Mondrian

Parallel blocked matrix library, v0.0

Current snapshot: January 2017

Risi Kondor
Department of Computer Science, The University of Chicago

Contents

Overview

Mondrian is a C++11 software library intended to serve as the intermediate layer between low level matrix libraries such as `BLAS` and high level numerical algebra, optimization and machine learning code. Mondrian was originally developed as in-house code for implementing parallel matrix algorithms in the author's research group. The library was designed with the following objectives in mind:

1. **Interoperability:** Mondrian provides a unified interface to a range of dense and sparse vector/matrix formats, making it possible to write high level code that does not commit to a specific low level matrix representation.
2. **Performance:** Mondrian is designed with large scale computations on parallel architectures in mind, so every effort was made to make the code as efficient as possible.
3. **Ease of use:** Mondrian has a simple syntax and consistent, object oriented design.
4. **Easy linking:** Mondrian is a template library, therefore the overhead of installing it is minimal. To call Mondrian from C++11 code, just include the appropriate header files[1].

Mondrian is in the development phase and has not yet reached the stage of a numbered release. The present snapshot of the code is released to the community on a strictly "as-is" basis. In the following list of features, features whose implementation or documentation is not yet complete appear in gray.

1. **Vector/matrix classes:**
   (a) Custom dense matrix/vector classes based on native C arrays (`Cvector` and `Cmatrix`).
   (b) Custom sparse matrix/vector classes based on `stl` containers (`Vectorv`, `Vectorl`, `Vectorh`, etc.).
   (c) Custom matrix types for representing low rank matrices.
   (d) Interface to `Eigen`.
2. **Blocked vector/matrix classes:**
   (a) Blocked vector/matrix template classes supporting the same linear algebra operations as the elementary matrix types, but with automatic block-level parallelism.
   (b) Support for hierarchical matrices (HODLR, H, H_2, HSS) via recursive blocking.
   (c) Fast, parallel routines for blocking matrices by clustering rows/columns, and for reblocking matrices from one block structure to another.
3. **Operators:**
   (a) Specialized, highly optimized classes for elementary operators such as Givens rotations, k–point rotations, and so on.
4. **Abstractions:**
   (a) Matrix/vector views providing access to subvectors/submatrices without copying elements.
   (b) Methods for mapping functions or lambda expressions over elements of objects such as vectors and matrices in a functional programming style.
   (c) Active containers that automatically perform certain operations when their contents in changed.
   (d) Polymorphic vector/matrix classes that delegate operations to the derived type.
   (e) Generalized matrix classes to encapusulate products/sums of operators and other matrices.
   (f) Atomic vectors/matrices protected from race conditions in multithreaded code.

[1] Currently the only exceptions to this rule are the `CmatrixLA` and `SymmCmatrixLA` classes, which provide functionality from `Eigen`[1], and are separately compiled to isolate the two libraries and thus reduce compilation time.

5. **Parallelism:**
   (a) Multithreading support with a configurable custom thread scheduler.
6. **Transparency and convenience functions:**
   (a) Consistent object oriented design fully utilizing C++11 features.
   (b) Expression templates for matrix/vector operations.
7. **Input/output:**
   (a) Capability to load/save vectors and matrices in some of the most common file formats.
   (b) Capability to load/save almost all objects in binary format.
   (c) Ability to print most objects directly to `stdout` for diagnostic purposes.
8. **Interfaces to other languages and environments:**
   (a) Matlab and Python interfaces to most objects.

Mondrian was conceived and developed at The University of Chicago by Risi Kondor, starting in 2016. Certain parts of the library are based on the earlier `pMMF` library by Risi Kondor, Nedelina Teneva and Pramod K Mudrakarta [2].

Usage

Mondrian is distributed in C++ source code format. Using the library requires

1. A C++11 compatible compiler, such as clang or a recent version of gcc.
2. The Standard Template Library (STL) included in any C++11 installation.

The following optional components have additional dependencies:

1. The `CmatrixLA` and `SymmCmatrixLA` classes require the `Eigen` linear algebra library for higher level linear algebra operations [1].

Installation

Mondrian is primarily a header library, that does not require any specific installation or compilation process. To use the library it is sufficient to download it from `http://github.com/risi-kondor/Mondrian`, place it in an appropriate location on your local file system, and call Mondrian objects directly from your own C++ programs.

A small subset of the library (at the present time only the `CmatrixLA` and `SymmCmatrixLA` classes) plus the tutorial examples need to be compiled into object files/executables. To this end:

1. Edit `Makefile.options` to reflect the location of certain components on your system (see table below).
2. At the root level of the library issue the command `make all` (it is assumed that make is installed on your system).

Parameter	Description
`CC`	Name of the compiler. Example: `clang`.
`EIGENDIR`	Path to the directory in which `Eigen` is found. If empty, Mondrian will be compiled without `Eigen` support. Example: `/usr/local/include`.

Customization

The library can be customized by changing the following typedefs and preprocessor variables in the global header file `include/Mondrian_base.hpp`.

Type name	Default	Description
`SCALAR`	`double`	The basic numeric type used in all `Vector` and `Matrix` objects, as well as most computations.
`INDEX`	`int`	The type used for vector/matrix indices (in part of the code this is fixed as `int`).

Variable name	Default	Description
_UTILITYCOPYWARNING _MATRIXCOPYWARNING _BLOCKEDCOPYWARNING _MMFCOPYWARNING	set	Deep copying/assigning large objects is an expensive operation which, for the most part, should be avoided. If these variables are set (defined), a warning will be written to `cout` whenever an object of the given category is copied or assigned.
_UTILITYMOVEWARNING _MATRIXMOVEWARNING _BLOCKEDMOVEWARNING _MMFMOVEWARNING	not set	C++11's so-called move semantics can often circumvent having to make expensive deep copies of objects. If these variables are set (defined), a warning will be written to `cout` whenever an object of the given category is moved or move-assigned.

Calling Mondrian objects

To access Mondrian objects, simply `#include` the appropriate Mondrian headers in your source code. Note that Mondrian has its own namespace, therefore, to use, e.g., the `Cmatrix` class, refer to it as `Mondrian::Cmatrix`, or use the command `using namespace Mondrian;`.

Mondrian uses a small number of global variables (see below for a partial list). To make sure that these are approriately defined, any top level executable should `#include` the file `include/Mondrian_base.inc`.

Variable name	Default	Description
`Mondrian::multithreading`	`true`	Multithreading is disabled if `false`.
`Mondrian::threadManager`		The global thread manager object. The maximum number of threads that can be simultaneously active is controlled by setting `threadManager.maxthreads`.

Tutorial Examples

1. Basic matrix/vector operations

The following example demonstrates how to define vectors and matrices in Mondrian and perform elementary linear algebra operations.

```
1   #include "Cmatrix.hpp"
2   #include "Mondrian_base.inc"
3   using namespace Mondrian;
4
5   int main(int argc, char** argv){
6
7     Cvector v={1,0,3};
8     cout<<"v="<<v<<endl<<endl;
9
10    SCALAR s=v.dot(v);
11    cout<<"s="<<s<<endl<<endl;
12
13    Cmatrix A={{1,2,3},{4,5,6},{7,8,9}};
14    cout<<"A="<<endl<<A<<endl;
15
16    Cvector u=A*v;
17    cout<<"u="<<u<<endl<<endl;
18
19  }
```

example1.cpp

Most of the code is self-explanatory:

- Line 7 creates a vector $\mathbf{v} = (1, 0, 3)^\top$ and stores it in a `Cvector` object.
- Line 10 computes the dot product $\mathbf{v} \cdot \mathbf{v}$ (equivalently, $\mathbf{v}^\top \mathbf{v}$).
- Line 13 creates the `Cmatrix` $A = [1, 4, 7; 2, 5, 8; 3, 6, 9]$ (in Matlab notation). Note that the matrix elements are listed in column major order.
- Line 16 computes the matrix/vector product $A\mathbf{v}$.

Note the simple syntax for printing each of the objects `v`, `A` and `s` to standard output in lines 8, 11, 14 and 17. Also note the `#include` statement in line 2: `Mondrian_base.inc` declares certain static objects required by the library, so this file must be included in all top level executables. Finally, the `using namespace Mondrian` directive is included to make the code a little easier to read: without it, we would have to preface the name of each Mondrian object such as `Cvector`, `Cmatrix`, etc. with `Mondrian::`. The output of the code is as follows.

```
v=(1,0,3)

s=10

A=
[  1.000  4.000  7.000  ]
[  2.000  5.000  8.000  ]
[  3.000  6.000  9.000  ]

u=(22,26,30)
```

Output of `example1.cpp`

2. Accessing matrix/vector elements

The following example demonstrates the basic ways to access individual vector/matrix elements.

```
#include "Cmatrix.hpp"
#include "Mondrian_base.inc"
using namespace Mondrian;

int main(int argc, char** argv){

  Cvector v=Cvector::Gaussian(5);
  cout<<"v="<<v<<endl;
  cout<<"v(2) = "<<v(2)<<endl<<endl;

  Cmatrix A=Cmatrix::Bernoulli(5,5);
  cout<<A<<endl;
  cout<<"A(2,3) = "<<A(2,3)<<endl;

  A(2,3)=2;
  cout<<"A(2,3) = "<<A(2,3)<<endl;

  cout<<"row 1   : "<<A.row<Cvector>(1)<<endl;
  cout<<"column 2: "<<A.column<Cvector>(2)<<endl;
  cout<<"diagonal: "<<A.diag<Cvector>()<<endl;
  //cout<<"submatrix:"<<A.submatrix(2,2)<<endl;

}
```

`example2.cpp`

```
v=(-1.23974,-0.407472,1.61201,0.399771,1.3828)
v(2) = 1.61201

[  0.000  0.000  1.000  0.000  0.000  ]
[  0.000  1.000  1.000  0.000  0.000  ]
[  0.000  1.000  1.000  0.000  0.000  ]
[  1.000  0.000  1.000  0.000  1.000  ]
[  1.000  0.000  1.000  1.000  1.000  ]

A(2,3) = 0
A(2,3) = 2
row 1   : (0,1,1,0,0)
column 2: (1,1,1,1,1)
diagonal: (0,1,1,0,1)
```

Output of `example2.cpp`

3. Copying, moving and assigning objects

The following example illustrates four different ways of copying/assigning objects.

```
#include "Cmatrix.hpp"
#include "Mondrian_base.inc"
using namespace Mondrian;

int main(int argc, char** argv){

  Cmatrix A(4,4); //        Construct a 4-by-4 matrix called A

  for(int i=0; i<4; i++) //    Fill its entries with something
    for(int j=0; j<4; j++)
      A(i,j)=i+j;

  cout<<"A="<<endl<<A<<endl; //                    Print out A

  Cmatrix B(A); //                Initialize B from A (copying)
  cout<<"B="<<endl<<B<<endl;

  Cmatrix C;
  C=A; //                                  Set C=A (assignment)
  cout<<"C="<<endl<<C<<endl;

  Cmatrix D;
  D=A*A; //                          Set D=A*A (move-assignment)
  cout<<"D="<<endl<<D<<endl;

  Cmatrix E=A.copy(); //                 Set E=A (silent copy)
  cout<<"E="<<endl<<E<<endl;
}
```

`example3.cpp`

The four different ways are the following:

(a) Line 15 constructs `B` from `A`, which is done with the `Cmatrix` class's *copy constructor*.

(b) Line 19 set `C` to the same value as `A`, which again involves explicit copying, this time using the the `Cmatrix` class's *assignment operator*.

(c) Line 23 sets `D=A*A` which seemingly again involves calling the assignment operator. However, as this statement is executed, the compiler knows that `A*A` is a temporary object (a so-called r-value), therefore it can use the *move-assignment operator* which is generally more efficient because it can avoid explicit copying.

(d) Line 26 is equivalent to writing `E=A` except that the `copy()` function suppresses the copy warning.

```
 1  A=
 2  [  0.000  1.000  2.000  3.000  ]
 3  [  1.000  2.000  3.000  4.000  ]
 4  [  2.000  3.000  4.000  5.000  ]
 5  [  3.000  4.000  5.000  6.000  ]
 6
 7  Warning: "Cmatrix" copied.
 8  B=
 9  [  0.000  1.000  2.000  3.000  ]
10  [  1.000  2.000  3.000  4.000  ]
11  [  2.000  3.000  4.000  5.000  ]
12  [  3.000  4.000  5.000  6.000  ]
13
14  Warning: "Cmatrix" assigned.
15  C=
16  [  0.000  1.000  2.000  3.000  ]
17  [  1.000  2.000  3.000  4.000  ]
18  [  2.000  3.000  4.000  5.000  ]
19  [  3.000  4.000  5.000  6.000  ]
20
21  D=
22  [ 14.000 20.000 26.000 32.000  ]
23  [ 20.000 30.000 40.000 50.000  ]
24  [ 26.000 40.000 54.000 68.000  ]
25  [ 32.000 50.000 68.000 86.000  ]
26
27  E=
28  [  0.000  1.000  2.000  3.000  ]
29  [  1.000  2.000  3.000  4.000  ]
30  [  2.000  3.000  4.000  5.000  ]
31  [  3.000  4.000  5.000  6.000  ]
```

Output of `example3.cpp`

Note that lines 15 and 19 (by default) generate explicit copy warnings. To an inexperienced programmer it is sometimes not clear where exactly the compiler will make explicit copies, which, in the case of large objects can be very expensive. Mondrian tries to discourage unnecessary copying by issuing these explicit warnings. In the rare cases where it is absolutely necessary to make a copy, instead of standard copy constructor or assignment operator, the `copy` method should be used, which makes it immediately obvious in the code that a copy is being constructed, but does not issue a copy warning at run time.

