pMMF: a C++ library for parallel multiresolution matrix factorization

Risi Kondor, Nedelina Teneva and Pramod K. Mudrakarta
Department of Computer Science, The University of Chicago

Working draft, August 2015

Contents

1. Overview

Multiresolution Matrix Factorization (MMF) has applications in matrix compression, solving large linear systems, constructing wavelet bases on graphs, and learning problems [1]. The first efficient parallel algorithm for computing MMF factorizations was introduced in [2].

`pMMF` is an open source software library implementing the algorithm described in [2], specifically designed for large scale MMF computations on modern multi-core and multi-processor computer architectures. `pMMF` is written in C++, but also provides a MATLAB interface to its core functionality.

`pMMF` is optimized for speed and minimizing memory footprint. To maximally fulfill these objectives, the library defines its own vector, matrix and blocked matrix classes. The design of the library emphasizes modularity and expansibility, and follows a consistently object oriented approach.

MMF and pMMF

In the following, $[n]$ denotes the set $\{1, 2, \ldots, n\}$. Given a matrix $A \in \mathbb{R}^{n\times n}$ and two ordered sets $S_1, S_2 \subseteq [n]$, A_{S_1,S_2} denotes the $|S_1|\times|S_2|$ dimensional submatrix of A cut out by the rows indexed by S_1 and the columns indexed by S_2. Given $S \subseteq [n]$, $\overline{S}$ denotes $[n]\backslash S$. $A_{:,i}$ or $[A]_{:,i}$ denotes the i'th column of A.

The notation $B_1 \cup B_2 \cup \ldots \cup B_m = [n]$ signifies that the sets $B_1, \ldots, B_m$ form a partition of $[n]$.

Multiresolution Matrix Factorization (MMF)

Given a symmetric matrix $A \in \mathbb{R}^{n\times n}$, the **Multiresolution Matrix Factorization (MMF)** of A is a multi-level approximate factorization of the form

$$A \approx Q_1^\top \ldots Q_{L-1}^\top Q_L^\top H\, Q_L\, Q_{L-1} \ldots Q_1, \tag{1}$$

where $Q_1, \ldots, Q_L$ is a sequence of carefully chosen orthogonal matrices (rotations) obeying the following constraints:

MMF1. Each Q_ℓ is chosen from some subclass $\mathcal{Q}$ of highly sparse orthogonal matrices. In the simplest case, $\mathcal{Q}$ is the class of **Givens rotations**, i.e., orthogonal matrices that only differ from the identity

matrix in the four matrix elements

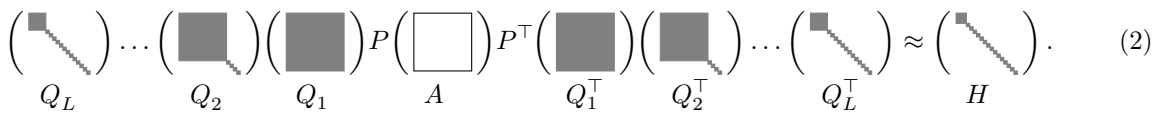
$$[Q_\ell]_{i,i} = \cos\theta, \qquad [Q_\ell]_{i,i} = -\sin\theta,$$
$$[Q_\ell]_{j,i} = \sin\theta, \qquad [Q_\ell]_{j,j} = \cos\theta,$$

for some pair of indices (i, j) and rotation angle θ. Alternatively, $\mathcal{Q}$ may be the class of so-called **k–point rotations**, which rotate not just two, but k coordinates, $(i_1, \ldots, i_k)$.

MMF2. The effective size of the rotations decreases according to a set schedule $n = \delta_0 \geq \delta_1 \geq \ldots \geq \delta_L$, i.e., there is a nested sequence of sets $[n] = S_0 \supseteq S_1 \supseteq \ldots \supseteq S_L$ with $|S_\ell| = \delta_\ell$ such that $[Q_\ell]_{\overline{S_{\ell-1}},\overline{S_{\ell-1}}}$ is the $n-\delta_{\ell-1}$ dimensional identity. S_ℓ is called the **active set** at level ℓ. In the simplest case, exactly one row/column is removed from the active set after each rotation.

MMF3. H is $\mathbf{S_L}$**–core-diagonal**, which means that it is all zero, except for (a) the submatrix $[H]_{S_L,S_L}$, called its core, and (b) the rest of its diagonal.

Moving $Q_1, Q_2, \ldots, Q_L$ over onto the left hand side of (1), MMF can be represented graphically as



(2)

Here P is a permutation matrix whose purpose is to ensure that $S_1, S_2, \ldots, S_L$ always comprise the *first* δ_ℓ indices in $[n]$. P is introduced solely for the sake of making the MMF structure easier to visualize: an actual MMF factorization would not involve such an explicit permutation matrix.

Parallel Multiresultion Matrix Factorization (pMMF)

In a **Parallel Multiresolution Matrix Factorization (pMMF)** the $Q_1, Q_2, \ldots, Q_L$ rotations satisfy additonal block diagonality constraints. Given a partition $B_1 \cup B_2 \cup \ldots \cup B_m$ of $[n]$, we say that a matrix $M \in \mathbb{R}^{n\times n}$ is $(B_1, \ldots, B_m)$–**block-diagonal** if $M_{i,j} = 0$ unless i and j fall in the same set B_u for some u. In a pMMF, $B_1 \cup \ldots \cup B_m$ is formed by clustering the rows/columns of A. Constraining a rotation Q to be $(B_1, \ldots, B_m)$–block-diagonal is equivalent to requiring that it only mix rows/columns of A within clusters rather than across clusters.

Similarly to e.g., block Jacobi methods, imposing such a block structure on the Q_ℓ's can greatly reduce the time required to compute the (approximate) MMF factorization of large matrices, especially on multiprocessor machines. Clustering also meshes well with the notion of locality, one of the key ingredients of the Harmonic Analysis theory behind MMF.

However, imposing a single, fixed block structure on all of the Q_ℓ's would be too restrictive, effectively decoupling the MMF into m separate factorizations. Instead, pMMF groups $Q_1, Q_2, \ldots, Q_L$ into P subsequences $(Q_1, \ldots, Q_{l_1})$, $(Q_{l_1+1}, \ldots, Q_{l_2})$, $\ldots$, $(Q_{l_{P-1}+1}, \ldots, Q_{l_P})$, and allows each subsequence to have its own clustering structure $B_1^p \cup \ldots \cup B_m^p$. The p'th subsequence is called the p'th **stage** of the factorization. Letting $\overline{Q_p} = Q_{l_p} \ldots Q_{l_{p-1}}$, this results in a factorization of the form

$$A \approx \overline{Q}_1^\top \overline{Q}_2^\top \ldots \overline{Q}_P^\top H \, \overline{Q}_P \ldots \overline{Q}_2 \overline{Q}_1, \tag{3}$$

where each $\overline{Q_p}$ is $(B_1^p, \ldots, B_m^p)$–block diagonal.

The big advantage of pMMF is that given the partition $B_1^p \cup \ldots \cup B_m^p$, each diagonal block $[\overline{Q_p}]_{B_u,B_u}$ of $\overline{Q_p}$ can be computed independently of the other blocks. The pMMF algorithm described in [2] exploits this fact, as well as some other computational tricks, to fully parallelize the factorization. Empirically, the running time of pMMF has been observed to scale close to linearly in n, assuming that A is sparse.

Algorithms

Fixing $\mathcal{Q}$, L, and $\delta_1, \delta_2, \ldots, \delta_L$, in general, there is no guarantee that one can find an exact factorization

$$A = Q_1^\top \ldots Q_{L-1}^\top Q_L^\top H\, Q_L\, Q_{L-1} \ldots Q_1, \tag{4}$$

where $Q_1, Q_2, \ldots, Q_L$ and H obey the MMF constraints, MMF1–MMF3. If an exact factorization does exist, there is no guarantee that it is unique.

