major,  /* column major ordering */
    mtxfile_morton,        /* Morton (Z-order curve) ordering */
    mtxfile_permutation,   /* user-defined sorting permutation */
```

```
};
```

Matrices and vectors in array format are assumed to be sorted in row major order (`mtxfile_row_major`). Matrices and vectors in coordinate format, on the other hand, are generally unsorted (`mtxfile_unsorted`).

To sort a matrix or vector, use the function `mtxfile_sort`:

```
int mtxfile_sort(
    struct mtxfile * mtxfile,
    enum mtxfilesorting sorting,
    int64_t size,
    int64_t * perm);
```

If successful, `mtxfile_sort` returns '`MTX_SUCCESS`', and the values of `mtxfile` will be sorted in the order specified by `sorting`. The underlying sorting algorithm is a radix sort.

If `perm` is '`NULL`', then it is ignored. Otherwise, it must point to an array of length `size`, which is used to store the permutation of the Matrix Market entries. `size` must therefore be at least equal to the number of data lines in the Matrix Market file `mtxfile`.

When dealing with matrices in coordinate format, there may in general be multiple entries in a Matrix Market file with the same row and column index. Similarly, for vectors in coordinate format, there may be multiple entries with the same row offset. In some cases, these duplicate entries need to be merged into a single, unique entry, because certain computations involving sparse matrices and vectors assume that there are no such duplicates.

If a Matrix Market file is already sorted, then `mtxfile_compact` can be used to perform a *compaction*. This will merge duplicate matrix or vector entries, if they are adjacent to each other in the Matrix Market file.

```
int mtxfile_compact(
    struct mtxfile * mtxfile,
    int64_t size,
    int64_t * perm);
```

For a matrix or vector in array format, this does nothing.

The number of nonzero matrix or vector entries, ('`mtxfile->size.num_nonzeros`'), is updated to reflect entries that were removed as a result of compacting. However, the underlying storage is not changed or reallocated. This may result in large amounts of unused memory if many entries are removed. In such cases, it may be necessary to allocate new storage, copy the compacted data, and, finally, free the old storage.

If `perm` is not '`NULL`', then it must point to an array of length '`size`'. Each entry in `perm` is used to store the offset to the corresponding entry in the compacted array that the entry was moved to or merged with. Note that the indexing is 1-based.

To give an example, consider the following Matrix Market file:

```
%%MatrixMarket matrix coordinate real general
3 3 8
1 1 2.0
1 2 -2.0
2 1 -2.0
2 2 2.0
```

```
2 2 2.0
2 3 -2.0
3 2 -2.0
3 3 2.0
```

The above Matrix Market file consists of two 2-by-2 *element matrices* from a finite element discretisation of the one-dimensional Poisson equation on the unit interval using two equal-sized first-order elements. Note that the two element matrices overlap, since they both contribute a value to the matrix entry '`(2,2)`', that is, the second column of the second row. After compaction, these two entries are merged into one, and the matrix instead becomes:

```
%%MatrixMarket matrix coordinate real general
3 3 7
1 1 2.0
1 2 -2.0
2 1 -2.0
2 2 4.0
2 3 -2.0
3 2 -2.0
3 3 2.0
```

If a Matrix Market file is not already sorted, then duplicate entries are not necessarily adjacent, and `mtxfile_compact` will not work. In this case, one should instead use `mtxfile_assemble`, which will sort the Matrix Market file before performing the compaction.

```
int mtxfile_assemble(
    struct mtxfile * mtxfile,
    enum mtxfilesorting sorting,
    int64_t size,
    int64_t * perm);
```

4.5.3 Partition

Partitioning matrices and vectors is a prerequisite for most distributed-memory, parallel operations in linear algebra. This is usually done to divide up the work among multiple processes. But Partitioning can also be done for other reasons, such as reordering matrix rows and columns. In the most general case, Libmtx provides functions for partitioning the nonzeros of sparse matrices and vectors in an arbitrary manner. However, there are also functions to cover more common use cases, such as partitioning a matrix by rows or columns.

The function `mtxfile_partition_nonzeros` is used to partition the nonzeros of a Matrix Market file.

```
int mtxfile_partition_nonzeros(
    const struct mtxfile * mtxfile,
    enum mtxpartitioning parttype,
    int num_parts,
    const int64_t * partsizes,
    int64_t blksize,
    const int * parts,
```

```
        int * dstpart,
        int64_t * dstpartsizes);
```

The array pointed to by `dstpart` is used to store the part number assigned to each matrix or vector nonzero, and its length must therefore be at least equal to the number of nonzeros. If `dstpartsizes` is not '`NULL`', then it must be an array of length '`num_parts`', and it is used to store the number of nonzeros assigned to each part.

The type of partitioning performed is determined by the arguments `parttype`, `num_parts`, `partsizes`, `blksize` and `parts`. More specifically, if the set to be partitioned consists of '`N`' nonzeros, then

- if `type` is '`mtx_cyclic`', nonzeros are partitioned in a cyclic fashion into `num_parts` parts; or,
- if `type` is '`mtx_block`', the array `partsizes` contains `num_parts` integers, specifying the size of each block of the partitioned set; or,
- if `type` is '`mtx_block_cyclic`', nonzeros are arranged in contiguous blocks of size '`blksize`', which are then partitioned in a cyclic fashion into `num_parts` parts; or,
- if `type` is '`mtx_custom_partition`', the array `parts` must be of length '`size`' and should contain the part number (i.e., an integer in the range '`[0,num_parts)`') for each nonzero.

To partition the entries of a Matrix Market file by rows or columns, the functions `mtxfile_partition_rowwise` or `mtxfile_partition_columnwise` can be used.

```
    int mtxfile_partition_rowwise(
        const struct mtxfile * mtxfile,
        enum mtxpartitioning parttype,
        int num_parts,
        const int64_t * partsizes,
        int64_t blksize,
        const int * parts,
        int * dstpart,
        int64_t * dstpartsizes);

    int mtxfile_partition_columnwise(
        const struct mtxfile * mtxfile,
        enum mtxpartitioning parttype,
        int num_parts,
        const int64_t * partsizes,
        int64_t blksize,
        const int * parts,
        int * dstpart,
        int64_t * dstpartsizes);
```

In the same way as for `mtxfile_partition_nonzeros`, the output is a partitioning of the nonzeros, which is written to the array pointed to by `dstpart` (whose length must therefore be at least equal to the number of nonzeros). Similarly, if `dstpartsizes` is not '`NULL`', then it must be an array of length '`num_parts`', and it is used to store the number of nonzeros assigned to each part.

The type of partitioning performed is determined by the arguments `parttype`, `num_parts`, `partsizes`, `blksize` and `parts`, as described above. However, the chosen method is applied to partition the matrix rows (or columns) instead of partitioning the nonzeros directly. Afterwards, the partitioning of the nonzeros is determined based on the row (or column) they belong to.

In cases where a 2D partitioning is needed, the function `mtxfile_partition_2d` can be used.

```
int mtxfile_partition_2d(
    const struct mtxfile * mtxfile,
    enum mtxpartitioning rowparttype,
    int num_row_parts,
    const int64_t * rowpartsizes,
    int64_t rowblksize,
    const int * rowparts,
    enum mtxpartitioning colparttype,
    int num_column_parts,
    const int64_t * colpartsizes,
    int64_t colblksize,
    const int * colparts,
    int * dstpart,
    int64_t * dstpartsizes);
```

The rows and columns of the matrix are then partitioned independently. The total number of parts is equal to the product of '`num_row_parts`' and '`num_column_parts`'.

The meaning of the arguments `rowparttype`, `num_row_parts`, `rowpartsizes` and `rowblksize`, and `rowparts`, are the same as described above for `mtxfile_partition_rowwise`.

Finally, there is a function `mtxfile_partition`, which can be used to perform any of the partitionings described above, as well as partitioning sparse matrices by using an external graph partitioning library, METIS.

```
int mtxfile_partition(
    struct mtxfile * mtxfile,
    enum mtxmatrixparttype matrixparttype,
    enum mtxpartitioning nzparttype,
    int num_nz_parts,
    const int64_t * nzpartsizes,
    int64_t nzblksize,
    const int * nzparts,
    enum mtxpartitioning rowparttype,
    int num_row_parts,
    const int64_t * rowpartsizes,
    int64_t rowblksize,
    const int * rowparts,
    enum mtxpartitioning colparttype,
    int num_column_parts,
    const int64_t * colpartsizes,
```

```
        int64_t colblksize,
        const int * colparts,
        int * dstnzpart,
        int64_t * dstnzpartsizes,
        bool * rowpart,
        int * dstrowpart,
        int64_t * dstrowpartsizes,
        bool * colpart,
        int * dstcolpart,
        int64_t * dstcolpartsizes,
        int verbose);
```

The type of partitioning to perform is determined by `matrixparttype`:

- If `matrixparttype` is '`mtx_matrixparttype_nonzeros`', the nonzeros of the underlying matrix or vector are partitioned as a one-dimensional array. The nonzeros are partitioned into `num_nz_parts` parts according to the partitioning `nzparttype`. If `nzparttype` is '`mtx_block`', then `nzpartsizes` may be used to specify the size of each part. If `nzparttype` is '`mtx_block_cyclic`', then `nzblksize` is used to specify the block size.
- If `matrixparttype` is '`mtx_matrixparttype_rows`', the nonzeros of the underlying matrix or vector are partitioned rowwise.
- If `matrixparttype` is '`mtx_matrixparttype_columns`', the nonzeros of the underlying matrix are partitioned columnwise.
- If `matrixparttype` is '`mtx_matrixparttype_2d`', the nonzeros of the underlying matrix are partitioned in rectangular blocks according to the partitioning of the rows and columns.
- If `matrixparttype` is '`mtx_matrixparttype_metis`', then the rows and columns of the underlying matrix are partitioned by the METIS graph partitioner, and the matrix nonzeros are partitioned accordingly.

In any case, the array `dstnzpart` is used to store the part numbers assigned to the matrix nonzeros, and must therefore be of length '`mtxfile->datasize`'.

If the rows are partitioned, then the array `dstrowpart` must be of length '`mtxfile->size.num_rows`', and it is used to store the part numbers assigned to the matrix rows. Furthermore, the value pointed to by rowpart is also set to '`true`', whereas it is '`false`' otherwise.

Similarly, if the columns are partitioned (e.g., when partitioning columnwise, 2d or a graph-based partitioning of a non-square matrix), then `dstcolpart` is used to store the part numbers assigned to the matrix columns, and it must therefore be an array of length '`mtxfile->size.num_columns`'. Moreover, the value pointed to by colpart is set to '`true`', whereas it is '`false`' otherwise.

Unless they are set to '`NULL`', then `dstnzpartsizes`, `dstrowpartsizes` and `dstcolpartsizes` must be arrays of length '`num_parts`', which are then used to store the number of nonzeros, rows and columns assigned to each part, respectively.

Given a partitioning of the (nonzero) entries of a Matrix Market file, the function `mtxfile_split` can be used to split the file into several files, one for each part.