4. Mapping functions over vectors and matrices

Mondrian supports functional programming style operations that maps functions or lambda expressions over elements of objects. In the following example, a lambda expression is mapped over each element of vector, to compute the sum of the elements.

```
1   #include "Cmatrix.hpp"
2   #include "Mondrian_base.inc"
3   using namespace Mondrian;
4
5   int main(int argc, char** argv){
6
7     Cvector u=Cvector::Gaussian(7);
8     cout<<"u="<<u<<endl;
9
10    double t=0;
11    u.for_each([&t](int i, SCALAR d){t+=d;});
12    cout<<"Sum of elements: "<<t<<endl;
13
14  }
```

example4.cpp

```
1   u=(-1.23974,-0.407472,1.61201,0.399771,1.3828,0.0523187,-0.904147)
2   Sum of elements: 0.895546
```

Output of example4.cpp

Classes

General design

Standard methods

The following standard methods are implemented by most classes in Mondrian, and are not listed separately for each class. `CLASS` stands for the name of the class, and `x` for the class instance.

CONSTRUCTORS AND COPYING

`CLASS(const CLASS& y)`
: Construct a deep copy of `y`. Making a deep copy involves copying not just the member variables of `y`, but also recursively constructing a copy of every object owned by `y`. For large objects this is an expensive operation, which can often be avoided with the move-contructor paradigm (see below). To discourage deep copying, by default, the copy constructor of many classes, including all matrix classes, prints a copy warning to standard output.

`CLASS(CLASS&& y)`
: Move-construct a copy of `y`. In C++11, `&&` signifies an rvalue reference, which means that this method is invoked instead of the regular copy constructor when `y` is a temporary. In constrast to deep copying, the move constructor changes the ownership of each object owned by `y`, rather than copying them, potentially resulting in large run-time savings. Mondrian extensively uses this paradigm.

`CLASS copy()`
: Create a deep copy of `x`. The behavior of this member function is identical to the behavior of the copy constructor *except* that it does not generate a copy warning. This is the function that should be used when it is unavoidable to make an explicit copy of an object such as a matrix.

`CLASS shallow()`
: Create a shallow copy of `x`, i.e., copy each of its member variables and pointers to the objects it owns, without making copies of the owned objects. This is the function used by the `Detached<CLASS>` wrapper to create a detached version of `x`. Therefore, every class derived from the abstract class `Detachable` must implement this method.

ASSIGNMENT OPERATORS

`CLASS& operator=(const CLASS& y)`
: Delete the current content of `x`, make `x` a deep copy of `y`, and finally return a reference to `x`. Similarly to the copy constructor, in most classes the assignment operator prints a warning to standard output.

`CLASS& operator=(CLASS&& y)`
: Move-assign `y` to `x`. The same as above, except with move semantics, similarly to the move-copy constructor.

DESTRUCTOR

`~CLASS()`
: Recursively delete every object owned by `x`, and then delete `x` itself.

COMPARATORS

`bool operator==(const CLASS& y)`
: The equality operator. Returns `TRUE` if `x` is equal to `y`.

`bool operator!=(const CLASS& y)`
: Returns `TRUE` if `x` is not equal to `y`.

BINARY I/O (SERIALIZATION)

`CLASS(const Filename& filename)`
`saveto(const Filename& filename)`
: Load/save `x` form/to the binary file named `filename`.

`CLASS(Bifstream& ifs)`
`serialize(Bofstream& ofs)`
: Load `x` from the binary file stream `ifs` / save `x` to the binary file stream `ofs`.

PRETTY PRINTING

`static string classname()`
: Return the name of the class `CLASS`.

`string str()`
: Return a human-readable representation of `x` as a string. In some classes, `str` can take arguments, for example, `Dense()`, to signify that a matrix is to be printed to string in dense format.

`ostream&::operator<<(ostream& os, const CLASS& x)`
: Write a human-readable representation of `x` to the stream `os`.

Property classes

Mondrian uses abstract classes to signify that classes derived from them have certain specific properties:

1. `Detachable`. The `Detached<CLASS>` wrapper allows constructing an interface to data stored in another object. The resulting object is said to be detached, because deleting it does not delete the original object (of type `CLASS`). `Detachable` is the abstract base class of all classes whose instances can be detached via this mechanism.
2. `Serializable`. The process of recursively saving objects in a binary file is called serialization. `Serializable` is the abstract base class of all classes that support this facility.
3. `Interruptable`. An interruptable method in a class derived from the abstract class `Interruptable` can be stopped from another thread by calling the `halt()` method. This is useful for classes with time consuming operations on large data objects.

Packages

It often happens that a function needs to return two or more separate objects (e.g., the eigenvalues and eigenvectors of a matrix). In Mondrian this is accomplished with the `package` helper class.
Let us assume that `foo()` is a function that needs to return a `Cmatrix A` and a `Cvector v`. Then:

(a) The return type of `foo` would be `package<Cmatrix,Cvector>`.
(b) The actual return statement in `foo` is `return package<Cmatrix,Cvector>(A,v);`.
(c) If `P` is the package returned by `foo`, then the matrix part is extracted by calling `P.first()` and the vector part by calling `P.second()` (other ways are also possible).

All the above operations are done by move constructors and move assignment operators, so the overhead of using packages is usually minimal.

Downcasting operators

Mondrian has various instances of base class/derived class (parent class/child class) pairs where the two classes share the same data layout, and only differ in some of their methods. A simple example are the matrix classes `Cmatrix` and `SymmCmatrix`, which store matrices in exactly the same format, but in the latter case the matrix is assumed to be symmetric, hence some operations on it can be performed more efficiently.

Assume that a base class `Base` defines a method called `foo` that is overriden by a derived class `Derived`. Then if `x` is of type `Derived`, but for some reason one temporarily wishes to treat it as if it were a base class object (called upcasting), and call the `foo` method of the base class on it, this can easily be achieved by `x.Base::foo(...)`. However, in general, the opposite action of temporarily downcasting an object is not possible, since the base class need not have any knowledge of the derived class.

Mondrian offers a solution to this problem involving move-copying to a temporary object (which, in the above example would be called `AsDerived`, but is opaque to the user) and appropriate convenience functions. In particular:

(a) If `A` is a matrix object of any non-symmetric matrix type `MATRIX`, the convenience function `as_symmetric` temporarily downcasts it to the corresponding symmetric matrix type `SymmMatrix`.
(b) If `A` is a matrix object of type `MATRIX`, the convenience function `linalg` downcasts it to type `MATRIXLA`, which supports externally called linear algebra operations.
(c) If `A` is a non-multithreaded object of type `CLASS`, the convenience function `as_multithreaded` can downcast it to the corresponding multithreaded type `CLASSm`.

Example:

```
1   // The 12 dimensional identity matrix
2   Cmatrix A=Cmatrix::Identity(12);
3
4   // A random Givens rotation
5   GivensRotation Q=GivensRotation::Random(12);
6
7   // Conjugate A with Q without taking advantage of symmetry
8   A.conjugate(Q);
9
10  // Conjugate A with Q with taking advantage of symmetry
11  as_symmetric(A).conjugate(Q);
```

Downcasting operators are also useful for differentiating between downcasting by conversion vs. downcasting by assumption, as illustrated by the following example:

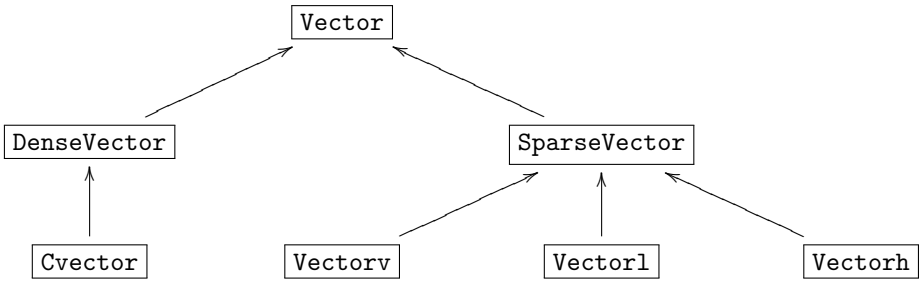
```
1   Cmatrix A(12);
2
3   // Construct B by explicitly symmetrizing A, i.e., B=(A+A^T)/2
4   SymmCmatrix B(A);
5
6   // Construct C by assuming that A is symmetric
7   SymmCmatrix C(as_symmetric(A));
```

Vectors and matrices

1. Basic vectors

Mondrian supports both dense and sparse vector arithmetic. The basic dense vector class is `Cvector`, which simply stores the vector as a dense C-style array of numbers. The basic sparse vector classes are `Vectorv`, `Vectorl` and `Vectorh`, all three of which are based on `stl` containers: `Vectorv` stores the vector as an `stl::vector` of (index, value) pairs, `Vectorl` as a linked list, and `Vectorh` as a hash table. Which one of these three classes is the most appropriate for a given application depends on many factors, such as the size of the vector, what operations are to be performed on it, how often elements are to be inserted, and so on.

The important feature of each of these vector classes is that since they all inherit the same interface defined in the abstract class `Vector`, they are fully interchangeable. The preferred method for writing polymorphic code in Mondrian, specifically code that will work with any vector class, is the use of templates (generic programming). However, the library does provide an additional class called `GenericVector`, which, by encapsulating any of the elementary vector types and delegating operations to their respective methods, can function as a universal polymorphic vector class. This functionality however comes at a slight performance overhead, and may not be compatible with some higher level features, such as active vectors.



Vector

`Vector` is the abstract class that defines the common interface to all classes in Mondrian that represent finite dimensional vectors, $\mathbf{v} = (v_1, v_2, \ldots, v_n)^\top \in \mathbb{F}^n$. Each vector component v_i (also referred to as the i'th *element* of $\mathbf{v}$) is of type `SCALAR`. Every concrete vector class `VECTOR` provides each of the constructors and methods listed below.

Mondrian supports both dense and sparse vectors (see the attribute `isSparse()`). Sparse vector classes only allocate storage for a subset of their elements. These elements are said to be *filled in*. Typically the filled in elements are the non-zero elements of $\mathbf{v}$, although when a non-zero elkelent is set to zero, it might remain filled in. In dense vector classes all elements are filled in.

CONSTRUCTORS

`VECTOR()`
: Create a new zero dimensional vector.

`VECTOR(int n)`
: Create a new n dimensional vector, $\mathbf{v} \in \mathbb{F}^n$. Storage is allocated but the entries of $\mathbf{v}$ are not initialized.

```
VECTOR(const initializer_list<SCALAR>& list)
VECTOR(int n, const initializer_list<ivpair>& list)
```
: Initialize $\mathbf{v}$ from the initializer list `list` (see `example1.cpp`).

NAMED CONSTRUCTORS

`VECTOR::Zero(int n)`
: The n dimensional zero vector, $\mathbf{0} \in \mathbb{F}^n$.

`VECTOR::Filled(int n, SCALAR t)`
: An n dimensional vector in which each element is initialized to `t`.

```
VECTOR::Uniform(int n)
VECTOR::Gaussian(int n)
VECTOR::Bernoulli(int n, double p=0.5)
```
: An n dimensional random vector in which each element is drawn from (a) the Uniform$(0, 1)$ distribution; (b) the Normal$(0, 1)$ distribution; (c) the Bernoulli(p) distribution.

ATTRIBUTES

`bool isSparseFormat() const`
: Return `true` if `VECTOR` is a sparse vector class.

ELEMENT ACCESS

```
SCALAR read(int i) const
SCALAR operator()(int i) const
```
: Return the value of the matrix element v_i.

`void set(int i, SCALAR x)`
Set $v_i = x$.

`SCALAR& operator()(int i)`
Return a reference to v_i. For most vector types `v(i)=x` can be used as a simpler alternative to `v.set(i,x)`.

`bool isFilled(int i) const`
Return `true` if v_i is filled in. For dense vectors always `true`.

`int nFilled() const`
The number of filled in elements of `v`.

ITERATORS

`void for_each(std::function<void(INDEX,SCALAR&)> lambda)`
`void for_each(std::function<void(INDEX,const SCALAR)> lambda) const`
Apply the function `lambda` to each element of **v**. The two arguments of `lambda` are i and (a reference to) v_i.

`void for_each_filled(std::function<void(INDEX,SCALAR&)> lambda)`
`void for_each_filled(std::function<void(INDEX,const SCALAR)> lambda) const`
Apply the function `lambda` to each filled in element of **v**. The two arguments of `lambda` are i and (a reference to) v_i.

VIEWS

`VectorView<VECTOR> operator()(const IndexMap& phi)`
`const VectorView<VECTOR> operator()(const IndexMap& phi) const`
Given an index map ϕ: $\{0, 1, 2, \ldots, n_s - 1\} \mapsto \{0, 1, 2, \ldots, m - 1\}$, return a view of the subvector $(v_{\phi(1)}, v_{\phi(2)}, \ldots, v_{\phi(n_s-1)})^\top$.

CONVERSIONS

`VECTOR(const Cvector v)`
Covert the `Cvector` **v** to a `VECTOR`.

