Partensor

Generated by Doxygen 1.9.3

Chapter 1

Chapters

Here is a list of all modules.

- Factorization
- Completion
- Matrix and Tensor Operations
- Read/Write Data from/in Files
- Timers

1.1 Factorization

This is a user guide of how to use `Partensor` factorization functions.

Tensors are mathematical objects that have recently gained great popularity due to their ability to model multi-way data dependencies. Tensor factorization (or decomposition) into latent factors is very important for numerous tasks, such as feature selection, dimensionality reduction, compression, data visualization and interpretation. Tensor factorizations are usually computed as solutions of optimization problems. The Canonical Decomposition or Canonical Polyadic Decomposition (CANDECOMP or CPD), also known as Parallel Factor Analysis (PARAFAC), and the Tucker Decomposition are the two most widely used tensor factorization models. Recent tensor applications, such as social network analysis, movie recommendation systems and targeted advertising, need to handle large-scale tensors, making the implementation of efficient parallel algorithms the only viable solution. The main focus of the PARTENSOR toolbox is the design and implementation of efficient parallel algorithms for tensor factorization.

- Canonical Polyadic Decomposition
- Canonical Polyadic Decomposition with Dimension Tree
- Options for ALS Method

1.1.1 Canonical Polyadic Decomposition

This is a simple user guide of how to use the Canonical Polyadic Decomposition. In this implementation, the `Matrix` Module from `Eigen` is used. In all cases of Sequential Policy the function needs a `gcc` compiler, `Eigen` library, and `spdlog` library, as specified in the library requirements. For the Parallel Policy with MPI case, `Boost` library and either `OpenMPI` or `MPICH` are additionally necessary for the tests to be executed, as specified in the library requirements.

The `CMakeLists.txt` in the `test` directory can be used as a guide.

Note

In our implementations, we adopt the alternating optimization framework.

1.1.1.1 Sequential Policy

1.1.1.1.1 Implementations with Randomly Generated Data In this case, the tensor is already stored in a `Tensor` data structure. This tensor can either be read from a file, using `read` function from ReadWrite.hpp, or generated randomly, using `makeTensor`, implemented in DataGeneration.hpp, as in the following example. Examples for both functions can be found also in Read/Write Data from/in Files and Data Generation. The user can then specify some options to execute the algorithm. A full list can be found in Options for ALS Method. The initial factors are created either randomly or an `stl` array containing them, can be passed as a parameter. These factors must be stored in `Matrix` data structure, and can also be read from files as the tensor or randomly created using `makeFactors`, which is also in Data Generation. The rank of the factorization `rank` is required.

Note

- There are 3 additional implementations, where either the options can be ignored and the default values can be used, or start with randomly generated factors. A user can also run tests without specifying the options and the initial factors. There are tests for each of these cases in the `test` directory (`cpd_options.cpp`, `cpd_factorsinit.cpp`, `cpd.cpp`).

- By enabling `writeToFile` and specifying `final_factors_paths` in the `options` object, the resulting factors can be saved in files. Their location will be indicated in the `final_factors_paths` variable.

```
#include "PARTENSOR.hpp"
int main(int argc, char** argv)
{
 partensor::Init(argc,argv);
 constexpr std::size_t tensor_order = 3;
 constexpr std::size_t rank = 2;
 using Matrix = partensor::Matrix;
 using Tensor = partensor::Tensor<tensor_order>;
 using Status = partensor::Status<Tensor>;
 using Options = partensor::Options<Tensor>;
 using Constraint = partensor::Constraint;
 std::array<int,tensor_order> tnsDims = {10, 11, 12};
 std::array<Matrix,tensor_order> init_factors;
 Tensor tnsX;
 Options options;
 options.max_iter = 50;
 options.constraints[0] = Constraint::nonnegativity;
 options.constraints[1] = Constraint::unconstrained;
 options.constraints[2] = Constraint::orthogonality;
 partensor::makeTensor(tnsDims, options.constraints, rank, tnsX);
 partensor::makeFactors(tnsDims, options.constraints, rank, init_factors);
 Status status = partensor::cpd(tnsX, rank, options, init_factors);
 for(std::size_t i=0; i<tensor_order; ++i)
 std::cout « "\n factor " « i « "\n" « status.factors[i] « std::endl;
 return 0;
}
```

1.1.1.1.2 Implementations Reading Tensor/Factors from Files PARTENSOR toolbox provides also more compact implementations, when the Tensor to be factorized or even the starting points-factors are stored on files. Follows, the same implementation as the previous section, but in this case the Tensor and the initialized factors are stored in files in the disk. In this case the length of each dimension of the tensor is needed to be stored in an `stl` array `tnsDims`. Again the factorization `rank` is essential. As for the options, as mentioned in the previous section, a full list of the options can be found in Options for ALS Method. Finally, an `stl` array containing `strings` with length of `tensor_order+1`, where index 0 contains the path for the tensor and the rest the paths for the factors.

Note

- There are 3 additional implementations, where either the options can be ignored and the default values can be used, or start with randomly generated factors. A user can also run tests without specifying the options and the initial factors. There are tests for each of these cases in the `test` directory (`cpd_options_file.cpp`, `cpd_factorsinit_file.cpp`, `cpd_file.cpp`).

- By enabling `writeToFile` and specifying `final_factors_paths` in the `options` object, the resulting factors can be saved in files. Their location will be indicated in the `final_factors_paths` variable.

```
#include "PARTENSOR.hpp"
int main(int argc, char** argv)
{
 partensor::Init(argc,argv);
 constexpr std::size_t tensor_order = 3;
 constexpr std::size_t rank = 2;
 using Tensor = partensor::Tensor<tensor_order>;
 using Status = partensor::Status<Tensor>;
 using Options = partensor::Options<Tensor>;
 using Constraint = partensor::Constraint;
 std::array<int,tensor_order> tnsDims = {10, 11, 12};
 std::array<std::string, tensor_order+1> paths;
 Options options;
 options.max_iter = 50;
 options.constraints[0] = Constraint::nonnegativity;
 options.constraints[1] = Constraint::unconstrained;
 options.constraints[2] = Constraint::orthogonality;
 paths[0] = "../data/tns.bin";
 paths[1] = "../data/A.bin";
 paths[2] = "../data/B.bin";
 paths[3] = "../data/C.bin";
 Status status = partensor::cpd(tnsDims, rank, paths, options);
 for(std::size_t i=0; i<tensor_order; ++i)
 std::cout « "\n factor " « i « "\n" « status.factors[i] « std::endl;
 return 0;
}
```

1.1.1.2 Parallel Policy with MPI

1.1.1.2.1 Implementations with Randomly Generated Data In this case, the tensor is already stored in a `Tensor` data structure. This tensor can be generated randomly using `makeTensor`, implemented in DataGeneration.hpp, as in the following example. More information about functions to generate pseudo random data can be found in Data Generation. The user can then specify some options to execute the algorithm. A full list resides here Options for ALS Method.

Note

In the options list, the `proc_per_mode` array can be used only in an MPI environment.

The initial factors are created either randomly or an `stl` array containing them, can be passed as a parameter. These factors must be stored in `Matrix` data structure, and can also be randomly created using `makeFactors`, which is also in Data Generation. Finally, the rank of the factorization `rank` is necessary.