```
    int mtxfile_split(
```

```
    int num_parts,
    struct mtxfile ** dsts,
    const struct mtxfile * src,
    int64_t size,
    int * parts,
    const int64_t * num_rows_per_part,
    const int64_t * num_columns_per_part);
```

The partitioning of the matrix or vector entries is given by the array `parts`. The length of the `parts` array is given by `size`, and it must match the number of (nonzero) matrix or vector entries in `src`. Each entry in the array is an integer in the range '`[0, num_parts)`' designating the part to which the corresponding nonzero element belongs.

The argument `dsts` is an array of `num_parts` pointers to objects of type `struct mtxfile`. If successful, then '`dsts[p]`' points to a Matrix Market file consisting of (nonzero) entries from src that belong to the '`p`'th part, according to the `parts` array.

If `src` is a matrix (or vector) in coordinate format, then each of the matrices or vectors in `dsts` is also a matrix (or vector) in coordinate format with the same number of rows and columns as `src`. In this case, the arguments `num_rows_per_part` and `num_columns_per_part` are not used and may be set to '`NULL`'.

Otherwise, if `src` is a matrix (or vector) in array format, then the arrays `num_rows_per_part` and `num_columns_per_part` (both of length '`num_parts`') are used to specify the dimensions of each matrix (or vector) in `dsts`. For a given part '`p`', the number of matrix (or vector) elements assigned to that part must be equal to the product of '`num_rows_per_part[p]`' and '`num_columns_per_part[p]`'.

The user is responsible for freeing storage allocated for each Matrix Market file in the `dsts` array.

4.5.4 Reorder

There are a number of commonly used schemes for reordering the rows and columns of sparse matrices. The goal can be to reduce fill-in that occurs during the factorisation stage of sparse direct solvers or to improve the performance of operations such as sparse matrix-vector multiplication.

The function `mtxfile_permute` can be used to permute the rows of a vector or the rows and columns of a matrix based on given row and column permutations.

```
int mtxfile_permute(
    struct mtxfile * mtxfile,
    const int * rowperm,
    const int * colperm);
```

The array `rowperm` is used to reorder the rows of a matrix or vector, and the array `colperm` is used to reorder the columns of a matrix. Therefore, `rowperm` must be a permutation of the integers `1,2,...,`*M*, where *M* is the number of rows in the matrix or vector. If `mtxfile` is a matrix, then the array `colperm` must be a permutation of the integers `1,2,...,`*N*, where *N* is the number of columns in the matrix. If `mtxfile` is a vector, then `colperm` is ignored. If successful, the element belonging to row `i` and column `j` in the permuted matrix will be equal to the element in row `rowperm[i-1]` and column `colperm[j-1]` of the original matrix, for `i=1,2,...,`*M* and `j=1,2,...,`*N*.

In addition to permuting a matrix or vector, Libmtx also provides functions to obtain row and column permutations for certain orderings. The enum type `mtxfileordering` is used to enumerate different orderings for the rows and columns of a matrix.

```
enum mtxfileordering {
    mtxfile_custom_order,  /* general, user-defined ordering */
    mtxfile_rcm,           /* Reverse Cuthill-McKee ordering */
    mtxfile_nd,            /* nested dissection ordering */
};
```

The function `mtxfile_reorder` reorders the rows and columns of a matrix according to the specified ordering method.

```
int mtxfile_reorder(
    struct mtxfile * mtxfile,
    enum mtxfileordering ordering,
    int * rowperm,
    int * rowperminv,
    int * colperm,
    int * colperminv,
    bool permute,
    bool * symmetric,
    int * rcm_starting_vertex);
```

If successful, `mtxfile_reorder` returns '`MTX_SUCCESS`', and the rows and columns of `mtxfile` have been reordered according to the specified method. If `rowperm` is not '`NULL`', then it must point to an array whose length is at least equal to the number of rows in the matrix. In this case, the array is used to store the permutation for reordering the matrix rows. Similarly, `colperm` may be used to store the permutation for reordering the matrix columns.

In some cases, only the row and column permutations are needed, and the permutations should not be applied to `mtxfile`. Therefore, the computed permutations are only applied if `permute` is '`true`'.

Finally, if `ordering` is '`mtxfile_rcm`', then `rcm_starting_vertex` can be used to specify a starting vertex for the Reverse Cuthill-McKee algorithm. Moreover, if the starting vertex is set to '`0`', then a starting vertex is chosen automatically, and `rcm_starting_vertex` will be used to return the chosen starting vertex.

4.5.4.1 Reverse Cuthill-McKee (RCM)

If `mtxfile_reorder` is called with `ordering` set to '`mtxfile_rcm`', then the rows and columns of a matrix are reordered according to the Reverse Cuthill-McKee algorithm (see [E. Cuthill and J. McKee (1969)], page 65). See Figure 4.1 for an example of the RCM reordering applied to the matrix "webbase-1M" from the "Williams" group in the SuiteSparse Matrix Collection ([T. Davis, Y. Hu and S. Kolodziej (2021)], page 65).

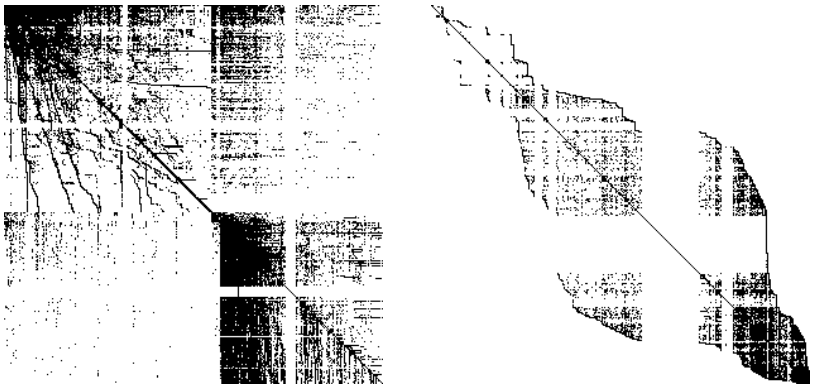

Figure 4.1: Sparsity pattern of the matrix "webbase-1M" with original ordering (left) and RCM ordering (right).

The RCM algorithm considers the matrix as the adjacency matrix of an undirected graph. The vertices of the graph, which correspond to rows and column of the matrix, are ordered by choosing a starting vertex and then traversing the graph in a breadth-first search, where the vertices at each level are ordered ascendingly by degree. In the end, after traversing the entire graph, the obtained ordering is reversed.

For a square matrix, the Cuthill-McKee algorithm is carried out on the adjacency matrix of the symmetrisation `A+A'`, where `A'` denotes the transpose of `A`. For a rectangular matrix, the Cuthill-McKee algorithm is carried out on a bipartite graph formed by the matrix rows and columns. The adjacency matrix `B` of the bipartite graph is square and symmetric and takes the form of a 2-by-2 block matrix where `A` is placed in the upper right corner and `A'` is placed in the lower left corner:

```
    | 0   A |
B = |       |.
    | A'  0 |
```

A starting vertex may either be chosen explicitly by the user. Otherwise, the starting vertx is chosen automatically by selecting a pseudo-peripheral vertex. In the case of a square matrix, the starting vertex must be in the range `[1,`*M*`]`, where *M* is the number of rows (and columns) of the matrix. Otherwise, if the matrix is rectangular, a starting vertex in the range `[1,`*M*`]` selects a vertex corresponding to a row of the matrix, whereas a starting vertex in the range `[`*M*`+1,`*M*`+`*N*`]`, where *N* is the number of matrix columns, selects a vertex corresponding to a column of the matrix.

4.5.4.2 Nested Dissection

If `mtxfile_reorder` is called with `ordering` set to '`mtxfile_nd`', then the nested dissection algorithm is used to reorder the rows and columns of the given sparse matrix. The METIS graph partitioning library is used to perform the ordering, which is based on a multilevel recursive bisection algorithm,

4.6 Communicating Matrix Market files

If Libmtx is built with MPI support, then some additional functionality becomes available to allow sending and receiving Matrix Market files between MPI processes. Note that these functions rely on the error handling functionality described in Section 5.1 [Error handling for distributed Matrix Market files], page 31.

The most basic functions for communicating `mtx` files are `mtxfile_send` and `mtxfile_recv`. The former sends a Matrix Market file to another MPI process, whereas the latter receives a Matrix Market file from another MPI process.

```
int mtxfile_send(
    const struct mtxfile * mtxfile,
    int dest, int tag, MPI_Comm comm,
    struct mtxdisterror * disterr);

int mtxfile_recv(
    struct mtxfile * mtxfile,
    int source, int tag, MPI_Comm comm,
    struct mtxdisterror * disterr);
```

These functions are analogous to `MPI_Send` and `MPI_Recv`. Thus, a call to `mtxfile_send` requires the receiving process (`dest`) to perform a matching call to `mtxfile_recv`. Similarly, `mtxfile_recv` requires the sending process (`source`) to perform a matching call to `mtxfile_send`.

Note that the MPI communicator `comm` must be the same MPI communicator that was passed to `mtxdisterror_alloc` to create `disterr`. This applies to all of the functions in this section.

The function `mtxfile_bcast` broadcasts a Matrix Market file from an MPI root process to other processes in a communicator.

```
int mtxfile_bcast(
    struct mtxfile * mtxfile,
    int root, MPI_Comm comm,
    struct mtxdisterror * disterr);
```

This function is analogous to `MPI_Bcast` and therefore requires every process in a communicator to perform matching calls to `mtxfile_bcast`.

There are also a number of other functions that mirror the functionality of `MPI_Gather`, `MPI_Allgather`, `MPI_Scatter` and `MPI_Alltoall`. These functions are all collective and therefore require every process in a communicator to perform matching calls to the relevant function.

The following is a brief description of each function:

- `mtxfile_gather` gathers Matrix Market files onto an MPI root process from other processes in a communicator.
- `mtxfile_allgather` gathers Matrix Market files onto every MPI process from other processes in a communicator.
- `mtxfile_scatter` scatters Matrix Market files from an MPI root process to other processes in a communicator.

- `mtxfile_alltoall` performs an all-to-all exchange of Matrix Market files between MPI process in a communicator.

```
int mtxfile_gather(
    const struct mtxfile * sendmtxfile,
    struct mtxfile * recvmtxfiles,
    int root, MPI_Comm comm,
    struct mtxdisterror * disterr);

int mtxfile_allgather(
    const struct mtxfile * sendmtxfile,
    struct mtxfile * recvmtxfiles,
    MPI_Comm comm,
    struct mtxdisterror * disterr);

int mtxfile_scatter(
    const struct mtxfile * sendmtxfiles,
    struct mtxfile * recvmtxfile,
    int root, MPI_Comm comm,
    struct mtxdisterror * disterr);

int mtxfile_alltoall(
    const struct mtxfile * sendmtxfiles,
    struct mtxfile * recvmtxfiles,
    MPI_Comm comm,
    struct mtxdisterror * disterr);
```

`mtxfile_scatterv` scatters a Matrix Market file from an MPI root process to other processes in a communicator, while allowing different amounts of data lines or values to be sent to each process.