`operator Cvector() const`
Convert **v** to a `Cvector` (this method is repeated in the description of the `Cvector` class).

`VECTOR(const BlockedVector<VECTOR>& w)`
Convert the blocked vector `w` to plain (non-blocked) vector format.

REMAPPINGS

`VECTOR remap(const IndexMap& map) const`
`VECTOR remap(const Inverse<IndexMap>& map) const`
Remap the elements of **v** by `map` or the inverse of `map`.

IN-PLACE ARITHMETIC

`VECTOR& operator+=(SCALAR c)`
`VECTOR& operator-=(SCALAR c)`
`VECTOR& operator*=(SCALAR c)`
`VECTOR& operator/=(SCALAR c)`
Increment/decrement/multiply/divide each element of $\mathbf{v}$ by c.

`VECTOR& operator+=(const VECTOR2& x)`
`VECTOR& operator-=(const VECTOR2& x)`
`VECTOR& operator*=(const VECTOR2& x)`
`VECTOR& operator/=(const VECTOR2& x)`
Increment/decrement/multiply/divide each element of $\mathbf{v}$ by the corresponding element of $\mathbf{x}$.

`VECTOR& apply(const OPERATOR& Q)`
`VECTOR& applyT(const OPERATOR& Q)`
Apply the operator `Q` or the transpose of `Q` to `v` in place.

SCALAR VALUED ARITHMETIC

`SCALAR dot(const VECTOR& x) const`
Return the dot product of $\mathbf{v}$ with $\mathbf{x}$.

VECTOR VALUED ARITHMETIC

`VECTOR mult(SCALAR c) const`
`VECTOR plus(const VECTOR& x) const`
`VECTOR minus(const VECTOR& x) const`
Compute $c\mathbf{v}$, $\mathbf{v}+\mathbf{x}$ or $\mathbf{v}-\mathbf{x}$.

`VECTOR mult(const OPERATOR& Q) const`
`VECTOR multT(const OPERATOR& Q) const`
Apply the operator `Q` or the transpose of `Q` to `v` and return the result.

`VECTOR operator*( SCALAR c) const`
`VECTOR operator+(const VECTOR& x) const`
`VECTOR operator-(const VECTOR& x) const`
Synonyms of the `mult`, `plus` and `minus` methods.

`VECTOR& add(const VECTOR& x)`
`VECTOR& add(const VECTOR& x, SCALAR c)`
Compute $\mathbf{v}+\mathbf{x}$ or $\mathbf{v}+c\mathbf{x}$. These methods are performance critical, because they are used e.g., when applying Givens rotations from the right to `MatrixX<VECTOR>` matrices.

SCALAR METHODS

`SCALAR max() const`
`SCALAR max_abs() const`
`int argmax() const`
`int argmax_abs() const`

(a-b) The value of the largest (resp. largest in absolute value) element of $\mathbf{v}$. (c-d) he index of the largest (resp. largest in absolute value) element of $\mathbf{v}$. If not unique, then the index of the first (lowest index) maximal element is returned.

`SCALAR min() const`
`SCALAR min_abs() const`
`int argmin() const`
`int argmin_abs() const`
The same as above, but for the least (in absolute value) element.

`SCALAR sum() const`
The sum of the vector elements, $\sum_{i=1}^{n} v_i$.

`SCALAR norm1() const`
The ℓ_1 norm of $\mathbf{v}$, $\|\mathbf{v}\|_1 = \sum_{i=1}^{n} |\, v_i \,|$.

`SCALAR norm2() const`
The squared ℓ_2–norm of $\mathbf{v}$, $\|\mathbf{v}\|^2$.

`SCALAR diff2(const VECTOR& x) const`
The squared ℓ_2–norm difference $\|\mathbf{v}-\mathbf{x}\|^2$.

`int nnz() const`
The number of non-zero elements of $\mathbf{v}$. Different from `nFilled` in that it does not count zero-valued, but filled in elements.

FORMATTED I/O

`VECTOR(MatrixIF& file)`
Load $\mathbf{v}$ from the file `file` (see the section on matrix filetypes).

`saveto(MatrixOF& file) const`
Save $\mathbf{v}$ to the file `file` (see the section on matrix filetypes).

PYTHON INTERFACE

`VECTOR(<array>)`
Initialize $\mathbf{v}$ from the `numpy` array `<array>`.

`np()`
Return $\mathbf{v}$ in the form of a `numpy` array.

VARIABLES

`int n`
The dimensionality of the vector, n.

Cvector

`Cvector` is Mondrian's most basic dense vector class, which simply stores **v** in a C-style array `SCALAR[n]`. `Cvector` provides all the functionality defined in `Vector`, plus the following additional methods.

Derived from: `Vector, Detachable, Serializable`

CONVERSIONS

`Cvector(const VECTOR& v)`
: Convert `v` to `Cvector` format. While this method looks like a `Cvector` constructor, it is implemented as a conversion operator `VECTOR::operator Cvector()` and defined in the source class `VECTOR` rather than in `Cvector` (see the documentation for the abstract class `Vector`).

VIEWS

`Detached<Cvector> subvectorView(const int i, const int n)`
`const Detached<Cvector> subvectorView(const int i, const int n) const`
: Return a view of the subvector $(v_i, v_{i+1}, \ldots, v_{i+n-1})^\top$.

VARIABLES

`SCALAR* array`
: The array of vector elements.

Vectorv, Vectorl, Vectorh

`Vectorv`, `Vectorl` and `Vectorh` are sparse vector classes directly built atop Standard Template Library containers:

1. `Vectorv` stores $\boldsymbol{v}$ as a `std::vector` of `IndexValuePair` objects,
2. `Vectorl` stores $\boldsymbol{v}$ as a `std::vector` of `IndexValuePair` objects,
3. `Vectorh` stores $\boldsymbol{v}$ as a hash map `std::unordered_map<INDEX,SCALAR>` that maps indices to the corresponding matrix elements.

All three classes inherit the generic `Vector` interface.

Derived from: `SparseVector, (Vector), (Serializable),` {`std::vector<IndexValuePair>` or `std::list<IndexValuePair>` or `std::unordered_map<INDEX,SCALAR>`}

GenericVector

`GenericVector` is Mondrian's polymorphic vector class which can represent a vector of any class derived from `Vector`. `GenericVector` is implemented as a wrapper: the encapsulated concrete vector is pointed to by `obj` pointer. In contrast to the deafult way that polymorphism is implemented in C++, function calls to a `GenericVector` are delegated to the actual class of `*obj` rather than being handled by the base class.

Derived from: `Vector, (Detachable), (Serializable)`
Owned objects: The concrete vector object `*obj`.

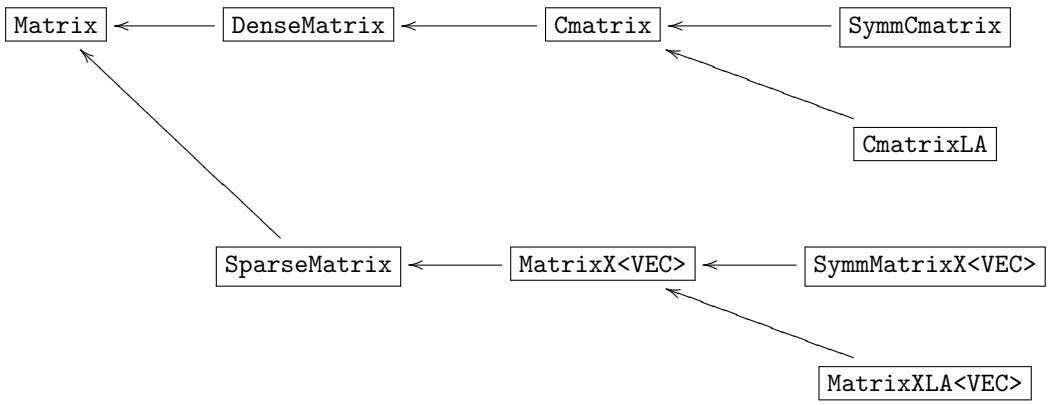
VARIABLES

`Vector* obj`
: Pointer to the concrete vector object.

2. Basic matrices

Similarly to the vector classes, the basic division between Mondrian's matrix classes is between the dense and sparse cases. While this kept opaque to the user, to optimize performance, matrix/matrix and matrix/vector operations are generally separately implemented for all combinations of operands. For example, the expression `A*v` where `A` is a matrix and `v` is a vector will be be evaluated using different methods based on whether `A` is of type `Cmatrix` or `MatrixX<VECTOR>` (for some vector type `VECTOR`) and whether `v` is of type `Cvector`, `Vectorv`, `Vectorl` or `Vectorh`. However, similarly to the vector types, all matrix classes conform to the same API given in `Matrix`. `GenericMatrix` is the universal polymorphic matrix class.

Classes prefaced by `Symm` are for storing matrices that are known to be symmetric. In general, these classes do not use less storage, but certain operations on them are faster. The "`LA`" classes (standing for "linear algebra") provide methods for higher level linear algebra operations, such as eigendecomposition, by calling an appropriate external library, which, in the present version of the code is `Eigen`.



Matrix

`Matrix` defines the common interface to all matrix classes in Mondrian. Every concrete matrix class `MATRIX` provides the following methods and constructors, either by inheritance or separate implementation.

CONSTRUCTORS

`MATRIX()`
: Create a new 0×0 matrix.

`MATRIX(int n, int m)`
: Create a new $n \times m$ dimensional matrix $A \in \mathbb{F}^{n \times m}$. Storage is allocated but the entries of A are not initialized. To ensure that matrix elements default to 0, use the `Zero` constructor, below.

`MATRIX(const initializer_list<Cvector>& list)`
`MATRIX(const intializer_list<iivtriple>& list)`
: Initialize A from an initializer list of `Cvector` objects or `(INDEX,INDEX,SCALAR)` triples (see `example1.cpp` for an example of usage).

NAMED CONSTRUCTORS

`MATRIX::Zero(int n, int m)`
: The $n \times m$ dimensional zero matrix.

`MATRIX::Identity(int n)`
: The $n \times n$ dimensional identity matrix.

`MATRIX::Filled(int n, int m, SCALAR t)`
: The constant matrix $A \in \mathbb{F}^{n \times m}$ in which $A_{i,j} = t$ for all i and j.

`MATRIX::Diagonal(const Cvector& v)`
: The diagonal matrix in which $A_{i,i} = v_i$.

`MATRIX::Uniform(int n, int m)`
`MATRIX::Gaussian(int n, int m)`
`MATRIX::Bernoulli(int n, int m, double p=0.5)`
: An $n \times m$ random matrix in which each element is drawn i.i.d. from (a) the Uniform$(0, 1)$ distribution; (b) the Normal$(0, 1)$ distribution; (c) the Bernoulli(p) distribution. In symmetric matrix classes such as `SymmCmatrix` and `SymmMatrixX<VECTOR>`, the upper triangle of the matrix is mirrored into the lower triangle to ensure symmetry.

ATTRIBUTES

`bool isSparseFormat() const`
: Return `true` if `MATRIX` is a sparse matrix class.

`bool isSymmetricFormat() const`
: Return `true` if `MATRIX` is a symmetric matrix class.

ELEMENT ACCESS

`SCALAR read(const int i, const int j) const`
`SCALAR operator()(const int i, const int j) const`
Return the value of the matrix element $A_{i,j}$.

`SCALAR& operator()(const int i, const int j)`
Return a reference to the matrix element $A_{i,j}$.

`SCALAR set(const int i, const int j, const SCALAR v)`
Set $A_{i,j} = v$. In most cases this is equivalent to `A(i,j)=v` except that when there are restrictions on A (i.e., that it must be symmetric), `SCALAR& operator()` might be disabled.

`bool isFilled(const int i, const int j) const`
Returns `true` if element `(i,j)` is filled in. For dense matrices always `true`.

`int nFilled() const`
The number of filled in elements of `A`.

SUBVECTORS

`VECTOR row<VECTOR>(const int i) const`
Return the i'th row of A.

`VECTOR column<VECTOR>(const int j) const`
Return the j'th column of A.

`VECTOR diag<VECTOR>() const`
Return the vector of diagonal elements of A.

ITERATORS

`void for_each_filled(std::function<void(INDEX,INDEX,SCALAR&)> lambda)`
`void for_each_filled(std::function<void(INDEX,INDEX,const SCALAR)> lambda) const`
Apply the function `lambda` to each filled in entry $A_{i,j}$ of `A`. The three arguments of `lambda` are i, j and (a reference to) $A_{i,j}$.

`void for_each_filled_in_row(const int i, std::function<void(INDEX,SCALAR&)> lambda)`
`void for_each_filled_in_row(const int i, std::function<void(INDEX,const SCALAR)> lambda) const`
Apply the function `lambda` to each filled in entry $A_{i,j}$ in row i of `A`. The two arguments of `lambda` are j and (a reference to) $A_{i,j}$.

`void for_each_filled_in_column(const int j, std::function<void(INDEX,SCALAR&)> lambda)`
`void for_each_filled_in_column(const int j, std::function<void(INDEX,const SCALAR)> lambda) const`
Apply the function `lambda` to each filled in entry $A_{i,j}$ in column j of `A`. The two arguments of `lambda` are i and (a reference to) $A_{i,j}$.