Note

- There are 3 additional implementations, where either the options can be ignored and the default values can be used, or start with randomly generated factors. A user can also run tests without specifying the options and the initial factors. There are tests for each of these cases in the `test` directory (`cpd_mpi_options.cpp`, `cpd_mpi_factorsinit.cpp`, `cpd_mpi.cpp`).

- By enabling `writeToFile` and specifying `final_factors_paths` in the `options` object, the resulting factors can be saved in files. Their location will be indicated in the `final_factors_paths` variable.

```
#define USE_MPI 1
#include "PARTENSOR.hpp"
int main(int argc, char** argv)
{
 partensor::Init(argc,argv);
 partensor::MPI_Communicator _comm = partensor::Partensor()->MpiCommunicator();
 constexpr std::size_t tensor_order = 3;
 constexpr std::size_t rank = 2;
 using Matrix = partensor::Matrix;
 using Tensor = partensor::Tensor<tensor_order>;
 using Status = partensor::MpiStatus<Tensor>;
 using Options = partensor::MpiOptions<Tensor>;
 using Constraint = partensor::Constraint;
 std::array<int,tensor_order> tnsDims = {10, 11, 12};
 std::array<Matrix,tensor_order> init_factors;
 Tensor tnsX;
 Options options;
 options.max_iter = 50;
 options.constraints[0] = Constraint::nonnegativity;
 options.constraints[1] = Constraint::unconstrained;
 options.constraints[2] = Constraint::orthogonality;
 partensor::makeTensor(tnsDims, options.constraints, rank, tnsX);
 partensor::makeFactors(tnsDims, options.constraints, rank, init_factors);
 Status status = partensor::cpd(partensor::execution::mpi, tnsX, rank, options, init_factors);
 if(_comm.rank() == 0)
 {
 for(std::size_t i=0; i<tensor_order; ++i)
 std::cout « "\n factor " « i « "\n" « status.factors[i] « std::endl;
 }
 return 0;
}
```

1.1.1.2.2 Implementations Reading Tensor/Factors from Files PARTENSOR toolbox provides implementations for these cases, where either the tensor or the factors are stored in files in the disk. They follow, the same implementation as in the previous section. Here, the length of each dimension of the tensor needs to be stored in an `stl` array, `tnsDims`. Again, the factorization `rank` is required. A full list of the options can be found in Options for ALS Method.

Note

In the options list, the `proc_per_mode` array can be used only in an MPI environment.

Finally, an `stl` array of length `tensor_order+1` is required, which contains `strings`. The element at index 0 points to the path of the tensor and the rest of the elements point to the paths for the factors respectively.

Note

- There are 3 additional implementations, where either the options can be ignored and the default values can be used, or start with randomly generated factors. A user can also run tests without specifying the options and the initial factors. There are tests for each of these cases in the `test` directory (`cpd_mpi_options_file.cpp`, `cpd_mpi_factorsinit_file.cpp`, `cpd_mpi_file.cpp`).

- By enabling `writeToFile` and specifying `final_factors_paths` in the `options` object, the resulting factors can be saved in files. Their location will be indicated in the `final_factors_paths` variable.

```
#define USE_MPI 1
#include "PARTENSOR.hpp"
int main(int argc, char** argv)
{
 partensor::Init(argc,argv);
 partensor::MPI_Communicator _comm = partensor::Partensor()->MpiCommunicator();
 constexpr std::size_t tensor_order = 3;
 constexpr std::size_t rank = 2;
 using Tensor = partensor::Tensor<tensor_order>;
 using Status = partensor::MpiStatus<Tensor>;
 using Options = partensor::MpiOptions<Tensor>;
 using Constraint = partensor::Constraint;
 Options options;
 std::array<int,tensor_order> tnsDims = {10, 11, 12};
 std::array<std::string, tensor_order+1> paths;
 options.max_iter = 50;
 options.constraints[0] = Constraint::nonnegativity;
 options.constraints[1] = Constraint::unconstrained;
 options.constraints[2] = Constraint::orthogonality;
 paths[0] = "../data/tnsX_3.bin";
 paths[1] = "../data/A_3.bin";
 paths[2] = "../data/B_3.bin";
 paths[3] = "../data/C_3.bin";
 Status status = partensor::cpd(partensor::execution::mpi, tnsDims, rank, paths, options);
 if(_comm.rank() == 0)
 {
 for(std::size_t i=0; i<tensor_order; ++i)
 std::cout « "\n factor " « i « "\n" « status.factors[i] « std::endl;
 }
 return 0;
}
```

1.1.1.3 Parallel Policy with OpenMP

1.1.1.3.1 Implementations with Randomly Generated Data In this case, the tensor is already stored in a `Tensor` data structure. This tensor can be generated randomly using `makeTensor`, implemented in DataGeneration.hpp, as in the following example. More information about functions to generate pseudo random data can be found in Data Generation. The user can then specify some options to execute the algorithm. A full list resides here Options for ALS Method.

Furthermore, an `stl` array, containing the initial factors, can be passed as a parameter. These factors must be stored in `Matrix` data structure. They can also be randomly generated using `makeFactors`, which is also defined in Data Generation. Finally, the rank of the factorization `rank` is required.

Note

- There are 3 additional implementations, where either the options can be ignored and the default values can be used, or start with randomly generated factors. A user can also run tests without specifying the options and the initial factors. There are tests for each of these cases in the `test` directory (`cpd_omp_options.cpp`, `cpd_omp_factorsinit.cpp`, `cpd_omp.cpp`).

- By enabling `writeToFile` and specifying `final_factors_paths` in the `options` object, the resulting factors can be saved in files. Their location will be indicated in the `final_factors_paths` variable.

```
#define USE_OPENMP 1
#include "PARTENSOR.hpp"
int main(int argc, char** argv)
{
 partensor::Init(argc,argv);
 constexpr std::size_t tensor_order = 3;
 constexpr std::size_t rank = 2;
 using Matrix = partensor::Matrix;
 using Tensor = partensor::Tensor<tensor_order>;
 using Status = partensor::OmpStatus<Tensor>;
 using Options = partensor::OmpOptions<Tensor>;
 using Constraint = partensor::Constraint;
 std::array<int,tensor_order> tnsDims = {10, 11, 12};
 std::array<Matrix,tensor_order> init_factors;
 Tensor tnsX;
 Options options;
 options.max_iter = 50;
 options.constraints[0] = Constraint::nonnegativity;
 options.constraints[1] = Constraint::unconstrained;
 options.constraints[2] = Constraint::orthogonality;
 partensor::makeTensor(tnsDims, options.constraints, rank, tnsX);
 partensor::makeFactors(tnsDims, options.constraints, rank, init_factors);
 Status status = partensor::cpd(partensor::execution::omp, tnsX, rank, options, init_factors);
 for(std::size_t i=0; i<tensor_order; ++i)
 std::cout « "\n factor " « i « "\n" « status.factors[i] « std::endl;
 return 0;
}
```

1.1.1.3.2 Implementations Reading Tensor/Factors from Files PARTENSOR toolbox provides implementations for these cases, where either the tensor or the factors are stored in files in the disk. They follow, the same implementation as in the previous section. Here, the length of each dimension of the tensor needs to be stored in an `stl` array, `tnsDims`. Again, the factorization `rank` is required. A full list of the options can be found in Options for ALS Method.

Finally, an `stl` array of length `tensor_order+1` is required, which contains `strings`. The element at index 0 points to the path of the tensor and the rest of the elements point to the paths for the factors respectively.

