cppmat Documentation

Tom de Geus

Apr 25, 2019

Contents

Note: This library is free to use under the MIT license. Any additions are very much appreciated, in terms of suggested functionality, code, documentation, testimonials, word of mouth advertisement, ... Bugs or feature requests can be filed on GitHub. As always, the code comes with no guarantee. None of the developers can be held responsible for possible mistakes.

Tip: This document should be considered as a quick-start guide. A lot effort has been spent on the readability of the code itself (in particular the `*.h` files should be instructive). One is highly encouraged to answer more advanced questions that arise from this guide directly using the code. Download buttons to the relevant files are included throughout this reader.

This header-only module provides C++ classes and several accompanying methods to work with n-d arrays and/or tensors. It's usage, programmatically and from a compilation perspective, is really simple. One just has to `#include <cppmat/cppmat.h>` and tell your compiler where cppmat is located (and to use the C++14 or younger standard). Really, that's it!

CHAPTER 1

Overview

The following dynamically sized classes can be used.

Class	Description
cppmat::array	array of arbitrary rank
cppmat::matrix	matrix (array of rank 2)
cppmat::vector	vector (array of rank 1)
cppmat::symmetric::matrix	symmetric, square, matrix
cppmat::diagonal::matrix	diagonal, square, matrix
cppmat::cartesian::tensor4	4th-order tensor
cppmat::cartesian::tensor2	2nd-order tensor
cppmat::cartesian::tensor2s	2nd-order symmetric tensor
cppmat::cartesian::tensor2d	2nd-order diagonal tensor
cppmat::cartesian::vector	1st-order tensor (a.k.a. vector)

Each of these classes has a fixed size equivalent (that is usually more efficient):

Fixed size	Dynamical size
cppmat::tiny::array	*cppmat::array*
cppmat::tiny::matrix	*cppmat::matrix*
cppmat::tiny::vector	*cppmat::vector*
cppmat::tiny::symmetric::matrix	*cppmat::symmetric::matrix*
cppmat::tiny::diagonal::matrix	*cppmat::diagonal::matrix*
cppmat::tiny::cartesian::tensor4	*cppmat::cartesian::tensor4*
cppmat::tiny::cartesian::tensor2	*cppmat::cartesian::tensor2*
cppmat::tiny::cartesian::tensor2s	*cppmat::cartesian::tensor2s*
cppmat::tiny::cartesian::tensor2d	*cppmat::cartesian::tensor2d*
cppmat::tiny::cartesian::vector	*cppmat::cartesian::vector*

Each fixed size class has an equivalent which can view a `const`-pointer (with limited functionality):

View pointer	Fixed size
cppmat::view::array	*cppmat::tiny::array*
cppmat::view::matrix	*cppmat::tiny::matrix*
cppmat::view::vector	*cppmat::tiny::vector*
cppmat::view::symmetric::matrix	*cppmat::tiny::symmetric::matrix*
cppmat::view::diagonal::matrix	*cppmat::tiny::diagonal::matrix*
cppmat::view::cartesian::tensor4	*cppmat::tiny::cartesian::tensor4*
cppmat::view::cartesian::tensor2	*cppmat::tiny::cartesian::tensor2*
cppmat::view::cartesian::tensor2s	*cppmat::tiny::cartesian::tensor2s*
cppmat::view::cartesian::tensor2d	*cppmat::tiny::cartesian::tensor2d*
cppmat::view::cartesian::vector	*cppmat::tiny::cartesian::vector*

CHAPTER 2

Example

```
#include <cppmat/cppmat.h>

int main()
{
    cppmat::array<double> A({10,10,10});

    A(0,0,0) = ...

    ...

    return 0;
}
```

CHAPTER 3

Contents

3.1 cppmat

3.1.1 cppmat::array

[var_regular_array.h, var_regular_array.hpp]

A C++ class for dynamically sized arrays or arbitrary rank. For example, a rank 3 array is allocated as follows:

```
#include <cppmat/cppmat.h>

int main()
{
    cppmat::array<double> A({10,10,10});

    A(0,0,0) = ...

    ...

    std::cout << A << std::endl;

    return 0;
}
```

Tip:

- If you know that you will work exclusively with a rank 1 or 2 array (i.e. a vector or a matrix), consider using *cppmat::vector*, *cppmat::matrix*, *cppmat::symmetric::matrix*, and , *cppmat::diagonal::matrix*. This can enhance readability and/or efficiency.
- If your array is not very big and its size is known at compile time consider using *cppmat::tiny::array* (or the fixed size equivalents of the other classes). This avoids dynamic memory allocation, and usually speeds-up your code.

- If your array is part of an external array (for example a bigger array) which you want to just read from, consider using *cppmat::view::array*.
- To format the print use the regular C++ mechanism, e.g. `std::cout << std::setw(5) << std::setprecision(3) << A << std::endl;`

Methods

- `A(i,j,k)`

  Returns the entry at `(i,j,k)`. Use this to read or write.

  A negative index may also be used (in that case the indices have to be `int`) which counts down from the last index along that axis. For example `A(-1,-1,-1)` in the last index of the above array. This implies some extra operations, so if you do not use this feature input the indices as `size_t`.

  The number of indices (i.e. `A(i)`, `A(i,j)`, `A(i,j,k)`, ...) may be lower or equal to the rank, all 'omitted' indices are assumed to be zero.

  See *Indexing* for additional directives.

- `A[i]`

  Returns the `i`-th entry of the plain storage. Use this to read or write.

- `A.at(first, last)`

  Returns the entry `{i,j,k}`, which are stored in a list. The function takes an iterator to the first and the last index of this list. See *Advanced indexing*.

- `A.item(i,j,k)`

  Returns an iterator to the entry at `(i,j,k)`.

- `A.index(i)`

  Returns an iterator to the `i`-th entry of the plain storage.

- `A.data(),A.begin(),A.end()`

  Return an iterator to the data, the first, or the last entry of the matrix.

- `A.rank()`

  Returns the ranks of the array (i.e. the number of axes).

- `A.size()`

  Returns the total number of entries in the matrix.