Consequently, rather than trying to find a single perfect factorization, MMF algorithms usually take an optimization approach, trying to find a combination of the sets $S_1, S_2, \ldots, S_L$ and the rotations $Q_1, Q_2, \ldots, Q_L$ that minimize some notion of error quantifying how close H is to core-diagonal form. The optimization problem is attacked in a greedy way, taking A through the sequence of transformations

$$A \longmapsto \underbrace{Q_1 A\, Q_1^\top}_{A_1} \longmapsto \underbrace{Q_2\, Q_1 A\, Q_1^\top Q_2^\top}_{A_2} \longmapsto \underbrace{Q_3\, Q_2\, Q_1 A\, Q_1^\top Q_2^\top Q_3^\top}_{A_3} \longmapsto \ldots,$$

choosing each Q_ℓ so as to minimize the "out-of-core energy" of the next matrix, A_ℓ (see [1] for details). Parallel MMF similarly maps

$$A \longmapsto \underbrace{\overline{Q}_1 A\, \overline{Q}_1^\top}_{\overline{A}_1} \longmapsto \underbrace{\overline{Q}_2\, \overline{Q}_1 A\, \overline{Q}_1^\top \overline{Q}_2^\top}_{\overline{A}_2} \longmapsto \underbrace{\overline{Q}_3\, \overline{Q}_2\, \overline{Q}_1 A\, \overline{Q}_1^\top \overline{Q}_2^\top \overline{Q}_3^\top}_{\overline{A}_3} \longmapsto \ldots.$$

However, now

(a) Each stage $\overline{A}_{p-1} \mapsto \overline{A}_p$ also involves finding the clustering $B_1^p \cup \ldots \cup B_m^p$ for the active part of $\overline{A}_{p-1}$.

(b) Each $\overline{Q}_p$ is a $(B_1^p, \ldots, B_m^p)$–block diagonal matrix in which each diagonal block $[\overline{Q}_p]_{B_u, B_u}$ is a product of a potentially large number of separate rotations from $\mathcal{Q}$.

Parallel MMF computes each $[\overline{Q}_p]_{B_u, B_u}$ block independently and in parallel with the other blocks.

Given a partition $B_1 \cup B_2 \cup \ldots \cup B_m$ of $[n]$, the blocked matrix form of $M \in \mathbb{R}^{n \times n}$ conceives of M as the union of $m \times m$ smaller matrices, $\{M_{B_i, B_j}\}_{i,j=1}^m$. Storing large matrices in blocked form potentially has many advantages, especially if the different blocks can be allocated to different processors or different machines.

A key observation in pMMF is that not only $\overline{Q}_p$, but $\overline{A}_{p-1}$ can also be stored in blocked matrix form. Moreover, computing the u'th block of rotations, $[\overline{Q}_p]_{B_u, B_u}$, only requires the part of $\overline{A}_{p-1}$ formed by the column of blocks $\{[\overline{A}_{p-1}]_{B_i, B_u}\}_{i=1}^m$. Therefore, once $\overline{A}_{p-1}$ has been separated into $m \times m$ blocks according to $B_1^p \cup B_2^p \cup \ldots \cup B_m^p$, each column of blocks can be sent to a different processor or machine, which can perform all computations necessary to determine the appropriate $[\overline{Q}_p]_{B_u, B_u}$ block without having to communicate with the other $m-1$ processors/machines. Once all the $[\overline{Q}_p]_{B_u, B_u}$ rotations have been determined, the matrix $\overline{A}_p$ is reassembled and reblocked according to the next clustering, $B_1^{p+1} \cup B_2^{p+1} \cup \ldots \cup B_m^{p+1}$. Efficiently performing this repeated blocking and reblocking process with minimum communication overhead is a critical component of `pMMF`.

Another important point is that given the size of matrices that `pMMF` is designed for, the $\overline{Q}_p$ matrices or the $[\overline{Q}_p]_{B_u, B_u}$ blocks must not be stored in explicit matrix form. Rather, `pMMF` defines specialized classes for Givens rotations and k–point rotations (consisting of minimal data structures comprising only the indicies and coefficients), and stores $\overline{Q}_p$ as a sequence of these elementary objects.

In many downstream applications of MMF, including preconditioning and other numerical linear algebra tasks, the factorized form of A is used to repeatedly multiply a vector $\boldsymbol{v}$. Once again, this operation is performed by multiplying $\boldsymbol{v}$ by the rotations individually with specialized routines, rather than expressing the rotations in matrix form. Moreover, to parallelize this operations as well, $\boldsymbol{v}$ is blocked just like the $\overline{Q}_p$ matrices.

Design

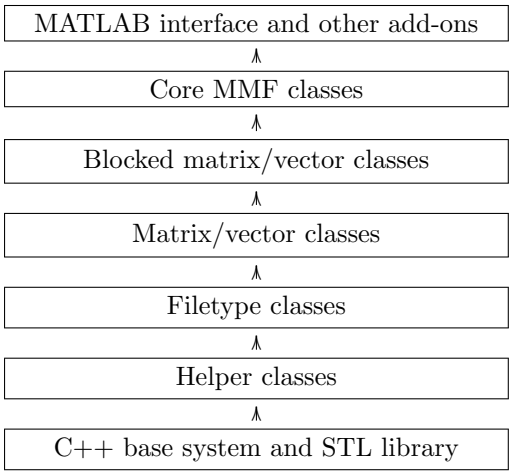
The `pMMF` library is made up of: (a) the `pMMF` base system, comprising all the classes involved in the mechanics of computing MMF factorizations (b) various add-on modules, including, or soon to include code for:

(i) preconditioning and solving linear systems

(ii) visualization

(iii) the MATLAB interface.

The library's object oriented design makes the code highly modular, and easy to adapt to new hardware environments. For example, to port the library to a new type of multiprocessor or GPU architecture, users can simply replace the performance-critical sparse matrix classes with their own hardware specific implementations. The `pMMF` classes form the following hierarchy:



A class at a given level in the hierarchy may only depend on classes at the same or lower levels.

2. Using the library

Installation

pMMF is distributed in C++ source code format. The library makes extensive use of the Standard Template Library and C++11 specific language features, therefore compiling it requires a C++11 compatible compiler, such as `clang`. Newer versions of `gcc` also support C++11, but only on an experimental basis.

Optional dependencies

The pMMF base system is designed to be stand alone software and does not require anything beyond a standard C++11 installation. However, some of the added functionality does use external libraries:

1. The preconditioning functions, and certain other routines require standard linear algebra operations, such as matrix inversion, computing eigenvectors, etc.. To use the `Eigen` template library for these operations, compile the code with the option `_withEigen`. To use `LAPACK`, compile the code with `_withLAPACK`. If neither library is specified, pMMF will still compile, but some of its functionality will be unavailable.
2. To compile pMMF with Matlab file support, make sure that `MatIO` is installed on your system, and compile the library with the option `_withMatIO`.
3. To include support for the Boeing matrix file format, install `hb_io` on your system, and compile the library with the option `_withMatIO`.

Customization

Preprocessor variables

The following user definable preprocessor variables are set in the global header file `pMMFbase.hpp`.

Variable name	Default	Description

`_UTILITYCOPYWARNING`
`_MATRIXCOPYWARNING`
`_BLOCKEDCOPYWARNING`
`_MMFCOPYWARNING`
	undefined	Deep copying/assigning large objects is an expensive operation which, for the most part, should be avoided. If these variables are defined, a warning will be written to `cout` whenever an object of the given category is copied or assigned.

Typedefs

The following user definable typedefs are also set in `pMMFbase.hpp`.

Type name	Default	Description
`FIELD`	`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. Not yet consistently implemented.

Global variables

The following global variables are defined in the global include file `pMMFglobal.inc`, but their value can be changd dynamically, during run time.

Variable name	Default	Description
`int mlog.verbosity`	0	The verbosity level (from 0 to 6).
`bool multithreading`	`true`	Multithreading is disabled if `false`.
`int threadManager.maxthreads`	4	The maximum number of threads that can be simultaneously active.

3. Tutorial examples

The directory `examples` cointains a number of example programs to showcase different features of `pMMF`.