```
int mtxfile_scatterv(
    const struct mtxfile * sendmtxfile,
    const int * sendcounts,
    const int * displs,
    struct mtxfile * recvmtxfile,
    int recvcount,
    int root, MPI_Comm comm,
    struct mtxdisterror * disterr);
```

Note that for a matrix in array format, entire rows are scattered, which means that the send and receive counts must be multiples of the number of matrix columns.

5 Distributed Matrix Market files

This chapter describes how to distribute Matrix Market files among multiple processes using MPI, and how to perform various operations on those files in a distributed manner. To make use of these features, you will need to build Libmtx with MPI support.

5.1 Error handling

In addition to the error handling routines described in Section 4.1 [Error handling], page 9, Libmtx provides some additional error handling functionality when working with MPI and distributed data. First, some MPI functions may return an error code on failure, which should be handled correctly. Second, whenever multiple processes are involved, there are cases where only one or a few of those processes may encounter errors. These errors must be handled appropriately to ensure accurate reporting and that the program exits in a graceful manner instead of hanging indefinitely.

5.1.1 MPI errors

Some functions in Libmtx may fail due to MPI errors. In these cases, some additional information is needed to provide helpful error descriptions, and the function `mtxdiststrerror` should be used (instead of `mtxstrerror`).

```
const char * mtxdiststrerror(
    int err, int mpierrcode, char * mpierrstr);
```

The error code `err` is an integer corresponding to one of the error codes from the `mtxerror` enum type. The arguments `mpierrcode` and `mpierrstr` are only used if `err` is '`MTX_ERR_MPI`'.

If `err` is '`MTX_ERR_MPI`', then the argument `mpierrcode` should be set to the error code that was returned from the MPI function call that failed. In addition, the argument `mpierrstr` must be a char array whose length is at least equal to `MPI_MAX_ERROR_STRING`. Internally, `mtxdiststrerror` uses `MPI_Error_string` to obtain a description of the error.

The example below shows how `mtxdiststrerror` is typically used.

```
int err, mpierr;
char mpierrstr[MPI_MAX_ERROR_STRING];
struct mtxdisterror disterr;
err = mtxdisterror_alloc(&disterr, MPI_COMM_WORLD, &mpierr);
if (err) {
    fprintf(stderr, "error: %s\n",
            mtxdiststrerror(err, mpierr, mpierrstr));
    MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
}
```

If `mtxdisterror_alloc` returns '`MTX_ERR_MPI`' and `mpierr` is set to '`MPI_ERR_COMM`', then the following message will be printed:

```
error: MPI_ERR_COMM: invalid communicator
```

5.1.2 Distributed error handling

To more easily handle errors in cases where one or more processes may fail, Libmtx uses the data type `struct mtxdisterror`. Most of the functions in Libmtx that involve distributed

computing take an additional argument of type `struct mtxdisterror` to provide robust error handling in these cases.

To use `struct mtxdisterror`, one must first allocate storage using `mtxdisterror_alloc`.

```
int mtxdisterror_alloc(
    struct mtxdisterror * disterr,
    MPI_Comm comm,
    int * mpierrcode);
```

An example of this was already shown in the previous section.

Note that the storage allocated for `mtxdisterror` should be freed by calling `mtxdisterror_free`.

```
void mtxdisterror_free(struct mtxdisterror * disterr);
```

If an error occurs, then a description of the error can be obtained by calling `mtxdisterror_description`.

```
char * mtxdisterror_description(struct mtxdisterror * disterr);
```

Note that if `mtxdisterror_description` is called more than once, the pointer that was returned from the previous call will no longer be valid and using it will result in a use-after-free error.

Finally, the function `mtxdisterror_allreduce` can be used to communicate error status among multiple processes.

```
int mtxdisterror_allreduce(struct mtxdisterror * disterr, int err);
```

More specifically, `mtxdisterror_allreduce` performs a collective reduction on error codes provided by each MPI process in the communicator used by `disterr`. This is the same MPI communicator that was provided as the `comm` argument to `mtxdisterror_alloc`.

Because `mtxdisterror_allreduce` is a collective operation, it must be performed by every process in the communicator of `disterr`. Otherwise, the program may hang indefinitely.

Each process gathers the error code and rank of every other process. If the error code of each and every process is '`MTX_SUCCESS`', then `mtxdisterror_allreduce` returns '`MTX_SUCCESS`'. Otherwise, '`MTX_ERR_MPI_COLLECTIVE`' is returned. Moreover, the rank and error code of each process is stored in `disterr`.

If the error code `err` is '`MTX_ERR_MPI_COLLECTIVE`', then it is assumed that a reduction has already been performed, and `mtxdisterror_allreduce` returns immediately with '`MTX_ERR_MPI_COLLETIVE`'. As a result, if any process calls `mtxdisterror_allreduce` with `err` set to '`MTX_ERR_MPI_COLLETIVE`', then every other process in the communicator must also set `err` to '`MTX_ERR_MPI_COLLECTIVE`', or else the program may hang indefinitely.

The following example shows how `mtxdisterror_allreduce` is used.

```
int err;
struct mtxdisterror disterr;
err = mtxdisterror_alloc(&disterr, MPI_COMM_WORLD);
if (err)
    MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
```

```
// Get the MPI rank of the current process.
// Perform an all-reduction on the error code from
// MPI_Comm_rank, so that if any process fails,
// then we can exit gracefully.
int comm_err, rank;
err = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
comm_err = mtxdisterror_allreduce(&disterr, err);
if (comm_err)
    return comm_err;

...
```

5.2 Data structures

The file `libmtx/mtxfile/mtxdistfile.h` defines the type `struct mtxdistfile`, which is used to represent a Matrix Market file distributed among one or more MPI processes. Conceptually, processes are arranged as a one-dimensional linear array. Furthermore, the data is also arranged as a one-dimensional linear array, which is then distributed among the processes of the communicator `comm`. The definition of the `mtxdistfile` struct is shown below.

```
struct mtxdistfile {
    MPI_Comm comm;
    int comm_size;
    int rank;
    struct mtxfileheader header;
    struct mtxfilecomments comments;
    struct mtxfilesize size;
    enum mtxprecision precision;
    int64_t datasize;
    int64_t localdatasize;
    int64_t * idx;
    union mtxfiledata data;
};
```

The first three struct members contain some information about the group of processes sharing the distributed Matrix Market file, including their MPI communicator (`comm`), the number of processes (`comm_size`) and the rank of the current process (`rank`).

Thereafter, follows the header line, comments, size line and the chosen precision, all of which must be identical on every process in the communicator. The final struct member, `data`, is used to store those data lines of the Matrix Market file that reside on the current process.

The nonzero entries of the underlying Matrix Market file are distributed among the processes such that `datasize` is the total number of entries in the entire Matrix Market file, while `localdatasize` is the number of entries stored on the current process. In addition, the array `idx`, whose length is equal to '`localdatasize`', contains the global offset for each entry stored on the current process. (Note that these offsets are 0-based.)

5.3 Creating distributed Matrix Market files

Constructing distributed Matrix Market files works in much the same way as the non-distributed case, which was described in Section 4.4 [Creating Matrix Market files], page 17. First of all, `mtxdistfile_free` is used to free storage that is allocated when creating a distributed Matrix Market file.

```
void mtxdistfile_free(struct mtxdistfile * mtxdistfile);
```

To allocate storage for a distributed Matrix Market file with the given header line, comment lines, size line and precision, use `mtxdistfile_alloc`.

```
int mtxdistfile_alloc(
    struct mtxdistfile * mtxdistfile,
    const struct mtxfileheader * header,
    const struct mtxfilecomments * comments,
    const struct mtxfilesize * size,
    enum mtxprecision precision,
    int64_t localdatasize,
    const int64_t * idx,
    MPI_Comm comm,
    struct mtxdisterror * disterr);
```

`comments` may be '`NULL`', in which case it is ignored. `localdatasize` is the number of entries in the underlying Matrix Market file that are stored on the current process. It is also the length of the `idx` array, which is used to specify the global offsets of the entries stored on the current process. Finally, `comm` must be the same MPI communicator that was used to create `disterr`.

To allocate storage for a copy of an existing `mtxdistfile`, the function `mtxdistfile_alloc_copy` is used. This function does not initialise the underlying matrix or vector values. If the matrix or vector values should also be copied, then `mtxdistfile_init_copy` is used.

```
int mtxdistfile_alloc_copy(
    struct mtxdistfile * dst,
    const struct mtxdistfile * src,
    struct mtxdisterror * disterr);

int mtxdistfile_init_copy(
    struct mtxdistfile * dst,
    const struct mtxdistfile * src,
    struct mtxdisterror * disterr);
```

5.3.1 Creating distributed `mtx` files in array format

The functions `mtxdistfile_alloc_matrix_array` and `mtxdistfile_alloc_vector_array` are used to allocate distributed matrices and vectors in array format.

```
int mtxdistfile_alloc_matrix_array(
    struct mtxdistfile * mtxdistfile,
    enum mtxfilefield field,
    enum mtxfilesymmetry symmetry,
    enum mtxprecision precision,
    int64_t num_rows,
```

```
        int64_t num_columns,
        int64_t localdatasize,
        const int64_t * idx,
        MPI_Comm comm,
        struct mtxdisterror * disterr);

    int mtxdistfile_alloc_vector_array(
        struct mtxdistfile * mtxdistfile,
        enum mtxfilefield field,
        enum mtxprecision precision,
        int64_t num_rows,
        int64_t localdatasize,
        const int64_t * idx,
        MPI_Comm comm,
        struct mtxdisterror * disterr);
```

`field` must be '`mtxfile_real`', '`mtxfile_complex`' or '`mtxfile_integer`'. Moreover, `field` and `precision` must be the same on every process in the MPI communicator. Likewise, `num_rows` and `num_columns` specify the total number of rows and columns in the distributed matrix or vector, and must therefore be the same on every process.

The above functions allocate storage, but they do not initialise the underlying matrix or vector values. It is therefore up to the user to initialise these values.

If the matrix or vector values are already known, then the functions `mtxdistfile_init_`*`object`*`_array_`*`field`*`_`*`precision`* can be used to allocate storage and initialise values. Here *`object`*, *`field`* and *`precision`* correspond to the desired object ('`matrix`' or '`vector`'), field ('`real`', '`complex`' or '`integer`'), and precision ('`single`' or '`double`'). For example, for a distributed matrix in array format with real, single precision coefficients, the function `mtxdistfile_init_matrix_array_real_single` is used, as shown below.

```
    int mtxdistfile_init_matrix_array_real_single(
        struct mtxdistfile * mtxdistfile,
        enum mtxfilesymmetry symmetry,
        int64_t num_rows,
        int64_t num_columns,
        int64_t localdatasize,
        const int64_t * idx,
        const float * data,
        MPI_Comm comm,
        struct mtxdisterror * disterr);
```

The corresponding function for a vector is `mtxdistfile_init_vector_array_real_single`.

```
    int mtxdistfile_init_vector_array_real_single(
        struct mtxdistfile * mtxdistfile,
        int64_t num_rows,
        int64_t localdatasize,
        const int64_t * idx,
        const float * data,
```

```
    MPI_Comm comm,
    struct mtxdisterror * disterr);
```

5.3.2 Creating distributed `mtx` files in coordinate format

Matrices and vectors in coordinate format are created in a similar way to what was shown in the previous section. The functions `mtxdistfile_alloc_matrix_coordinate` and `mtxdistfile_alloc_vector_coordinate` can be used to allocate distributed matrices and vectors in coordinate format.