VIEWS

`MatrixView<MATRIX> operator()(const IndexMap& rmap, const IndexMap& cmap)`
`const MatrixView<MATRIX> operator()(const IndexMap& rmap, const IndexMap& cmap) const`
Return a `MatrixView` referencing the submatrix of A cut out by rows `rmap` and columns `cmap`.

CONVERSIONS

`MATRIX(const Cmatrix X)`
Covert the `Cmatrix` X to a `MATRIX`.

`operator Cmatrix() const`
Convert M to a `Cmatrix` (this method is repeated in the description of the `Cmatrix` class).

`MATRIX(const BlockedMatrix<MATRIX>& B)`
Convert the blocked matrix `B` to plain (non-blocked) format.

REMAPPINGS

`MATRIX2 remapRows(const IndexMap& map) const`
`MATRIX2 remapRows(const Inverse<IndexMap>& map) const`
Remap the rows of M by `map` / the inverse of `map`. If `MATRIX` is a non-symmetric matrix type, `MATRIX2=MATRIX`. For symmetric matrix types, `MATRIX2` is the non-symmetric version of `MATRIX`. For example, if `MATRIX` is `SymmCmatrix`, then `MATRIX2` will be `Cmatrix`.

`MATRIX2 remapCols(const IndexMap& map) const`
`MATRIX2 remapCols(const Inverse<IndexMap>& map) const`
Remap the columns of M by `map` / the inverse of `map`. See the comment about types above.

`MATRIX2 remap(const IndexMap& rmap, const IndexMap& cmap) const`
`MATRIX2 remap(const Inverse<IndexMap>& rmap, const Inverse<IndexMap>& cmap) const`
Remap the rows and columns of M by (the inverse of) `rmap` and `cmap`. See the comment about types above.

IN-PLACE ARITHMETIC

`MATRIX& operator+=(const SCALAR c)`
`MATRIX& operator-=(const SCALAR c)`
`MATRIX& operator*=(const SCALAR c)`
`MATRIX& operator/=(const SCALAR c)`
Increment/decrement/multiply/divide each element of A by c.

`MATRIX& operator+=(const MATRIX2& B)`
`MATRIX& operator-=(const MATRIX2& B)`
`MATRIX& operator*=(const MATRIX2& B)`
`MATRIX& operator/=(const MATRIX2& B)`
Increment/decrement/multiply/divide each element of A by the corresponding element of B.

IN-PLACE OPERATIONS

`MATRIX& symmetrize()`
Set A to $(A+A^\top)/2$.

`MATRIX& normalizeRows()`
`MATRIX& normalizeColumns()`
Normalize each row/column of A so that it sums to 1.

`MATRIX& multiplyRowsBy(const VECTOR& v)`
`MATRIX& divideRowsBy(const VECTOR& v)`
Multiply/divide the i'th row of A by v_i.

`MATRIX& multiplyColumnsBy(const VECTOR& v)`
`MATRIX& divideColumnsBy(const VECTOR& v)`
Multiply/divide the j'th column of A by v_j.

`MATRIX& applyFromLeft(const OPERATOR& Q)`
`MATRIX& applyFromLeftT(const OPERATOR& Q)`
Apply the operator `Q` or the transpose of `Q` from the left in place. In matrix notation, $A \leftarrow QA$ or $A \leftarrow Q^\top A$.

`MATRIX& applyFromRight(const OPERATOR& Q)`
`MATRIX& applyFromRightT(const OPERATOR& Q)`
Apply the operator `Q` or the transpose of `Q` to `v` from the right in place. In matrix notation, $A \leftarrow AQ^\top$ or $A \leftarrow AQ$. Note that applying the operator from the right corresponds to multiplication by $Q^\top$.

`MATRIX& conjugateBy(const OPERATOR& Q)`
`MATRIX& conjugateByT(const OPERATOR& Q)`
Conjugate A by (the transpose of) Q in place. In matrix notation, $A \leftarrow QAQ^\top$ or $A \leftarrow Q^\top AQ$.

VECTOR VALUED ARITHMETIC

`VECTOR& mult(const VECTOR& v) const`
`VECTOR& dot(const VECTOR& v) const`
Compute the matrix/vector products $A\mathbf{v}$ and $A^\top\mathbf{v}$, respectively. Here `VECTOR` can be any concrete vector class, such as a `Cvector`, `Vectorv`, etc..

`VECTOR& operator*(const VECTOR& v) const`
Synonym of `mult`.

MATRIX VALUED ARITHMETIC

`MATRIX mult(const SCALAR c) const`
Compute cA.

`MATRIX plus(const MATRIX& B) const`
`MATRIX minus(const MATRIX& B) const`
Compute $A+B$ or $A-B$.

`MATRIX mult(const MATRIX& B) const`
`MATRIX dot(const MATRIX& B) const`
`MATRIX outer(const MATRIX& B) const`
`MATRIX dott(const MATRIX& B) const`
Compute the matrix/matrix product (a) AB, (b) $A^\top B$, and (c) $AB^\top$ and (d) $A^\top B^\top$.

`MATRIX operator*(const SCALAR c) const`
`MATRIX operator+(const MATRIX& B) const`
`MATRIX operator-(const MATRIX& B) const`
`MATRIX operator*(const MATRIX& B) const`
Synonyms of `plus`, `minus` and `mult` methods.

`MATRIX rowGram() const`
`MATRIX colGram() const`
Compute the symmetric matrix G defined (a) $G_{i,j} = A_{i,*} \cdot A_{j,*}$ (b) $G_{i,j} = A_{*,i} \cdot A_{*,j}$.

SCALAR VALUED METHODS

`int nnz() const`
The number of non-zero matrix elements of A.

`SCALAR norm2() const`
The squared Frobenius norm of the matrix, $\|A\|^2_{\text{Frob}}$.

`SCALAR diff2(const MATRIX& X) const`
The squared Frobenius norm difference between A and X, i.e., $\|A-X\|^2_{\text{Frob}}$.

FORMATTED I/O

`MATRIX(MatrixIF& file)`
Load A from the file `file` (see the section on matrix filetypes).

`saveto(MatrixOF& file) const`
Save A to the file `file` (see the section on matrix filetypes).

PYTHON INTERFACE

`MATRIX(<array>)`
Initialize A from the `numpy` array `<array>`.

`np()`
Return A in the form of a `numpy` array.

VARIABLES

`int nrows`
The number of rows, n.

`int ncols`
The number of columns, m.

Cmatrix

`Cmatrix` is the simplest concrete matrix class, which simply stores A in the form of a C-style array `SCALAR[nrows*ncols]` in column major order. This class provides all the standard matrix methods described in `Matrix`, plus the following additional methods.

Derived from: `Matrix, Detachable, Serializable`

CONVERSIONS

`Cmatrix(const MATRIX& X)`
: Convert `X` to `Cmatrix` format. While syntactically this method looks like a `Cmatrix` constructor, it is actually a conversion operator of type `MATRIX::operator Cmatrix()`, defined in the source class.

VIEWS

`Detached<Cvector> viewOfColumn(const int j)`
`const Detached<Cvector> viewOfColumn(const int j)const`
: Return a view of the j'th column of A in the form of a detached `Cvector`.

`Detached<Cmatrix> viewOfColumns(const int j, const int k)`
`const Detached<Cmatrix> viewOfColumns(const int j, const int k) const`
: Return a view of columns $j, j+1, \ldots, j+k-1$ in the form of a detached `Cmatrix`.

VECTOR VALUED METHODS

`Cvector vectorize() const`
: Reshape A into an nm dimensional vector.

MATRIX VALUED METHODS

`Cmatrix reshape(const int nnew, const int mnew) const`
: Reshape A into an $n_{\text{new}} \times m_{\text{new}}$ dimensional matrix.

VARIABLES

`SCALAR* array`
: An $n \times m$ column major array holding the matrix elements of A.

MatrixX<VECTOR>

MatrixX<VECTOR> is Mondrian's basic sparse matrix class, which represents $A \in \mathbb{F}^{n \times m}$ as m VECTORs, where VECTOR can be any (dense or sparse) vector class. MatrixX<VECTOR> implements all the methods in Matrix.

Derived from: Matrix, Detachable, Serializable
Owned objects: The VECTOR objects storing each column of the matrix.

VIEWS

```
Detached<VECTOR> viewOfColumn(const int j)
const Detached<VECTOR> viewOfColumn(const int j) const
```
Return a view of the j'th column of A in the form of a detached VECTOR.

VARIABLES

```
vector<VECTOR*> column
```
Pointers to the VECTOR objects storing the individual columns of A.

GenericMatrix

`GenericMatrix` is Mondrian's polymorphic matrix class, which can represent a matrix of any class. The class is implemented as a wrapper: the encapsulated concrete matrix is pointed to by the `obj` pointer. In contrast to the deafult way that polymorphism is implemented in C++, function calls to a `GenericMatrix` are delegated to the actual class of `*obj` rather than being handled by the base class.

Derived from: `Matrix, (Detachable), (Serializable)`
Owned objects: The concrete vector object `*obj`.

VARIABLES

`Vector* obj`
: Pointer to the concrete vector object.

3. Specialized matrices

SymmCmatrix

`SymmCmatrix` is the symmetric variant of the `Cmatrix` class. While the two classes store A in exactly the same format (which makes conversions between them easy) the fact that the matrix stored in `SymmCmatrix` is known to be symmetric allows `SymmCmatrix` to perform certain operations much faster.

Derived from: `Cmatrix, (Matrix), (Detachable), (Serializable)`

NAMED CONSTRUCTORS

`ColumnGram(const MATRIX& B)`
`RowGram(const MATRIX& B)`

Initialize A to be the column Gram matrix $B^\top B$ or the row Gram matrix $BB^\top$. Computing Gram matrices is often a time critical operation, so these constructors are implemented separately for various different `MATRIX` classes.

CONVERSIONS

`SymmCmatrix(const Cmatrix& M)`
`SymmCmatrix(Cmatrix&& M)`

Construct A by symmetrizing M, i.e $A = (M + M^\top)/2$. Move construction is possible because `SymmCmatrix` and `Cmatrix` use the same storage format.

`SymmCmatrix(assume_symmetric(const Cmatrix& M))`
`SymmCmatrix(assume_symmetric(Cmatrix&& M))`

The same as above, except without explicit symmetrization, because M is assumed to be already symmetric. The second of these methods effectively just changes the data type of M without any computations, and therefore incurs almost no computational overhead.

SymmMatrixX<VECTOR>

SymmMatrixX<VECTOR> is the symmetric variant of the MatrixX<VECTOR> class. While the the two classes store the elements of A in exactly the same format (which makes conversions between them easy) the fact that the matrix stored in SymmMatrixX<VECTOR> is known to be symmetric allows SymmMatrixX<VECTOR> to perform certain operations much faster.

Derived from: MatrixX<VECTOR>, (Matrix), (Detachable), (Serializable)

NAMED CONSTRUCTORS

ColumnGram(const MATRIX& B)
RowGram(const MATRIX& B)

Initialize A to be the column Gram matrix $B^\top B$ or the row Gram matrix $BB^\top$. Computing Gram matrices is often a time critical operation, so these constructors are implemented separately for various different MATRIX classes.

CONVERSIONS

SymmMatrixX<VECTOR>(const MatrixX<VECTOR>& M)
SymmMatrixX<VECTOR>(MatrixX<VECTOR>&& M)

Construct A by symmetrizing M, i.e $A = (M + M^\top)/2$. Move construction is possible because SymmMatrixX<VECTOR> and MatrixX<VECTOR> use the same storage format.

SymmMatrixX<VECTOR>(as_symmetric(const MatrixX<VECTOR>& M))
SymmMatrixX<VECTOR>(as_symmetric(MatrixX<VECTOR>&& M))

The same as above, except without explicit symmetrization, because M is assumed to be already symmetric. The second of these methods effectively just changes the data type of M without any computations, and therefore incurs almost no computational overhead (see the section on downcasting).

CmatrixLA

`CmatrixLA` is an extension of the `Cmatrix` class that includes certain linear algebra operations, such as eigendecomposition, that are performed by calling routines from outside linear algebra pacckages, in the current implementation of the library, `Eigen`.

`CmatrixLA` is separated from `Cmatrix` to keep the library modular and simplify compilation. In particular, `Cmatrix` is a "header only" class, therefore it can be directly `#include`-ed in user code with no separate compilation. Any functions in `Cmatrix` that use `Eigen` would greatly slow down compiling user code, because they would "pull in" a large number of header files from `Eigen`. Segregating these functions in `CmatrixLA` solves this problem because `CmatrixLA` is a compiled class.

Derived from: `Cmatrix, (Matrix), (Detachable), (Serializable)`

OPERATIONS

`package<CmatrixLA,Cvector> symmetricEVD() const`

Returns a matrix whose rows are the eigenvectors of A and a vector holding the corresponding eigenvalues. It is assumed that A is symmetric.

4. Blocked vectors/matrices

BlockedVector<VECTOR>

A `BlockedVector` **v** consists of n_b blocks, where each block is a vector of type `VECTOR`. Here `VECTOR` can be any dense or sparse matrix type derived from the abstract class `Vector`.

Derived from: `Vector, Detachable, Serializable`

CONSTRUCTORS

`BlockedVector<VECTOR>(const int nb)`
: Create a blocked vector consisting of `nb` zero dimensional blocks constructed using the `VECTOR` class's default constructor `VECTOR()`.

`BlockedVector<VECTOR>(const BlockStructure& st)`
: Create a blocked vector with block structure `st`=$(b_1, \ldots, b_p)$. The i'th block is constructed using the `VECTOR` class's `VECTOR(n)` constructor with $n = b_i$.