1. Basic class functionality

The example `basic.cpp` uses the `Cmatrix` class (which stands for "C-style dense matrix") to demonstrate some of the basic functionality implemented in almost all `pMMF` classes, including deep copying and move-assignment, loading/saving to binary file, and self-reporting via the overloaded `<<` operator.

```
#include "Cmatrix.hpp"
#include "pMMFglobal.inc" // Include this in all top level executables


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

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

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

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

  Cmatrix B(A); //                                  Construct a copy of A
  cout<<"B="<<endl<<B<<endl; //                               Print out B

  B=A*A; //                                                 Now set B=A*A
  cout<<"B="<<endl<<B<<endl; //                        Print it out again

  B.save("B.bin"); //                     Save B to a file called 'B.bin'

  Bifstream ifs("B.bin"); //        Load it into a new matrix called C
  Cmatrix C(ifs);
  cout<<"C="<<endl<<C<<endl; //                               Print out C

}
```

The output of this code is as follows.

```
A=
[  0.000  1.000  2.000  3.000  ]
[  1.000  2.000  3.000  4.000  ]
[  2.000  3.000  4.000  5.000  ]
[  3.000  4.000  5.000  6.000  ]

WARNING: Cmatrix copied.
B=
[  0.000  1.000  2.000  3.000  ]
[  1.000  2.000  3.000  4.000  ]
[  2.000  3.000  4.000  5.000  ]
[  3.000  4.000  5.000  6.000  ]

B=
[ 14.000 20.000 26.000 32.000  ]
[ 20.000 30.000 40.000 50.000  ]
[ 26.000 40.000 54.000 68.000  ]
[ 32.000 50.000 68.000 86.000  ]

C=
[ 14.000 20.000 26.000 32.000  ]
[ 20.000 30.000 40.000 50.000  ]
[ 26.000 40.000 54.000 68.000  ]
[ 32.000 50.000 68.000 86.000  ]
```

Note the warning generated by the copy constructor in the line `'Cmatrix B(A)'`. This warning can be suppressed by commenting out the definition of the preprocessor variable `_MATRIXCOPYWARNING` in `pMMF.hpp`.

Also note that the assignment `'B=A*A'` does not generate a warning, despite the fact that, at first sight, it also seems to involve copying a matrix object. This is because `A*A` is a temporary (or a so-called rvalue), and so, rather than invoking the regular assignment method `'Cmatrix& operator=(const Cmatrix& y)'`, the compiler invokes the move-assignment operator `'Cmatrix& operator=(Cmatrix&& y)'`, which avoids making an explicit deep copy of `y`, effectively by pilfering its contents. This C++11 construct is extensively used in `pMMF` to avoid unnecessarily copying large objects.

2. A simple MMF computation

The following program, `randomMMF.cpp` computes the MMF of a random symmetric matrix.

```
#include "MMF.hpp"
#include "pMMFglobal.inc"


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

  mlog.verbosity=4;

  Cmatrix A=Cmatrix::RandomSymmetric(12);

  MMF mmf(A,MMFparams::k(2));

}
```

4. The pMMF classes

This section describes the APIs of some of the more important `pMMF` classes. For brevity, not all classes are listed, and not all methods/variables are necessarily listed for each class.

`pMMF` makes extensive use of inheritance. In accordance with C++ terminology, the parent class is referred to as to as the **base class** and the child class is referred to as the **derived class**. A base class whose sole function is to define a common interface for its descendants, and no actual objects can be of that class directly (in particular, because it has "pure virtual" functions), is called an **abstract class**.

`pMMF` also uses **templates**. The template argument is always in uppercase. For example, `VECTOR` stands for a generic vector class, `MATRIX` stands for a generic matrix class, and so on.

An object `x` is said to **own** another object `y` if (a) `x` has the information for accessing `y` in memory; (b) when `x` is deleted, it is responsible for also deleting `y`. As usual in C++, there are two ways that `x` can own `y`: either `y` is explicitly a member variable of `x`, or `x` has a pointer to `y`. Whenever the latter happens, it is explicitly indicated in the class descriptions.

Standard methods

The following standard methods/functions are not listed separately for each class, because they implemented by almost all classes. Here `CLASS` is the name of the class and `x` is the class instance.

CONSTRUCTORS

`CLASS(const CLASS& y)`

Construct a deep copy of the object `y`. Recall that 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`.

`CLASS(CLASS&& y)`

Move-construct `y`. In C++11, `&&` signifies an rvalue reference, so this method is invoked instead of the regular copy constructor when `y` is a temporary, which allows it to "move" each object owned by `y` (i.e., just change their ownership), rather than copying them, potentially resulting in large run-time savings. `pMMF` extensively uses such "move semantics".

DESTRUCTOR

`~CLASS()`

Recursively delete every object owned by `x`, and then delete `x` itself.

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`.

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

DEBUGGING

`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`. Note that, as signified by the `::`, this is a global function, rather than a method of `CLASS`.

Serialization

Most core classes in `pMMF` can load/save their instances from/to files in binary format via the classes `Bifstream` and `Bofstream` using a process called serialization. Serialization is a recursive process, whereby a given object first loads/saves its member variables, then serializes every other object that it owns. All serializable classes are derived from (i.e., children of) the abstract class `Serializable`, and implement each of the following methods.

LOADING

`CLASS (Bifstream& ifs)`
: Construct a new object of class `CLASS` by loading it from `ifs`.

SAVING

`serialize(Bofstream& ofs)`
: Serialize the object to `ofs`.

`save(const char* filename)`
: Save the object to a file named `filename` by serialization.

0. Global objects and variables

The following global objects are defined in the file `pMMFglobal.inc`, which must be `#include`-ed in all top level source files (i.e., `.cpp` files that have a `main` function).

GLOBAL OBJECTS

`Log mlog`
: The object to which status/log messages are written. The level of verbosity is controlled by the variable `mlog.verbosity`. A verbosity level of 0 corresponds to the fewest log messages and 6 corresponds to the most.

`ThreadManager threadManager`
: This class controls the number of queued or active threads. The maxixmum number of threads is set by `threadManager.maxthreads`.

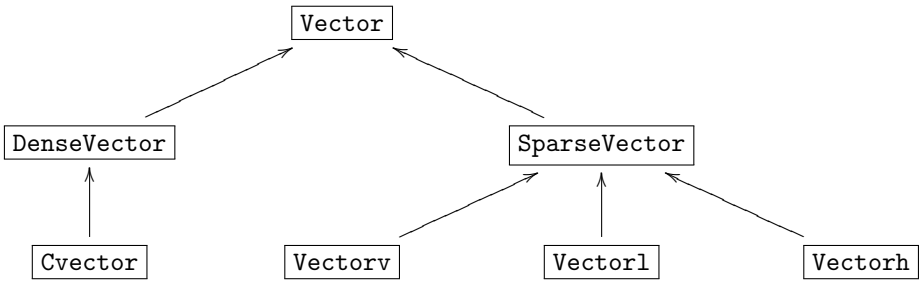
GLOBAL VARIABLES

`bool multithreading`
: Multithreading is enabled if `true`.

1. Vector classes

The abstract class `Vector` provides the generic API for all `pMMF` classes representing vectors in $\mathbb{R}^n$. In addition to the usual basic linear algebra operations, any class derived from `Vector` must have methods for applying Givens rotations and k–point rotations to the vector. `DenseVector` and `SparseVector` are abstract classes that specialize `Vector` to the dense and sparse cases.

The basic dense vector class is `Cvector`, which stores $\boldsymbol{v} \in \mathbb{R}^n$ as a plain C-style array `FIELD[n]`. `Vectorv`, `Vectorl` and `Vectorh` are sparse vector classes which respectively store $\boldsymbol{v}$ using the `std::vector`, `std::list` and `std::unordered_map` Standard Template Library containers.



Vector

`Vector` is the abstract class that defines the common interface to all classes used to represent vectors, $\boldsymbol{v} \in \mathbb{R}^n$. Any class `VECTOR` derived from `Vector` must provide the following constructors and methods.

CONSTRUCTORS

`VECTOR(const int n)`
: A new `n` dimensional vector. Storage is allocated but the entries of $\boldsymbol{v}$ may not be initialized.

NAMED CONSTRUCTORS

`VECTOR::Zero(const int n)`
: The n dimensional zero vector.

`VECTOR::Random(const int n)`
: A random vector in which each component is drawn from the uniform distribution on $[0, 1]$.

ELEMENT ACCESS

`FIELD& operator()(const int i)`
: Returns a reference to the i'th component of $\boldsymbol{v}$. This can be used to both read v_i and to set its value. For sparse vector classes, if there is no (index,value) pair with index i, this method may add a new (index,value) pair, even when used to just read $\boldsymbol{v}$. If this is undesirable, use the `read` method instead.

