COMPUTING OBJECTS

CM2 Math1

Version 5.6

overview

Revision February 2025.

https://www.computing-objects.com

Forewords

CM2 Math1 is the core mathematical library used by the mesh generators of **CM2 MeshTools**® SDK and the FEA **CM2 FEM**® SDK.

For maximum performance all the CM2 software components are developed using standard C++ with efficient object-oriented programming techniques (templates, very few virtual classes).

They have been ported to most major platforms (Windows®, Linux, macOS®).

With a binary license these libraries are shipped as C++ headers files together with dynamic libraries – Win32/ Win64 `.dll/.lib`, shared Linux x86/x86-64 `.so` or macOS `.dylib`.

Source codes are also available.

Table of contents

Introduction

We intend here to give only a short description of the classes in the **CM2 Math1** library and its associated template libraries. We'll simply explain the basic traits of these math containers and show a few usages of the template functions. Please read the CM2 Math1 HTML reference manual (Doxygen© generated)[1] for full description.

To use **CM2 Math1** you must include the header `math1.h` and link against the `cm2math1` lib (for example `cm2math1_x64_56.lib` for version 5.6, Windows x64).

Classes of **CM2 Math1** are defined in the `cm2` namespace. The template math functions are nested in `cm2::vecscal`, `cm2::vecvec`, `cm2::matscal`, `cm2::matvec` and `cm2::matmat` (see Appendix for lists of functions).

For sake of simplification we'll assume in the rest of this document a `using namespace cm2` directive to allow us omitting the `cm2::` namespace before any of the **CM2 Math1** class and any nested template math library.

The **CM2 Math1** library exports 21 types of vector, 17 types of rectangular matrix and 14 types of symmetric matrix[2].

They can be divided into two main categories:

- The variable-size containers such as `DoubleVec`, `DoubleMat`, `DoubleSym`, `DoubleSparse.`
- The fixed-size containers such as `DoubleVec3`, `DoubleMat2x2`, `DoubleSym2`.

☞ Following the C usage, all these math containers are zero based: a vector of size N extends from index 0 to $N-1$.

[1] See math1.html or math1.chm.

[2] Based on the template classes: `cm2::vector_fixed<T, N>`, `cm2::matrix_fixed<T, M, N>`, `cm2::dense1D<T>`, `cm2::dense2D<T>`, `cm2::symmetric_full<T>`, `cm2::symmetric_sparse<T>`.

Variable-size containers and fixed-size containers

The containers of variable-sized category can be resized, automatically or manually. They have also *shallow* copy constructors and copy operators. A variable-size container holds a reference to a data array (mere view) and a copy implies only a copy of that reference not the data (shallow copy).

On the other hand, a fixed-size container actually holds the data as a member array and a copy actually copies the data (deep copy).

Example

```
DoubleVec   V1;                 // Empty vector.
DoubleVec   V2(10, +1.);        // Vector of 10 values, all initialized to 1.

V1.push_back(1.)                // V1.size() = 1
V1.push_back(2.)                // V1.size() = 2
V1.push_back(3., 4., 5.);       // V1.size() = 5
V1.push_back_n(6., 10);         // V1.size() = 15

V1 = V2;                        // Shallow copy[3] (V1.size() = 10).
                                // Previous values of V1 are lost.

V1[0] = 0.;                     // V2[0] = 0. (because the data are shared).
V2.clear();                     // V2.size() = 0 but V1.size() = 10
                                // The data are not deleted because still viewed
                                // from V1.
```

[3] A deep copy can be obtained with the template function `vecvec::copy` :

```
V1.resize(V2.size());      // Resize V1 to V2.size().
vecvec::copy(V2, V1);      // Copy all V2 values into V1.
```

or with the "copy" member:

```
V1.copy(V2);               // Resize V1 to V2.size() and copies the data.
```

Vectors	Rectangular matrices	Symmetric matrices	Symmetric sparse matrices
DoubleVec	DoubleMat	DoubleSym	DoubleSymSparse
FloatVec	FloatMat	FloatSym	FloatSymSparse
IntVec	IntMat	IntSym	IntSymSparse
UIntVec	UIntMat	UIntSym	UIntSymSparse
DoubleZVec	DoubleZMat	DoubleZSym	DoubleZSymSparse
DoubleVec2			
DoubleVec3	DoubleMat3x1	DoubleSym2	
DoubleVec4	DoubleMat2x2		
DoubleVec5			
DoubleVec6	DoubleMat3x2		
DoubleMat2x3	DoubleSym3		
DoubleVec7			
DoubleVec8			
DoubleVec9	DoubleMat3x3		
	DoubleMat3x4		
UIntVec2			
UIntVec3	UIntMat3x1	UIntSym2	
UIntVec4	UIntMat2x2		
UIntVec5			
UIntVec6	UIntMat3x2		
UIntMat2x3	UIntSym3		
UIntVec7			
UIntVec8			
UIntVec9	UIntMat3x3		
	UIntMat3x4		

Table 1 – The **CM2 Math1** vector and matrix types
(first 5 lines are variable-size containers, the other lines are fixed-size containers).