Note

- There are 3 additional implementations, where either the options can be ignored and the default values can be used, or start with randomly generated factors. A user can also run tests without specifying the options and the initial factors. There are tests for each of these cases in the `test` directory (`cpd_omp_options↩_file.cpp`, `cpd_omp_factorsinit_file.cpp`, `cpd_omp_file.cpp`).

- By enabling `writeToFile` and specifying `final_factors_paths` in the `options` object, the resulting factors can be saved in files. Their location will be indicated in the `final_factors_paths` variable.

```
#define USE_OPENMP 1
#include "PARTENSOR.hpp"
int main(int argc, char** argv)
{
 partensor::Init(argc,argv);
 constexpr std::size_t tensor_order = 3;
 constexpr std::size_t rank = 2;
 using Tensor = partensor::Tensor<tensor_order>;
 using Status = partensor::OmpStatus<Tensor>;
 using Options = partensor::OmpOptions<Tensor>;
 using Constraint = partensor::Constraint;
 std::array<int,tensor_order> tnsDims = {10, 11, 12};
 std::array<std::string, tensor_order+1> paths;
 Options options;
 options.max_iter = 50;
 options.constraints[0] = Constraint::nonnegativity;
 options.constraints[1] = Constraint::unconstrained;
 options.constraints[2] = Constraint::orthogonality;
 paths[0] = "../data/tns.bin";
 paths[1] = "../data/A.bin";
 paths[2] = "../data/B.bin";
 paths[3] = "../data/C.bin";
 Status status = partensor::cpd(partensor::execution::omp, tnsDims, rank, paths, options);
 for(std::size_t i=0; i<tensor_order; ++i)
 std::cout « "\n factor " « i « "\n" « status.factors[i] « std::endl;
 return 0;
}
```

1.1.2 Canonical Polyadic Decomposition with Dimension Tree

This is a simple user guide of how to use the Canonical Polyadic Decomposition, using **Dimension Trees**. In this implementation, both the `Matrix` and the `Tensor` Modules from `Eigen` are used. In all cases of Sequential Policy the function needs a `gcc` compiler, `Eigen` library, and `spdlog` library, as specified in the library requirements. For the Parallel Policy with MPI case, `Boost` library and either `OpenMPI` or `MPICH` are additionally necessary for the tests to be executed, as specified in the library requirements.

The `CMakeLists.txt` in `test` directory can be used as a guide.

Note

In our implementations, we adopt the alternating optimization framework.

1.1.2.1 Sequential Policy

1.1.2.1.1 Implementations with Randomly Generated Data In this case, the tensor is already stored in a `Tensor` data structure. This tensor can either be read from a file, using `read` function from ReadWrite.hpp, or generated randomly, using `makeTensor`, implemented in DataGeneration.hpp, as in the following example. Examples for both functions can be found also in Read/Write Data from/in Files and Data Generation. The user can then specify some options to execute the algorithm. A full list can be found in Options for ALS Method. The initial factors are created either randomly or an `stl` array containing them, can be passed as a parameter. These factors must be stored in `Matrix` or in `Tensor<2>` data structure, and can also be read from files as the tensor or randomly created using `makeFactors`, which is also in Data Generation. The rank of the factorization `rank` is required.

Note

- There are 3 additional implementations, where either the options can be ignored and the default values can be used, or start with randomly generated factors. A user can also run tests without specifying the options and the initial factors. There are tests for each of these cases in the `test` directory (`cpd_dimtrees_options.cpp`, `cpd_dimtrees_factorsinit.cpp`, `cpd_dimtrees.cpp`).

- By enabling `writeToFile` and specifying `final_factors_paths` in the `options` object, the resulting factors can be saved in files. Their location will be indicated in the `final_factors_paths` variable.

```
#include "PARTENSOR.hpp"
int main(int argc, char** argv)
{
 partensor::Init(argc,argv);
 constexpr std::size_t tensor_order = 3;
 constexpr std::size_t rank = 2;
 using Matrix = partensor::Matrix;
 using Tensor = partensor::Tensor<tensor_order>;
 using Status = partensor::Status<Tensor>;
 using Options = partensor::Options<Tensor>;
 using Constraint = partensor::Constraint;
 std::array<int,tensor_order> tnsDims = {10, 11, 12};
 std::array<Matrix,tensor_order> init_factors;
 Tensor tnsX;
 Options options;
 options.max_iter = 50;
 options.constraints[0] = Constraint::nonnegativity;
 options.constraints[1] = Constraint::unconstrained;
 options.constraints[2] = Constraint::orthogonality;
 partensor::makeTensor(tnsDims, options.constraints, rank, tnsX);
 partensor::makeFactors(tnsDims, options.constraints, rank, init_factors);
 Status status = partensor::cpdDimTree(tnsX, rank, options, init_factors);
 for(std::size_t i=0; i<tensor_order; ++i)
 std::cout « "\n factor " « i « "\n" « status.factors[i] « std::endl;
 return 0;
}
```

1.1.2.1.2 Implementations Reading Tensor/Factors from Files PARTENSOR toolbox provides also more compact implementations, when the Tensor to be factorized or even the starting points-factors are stored on files. Follows, the same implementation as the previous section, but in this case the Tensor and the initialized factors are stored in files in the disk. In this case the length of each dimension of the tensor is needed to be stored in an `stl` array `tnsDims`. Again the factorization `rank` is essential. As for the options, as mentioned in the previous section, a full list of the options can be found in Options for ALS Method. Finally, an `stl` array containing `strings` with length of `tensor_order+1`, where index 0 contains the path for the tensor and the rest the paths for the factors.

Note

- There are other 3 implementations, where either the options can be ignored and the default values being used, or the initialized factors ignored and start from random factors or even both options and starting points-factors be ignored and run the algorithm with randomly generated starting points. There are test showing these cases in `test` directory (`cpd_dimtrees_options_file.cpp`, `cpd_dimtrees_factorsinit←_file.cpp`, `cpd_dimtrees_file.cpp`).

- By enabling `writeToFile` and specifying `final_factors_paths` in the `options` object, the resulting factors can be saved in files. Their location will be indicated in the `final_factors_paths` variable.

```
#include "PARTENSOR.hpp"
int main(int argc, char** argv)
{
 partensor::Init(argc,argv);
 constexpr std::size_t tensor_order = 3;
 constexpr std::size_t rank = 2;
 using Tensor = partensor::Tensor<tensor_order>;
 using Status = partensor::Status<Tensor>;
 using Options = partensor::Options<Tensor>;
 using Constraint = partensor::Constraint;
 std::array<int,tensor_order> tnsDims = {10, 11, 12};
 std::array<std::string, tensor_order+1> paths;
 Options options;
 options.max_iter = 50;
 options.constraints[0] = Constraint::nonnegativity;
 options.constraints[1] = Constraint::unconstrained;
 options.constraints[2] = Constraint::orthogonality;
 paths[0] = "../data/tns.bin";
 paths[1] = "../data/A.bin";
 paths[2] = "../data/B.bin";
 paths[3] = "../data/C.bin";
 Status status = partensor::cpdDimTree(tnsDims, rank, paths, options);
 for(std::size_t i=0; i<tensor_order; ++i)
 std::cout « "\n factor " « i « "\n" « status.factors[i] « std::endl;
 return 0;
}
```

1.1.2.2 Parallel Policy with MPI

1.1.2.2.1 Implementations with Randomly Generated Data In this case, the tensor is already stored in a `Tensor` data structure. This tensor can be generated randomly using `makeTensor`, implemented in DataGeneration.hpp, as in the following example. More information about functions to generate pseudo random data can be found in Data Generation. The user can then specify some options to execute the algorithm. A full list resides here Options for ALS Method.