`FIELD operator()(const int i) const`
: The `const` version of the above, which can be used to read v_i, but not set it. Unlike the previous method, this does create a new (index,value) pair.

`FIELD read(const int i) const`
: Synonym of the above, intended to guarantee side-effect free behavior, even when $\boldsymbol{v}$ is not `const`.

`void foreach(std::function<void(INDEX,FIELD&)> lambda)`
: Applies the function `lambda` to each filled-in entry of $\boldsymbol{v}$.

`bool isFilled(const int i) const`
: Returns `true` if element `i` is filled in (i.e., if the vector contains an (index,value) pair with index `i`). For dense vectors always `true`.

`int nFilled() const`
: The number of filled-in entries of $\boldsymbol{v}$.

SCALAR VALUED OPERATIONS

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

`int argmax() const`
`int argmax_abs() const`
The index of the largest (resp. largest in absolute value) component of $\boldsymbol{v}$. If not unique, then the index of the first (lowest index) maximal component is returned.

`FIELD norm2() const`
The squared ℓ_2–norm $\|\boldsymbol{v}\|^2$.

`FIELD diff2(const VECTORCLASS& x) const`
The squared ℓ_2–norm difference $\|\boldsymbol{v}-\boldsymbol{x}\|^2$.

`FIELD dot(const VECTORCLASS& x) const`
The dot product of $\boldsymbol{v}$ with $\boldsymbol{x}$.

IN-PLACE OPERATIONS

`VECTOR& operator*=(const FIELD c)`
`VECTOR& operator/=(const FIELD c)`
Multiply/divide $\boldsymbol{v}$ by the scalar c.

`VECTOR& operator*=(const Cvector& x)`
`VECTOR& operator/=(const Cvector& x)`
Elementwise multiply/divide $\boldsymbol{v}$ by the vector $\boldsymbol{x}$.

`VECTOR& operator+=(const VECTOR& x)`
`VECTOR& operator-=(const VECTOR& x)`
`VECTOR& operator*=(const VECTOR& x)`
`VECTOR& operator/=(const VECTOR& x)`
Elementwise add/subtract/multiply/divide $\boldsymbol{v}$ by the vector $\boldsymbol{x}$.

`void apply(const GivensRotation& Q)`
`void apply(const KpointRotation& Q)`
`void applyInverse(const GivensRotation& Q)`
`void applyInverse(const KpointRotation& Q)`
Apply the (inverse of the) Givens or k–point rotation Q to $\boldsymbol{v}$.

`VECTOR& add(VECTOR& x, const FIELD c=1)`
Add c times $\boldsymbol{x}$ to $\boldsymbol{v}$. The methods in `MatrixX` implementing rotations from the right use this operation as their primitive, so efficiency is critical.

VARIABLES

`int n`
The dimension n.

Cvector

The plain vanilla C-style dense vector that stores its entries in a simple array of type `FIELD[n]`. The interface of `Cvector` is inherited from `Vector` via `DenseVector`.

Derived from: `DenseVector, Serializable`

VARIABLES

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

SparseVector

`MMFc` has three different classes to represent sparse vectors: `Vectorv`, which represents $\boldsymbol{v}$ as an unordered list of (i, v_i) pairs, implemented as an `std::vector` container; `Vectorl`, which represents $\boldsymbol{v}$ as an ordered list of (i, v_i) pairs, implemented as an `std::list` container; and `Vectorh`, which represents $\boldsymbol{v}$ as a hash map, implemented with `std::unordered_map`. `SparseVector` provides the common interface to these three classes.

Derived from: `Vector`

METHODS

`virtual void insert(const int i, const FIELD x)`
: Add (i, x) to the set of (index, value) pairs without checking whether a pair with index i exists already.

`virtual void append(const int i, const FIELD x)`
: Add (i, x) to the set of (index, value) pairs without checking whether a pair with index i exists already. The difference to `insert` is that for list/vector based implementations, the new pair is added at the end, and it is assumed that this does not violate the index-based ordering of the (index,value) pairs.

`virtual void zero(const int i)`
: Set $v_i = 0$ and remove the pair $(i, 0)$ from the list, if practicable, to increase sparsity. Currently, `SparseVectorl` and `SparseVectorh` remove such zeroed entries, but `SparseVectorv` does not.

`virtual void sort()`
: Certain operations, such as multiplying two sparse vectors together, involve traversing both vectors in parallel. In the case of ordered containers (such as `std::list` and `std::vector`) this is much faster when the (i, v_i) pairs are sorted by index. This method does the sorting. Vector classes that implement this function, such as `Vectorv`, maintain a flag, `sorted`, that signals whether the the list is currently in a sorted state. For example, the non–`const operator()` member function and the `insert` member function can destroy the ordering, so these revert `sorted` to `false`. On the other hand, `append` does not change the value of this flag.

Vectorv

A concrete sparse vector class that stores $\boldsymbol{v}$ as an `std::vector` of `SVpair` objects. `Vectorv` inherits its public interface from the `Vector` and `SparseVector` classes.

Derived from: `SparseVector, std::vector<SVpair>, Serializable`

Vectorl

A concrete sparse vector class that stores $\boldsymbol{v}$ as an `std::list` of `SVpair` objects. `Vectorv` inherits its public interface from the `Vector` and `SparseVector` classes.

Derived from: `SparseVector, std::list<SVpair>, Serializable`

Vectorh

A concrete sparse vector class that stores $\boldsymbol{v}$ as an `std::unordered_map`. `Vectorv` inherits its public interface from the `Vector` and `SparseVector` classes.

Derived from: `SparseVector, std::unordered_map<INDEX,FIELD>, Serializable`

SVpair

A helper class for holding the (i, v_i) pairs in `Vectorv` and `Vectorl` (but not `Vectorh`).

CONSTRUCTORS

`SVpair(const INDEX& index, const FIELD& value)`
: Construct a new (i, v_i) pair.

VARIABLES

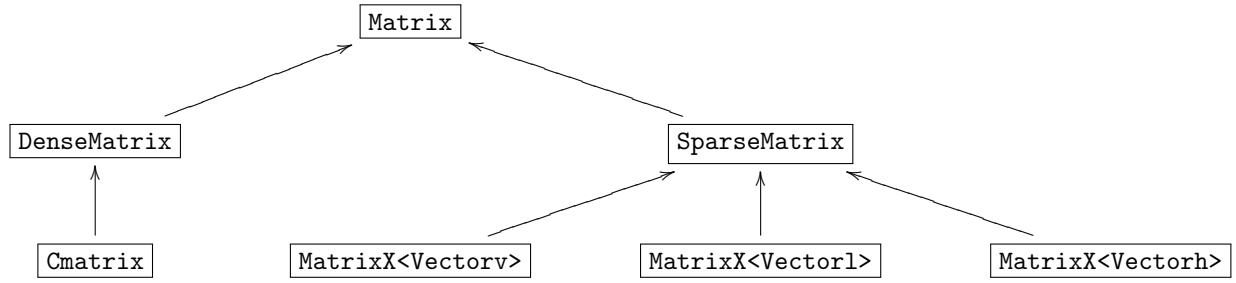
`INDEX first` The index i.

`FIELD second` The value v_i.

2. Matrix classes

The matrix classes form a similar hierarchy to the vector classes. The abstract class `Matrix` provides the generic API for all `pMMF` classes representing real matrices, $M \in \mathbb{R}^{n \times m}$. In addition to the usual basic linear algebra operations, any class derived from `Matrix` must support a number of specialized operations needed in the process of computing MMF factorizations, including fast routines for multiplying matrices on the left and the right by (the transposes of) Givens rotations and k–point rotations. `DenseMatrix` and `SparseMatrix` are abstract classes that specialize `Matrix` to the dense and sparse cases.

The basic dense matrix class is `Cmatrix`, which stores M as a C-style array `FIELD[n*m]`. The basic sparse matrix class is the template class `MatrixX<COLUMNTYPE>`, which stores M as a collection of m vectors of type `COLUMNTYPE`, where `COLUMNTYPE` can be any of the sparse matrix classes, such as `Vectorv`, `Vectorl` or `Vectorh`.