Views of the variable-size containers

Several variable-sized containers can have view on the same array of data but the views can be different from each other. The beginning and the size in the array are specific to each container.
For instance, in an array of 30 items, a first vector views items from 0 to 9 and a second one views items from 5 to 20.

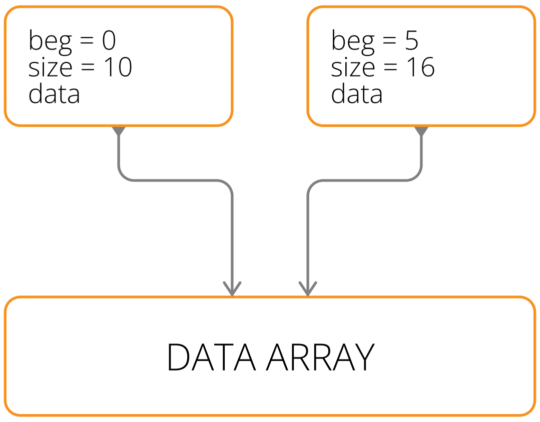

Figure 1 – Views and data in variable-sized vectors

Elements from 5 to 9 are accessible from the two vectors.

When a destructor is called on a variable-size container the data is destroyed only when no other container references this data anymore. A smart pointer mechanism is used to count the number of references on the data and the deallocation actually occurs when the count reaches zero. The memory management is automatic (automatic garbage collection).

Example

```
DoubleVec  *V1(new DoubleVec(10, -1.));
DoubleVec  *V2(new DoubleVec(*V1));     // Shallow copy (share the data).

delete V1;   // Data is still referenced by V2.
delete V2;   // Now data is destroyed too.
```

The variable-size matrix types (`IntMat`, `UIntMat`, `DoubleMat`, `DoubleZMat`...) have similar behavior with respect to the memory management and the copy operators (and copy-constructors). Data is shared, copies are shallow.

Fixed-size containers

The fixed-size math containers are deep-copy containers.

The copy-constructor and the copy-operator do not share the data anymore but leads to actually different arrays in memory. They are simpler than the variable-size containers and faster for short arrays whereas the variable-size containers are more suited for big arrays.

The fixed-size vectors are similar to the STL `std::array` class.

Example

```
DoubleVec2   V1;            // Vector of 2 uninitialized values (double).
DoubleVec2   V2(1.);        // Vector of 2 values initialized to 1.

V1[0] = 0.;
V1[1] = -1.;

V1 = V2;                    // Deep copy: V1 = {1, 1}, V2 = {1, 1}.
V2[0] = 0.;                 // V1 = {1, 1}, V2 = {0, 1}
V1[1] = 0.;                 // V1 = {1, 0}, V2 = {0, 1}
```

The fixed-size matrix types (`DoubleMat3x3`, `UIntMat2x3`...) have similar behavior with respect to the memory management and the copy operators (and copy-constructors). Data are not shared, copies are deep.

STL-like iterators and the template math library

The vector containers - variable-size and fixed-size - are equipped with STL-like iterators `begin()` and `end()` to make them compatible with most of the C++ Standard Template Library (STL) algorithms.

They also have access operators such as `operator[]` and the usual functions for a vector class: `size()`, `empty()`, `front()`, `back()`...

The variable-size vectors are also equipped with members such as `reserve`, `resize`, `push_back` and `pop_back`.

Aside from the STL algorithms, one can also use the CM2 Math1 template functions in `cm2::vecscal`, `cm2::vecvec`, `cm2::matscal`, `cm2::matvec` and `cm2::matmat` (cf. CM2 Math1 HTML reference manual).

Example

```
DoubleVec    V1(3), V2(3,-1.);
DoubleMat    M(3, 10, 0.);          // Matrix of doubles 3 by 10 set to 0.

vecvec::copy(V2, V1);               // Hard copy of V2 into V1 (sizes match).
vecvec::axpy(2., V1, V2);           // V2 += 2 * V1
vecscal::mult(-1., V1);             // V1 = -V1
```

In matrix classes (variable-sized or fixed-sized), the rows and the columns are equipped with iterators just like the vectors and the same template functions can be used on them.

Example

```
const size_t   N(10);
const double   PI(3.14159);
DoubleMat      pos(2, N);           // Uninitialized 2xN matrix.
DoubleVec2     V;                   // Uninitialized vector of size 2.

// Points on a circle.
for (size_t j = 0; j < N; ++j)
{
   pos(0, j) = ::cos(j*2*PI/N);
   pos(1, j) = ::sin(j*2*PI/N);
}

// Set radius to 3.
matscal::mult(3., pos);

// Copy segment/column #2 to V (ok dimensions match).
vecvec::copy(pos.seg(2), V);

// Copy V to segment/column #9 (the last one).
vecvec::copy(V, pos.seg(9));

// Copy segment/column #9 to segment/column #0.
vecvec::copy(pos.seg(9), pos.seg(0));

// Append V in a new column of pos (dimensions match).
pos.push_back(V);                  // pos.cols() = 11 after this line.
```

Bound checking

In debug mode (with macro NDEBUG not defined) bound violations abort the program. In release mode however, for best performance, no check is performed and the user must take care not to out-value the limits of the vectors and matrices.