```
int mtxdistfile_alloc_matrix_coordinate(
    struct mtxdistfile * mtxdistfile,
    enum mtxfilefield field,
    enum mtxfilesymmetry symmetry,
    enum mtxprecision precision,
    int64_t num_rows,
    int64_t num_columns,
    int64_t num_nonzeros,
    int64_t localdatasize,
    const int64_t * idx,
    MPI_Comm comm,
    struct mtxdisterror * disterr);

int mtxdistfile_alloc_vector_coordinate(
    struct mtxdistfile * mtxdistfile,
    enum mtxfilefield field,
    enum mtxprecision precision,
    int64_t num_rows,
    int64_t num_nonzeros,
    int64_t localdatasize,
    const int64_t * idx,
    MPI_Comm comm,
    struct mtxdisterror * disterr);
```

The main differences compared to array formats are: `field` is allowed to be '`mtxfile_pattern`', and an additional argument (`num_nonzeros`) is needed to specify the number of (nonzero) matrix or vector entries. Note that `num_nonzeros` is the total number of nonzeros in the distributed Matrix Market file (every process must specify the same value for this argument). The number of nonzeros that will reside on the current process is specified by `localdatasize`.

The above functions allocate storage, but they do not initialise the underlying matrix or vector values. It is therefore up to the user to initialise these values. Alternatively, if the matrix or vector values are readily available, then the functions `mtxdistfile_init_`*`object`*`_coordinate_`*`field`*`_`*`precision`* can be used to allocate storage and initialise the matrix or vector values at the same time. As before, *`object`*, *`field`* and *`precision`* correspond to the desired object ('`matrix`' or '`vector`'), field ('`real`', '`complex`', '`integer`' or '`pattern`'), and precision ('`single`' or '`double`'). For example, for a distributed matrix in coordinate format with real, single precision coefficients, the function `mtxdistfile_init_matrix_coordinate_real_single` is used, as shown below.

```
int mtxdistfile_init_matrix_coordinate_real_single(
    struct mtxdistfile * mtxdistfile,
    enum mtxfilesymmetry symmetry,
    int64_t num_rows,
    int64_t num_columns,
    int64_t num_nonzeros,
    int64_t localdatasize,
    const int64_t * idx,
    const struct mtxfile_matrix_coordinate_real_single * data,
    MPI_Comm comm,
    struct mtxdisterror * disterr);
```

In the case of a vector, the corresponding function is `mtxdistfile_init_vector_coordinate_real_single`.

```
int mtxdistfile_init_vector_coordinate_real_single(
    struct mtxdistfile * mtxdistfile,
    int64_t num_rows,
    int64_t num_nonzeros,
    int64_t localdatasize,
    const int64_t * idx,
    const struct mtxfile_vector_coordinate_real_single * data,
    MPI_Comm comm,
    struct mtxdisterror * disterr);
```

5.3.3 Setting matrix or vector values

For convenience, the functions `mtxdistfile_set_constant_`*`field_precision`* are provided to initialise every value of a distributed matrix or vector to the same constant. Here *`field`* and *`precision`* should match the field (‘`real`’, ‘`complex`’, ‘`integer`’ or ‘`patter`’) and precision (‘`single`’ or ‘`double`’) of `mtxdistfile`.

```
int mtxdistfile_set_constant_real_single(
    struct mtxdistfile * mtxdistfile, float a,
    struct mtxdisterror * disterr);

int mtxdistfile_set_constant_real_double(
    struct mtxdistfile * mtxdistfile, double a,
    struct mtxdisterror * disterr);

int mtxdistfile_set_constant_complex_single(
    struct mtxdistfile * mtxdistfile, float a[2],
    struct mtxdisterror * disterr);

int mtxdistfile_set_constant_complex_double(
    struct mtxdistfile * mtxdistfile, double a[2],
    struct mtxdisterror * disterr);

int mtxdistfile_set_constant_integer_single(
    struct mtxdistfile * mtxdistfile, int32_t a,
```

```
    struct mtxdisterror * disterr);

int mtxdistfile_set_constant_integer_double(
    struct mtxdistfile * mtxdistfile, int64_t a,
    struct mtxdisterror * disterr);
```

5.4 Converting to and from Matrix Market files

This section describes how to convert a Matrix Market file that resides on a single process to a Matrix Market file that is distributed among multiple processes.

The function `mtxdistfile_from_mtxfile_rowwise` takes a Matrix Market file stored on a single root process, partitions the underlying matrix or vector rowwise and distributes it among processes in a communicator.

```
int mtxdistfile_from_mtxfile_rowwise(
    struct mtxdistfile * dst,
    struct mtxfile * src,
    enum mtxpartitioning parttype,
    int64_t partsize, int64_t blksize, const int * parts,
    MPI_Comm comm, int root,
    struct mtxdisterror * disterr);
```

The Matrix Market file `src` is distributed by first broadcasting the header line and precision from the root process to the other processes. Next, the number of matrix or vector elements to send to each process is determined and data is distributed accordingly.

The arguments `parttype`, `partsize`, `blksize` and `parts` may be used to specify the manner in which the rows should be partitioned. For an explanation of these arguments, refer to Section 4.5.3 [Partition], page 22.

This function performs collective communication and therefore requires every process in the communicator to perform matching calls to the function.

The function `mtxdistfile_to_mtxfile` gathers a distributed Matrix Market file onto a single, root process, creating a single Matrix Market file on that process.

```
int mtxdistfile_to_mtxfile(
    struct mtxfile * dst,
    const struct mtxdistfile * src,
    int root,
    struct mtxdisterror * disterr);
```

The resulting Matrix Market file `dst` is only allocated on the process `root`, and so only this process should call `mtxfile_free` to free the allocated storage.

5.5 Reading and writing distributed Matrix Market files

This section explains how to read from and write to files in the Matrix Market format whenever data is distributed among multiple MPI processes.

In the case of reading or writing a distributed matrix or vector in Matrix Market format, there are essentially two options. The first option is the *file-per-process* model, where each process uses its own file to read or write its part of the matrix or vector. The second option

is the *shared file* model, where processes send or receive their data to or from a single root process, and the root process uses a single, shared file to read or write data.

Each of the I/O models mentioned above have advantages and disadvantages. The file-per-process model allows processes to read or write their data in parallel, and may therefore be much faster. However, when a large number of MPI processes are involved, there will also be many files. It is often more difficult for the user to manage multiple files. Furthermore, it also results in significant overhead due to the file system's handling of metadata associated with each file. The shared file model, on the other hand, produces only a single file. This can be much simpler to deal with and there is no overhead associated with metadata beyond that one file. Unfortunately, the I/O performance can be severely limiting due to the fact that only a single process is responsible for reading from or writing to the file.

Note that when using a very large number of MPI processes and very large files, a high-performance I/O library such as MPI-IO ([W. Gropp, E. Lusk and R. Thakur (1999)], page 65) or HDF5 ([HDF5], page 65) may be a better alternative. However, this is not currently supported in Libmtx.

5.5.1 Reading distributed Matrix Market files

To read an `mtx` file from a `FILE` stream, partition the data and distribute it among MPI processes in a communicator based on the shared file model, use `mtxdistfile_fread_rowwise`:

```
int mtxdistfile_fread_rowwise(
    struct mtxdistfile * mtxdistfile,
    enum mtxprecision precision,
    enum mtxpartitioning parttype,
    int64_t partsize, int64_t blksize, const int * parts,
    FILE * f, int64_t * lines_read, int64_t * bytes_read,
    size_t line_max, char * linebuf,
    MPI_Comm comm, int root,
    struct mtxdisterror * disterr);
```

For the most part, `mtxdistfile_fread_rowwise` works just like `mtxfile_fread` (see Section 4.3 [Reading and writing Matrix Market files], page 15). If successful, '`MTX_SUCCESS`' is returned, and `mtxdistfile` will contain the distributed Matrix Market file. The user is responsible for calling `mtxdistfile_free` to free any storage allocated by `mtxdistfile_fread_rowwise`. If `mtxdistfile_fread_rowwise` fails, an error code is returned and `lines_read` and `bytes_read` are used to indicate the line number and byte of the Matrix Market file where an error was encountered. `lines_read` and `bytes_read` are ignored if they are set to '`NULL`'.

Moreover, `precision` is used to choose the precision for storing the values of matrix or vector entries, as described in Section 4.2.4 [Precision], page 12. If `linebuf` is not '`NULL`', then it must point to an array that can hold a null-terminated string whose length (including the terminating null-character) is at most `line_max`. This buffer is used for reading lines from the stream. Otherwise, if `linebuf` is '`NULL`', then a temporary buffer is allocated and used, and the maximum line length is determined by calling `sysconf()` with `_SC_LINE_MAX`.

Only a single root process will read from the specified stream. The data is partitioned rowwise as determined by the arguments `parttype`, `partsize`, `blksize`, and `parts`.

This function performs collective communication and therefore requires every process in the communicator to perform matching calls to the function.

If Libmtx is built with zlib support, then `mtxdistfile_gzread_rowwise` can be used to read gzip-compressed `mtx` files. The data is partitioned and distributed among MPI processes in the same way as with `mtxdistfile_fread_rowwise`.

```
int mtxdistfile_gzread_rowwise(
    struct mtxdistfile * mtxdistfile,
    enum mtxprecision precision,
    enum mtxpartitioning parttype,
    int64_t partsize, int64_t blksize, const int * parts,
    gzFile f, int64_t * lines_read, int64_t * bytes_read,
    size_t line_max, char * linebuf,
    MPI_Comm comm, int root,
    struct mtxdisterror * disterr);
```

For convenience, the function `mtxdistfile_read_rowwise` can be used to read an `mtx` file from a given path.

```
int mtxdistfile_read_rowwise(
    struct mtxdistfile * mtxdistfile,
    enum mtxprecision precision,
    const char * path,
    bool gzip,
    int * lines_read, int64_t * bytes_read,
    MPI_Comm comm,
    struct mtxdisterror * disterr);
```

The file is assumed to be gzip-compressed if `gzip` is '`true`', and uncompressed otherwise. If `path` is '`-`', then the standard input stream is used.

5.5.2 Writing distributed Matrix Market files

To write a distributed `mtx` file to a `FILE` stream using the shared file model, use `mtxdistfile_fwrite`:

```
int mtxdistfile_fwrite(
    const struct mtxdistfile * mtxdistfile,
    FILE * f,
    const char * fmt,
    int64_t * bytes_written,
    int root,
    struct mtxdisterror * disterr);
```

Here, `f` should point to a different stream on every process. The processes involved are those from the MPI communicator `mtxdistfile->comm`. If successful, '`MTX_SUCCESS`' is returned, and each process sent its part of the matrix or vector to the root process, which wrote it to the output stream. Moreover, if `bytes_written` is not '`NULL`', then it is used to return the number of bytes written to the stream.

The `fmt` argument may optionally be used to specify a format string for outputting numerical values, in the same way as with `mtxfile_write` (see Section 4.3.2 [Writing Matrix Market files], page 16).

6 Matrices and vectors

This chapter explains how to convert Matrix Market files to matrices and vectors based on other storage formats, and how to perform various linear algebra operations involving matrices and vectors. For now, we are concerned with matrices and vectors on a single, shared-memory machine or node.

One of the goals of Libmtx is to allow for experimenting with various storage formats for matrices and vectors, as well as different implementations of basic linear algebra operations. Moreover, it should be simple to switch from one storage format or implementation to another. To achieve this, Libmtx defines a single, common data type, `struct mtxvector`, for working with vectors that may have different underlying storage formats or implementations. Similarly, a single data type, `struct mtxmatrix`, is used for matrices that may have different underlying implementations.

This chapter starts by introducing the common matrix and vector data types and their interfaces. Thereafter, detailed descriptions are given for the different matrix and vector implementations.

6.1 Vectors

The file `libmtx/linalg/local/vector.h` defines the type `struct mtxvector`. This is a single, abstract data type used to represent a vector with different options available for the underlying storage and implementation of vector operations.

The enum type `enum mtxvectortype` is used to control the underlying implementation of `struct mtxvector`. The following types of vectors are defined:

- '`mtxbasevector`' provides a basic, serial implementation of most vector operations
- '`mtxblasvector`' provides vector operations that use an external BLAS library, and are therefore usually much faster than '`mtxbasevector`'
- '`mtxompvector`' provides vector operations using OpenMP for shared-memory parallelism

6.1.1 Creating vectors

This section covers functions that are provided to construct vectors.

The function `mtxvector_free` is used to free storage allocated for a vector.