Matrix

`Matrix` is the abstract class that defines the common interface to all classes that represent matrices, $M \in \mathbb{R}^{n \times m}$. All classes derived from `Matrix` must implement the following methods and constructors. `MATRIX` stands for the concrete matrix type derived from `Matrix`.

CONSTRUCTORS AND I/O

`MATRIX(const int n, const int m)`
: A new $n \times m$ matrix. Storage is allocated but the entries of M may not be initialized.

`MATRIX(SparseMatrixFile& file)`
`MATRIX(DenseMatrixFile& file)`
: Load M from the file `file`. This method is intended for interfacing to a number of different standard matrix formats, and is distinct from the serialization mechanism. Deprecated.

`MATRIX(MatrixIF& file)`
: Load M from the file `file`. This method is intended for interfacing to a number of different standard matrix formats, and is distinct from the serialization mechanism.

`saveTo(MatrixOF& file) const`
: This method is intended for interfacing to a number of different standard matrix formats, and is distinct from the serialization mechanism.

NAMED CONSTRUCTORS

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

`MATRIX::Identity(const int n)`
: The n dimensional identity matrix.

`MATRIX::Random(const int n, const int m)`
`MATRIX::RandomSymmetric(const int n)`
: An $n \times m$ random matrix or $n \times n$ random symmetric matrix with Uniform$(0, 1)$ entries.

ELEMENT ACCESS

`FIELD& operator()(const int i, const int j)`
`FIELD operator()(const int i, const int j) const`
`FIELD read(const int i, const int j) const`
: (a) Return a reference to the (i, j) element of M. As with `Vector`, if M is sparse and $M_{i,j}$ is not filled in, then a new entry is created with $M_{i,j} = 0$. (b) Return the *value* stored at (i, j). If M is in sparse format and $M_{i,j}$ is not filled in, return 0. (c) Equivalent to (b).

`void foreach(std::function<void(int,int,FIELD)> lambda)`
: Apply the function `lambda(int i, int j, FIELD& val)` to each filled in entry of M.

`void foreach_in_column(const int j, std::function<void(INDEX,FIELD)> lambda)`
Apply the function `lambda(int i, FIELD& val)` to each filled in entry in the j'th column of M.

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

`int nFilled() const`
The number of filled-in entries in M.

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

SCALAR VALUED OPERATIONS

`int nnz() const`
The number of non-zero matrix entries of M.

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

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

VECTOR VALUED OPERATIONS

`VECTOR operator*(const VECTOR& v)`
Compute the matrix/vector product $M\boldsymbol{v}$.

`VECTOR dot(const VECTOR& v)`
Compute the matrix/vector product $M^\top \boldsymbol{v}$.

MATRIX VALUED OPERATIONS

`MATRIX operator*(const MATRIX& X)`
Compute the matrix/matrix product MX.

`MATRIX dot(const MATRIX& X)`
Compute the matrix/matrix product $M^\top X$.

IN-PLACE METHODS

`MATRIX& operator+=(const MATRIX& X)`
`MATRIX& operator-=(const MATRIX& X)`
Set M to resp. $M + X$ or $M - X$.

`MATRIX& MultiplyRowsBy(const Cvector& v)`
`MATRIX& DivideRowsBy(const Cvector& v)`
Multiply/divide the i'th row of M by v_i.

`MATRIX& MultiplyColsBy(const Cvector& v)`
`MATRIX& DivideColsBy(const Cvector& v)`
: Multiply/divide the j'th column of M by v_j.

`void applyFromLeft(const GivensRotation& Q)`
`void applyFromLeft(const KpointRotation& Q)`
`void applyFromLeftT(const GivensRotation& Q)`
`void applyFromLeftT(const KpointRotation& Q)`
: Multiply M from the left by Q or $Q^\top$, respectively.

`void applyFromRight(const GivensRotation& Q)`
`void applyFromRight(const KpointRotation& Q)`
`void applyFromRightT(const GivensRotation& Q)`
`void applyFromRightT(const KpointRotation& Q)`
: Multiply M from the right by Q or $Q^\top$, respectively.

VARIABLES

`int nrows`
: The number of rows, n.

`int ncols`
: The number of columns, m.

Cmatrix

The plain vanilla dense matrix class that stores M as a C-style array `FIELD[n*m]` in column major order.

Derived from: `DenseMatrix, Serializable`

METHODS

`Cvector operator()(const Cvector& v)`
: Compute the product Mv.

`Cvector::Virtual column(const int j) const`
: Return a virtual copy of column j.

VARIABLES

`FIELD* array`
: A column major array holding the matrix entries.

MatrixX<COLUMNTYPE>

A generic column–based sparse matrix class that represents $M \in \mathbb{F}^{n \times m}$ as a collection of m sparse vectors.

Derived from: `SparseMatrix, Serializable`
Owned objects: The sparse vectors of type `COLUMNTYPE` storing each column.

METHODS

`void applyFromLeft(const GivensRotation& Q)`
`void applyFromLeft(const KpointRotation& Q)`
Multiply M from the left by `Q`. Since each column is multiplied by Q independently, the implementation is relegated to the `COLUMNTYPE` class.

`void applyFromRightT(const GivensRotation& Q)`
`void applyFromRightT(const KpointRotation& Q)`
Multiply M from the right by the transpose of `Q`. Since this involves mixing columns, these functions are implemented in the present class and not passed down to `COLUMNTYPE`.

`COLUMNTYPE operator*(const COLUMNTYPE& v)`
Compute the product Mv.

VARIABLES

`vector<COLUMNTYPE*> column`
Pointers to the columns.

3. Blocked vector/matrix classes

The "blocked vector" and "blocked matrix" data structures are critical for parallelizing MMF. In `pMMF`, they are implemented via the `BlockedVector<VECTOR>` and `BlockedMatrix<MATRIX>` template classes, where `VECTOR` can be any class derived from `Vector` and `MATRIX` can be any class derived from `Matrix`. For good performance on parallel architectures it is important to be able to distribute the blocks across multiple processors with minimum communication overhead. The critical operation in pMMF from this point of view is the reblocking from one stage to the next. `BlockedRemap` is a specialized class designed for this purpose.

Note that, at least in the present version of `pMMF`, `BlockedVector<VECTOR>` is not a derived type of `Vector`, and `BlockedMatrix<MATRIX>` is not a derived type of `Matrix`. Therefore, the blocking construction cannot be applied recursively.

BlockedVector<VECTOR>

A vector $\boldsymbol{v}$ consisting of N blocks, where each block is a vector of type `VECTOR`. `VECTOR` can be any dense or sparse vector class derived from the abstract class `Vector`.

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

CONSTRUCTORS

`BlockedVector<VECTOR>(const int _nblocks)`
Construct a placeholder for a blocked vector in which each element of `block` is `nullptr`.

```
BlockedVector<VECTOR>(const BlockedVector<VECTOR>& x, const BlockedRemap& map,
                      const bool inverse=false)
```
Construct a new `BlockedVector` from `x` by remapping its entries according to `map`. When `inverse` is `true`, the inverse remapping is applied.

NAMED CONSTRUCTORS

`BlockedVector<VECTOR>::Zero(Bstructure& structure)`
`BlockedVector<VECTOR>::Random(Bstructure& structure)`
Construct a blocked vector with block structure `structure` with (a) all entries initialized to 0; (b) each entry initialized to a random number in $[0, 1]$.

METHODS

`FIELD& operator()(const int I, const int i)`
`FIELD operator()(const int I, const int i) const`
`FIELD read()(const int I, const int i) const`
Return a reference to the i'th entry in the I'th block or return the actual value. The semantics is the same as of the analogous methods in `Vector` and `Matrix`.

`int nnz() const`
The number of non-zero elements of $\boldsymbol{v}$.

VARIABLES

`int nblocks`
The number of blocks, N.

`VECTOR** block`
An array of pointers to the individual blocks.

BlockedMatrix<MATRIX>

A matrix M consisting of $n \times m$ blocks, where each block is a matrix of type `MATRIX`. `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: `Serializable`
Owned objects: The individual blocks pointed to by the elements of the array `block`.

CONSTRUCTORS

`BlockedMatrix<MATRIX>(const int _nstreets, const int _ntowers)`
Construct a placeholder for a blocked matrix, in which each element of `block` is `nullptr`.