- `A.shape(i)`

  Returns the shape along dimension `i` (a negative number may be used that counts down from the last axis, e.g. `A.shape(-1)` is the same as `A.shape(A.rank()-1)`.

- `A.shape()`

  Returns the shape along all dimensions (vector).

- `A.resize({...}[, D])`

  Resize the matrix. Enter a value to initialize all allocated entries.

- `A.reshape({...})`

  Change the shape of the matrix. It is required that the total number of entries does not change.

- `A.chrank(N)`

  Change the rank to `N` (with shape 1 along the added axes). A reduction of rank is only allowed if the shape is 1 along the reduced axes.

- `A.setZero()`, `A.setOnes()`, `A.setConstant(D)`, `A.setArange()`, `A.setRandom([start, end])`

  Set all entries to zero or one, a constant, the index in the flat storage, or a random value.

- `A.setCopy(first[, last])`

  Copy the individual entries from some external object that is specified using iterators. Note that the flat-size has to match, i.e. `last - first == size()`.

- `A.copyTo(first[, last])`

  Copy the individual entries to an external iterator.

- `A.abs()`

  Returns an array with the absolute values of each entry.

- `A.norm()`

  Returns the norm (sum of absolute values).

- `A.argmin()`,`A.argmax()`

  Return the plain storage index of the minimum/maximum.

- `A.min([axis])`,`A.max([axis])`

  Return the minimum or the maximum entry.

- `A.sum([axis])`

  Return the sum of all entries, or along one or more axes.

- `A.mean([axis])`

  Return the mean of all entries, or along one or more axes.

- `A.average(weights[, axis, normalize])`

  Compute the weighted average of all entries, or along one or more axes. See NumPy and Wikipedia. Optionally the result can be returned without normalization.

- `A.where()`

  Returns a vector with the plain storage indices of all non-zero entries.

- `A.equal(D)`, `A.not_equal(D)`, `A.greater(D)`, `A.greater_equal(D)`, `A.less(D)`, `A.less_equal(D)`

  Return array of booleans, based on the condition.

- `A.slice(...)`

  Returns a slice of the array. The input are `std::vector<size_t>` with the indices to select along that axis (these vectors can be also input using the `{...}` syntax). An empty vector (or simply `{}`) implies that all indices along that axis are selected.

Tip: If you use something other than `size_t` as the type for indices (e.g. `int`), the functions `size`, `shape`, `rank`, and `strides` can be templated to directly get the type you want. For example:

```
cppmat::array<double> A({10,10,10});

for ( int i = 0 ; i < A.size<int>() ; ++i )
   ...
```

(Named) constructors

- `cppmat::array<double>(shape)`

  Allocate to a certain shape, nothing is initialized. The `shape` has to be specified as a `std::vector<size_t>`, from which the rank is automatically deduced. Alternatively the `{...}` notation can be used, to avoid a separate variable.

- `cppmat::array<double>::Random(shape[, start, end])`

  Allocate to a certain shape, set entries to a random value.

- `cppmat::array<double>::Arange(shape)`

  Allocate to a certain shape, set entries to its index in the flat storage.

- `cppmat::array<double>::Zero(shape)`

  Allocate to a certain shape, set all entries to zero.

- `cppmat::array<double>::Ones(shape)`

  Allocate to a certain shape, set all entries to one.

- `cppmat::array<double>::Constant(shape, constant)`

  Allocate to a certain shape, set all entries to a certain constant.

- `cppmat::array<double>::Copy(shape, first[, last])`

  Allocate to a certain shape, copy the individual entries from some external object that is specified using iterators. Note that the flat-size has to match, i.e. `last - first == size()`.

External operations

- `cppmat::array<double> = cppmat::min(A, B)`

  Construct an array taking the minimum of two arrays for each entry.

- `cppmat::array<double> = cppmat::max(A, B)`

  Construct an array taking the maximum of two arrays for each entry.

Indexing

In principle the number of indices should match the rank of the array (i.e. `A.rank()`). Though one can:

- Reference to a certain index using a higher-dimensional equivalent. For example:

```
cppmat::array<double> A({10,10});

A(5,5,0) = ...
```

is perfectly acceptable. Note that higher-dimensions can only be trailing ones, using for example `A(0,5,5)` is not acceptable, nor is, of course, `A(5,5,1)`.

- Refer to the beginning of a block (e.g. a row) by omitting the trailing zero indices. For example, a pointer to the beginning of the second row of the above matrix is obtained by `&A(1)` (which is fully equivalent to `&A(1,0)`).

Tip: A negative index may also be used (in that case the indices have to be `int`) which counts down from the last index along that axis. For example `A(-1,-1)` in the last index of the above matrix. To input any *periodic* index (i.e. to turn-off the bound-checks) use `.setPeriodic(true)` on the array object. In that case `A(-1,-1) == A(10,10)` for the above matrix.

This does involve some extra operations, so if you do not use this feature input the indices as `size_t`.

Advanced indexing

To allow an arbitrary number of indices at runtime (i.e. the case in which the number of indices is not known at compile time), `cppmat::array` can also be supplied with the indices stored in a list, using the `.at(first,last)`, where `first` and `last` are iterators to the beginning and the end of this list of indices. When the indices are also stored in a `cppmat::array` these iterators can be easily obtained using `.item(i,j)`. Consider this example:

```
#include <cppmat/cppmat.h>

int main()
{
  // example matrix
  // --------------

  cppmat::array<size_t> A({2,4});

  A(0,0) =  0; A(0,1) =  1; A(0,2) =  2; A(0,3) =  3;
  A(1,0) = 10; A(1,1) = 11; A(1,2) = 12; A(1,3) = 13;

  // view, based on list of indices
  // ------------------------------

  cppmat::array<size_t> index({2,2});

  index(0,0) = 0; index(0,1) = 1;
  index(1,0) = 1; index(1,1) = 2;

  for ( size_t i = 0 ; i < index.shape(0) ; ++i )
    std::cout << A.at(index.item(i), index.item(i)+index.shape(1)) << std::endl;

  return 0;
}
```