Note

In the options list, the `proc_per_mode` array can be used only in an MPI environment.

The initial factors are created either randomly or an `stl` array containing them, can be passed as a parameter. These factors must be stored in `Matrix` or `Tensor<2>` data structure, and can also be randomly created using `makeFactors`, which is also in Data Generation. Finally, the rank of the factorization `rank` is necessary.

Note

- There are 3 additional implementations, where either the options can be ignored and the default values can be used, or start with randomly generated factors. A user can also run tests without specifying the options and the initial factors. There are tests for each of these cases in the `test` directory (`cpd_dimtrees_mpi_options.cpp`, `cpd_dimtrees_mpi_factorsinit.cpp`, `cpd_dimtrees_mpi.cpp`).

- By enabling `writeToFile` and specifying `final_factors_paths` in the `options` object, the resulting factors can be saved in files. Their location will be indicated in the `final_factors_paths` variable.

```
#define USE_MPI 1
#include "PARTENSOR.hpp"
int main(int argc, char** argv)
{
 partensor::Init(argc,argv);
 partensor::MPI_Communicator _comm = partensor::Partensor()->MpiCommunicator();
 constexpr std::size_t tensor_order = 3;
 constexpr std::size_t rank = 2;
 using Matrix = partensor::Matrix;
 using Tensor = partensor::Tensor<tensor_order>;
 using Status = partensor::MpiStatus<Tensor>;
 using Options = partensor::MpiOptions<Tensor>;
 using Constraint = partensor::Constraint;
 std::array<int,tensor_order> tnsDims = {10, 11, 12};
 std::array<Matrix,tensor_order> init_factors;
 Tensor tnsX;
 Options options;
 options.max_iter = 50;
 options.constraints[0] = Constraint::nonnegativity;
 options.constraints[1] = Constraint::unconstrained;
 options.constraints[2] = Constraint::orthogonality;
 partensor::makeTensor(tnsDims, options.constraints, rank, tnsX);
 partensor::makeFactors(tnsDims, options.constraints, rank, init_factors);
 Status status = partensor::cpdDimTree(partensor::execution::mpi, tnsX, rank, options, init_factors);
 if(_comm.rank() == 0)
 {
 for(std::size_t i=0; i<tensor_order; ++i)
 std::cout « "\n factor " « i « "\n" « status.factors[i] « std::endl;
 }
 return 0;
}
```

1.1.2.2.2 Implementations Reading Tensor/Factors from Files

PARTENSOR toolbox provides implementations for these cases, where either the tensor or the factors are stored in files in the disk. They follow, the same implementation as in the previous section. Here, the length of each dimension of the tensor needs to be stored in an `stl` array, `tnsDims`. Again, the factorization `rank` is required. A full list of the options can be found in Options for ALS Method.

Note

In the options list, the `proc_per_mode` array can be used only in an MPI environment.

Finally, an `stl` array of length `tensor_order+1` is required, which contains `strings`. The element at index 0 points to the path of the tensor and the rest of the elements point to the paths for the factors respectively.

Note

- There are 3 additional implementations, where either the options can be ignored and the default values can be used, or start with randomly generated factors. A user can also run tests without specifying the options and the initial factors. There are tests for each of these cases in the `test` directory (`cpd_mpi_options_file.cpp`, `cpd_mpi_factorsinit_file.cpp`, `cpd_mpi_file.cpp`).

- By enabling `writeToFile` and specifying `final_factors_paths` in the `options` object, the resulting factors can be saved in files. Their location will be indicated in the `final_factors_paths` variable.

```
#define USE_MPI 1
#include "PARTENSOR.hpp"
int main(int argc, char** argv)
{
 partensor::Init(argc,argv);
 partensor::MPI_Communicator _comm = partensor::Partensor()->MpiCommunicator();
 constexpr std::size_t tensor_order = 3;
 constexpr std::size_t rank = 2;
 using Tensor = partensor::Tensor<tensor_order>;
 using Status = partensor::MpiStatus<Tensor>;
 std::array<int,tensor_order> tnsDims = {10, 11, 12};
 std::string path = "path to where your Tensor file is located";
 Status status = partensor::cpdDimTree(partensor::execution::mpi, tnsDims, rank, path);
 if(_comm.rank() == 0)
 {
 for(std::size_t i=0; i<tensor_order; ++i)
 std::cout « "\n factor " « i « "\n" « status.factors[i] « std::endl;
 }
 return 0;
}
```

1.1.3 Options for ALS Method

The following list specifies all the parameters the user can change. Each variable takes a default value.

For further information check PARTENSOR_basic.hpp in struct `Options`.

- **method**, Currently, ONLY the Alternating Least Squares (ALS) is available.
- **constraints**, `stl` array with constraints (Constants.hpp) for each factor.
- **threshold_error**, when the relative cost function is below this value the algorithm terminates.
- **nesterov_delta_1**, terminating condition parameter for the Nesterov-type algorithm (NesterovMNLS.hpp) based on the Karush Kuhn Tucker conditions.
- **nesterov_delta_2**, terminating condition parameter for the Nesterov-type algorithm (NesterovMNLS.hpp) based on the Karush Kuhn Tucker conditions.
- **max_iter**, maximum number of AO (Alternating Optimization) iterations.
- **normalization**, boolean variable that specifies if factors will be normalized after updating all factors.
- **acceleration**, boolean variable that specifies if an acceleration step takes place.
- **accel_coeff**, ONLY if acceleration is enabled then it is used to compute the acceleration step.
- **accel_fail**, ONLY if acceleration is enabled, describes after how many failures the accel_coeff will be increased.
- **writeToFile**, boolean variable that enables the user to save the final factors to files in the disk.
- **final_factors_paths**, ONLY if writeToFile is enabled, a `stl` array containing the paths where the resulting factors will be saved.
- **proc_per_mode**, ONLY if mpi is enabled, then it specifies the communication grid.

Warning

Variable's `final_factors_paths` default values may not exist in your system, or replace the existing ones.

1.2 Completion

This is a user guide of how to use Partensor completion functions.

The problem of tensor completion arises in many modern applications, such as machine learning, signal processing, and scientific computing, where our aim is to estimate missing values in multi-way data, using only the available elements and structural properties of the data. Matrix completion problems are closely related to recommendation problems, which can be viewed as completing a partially observable matrix whose entries are ratings. Matrix factorization was empirically shown to be a better model than traditional nearest-neighbour based approaches in the Netflix Prize competition. In many real world applications, when nonnegativity constraints are imposed to the factorization/completion, the results have more natural interpretations. In this work, we focus on multiway data and Nonnegative Tensor Completion (NTC). Similar to the matrix case, we employ factorization techniques to provide accurate recommendations. Other approaches for contextual recommendations use context as a means to pre-filter or postfilter the recommendations made.