`BlockedVector<VECTOR>(VECTOR&& w)`
`BlockedVector<VECTOR>(const VECTOR& w)`
: Create a blocked vector with a single block, `w`. The first method destroys the original vector.

`BlockedVector<VECTOR>(const BlockStructure& st, const VECTOR& w)`
: Convert the vector `w` to blocked vector format with block structure `st`.

`BlockedVector<VECTOR>(vector<VECTOR>&& vlist)`
`BlockedVector<VECTOR>(const vector<const VECTOR>& vlist)`
: Create a blocked vector composed of the elements of `vlist`. The first of these constructors destroys the original vectors.

`BlockedVector<VECTOR>(vector<BlockedVector<VECTOR>>&& vlist)`
`BlockedVector<VECTOR>(const vector<const BlockedVector<VECTOR>>& vlist)`
: Create a blocked vector composed of each of the blocks of the blocked vectors in `vlist`. The first of these constructors destroys the original blocked vectors.

NAMED CONSTRUCTORS

`BlockedVector<VECTOR>::Zero(BlockStructure& bstruct)`
`BlockedVector<VECTOR>::Filled(BlockStructure& bstruct, const SCALAR t)`
`BlockedVector<VECTOR>::Uniform(BlockStructure& bstruct)`
`BlockedVector<VECTOR>::Gaussian(BlockStructure& bstruct)`
`BlockedVector<VECTOR>::Bernoulli(BlockStructure& bstruct, const double p=0.5)`
: Create a `bstruct`–structured blocked vector with block structure `bstruct`, composed of blocks constructed with the appropriate constructors of `VECTOR`.

BLOCK LEVEL ACCESS

`VECTOR& block(const int i)`
`VECTOR block(const int i) const`
Return (a reference to) the i'th block of $\mathbf{v}$.

`VECTOR* block_ptr(const int i)`
`const VECTOR* block_ptr(const int i) const`
Return a pointer to the i'th block of $\mathbf{v}$.

`Detached<BlockedVector<VECTOR>> blocks(const IndexMap& phi)`
`const Detached<BlockedVector<VECTOR>> blocks(const IndexMap& phi) const`
Return a detached blocked vector of blocks $([\![\mathbf{v}]\!]_{\phi(0)}, \ldots, [\![\mathbf{v}]\!]_{\phi(k-1)})$

ELEMENT ACCESS

`SCALAR& operator()(const int i)`
`SCALAR operator()(const int i) const`
`SCALAR& operator()(const int I, const int i)`
`SCALAR operator()(const int I, const int i) const`
`SCALAR& operator()(const IndexPair& ip)`
`SCALAR operator()(const IndexPair& ip) const`
Return (a reference to) v_i or $v_{(I,i)}$. In the case of sparse vectors, a potential side effect of returning a reference is that it might fill in $v_{(I,i)}$, even if the intent is only to read it. To avoid this behavior, use the `read` method.

`SCALAR read(const int i) const`
`SCALAR read(const int I, const int i) const`
`SCALAR read(const IndexPair& ip) const`
Return the value of v_i or $v_{(I,i)}$, guaranteeing side effect free behavior, even when `v` is not `const`.

`bool isFilled(const int i) const`
`bool isFilled(const int I, const int i) const`
`bool isFilled(const IndexPair& ip) const`
Return `true` if v_i or $v_{(I,i)}$ is filled in. For dense vectors always `true`.

`int nFilled() const`
The number of filled in elements of `v`.

ITERATORS

`void for_each_block(std::function<void(INDEX,VECTOR&)> lambda)`
`void for_each_block(std::function<void(INDEX,const VECTOR&)> lambda) const`
Apply the function `lambda` to each block of $\mathbf{v}$. The two arguments are i and a reference to $[\![\mathbf{v}]\!]_i$.

VIEWS

`BlockedVectorView<VECTOR> operator()(const BindexMap& map)`
`const BlockedVectorView<VECTOR> operator()(const BindexMap& map) const`
Return a view of the subvector $(v_{\phi(0)}, v_{\phi(1)}, \ldots, v_{\phi(k-1)})^\top$.

CONVERSIONS

`BlockedVector<VECTOR>(const BlockedVector<Cvector> w)`
Covert `w` to a `BlockedVector<VECTOR>` type vector.

MERGING

`void merge(VECTOR&& w)`
`void merge(const VECTOR& w)`
Add `w` to `v` as its last block. The first method destroys `w`.

`void merge(BlockedVector<VECTOR>&& w)`
`void merge(const BlockedVector<VECTOR>& w)`
Merge `w` to `v` by adding its blocks to the end of `v`. The first method destroys `w`.

REMAPPINGS

`VECTOR remap(const BindexMap& map) const`
`VECTOR remap(const Inverse<BindexMap>& map) const`
Remap the elements of v by `map` or the inverse of `map`.

IN-PLACE ARITHMETIC

`BlockedVector<VECTOR>& operator+=(const SCALAR c)`
`BlockedVector<VECTOR>& operator-=(const SCALAR c)`
`BlockedVector<VECTOR>& operator*=(const SCALAR c)`
`BlockedVector<VECTOR>& operator/=(const SCALAR c)`
Increment/decrement/multiply/divide each element of $\mathbf{v}$ by c.

`BlockedVector<VECTOR>& operator+=(const BlockedVector<VECTOR2>& x)`
`BlockedVector<VECTOR>& operator-=(const BlockedVector<VECTOR2>& x)`
Set $\mathbf{v}$ to $\mathbf{v}+\mathbf{x}$ or $\mathbf{v}-\mathbf{x}$, respectively.

SCALAR VALUED ARITHMETIC

`SCALAR dot(const BlockedVector<VECTOR2>& x) const`
The dot product of $\mathbf{v}$ with $\mathbf{x}$. $\mathbf{v}$ and $\mathbf{x}$ must have the same block structure.

VECTOR VALUED ARITHMETIC

`BlockedVector<VECTOR> mult(const SCALAR c) const`
`BlockedVector<VECTOR> plus(const BlockedVector<VECTOR2>& x) const`
`BlockedVector<VECTOR> minus(const BlockedVector<VECTOR2>& x) const`
Compute $c\mathbf{v}$, $\mathbf{v}+\mathbf{x}$ and $\mathbf{v}-\mathbf{x}$, respectively.

`BlockedVector<VECTOR> operator*(const SCALAR c) const`
`BlockedVector<VECTOR> operator+(const BlockedVector<VECTOR2>& x) const`
`BlockedVector<VECTOR> operator-(const BlockedVector<VECTOR2>& x) const`
Synonyms of `mult`, `plus` and `minus`.

MATRIX VALUED ARITHMETIC

`BlockedMatrix<MATRIX> outer<MATRIX>(const BlockedVector<VECTOR>& x) const`
Compute the outer product $\mathbf{v}\,\mathbf{x}^\top$.

OTHER METHODS

`BlockStructure structure() const`
Return the block structure of `v`.

FORMATTED I/O

`BlockedVector<VECTOR>(const BlockStructure& st, const MatrixIF& file)`
Load $\mathbf{v}$ from the file `file` and block it according to `st`.

`saveto(MatrixOF& file)`
Save $\mathbf{v}$ to the file `file`.

VARIABLES

`int n`
The total dimensionality of the vector (inherited from `Vector`).

`int nb`
The number of blocks.

`VECTOR** blockp`
An array of pointers to the individual blocks of $\mathbf{v}$.

BlockedMatrix<MATRIX>

A `BlockedMatrix` M consists of $n_b \times m_b$ blocks, where each block is a matrix of type `MATRIX`. Here `MATRIX` can be any dense or sparse matrix type derived from the abstract class `Matrix`. The i'th block of rows we sometimes call the i'th **street**, and the j'th column of blocks the j'th **tower**.

Derived from: `Matrix, Detachable, Serializable`
Owned objects: The individual blocks pointed to by the elements of the array `blocks`.

CONSTRUCTORS

`BlockedMatrix<MATRIX>(const int nb, const int mb)`
Construct a blocked matrix consisting of `nb`×`mb` zero dimensional blocks constructed using the `MATRIX` class's `MATRIX()` constructor.

`BlockedMatrix<MATRIX>(const Bstructure& rst, const Bstructure& cst)`
Construct a blocked matrix that is blocked horizontally according to `rst` and vertically according to `cst`. The individual blocks are constructured using the `MATRIX` class's `MATRIX(n,m)` constructor.

`BlockedMatrix<MATRIX>(MATRIX&& B)`
`BlockedMatrix<MATRIX>(const MATRIX& B)`
Create a blocked matrix with a single block, `B`. The first method destroys the original matrix.

`BlockedMatrix<MATRIX>(const Bstructure& rst, const Bstructure& cst, const MATRIX& B)`
Convert `B` to blocked matrix format with block structure given by `rst` and `cst`.

NAMED CONSTRUCTORS

`BlockedMatrix<MATRIX>::Zero(const BlockStructure& rst, const BlockStructure& cst)`
`BlockedMatrix<MATRIX>::Filled(const BlockStructure& rst, const BlockStructure& cst)`
`BlockedMatrix<MATRIX>::RandomUniform(const BlockStructure& rst, const BlockStructure& cst)`
`BlockedMatrix<MATRIX>::RandomGaussian(const BlockStructure& rst, const BlockStructure& cst)`
`BlockedMatrix<MATRIX>::RandomBernoulli(const BlockStructure& rst, const BlockStructure& cst, const double p=0)`
Create an `rst`×`cst` structured blocked matrix whose individual blocks are constructed with the appropriate constructor of `MATRIX`.

`BlockedMatrix<MATRIX>::Identity(const BlockStructure& st)`
Construct the `st`×`st` structured identity matrix.

`BlockedMatrix<MATRIX>::Diagonal(const BlockedVector& v)`
Construct a diagonal blocked matrix whose diagonal elements are given by **v** and whose block structure is dictated by the block structure of `v`.

`BlockedMatrix<MATRIX>::ConcatVertically(vector<BlockedMatrix<MATRIX>&& blist>)`
`BlockedMatrix<MATRIX>::ConcatVertically(const vector<const BlockedMatrix<MATRIX>& blist>)`
Construct a blocked matrix by concatenating the elements of `blist` vertically. The first method destroys the elements of `blist`.

`BlockedMatrix<MATRIX>::ConcatHorizontally(vector<BlockedMatrix<MATRIX>&& blist>)`
`BlockedMatrix<MATRIX>::ConcatHorizontally(const vector<const BlockedMatrix<MATRIX>& blist>)`
Construct a blocked matrix by concatenating the elements of `blist` horizontally. The first method destroys the elements of `blist`.

BLOCK LEVEL ACCESS

`MATRIX& block(const int i, const int j)`
`MATRIX block(const int i, const int j) const`
Return (a reference to) the (i, j) block of M.

`MATRIX* block_ptr(const int i, const int j)`
`const MATRIX* block_ptr(const int i, const int j) const`
Return a pointer to the (i, j) block of M.

`Detached<BlockedMatrix<MATRIX>> blocks(const IndexMap& phi, const IndexMap& psi)`
`const Detached<BlockedMatrix<MATRIX>> blocks(const IndexMap& phi, const IndexMap& psi) const`
Return a detached blocked matrix B in which $[\![B]\!]_{i,j} = [\![M]\!]_{\phi(i),\psi(j)}$.

ELEMENT ACCESS

`SCALAR& operator()(const int i, const int j)`
`SCALAR operator()(const int i, const int j) const`
`SCALAR& operator()(const int I, const int i, const int J, const int j)`
`SCALAR operator()(const int I, const int i, const int J, const int j) const`
`SCALAR& operator()(const IndexPair& ip, const IndexPair& jp)`
`SCALAR operator()(const IndexPair& ip, const IndexPair& jp) const`
Return (a reference to) the $M_{i,j}$ or $M_{(I,i),(J,j)}$ matrix element. In sparse matrix classes, returning a reference might have the unintended side effect of filling in the matrix element (assuming that it is not already filled in), even when the intent is to just read it. To avoid this behavior, use the `read` method, below.

`SCALAR read(const int i, const int j) const`
`SCALAR read(const int I, const int i, const int J, const int j) const`
`SCALAR read(const IndexPair& ip, const IndexPair& jp) const`
Return the value of $M_{i,j}$ or $M_{(I,i),(J,j)}$, guaranteeing side effect free behavior, even when `M` is not `const`.

`BlockedVector<Cvector> row(const int i) const`
`BlockedVector<Cvector> row(const int I, const int i) const`
`BlockedVector<Cvector> row(const IndexPair& ip) const`
Return the i'th or (I, i) row of M.

`BlockedVector<Cvector> column(const int j) const`
`BlockedVector<Cvector> column(const int J, const int j) const`
`BlockedVector<Cvector> column(const IndexPair& jp) const`
Return the j'th or (J, j) column of M.

`BlockedVector<Cvector> diag() const`
Return the vector of diagonal elements of M.

ITERATORS

`void for_each_block(std::function<void(INDEX,INDEX,MATRIX&)> lambda)`
`void for_each_block(std::function<void(INDEX,INDEX,const MATRIX&)> lambda) const`
Apply the function `lambda` to each block of M. The three arguments of `lambda` are I, J, and a reference to $[\![M]\!]_{I,J}$.