Interoperability with other math containers

The API of the **CM2 MeshTools** library use exclusively vectors and matrices of the **CM2 Math1** library (such as DoubleMat, UIntMat, FloatVec...)

To use the meshers with other types of vectors and matrices, the variable-size containers are equipped with constructors with raw pointers as arguments. Hence they can view the data in any other math containers as long as the latter provide a way to get a raw pointer to their data and that these data are contiguous in memory.

Remember that the variable-size containers implement shallow copies. This means that the arrays are *shared* not copied. Therefore the memory management becomes a point to take care of and the user must keep in mind which library is responsible for deleting the memory upon exit. The default is that the allocator of the array remains responsible for its deletion.

Example

```
double                   buff1[100];              // Static-allocated C array.
double*                  buff2(new double[100]);  // Dynamic-allocated C array.
std::vector<double>      buff3(100);              // STL vector.
std::array<double, 100>  buff4;                   // STL array.

DoubleVec   V1(50, buff1);          // Views the first 50 elements in buff1.
DoubleVec   V2(50, buff2);          // Views the first 50 elements in buff2.
DoubleVec   V3(50, buff3.begin()); // Views the first 50 elements in buff3.
DoubleVec   V4(50, buff4.begin()); // Views the first 50 elements in buff4.

V2.clear();           // buff2 is not deallocated.
V2 = V1;              // buff2 is not deallocated.
V3.clear();           // buff3 is not deallocated.
V4.clear();           // buff4 is not deallocated.
delete[] buff2;       // Dangerous: V2 is still alive (though emptied).
                      // Don’t use V2 anymore or use V2.clear_hard().
buff3.clear();        // Dangerous: data in buff3 may has been deallocated.
                      // Don't use V3 anymore or use V3.clear_hard().
```

This can be done also with the fixed-size containers of **CM2 Math1**:

```
DoubleVec3  V3(1., 0., -1.);
DoubleVec   V(3, V1.begin());     // Views the elements in V3.
```

Similarly, the matrices of **CM2 Math1** can view the data in an external container. Together with the raw pointer to the data, the user must provide the number of rows, the number of columns and the stride between two columns (so-called leading dimension):

```
unsigned*   buff(new unsigned[30]);
UIntMat     M(3, 10, /*ld=>*/ 3, buff);
```

As before, the matrix is not responsible for the deletion of the underlying buffer[4].

In the case where a container constructed this way is subsequently resized, it may "point" to another array of memory but the initial buffer remains valid:

```
double*     buff(new double[6]);
DoubleVec   V(5, buff);

V.push_back(2.0);    // Reallocation and copy performed.
                     // buff is still alive, but V does not "point" to
                     // it anymore.
V.clear_hard();      // The new array of V is deallocated, not buff.
```

☞ As a rule of thumb, the lifetime of the external buffer must span the lifetime of the math1 container.

```
double*     buff(new double[6]);

{
   DoubleVec    V(5, buff);
   ...    // Use V here.
}  // V is killed here but the buffer is spared.

delete[] buff;  // So long with buff.
```

[4] A default parameter `protect` in the constructors can be used to change this behavior.

We have seen how to construct **CM2 Math1** variable-size container upon other containers or buffers.
To do the other way, we use the `data()` or `begin()` members to access the underlying data:

Example

```
DoubleVec      V(50, 0.);
double*        buff(V.data());
size_t         N(V.size());         // Equals to 50

for (size_t i = 0; i < N; ++i, ++buff)
   *buff = double(i);

assert (V[10] == 10.);              // Changes in buff have been seen in V.
```

```
DoubleMat      P(3, 40);
double*        buff(P.data());
size_t         M(P.rows());         // Equals to 3
size_t         N(P.cols());         // Equals to 40
size_t         LD(P.ld());          // Equals to 3 (here stride = rows).

for (size_t j = 0; j < N; ++j)
   for (size_t i = 0; i < M; ++i)
      buff[i + j*LD] = double(i + j*LD);

assert(P(0,10) == 30.);             // Changes in buff have been seen in P.
```

Here the **CM2 Math1** vectors and matrices are responsible for the deletion of its data:

```
DoubleMat      P(3, 40);
double*        buff(P.data());

delete[] buff;       // Don't do that!
P.resize(3, 80);     // Crash now or maybe later...
```