- General Tensor Completion
- General Tensor Completion using Stochastic Methods
- Options for GTC

1.2.1 General Tensor Completion

This is a simple user guide of how to use the General Tensor Completion. In this implementation, the `Matrix` and `Tensor` Modules from Eigen are used. In all cases of Sequential Policy the function needs a gcc compiler, Eigen library, and spdlog library, as specified in the library requirements. For the Parallel Policy with MPI case, Boost library and either OpenMPI or MPICH are additionally necessary for the tests to be executed, as specified in the library requirements.

The `CMakeLists.txt` in the `test` directory can be used as a guide.

1.2.1.1 Sequential Policy

1.2.1.1.1 Implementations with Randomly Generated Data In this case, the tensor is already stored in a `Matrix` data structure in a compressed format. This tensor can either be read from a file, using `read` function from ReadWrite.hpp, or generated randomly as in the following example. The user can then specify some options to execute the algorithm. A full list can be found in Options for GTC. The initial factors are created either randomly or an `stl` array containing them, can be passed as a parameter. These factors must be stored in `Matrix` data structure, and can also be read from files as the tensor or randomly created using `makeFactors`. The rank of the factorization `rank` is required.

Note

- By enabling `writeToFile` and specifying `final_factors_paths` in the `options` object, the resulting factors can be saved in files. Their location will be indicated in the `final_factors_paths` variable.

```
#include <iostream>
#include <cstdlib>
#include "PARTENSOR.hpp"
using namespace partensor;
int main(int argc, char** argv)
{
 static constexpr std::size_t TnsSize = 3;
 const std::size_t R = 10;
 const int nnz = 3000;
 std::array<int, TnsSize> tnsDims = {50, 50, 50};
 using SparseTensor = partensor::SparseTensor<TnsSize>;
 using DataType = SparseTensorTraits<SparseTensor>::DataType;
 using Matrix = SparseTensorTraits<SparseTensor>::MatrixType;
 using MatrixArray = SparseTensorTraits<SparseTensor>::MatrixArray;
 using Constraints = SparseTensorTraits<SparseTensor>::Constraints;
 using Status = partensor::SparseStatus<TnsSize,execution::sequenced_policy,SparseDefaultValues>;
 using Options = partensor::SparseOptions<TnsSize,execution::sequenced_policy,SparseDefaultValues>;
 Constraints constraints;
 MatrixArray factorsInit;
 Matrix Ratings_Base_T(static_cast<int>(TnsSize)+1, nnz);
 std::string path;
 for (int mode = 0; mode < static_cast<int>(TnsSize) + 1; mode++)
 {
 if(mode < static_cast<int>(TnsSize))
 {
 for (int counter=0; counter<nnz; counter++)
 {
 Ratings_Base_T(mode, counter) = rand() % (tnsDims[mode]);
 }
 }
 else
 {
 for (int counter=0; counter<nnz; counter++)
 {
 Ratings_Base_T(mode, counter) = static_cast<double>(rand()) / RAND_MAX;
 }
 }
 }
 std::fill(constraints.begin(), constraints.end(), Constraint::nonnegativity);
 makeFactors(tnsDims, constraints, R, factorsInit);
 Options opt;
 opt.rank = R;
 opt.tnsDims = tnsDims;
 opt.nonZeros = nnz;
 opt.constraints = constraints;
 opt.acceleration = false;
 opt.max_nesterov_iter = 20;
 opt.initialized_factors = true;
 opt.factorsInit = factorsInit;
 opt.read_factors_from_file = false;
 opt.writeToFile = false;
 Status s1 = partensor::gtc(ptl::execution::seq, Ratings_Base_T, opt);
 std::cout « "Relative cost function:  " « s1.f_value/s1.frob_tns « std::endl;
 return 0;
}
```

1.2.1.1.2 Implementations Reading Tensor from File PARTENSOR toolbox provides also more compact implementations, when the Tensor to be factorized is stored in a file. Follows, the same implementation as the previous section, but in this case the Tensor and the initialized factors are stored in files in the disk. In this case the length of each dimension of the tensor is needed to be stored in an `stl` array `tnsDims`. Again the factorization `rank` is essential. As for the options, as mentioned in the previous section, a full list of the options can be found in Options for GTC.

Note

- By enabling `writeToFile` and specifying `final_factors_paths` in the `options` object, the resulting factors can be saved in files. Their location will be indicated in the `final_factors_paths` variable.

```
#include <iostream>
#include <cstdlib>
#include "PARTENSOR.hpp"
using namespace partensor;
int main(int argc, char** argv)
{
 static constexpr std::size_t TnsSize = 3;
 const std::size_t R = 10;
 const int nnz = 3000;
 std::array<int, TnsSize> tnsDims = {50, 50, 50};
 using SparseTensor = partensor::SparseTensor<TnsSize>;
 using DataType = SparseTensorTraits<SparseTensor>::DataType;
 using Matrix = SparseTensorTraits<SparseTensor>::MatrixType;
 using MatrixArray = SparseTensorTraits<SparseTensor>::MatrixArray;
 using Constraints = SparseTensorTraits<SparseTensor>::Constraints;
 using Status = partensor::SparseStatus<TnsSize,execution::sequenced_policy,SparseDefaultValues>;
 using Options = partensor::SparseOptions<TnsSize,execution::sequenced_policy,SparseDefaultValues>;
 Constraints constraints;
 std::string path;
 std::fill(constraints.begin(), constraints.end(), Constraint::nonnegativity);
 Options opt;
 opt.rank = R;
 opt.tnsDims = tnsDims;
 opt.nonZeros = nnz;
 opt.constraints = constraints;
 opt.acceleration = false;
 opt.max_nesterov_iter = 20;
 opt.ratings_path = "../data/Ratings_Base_T.bin";
 opt.initialized_factors = false;
 opt.read_factors_from_file = false;
 opt.writeToFile = false;
 Status s1 = partensor::gtc(ptl::execution::seq, opt);
 std::cout « "Relative cost function:  " « s1.f_value/s1.frob_tns « std::endl;
 return 0;
}
```

1.2.1.2 Parallel Policy with MPI

1.2.1.2.1 Implementations with Randomly Generated Data In this case, the tensor is already stored in a `Matrix` data structure in a compressed format. This tensor can be generated randomly as in the following example. The user can then specify some options to execute the algorithm. A full list resides here Options for GTC.

Note

In the options list, the `proc_per_mode` array can be used only in an MPI environment.

The initial factors are created either randomly or an `stl` array containing them, can be passed as a parameter. These factors must be stored in `Matrix` data structure, and can also be randomly created using `makeFactors`, which is also in Data Generation. Finally, the rank of the factorization `rank` is necessary.