`BlockedMatrix(const BlockedMatrix<MATRIX>& X, const BlockedRemap& rmap, Identity(), bool inverse)`
`BlockedMatrix(const BlockedMatrix<MATRIX>& X, Identity(), const BlockedRemap& cmap, bool inverse)`
`BlockedMatrix(const BlockedMatrix<MATRIX>& X, const BlockedRemap& rmap, const BlockedRemap& cmap, bool inverse)`

Construct a new blocked matrix from `X` by (a) remapping its rows by `rmap`; (b) remapping its columns by `cmap`; (c) remapping both its rows and columns. Whether the pull or push method is used depends on `MATRIX`, so these constructors should only be used when the mappings are bijections. When `inverse` is `true`, the inverse mapping is applied. The default is `false`.

NAMED CONSTRUCTORS

`BlockedMatrix<MATRIX>::Zero(const BlockStructure& rst, const BlockStructure& cst)`
An all-zeros blocked matrix with block structure `rst`×`cst`.

`BlockedMatrix<MATRIX>::Identity(const BlockStructure& st)`
An identity blocked matrix with block structure `st`.

`BlockedMatrix<MATRIX>::Random(const BlockStructure& rst,const BlockStructure& rst)`
`BlockedMatrix<MATRIX>::RandomSymmetric(const BlockStructure& st)`
(a) a random blocked matrix with with block structure `rst`×`cst`, (b) a random symmetrix blocked matrix with with block structure `st`×`st`.

ELEMENT ACCESS

`FIELD& operator()(const int I, const int i, const int J, const int j)`
`FIELD operator()(const int I, const int i, const int J, const int j) const`
`FIELD read(const int I, const int i, const int J, const int j) const`
Return a reference to the (i, j) entry in the (I, J) block or return the actual value. The semantics is the same as for the `Vector` and `Matrix` classes.

`bool isFilled(const int I, const int i, const int J, const int j)`
If `MATRIX` is a sparse matrix type, `true` if element `((I,i),(J,j))` is filled in. If `MATRIX` is a dense matrix type, always `true`.

`Street<MATRIX> virtual_street(const int I)`
`const Street<MATRIX> virtual_street(const int I) const`
Return a virtual `Street` consisting of the I'th row of blocks.

`Tower<MATRIX> virtual_tower(const int J)`
`const Tower<MATRIX> virtual_tower(const int J) const`
Return a virtual `Tower` consisting of the J'th column of blocks.

VECTOR VALUED OPERATIONS

`BlockedVector<VECTOR> operator*(const BlockedVector<VECTOR> v) const`
Compute $M\boldsymbol{v}$. The block structure of $\boldsymbol{v}$ must be the same as the column structure of M.

MATRIX VALUED OPERATIONS

`BlockedMatrix<MATRIX> operator*(const BlockedVector<MATRIX> B) const`
Compute MB. The row structure of B must be the same as the column structure of M.

VARIABLES

`int nstreets` n

`int ntowers` m

`MATRIX** block`
An $n \times m$ column major array of pointers to the individual blocks.

Street<MATRIX>

A special kind of blocked matrix consisting of $1 \times m$ blocks.

Owned objects: The individual blocks pointed to by elements of the array `block`.

CONSTRUCTORS

`Street(const int nblocks, const int nrows)`
Construct a placeholder `Street`, where each element of the array `block` is `nullptr`.

`Street(const Street<MATRIX>& X, const BlockedRemap& cmap, const bool inverse=false)`
Construct a new `Street` from `X` by remapping its columns by `cmap`. Whether the pull method or push method is used might depend on `MATRIX`, so this constructor should only be used when `cmap` is a bijection. When `inverse` is `true`, the inverse mapping is applied.

METHODS

`Street<MATRIX> pullColumns(const Street<MATRIX>& X, const BlockedRemap& cmap, bool inverse=false)`
Similar to the remapping constructor above, except guaranteed to use the pull method, therefore `cmap` (or its inverse) need only be surjective.

IN-PLACE OPERATIONS

`void multiplyRowsBy(const Cvector& v)`
Mulitply row i of each block by v_i.

`applyFromLeft(const ROTATION& Q)`
`applyFromLeftT(const ROTATION& Q)`
Apply (the transpose of) Q to each block from the left.

VARIABLES

`int nrows`
The number of rows in each matrix making up the street.

`int nblocks`
The number of blocks, m.

`MATRIX**`
An array of pointers to the blocks.

Tower<MATRIX>

A special kind of blocked matrix consisting of $N \times 1$ blocks.

Owned objects: The individual blocks pointed to by the elements of `block`.

CONSTRUCTORS

`Tower(const int nblocks, const int ncols)`
Construct a placeholder `Tower`, in which each block is `NULL`.

`Tower(const Tower<MATRIX>& X, const BlockedRemap& rmap, const inverse=false)`
Construct a new `Tower` from `X` by remapping its rows by `rmap`. Whether the pull method or push method is used might depend on `MATRIX`, so this constructor should only be used when `rmap` is a bjection. When `inverse` is `true`, the inverse mapping is applied.

IN-PLACE OPERATIONS

`void multiplyColsBy(const Cvector& v)`
Mulitply column i of each block by v_i.

`void applyFromRight(const ROTATION& Q)`
`void applyFromRightT(const ROTATION& Q)`
Apply (the transpose of) Q to each block from the right.

VARIABLES

`int ncols`
The number of columns in each block.

`int nblocks`
The number of blocks, N.

`MATRIX**`
An array of pointers to the blocks.

BlockedRemap

The map between the rows (or columns) of one blocked matrix or vector object and the rows (or columns) of another. The map need not be a bijection.

CONSTRUCTORS

`BlockedRemap(const int _nsource, const int _ndest)`
Construct a placeholder object in which the `forward` and `backward` arrays consist of `NULL` pointers.

METHODS

`BlockIndexPair& operator()(const int I, const int i)`
Return a reference to the `BlockIndexPair` object describing where (I, i) is mapped.

`BlockIndexPair& inv(const int J, const int j)`
Return a reference to the `BlockIndexPair` object describing what is mapped to (J, j).

`void set(const int I, const int i, const int J, const int j)`
Set the `forward` and `backward` maps so that (I, i) is mapped to (J, j).

VARIABLES

`int nsource` The number of blocks in the domain of the mapping.

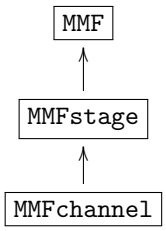
`int ndest` The number of blocks in the range of the mapping.

`vector<BlockIndexPair>** forward`
The I'th element in this array (of size `nsource`) is a pointer to a `vector`, in which the i'th entry specifies where (I, i) is mapped.

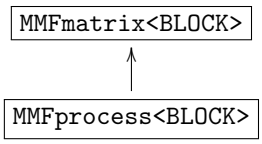
`vector<BlockIndexPair>** backward`
The J'th element in this array (of size `ndest)` is a pointer to a `vector`, in which the j'th entry specifies what is mapped to (J, j).

4. The MMF classes

This section describes the classes that do the "heavy lifting" of computing MMF factorizations, chief amongst them, the class `MMF`, each instance of which holds the actual MMF factorization of some matrix A. An `MMF` objects owns a number of `MMFstage` objects, each one corresponding to a single stage of the factorization. Each `MMFstage` object, in turns, owns a number of `MMFchannel` objects, each one corresponding to a single cluster of rows/columns (one of the B_u^p's in the partitioning $B_1^p \cup B_2^p \cup \ldots \cup B_m^p$). This nomenclature is inspired by the flow of data when a matrix $\boldsymbol{v}$ by an MMF factorization (see the `MMF::operator*(const VECTOR& )` method): at each stage the vector is reblocked, and each of its blocks is sent through a different "channel", where each of the rotations in the corresponding $[\overline{Q}_p]_{B_u,B_u}$ submatrix are applied to it.



The process of building MMF factorizations requires two more classes. The current active matrix $\overline{A}_p$ is stored in an `MMFmatrix<MATRIX>` object, and each cluster of rows/columns corresponds to an `MMFprocess<MATRIX>` object.



The routines involved in finding the MMF rotations, applying them to $\overline{A}_p$, etc., are implemented in these two classes. The naming of `MMFprocess` reflects the fact that the computations involved in each `MMFprocess` can be carried out in parallel to the others.