Storage

The matrix is stored row-major. For a 2-d matrix of size (3,4) this implies the following storage

```
[[0, 1, 2, 3],
 [4, 5, 6, 7]]
```

The `strides` indicate per axis how many entries one needs to skip to proceed to the following entry along that axis. For this example

```
strides = [4, 1]
```

Note: References

- Row- and column-major order (Wikipedia)
- Reduction (sum) along arbitrary axes of a multidimensional array (StackOverflow)

Tip: One can switch back-and-forth between matrix indices and the plain storage using the `compress` and `decompress` functions. For example:

```
#include <cppmat/cppmat.h>

int main()
{
  cppmat::array<size_t> A({2,4});

  std::cout << A.compress(1,2) << std::endl;

  std::vector<size_t> idx = A.decompress(6);

  for ( auto &i : idx )
    std::cout << i << ", ";
  std::cout << std::endl;

  return 0;
}
```

Prints

```
6
1, 2,
```

3.1.2 cppmat::matrix

[`var_regular_matrix.h`, `var_regular_matrix.hpp`]

Class for 2-d matrices. For example:

```
#include <cppmat/cppmat.h>

int main()
{
    cppmat::matrix<double> A(10,10);

    A(0,0) = ...

    ...

    return 0;
}
```

The entire interface is the same as for *cppmat::array*, though there is obviously no `chrank` method.

3.1.3 cppmat::vector

[`var_regular_vector.h`, `var_regular_vector.hpp`]

Class for 1-d matrices (a.k.a. vectors). For example:

```
#include <cppmat/cppmat.h>

int main()
{
    cppmat::vector<double> A(10);

    A(0) = ...

    ...

    return 0;
}
```

The entire interface is the same as for *cppmat::array*, though there is obviously no `chrank` method.

Tip: One can almost seamlessly switch between `std::vector` and `cppmat::vector`. For example the following would work:

```
std::vector<double> A = cppmat::vector<double>::Random(10);
```

3.2 cppmat::symmetric

3.2.1 cppmat::symmetric::matrix

[`var_symmetric_matrix.h`, `var_symmetric_matrix.hpp`]

Square, symmetric, matrix, whereby only the upper-diagonal components are stored:

```
[ X, X, X ;
     X, X ;
        X ]
```

The remaining components are inferred from symmetry. This offers memory advantages, but also computational advantages as the library is fed with additional knowledge of the matrix.

```
#include <cppmat/cppmat.h>

int main()
{
    cppmat::symmetric::matrix<double> A(3,3);

    A(0,0) = ...

    // A(0,1) = ... -> same as A(1,0) = ...
```

(continues on next page)

(continued from previous page)

```
    ...

    std::cout << A << std::endl;

    return 0;
}
```

Storage

The storage order is as follows:

```
[ 0, 1, 2 ;
     3, 4 ;
        5 ]
```

For an `NxN` matrix (square by definition), the component `(i,j)` can be extracted by

```
if (i <= j) i*N - (i-1)*i/2 + j - i;
else        j*N - (j-1)*j/2 + i - j;
```

3.3 cppmat::diagonal

3.3.1 cppmat::diagonal::matrix

[`var_diagonal_matrix.h`, `var_diagonal_matrix.hpp`]

Square, diagonal, matrix, whereby only the diagonal components are stored:

```
[ X       ;
     X    ;
        X ]
```

The remaining components are imposed to be zero. This offers memory advantages, but also computational advantages as the library is fed with additional knowledge of the matrix.

```
#include <cppmat/cppmat.h>

int main()
{
    cppmat::diagonal::matrix<double> A(3,3);

    A(0,0) = ...

    // A(0,1) = ... -> not allowed
    // ... = A(0,1) -> allowed, returns zero

    ...

    std::cout << A << std::endl;

    return 0;
}
```

Storage

The storage order is as follows:

```
[ 0       ;
      1   ;
          2 ]
```

3.4 cppmat::cartesian

Provides classes for 4th- and 2nd order tensors and vectors. For example, a fourth order identity tensor in 3-D is obtained as follows:

```
#include <cppmat/cppmat.h>

int main()
{
  cppmat::cartesian::tensor4<double> A = cppmat::cartesian::tensor4<double>::I(3);

  ...

  std::cout << A << std::endl;

  return 0;
}
```

Tip: If you know that you will work in a fixed (and small) number of dimensions (e.g. 2 or 3), please consider using *cppmat::tiny::cartesian* instead of *cppmat::cartesian*. This is generally more efficient as it can take advantage of the knowledge that the arrays are fixed size and relatively small. Also several loops are unrolled.

Tip: The notation can be shortened to:

```
#include <cppmat/cppmat.h>

using T4 = cppmat::cartesian::tensor4<double>;

int main()
{
  T4 A = T4::I(3);

  ...

  return 0;
}
```

3.4.1 Classes

cppmat::cartesian::tensor4

[var_cartesian_tensor4.h, var_cartesian_tensor4.hpp]

4th-order tensor (rank 4 tensor) of arbitrary dimension.

```
cppmat::cartesian::tensor4<double> A(3);

A(0,0,0,0) = ...
```

cppmat::cartesian::tensor2

[var_cartesian_tensor2.h, var_cartesian_tensor2.hpp]

2nd-order tensor (rank 2 tensor) of arbitrary dimension.

```
cppmat::cartesian::tensor2<double> A(3);

A(0,0) = ...
```

cppmat::cartesian::tensor2s

[var_cartesian_tensor2s.h, var_cartesian_tensor2s.hpp]

Symmetric 2nd-order tensor.

```
cppmat::cartesian::tensor2s<double> A(3);

A(0,0) = ...
```

For example, for the case of 3 dimensions, the following components are stored:

```
[ X , X , X ;
      X , X ;
          X ]
```

The remaining components are inferred from symmetry. See *cppmat::symmetric::matrix*.

cppmat::cartesian::tensor2d

[var_cartesian_tensor2d.h, var_cartesian_tensor2d.hpp]

diagonal 2nd-order tensor.

```
cppmat::cartesian::tensor2d<double> A(3);

A(0,0) = ...
```