Note

- By enabling `writeToFile` and specifying `final_factors_paths` in the `options` object, the resulting factors can be saved in files. Their location will be indicated in the `final_factors_paths` variable.

```
#define USE_MPI 1
#include <iostream>
#include <cstdlib>
#include "PARTENSOR.hpp"
using namespace partensor;
int main(int argc, char** argv)
{
 static constexpr std::size_t TnsSize = 3;
 const std::size_t R = 10;
 const int nnz = 3000;
 std::array<int, TnsSize> tnsDims = {50, 50, 50};
 using SparseTensor = partensor::SparseTensor<TnsSize>;
 using DataType = SparseTensorTraits<SparseTensor>::DataType;
 using Matrix = SparseTensorTraits<SparseTensor>::MatrixType;
 using MatrixArray = SparseTensorTraits<SparseTensor>::MatrixArray;
 using Constraints = SparseTensorTraits<SparseTensor>::Constraints;
 using Status = partensor::MpiSparseStatus<TnsSize>;
 using Options = partensor::MpiSparseOptions<TnsSize>;
 Constraints constraints;
 MatrixArray factorsInit;
 Matrix Ratings_Base_T(static_cast<int>(TnsSize)+1, nnz);
 std::array<int, TnsSize> procs = {2,2,2};
 for (int mode = 0; mode < static_cast<int>(TnsSize) + 1; mode++)
 {
 if(mode < static_cast<int>(TnsSize))
 {
 for (int counter=0; counter<nnz; counter++)
 {
 Ratings_Base_T(mode, counter) = rand() % (tnsDims[mode]);
 }
 }
 else
 {
 for (int counter=0; counter<nnz; counter++)
 {
 Ratings_Base_T(mode, counter) = static_cast<double>(rand()) / RAND_MAX;
 }
 }
 }
 std::fill(constraints.begin(), constraints.end(), Constraint::nonnegativity);
 makeFactors(tnsDims, constraints, R, factorsInit);
 ptl::MPI_Communicator _comm = ptl::Partensor()->MpiCommunicator();
 Options opt;
 opt.proc_per_mode = procs;
 opt.rank = R;
 opt.tnsDims = tnsDims;
 opt.nonZeros = nnz;
 opt.max_nesterov_iter = 20;
 opt.constraints = constraints;
 opt.acceleration = true;
 opt.initialized_factors = true;
 opt.factorsInit = factorsInit;
 opt.read_factors_from_file = false;
 ptl::Init(argc,argv);
 Status s1 = partensor::gtc(ptl::execution::mpi, Ratings_Base_T, opt);
 if(_comm.rank() == 0)
 {
 std::cout « "Relative cost function:  " « s1.f_value/s1.frob_tns « std::endl;
 }
 return 0;
}
```

1.2.1.2.2 Implementations Reading Tensor from File PARTENSOR toolbox provides implementations for these cases, where the tensor is stored in a file in the disk. They follow, the same implementation as in the previous section. Here, the length of each dimension of the tensor needs to be stored in an `stl` array, `tnsDims`. Again, the factorization `rank` is required. A full list of the options can be found in Options for GTC.

Note

In the options list, the `proc_per_mode` array can be used only in an MPI environment.

- By enabling `writeToFile` and specifying `final_factors_paths` in the `options` object, the resulting factors can be saved in files. Their location will be indicated in the `final_factors_paths` variable.

```
#define USE_MPI 1
#include <iostream>
#include <cstdlib>
#include "PARTENSOR.hpp"
using namespace partensor;
int main(int argc, char** argv)
{
 static constexpr std::size_t TnsSize = 3;
 const std::size_t R = 10;
 const int nnz = 3000;
 std::array<int, TnsSize> tnsDims = {50, 50, 50};
 using SparseTensor = partensor::SparseTensor<TnsSize>;
 using DataType = SparseTensorTraits<SparseTensor>::DataType;
 using Matrix = SparseTensorTraits<SparseTensor>::MatrixType;
 using MatrixArray = SparseTensorTraits<SparseTensor>::MatrixArray;
 using Constraints = SparseTensorTraits<SparseTensor>::Constraints;
 using Status = partensor::MpiSparseStatus<TnsSize>;
 using Options = partensor::MpiSparseOptions<TnsSize>;
 Constraints constraints;
 std::string path;
 std::array<int, TnsSize> procs = {2,2,2};
 std::fill(constraints.begin(), constraints.end(), Constraint::nonnegativity);
 ptl::MPI_Communicator _comm = ptl::Partensor()->MpiCommunicator();
 Options opt;
 opt.proc_per_mode = procs;
 opt.rank = R;
 opt.tnsDims = tnsDims;
 opt.nonZeros = nnz;
 opt.max_nesterov_iter = 20;
 opt.constraints = constraints;
 opt.acceleration = true;
 opt.ratings_path = "../data/Ratings_Base_T.bin";
 opt.initialized_factors = false;
 opt.read_factors_from_file = false;
 ptl::Init(argc,argv);
 Status s1 = partensor::gtc(ptl::execution::mpi, opt);
 if(_comm.rank() == 0)
 {
 std::cout « "Relative cost function:  " « s1.f_value/s1.frob_tns « std::endl;
 }
 return 0;
}
```

1.2.1.3 Parallel Policy with OpenMP

1.2.1.3.1 Implementations with Randomly Generated Data In this case, the tensor is already stored in a `Matrix` data structure in a compressed format. This tensor can be generated randomly as in the following example. The user can then specify some options to execute the algorithm. A full list resides here Options for GTC.

Furthermore, an `stl` array, containing the initial factors, can be passed as a parameter. These factors must be stored in `Matrix` data structure. They can also be randomly generated using `makeFactors`, which is also defined in Data Generation. Finally, the rank of the factorization `rank` is required.

Note

- By enabling `writeToFile` and specifying `final_factors_paths` in the `options` object, the resulting factors can be saved in files. Their location will be indicated in the `final_factors_paths` variable.

```
#define USE_OPENMP 1
#define EIGEN_DONT_PARALLELIZE
#include <iostream>
#include <cstdlib>
#include "PARTENSOR.hpp"
using namespace partensor;
int main(int argc, char** argv)
{
 static constexpr std::size_t TnsSize = 3;
 const std::size_t R = 10;
 const int nnz = 3000;
 std::array<int, TnsSize> tnsDims = {50, 50, 50};
 using SparseTensor = partensor::SparseTensor<TnsSize>;
 using DataType = SparseTensorTraits<SparseTensor>::DataType;
 using Matrix = SparseTensorTraits<SparseTensor>::MatrixType;
 using MatrixArray = SparseTensorTraits<SparseTensor>::MatrixArray;
 using Constraints = SparseTensorTraits<SparseTensor>::Constraints;
 using Status = partensor::OmpSparseStatus<TnsSize>;
 using Options = partensor::OmpSparseOptions<TnsSize>;
 Constraints constraints;
 MatrixArray factorsInit;
 Matrix Ratings_Base_T(static_cast<int>(TnsSize)+1, nnz);
 for (int mode = 0; mode < static_cast<int>(TnsSize) + 1; mode++)
 {
 if(mode < static_cast<int>(TnsSize))
 {
 for (int counter=0; counter<nnz; counter++)
 {
 Ratings_Base_T(mode, counter) = rand() % (tnsDims[mode]);
 }
 }
 else
 {
 for (int counter=0; counter<nnz; counter++)
 {
 Ratings_Base_T(mode, counter) = static_cast<double>(rand()) / RAND_MAX;
 }
 }
 }
 std::fill(constraints.begin(), constraints.end(), Constraint::nonnegativity);
 makeFactors(tnsDims, constraints, R, factorsInit);
 Options opt;
 opt.rank = R;
 opt.tnsDims = tnsDims;
 opt.nonZeros = nnz;
 opt.constraints = constraints;
 opt.acceleration = false;
 opt.max_nesterov_iter = 20;
 opt.initialized_factors = true;
 opt.factorsInit = factorsInit;
 opt.read_factors_from_file = false;
 opt.writeToFile = false;
 Status s1 = partensor::gtc(ptl::execution::omp, Ratings_Base_T, opt);
 std::cout « "Relative cost function:  " « s1.f_value/s1.frob_tns « std::endl;
 return 0;
}
```

1.2.1.3.2 Implementations Reading Tensor/Factors from Files PARTENSOR toolbox provides implementations for these cases, where the tensor is stored in a file in the disk. They follow, the same implementation as in the previous section. Here, the length of each dimension of the tensor needs to be stored in an `stl` array, `tnsDims`. Again, the factorization `rank` is required. A full list of the options can be found in Options for GTC.