MMF

The object representing an entire MMF factorization $\tilde{A} = \overline{Q}_1^\top \dots \overline{Q}_P^\top H Q_P \dots \overline{Q}_1$. The factorization is computed by calling the appropriate constructor of this class, with the matrix `A` and the parameter helper object `params` as arguments. Once the factorization has been computed (or loaded from file), it can be applied to vectors, matrices, etc., to various ends.

Derived from: `Serializable`
Owned objects: The `MMFstage` objects corresponding to each stage, and the core matrix `finalA`.

CONSTRUCTORS

`MMF(const MATRIX& A, const MMFparams& params)`
`MMF(const MMFmatrix<BLOCK>& A, const MMFparams& params)`
Compute the MMF of the matrix A with the parameters in `params`. This constructor is the workhorse of the entire library.

METHODS

`WaveletTransform<VECTOR> transform(const BlockedVector<VECTOR>& v)`
Compute the MMF wavelet transform of the vector $\boldsymbol{v}$.

`BlockedVector<VECTOR> inverseTransform(const Wtransform<VECTOR>& W)`
Compute $\boldsymbol{v}$ from its MMF wavelet transform W.

`VECTOR operator*(const VECTOR& v)`
`BlockedVector<VECTOR> operator*(const BlockedVector<VECTOR>& v)`
Apply the MMF to the (blocked) vector $\boldsymbol{v}$.

`VECTOR hit(const VECTOR& v)`
`BlockedVector<VECTOR> hit(const BlockedVector<VECTOR>& v)`
Synonym of the `operator*` methods.

`FIELD diffSpectral(const BlockedMatrix<BLOCK>& M)`
Approximate the spectral norm of $\tilde{A} - M$.

`void invert(FIELD eps=0)`
`void invertSqrt(FIELD eps=0)`
(a) Invert the MMF, i.e., convert it to $\tilde{A}^{-1}$. (b) Convert the MMF to $\tilde{A}^{-1/2}$. To avoid numerical overflow, any entries on the diagonal of H less than `eps` in absolute value will be set to 0.

VARIABLES

`vector<MMFstage*> stage`
The first `MMFstage` is a dummy stage that contains no rotations and just clusters the rows/columns of A. The rest of the `MMFstage` objects are the actual stages of the factorization.

`BlockedMatrix<Cmatrix> core`
The final core matrix.

MMFparams

`MMFparams` is a helper class that stores the parameters needed to compute MMF factorizations. `MMFparams` objects are typically constructed using the `MMFparams::k(const int _k)` named constructor that sets the order of rotations parameter, k. The signature of the parameter setting methods allows the rest of them to be daisy-chained. For example,

```
MMFparams params=MMFparams::k(2).nsparsestages(5).nclusters(7).fraction(0.3);
```

constructs an `MMFparams` object in which $k = 2$, the number of sparse stages is 5, the nominal number of clusters in each stage is 7, and so on.

NAMED CONSTRUCTORS

`MMFparams::k(const int _k)`
: Construct a new `MMFparams` object with k set to `_k`.

METHODS

`MMFparams& nsparsestages(const int x)` The number of sparse stages in the factorization.

`MMFparams& ndensestages(const int x)` The number of dense stages in the factorization.

`MMFparams& nclusters(const int x)` The target number of clusters in each stage of the factorization.

`MMFparams& minclustersize(const int x)` The lower bound on the number of rows/columns in each cluster. Any clusters that have fewer rows/columns will get merged in the clustering process.

`MMFparams& minclustersize(const int x)` The upper bound on the number of rows/columns in each cluster. Any clusters that have more rows/columns will get split in the clustering process, `maxclusterdepth` permitting.

`MMFparams& maxclusterdepth(const int x)` The maximum depth of the recursive cluster refinement process.

`MMFparams& fraction(const double f)`

`MMFparams& bypass`

`MMFparams& prenormalize`

`MMFparams& ncoreclusters`

`MMFparams& dcore`

`MMFparams& selection_normalize`

`MMFparams& n_eliminate_per_rotation`

`MMFparams& fraction_eliminate_after`

`MMFparams& selection_criterion`

MMFstage

The object corresponding to a given MMF stage $\overline{Q}_p = Q_{\ell_{p-1}+1} Q_{\ell_{p-1}+1} \dots Q_{\ell_p}$.

Derived from: `Serializable`
Owned objects: The `MMFchannel` objects corresponding to each channel of rotations, and the `remap`.

CONSTRUCTORS

`MMFstage(const MMFmatrix<BLOCK>& M)`
: Construct an `MMFstage` for `M`. All the `MMFchannel` objects are constructed and the rotations are copied over from the `MMFprocess` objects. However, `remap` is kept `NULL`.

METHODS

`pair<BlockedVector<VECTOR>*,VECTOR*> transform(BlockedVector<VECTOR>& v)`
: Apply the rotations $Q^\top_{\ell_{p-1}+1}, Q^\top_{\ell_{p-1}+1} \dots Q^\top_{\ell_p}$ to $\boldsymbol{v}$. The first element of the returned pair is a pointer to the scaling space part of the resulting vector, structured according to the block structure of the next stage. The second element of the returned pair is a pointer to the wavelet space part. Warning: this method operates directly on `v`, therefore, if `v` is not to be modified, first a copy should be made.

`BlockedVector<VECTOR>* inverseTransform(const BlockedVector<VECTOR>& v, const VECTOR& w)`
: Merges v and w into a single blocked vector structured according to the block structure of this stage, and then applies the rotations $Q_p^{r_p\top}$, $Q_p^{r_p-1\top}$, …, $Q_p^{1\top}$.

`BlockedMatrix<CstyleDense>* matrix()`
: Return the matrix form of this stage.

VARIABLES

`nchannels` The number of channels.

`vector<MMFchannel*> channel` The channels constituting this stage.

`BlockedRemap remap`
: The remapping applied to vectors after the rotations. The last block of the remapped vector is the wavelet space part. The other blocks constitute the scaling space part.

`CstyleDenseVector freq` The wavelet frequencies (diagonal elements of the eliminated rows/columns).

MMFchannel

A single channel of a given MMF stage. Mostly just a container for the rotations.

Derived from: `Serializable`

CONSTRUCTORS

`MMFchannel(const int n)`
Create a new MMFchannel with n rows/columns.

`MMFchannel(const int n, Random(nrot), const int k=2)`
Create new random MMFchannel with `n` rows/columns and `nrot` random `k`'th order rotations.

`MMFchannel(const MMFprocess<BLOCK>& X)`
Create a new channel by copying the rotations from `X`.

METHODS

`void applyTo(VECTOR& v)`
`void applyInverseTo(VECTOR& v)`
Apply (the transpose of) the rotations in this channel to $\boldsymbol{v}$.

`void applyFromLeftTo(Matrix& M)`
`void applyFromLeftToT(Matrix& M)`
`void applyFromRightTo(Matrix& M)`
`void applyFromRightToT(Matrix& M)`
Apply (the transposes of) the rotations in this channel from the left/right to the matrix M.

VARIABLES

`n`
The number of rows/columns in this channel.

`vector<GivensRotation*> givens`
The list of Givens rotations in this channel.

`vector<KpointRotation*> kpoint`
The list of k–point rotations in this channel.

`bool normalized=false`
`Cvector normalizer`
When `normalized` is `true`, tow/column i is multiplied by the i'th element of the normalizer vector before any rotations take place.

MMFmatrix<MATRIX>

An MMFmatrix<MATRIX> $\overline{A}_p$ is a BlockedMatrix<MATRIX> with extra functionality. The methods needed to compute a given stage of MMF are implemented here and in the related class MMFprocess<MATRIX>.

Derived from: BlockedMatrix<MATRIX>
Owned objects: The MMFprocess objects corresponding to each channel.

CONSTRUCTORS

`MMFmatrix(const MATRIX M)`
Construct a single channel MMFmatrix from a single unblocked matrix M.

`MMFmatrix(const MMFmatrix& X, const BlockedRemap& map)`
Create a new MMFmatrix by compressing and reblocking the previous one, X, by map. This is how $\overline{A}_{p+1}$ is constructed from $\overline{A}_p$.

`MMFmatrix(const BlockStructure& structure, const Random& dummy)`
Create a random MMFmatrix with row/column block structure structure.