For example, for the case of 3 dimensions, the following components are stored:

```
[ X         ;
      X     ;
          X ]
```

The remaining components are imposed to be zero. See *cppmat::diagonal::matrix*.

cppmat::cartesian::vector

[var_cartesian_vector.h, var_cartesian_vector.hpp]

Vector (rank 1 tensor) of arbitrary dimension. For example:

```
cppmat::cartesian::vector<double> A(3);

A(0) = ...
```

Note: Because of the flexibility of C++ it is easy to switch between these specialized classes and the more general `cppmat::cartesian::tensor2` classes. For example, the following will work:

```
using T2  = cppmat::cartesian::tensor2 <double>;
using T2d = cppmat::cartesian::tensor2d<double>;

T2d I = T2d::I(3);
T2  A = I;
```

or even

```
T2 I = T2d::I(3);
```

Also arithmetic works:

```
T2d A = 3.0 * I;
```

Note that it is even possible to perform arithmetic between the three different 2nd-order tensor classes, whereby the output type depends on the type of operator.

Finally, all the *Methods* accept all three classes - `cppmat::cartesian::tensor2`, `cppmat::cartesian::tensor2s`, `cppmat::cartesian::tensor2d` - allowing their usage without any prior type casting. In fact the methods will often perform better for the specialized classes since fewer operations are needed.

Note: The easy automatic conversion described above is not possible from a class to another where more assumptions on the structure are made (e.g. from `cppmat::cartesian::tensor2` to `cppmat::cartesian::tensor2d`) because information is (potentially) lost.

3.4.2 Methods

All the methods of *cppmat::array* (or *cppmat::matrix*, *cppmat::symmetric::matrix*, *cppmat::symmetric::matrix*, or *cppmat::vector*) are overloaded. In addition, the following tensor algebra is available.

Note: Below the rank can be inferred from the indices, but should be easy to understand even without them. Pseudo-code is used to introduce the methods. For the first method it is short for:

```
cppmat::cartesian::tensor4<double> A = cppmat::cartesian::tensor4<double>::I(3);
cppmat::cartesian::tensor2<double> B = cppmat::cartesian::tensor2<double>::I(3);

cppmat::cartesian::tensor2<double> C = A.ddot(B);
```

Finally, each occurrence of `cppmat::cartesian::tensor2` can be replaced by `cppmat::cartesian::tensor2s` or `cppmat::cartesian::tensor2d`. The latter two often perform better.

- `cppmat::cartesian::tensor4<X>`:
  - `cppmat::cartesian::tensor4<X> C = A.ddot(const cppmat::cartesian::tensor4<X> &B)`

    Double tensor contraction : $C_{ijmn} = A_{ijkl}B_{lkmn}$
  - `cppmat::cartesian::tensor2<X> C = A.ddot(const cppmat::cartesian::tensor2<X> &B)`

    Double tensor contraction $C_{ij} = A_{ijkl}B_{lk}$
  - `cppmat::cartesian::tensor4<X> C = A.T()`

    Transposition $C_{lkji} = A_{ijkl}$
  - `cppmat::cartesian::tensor4<X> C = A.LT()`

    Left transposition $C_{jikl} = A_{ijkl}$
  - `cppmat::cartesian::tensor4<X> C = A.RT()`

    Right transposition $C_{ijlk} = A_{ijkl}$
- `cppmat::cartesian::tensor2<X>`:
  - `cppmat::cartesian::tensor2<X> C = A.ddot(const cppmat::cartesian::tensor4<X> &B)`

    Double tensor contraction $C_{kl} = A_{ij}B_{jikl}$
  - `X C = A.ddot(const cppmat::cartesian::tensor2<X> &B)`

    Double tensor contraction $C = A_{ij}B_{ji}$
  - `cppmat::cartesian::tensor2<X> C = A.dot(const cppmat::cartesian::tensor2<X> &B)`

    Tensor contraction $C_{ik} = A_{ij}B_{jk}$
  - `cppmat::cartesian::vector<X> C = A.dot(const cppmat::cartesian::vector<X> &B)`

    Tensor contraction $C_i = A_{ij}B_j$
  - `cppmat::cartesian::tensor4<X> C = A.dyadic(const cppmat::cartesian::tensor2<X> &B)`

    Dyadic product $C_{ijkl} = A_{ij}B_{kl}$
  - `cppmat::cartesian::tensor2<X> C = A.T()`

    Transposition $C_{ji} = A_{ij}$
  - `X C = A.trace()`

    The trace of the tensor (i.e. the sum of the diagonal components) $C = A_{ii}$
  - `X C = A.det()`

    The determinant $C = \det \underline{A}$

  - `cppmat::cartesian::tensor2<X> C = A.inv()`

    The inverse $C_{ij} = A^{-1}_{ij}$

- `cppmat::cartesian::vector<X>`:
  - `X C = A.dot(const cppmat::cartesian::vector<X> &B)`

    Tensor contraction $C = A_i B_i$
  - `cppmat::cartesian::vector<X> C = A.dot(const cppmat::cartesian::tensor2<X> &B)`

    Tensor contraction $C_j = A_i B_{ij}$
  - `cppmat::cartesian::tensor2<X> C = A.dyadic(const cppmat::cartesian::vector<X> &B)`

    Dyadic product $C_{ij} = A_i B_j$
  - `cppmat::cartesian::vector<X> C = A.cross(const cppmat::cartesian::vector<X> &B)`

    Cross product $\vec{C} = \vec{A} \otimes \vec{B}$

Note: One can also call the methods as functions using `cppmmat::ddot(A,B)`, `cppmmat::dot(A,B)`, `cppmmat::dyadic(A,B)`, `cppmmat::cross(A,B)`, `cppmmat::T(A)`, `cppmmat::RT(A)`, `cppmmat::LT(A)`, `cppmmat::inv(A)`, `cppmmat::det(A)`, and `cppmmat::trace(A)`. This is fully equivalent (in fact the class methods call these external functions).