Appendix - template math functions list

Vector - Scalar functions (`cm2::vecscal`)

```
// Inserts a value inside a vector.
template <class Vec>
void insert (typename Vec::value_type v, size_t i, Vec &X);

// Removes a value inside a vector.
template <class Vec>
void remove (size_t i, Vec &X);

// Copies a value into a vector.
template <class Vec>
void copy (typename Vec::value_type v, Vec &X);

// Adds a value to a vector.
template <class Vec>
void add (typename Vec::value_type v, Vec &X);

// Subtracts a value  from a vector.
template <class Vec>
void subtract (typename Vec::value_type v, Vec &X);

// Multiplies a vector by a scalar.
template <class Vec>
void mult (typename Vec::value_type v, Vec &X);

// Divides a vector by a scalar.
template <class Vec>
void div (typename Vec::value_type v, Vec &X);

// Negates a vector.
template <class Vec>
void negate (Vec &X);

// Sets a vector to its reciprocal (element by element).
template <class Vec>
void reciprocal (Vec &X);

// The index of the max in a vector.
template <class Vec>
size_t max_index (const Vec &x);

// The index of the min in a vector.
template <class Vec>
size_t min_index (const Vec &x);

// The index of the max magnitude (max(|x[i]|) in a vector.
template <class Vec>
size_t max_norm_index (const Vec &x);

// The index of the min magnitude (min(|x[i]|) in a vector.
template <class Vec>
size_t min_norm_index (const Vec &x);

// The maximum value in a vector.
template <class Vec>
typename Vec::value_type max_value (const Vec &x);

// The minimum value in a vector.
template <class Vec>
typename Vec::value_type min_value (const Vec &x);

// Increases range R with the range of a vector.
template <class Vec, class Range>
void inc_range (const Vec &X, Range &R);

// The range of values in a vector (first = min, second = max)
template <class Vec>
std::pair<typename Vec::value_type, typename Vec::value_type> range (const Vec &X);

// The sum of the elements in a vector.
template <class Vec>
typename Vec::value_type sum (const Vec &X);

// The product of the elements in a vector.
template <class Vec>
typename Vec::value_type prod (const Vec &X);
```

```
// The one-norm of a vector (the sum of absolute elements).
template <class Vec>
typename Vec::value_type one_norm (const Vec &X);

// The two-norm of a vector (usual Euclidean norm).
template <class Vec>
typename Vec::value_type two_norm (const Vec &X);

// The max-norm (infinite norm).
template <class Vec>
typename Vec::value_type max_norm (const Vec &X);

// The square of the two-norm of a vector.
template <class Vec>
typename Vec::value_type sqr_two_norm (const Vec &X);

// Normalizes a vector (if not null).
template <class Vec>
typename Vec::value_type normalize (Vec &X);

// The arithmetic mean of a vector.
template <class Vec>
typename Vec::value_type mean (const Vec &X);

// Statistics on a vector (mean and variance).
template <class Vec, class T>
void statistics (const Vec &X, T &mean, T &variance);

// The number of elements in a vector that match a predicate.
template <class Vec , class Predicate>
size_t count_if (const Vec &X, Predicate pred);

// Tests all elements in a vector against a scalar.
template <class Vec>
bool equal (const Vec &X, typename Vec::value_type v);

// Has the vector only finite values (float and double only).
template <class Vec>
bool isfinite (const Vec &X);

// Adds random values between 0 and v to all elements of a vector.
template <class Vec>
void randomize (Vec &X, typename Vec::value_type v);

// Adds random values between v1 and v2 to all elements of a vector.
template <class Vec>
void randomize_range (Vec &X, typename Vec::value_type v1, typename Vec::value_type v2);

// Multiplies the elements of a vector by random values between 1-v and 1+v.
template <class Vec>
void randomize_mult (Vec &X, typename Vec::value_type v);

// Adds random values between 0 and v to all elements of a vector.
template <class Vec, class RandGen>
void randomize (Vec &X, typename Vec::value_type v, RandGen &randg);

// Adds random values between v1 and v2 to all elements of a vector.
template <class Vec, class RandGen>
void randomize_range (Vec &X, typename Vec::value_type v1, typename Vec::value_type v2, RandGen
&randg)

// Multiplies the elements of a vector by random values between 1-v and 1+v.
template <class Vec, class RandGen>
void randomize_mult (Vec &X, typename Vec::value_type v, RandGen &randg);
```

Table 2 – The `cm2::vecscal` template math library.