`void for_each_block_in_street(I,std::function<void(INDEX,MATRIX&)> lambda)`
`void for_each_block_in_street(I,std::function<void(INDEX,const MATRIX&)> lambda) const`
Apply the function `lambda` to each block of M. The two arguments of `lambda` are J, and a reference to $[\![M]\!]_{I,J}$.

`void for_each_block_in_tower(J,std::function<void(INDEX,MATRIX&)> lambda)`
`void for_each_block_in_tower(J,std::function<void(INDEX,const MATRIX&)> lambda) const`
Apply the function `lambda` to each block of M. The two arguments of `lambda` are I, and a reference to $[\![M]\!]_{I,J}$.

VIEWS

`BlockedMatrixView<VECTOR> operator()(const BindexMap& phi, const BindexMap& psi)`
`const BlockedMatrixView<VECTOR> operator()(const BindexMap& phi, const BindexMap& psi) const`
Return a view to the submatrix B in which $B_{i,j} = M_{\phi(i),\psi(j)}$.

CONVERSIONS

`BlockedMatrix<MATRIX>(const BlockedMatrix<MATRIX2> B)`
Covert `B` to a `BlockedMatrix<MATRIX>` type matrix.

MERGING

`void mergeVertically(BlockedMatrix<MATRIX>&& B)`
`void mergeVertically(const BlockedMatrix<MATRIX>& B)`
Merge `B` to M vertically, i.e., as its last streets. The first method destroys `B`.

`void mergeHorizontally(BlockedMatrix<MATRIX>&& B)`
`void mergeHorizontally(const BlockedMatrix<MATRIX>& B)`
Merge `B` to M horizontally, i.e., as its last towers. The first method destroys `B`.

REMAPPINGS

`BlockedMatrix<MATRIX> remapRows(const BindexMap& map) const`
`BlockedMatrix<MATRIX> remapRows(const Inverse<BindexMap>& map) const`
Remap the rows of M by `map` / the inverse of `map`.

`BlockedMatrix<MATRIX> remapCols(const BindexMap& map) const`
`BlockedMatrix<MATRIX> remapCols(const Inverse<BindexMap>& map) const`
Remap the columns of M by `map` / the inverse of `map`.

`BlockedMatrix<MATRIX> remap(const BindexMap& rmap, const BindexMap& cmap) const`
`BlockedMatrix<MATRIX> remap(const Inverse<BindexMap>& rmap,`
`const Inverse<BindexMap>& cmap) const`
Remap the rows and columns of M by (the inverse of) `rmap` and `cmap`.

IN-PLACE ARITHMETIC

`BlockedMatrix<MATRIX>& operator+=(const SCALAR c)`
`BlockedMatrix<MATRIX>& operator-=(const SCALAR c)`
`BlockedMatrix<MATRIX>& operator*=(const SCALAR c)`
`BlockedMatrix<MATRIX>& operator/=(const SCALAR c)`
Increment/decrement/multiply/divide each element of M by c.

`BlockedMatrix<MATRIX>& operator+=(const BlockedMatrix<MATRIX>2& B)`
`BlockedMatrix<MATRIX>& operator-=(const BlockedMatrix<MATRIX>2& B)`
Set M to $M+B$ or $M-B$, respectively.

IN-PLACE OPERATIONS

`BlockedMatrix<MATRIX>& multiplyRowsBy(const BlockedVector<Cvector>& v)`
`BlockedMatrix<MATRIX>& divideRowsBy(const BlockedVector<Cvector>& v)`
Multiply/divide the i'th row of M by v_i.

`BlockedMatrix<MATRIX>& multiplyColsBy(const BlockedVector<Cvector>& v)`
`BlockedMatrix<MATRIX>& divideColsBy(const BlockedVector<Cvector>& v)`
Multiply/divide the j'th column of M by v_j.

VECTOR VALUED ARITHMETIC

`BlockedVector<VECTOR> mult(const BlockedVector<VECTOR>& v) const`
`BlockedVector<VECTOR> dot(const BlockedVector<VECTOR>& v) const`
Compute $M\mathbf{v}$ or $M^\top\mathbf{v}$. The block structure of $\boldsymbol{v}$ must match the column resp. row structure of M.

`BlockedVector<VECTOR> operator*(const BlockedVector<VECTOR>& v) const`
Synonym of `mult`.

MATRIX VALUED ARITHMETIC

`BlockedMatrix<MATRIX> mult(const SCALAR c) const`
Compute cA.

`BlockedMatrix<MATRIX> plus(const BlockedMatrix<MATRIX>& B) const`
`BlockedMatrix<MATRIX> minus(const BlockedMatrix<MATRIX>& B) const`
Compute $A+B$ or $A-B$.

`BlockedMatrix<MATRIX> mult(const BlockedMatrix<MATRIX>& B) const`
`BlockedMatrix<MATRIX> dot(const BlockedMatrix<MATRIX>& B) const`
`BlockedMatrix<MATRIX> outer(const BlockedMatrix<MATRIX>& B) const`
`BlockedMatrix<MATRIX> dott(const BlockedMatrix<MATRIX>& B) const`
Compute the matrix/matrix product (a) AB, (b) $A^\top B$, and (c) $AB^\top$ and (d) $A^\top B^\top$.

`BlockedMatrix<MATRIX> rowGram() const`
`BlockedMatrix<MATRIX> colGram() const`
Compute the symmetric matrix G defined (a) $G_{i,j} = A_{i,*} \cdot A_{j,*}$ (b) $G_{i,j} = A_{*,i} \cdot A_{*,j}$.

`BlockedMatrix<MATRIX> operator*(const SCALAR c) const`
`BlockedMatrix<MATRIX> operator+(const BlockedMatrix<MATRIX>& B) const`
`BlockedMatrix<MATRIX> operator-(const BlockedMatrix<MATRIX>& B) const`
`BlockedMatrix<MATRIX> operator*(const BlockedMatrix<MATRIX>& B) const`
Synonyms of `plus`, `minus` and `mult` methods.

OTHER METHODS

`Bstructure rowStructure() const`
`Bstructure columnStructure() const`
Return the row/column structure of `M`.

FORMATTED I/O

`BlockedMatrix<MATRIX>(const Bstructure& rst, const Bstructure& cst, const MatrixIF& file)`
Load M from the file `file` and block it according to `rst` and `cst`.

`saveto(MatrixOF& file)`
Save M to the file `file`.

VARIABLES

`int nb, mb`
The number of rows/columns of blocks in M (inherited from `BlockedArray`).

`int nrows, ncols`
The total number of rows/columns of matrix elements in M (inherited from `Matrix`).

`MATRIX** blockp`
An `nb`×`mb` column major array of pointers to the individual blocks.

5. Vector/matrix views

VectorView<VECTOR>

Given a parent object `v` of type `VECTOR` and an index map ϕ, `VectorView<VECTOR>` provides an interface to a subvector $\mathbf{w} = (v_{\phi(1)}, v_{\phi(2)}, \ldots, v_{\phi(k)})$ of $\mathbf{v}$ as if it were a separate k dimensional vector. All the usual vector functionality is available to `w`, but its elements remain tied to `v`: changing $v_{\phi(i)}$ will change w_i, and vice versa. Destroying `v` before `w` might result in an error or undefined behavior.

Derived from: `Vector, (Detachable), (Serializable)`
Dependent on: The parent vector, `v`.

CONSTRUCTORS

`VectorView<VECTOR>(VECTOR& v, const IndexMap& phi)`
`VectorView<VECTOR>(VECTOR& v, IndexMap&& phi)`
: Create a view of $(v_{\phi(1)}, v_{\phi(2)}, \ldots, v_{\phi(k)})$. The second version destroys `phi`.

CONVERSIONS

`operator Cvector()`
: Copy the view into a `Cvector`. While defined generically in `Vector`, this method is repeated here because it is the standard way of extracting a subvector of a vector.

ASSIGNMENTS

`operator=(const VECTOR2& u)`
: Set each element of `w` equal to the corresponding element of `u`. Note that, in contrast to the usual definition of `operator=`, this method cannot change the size of `w`, therefore `u` and `w` must be of the same dimension.

VARIABLES

`VECTOR& v`
: A reference to the parent object, `v`.

`IndexMap map`
: The index map defining which elements of $\mathbf{v}$ are included in $\mathbf{w}$.

MatrixView<MATRIX>

`MatrixView<MATRIX>` provides an interface to a submatrix of a `MATRIX` object as if it were a separate matrix. If `A` is a `MatrixView<MATRIX>` of a parent matrix `M` (of type `Matrix`), then all the usual matrix functionality is available to `A`, but its elements will remain tied to the corresponding elements of `M`. Destroying `M` before `A` may results in an error or undefined behavior.

Derived from: `Matrix, (Detachable), (Serializable)`
Dependent on: The parent `MATRIX M`.

CONSTRUCTORS

`MatrixView<MATRIX>(MATRIX& M, const IndexMap& phi, const IndexMap& psi)`
`MatrixView<MATRIX>(MATRIX& M, const IndexMap&& phi, const IndexMap&& psi)`
Create a view of the submatrix of M at the intersection of rows `phi` and columns `psi`. The second version destroys `phi` and `psi`.

`MatrixView<MATRIX>(MATRIX& M, const IdentityIndexMap& dummy, const IndexMap& psi)`
`MatrixView<MATRIX>(MATRIX& M, const IndetityIndexMap& dummy, const IndexMap&& psi)`
Create a view of the submatrix of M consisting of the columns `psi`. The second version destroys `psi`.

`MatrixView<MATRIX>(MATRIX& M, const IndexMap& phi, const IdentityIndexMap& dummy)`
`MatrixView<MATRIX>(MATRIX& M, const IndexMap&& phi, const IdentityIndexMap&& dummy)`
Create a view of the submatrix of M consiting of the rows `phi`. The second version destroys `phi`.

`MatrixView<MATRIX>(MATRIX& M, const IndexMap& phi)`
`MatrixView<MATRIX>(MATRIX& M, const IndexMap&& phi)`
Synonyms of the above.

CONVERSIONS

`operator Cmatrix()`
Copy the view into a `Cmatrix`. While defined generically in `Matrix`, this method is repeated here because it is the standard way of extracting a submatrix of a matrix.

ASSIGNMENTS

`operator=(const MATRIX2& B)`
Set each element of `A` equal to the corresponding element of `B`. Note that, in contrast to the usual definition of `operator=`, this method cannot change the dimensions of `A`, therefore `A` and `B` must be the same size.

VARIABLES

`MATRIX& M`
A reference to the parent object

`IndexMap rmap, cmap`
The index sets defining which rows/columns of M are included in A.

BlockedVectorView<VECTOR>

Given a parent object `v` of type `VECTOR` and an index set $I = (i_1, \ldots, i_k)$, `BlockedVectorView<VECTOR>` provides an interface to a subvector $\boldsymbol{w} = (v_{i_1}, v_{i_2}, \ldots, v_{i_k})$ of $\mathbf{v}$ as if it were a separate k dimensional vector. All the usual vector functionality is available to `w`, but its elements remain tied to `v`: changing v_{i_j} will change the corresponding element of `w`, and vice versa, changing `w` changes the corresponding elements of `v`. Destroying `v` before `w` results in an error or undefined behavior.

Derived from: `Vector, (Detachable), (Serializable)`
Dependent on: The parent vector `v`.

CONSTRUCTORS

`BlockedVectorView<VECTOR>(BlockedVector<VECTOR>& v, const BindexMap& phi)`
`BlockedVectorView<VECTOR>(BlockedVector<VECTOR>& v, const BindexMap&& phi)`
Create a view of $(v_{i_1}, v_{i_2}, \ldots, v_{i_k})$, where $I = (i_1, \ldots, i_k)$.

CONVERSIONS

`operator Cmatrix()`
Copy the view into a `Cvector`. While defined generically in `Vector`, this method is repeated here because it is the standard way of extracting a subvector of a vector.

ASSIGNMENTS

`operator=(const VECTOR2& u)`
Set each element of `w` equal to the corresponding element of `u`. Note that, in contrast to the usual definition of `operator=`, this method cannot change the size of `w`, therefore `u` and `w` must be of the same dimension.

VARIABLES

`BlockedVector<VECTOR>& v`
A reference to the parent object

`BindexMap map`
The index set defining which elements of v are included in w.

BlockedMatrixView<MATRIX>

`BlockedMatrixView<MATRIX>` provides an interface to a submatrix of a `MATRIX` object as if it were a separate matrix. If `A` is a `BlockedMatrixView<MATRIX>` of a parent matrix `M` (of type `Matrix`), then all the usual matrix functionality is available to `A`, but its elements will remain tied to the corresponding elements of `M`. Destroying `M` before `A` may results in an error or undefined behavior.

Derived from: `Matrix, (Detachable), (Serializable)`
Dependent on: The parent `MATRIX M`.

CONSTRUCTORS

`BlockedMatrixView<MATRIX>(BlockedMatrix<MATRIX>& M, const BindexMap& phi, const BindexMap& psi)`
`BlockedMatrixView<MATRIX>(BlockedMatrix<MATRIX>& M, const BindexMap&& phi, const BindexMap&& psi)`

Create a view of the submatrix of M at the intersection of rows *phi* and columns *psi*.

CONVERSIONS

`operator Cmatrix()`
Copy the view into a `Cmatrix`. While defined generically in `Vector`, this method is repeated here because it is the standard way of extracting a submatrix of a matrix.