3.5 cppmat::tiny

3.5.1 cppmat::tiny::array

[`fix_regular_array.h`, `fix_regular_array.hpp`]

Class for fixed size, small, n-d arrays. For example for a rank 3 array:

```
#include <cppmat/cppmat.h>

int main()
{
    cppmat::tiny::array<double,3,10,10,10> A;

    A(0,0,0) = ...

    ...

    return 0;
}
```

Note that the first 'shape' is the rank of the array, the rest are the shape along each axis.

Compared to *cppmat::array* the size of the array cannot be dynamically changed. Consequently there is no dynamic memory allocation, often resulting in faster behavior. For the rest, most methods are the same as for *cppmat::array*, though sometimes slightly more limited in use.

3.5.2 cppmat::tiny::matrix

[fix_regular_matrix.h, fix_regular_matrix.hpp]

Class for fixed size, small, matrices. For example:

```
#include <cppmat/cppmat.h>

int main()
{
    cppmat::tiny::matrix<double,10,10> A;

    A(0,0) = ...

    ...

    return 0;
}
```

Most methods are the same as for *cppmat::matrix*.

3.5.3 cppmat::tiny::vector

[fix_regular_vector.h, fix_regular_vector.hpp]

Class for fixed size, small, matrices. For example:

```
#include <cppmat/cppmat.h>

int main()
{
    cppmat::tiny::vector<double,10> A;

    A(0) = ...

    ...

    return 0;
}
```

Most methods are the same as for *cppmat::vector*.

3.5.4 cppmat::tiny::symmetric::matrix

[fix_symmetric_matrix.h, fix_symmetric_matrix.hpp]

Class for fixed size, small, symmetric, matrices. For example:

```
#include <cppmat/cppmat.h>

int main()
{
    cppmat::tiny::symmetric::matrix<double,10,10> A;

    A(0,0) = ...
```

(continues on next page)

(continued from previous page)

```
    ...

    return 0;
}
```

Most methods are the same as for *cppmat::symmetric::matrix*.

3.5.5 cppmat::tiny::diagonal::matrix

[fix_diagonal_matrix.h, fix_diagonal_matrix.hpp]

Class for fixed size, small, symmetric, matrices. For example:

```
#include <cppmat/cppmat.h>

int main()
{
    cppmat::tiny::diagonal::matrix<double,10,10> A;

    A(0,0) = ...

    ...

    return 0;
}
```

Most methods are the same as for *cppmat::diagonal::matrix*.

3.5.6 cppmat::tiny::cartesian

cppmat::tiny::cartesian::tensor4

[fix_cartesian_tensor4.h, fix_cartesian_tensor4.hpp]

Class for fixed size, small, fourth order tensors. For a 3-d tensor

```
#include <cppmat/cppmat.h>

int main()
{
    cppmat::tiny::cartesian::tensor4<double,3> A;

    A(0,0,0,0) = ...

    ...

    return 0;
}
```

Most methods are the same as for *cppmat::cartesian::tensor4*.

cppmat::tiny::cartesian::tensor2

[fix_cartesian_tensor2.h, fix_cartesian_tensor2.hpp]

Class for fixed size, small, second order tensors. For a 3-d tensor

```
#include <cppmat/cppmat.h>

int main()
{
    cppmat::tiny::cartesian::tensor2<double,3> A;

    A(0,0) = ...

    ...

    return 0;
}
```

Most methods are the same as for *cppmat::cartesian::tensor2*.

cppmat::tiny::cartesian::tensor2s

[fix_cartesian_tensor2s.h, fix_cartesian_tensor2s.hpp]

Class for fixed size, small, symmetric, second order tensors. For a 3-d tensor

```
#include <cppmat/cppmat.h>

int main()
{
    cppmat::tiny::cartesian::tensor2s<double,3> A;

    A(0,0) = ...

    ...

    return 0;
}
```

Most methods are the same as for *cppmat::cartesian::tensor2s*.

cppmat::tiny::cartesian::tensor2d

[fix_cartesian_tensor2d.h, fix_cartesian_tensor2d.hpp]

Class for fixed size, small, diagonal, second order tensors. For a 3-d tensor

```
#include <cppmat/cppmat.h>

int main()
{
    cppmat::tiny::cartesian::tensor2d<double,3> A;

    A(0,0) = ...

    ...

    return 0;
}
```

Most methods are the same as for *cppmat::cartesian::tensor2d*.

cppmat::tiny::cartesian::vector

[fix_cartesian_vector.h, fix_cartesian_vector.hpp]

Class for fixed size, small, vector. For a 3-d vector

```
#include <cppmat/cppmat.h>

int main()
{
    cppmat::tiny::cartesian::vector<double,3> A;

    A(0,0) = ...

    ...

    return 0;
}
```

Most methods are the same as for *cppmat::cartesian::vector*.

3.6 cppmat::view

3.6.1 cppmat::view::array

[map_regular_array.h, map_regular_array.hpp]

This class can be used to 'view' a `const` external pointer. This can be useful to refer to a part of a bigger array. For example:

```
#include <cppmat/cppmat.h>

int main()
{
    cppmat::array<double> container = cppmat::array<double>::Arange({100,4,2});

    cppmat::view::array<double,2,4,2> view;

    view.setMap(&container(10)); // equivalent to "view.setMap(&container(10,0,0));"

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

This prints:

```
80, 81;
82, 83;
84, 85;
86, 87;
```

Warning: Since C++ performs garbage collection you should use `cppmat::view` with care. You are responsible that pointers do not go out of scope.

Tip: One can also use the `Map` constructor instead of the `setMap` method:

```
using Mat = cppmat::view::matrix<double,4,2>;

Mat view = Mat::Map(&container(10));
```

Note: This function cannot make any modification to the view. Its usage is thus somewhat limited. To get a wider functionality use *cppmat::tiny::array*. For example:

```
#include <cppmat/cppmat.h>

int main()
{
    cppmat::array<double> container = cppmat::array<double>::Arange({100,4,2});

    cppmat::tiny::array<double,2,4,2> copy;

    copy.setCopy(container.item(10), container.item(10)+copy.size());

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

Note that `copy` is now a copy. I.e. any modification to `copy` will not result in a modification in `container`.