Vector - Vector functions (cm2::vecvec)

```
// Inserts some values at specific positions inside a vector.
template <class Vec1, class VecI, class Vec>
void insert (const Vec1& V1, const VecI& indices, Vec &X);

// Removes some values at specific positions inside a vector.
template <class VecI, class Vec>
void remove (const VecI& indices, Vec &X);

// Appends a vector to another vector.
template <class Vec1 , class Vec2>
bool push_back (const Vec1 &V1, Vec2 &V2);

// Appends a specific batch of elements to a vector.
template <class Vec1, class Vec2, class Vector>
bool push_back (const Vec1 &V1, const Vector &V1_indices, Vec2 &V2);

// Copies a vector to another.
template <class VecX, class VecY>
void copy (const VecX &X, VecY &Y);

// Swaps the values between two vectors.
template <class VecX, class VecY>
void swap (VecX &X, VecY &Y)

// Copies the scaling of a vector to another.
template <class T, class VecX, class VecY>
void copy_scale (T a, const VecX &X, VecY &Y);

// Adds a vector to another.
template <class VecX, class VecY>
void add (const VecX &X, VecY &Y);

// Subtracts a vector from another.
template <class VecX, class VecY>
void subtract (const VecX &X, VecY &Y);

// Multiplies a vector by another (element by element).
template <class VecX, class VecY>
void mult (const VecX &X, VecY &Y);

// Divides a vector by another (element by element).
template <class VecX, class VecY>
void div (const VecX &X, VecY &Y);

// The reciprocal vector.
template <class Vec>
typename Vec::value_type reciprocal (const Vec &x, Vec &inv_x);

// Multiplies two vectors and adds into a third (element by element).
template <class VecX, class VecY, class VecZ>
void mult (const VecX &X, const VecY &Y, VecZ &Z);

// Dot product between two vectors.
template <class VecX, class VecY>
cm2::biggest2_type<typename VecX::value_type, typename VecY::value_type>::type>
dot (const VecX &X, const VecY &Y);

// AXPY between two vectors.
template <class T, class VecX, class VecY>
void axpy (T a, const VecX &X, VecY &Y)

// Element-wise multiplication between three vectors.
template <class T, class VecX, class VecY, class VecZ>
void axypz (T a, const VecX &X, const VecY &Y, VecZ &Z);

// Interpolation between two vectors.
template <class T, class VecX, class VecY, class VecZ>
void interpol (T a, const VecX &X, const VecY &Y, VecZ &Z);

// Copies the sum of two vectors (element by element).
template <class VecX, class VecY, class VecZ>
void copy_add (const VecX &X, const VecY &Y, VecZ &Z);

// Copies the sum with scaling of two vectors (element by element).
template <class T, class VecX, class VecY, class VecZ>
void copy_add (T a, const VecX &X, const VecY &Y, VecZ &Z);

// Copies the difference between two vectors (element by element).
template <class VecX, class VecY, class VecZ>
void copy_diff (const VecX &X, const VecY &Y, VecZ &Z);
```

```
// Copies the difference with scaling between two vectors (element by element).
template <class T, class VecX, class VecY, class VecZ>
void copy_diff (T a, const VecX &X, const VecY &Y, VecZ &Z);

// The square of the two norm of the difference between two vectors.
template <class VecX, class VecY>
cm2::biggest2_type<typename VecX::value_type, typename VecY::value_type>::type>
diff_sqr_two_norm (const VecX &X, const VecY &Y);

// The two norm of the difference between two vectors.
template <class VecX, class VecY>
cm2::biggest2_type<typename VecX::value_type, typename VecY::value_type>::type>
diff_two_norm (const VecX &X, const VecY &Y);

// The max norm of the difference between two vectors.
template <class VecX, class VecY>
cm2::biggest2_type<typename VecX::value_type, typename VecY::value_type>::type>
diff_max_norm (const VecX &X, const VecY &Y);

// The dot product of the difference between two vectors and a third vector.
template <class VecX, class VecY, class VecZ>
cm2::biggest3_type<typename VecX::value_type, typename VecY::value_type, typename VecZ::value_
type>::type>
diff_dot (const VecX &X, const VecY &Y, const VecZ &Z);

// Cross product between two vectors.
template <class VecX, class VecY, class VecZ>
void cross (const VecX &x, const VecY &y, VecZ &z);

// Cross product between two vectors, with a scaling factor.
template <class T, class VecX, class VecY, class VecZ>
void cross (T a, const VecX &x, const VecY &y, VecZ &z);

// Copies the cross product between two vectors.
template <class VecX, class VecY, class VecZ>
void copy_cross (const VecX &x, const VecY &y, VecZ &z);

// Copies the cross product between two vectors, with a scaling factor.
template <class T, class VecX, class VecY, class VecZ>
void copy_cross (T a, const VecX &x, const VecY &y, VecZ &z);

// The cross product between two vectors (2-D version).
template <class IterX , class IterY>
cm2::biggest2_type<typenamestd::iterator_traits<IterX>::value_type, typenamestd::iterator_
traits<IterY>::value_type>::type>
cross2 (IterX x, IterY y);

// The square of the two-norm of the cross product between two vectors.
template <class IterX , class IterY>
cm2::biggest2_type<typenamestd::iterator_traits< IterX>::value_type, typenamestd::iterator_
traits<IterY>::value_type>::type>
cross_sqr_two_norm3 (IterX x, IterY y);

// The two-norm of the cross product between two vectors.
template <class IterX , class IterY>
cm2::biggest2_type<typenamestd::iterator_traits<IterX>::value_type, typenamestd::iterator_
traits<IterY>::value_type>::type>
cross_two_norm3 (IterX x, IterY y);

// The square of the two-norm of the cross product between two vectors.
template <class VecX, class VecY>
cm2::biggest2_type<typename VecX::value_type, typename VecY::value_type>::type>
cross_sqr_two_norm (const VecX &x, const VecY &y);

// The two-norm of the cross product between two vectors.
template <class VecX, class VecY>
cm2::biggest2_type<typename VecX::value_type, typename VecY::value_type>::type>
cross_two_norm (const VecX &x, const VecY &y);

// The mixed product of two vectors.
template <class IterX, class IterY , class IterZ>
cm2::biggest3_type<typename std::iterator_traits<IterX>::value_type, typename std::iterator_
traits<IterY>::value_type, typename std::iterator_traits<IterZ>::value_type>::type>
mixed_product (IterX x, IterY y, IterZ z);

// Tests the equality between two vectors.
template <class VecX, class VecY>
bool equal (const VecX &X, const VecY &Y);

// Tests the collinearity between two vectors.
```

```
template <class VecX, class VecY>
bool are_collinear (const VecX &x, const VecY &y, double tol = CM2_EPSILON);

// Tests the orthogonality between two vectors.
template <class VecX, class VecY>
bool are_orthogonal (const VecX &x, const VecY &y, double tol = CM2_EPSILON);
```