METHODS

`void findRotations(const int nrot, const int k=2)`
`void findRotations(const double frac, const int k=2)`
Compute nrot number of MMF rotations in each channel. If $k = 2$, then the rotations will be Givens rotations. Alternatively, compute a fraction frac of rotations for each channel.

`void applyRotations(const bool offdiag=false)`
Conjugate A by applying the rotations in each process from the left and the right. When offdiag is false, the diagonal blocks are spared.

`BlockedRemap computeCompressionMap(const int nclusters)`
Compute the remapping that separates out the wavelet coordinates and reclusters the rest according to a randomized greedy procedure.

VARIABLES

`int nchannels` The number of parts that the rows/columns of $\overline{A}_p$ are clustered into.

`vector<MMFprocess<MATRIX>*> MMFprocess`
The MMFprocess objects corresponding to each channel.

MMFprocess<MATRIX>

Each channel of an `MMFmatrix` has a corresponding `MMFprocess`, which is responsible for computing the actual rotations.

CONSTRUCTORS

`MMFprocess(const VirtualTower<MATRIX>& tower, const int chanl)`
: Initialize the process corresponding to channel number `chanl`. `tower` is the corresponding tower of blocks in the `MMFmatrix`.

METHODS

`void randomGreedyGivens(const int nrot)`
`void randomGreedyGivens(const double frac)`
: Use the randomized greedy method to find `nrot` Givens rotations or the number of rotations that is `frac` fraction of the number of rows/columns in this channel.

`void randomGreedyKpoint(const int nrot, const int k)`
`void randomGreedyGivens(const double frac, const int k)`
: Use the randomized greedy method to find `nrot` k–point rotations or the number of rotations that is `frac` fraction of the number of rows/columns in this channel.

`void applyFromLeftTo(MATRIX& M)`
`void applyFromRightToT(MATRIX& M)`
: Apply the rotations in this process to `M` from the left resp. the transpose of the rotations from the right. These are the methods use to conjugate the off-diagonal blocks of $\overline{A}_p$.

PRIVATE METHODS

`void doGivens(const int i1, const int i2)`
: Find the optimal rotation angle and perform a single Givens rotation on coordinates `i1` and `i2`.

`void doKpoint(const TopKlist I)`
: Find the optimal $k \times k$ rotation matrix and perform a single k-point rotation on the coordinates in `I`.

`removeFromActiveSet(const int r)`
: Remove row/column `r` from the active set, modify `gram` accordingly, and update `cumulativeError`.

VARIABLES

`VirtualTower<MATRIX> tower`
: The virtual tower comprising the blocks that make up this channel.

`int nactive`
: The number n_{act} of active rows/columns left in this channel.

`Remap activemap`
: A permutation that maps $\{0, 1, \ldots, n_{\text{act}} - 1\}$ to the indices of the presently active rows/columns. This is used for efficiently finding random active rows/columns.

`vector<bool> activeflag`
: Flags to indicate whether the i'th row/column is active.

`Cmatrix* gram`
: The matrix of inner products between any pair of columns in this channel.

`FIELD cumulativeError`
: The total energy of the off-diagonal parts of the eliminated columns.

`vector<FIELD> WaveletFreqs`
: The diagonal entries of the rows/columns of A that are eliminated.

WaveletTransform<VECTOR>

The MMF wavelet transform W of a vector $\boldsymbol{v}$. The wavelet transform is stored as a sequence of vectors corresponding to each stage of the factorization, plus a final stage corresponding to the core.

Derived from: `Serializable`

CONSTRUCTORS

`WaveletTransform<VECTOR>(const int nstages)`
: Create a new `Wtransform` object with all components of `w` initialized to `nullptr`.

VARIABLES

`BlockedVector<VECTOR> v`
: The scaling space part of the wavelet transform.

`vector<VECTOR*> w`
: The vectors of wavelet coefficients derived from each stage of the MMF.

5. Helper classes

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.

Derived from: `ElementaryRotation, Serializable`

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.

VARIABLES

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

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

KpointRotation

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

Derived from: `ElementaryRotation, Serializable`

CONSTRUCTORS

`KpointRotation(const int k)`
: A new k–point rotation, where the arrays `ix` and `q` allocated but unitialized.

`KpointRotation(const int k, Identity())`
: A new k–point rotation initialized to the identity.

METHODS

`CstyleDenseMatrix* matrix() const`
: Return the $k \times k$ orthogonal matrix corresponding to this rotation.

VARIABLES

`k`

`int* ix`
: The array of indices $i_1, \ldots, i_k$.

`double* q`
: The $k \times k$ rotation matrix stored as an array in column major format.

Log

The logging class.

CONSTRUCTORS

`Log()`
: Start the clock.

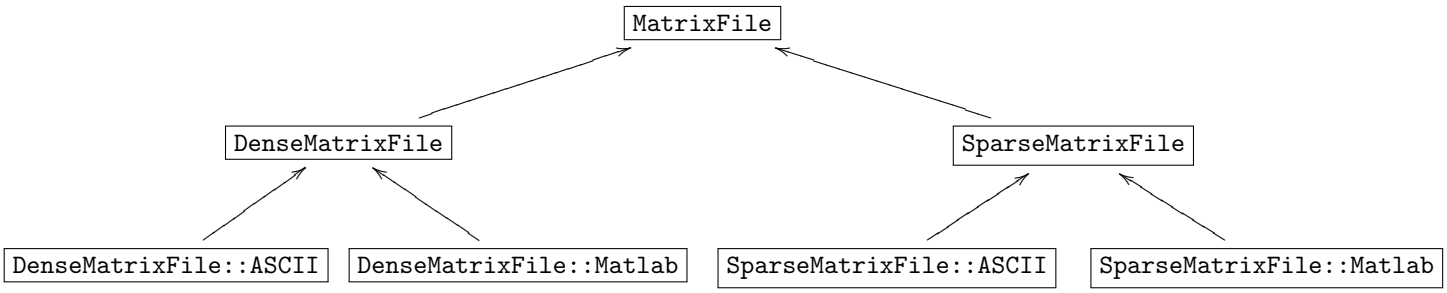
METHODS

`Log& operator<<(const char* s)`
: Print the message `s` to the log together with the current time.

`void startClock(const int i=0)`
: Reset the clock.

6. Filetype classes

The purpose of the "filetype" classes is to provide a common interface to loading/saving matrices in a variety of file formats. The routines for loading/saving from/to files in specific formats are implemented in the classes derived from the generic API `MatrixFile`.



To load a matrix from file, for example, a matrix of type `MatrixX<Vectorv>` from a sparse Matlab matrix file named `M1.mat`, write

```
SparseMatrixFile::Matlab mfile(M1.mat); \\   Open the file M1.mat for read
MatrixX<Vectorv> M(file); \\                 Construct M from mfile
```

The file is closed automatically when `mfile` goes out of scope. To save `M` in a new Matlab file called `M2.mat`, write

```
SparseMatrixFile::Matlab mfile(M2.mat, M);
```

MatrixFile

`MatrixFile` is the abstract class that defines the API for all matrix file classes. All `MATRIXFILE` classes derived from `MatrixFile` must implement the following methods.

CONSTRUCTORS

`MATRIXFILE(const char* filename)`
Open the matrix file called `filename` for read and determine `nrows` and `ncols`.

`MATRIXFILE(const char* filename, const MATRIX& M)`
Save the matrix M to a new file called `filename`. `MATRIX` can be any class derived from `Matrix`.

READ METHODS

`MATRIXFILE& operator>>(FIELD& x)`
Read the next matrix element from the file.

`MATRIXFILE& operator>>(IndexValueTriple& t)`
Read the next $(i, j, M_{i,j})$ triple into `t`.

WRITE METHODS

VARIABLES

`int nrows` The number of rows of the matrix.

`int ncols` The number of columns.

`ifstream ifs` The file stream object used for input files.

Bibliography

[1] Risi Kondor, Nedelina Teneva, and Vikas K. Garg. Multiresolution matrix factorization. 2014.

[2] Risi Kondor, Nedelina Teneva, and Pramod K. Mudrakarta. Parallel MMF: a multiresolution approach to matrix computation. *http://arxiv.org/abs/1507.04396*, pages 1–9, 2015.

[3] The GNU Public License, Version 3, http://www.gnu.org/licenses/, 2007.