```
void mtxvector_free(struct mtxvector * vector);
```

To create a copy of an existing vector, use the function `mtxvector_init_copy`.

```
int mtxvector_init_copy(
    struct mtxvector * dst,
    const struct mtxvector * src);
```

If storage for a copy of an existing vector is needed, but the vector values should not be copied or initialised, use the function `mtxvector_alloc_copy`.

```
int mtxvector_alloc_copy(
    struct mtxvector * dst,
    const struct mtxvector * src);
```

To allocate a vector in *full storage format*, the function `mtxvector_alloc` is used.

```
int mtxvector_alloc(
    struct mtxvector * x,
    enum mtxvectortype type,
    enum mtxfield field,
    enum mtxprecision precision,
    int64_t size);
```

The desired vector type, field and precision must be specified, as well as the size of the vector. Note that the vector values are not initialised, and so it is up to the user to initialise them.

If the vector values are already known, then there are also functions for allocating a vector and initialising the values directly. This can be done by calling `mtxvector_init_`*`field_precision`*, where *`field`* and *`precision`* denote the field (i.e., '`real`', '`complex`' or '`integer`') and precision (i.e., '`single`' or '`double`').

For example, to create a double precision, complex vector in array format, use `mtxvector_init_array_complex_double`.

```
int mtxvector_init_complex_double(
    struct mtxvector * x,
    enum mtxvectortype type,
    int64_t size,
    const double (* data)[2]);
```

The vector entries are provided by the array `data`, which must contain `size` values.

To create a double precision, complex vector in *packed storage format*, use `mtxvector_init_packed_complex_double`.

```
int mtxvector_init_packed_complex_double(
    struct mtxvector * x,
    enum mtxvectortype type,
    int64_t size,
    int64_t num_nonzeros,
    const int64_t * idx,
    const double (* data)[2]);
```

The arguments `idx` and `data` are arrays of length `num_nonzeros`. Each index '`idx[0]`', '`idx[1]`', ..., '`idx[num_nonzeros-1]`', is an integer in the range `[0,num_rows)`.

Note that duplicate entries are allowed, but this may cause some operations (e.g., `mtxvector_dot`, `mtxvector_nrm2`)) to produce incorrect results.

6.1.2 Modifying values

The functions `mtxvector_set_constant_`*`field_precision`* can be used to set every (non-zero) value of a vector equal to a constant scalar, where *`field`* and *`precision`* should match the field (i.e., '`real`', '`complex`' or '`integer`') and precision (i.e., '`single`' or '`double`') of `mtxvector`.

```
int mtxvector_set_constant_real_single(
    struct mtxvector *, float a);
int mtxvector_set_constant_real_double(
```

```
    struct mtxvector *, double a);
int mtxvector_set_constant_complex_single(
    struct mtxvector *, float a[2]);
int mtxvector_set_constant_complex_double(
    struct mtxvector *, double a[2]);
int mtxvector_set_constant_integer_single(
    struct mtxvector *, int32_t a);
int mtxvector_set_constant_integer_double(
    struct mtxvector *, int64_t a);
```

To access or modify individual vector elements, the underlying vector storage is accessed through the appropriate member of the `storage` union in the `mtxvector` struct.

6.1.3 Converting to and from Matrix Market format

In many cases, a vector may already be available in Matrix Market format. However, for reasons involving both performance and convenience, it is often a good idea to convert the data from Matrix Market format to a more suitable representation before carrying out computations.

To convert a vector in Matrix Market format to `struct mtxvector`, the function `mtxvector_from_mtxfile` can be used.

```
int mtxvector_from_mtxfile(
    struct mtxvector * mtxvector,
    const struct mtxfile * mtxfile,
    enum mtxvectortype type);
```

The user may use the `type` argument to specify a desired storage format or implementation for `mtxvector`. If `mtxfile` is in '`array`' format, the resulting vector will be in *full storage format*. Otherwise, if `mtxfile` is in '`coordinate`' format, the vector will be in *packed storage format*.

Conversely, having performed the necessary computations, it is sometimes useful to convert a vector back to Matrix Market format. For example, to make it easier to output the vector to a file. To convert `struct mtxvector` to a vector in Matrix Market format, the function `mtxvector_to_mtxfile` can be used.

```
int mtxvector_to_mtxfile(
    struct mtxfile * dst,
    const struct mtxvector * src,
    int64_t num_rows,
    const int64_t * idx,
    enum mtxfileformat mtxfmt);
```

The resulting Matrix Market represents a vector in array format if `mtxfmt` is '`mtxfile_array`', or a vector in coordinate format if `mtxfmt` is '`mtxfile_coordinate`'.

6.1.4 Reading and writing Matrix Market files

For convenience, the function `mtxvector_read`, `mtxvector_fread` and `mtxvector_gzread` are provided to more easily read a vector from a file in Matrix Market format and convert it to a desired vector representation. These functions are based on the functions described in Section 4.3 [Reading and writing Matrix Market files], page 15.

```
int mtxvector_read(
    struct mtxvector * vector,
    enum mtxprecision precision,
    enum mtxvectortype type,
    const char * path,
    bool gzip,
    int64_t * lines_read,
    int64_t * bytes_read);

int mtxvector_fread(
    struct mtxvector * vector,
    enum mtxprecision precision,
    enum mtxvectortype type,
    FILE * f,
    int64_t * lines_read,
    int64_t * bytes_read,
    size_t line_max, char * linebuf);

int mtxvector_gzread(
    struct mtxvector * vector,
    enum mtxprecision precision,
    enum mtxvectortype type,
    gzFile f,
    int64_t * lines_read,
    int64_t * bytes_read,
    size_t line_max, char * linebuf);
```

The `type` argument specifies which format to use for representing the vector.

Similarly, the functions `mtxvector_write`, `mtxvector_fwrite` and `mtxvector_gzwrite` are provided to write a vector to a file in Matrix Market format.

```
int mtxvector_write(
    const struct mtxvector * x,
    int64_t num_rows,
    const int64_t * idx,
    enum mtxfileformat mtxfmt,
    const char * path,
    bool gzip,
    const char * fmt,
    int64_t * bytes_written);

int mtxvector_fwrite(
    const struct mtxvector * x,
    int64_t num_rows,
    const int64_t * idx,
    enum mtxfileformat mtxfmt,
    FILE * f,
    const char * fmt,
```

```
    int64_t * bytes_written);

int mtxvector_gzwrite(
    const struct mtxvector * x,
    int64_t num_rows,
    const int64_t * idx,
    enum mtxfileformat mtxfmt,
    gzFile f,
    const char * fmt,
    int64_t * bytes_written);
```

The `mtxfmt` argument may be used to specify whether the vector should be written in array or coordinate format.

6.1.5 Level 1 BLAS

The Libmtx C library implements a subset of the Basic Linear Algebra Subprograms (BLAS) routines. For dense operations, Libmtx can use optimised, third-party BLAS libraries, such as OpenBLAS (`https://www.openblas.net/`). Otherwise, Libmtx uses internal routines for sparse matrix operations.

The following Level 1 BLAS operations are supported:

- `swap` — swap two vectors, `y <- x` and `x <- y`
- `copy` — copy a vector, `y = x`
- `scal` — scale by a constant, `x = a*x`
- `axpy` and `aypx` — add two vectors, `y = a*x + y` or `y = a*y + x`
- `dot` — Euclidean inner product
- `nrm2` — Euclidean norm
- `asum` — sum of absolute values
- `iamax` — find element with largest absolute value

The function `mtxvector_swap` swaps the values of two vectors, whereas `mtxvector_copy` copies the values from one vector to another.

```
int mtxvector_swap(struct mtxvector * x, struct mtxvector * y);
int mtxvector_copy(struct mtxvector * y, const struct mtxvector * x);
```

The functions `mtxvector_sscal` and `mtxvector_dscal` are used to scale a vector `x` by a floating point constant `a` in single or double precision, respectively. That is, `x = a*x`.

```
int mtxvector_sscal(
    float a,
    struct mtxvector * x,
    int64_t * num_flops);

int mtxvector_dscal(
    double a,
    struct mtxvector * x,
    int64_t * num_flops);
```

Note that most of the BLAS functions in Libmtx take an additional argument `num_flops`, which can be used to obtain the number of floating point operations that were carried out. If `num_flops` is '`NULL`', then it is ignored.

The functions `mtxvector_saxpy` and `mtxvector_daxpy` add a vector to another vector multiplied by a single or double precision floating point value, `y = a*x + y`.

```
int mtxvector_saxpy(
    float a,
    const struct mtxvector * x,
    struct mtxvector * y,
    int64_t * num_flops);

int mtxvector_daxpy(
    double a,
    const struct mtxvector * x,
    struct mtxvector * y,
    int64_t * num_flops);
```

Similarly, `mtxvector_saypx` and `mtxvector_daypx` multiply a vector by a single or double precision floating point scalar before adding the result to another vector, `y = a*y + x`.

```
int mtxvector_saypx(
    float a,
    struct mtxvector * y,
    const struct mtxvector * x,
    int64_t * num_flops);

int mtxvector_daypx(
    double a,
    struct mtxvector * y,
    const struct mtxvector * x,
    int64_t * num_flops);
```

The functions `mtxvector_sdot` and `mtxvector_ddot` compute the Euclidean dot product of two real- or integer-valued vectors.

```
int mtxvector_sdot(
    const struct mtxvector * x,
    const struct mtxvector * y,
    float * dot,
    int64_t * num_flops);

int mtxvector_ddot(
    const struct mtxvector * x,
    const struct mtxvector * y,
    double * dot,
    int64_t * num_flops);
```

For complex vectors, the functions `mtxvector_cdotu` and `mtxvector_zdotu` are used to compute the product of the transpose of a complex row vector with another complex

row vector, `x^T*y`, where `x^T` denotes the transpose of `x`. The functions `mtxvector_cdotc` and `mtxvector_zdotc` compute the Euclidean dot product of two complex vectors, `x^H*y`, where `x^H` denotes the conjugate transpose of `x`.