Note that the following syntax could also have been used:

```
using Mat = cppmat::tiny::matrix<double,4,2>;

Mat copy = Mat::Copy(container.item(10), container.item(10)+8);
```

Or the following:

```
using Mat = cppmat::tiny::matrix<double,4,2>;

Mat copy = Mat::Copy(container.item(10));
```

Or the following:

```
std::copy(container.item(10), container.item(10)+copy.size(), copy.data());
```

3.6.2 cppmat::view::matrix

```
#include <cppmat/cppmat.h>

int main()
{
    cppmat::view::matrix<double,10,10> A;
```

(continues on next page)

(continued from previous page)

```
    A.setMap(...)

    ... = A(0,0)

    ...

    return 0;
}
```

Most methods are the same as for *cppmat::tiny::matrix*.

3.6.3 cppmat::view::vector

[map_regular_vector.h, map_regular_vector.hpp]

```
#include <cppmat/cppmat.h>

int main()
{
    cppmat::view::vector<double,10> A;

    A.setMap(...)

    ... = A(0)

    ...

    return 0;
}
```

Most methods are the same as for *cppmat::tiny::vector*.

3.6.4 cppmat::view::symmetric::matrix

[map_symmetric_matrix.h, map_symmetric_matrix.hpp]

Class to view a pointer to a fixed size, symmetric, matrices. For example:

```
#include <cppmat/cppmat.h>

int main()
{
    cppmat::view::symmetric::matrix<double,10,10> A;

    A.setMap(...)

    ... = A(0,0)

    ...

    return 0;
}
```

Most methods are the same as for *cppmat::tiny::symmetric::matrix*.

3.6.5 cppmat::view::diagonal::matrix

[map_diagonal_matrix.h, map_diagonal_matrix.hpp]

Class to view a pointer to a fixed size, symmetric, matrices. For example:

```
#include <cppmat/cppmat.h>

int main()
{
    cppmat::view::diagonal::matrix<double,10,10> A;

    A.setMap(...)

    ... = A(0,0)

    ...

    return 0;
}
```

Most methods are the same as for *cppmat::tiny::diagonal::matrix*.

3.6.6 cppmat::view::cartesian

cppmat::view::cartesian::tensor4

[map_cartesian_tensor4.h, map_cartesian_tensor4.hpp]

Class to view a pointer to a fixed size, fourth order tensors. For a 3-d tensor

```
#include <cppmat/cppmat.h>

int main()
{
    cppmat::view::cartesian::tensor4<double,3> A;

    A.setMap(...)

    ... = A(0,0,0,0)

    ...

    return 0;
}
```

Most methods are the same as for *cppmat::tiny::cartesian::tensor4*.

cppmat::view::cartesian::tensor2

[map_cartesian_tensor2.h, map_cartesian_tensor2.hpp]

Class to view a pointer to a fixed size, second order tensors. For a 3-d tensor

```
#include <cppmat/cppmat.h>

int main()
{
    cppmat::view::cartesian::tensor2<double,3> A;

    A.setMap(...)

    ... = A(0,0)

    ...

    return 0;
}
```

Most methods are the same as for *cppmat::tiny::cartesian::tensor2*.

cppmat::view::cartesian::tensor2s

[map_cartesian_tensor2s.h, map_cartesian_tensor2s.hpp]

Class to view a pointer to a fixed size, symmetric, second order tensors. For a 3-d tensor

```
#include <cppmat/cppmat.h>

int main()
{
    cppmat::view::cartesian::tensor2s<double,3> A;

    A.setMap(...)

    ... = A(0,0)

    ...

    return 0;
}
```

Most methods are the same as for *cppmat::tiny::cartesian::tensor2s*.

cppmat::view::cartesian::tensor2d

[map_cartesian_tensor2d.h, map_cartesian_tensor2d.hpp]

Class to view a pointer to a fixed size, diagonal, second order tensors. For a 3-d tensor

```
#include <cppmat/cppmat.h>

int main()
{
    cppmat::view::cartesian::tensor2d<double,3> A;

    A.setMap(...)

    ... = A(0,0)
```

(continues on next page)

(continued from previous page)

```
    ...

    return 0;
}
```

Most methods are the same as for *cppmat::tiny::cartesian::tensor2d*.

cppmat::view::cartesian::vector

[map_cartesian_vector.h, map_cartesian_vector.hpp]

Class to view a pointer to a fixed size, vector. For a 3-d vector

```
#include <cppmat/cppmat.h>

int main()
{
    cppmat::view::cartesian::vector<double,3> A;

    A.setMap(...)

    ... = A(0,0)

    ...

    return 0;
}
```

Most methods are the same as for *cppmat::tiny::cartesian::vector*.

3.7 Access to storage

The storage of all the classes can be accessed through the `data()` method, which is complemented with the iterators `begin()` and `end()`. Consider the following examples

3.7.1 Creating a cppmat-object

Copy constructor

```
#include <cppmat/cppmat.h>

using T2 = cppmat::cartesian3d::tensor2<double>;

int main()
{
  std::vector<double> data(3*3);

  for ( size_t i = 0 ; i < 3*3 ; ++i )
    data[i] = static_cast<double>(i);

  T2 A = T2::Copy(data.begin(), data.end());
```

(continues on next page)

(continued from previous page)

```
  std::cout << "A = " << A << std::endl;
}
```

std::copy

```
#include <cppmat/cppmat.h>

using T2 = cppmat::cartesian3d::tensor2<double>;

int main()
{
  std::vector<double> data(3*3);

  for ( size_t i = 0 ; i < 3*3 ; ++i )
    data[i] = static_cast<double>(i);

  T2 A;

  std::copy(data.begin(), data.end(), A.begin());

  std::cout << "A = " << A << std::endl;
}
```

3.8 STL(-like) extensions

3.8.1 Out-of-place functions

- **min/max**

  ```
  cppmat::array<double> C = cppmat::min(A, B);
  ```

3.8.2 std::vector

- **Formatted print**

  If cppmat is loaded one can also view STL-vectors as easily as

  ```
  std::cout << A << std::endl;
  ```

- **Delete item**

  ```
  A = cppmat::del(A, -1);
  ```

- **argsort**

  ```
  std::vector<size_t> idx = cppmat::argsort(A);
  ```

- **linspace**

  ```
  std::vector<double> A = cppmat::linspace(0.0, 1.0, 11);
  ```

- **min/max**

  ```
  std::vector<double> C = cppmat::min(A, B);
  ```

3.9 Histogram

3.9.1 histogram

```
template<typename X>
std::tuple<std::vector<double>, std::vector<double>> histogram(
  const std::vector<X> &data, size_t bins=10, bool density=false, bool return_
↪edges=false
)
```

Create a histogram. Returns `std::tie(P, x)`: the count and the locations on the bins (their midpoints, or their edges if `return_edges=true`).