Note

- By enabling `writeToFile` and specifying `final_factors_paths` in the `options` object, the resulting factors can be saved in files. Their location will be indicated in the `final_factors_paths` variable.

□

1.2.2 General Tensor Completion using Stochastic Methods

This is a simple user guide of how to use the General Tensor Completion. In this implementation, the `Matrix` and `Tensor` Modules from `Eigen` are used. In all cases of Sequential Policy the function needs a `gcc` compiler, `Eigen` library, and `spdlog` library, as specified in the library requirements. For the Parallel Policy with MPI case, `Boost` library and either `OpenMPI` or `MPICH` are additionally necessary for the tests to be executed, as specified in the library requirements.

The `CMakeLists.txt` in the `test` directory can be used as a guide.

1.2.2.1 Sequential Policy

In this case, the tensor is already stored in a `Matrix` data structure in a compressed format. This tensor can either be read from a file, using `read` function from ReadWrite.hpp, or generated randomly as in the following example. The user can then specify some options to execute the algorithm. A full list can be found in Options for GTC. The initial factors are created either randomly or an `stl` array containing them, can be passed as a parameter. These factors must be stored in `Matrix` data structure, and can also be read from files as the tensor or randomly created using `makeFactors`. The rank of the factorization `rank` is required.

Note

- By enabling `writeToFile` and specifying `final_factors_paths` in the `options` object, the resulting factors can be saved in files. Their location will be indicated in the `final_factors_paths` variable.

```
#include <iostream>
#include <cstdlib>
#include "PARTENSOR.hpp"
using namespace partensor;
int main(int argc, char** argv)
{
 static constexpr std::size_t TnsSize = 3;
 const std::size_t R = 10;
 const int nnz = 3000;
 std::array<int, TnsSize> tnsDims = {50, 50, 50};
 using SparseTensor = partensor::SparseTensor<TnsSize>;
 using DataType = SparseTensorTraits<SparseTensor>::DataType;
 using Matrix = SparseTensorTraits<SparseTensor>::MatrixType;
 using MatrixArray = SparseTensorTraits<SparseTensor>::MatrixArray;
 using Constraints = SparseTensorTraits<SparseTensor>::Constraints;
 using Status = partensor::SparseStatus<TnsSize,execution::sequenced_policy,SparseDefaultValues>;
 using Options = partensor::SparseOptions<TnsSize,execution::sequenced_policy,SparseDefaultValues>;
 Constraints constraints;
 Matrix Ratings_Base_T(static_cast<int>(TnsSize)+1, nnz);
 for (int mode = 0; mode < static_cast<int>(TnsSize) + 1; mode++)
 {
 if(mode < static_cast<int>(TnsSize))
 {
 for (int counter=0; counter<nnz; counter++)
 {
 Ratings_Base_T(mode, counter) = rand() % (tnsDims[mode]);
 }
 }
 else
 {
 for (int counter=0; counter<nnz; counter++)
 {
 Ratings_Base_T(mode, counter) = static_cast<double>(rand()) / RAND_MAX;
 }
 }
 }
 std::fill(constraints.begin(), constraints.end(), Constraint::nonnegativity);
 Options opt;
 opt.rank = R;
 opt.tnsDims = tnsDims;
 opt.nonZeros = nnz;
 opt.constraints = constraints;
 opt.acceleration = false;
 opt.max_nesterov_iter = 20;
 opt.c_stochastic_perc = 0.5;
 opt.initialized_factors = false;
 opt.read_factors_from_file = false;
 opt.writeToFile = false;
 Status s1 = partensor::gtc_stochastic(ptl::execution::seq, Ratings_Base_T, opt);
 std::cout « "Relative cost function:  " « s1.f_value/s1.frob_tns « std::endl;
 return 0;
}
```

1.2.2.2 Parallel Policy with MPI

1.2.2.2.1 Parallel Policy with MPI In this case, the tensor is already stored in a `Matrix` data structure in a compressed format. This tensor can be generated randomly as in the following example. The user can then specify some options to execute the algorithm. A full list resides here Options for GTC.

Note

In the options list, the `proc_per_mode` array can be used only in an MPI environment.

The initial factors are created either randomly or an `stl` array containing them, can be passed as a parameter. These factors must be stored in `Matrix` data structure, and can also be randomly created using `makeFactors`, which is also in Data Generation. Finally, the rank of the factorization `rank` is necessary.

Note

- By enabling `writeToFile` and specifying `final_factors_paths` in the `options` object, the resulting factors can be saved in files. Their location will be indicated in the `final_factors_paths` variable.

```
#define USE_MPI 1
#include <iostream>
#include <cstdlib>
#include "PARTENSOR.hpp"
using namespace partensor;
int main(int argc, char** argv)
{
 static constexpr std::size_t TnsSize = 3;
 const std::size_t R = 10;
 const int nnz = 3000;
 std::array<int, TnsSize> tnsDims = {50, 50, 50};
 using SparseTensor = partensor::SparseTensor<TnsSize>;
 using DataType = SparseTensorTraits<SparseTensor>::DataType;
 using Matrix = SparseTensorTraits<SparseTensor>::MatrixType;
 using MatrixArray = SparseTensorTraits<SparseTensor>::MatrixArray;
 using Constraints = SparseTensorTraits<SparseTensor>::Constraints;
 using Status = partensor::MpiSparseStatus<TnsSize>;
 using Options = partensor::MpiSparseOptions<TnsSize>;
 Constraints constraints;
 Matrix Ratings_Base_T(static_cast<int>(TnsSize)+1, nnz);
 std::array<int, TnsSize> procs = {2,2,2};
 for (int mode = 0; mode < static_cast<int>(TnsSize) + 1; mode++)
 {
 if(mode < static_cast<int>(TnsSize))
 {
 for (int counter=0; counter<nnz; counter++)
 {
 Ratings_Base_T(mode, counter) = rand() % (tnsDims[mode]);
 }
 }
 else
 {
 for (int counter=0; counter<nnz; counter++)
 {
 Ratings_Base_T(mode, counter) = static_cast<double>(rand()) / RAND_MAX;
 }
 }
 }
 std::fill(constraints.begin(), constraints.end(), Constraint::nonnegativity);
 ptl::MPI_Communicator _comm = ptl::Partensor()->MpiCommunicator();
 Options opt;
 opt.proc_per_mode = procs;
 opt.rank = R;
 opt.tnsDims = tnsDims;
 opt.nonZeros = nnz;
 opt.max_nesterov_iter = 20;
 opt.c_stochastic_perc = 0.5;
 opt.constraints = constraints;
 opt.acceleration = true;
 opt.initialized_factors = false;
 opt.read_factors_from_file = false;
 ptl::Init(argc,argv);
 Status s1 = partensor::gtc_stochastic(ptl::execution::mpi, Ratings_Base_T, opt);
 if(_comm.rank() == 0)
 {
 std::cout « "Relative cost function:  " « s1.f_value/s1.frob_tns « std::endl;
 }
 return 0;
}
```

1.2.2.3 Parallel Policy with OpenMP

In this case, the tensor is already stored in a `Matrix` data structure in a compressed format. This tensor can be generated randomly as in the following example. The user can then specify some options to execute the algorithm. A full list resides here Options for GTC.