```
int mtxvector_cdotu(
    const struct mtxvector * x,
    const struct mtxvector * y,
    float (* dot)[2],
    int64_t * num_flops);

int mtxvector_zdotu(
    const struct mtxvector * x,
    const struct mtxvector * y,
    double (* dot)[2],
    int64_t * num_flops);

int mtxvector_cdotc(
    const struct mtxvector * x,
    const struct mtxvector * y,
    float (* dot)[2],
    int64_t * num_flops);

int mtxvector_zdotc(
    const struct mtxvector * x,
    const struct mtxvector * y,
    double (* dot)[2],
    int64_t * num_flops);
```

The functions `mtxvector_snrm2` and `mtxvector_dnrm2` compute the Euclidean norm of a vector. in single and double precision floating point, respectively.

```
int mtxvector_snrm2(
    const struct mtxvector * x,
    float * nrm2,
    int64_t * num_flops);

int mtxvector_dnrm2(
    const struct mtxvector * x,
    double * nrm2,
    int64_t * num_flops);
```

The functions `mtxvector_sasum` and `mtxvector_dasum` compute the sum of absolute values, or 1-norm, of a vector. in single and double precision floating point, respectively. If the vector is complex-valued, then the sum of the absolute values of the real and imaginary parts is computed.

```
int mtxvector_sasum(
    const struct mtxvector * x,
    float * asum,
    int64_t * num_flops);
```

```
int mtxvector_dasum(
    const struct mtxvector * x,
    double * asum,
    int64_t * num_flops);
```

The function `mtxvector_iamax` finds the index of the first element having the largest absolute value among all the vector elements. If the vector is complex-valued, then the index points to the first element having the maximum sum of the absolute values of the real and imaginary parts.

```
int mtxvector_iamax(
    const struct mtxvector * x,
    int * iamax);
```

6.2 Matrices

The file `libmtx/linalg/local/matrix.h` defines the type `struct mtxmatrix`. This is a single, abstract data type used to represent a matrix with different options available for the underlying storage and implementation of matrix operations.

The currently supported matrix types are defined by the enum type `enum mtxmatrixtype`, including the following:

- '`mtxbasecoo`' - coordinate format with sequential matrix operations
- '`mtxbasecsr`' - compressed sparse row format with sequential matrix operations
- '`mtxbasedense`' - dense matrices with sequential operations
- '`mtxblasdense`' - dense matrices with BLAS-accelerated operations
- '`mtxompcsr`' - compressed sparse row with shared-memory parallel operations using OpenMP

6.2.1 Creating matrices

This section covers functions that are provided to construct matrices.

The function `mtxmatrix_free` is used to free storage allocated for a matrix.

```
void mtxmatrix_free(struct mtxmatrix * matrix);
```

To create a copy of an existing matrix, use the function `mtxmatrix_init_copy`.

```
int mtxmatrix_init_copy(
    struct mtxmatrix * dst,
    const struct mtxmatrix * src);
```

If storage for a copy of an existing matrix is needed, but the matrix values should not be copied or initialised, use the function `mtxmatrix_alloc_copy`.

```
int mtxmatrix_alloc_copy(
    struct mtxmatrix * dst,
    const struct mtxmatrix * src);
```

To allocate storage for a matrix in *coordinate* format, the function `mtxmatrix_alloc_entries` may be used.

```
int mtxmatrix_alloc_entries(
    struct mtxmatrix * A,
```

```
    enum mtxmatrixtype type,
    enum mtxfield field,
    enum mtxprecision precision,
    enum mtxsymmetry symmetry,
    int num_rows,
    int num_columns,
    int64_t num_nonzeros,
    int idxstride,
    int idxbase,
    const int * rowidx,
    const int * colidx);
```

The desired matrix type, field and precision must be specified, as well as the number of rows, columns and nonzeros. Note that the matrix values are not initialised, and so it is up to the user to initialise them.

If the matrix values are already known, then there are also functions for allocating a matrix and initialising the values directly. This can be done by calling `mtxmatrix_init_entries_`*`field_precision`*, where *`field`* and *`precision`* denote the field (i.e., '`real`', '`complex`' or '`integer`') and precision (i.e., '`single`' or '`double`') of the matrix.

For example, to create a double precision, real matrix, use `mtxmatrix_init_entries_real_double`.

```
int mtxmatrix_init_entries_real_double(
    struct mtxmatrix * A,
    enum mtxmatrixtype type,
    enum mtxsymmetry symmetry,
    int num_rows,
    int num_columns,
    int64_t num_nonzeros,
    const int * rowidx,
    const int * colidx,
    const double * data);
```

The matrix entries are provided by the array `data`, whereas `rowidx` and `colidx` provide the row and column offsets of the nonzeros. All three array must contain '`num_nonzeros`' values.

Note that duplicate entries are allowed, but this may cause some operations (e.g., `mtxmatrix_dot`, `mtxmatrix_nrm2`, `mtxmatrix_sgemv`) to produce incorrect results.

6.2.2 Creating row and column vectors

Matrices of a given size are naturally associated with their row and column vectors. These are vectors whose length is equal to the length of a matrix row or column, respectively. A row vector, x, and a column vector, y, may therefore be used as source and destination vectors, respectively, in a matrix-vector multiplication $y = Ax$.

For convenience, Libmtx provides the functions `mtxmatrix_alloc_row_vector` and `mtxmatrix_alloc_column_vector` for creating row and column vectors that are compatible with a given matrix.

```
int mtxmatrix_alloc_row_vector(
```

```
    const struct mtxmatrix * matrix,
    struct mtxvector * vector,
    enum mtxvectortype vector_type);

int mtxmatrix_alloc_column_vector(
    const struct mtxmatrix * matrix,
    struct mtxvector * vector,
    enum mtxvectortype vector_type);
```

The argument `vector_type` is used to specify the desired, underlying storage type for the row or column vector.

6.2.3 Converting to and from Matrix Market format

In many cases, a matrix may already be available in Matrix Market format. However, for reasons involving both performance and convenience, it is often a good idea to convert the data from Matrix Market format to a more suitable representation before carrying out computations.

To convert a matrix in Matrix Market format to `struct mtxmatrix`, the function `mtxmatrix_from_mtxfile` can be used.

```
int mtxmatrix_from_mtxfile(
    struct mtxmatrix * mtxmatrix,
    enum mtxmatrixtype type,
    const struct mtxfile * mtxfile);
```

The user may use the `type` argument to specify a desired storage format or implementation for `mtxmatrix`.

Conversely, having performed the necessary computations, it is sometimes useful to convert a matrix back to Matrix Market format. For example, to make it easier to output the matrix to a file. To convert `struct mtxmatrix` to a matrix in Matrix Market format, the function `mtxmatrix_to_mtxfile` can be used.

```
int mtxmatrix_to_mtxfile(
    struct mtxfile * mtxfile,
    const struct mtxmatrix * mtxmatrix,
    int64_t num_rows,
    const int64_t * rowidx,
    int64_t num_columns,
    const int64_t * colidx,
    enum mtxfileformat mtxfmt);
```

The resulting Matrix Market represents a matrix in array format if `mtxfmt` is '`mtxfile_array`', or a matrix in coordinate format if `mtxfmt` is '`mtxfile_coordinate`'.

6.2.4 Reading and writing Matrix Market files

For convenience, the function `mtxmatrix_read`, `mtxmatrix_fread` and `mtxmatrix_gzread` are provided to more easily read a matrix from a file in Matrix Market format and convert it to a desired matrix representation. These functions are based on the functions described in Section 4.3 [Reading and writing Matrix Market files], page 15.

```
int mtxmatrix_read(
```

```
    struct mtxmatrix * matrix,
    enum mtxprecision precision,
    enum mtxmatrixtype type,
    const char * path,
    bool gzip,
    int64_t * lines_read,
    int64_t * bytes_read);

int mtxmatrix_fread(
    struct mtxmatrix * matrix,
    enum mtxprecision precision,
    enum mtxmatrixtype type,
    FILE * f,
    int64_t * lines_read,
    int64_t * bytes_read,
    size_t line_max,
    char * linebuf);

int mtxmatrix_gzread(
    struct mtxmatrix * matrix,
    enum mtxprecision precision,
    enum mtxmatrixtype type,
    gzFile f,
    int64_t * lines_read,
    int64_t * bytes_read,
    size_t line_max,
    char * linebuf);
```

The `type` argument specifies which format to use for representing the matrix. If `type` is '`mtxmatrix_auto`', then the underlying matrix is stored in array format or coordinate format according to the format of the Matrix Market file. Otherwise, an attempt is made to convert the matrix to the desired type.

Similarly, the functions `mtxmatrix_write`, `mtxmatrix_fwrite` and `mtxmatrix_gzwrite` are provided to write a matrix to a file in Matrix Market format.

```
int mtxmatrix_write(
    const struct mtxmatrix * matrix,
    const char * path,
    bool gzip,
    const char * fmt,
    int64_t * bytes_written);

int mtxmatrix_fwrite(
    const struct mtxmatrix * matrix,
    FILE * f,
    const char * fmt,
    int64_t * bytes_written);
```

```
int mtxmatrix_gzwrite(
    const struct mtxmatrix * matrix,
    gzFile f,
    const char * fmt,
    int64_t * bytes_written);
```

6.2.5 Level 1 BLAS

It is sometimes useful to treat a matrix as a vector (sometimes called the *vectorisation* of a matrix) and then apply level 1 BLAS operations. This section describes level 1 BLAS operations for `struct mtxmatrix`. These are more or less identical to the level 1 BLAS operations described for vectors in Section 6.1.5 [Level 1 BLAS for vectors], page 45, except that arguments with the type `struct mtxvector` are replaced with `struct mtxmatrix`.

The following Level 1 BLAS operations are supported:

- `swap` — swap two matrices, `Y <- X` and `X <- Y`
- `copy` — copy a matrix, `Y = X`
- `scal` — scale by a constant, `X = a*X`
- `axpy` and `aypx` — add two matrices, `Y = a*X + Y` or `Y = a*Y + X`
- `dot` — Frobenius inner product
- `nrm2` — Frobenius norm
- `asum` — sum of absolute values
- `iamax` — find element with largest absolute value

The function `mtxmatrix_swap` swaps the values of two matrices, whereas `mtxmatrix_copy` copies the values from one matrix to another.

```
int mtxmatrix_swap(struct mtxmatrix * X, struct mtxmatrix * Y);
int mtxmatrix_copy(struct mtxmatrix * X, const struct mtxmatrix * Y);
```

The functions `mtxmatrix_sscal` and `mtxmatrix_dscal` are used to scale a matrix `X` by a floating point constant `a` in single or double precision, respectively. That is, `X = a*X`.

```
int mtxmatrix_sscal(
    float a,
    struct mtxmatrix * X,
    int64_t * num_flops);

int mtxmatrix_dscal(
    double a,
    struct mtxmatrix * X,
    int64_t * num_flops);
```

Note that most of the BLAS functions in Libmtx take an additional argument `num_flops`, which can be used to obtain the number of floating point operations that were carried out. If `num_flops` is '`NULL`', then it is ignored.

The functions `mtxmatrix_saxpy` and `mtxmatrix_daxpy` add a matrix to another matrix multiplied by a single or double precision floating point value, `Y = a*X + Y`.