3.9.2 histogram_uniform

```
template<typename X>
std::tuple<std::vector<double>, std::vector<double>> histogram_uniform(
  const std::vector<X> &data, size_t bins=10, bool density=false, bool return_
↪edges=false
)
```

Create a histogram such that each bins contains the same number of entries. Returns `std::tie(P, x)`: the count and the locations on the bins (their midpoints, or their edges if `return_edges=true`).

3.10 Compiling

3.10.1 Introduction

This module is header only. So one just has to `#include <cppmat/cppmat.h>` somewhere in the source code, and to tell the compiler where the header files are. For the latter, several ways are described below.

Before proceeding, a word about optimization. Of course one should use optimization when compiling the release of the code (`-O2` or `-O3`). But it is also a good idea to switch off the assertions in the code (mostly checks on size) that facilitate easy debugging, but do cost time. Therefore, include the flag `-DNDEBUG`. Note that this is all C++ standard. I.e. it should be no surprise, and it is always a good idea to do.

3.10.2 Manual compiler flags

GNU / Clang

Add the following compiler's arguments:

```
-I${PATH_TO_CPPMAT}/src -std=c++14
```

Note: (Not recommended)

If you want to avoid separately including the header files using a compiler flag, `git submodule` is a nice way to go:

1. Include the submodule using `git submodule add https://github.com/tdegeus/cppmat.git`.
2. Include using `#include "cppmat/src/cppmat/cppmat.h"`.

   If you decide to manually copy the header file, you might need to modify this relative path to your liking.

Or see *(Semi-)Automatic compiler flags*. You can also combine the `git submodule` with any of the below compiling strategies.

3.10.3 (Semi-)Automatic compiler flags

Install

To enable (semi-)automatic build, one should 'install' `cppmat` somewhere.

Install systemwide (depends on your privileges)

1. Proceed to a (temporary) build directory. For example

```
$ cd /path/to/temp/build
```

2. 'Install' `cppmat`:

```
$ cmake /path/to/cppmat
$ make install
```

Note: One usually does not need any compiler arguments after following this protocol.

Install in custom location (user)

1. Proceed to a (temporary) build directory. For example

```
$ cd /path/to/temp/build
```

2. 'Install' `cppmat`, to install it in a custom location

```
$ mkdir /custom/install/path
$ cmake /path/to/cppmat -DCMAKE_INSTALL_PREFIX:PATH=/custom/install/path
$ make install
```

3. Add the following path to your `~/.bashrc` (or `~/.zshrc`):

```
export PKG_CONFIG_PATH=/custom/install/path/share/pkgconfig:$PKG_CONFIG_PATH
export CPLUS_INCLUDE_PATH=$HOME/custom/install/path/include:$CPLUS_INCLUDE_PATH
```

Note: One usually does not need any compiler arguments after following this protocol.

Note: **(Not recommended)**

If you do not wish to use `CMake` for the installation, or you want to do something custom. You can, of course. Follow these steps:

1. Copy the file `src/cppmat.pc.in` to `cppmat.pc` to some location that can be found by `pkg_config` (for example by adding `export PKG_CONFIG_PATH=/path/to/cppmat.pc:$PKG_CONFIG_PATH` to the `.bashrc`).
2. Modify the line `prefix=@CMAKE_INSTALL_PREFIX@` to `prefix=/path/to/cppmat`.
3. Modify the line `Cflags: -I${prefix}/@CPPMAT_INCLUDE_DIR@` to `Cflags: -I${prefix}/src`.
4. Modify the line `Version: @CPPMAT_VERSION_NUMBER@` to reflect the correct release version.

Compiler arguments from 'pkg-config'

Should the compiler for some reason not be able to find the headers, instead of `-I...` one can now use

```
`pkg-config --cflags cppmat` -std=c++14
```

as compiler argument.

Compiler arguments from 'cmake'

Add the following to your `CMakeLists.txt`:

```
set(CMAKE_CXX_STANDARD 14)

find_package(PkgConfig)

pkg_check_modules(CPPMAT REQUIRED cppmat)
include_directories(${CPPMAT_INCLUDE_DIRS})
```

Note: Except the C++ standard it should usually not be necessary to load cppmat explicitly, as it is installed in a location where the compiler can find it.

3.10.4 Compiling Python modules that use cppmat

To compile Python modules that use cppmat using for example

```
python setup.py build
python setup.py install
```

One can 'install' cppmat's headers to the include directory that Python uses. One can obtain cppmat from PyPi:

```
pip install cppmat
```

Or install from a local copy:

```
pip install /path/to/cppmat
```

3.11 Python interface

This library provides an interface to pybind11 such that an interface to NumPy arrays is automatically provided when including a function with any of the cppmat classes:

cppmat class	Rank of NumPy-array
`cppmat::array`	n
`cppmat::matrix`	2
`cppmat::vector`	1
`cppmat::tiny::array`	2
`cppmat::tiny::matrix`	2
`cppmat::tiny::vector`	1
`cppmat::cartesian::tensor4`	4
`cppmat::cartesian::tensor2`	2
`cppmat::cartesian::tensor2s`	2
`cppmat::cartesian::tensor2d`	2
`cppmat::cartesian::vector`	1
`cppmat::tiny::cartesian::tensor4`	4
`cppmat::tiny::cartesian::tensor2`	2
`cppmat::tiny::cartesian::tensor2s`	2
`cppmat::tiny::cartesian::tensor2d`	2
`cppmat::tiny::cartesian::vector`	1

Warning: On the Python side all the matrices (`cppmat::matrix`, `cppmat::symmetric::matrix`, and `cppmat::diagonal::matrix`) and 2nd-order tensors (`cppmat::cartesian::tensor2`, `cppmat::cartesian::tensor2s`, and `cppmat::cartesian::tensor2d`) are all square matrices (rank 2 NumPy arrays). This means that when a function that has `cppmat::symmetric::matrix` or `cppmat::cartesian::tensor2s` as argument, the upper-diagonal part is read; while when it has an argument `cppmat::diagonal::matrix` or `cppmat::cartesian::tensor2d` only the diagonal is considered.