Table 3 – The `cm2::vecvec` template math library.

Matrix - Scalar functions (cm2::matscal)

```
// Appends a scalar to a 1-row matrix.
template <class Matrix>
bool push_back (typename Matrix::value_type v, Matrix &A);

// The determinant of a 1x1, 2x2 or 3x3 matrix.
template <class Matrix>
typename Matrix::value_type det (const Matrix &A);

// A cofactor of a 2x2 or 3x3 matrix.
template <class Matrix>
typename Matrix::value_type cofactor (size_t i, size_t j, const Matrix &A);

// Sets the diagonal of a matrix to a scalar value.
template <class Matrix>
void copy_diag (typename Matrix::value_type v, Matrix &A);

// Adds a scalar to the diagonal of a matrix.
template <class Matrix>
void add_diag (typename Matrix::value_type v, Matrix &A);

// Multiplies the diagonal of a matrix by a scalar value.
template <class Matrix>
void mult_diag (typename Matrix::value_type v, Matrix &A);

// Divides the diagonal of a matrix by a scalar value.
template <class Matrix>
void div_diag (typename Matrix::value_type v, Matrix &A);

// The trace of a matrix.
template <class Matrix>
typename Matrix::value_type trace (const Matrix &A);

// Sets a row of a matrix to a scalar value.
template <class Matrix>
void copy_row (typename Matrix::value_type v, Matrix &A, size_t i);

// Adds a scalar value to a row of a matrix.
template <class Matrix>
void add_row (typename Matrix::value_type v, Matrix &A, size_t i);

// Multiplies a row of a matrix by a scalar value.
template <class Matrix>
void mult_row (typename Matrix::value_type v, Matrix &A, size_t i);

// Divides a row of a matrix by a scalar value.
template <class Matrix>
void div_row (typename Matrix::value_type v, Matrix &A, size_t i);

// Negates a matrix.
template <class Matrix>
void negate (Matrix &A);

// Copies a scalar to a matrix.
template <class Matrix>
void copy (typename Matrix::value_type v, Matrix &A);

// Adds a scalar to a matrix.
template <class Matrix>
void add (typename Matrix::value_type v, Matrix &A);

// Subtract a scalar to a matrix.
template <class Matrix>
void subtract (typename Matrix::value_type v, Matrix &A);

// Multiplies a matrix by a scalar.
template <class Matrix>
void mult (typename Matrix::value_type v, Matrix &A);

// Divides a matrix by a scalar.
template <class Matrix>
void div (typename Matrix::value_type v, Matrix &A);

// The one-norm.
template <class Matrix>
typename Matrix::value_type one_norm (const Matrix &A);

// The square of two-norm.
template <class Matrix>
typename Matrix::value_type sqr_two_norm (const Matrix &A);
```

```
// The two-norm (usual Euclidean norm).
template <class Matrix>
typename Matrix::value_type two_norm (const Matrix &A);

// Normalizes a matrix (divides by its two norm).
template <class Matrix>
typename Matrix::value_type normalize (Matrix &A);

// The max-norm (infinite norm).
template <class Matrix>
typename Matrix::value_type max_norm (const Matrix &A);

// The maximum value.
template <class Matrix>
Matrix::value_type max_value (const Matrix &A);

// The minimum value.
template <class Matrix>
typename Matrix::value_type min_value (const Matrix &A);

// Increases a range by the range of a matrix.
template <class Matrix , class Range>
void inc_range (const Matrix &A, Range &R);

// The range of a matrix (first = min, second = max).
template <class Matrix>
std::pair<typename Matrix::value_type, typename Matrix::value_type> range (const Matrix &A);

// The number of elements in a vector that match a predicate.
template <class Matrix , class Predicate>
size_t count_if (const Matrix &A, Predicate pred);

// Tests all elements in a matrix against a scalar.
template <class Matrix>
bool equal (const Matrix &A, typename Matrix::value_type v);

// Has the matrix only normal values (float and double only).
template <class Matrix>
bool isfinite (const Matrix &A);

// The pseudo determinant of a matrix A of size MxN.
template <class Matrix>
typename Matrix::value_type pseudo_det (const Matrix &A);

// Symmetrizes a matrix by adding its transpose matrix.
template <class MatA >
void symmetrize (MatA &A);
```

Table 4 – The `cm2::matscal` template math library.