ASSIGNMENTS

`operator=(const MATRIX2& B)`
Set each element of `A` equal to the corresponding element of `B`. Note that, in contrast to the usual definition of `operator=`, this method cannot change the dimensions of `A`, therefore `A` and `B` must be the same size.

VARIABLES

`BlockedMatrix<MATRIX>& M`
A reference to the parent object

`BindexMap rmap, cmap`
The index sets defining which rows/columns of M are included in A.

6. Active vectors/matrices

ActiveVector<VECTOR>

An `ActiveVector<VECTOR>` object behaves the same way as a `VECTOR` except that every time one of its elements changes, the `changed` method gets called. What this method actually does is defined in the concrete classes derived from `ActiveVector<VECTOR>` (see below).

Derived from: `VECTOR, (Vector), (Detachable), (Serializable)`

CHANGE FUNCTIONS

`virtual void changed(const INDEX i, const SCALAR x)`
This method is called every time element some vector element v_i is changed to value x.

`void changed(const INDEX i)`
A shortcut for `changed(i,VECTOR::read(i))`.

OrderedVector<VECTOR>

An `OrderedVector<VECTOR>` is an active vector that behaves identically to `VECTOR` except that it internally maintains and automatically updates an ordering of its elements

Derived from: `ActiveVector<VECTOR>, (VECTOR), (Vector), (Detachable), (Serializable)`

METHODS

`int best(const int k=0)`
Return the index of the $k+1$'th element of v from the top according to the ordering.

`int worst(const int k=0)`
Return the index of the $k+1$'th element of v from the bottom accoeding to the ordering.

AccumulatedVector<VECTOR>

An `AccumulatedVector<VECTOR>` is an active vector that behaves identically to `VECTOR` except that it internally maintains and automatically updates an accumulation (in the default case, the sum) of its elements.

Derived from: `ActiveVector<VECTOR>, (VECTOR), (Vector), (Detachable), (Serializable)`

CONSTRUCTORS

`SCALAR sum() const`
: Return the sum of the elements.

Operators, etc.

7. Operators

GivensRotation

A Givens rotation is a 2×2 elementary rotation

$$G_\theta = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

on some pair of indices (i_1, i_2). Being able to apply Givens rotations to vectors/matrices fast is critical, and strongly dependent on the exact way that the matrix/vector is stored. Therefore, implementing multiplication by Givens rotations is left to the vector and matrix classes, rather than being defined here.

CONSTRUCTORS

`GivensRotation(int i1, int i2, double cos, double sin)`
: A new Givens rotation on `i1` and `i2`.

`GivensRotation(int i1, int i2, double theta)`
: A new Givens rotation on `i1` and `i2` with angle θ.

NAMED CONSTRUCTORS

`GivensRotation::Random(int n)`
: A random Givens rotation, where i_1 and i_2 are chosen from $\{1, 2, \ldots, n\}$ (and are distinct), while $\theta \sim \text{Uniform}(0, 2\pi)$.

VARIABLES

`int i1, i2`
: The two indices i_1 and i_2.

`double cos, sin`
: $\cos\theta$ and $\sin\theta$.

KpointOp<k>

A KpointOp<K> object represents a $k \times k$ elementary rotation on some set of indices $(i_1, i_2, \ldots, i_k)$. As for Givens rotations, being able to apply such rotations to matrices and vectors fast is critical, therefore the implementation of this operation is relegated to the matrix and vector classes.

CONSTRUCTORS

`KpointOp<k>(const int k)`
: A new k–point rotation, in which `map` and `q` are allocated but unitialized.

`KpointOp<k>(const IndexMap& map)`
`KpointOp<k>(const IndexMap& map, const Cmatrix& M)`
`KpointOp<k>(IndexMap&& map, const Cmatrix&& M)`
: A new k–point operator initialized from `map` and `M`.

CONVERSIONS

`operator Cmatrix() const`
: Return the $k \times k$ non-trivial block of `O` as a `Cmatrix`.

METHODS

`applyTo(SCALAR* v)`
`applyTo(SCALAR* v)`
: Apply the (transpose of) T to the k-element array `v`.

`applyTo(SCALAR** vptr)`
`applyTo(SCALAR** vptr)`
: Apply the (transpose of) T to `[*vptr[0],*vptr[1],...,*vptr[k]]`.

VARIABLES

`IndexMap& map`
: The index map ϕ.

`double* q`
: The $k \times k$ block stored as a raw $k \times k$ array.

MatrixSum<MATRIX>

A `MatrixSum<MATRIX>` object represents a sum $M_1 + M_2 + \ldots + M_k$, where each M_i is a matrix of type `MATRIX`. `MatrixSum` supports all the usual matrix operations, except, of course, direct assignments to a `MatrixSum` object are not possible.

Derived from: `Matrix, (Detachable), (Serializable)`
Owned objects: The `MATRIX` constituents

CONSTRUCTORS

`MatrixSum<MATRIX>(const vector<MATRIX> mlist)`
`MatrixSum<MATRIX>(vector<MATRIX>&& mlist)`
: Create a `MatrixSum` composed of the elements of `mlist`. The second of these constructors destroys the original matrices.

VARIABLES

`vector<MATRIX> matrix`
: The indiviudal `MATRIX` terms of this sum.

MatrixProduct<MATRIX>

A `MatrixProduct<MATRIX>` object represents a product $M_1 M_2 \dots M_k$, where each M_i is a matrix of type `MATRIX`. `MatrixProduct` supports all the usual matrix operations, except, of course, direct assignments to a `MatrixProduct` object are not possible.

Derived from: `Matrix, (Detachable), (Serializable)`
Owned objects: The `MATRIX` constituents

CONSTRUCTORS

`MatrixProduct<MATRIX>(const vector<MATRIX> mlist)`
`MatrixProduct<MATRIX>(vector<MATRIX>&& mlist)`
: Create a `MatrixProduct` composed of the elements of `mlist`. The second of these constructors destroys the original matrices.

VARIABLES

`vector<MATRIX> matrix`
: The indiviudal `MATRIX` factors of this sum.

OuterProduct<CLASS1,CLASS2>

Given a matrix or vector object `V` of type `CLASS1` and another one, `V` of type `CLASS2` an object of type `OuterProduct<CLASS1,CLASS2>` formed from them represents $UV^\top$. The `OuterProduct` class supports all the usual matrix operations, except, of course, direct assignments to a `OuterProduct` object are not possible.

Derived from: `Matrix, (Detachable), (Serializable)`

CONSTRUCTORS

```
OuterProduct<CLASS1,CLASS2>(const CLASS1& U, const CLASS2& V)
OuterProduct<CLASS1,CLASS2>(CLASS1&& U, const CLASS2&& V)
```

VARIABLES

`CLASS1 U`
`CLASS2 V`
: The left and the right factors in the outer product.

BiMatrix<MATRIX1,MATRIX2>

Some composite matrix types mix blocks of different types. For example, each block in a HODLR matrix is either a low rank matrix (i.e, an `OuterProduct`) or a smaller HODLR matrix. `BiMatrix` makes such composite blocked matrices possible by functioning as a wrapper for two matrices at the same time, of different types.

Derived from: `Matrix, (Detachable), (Serializable)`

CONSTRUCTORS

```
BiMatrix<MATRIX1,MATRIX2>(const MATRIX1& M1, const MATRIX2& M2)
BiMatrix<MATRIX1,MATRIX2>(MATRIX1&& M1, const MATRIX2&& M2)
```

VARIABLES

`bool type`
: Switch to determine whether this object functions as a `MATRIX1` or `MATRIX2`.

`MATRIX1 M1`
`MATRIX2 M2`
: The alternative matrix object.

8. Index maps

IndexMap

An `IndexMap` ϕ maps $\{0, 1, 2, \ldots, n-1\}$ to (a subset of) the row or column indices $\{0, 1, 2, \ldots, m-1\}$ of a vector or a matrix. `IndexMap` is normally used in one of the following two ways: (a) to define a subset of the indices of a vector or a matrix (for example to construct a `VectorView`), (b) to define a (possibly non-surjective) mapping from the indices of one vector to the indices of another (for example, in `VECTOR::remap`).

CONSTRUCTORS

`IndexMap(const int n)`
: An `IndexMap` with domain $\{0, 1, 2, \ldots, n-1\}$ whose elements are uninitialized.

`IndexMap(const initializer_list<INDEX> list)`
: Initialize the `IndexMap` from the initializer list `list`.

NAMED CONSTRUCTORS

`IndexMap::Identity(const int n)`
: The identity map from $\{0, 1, 2, \ldots, n-1\}$ to itself.

`IndexMap::Random(const int n, const int m)`
: A random index map from $\{0, 1, 2, \ldots, n-1\}$ to $\{0, 1, 2, \ldots, m-1\}$.

CONVERSIONS

`IndexMap(const IndexBiMap& psi)`
`IndexMap(IndexBiMap&& psi)`
: Upcast an `IndexBiMap` to an `IndexMap`.

ELEMENT ACCESS

`INDEX& operator()(const int i)`
`INDEX operator()(const int i) const`
: Return (a reference to) $\phi(i)$.

METHODS

`void sort()`
Permute ϕ from the left so that $\phi(0) \leq \phi(1) \leq \ldots \leq \phi(n)$.

`IndexMap inverse() const`
Compute the inverse map ϕ^{-1}.

`IndexMap& applyFromLeft(const IndexMap& psi)`
Apply `psi` to `phi` in place to get the composite map $\psi \circ \phi$.

`IndexMap& applyFromRight(const IndexMap& psi)`
Apply `psi` to `phi` in place to get the composite map $\phi \circ \psi^{-1}$.

VARIABLES

`int nsource` n

`INDEX* forward`
The i'th element of this array is $\phi(i)$.

IndexBiMap

An `IndexBiMap` is a bidirectional index map ϕ from ϕ: $\{1, 2, \ldots, n\}$ to $\{1, 2, \ldots, m\}$. The map might or might not be a bijection. If $i \in \{0, 1, 2, \ldots, n-1\}$ is not mapped to any $\{0, 1, 2, \ldots, m-1\}$ by ϕ, `forw(i)` (equivalent to `operator()(i)`) returns `-1`. Similarly, if $j \in \{0, 1, 2, \ldots, m-1\}$ is not the preimage of any $i \in \{0, 1, 2, \ldots, n-1\}$, then `backw(j)` returns `-1`.

Derived from: `IndexMap`

CONSTRUCTORS

`IndexBiMap(const int k)`
: An unitialized `IndexBiMap` from $\{0, 1, 2, \ldots, k-1\}$ to itself.

`IndexBiMap(const initializer_list<INDEX> list)`
: Initialize the `IndexBiMap` from the initializer list `list`.

NAMED CONSTRUCTORS

`IndexBiMap::Identity(const int n)`
: The identity map from $\{0, 1, 2, \ldots, n-1\}$ to itself.

`IndexBiMap::Random(const int n, const int m)`
: A random index map from $\{0, 1, 2, \ldots, n-1\}$ to $\{0, 1, 2, \ldots, m-1\}$.

CONVERSIONS

`IndexBiMap(const IndexMap& psi, const int m=-1)`
`IndexBiMap(IndexMap&& psi, const int m=-1)`
: Construct an `IndexBiMap` from ψ with range $\{0, 1, 2, \ldots, m-1\}$. If `m=-1`, then m will be the largest index in the range of ψ.

ELEMENT ACCESS

`INDEX operator()(const int i) const`
`INDEX forw(const int i)`
: Return $\phi(i)$ or -1 is i is not mapped to any element of $\{1, 2, \ldots, m\}$.

`INDEX backw(const int j)`
: Return $\phi^{-1}(j)$ or -1 is j is not the preimage of any $i \in \{1, 2, \ldots, n\}$.

METHODS

`void sort()`
: Permute ϕ from the left so that $\phi(0) \leq \phi(1) \leq \ldots \leq \phi(n)$.

`IndexBiMap& swap(const int i1, const int i2)`
Swap the locations where i_1 and i_2 are mapped.

`IndexBiMap inverse() const`
Return the inverse map ϕ^{-1}.

`IndexBiMap& applyFromLeft(const IndexMap& psi)`
Apply `psi` to `phi` in place to get the composite map $\psi \circ \phi$.

`IndexBiMap& applyFromRight(const IndexMap& psi)`
Apply `psi` to `phi` in place to get the composite map $\phi \circ \psi^{-1}$.

VARIABLES

`int nsource` n

`int ndest` m

`INDEX* forward`
The i'th element of this array is $\phi(i)$ (inherited from `IndexMap`).

`INDEX* backward`
The j'th element of this array is $\phi^{-1}(j)$.

BindexMap

A `BindexMap` ϕ (short for "blocked index map") maps $\{0, 1, 2, \ldots, n-1\}$ to (a subset of) the row or column indices of a blocked vector or a blocked matrix. Therefore, for any $i \in \{0, 1, 2, \ldots, n-1\}$, $\phi(i)$ is a block/index pair (J, j) stored in an `iipair` object.

CONSTRUCTORS

`BindexMap(const int n)`
: An uninitialized index map with domain $\{0, 1, 2, \ldots, n-1\}$.

`BindexMap(const initializer_list<iipair> list)`
: Initialize the `BindexMap` from the initializer list `list` (see `example1.cpp`).

NAMED CONSTRUCTORS

`BindexMap::Identity(const Bstructure& st)`
: Construct the identity index map from $\{0, 1, 2, \ldots, n-1\}$ to `st`.