This requires extra attention as information might be lost. To optimize for speed and flexibility no checks are performed in the release libraries derived from cppmat!

You can ask cppmat to check for this, by omitting the `-DNDEBUG` compiler flag (this enables several assertions, so it may cost you some efficiency).

(The same holds for the classes under `cppmat::tiny::`.)

To use this feature one has to:

```
#include <cppmat/pybind11.h>
```

3.11.1 Building

[`tensorlib.zip`]

Building is demonstrated based on the 'tensorlib' example.

CMake

[CMakeLists.txt]

```
cmake_minimum_required(VERSION 2.8.12)
project(tensorlib)

# set optimization level
# - aggressive optimization
#   compiler: ... -O3
set(CMAKE_BUILD_TYPE Release)
# - switch off assertions (more risky but faster)
#   compiler: ... -DNDEBUG
add_definitions(-DNDEBUG)

# set C++ standard
# - compiler: ... -std=c++14
set(CMAKE_CXX_STANDARD 14)

# find cppmat
# - compiler: ... -I${CPPMAT_INCLUDE_DIRS}
find_package(PkgConfig)
pkg_check_modules(CPPMAT REQUIRED cppmat)
include_directories(${CPPMAT_INCLUDE_DIRS})

# find pybind11
find_package(pybind11 REQUIRED)
pybind11_add_module(tensorlib tensorlib.cpp)
```

Build using

```
cd /path/to/tempdir
cmake /path/to/tensorlib
make
```

For this to work, *pybind11* must be 'installed' on the system.

Tip: Alternatively you can include *pybind11* as a sub-folder (for example using `git submodule add https://github.com/pybind/pybind11.git`). In that case, replace `find_package(pybind11 REQUIRED)` by `add_subdirectory(pybind11)` in `CMakeLists.txt`.

Tip: To link to external libraries, include at the end of your `CMakeLists.txt`

```
target_link_libraries(${PROJECT_NAME} PUBLIC ${PROJECT_LIBS})
```

---

setup.py

[setup.py]

```
from setuptools import setup, Extension

import sys
```

(continues on next page)

(continued from previous page)

```
import setuptools
import pybind11
import cppmat

__version__ = '0.0.1'

ext_modules = [
  Extension(
    'tensorlib',
    ['tensorlib.cpp'],
    include_dirs=[
      pybind11.get_include(False),
      pybind11.get_include(True ),
      cppmat  .get_include(False),
      cppmat  .get_include(True )
    ],
    language='c++'
  ),
]

setup(
  name                = 'tensorlib',
  description         = 'Tensorlib',
  long_description    = 'This is an example module, it no real use!',
  keywords            = 'Example, C++, C++11, Python bindings, pybind11',
  version             = __version__,
  license             = 'MIT',
  author              = 'Tom de Geus',
  author_email        = 'tom@geus.me',
  url                 = 'https://github.com/tdegeus/cppmat/docs/examples/tensorlib',
  ext_modules         = ext_modules,
  extra_compile_args = ["-DNDEBUG"], # switch off assertions
  install_requires    = ['pybind11>=2.1.0','cppmat>=0.2.1'],
  cmdclass            = {'build_ext': cppmat.BuildExt},
  zip_safe            = False,
)
```

As shown in this example building using the `setup.py` can be simplified using some routines in the `cppmat` python module. These routines have been taken from pybind, most notably from Sylvain Corlay and Dean Moldovan. They are merely attached to this module to simplify your life.

Build using

```
python3 setup.py build
python3 setup.py install
```

Tip: Replace the executable with your favorite Python version, e.g. with `python`.

CMake & setup.py

CMake can be called from the `setup.py` to take advantage of both. In that case the `setup.py` would be simply

```
import setuptools, cppmat

setuptools.setup(
  name             = 'tensorlib',
  version          = '0.0.1',
  author           = 'Tom de Geus',
  author_email     = 'email@address.com',
  description      = 'Description',
  long_description = '',
  ext_modules      = [cppmat.CMakeExtension('tensorlib')],
  cmdclass         = dict(build_ext=cppmat.CMakeBuild),
  zip_safe         = False,
)
```

3.12 Notes for developers

3.12.1 Structure

Note:

3.12.2 Make changes / additions

Be sure to run the verification code in `develop/`! All existing checks should pass, while new check should be added to check new functionality.

The Catch library has been used to run the checks, it should be used also for all new checks. To compile the checks:

1. Download and install Catch.
2. Compile all test cases:

```
$ mkdir develop/build
$ cd develop/build
$ cmake ..
$ make
```

3. Run `./cppmatTest`.

3.12.3 Python

The Python package of this module `python/cppmat/__init__.py` is essentially used to allow distribution of the header files that constitute this library through PyPi. In addition a small Python package *cppmat* is provided that allows easy `setup.py` formulations of derived packages. These features can also be used when one is just interested in using *pybind11* and one does not intend to use *cppmat* itself.

3.12.4 Create a new release

1. Update the version number in `src/cppmat/macros.h`.
2. Upload the changes to GitHub and create a new release there (with the correct version number).

3. Upload the package to PyPi:

```
$ python3 setup.py bdist_wheel --universal
$ twine upload dist/*
```