```
int mtxmatrix_saxpy(
    float a,
```

```
    const struct mtxmatrix * X,
    struct mtxmatrix * Y,
    int64_t * num_flops);

int mtxmatrix_daxpy(
    double a,
    const struct mtxmatrix * X,
    struct mtxmatrix * Y,
    int64_t * num_flops);
```

Similarly, `mtxmatrix_saypx` and `mtxmatrix_daypx` multiply a matrix by a single or double precision floating point scalar before adding the result to another matrix, `Y = a*Y + X`.

```
int mtxmatrix_saypx(
    float a,
    struct mtxmatrix * Y,
    const struct mtxmatrix * X,
    int64_t * num_flops);

int mtxmatrix_daypx(
    double a,
    struct mtxmatrix * Y,
    const struct mtxmatrix * X,
    int64_t * num_flops);
```

The functions `mtxmatrix_sdot` and `mtxmatrix_ddot` compute the Frobenius dot product of two real- or integer-valued matrices.

```
int mtxmatrix_sdot(
    const struct mtxmatrix * X,
    const struct mtxmatrix * Y,
    float * dot,
    int64_t * num_flops);

int mtxmatrix_ddot(
    const struct mtxmatrix * X,
    const struct mtxmatrix * Y,
    double * dot,
    int64_t * num_flops);
```

For complex matrices, the functions `mtxmatrix_cdotu` and `mtxmatrix_zdotu` are used to compute the dot product of the transpose of a complex matrix with another complex matrix, `vec(X)^T*vec(Y)`, where `x^T` denotes the transpose of `x` and `vec(X)` is the vectorisation of the matrix `X`. The functions `mtxmatrix_cdotc` and `mtxmatrix_zdotc` compute the Frobenius dot product of two complex matrices, `vec(X)^H*vec(Y)`, where `x^H` denotes the conjugate transpose of `x`.

```
int mtxmatrix_cdotu(
    const struct mtxmatrix * X,
    const struct mtxmatrix * Y,
```

```
    float (* dot)[2],
    int64_t * num_flops);

int mtxmatrix_zdotu(
    const struct mtxmatrix * X,
    const struct mtxmatrix * Y,
    double (* dot)[2],
    int64_t * num_flops);

int mtxmatrix_cdotc(
    const struct mtxmatrix * X,
    const struct mtxmatrix * Y,
    float (* dot)[2],
    int64_t * num_flops);

int mtxmatrix_zdotc(
    const struct mtxmatrix * X,
    const struct mtxmatrix * Y,
    double (* dot)[2],
    int64_t * num_flops);
```

The functions `mtxmatrix_snrm2` and `mtxmatrix_dnrm2` compute the Frobenius norm of a matrix. in single and double precision floating point, respectively.

```
int mtxmatrix_snrm2(
    const struct mtxmatrix * X,
    float * nrm2,
    int64_t * num_flops);

int mtxmatrix_dnrm2(
    const struct mtxmatrix * X,
    double * nrm2,
    int64_t * num_flops);
```

The functions `mtxmatrix_sasum` and `mtxmatrix_dasum` compute the sum of absolute values of a matrix in single and double precision floating point, respectively. (Note that this is not the same as the 1-norm of a matrix.) If the matrix is complex-valued, then the sum of the absolute values of the real and imaginary parts is computed.

```
int mtxmatrix_sasum(
    const struct mtxmatrix * X,
    float * asum,
    int64_t * num_flops);

int mtxmatrix_dasum(
    const struct mtxmatrix * X,
    double * asum,
    int64_t * num_flops);
```

The function `mtxmatrix_iamax` finds the index of the first element having the largest absolute value among all the matrix elements. If the matrix is complex-valued, then the

index points to the first element having the maximum sum of the absolute values of the real and imaginary parts.

```
int mtxmatrix_iamax(
    const struct mtxmatrix * X,
    int * iamax);
```

6.2.6 Level 2 BLAS

Some of the most useful linear algebra operations are covered by the Level 2 BLAS routines, which involve a matrix and one or more vectors. This section describes level 2 BLAS operations for `struct mtxmatrix`, in particular matrix-vector multiplication.

The following Level 2 BLAS operations are supported:

- `sgemv`, `dgemv` — general, real matrix-vector multiplication, `y = *A*x + *y` or `y = *A'*x + *y`
- `cgemv`, `zgemv` — general, complex matrix-vector multiplication, `y = *A*x + *y`, `y = *A'*x + *y` or `y = *A^H*x + *y`

The function `mtxmatrix_sgemv` multiplies a matrix `A` or its transpose '`A'`' by a real scalar `alpha` () and a vector `x`, before adding the result to another vector `y` multiplied by another real scalar `beta` (). That is, `y = *A*x + *y` or `y = *A'*x + *y`. In this version, the scalars `alpha` and `beta` are given as single precision floating point numbers.

```
int mtxmatrix_sgemv(
    enum mtxtransposition trans,
    float alpha,
    const struct mtxmatrix * A,
    const struct mtxvector * x,
    float beta,
    struct mtxvector * y);
```

If `trans` is '`mtx_notrans`', the matrix `A` is used. If `trans` is '`mtx_trans`', then `A'` is used instead.

The function `mtxmatrix_dgemv` performs the same operation as `mtxmatrix_sgemv`, except that the scalars `alpha` and `beta` are now given as double precision floating point numbers.

```
int mtxmatrix_dgemv(
    enum mtxtransposition trans,
    double alpha,
    const struct mtxmatrix * A,
    const struct mtxvector * x,
    double beta,
    struct mtxvector * y);
```

There are also two analogous routines, `mtxmatrix_cgemv` and `mtxmatrix_zgemv` for the cases where `alpha` and `beta` are given as complex numbers in single and double precision floating point, respectively. These functions can also be used to multiply with the conjugate transpose '`A^H`', if `trans` is '`mtx_conjtrans`'.

```
int mtxmatrix_cgemv(
    enum mtxtransposition trans,
```

```
    float alpha[2],
    const struct mtxmatrix * A,
    const struct mtxvector * x,
    float beta[2],
    struct mtxvector * y);

int mtxmatrix_zgemv(
    enum mtxtransposition trans,
    double alpha[2],
    const struct mtxmatrix * A,
    const struct mtxvector * x,
    double beta[2],
    struct mtxvector * y);
```

7 Commands

This chapter decribes a collection of command-line programs that are provided by Libmtx for workiang with Matrix Market files.

`mtxaxpy` adds two vectors.

`mtxdot` computes the dot product of two vectors.

`mtxgemv` multiplies a general, unsymmetric matrix by a vector.

`mtxinfo` reads a Matrix Market file, validates the contents and displays some high-level information about the Matrix Market object.

`mtxnrm2` computes the Euclidean norm of a vector.

`mtxpartition`
partitions a sparse matrix by rows, columns or nonzeros, or using a graph partitioner such as METIS.

`mtxreorder`
reorders the nonzeros of a sparse matrix, for example, using the Reverse Cuthill-McKee (RCM) ordering (see [E. Cuthill and J. McKee (1969)], page 65).

`mtxscal` scales a vector by a scalar.

`mtxsort` sorts the entries of a dense or sparse matrix, for example, in row- or column-major order.

`mtxspy` draws an image of a matrix sparsity pattern and writes it to a PNG file.

Further details about each program are given in the following sections.

7.1 `mtxaxpy`

The command `mtxaxpy` is used to add two vectors. The result is written to standard output in the form of a Matrix Market file representing the result vector. More specifically, the calculation carried out is '*y* `:=` *alpha*`*`*x* `+` *y*'.

```
mtxaxpy [OPTION..] [alpha] x [y]
```

The positional arguments are:

alpha The scalar floating-point value *alpha*. If this argument is omitted, then *alpha* defaults to '`1.0`'.

x Path to a Matrix Market file containing the vector *x*.

y Path to a Matrix Market file containing the vector *y*. If this argument is omitted, then a vector of zeros of length equal to *x* is used.

In addition, the following options are accepted:

`-z, --gzip, --gunzip, --ungzip`
Filter files through gzip.

`--format=FORMAT`
: Format string for outputting numerical values. For real, double and complex values, the format specifiers '`%e`', '`%E`', '`%f`', '`%F`', '`%g`' or '`%G`' may be used, whereas '`%d`' must be used for integers. Flags, field width and precision can optionally be specified, e.g., '`%+3.1f`'.

`--repeat=N`
: The number of times to repeat the matrix-vector multiplication.

`-q, --quiet`
: Do not print the resulting Matrix Market file to standard output.

`-v, --verbose`
: Print some diagnostics to the standard error stream.

The `mtxaxpy` command can be used in the same way as the STREAM benchmark (see [J.D. McCalpin (2013)], page 65). to measure realistically achievable memory bandwidth of a single core. For example, the following command will run the vector addition one hundred times using a double precision floating point vector with ten million elements:

```
$ ./mtxaxpy --verbose -q --repeat=100 1.0 - < <( \
    N=10000000; \
    printf "%%%%MatrixMarket vector array double general\n"; \
    printf "${N}\n"; \
    for i in $(seq ${N}); do printf "1.0\n"; done)
mtx_read: 39.443018 seconds
mtx_daxpy: 0.013981 seconds
mtx_daxpy: 0.013948 seconds
[...]
```

At eight bytes per element, each vector occupies 80 MB of memory. Assuming that the data is too large to fit in cache, then every vector addition causes 160 MB of data to be read from main memory. Thus, dividing the volume of memory traffic by the time required for a single vector addition, we find that, in this example, the memory throughput is about 11.47 GB/s.

7.2 mtxdot

The command `mtxdot` is used to compute the dot product of two vectors. That is, '*dot* := *x*'*y*', where '*x*' and '*y*' are vectors, and '*x*'' denotes the transpose of '*x*'.