ELEMENT ACCESS

`iipair& operator(const int i)`
`iipair operator(const int i) const`
: Return (a reference to) $\phi(i)$.

VARIABLES

`int nsource` n

`vector<iipair>`
: The i'th element of this `vector` is $\phi(i)$.

BtoBindexMap

A `BtoBindexMap` ϕ: $(I, i) \mapsto (J, j)$ (short for "blocked to blocked index map") maps the (row or column) indices of one blocked vector or matrix to the (row or column) indices of another.

CONSTRUCTORS

`BtoBindexMap(const int bsource)`
: An uninitialized index map with `_nsource` blocks in its domain.

NAMED CONSTRUCTORS

`BtoBindexMap::Identity(const Bstructure& st)`
: Construct the identity index map from an index set of structure `st` to itself.

CONVERSIONS

`BtoBindexMap(const BtoBindexBiMap& psi)`
`BtoBindexMap(BtoBindexBiMap&& psi)`
: Upcast `psi` to a `BtoBindexMap`.

ELEMENT ACCESS

```
iipair& operator(const int I, const int i)
iipair operator(const int I, const int i) const
iipair& operator(const iipair& ip)
iipair operator(const iipair& ip) const
```
: Return (a reference to) $\phi((I, i))$.

METHODS

`BtoBindexMap inverse() const`
: Return the inverse of ϕ (if it exists).

VARIABLES

`int nsource` The number of source blocks

`vector<BindexMap> forward`
: The I'th element of this array is the `BindexMap` describing where each element of the I'th source block is mapped.

BtoBindexBiMap

A `BtoBindexBiMap` ϕ is a bidirectional `BtoBIndexMap` mapping the (row or column) indices of one blocked vector or matrix to the (row or column) indices of another.

Derived from: `BtoBindexMap`

CONSTRUCTORS

`BtoBindexBiMap(const int bsource, const int bdest)`
: An uninitialized `BtoBindexBiMap` with `bsource` source blocks and `bdest` destination blocks.

NAMED CONSTRUCTORS

`BtoBindexBiMap::Identity(const Bstructure& st)`
: Construct the identity index map from an index set of structure `st` to itself.

CONVERSIONS

`BtoBindexBiMap(const BtoBindexMap& psi)`
`BtoBindexBiMap(BtoBindexMap&& psi)`
: Construct a bidirectional index map from `psi`.

ELEMENT ACCESS

`iipair operator(const int I, const int i) const`
`iipair operator(const iipair& ip) const`
`iipair forw(const int I, const int i) const`
`iipair forw(const iipair& ip) const`
: Return $\phi((I, i))$.

`iipair backw(const int J, const int j) const`
`iipair backw(const iipair& jp) const`
: Return $\phi^{-1}((J, j))$.

METHODS

`BtoBindexMap inverse() const`
: Return the inverse of ϕ (inherited from `BtoBIndexMap`).

VARIABLES

`int nsource`
: The number of source blocks (inherited from `BtoBindexMap`).

`int ndest`
: The number of destination blocks.

`vector<BindexMap> forward`
: The I'th element of this array is the `BIndexMap` describing where each element of the I'th source block is mapped (inherited from `BtoBindexMap`).

`vector<BindexMap> backward`
: The J'th element of this array is the `BIndexMap` describing where each element of the J'th source block is mapped by ϕ^{-1}.

Activemap

Various algorithms involve eliminating rows/columns of a matrix one-by-one. An `Activemap` is an `IndexBiMap` that is used in such situations to keep track of which rows/columns are active at any point in time.

Derived from: `IndexBiMap`

CONSTRUCTORS

`Activemap::AllActive(const int n)`
: An active map in which all the indices $\{0, 1, 2, \ldots, n-1\}$ are initialized to be active.

`Activemap::NoneActive(const int n)`
: An active map in which all the indices $\{0, 1, 2, \ldots, n-1\}$ are initialized to be inactive.

ELEMENT ACCESS

`INDEX operator()(const int i)`
: Return the i'th active index (it is assumed that $i < n_{\text{active}}$, and the indices do not necessarily appear in order).

MAPPING OVER ELEMENTS

`void for_each_active(std::function<void(const INDEX)> lambda)`
: Apply the function `lambda` to each active index.

ACTIVATING/DEACTIVATING INDICES

`void activate(const int j)`
`void deactivate(const int j)`
: Add/remove j to/from the list of active indices.

`void activate_at_pos(const int j)`
`void deactivate_at_pos(const int j)`
: Add/remove the index at the i'th position to/from the list of active indices.

METHODS

`INDEX random()`
: Return one of the n_{active} active indices, chosen uniformly at random.

`IndexMap sample(const int k)`
: Sample k indices uniformly at random (without replacement) from the list of active indices.

VARIABLES

`nactive`
: The number of active indices, n_{active}.

Bactivemap

`Bactivemap` is the analog of `Activemap` for indexing into blocked vectors/matrices.

Derived from: `BindexBiMap`

CONSTRUCTORS

`Bactivemap::AllActive(const Bstructure& st)`
An active map in which all the indices are initialized to be active.

`Bactivemap::NoneActive(const Bstructure& st)`
An active map in which all the indices are initialized to be inactive.

ELEMENT ACCESS

`iipair operator()(const int i)`
Return the i'th active index block/index pair (J, j) (it is assumed that $i < n_{\text{active}}$.)

MAPPING OVER ELEMENTS

`void for_each_active(std::function<void(const iipair)> lambda)`
Apply the function `lambda` to each active (J, j) pair.

ACTIVATING/DEACTIVATING INDICES

`void activate(const iipair& p)`
`void activate(const int J, const int j)`
Add (J, j) to the list of active block/index pairs.

`void deactivate(const iipair& p`
`void deactivate(const int J, const int j)`
Remove (J, j) from the list of active block/index pairs.

`void activate_at_pos(const int i)`
`void deactivate_at_pos(const int i)`
Add/remove the block/index pair at the i'th position to/from the list of active indices.

METHODS

`INDEX random()`
Return one of the n_{active} active block/indices pairs, chosen uniformly at random.

`IndexMap sample(const int k)`
Sample k block/index pairs uniformly at random (without replacement) from the active list.

VARIABLES

`nactive`
: The number of active block/index pairs, n_{active}.

Multithreading

9. Basic multithreading

ThreadManager

Every Mondrian program must have a single global `ThreadManager` object, called `threadManager` (defined in the `Mondrian_base.inc`) which coordinates the sequence in which `MultiLoop` and `ThreadBank` objects launch threads. Amongst other things, `threadManager` ensures that the system's limit on the number of enqued threads is not exceeded.

Note that "launching" a thread in this context really means passing it to the system's own low level thread scheduler. Similarly, by the number of active threads we mean the number of threads that are either running or waiting to be launched by the low level scheduler.

CONSTRUCTORS

`ThreadManager(const int _maxthreads)`
: Construct a `ThreadManager` object, which will limit the total number of active threads (i.e., running or queued to the low level scheduler) to `_maxthreads`. In general, there is no advantage to `_maxthreads` exceeding the number of cores on the system.

METHODS

`void enqueue(ThreadBank* bank)`
: Add `bank` to the list of `ThreadBank` objects waiting to launch a new thread. When a new slot becomes available (i.e., `nthreads` falls below `maxthreads`), the manager will signal to one of the `ThreadBank` objects on this list that it may launch a single new thread (see `release`).

`void release(ThreadBank* bank)`
: This function is called when a thread from `bank` terminates. It decrements `bank`'s number of active threads and attempts to find another `ThreadBank` waiting to launch a new thread in place of the one that is terminating.

`int get_nthreads()`
: Return the number of active threads, not counting privileged threads (see `ThreadBank`).

MultiLoop

`MultiLoop` is the simplest multithreading class in Mondrian, which simply implements a parallel `for` loop. Internally, every `MultiLoop` instance has its own `ThreadBank`.

CONSTRUCTORS

`MultiLoop(const int n, std::function<void(int)> lambda)`
: Execute the function `lambda(const int i)` a total of `n` times, in parallel, with `i` set to $1, 2, \ldots, n$.

ThreadBank

The `ThreadBank` class is used to spawn independent threads in a given block of code and then wait for them to finish (the `ThreadBank`'s destructor joins all the threads, therefore the code will wait on all the threads to finish before exiting the block). In addition to the global number of active threads being limited by `threadManager.maxthreads`, each `ThreadBank` can individually limit how many of its threads may be active at any one time. This can be useful for balancing resources in situations where multiple threads spawn further threads of their own via separate `ThreadBank` objects.

Another issue that may arise when n threads each attempt to launch further threads is that the global thread manager delays each of them because n already counts towards the number of active threads. To avoid this problem, each `ThreadManager` instance is allowed to have a certain number of "privileged" threads that do not count towards the grand total.

CONSTRUCTORS

`ThreadBank(const int _maxthreads=1000, const int _maxprivileged=1)`
: Construct a new `ThreadBank` in which the number of active threads is limited to `_maxthreads`, and which has the given number of privileged threads.

METHODS

`void add(FUNCTION lambda, const OBJ x)`
: Launch `lambda` as an independent thread with argument `x`. The type of `lambda` must match the type of `x`. For example if `OBJ` is `int`, `FUNCTION` might be `std::function<void(int)>`. If the global limit on the number of active threads has not been reached, `lambda` is launched immediately. Otherwise, execution stalls until the global thread manager signals that a slot has become available.

`void add(FUNCTION lambda, const OBJ1 x1, const OBJ2 x2)`
`void add(FUNCTION lambda, const OBJ1 x1, const OBJ2 x2, const OBJ3 x3)`
: The same as above, but with multiple arguments.

`bool is_ready()`
: True if this `ThreadBank` has a launchable thread, and is waiting on the global manager for permission to launch it.

10. Atomic objects

AtomicVector<VECTOR>

When multiple threads concurrently try to access the same vector, race conditions may occur. This issue is particularly serious for sparse vectors, because if one thread forces the underlying data structure (such as an `stl::vector` for `Vectorv`) to be reorganized, and another thread attempts to access it before the reorganization is complete, the code may crash.

The `AtomicVector` wrapper solves this problem by using a `mutex` to ensure that any one time only on thread is allowed to write to the vector. Read accesses are not protected from race conditions. In every other aspect `AtomicVector<VECTOR>` behaves the same way as a `VECTOR` object.

Derived from: `VECTOR, (Vector),`

PROTECTED MEMBER FUNCTIONS

```
void for_each(std::function<void(INDEX,SCALAR&)> lambda)

VECTOR& operator+=(const SCALAR c)
VECTOR& operator-=(const SCALAR c)
VECTOR& operator*=(const SCALAR c)
VECTOR& operator/=(const SCALAR c)
VECTOR& operator+=(const VECTOR& x)
VECTOR& operator-=(const VECTOR& x)

void apply(const OPERATOR& Q)
void applyInv(const OPERATOR& Q)
```

VARIABLES

`mutex mx`

The mutex used to block concurrent write accesses to `v`.

AtomicMatrix<MATRIX>

AtomicMatrix<MATRIX> functions similarly to AtomicVector<VECTOR> but for matrices.

Derived from: MATRIX, (Matrix)

PROTECTED MEMBER FUNCTIONS

```
void for_each(std::function<void(INDEX,INDEX,SCALAR&)> lambda)
void for_each_in_row(const int i, std::function<void(INDEX,SCALAR&)> lambda)
void for_each_in_column(const int j, std::function<void(INDEX,SCALAR&)> lambda)
```

```
MATRIX& operator+=(const SCALAR c)
MATRIX& operator-=(const SCALAR c)
MATRIX& operator*=(const SCALAR c)
MATRIX& operator/=(const SCALAR c)
MATRIX& operator+=(const MATRIX& x)
MATRIX& operator-=(const MATRIX& x)
```

```
MATRIX& MultiplyRowsBy(const Cvector& v)
MATRIX& DivideRowsBy(const Cvector& v)
MATRIX& MultiplyColsBy(const Cvector& v)
MATRIX& DivideColsBy(const Cvector& v)
```

```
void applyFromLeft(const OPERATOR& Q)
void applyFromLeftInv(const OPERATOR& Q)
void applyFromRight(const OPERATOR& Q)
void applyFromRightInv(const OPERATOR& Q)
```

Other Objects

11. Wrappers

Detached<CLASS>

The `Detached<CLASS>` wrapper allows constructing an interface to data stored in another object (called the mother object). The resulting object is said to be *detached*, because deleting it does not delete the mother object (of type `CLASS`). This mechanism is often used in situations where normally bare pointers might be employed.

The `Detached` wrapper works by overriding the destructor of the underlying object, together with some of its copy constructors and assignment operators. Any Mondrian object can be detached in this way, as long as its type, `CLASS`, is derived from the abstract property class `Detachable`. In particular, `CLASS` must define the shallow copy function `shallow()`.

Derived from: `CLASS`
Dependent on: The mother object (of type `CLASS`).

Bibliography

[1] Gaël Guennebaud, Benoît Jacob, et al. Eigen, version 3. http://eigen.tuxfamily.org, 2010.

[2] Risi Kondor, Nedelina Teneva, and Pramod K. Mudrakarta. pMMF: a parallel MMF library. Hosted at https://github.com/risi-kondor/pMMF, 2015.

[3] The GNU Public License, version 3. http://www.gnu.org/licenses/gpl-3.0.en.html, 2007.