Matrix - Vector functions (`cm2::matvec`)

```
// Inserts a new column inside a matrix.
template <class Vec, class Mat>
void insert (const Vec& V, size_t j, Mat& A);

// Removes a column from a matrix.
template <class Mat>
void remove (size_t j, Mat& A);

// Appends a column to a matrix.
template <class Matrix, class Vector>
bool push_back (const Vector &V, Matrix &A);

// Copies a vector to the diagonal of a matrix.
template <class Matrix, class Vec>
void set_diag (const Vec &V, Matrix &A);

// Copies the diagonal of a matrix into a vector.
template <class Matrix, class Vec>
void get_diag (const Matrix &A, Vec &V);

// Adds a vector to the diagonal of a matrix.
template <class Matrix, class Vec>
void add_diag (const Vec &V, Matrix &A);

// Subtracts a vector to the diagonal of a matrix.
template <class Matrix, class Vec>
void subtract_diag (const Vec &V, Matrix &A);

// Multiplies the diagonal of a matrix by a vector (element by element).
template <class Matrix, class Vec>
void mult_diag (const Vec &V, Matrix &A);

// Divides the diagonal of a matrix by a vector (element by element).
template <class Matrix, class Vec>
void div_diag (const Vec &V, Matrix &A);

// Copies a row of a matrix into a vector.
template <class Matrix, class Vector>
void get_row (const Matrix &A, size_t i, Vector &V);

// Copies a vector into a row of a matrix.
template <class Matrix, class Vector>
void set_row (const Vector &V, Matrix &A, size_t i);

// Adds a vector to a row of a matrix.
template <class Matrix, class Vector>
void add_row (const Vector &V, Matrix &A, size_t i);

// Adds a row of a matrix to a vector.
template <class Matrix, class Vector>
void add_row (const Matrix &A, size_t i, Vector &V);

// AXPY between a row of a matrix and a vector.
template <class T, class Matrix, class Vector>
void axpy_row (T a, const Vector &V, Matrix &A, size_t i);

// AXPY between a row of a matrix and a vector.
template <class T, class Matrix, class Vector>
void axpy_row (T a, const Matrix &A, size_t i, Vector &V);

// Multiplies a matrix by a vector, with scaling.
template <class T, class Matrix, class VecX , class VecY>
void mult (T a, const Matrix &A, const VecX &x, VecY &y, int nthreads=1);

// Multiplies a matrix by a vector.
template <class Matrix, class VecX , class VecY>
void mult (const Matrix &A, const VecX &x, VecY &y, int nthreads=1);

// Multiplies a transposed matrix by a vector, with scaling.
template <class T, class Matrix, class VecX , class VecY>
void transpose_mult (T a, const Matrix &A, const VecX &x, VecY &y, int nthreads=1);

// Multiplies a transposed matrix by a vector.
template <class Matrix, class VecX , class VecY>
void transpose_mult (const Matrix &A, const VecX &x, VecY &y, int nthreads=1)

// transpose(V) % A % V
template <class Matrix, class Vec>
cm2::biggest2_type<typenameMatrix::value_type, typenameVec::value_type>::type>
Vt_A_V (const Matrix &A, const Vec &V, int nthreads=1);
```

```
// Rank-1 update with one vector.
template <class T, class Vec, class Matrix>
void rank1 (T a, const Vec &x, Matrix &A);

// Rank-1 update with one vector.
template <class Vec, class Matrix>
void rank1 (const Vec &x, Matrix &A);

// Rank-2 update between two vectors.
template <class T, class VecX, class VecY, class Matrix>
void rank2 (T a, const VecX &x, const VecY &y, Matrix &A);

// Symmetric rank-2 update between two vectors.
template <class T, class VecX, class VecY, class Matrix>
void rank2sym (T a, const VecX &x, const VecY &y, Matrix &A);

// Cross products between all columns of a matrix and a vector.
template <class Vec, class Matrix>
void cross (const Matrix &A, const Vec &x, Matrix &B);

// Cross products between a vector and all columns of a matrix.
template <class Vec, class Matrix>
void cross (const Vec &x, const Matrix &A, Matrix &B);

// Skew matrix update (hat operator).
template <class Vec, class Matrix>
void skew (const Vec &x, Matrix &A);

// Skew matrix update, with scaling (hat operator).
template <class T, class Vec, class Matrix>
void skew (T a, const Vec &x, Matrix &A);

// Square skew matrix update, with scaling (square hat operator).
template <class T, class Vec, class Matrix>
void square_skew (T a, const Vec &V, Matrix &A);

// Square skew matrix update(square hat operator).
template <class Vec, class Matrix>
void square_skew (const Vec &V, Matrix &A);
```

Table 5 – The `cm2::matvec` template math library.