```
mtxdot [OPTION..] x [y]
```

If matrices are provided instead of vectors, then the Frobenius inner product is computed.

The positional arguments are:

x
: Path to a Matrix Market file containing the vector *x*.

y
: Path to a Matrix Market file containing the vector *y*. If this argument is omitted, then a vector of ones of length equal to *x* is used.

In addition, the following options are accepted:

`-z, --gzip, --gunzip, --ungzip`
: Filter files through gzip.

`--format=FORMAT`
: Format string for outputting numerical values. For real, double and complex values, the format specifiers '`%e`', '`%E`', '`%f`', '`%F`', '`%g`' or '`%G`' may be used, whereas '`%d`' must be used for integers. Flags, field width and precision can optionally be specified, e.g., '`%+3.1f`'.

`-q, --quiet`
: Do not print the resulting Matrix Market file to standard output.

`-v, --verbose`
: Print some diagnostics to the standard error stream.

7.3 mtxgemv

The command `mtxgemv` is used to multiply a general, unsymmetric matrix with a vector. The result is written to standard output in the form of a Matrix Market file representing the result vector. More specifically, the calculation carried out is '*y* := *alpha***A***x* + *beta***y*'.

The `mtxgemv` command accepts a number of positional arguments corresponding to the variables in the matrix-vector multiplication:

```
mtxgemv [OPTION..] alpha A [x] [beta] [y]
```

The positional arguments are:

alpha
: The scalar floating-point value *alpha*.

A
: Path to a Matrix Market file containing the matrix *A*.

x
: Path to a Matrix Market file containing the vector *x*. If this argument is omitted or an empty string (i.e., `""`), then a vector of ones of length equal to the number of columns of *A* is used.

beta
: The scalar floating-point value *beta*. If this argument is omitted, then *beta* is set equal to one.

y
: Path to a Matrix Market file containing the vector *y*. If this argument is omitted, then a vector of zeros of length equal to the number of rows of *A* is used.

In addition, the following options are accepted:

`-z, --gzip, --gunzip, --ungzip`
: Filter files through gzip.

`--format=FORMAT`
: Format string for outputting numerical values. For real, double and complex values, the format specifiers '`%e`', '`%E`', '`%f`', '`%F`', '`%g`' or '`%G`' may be used, whereas '`%d`' must be used for integers. Flags, field width and precision can optionally be specified, e.g., '`%+3.1f`'.

`--repeat=N`
: The number of times to repeat the matrix-vector multiplication.

`-q, --quiet`
: Do not print the resulting Matrix Market file to standard output.

`-v, --verbose`
: Print some diagnostics to the standard error stream.

7.4 mtxinfo

The command `mtxinfo` reads a Matrix Market file, validates the contents and displays some high-level information about the Matrix Market object.

```
mtxinfo [OPTION..] FILE
```

The following options are accepted:

`-z, --gzip, --gunzip, --ungzip`
: Filter the file through gzip.

`-v, --verbose`
: Print diagnostics to standard error.

7.5 mtxnrm2

The command `mtxnrm2` is used to compute the Euclidean norm of a vector. That is, '*nrm2* `:= sqrt(`*x*`'`*x*`)`', where '*x*' is a vector and '*x*'' denotes its transpose.

```
mtxnrm2 [OPTION..] x [y]
```

If matrices are provided instead of vectors, then the Frobenius norm of the matrices is computed.

The positional arguments are:

x
: Path to a Matrix Market file containing the vector *x*.

In addition, the following options are accepted:

`-z, --gzip, --gunzip, --ungzip`
: Filter files through gzip.

`--format=FORMAT`
: Format string for outputting numerical values. For real, double and complex values, the format specifiers '`%e`', '`%E`', '`%f`', '`%F`', '`%g`' or '`%G`' may be used, whereas '`%d`' must be used for integers. Flags, field width and precision can optionally be specified, e.g., '`%+3.1f`'.

`-q, --quiet`
: Do not print the resulting Matrix Market file to standard output.

`-v, --verbose`
: Print some diagnostics to the standard error stream.

7.6 mtxpartition

The command `mtxpartition` is used to partition sparse matrices.x

```
mtxpartition [OPTION..] FILE
```

`-z, --gzip, --gunzip, --ungzip`
: Filter the file through gzip

`--format=FORMAT`
: Format string for outputting numerical values. For real, double and complex values, the format specifiers '`%e`', '`%E`', '`%f`', '`%F`', '`%g`' or '`%G`' may be used, whereas '`%d`' must be used for integers. Flags, field width and precision can optionally be specified, e.g., '`%+3.1f`'.

`--row-part-path=FILE`
: Path for outputting row partition as a dense vector in Matrix Market format.

`--col-part-path=FILE`
: Path for outputting column partition as a dense vector in Matrix Market format.

`--part-type=TYPE`
: The method of partitioning algorithm to use: '`nonzeros`' (default), '`rows`', '`columns`', '`2d`' or '`metis`'.

`--nz-parts=N`
: number of parts to use when partitioning nonzeros.

`--nz-part-type=TYPE`
: method of partitioning nonzeros if `--part-type=nonzeros`: '`block`' (default), '`cyclic`' or '`block-cyclic`'.

`--nz-blksize=N`
: block size to use if –nz-part-type is '`block-cyclic`'.

`--row-parts=N`
: number of parts to use when partitioning rows.

`--row-part-type=TYPE method of partitioning`
: rows if `--part-type` is '`rows`' or '`2d`': '`block`' (default), '`cyclic`' or '`block-cyclic`'.

`--row-blksize=N`
: block size to use if `--row-part-type` is '`block-cyclic`'.

`--column-parts=N`
: number of parts to use when partitioning columns.

`--column-part-type=TYPE`
: method of partitioning columns if `--part-type` is '`columns`' or '`2d`': '`block`' (default), '`cyclic`' or '`block-cyclic`'.

`--column-blksize=N`
: block size to use if `--column-part-type` is '`block-cyclic`'.

`-q, --quiet`
: Do not print the resulting Matrix Market file to standard output.

`-v, --verbose`
: Print diagnostics to standard error.

7.7 mtxreorder

The command `mtxreorder` is used to reorder the rows and columns of a sparse matrix, for example, using the Reverse Cuthill-McKee (RCM) ordering (see [E. Cuthill and J. McKee (1969)], page 65).

```
mtxreorder [OPTION..] FILE
```

The following options are accepted:

`-z, --gzip, --gunzip, --ungzip`
: Filter the file through gzip

`--format=FORMAT`
: Format string for outputting numerical values. For real, double and complex values, the format specifiers '`%e`', '`%E`', '`%f`', '`%F`', '`%g`' or '`%G`' may be used, whereas '`%d`' must be used for integers. Flags, field width and precision can optionally be specified, e.g., '`%+3.1f`'.

`--rowperm-path=FILE`
: Path for outputting row permutation as a dense vector in Matrix Market format.

`--colperm-path=FILE`
: Path for outputting column permutation as a dense vector in Matrix Market format.

`--ordering=ORDERING`
: The reordering algorithm to use. For now, the only supported algorithm is '`rcm`'.

`--rcm-starting-row=N`
: Starting row for the RCM algorithm. The default value is '`0`', which means to choose a starting row automatically.

`-q, --quiet`
: Do not print the resulting Matrix Market file to standard output.

`-v, --verbose`
: Print diagnostics to standard error.

7.8 mtxscal

The command `mtxscal` is used to scale a vector by a scalar, floating point value. The result is written to standard output in the form of a Matrix Market file representing the scaled vector. More specifically, the calculation carried out is '*`x := alpha*x`*', where '*`x`*' is a vector and '*`alpha`*' is a scalar.

The `mtxscal` command accepts the following positional arguments:

```
mtxscal [OPTION..] alpha x
```

The positional arguments are:

`alpha`
: The scalar floating-point value *`alpha`*.

`x`
: Path to a Matrix Market file containing the vector *`x`*.

In addition, the following options are accepted:

`-z, --gzip, --gunzip, --ungzip`
: Filter files through gzip.

`--format=FORMAT`
: Format string for outputting numerical values. For real, double and complex values, the format specifiers '`%e`', '`%E`', '`%f`', '`%F`', '`%g`' or '`%G`' may be used, whereas '`%d`' must be used for integers. Flags, field width and precision can optionally be specified, e.g., '`%+3.1f`'.

`--repeat=N`
: The number of times to repeat the matrix-vector multiplication.

`-q, --quiet`
: Do not print the resulting Matrix Market file to standard output.

`-v, --verbose`
: Print some diagnostics to the standard error stream.

7.9 mtxsort

The command `mtxsort` is used to sort the entries of a dense or sparse matrix, for example, in row- or column-major order.

```
mtxsort [OPTION..] FILE
```

The following options are accepted:

`-z, --gzip, --gunzip, --ungzip`
: Filter the file through gzip.

`--format=FORMAT`
: Format string for outputting numerical values. For real, double and complex values, the format specifiers '`%e`', '`%E`', '`%f`', '`%F`', '`%g`' or '`%G`' may be used, whereas '`%d`' must be used for integers. Flags, field width and precision can optionally be specified, e.g., '`%+3.1f`'.

`--sorting=SORTING`
: The ordering to use when sorting the data. This is either '`row-major`' or '`column-major`'. By default, '`row-major`' is used.

`-q, --quiet`
: Do not print the resulting Matrix Market file to standard output.

`-v, --verbose`
: Print diagnostics to standard error.

7.10 mtxspy

The command `mtxspy` draws an image of the sparsity pattern of a matrix, saving it to a PNG file. This command is only available if Libmtx is compiled with libpng support.

```
mtxspy [OPTION..] FILE
```

The following options are accepted:

`--output-path=FILE`
: Output path for the PNG image file. If not specified, the default output path is '`out.png`'.

`-z, --gzip, --gunzip, --ungzip`
: Filter files through gzip.

`--max-height=M`
`--max-width=N`
: Maximum width and height of the rendered image in pixels. The default maximum image size is 1000-by-1000 pixels.

`fgcolor=COLOR`
`bgcolor=COLOR`
: Foreground and background colors used to indicate sparse matrix entries that are present and absent in the sparsity pattern, respectively. Colors are specified in hexadecimal, optionally prefixed with a '`#`' character (e.g., '`#38B6F1`'). The default a black foreground and white background.

`gamma=GAMMA`
: Gamma value to embed in the PNG.

`--title=TEXT`
`--author=TEXT`
`--description=TEXT`
`--copyright=TEXT`
`--email=TEXT`
`--url=TEXT`
: These options specify various text fields that may be stored in the PNG image to provide additional metadata about the image.

`-v, --verbose`
: Print diagnostics to standard error.

References

- Ronald F. Boisvert, Roldan Pozo, and Karin Remington. *The Matrix Market exchange formats: Initial design*. Technical Report NISTIR 5935, National Institute of Standards and Technology, Gaithersburg, MD, USA, December 1996.
- William Gropp, Ewing Lusk and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*, MIT Press, 1999. ISBN 978-0-262-57133-3.
- The HDF Group. *Hierarchical data format version 5*, 1997–2022. `http://www.hdfgroup.org/HDF5`.
- National Institute of Standards and Technology (NIST). *Matrix Market*. Mathematical and Computational Sciences Division, Information Technology Laboratory, NIST, 10 May 2007. `https://math.nist.gov/MatrixMarket/index.html`.
- National Institute of Standards and Technology (NIST). *ANSI C library for Matrix Market I/O*. Mathematical and Computational Sciences Division, Information Technology Laboratory, NIST, 2 May 2000. `https://math.nist.gov/MatrixMarket/mmio-c.html`.
- Tim Davis, Yifan Hu, and Scott Kolodziej. *SuiteSparse Matrix Collection*. Computer Science and Engineering Department, Texas A&M University. `https://sparse.tamu.edu/`.
- E. Cuthill and J. McKee. *Reducing the bandwidth of sparse symmetric matrices*. Proc. 24th Nat. Conf. ACM, pages 157172, 1969.
- J. A. George and J. W-H. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, 1981.
- Zhang Xianyi, *OpenBLAS*. `https://www.openblas.net/`.
- J. D. McCalpin *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. Department of Computer Science School of Engineering and Applied Science, University of Virginia, Charlottesville, Virginia. June 9, 2020. `https://www.cs.virginia.edu/stream/`.

Appendix A GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released

under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any,

be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their

titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

General index

O

P

R

S

T

U

V

W

Z

Function index

Data type index

E

S

Program index