Matrix - Matrix functions (cm2::matmat)

```
// Inserts some new columns at specific positions in a matrix.
template <class Mat1, class VecI, class Mat>
void insert (const Mat1& M1, const VecI& indices, Mat& M);

// Removes some columns from a matrix.
template <class VecI, class Mat>
void remove (const VecI& indices, Mat& A);

// Appends a matrix to another one.
template <class Mat1, class Mat2>
bool push_back (const Mat1 &M1, Mat2 &M2);

// Appends a batch of matrices to another one.
template <class Mat1Iterator, class Mat2>
bool push_back (Mat1Iterator M1_beg, Mat1Iterator M1_end, Mat2 &M2);

// Appends a specific batch of columns to a matrix.
template <class Mat1, class Mat2, class Vector>
bool push_back (const Mat1 &M1, const Vector &M1_cols, Mat2 &M2);

// Appends a specific column of a matrix to another a matrix.
template <class Mat1, class Mat2>
bool push_back (const Mat1 &M1, size_t j1, Mat2 &M2);

// Tests the equality between two matrices.
template <class Mat1, class Mat2>
bool equal (const Mat1 &M1, const Mat2 &M2);

// Copies  between the diagonal of two matrices, with shift.
template <class T, class MatA, class MatB>
void copy_diag (T a, const MatA &A, MatB &B, size_t shift=0);

// AXPY between the diagonal of two matrices.
template <class T, class MatA, class MatB>
void axpy_diag (T a, const MatA &A, MatB &B, size_t shift=0);

// Copies a row of a matrix to a row of another matrix.
template <class MatA, class MatB>
void copy_row (const MatA &A, size_t ia, MatB &B, size_t ib);

// Adds a row of a matrix to a row of another matrix.
template <class MatA, class MatB>
void add_row (const MatA &A, size_t ia, MatB &B, size_t ib);

// AXPY a row of a matrix to a row of another matrix.
template <class MatA, class MatB, class T>
void axpy_row (T a, const MatA &A, size_t ia, MatB &B, size_t ib);

// AXPY between two matrices.
template <class MatA, class MatB, class T>
void axpy (T a, const MatA &A, MatB &B);

// Adds a matrix to another matrix.
template <class MatA, class MatB>
void add (const MatA &A, MatB &B);

// Subtracts a matrix to another matrix.
template <class MatA, class MatB>
void subtract (const MatA &A, MatB &B);

// Copies with scaling a matrix to another matrix.
template <class MatA, class MatB, class T>
void copy_scale (T a, const MatA &A, MatB &B);

// Copies a matrix to another matrix.
template <class MatA, class MatB>
void copy (const MatA &A, MatB &B);

// Adds the transpose of a matrix into another matrix.
template <class MatA, class MatB>
void transpose (const MatA &A, MatB &B);

// Matrix-matrix multiplication, with scaling.
template <class T, class MatA, class MatB, class MatC>
void mult (T a, const MatA &A, const MatB &B, MatC &C, size_t cache_size=0, int nthreads=1);

// Matrix-matrix multiplication.
template <class MatA, class MatB, class MatC>
void mult (const MatA &A, const MatB &B, MatC &C, size_t cache_size=0, int nthreads=1)
```

```
// Matrix-matrix transpose multiplication, with scaling.
template <class T, class MatA, class MatB, class MatC>
void transpose_mult (T a, const MatA &A, const MatB &B, MatC &C, size_t cache_size=0, int
nthreads=1);

// Matrix-matrix transpose multiplication.
template <class MatA, class MatB, class MatC>
void transpose_mult (const MatA &A, const MatB &B, MatC &C, size_t cache_size=0, int
nthreads=1);

// Diagonal mult (with scaling).
template <class T, class DiagD, class MatB, class MatC>
void diag_mult (T a, const DiagD &D, MatB &B, MatC &C, int nthreads=1);

// Diagonal mult (with scaling).
template <class DiagD, class MatB>
void diag_mult (const DiagD &D, MatB &B, int nthreads=1);

// transpose(B) % A % B
template <class T, class MatA, class MatB, class MatC>
void Bt_A_B (T a, const MatA &A, const MatB &B, MatC &C, size_t cache_size=0, int nthreads=1);

// B % A % transpose(B)
template <class T, class MatA, class MatB, class MatC>
void B_A_Bt (T a, const MatA &A, const MatB &B, MatC &C, size_t cache_size=0, int nthreads=1);

// Generalized transposed inverse of a MxN matrix, with (M,N) = (2,2), (3,2) or (3,3).
template <class Matrix , class T>
void transpose_inverse (const Matrix &A, Matrix &invT_A, T &det);

// Compute the lower-left Cholesky factorization of a small symmetric real matrix.
template <class Matrix , class Factor>
bool Cholesky_factor (const Matrix &A, Factor &L);
```

Table 6 – The `cm2::matmat` template math library.

COMPUTING
OBJECTS