Furthermore, an `stl` array, containing the initial factors, can be passed as a parameter. These factors must be stored in `Matrix` data structure. They can also be randomly generated using `makeFactors`, which is also defined in Data Generation. Finally, the rank of the factorization `rank` is required.

Note

- By enabling `writeToFile` and specifying `final_factors_paths` in the `options` object, the resulting factors can be saved in files. Their location will be indicated in the `final_factors_paths` variable.

```
#define USE_OPENMP 1
#define EIGEN_DONT_PARALLELIZE
#include <iostream>
#include <cstdlib>
#include "PARTENSOR.hpp"
using namespace partensor;
int main(int argc, char** argv)
{
 static constexpr std::size_t TnsSize = 3;
 const std::size_t R = 10;
 const int nnz = 3000;
 std::array<int, TnsSize> tnsDims = {50, 50, 50};
 using SparseTensor = partensor::SparseTensor<TnsSize>;
 using DataType = SparseTensorTraits<SparseTensor>::DataType;
 using Matrix = SparseTensorTraits<SparseTensor>::MatrixType;
 using MatrixArray = SparseTensorTraits<SparseTensor>::MatrixArray;
 using Constraints = SparseTensorTraits<SparseTensor>::Constraints;
 using Status = partensor::OmpSparseStatus<TnsSize>;
 using Options = partensor::OmpSparseOptions<TnsSize>;
 Constraints constraints;
 Matrix Ratings_Base_T(static_cast<int>(TnsSize)+1, nnz);
 for (int mode = 0; mode < static_cast<int>(TnsSize) + 1; mode++)
 {
 if(mode < static_cast<int>(TnsSize))
 {
 for (int counter=0; counter<nnz; counter++)
 {
 Ratings_Base_T(mode, counter) = rand() % (tnsDims[mode]);
 }
 }
 else
 {
 for (int counter=0; counter<nnz; counter++)
 {
 Ratings_Base_T(mode, counter) = static_cast<double>(rand()) / RAND_MAX;
 }
 }
 }
 std::fill(constraints.begin(), constraints.end(), Constraint::nonnegativity);
 Options opt;
 opt.rank = R;
 opt.tnsDims = tnsDims;
 opt.nonZeros = nnz;
 opt.constraints = constraints;
 opt.acceleration = false;
 opt.max_nesterov_iter = 20;
 opt.c_stochastic_perc = 0.5;
 opt.initialized_factors = false;
 opt.read_factors_from_file = false;
 opt.writeToFile = false;
 Status s1 = partensor::gtc_stochastic(ptl::execution::omp, Ratings_Base_T, opt);
 std::cout « "Relative cost function:  " « s1.f_value/s1.frob_tns « std::endl;
 return 0;
}
```

1.2.3 Options for GTC

The following list specifies all the parameters the user can change. Each variable takes a default value.

For further information check PARTENSOR_basic.hpp in struct `SparseOptions`.

- **rank**, the rank of the decomposition.
- **tnsDims**, an array with the dimensions of the tensor.
- **nonZeros**, the number of non-zeros in the sparse tensor.
- **initialized_factors**, a boolean that we set to true if we want to initialize the factors.
- **read_factors_from_file**, a boolean that we set to true if we want to read initialized factors from a file.
- **factorsInit**, an array of Matrices with the initialized factors.
- **method**, currently, ONLY the Alternating Least Squares (ALS) is available.
- **constraints**, `stl` array with constraints (Constants.hpp) for each factor.
- **threshold_error**, when the relative cost function is below this value the algorithm terminates.
- **nesterov_delta_1**, terminating condition parameter for the Nesterov-type algorithm (NesterovMNLS.hpp) based on the Karush Kuhn Tucker conditions.
- **nesterov_delta_2**, terminating condition parameter for the Nesterov-type algorithm (NesterovMNLS.hpp) based on the Karush Kuhn Tucker conditions.
- **max_nesterov_iter**, maximum number of nesterov algorithm iterations.
- **c_stochastic_perc**, option for stochastic version.
- **lambdas**, option for nesterov algorithm.
- **max_iter**, maximum number of AO (Alternating Optimization) iterations.
- **max_duration**, maximum duration.
- **normalization**, boolean variable that specifies if factors will be normalized after updating all factors.
- **acceleration**, boolean variable that specifies if an acceleration step takes place.
- **averaging**,
- **accel_coeff**, ONLY if acceleration is enabled then it is used to compute the acceleration step.
- **accel_fail**, ONLY if acceleration is enabled, describes after how many failures the accel_coeff will be increased.
- **writeToFile**, boolean variable that enables the user to save the final factors to files in the disk.
- **final_factors_paths**, ONLY if writeToFile is enabled, a `stl` array containing the paths where the resulting factors will be saved.
- **ratings_path**, contains the path of the compressed sparse tensor.
- **initial_factors_paths**, a `stl` array containing the paths where the initialised factors are saved.
- **proc_per_mode**, ONLY if mpi is enabled, then it specifies the communication grid.

Warning

Variable's `final_factors_paths` default values may not exist in your system, or replace the existing ones.

1.3 Matrix and Tensor Operations

The `Partensor` library also provides the following operations,

- Element-wise Product
- Khatri-Rao Product
- Kronecker Product
- Tensor Matricization
- Data Generation

1.3.1 Element-wise Product

Given two 33 matrices $\mathbf{A}$ and $\mathbf{B}$, the Hadamard or Element-wise product is defined as

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} .* \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{11}\,b_{11} & a_{12}\,b_{12} & a_{13}\,b_{13} \\ a_{21}\,b_{21} & a_{22}\,b_{22} & a_{23}\,b_{23} \\ a_{31}\,b_{31} & a_{32}\,b_{32} & a_{33}\,b_{33} \end{bmatrix}.$$

The library provides an implementation of the `Hadamard` product between 2 matrices, but also expands the operation for more than 2 matrices. In `CwiseProd.hpp` we describe the implementation, which is computed according to the formula

$$\mathbf{H} = \mathbf{A_1} \ .* \ \mathbf{A_2} \ .* \ \cdots \ .* \ \mathbf{A_n}$$

1.3.1.1 How to use this function

A simple example follows, that shows the use of this operation. The example is described in more detail in the next section.

```
#include "PARTENSOR.hpp"
using namespace partensor;
int main()
{
 const int row = 5;
 const int col = 5;
 Matrix A(row, col);
 Matrix B(row, col);
 Matrix C(row, col);
 Matrix D(row, col);
 generateRandomMatrix(A);
 generateRandomMatrix(B);
 generateRandomMatrix(C);
 generateRandomMatrix(D);
 Matrix result_2(row, col);
 Matrix result_3(row, col);
 Matrix result_4(row, col);
 result_2 = CwiseProd(A, B);
 result_3 = CwiseProd(A, B, C);
 result_4 = CwiseProd(A, B, C, D);
 return 0;
}
```

1.3.1.2 Comments on the Example

First of all, the rows and the columns of all four matrices are initialized. Then, these matrices are generated randomly, using the `generateRandomMatrix` function. Then 3 more matrices are initialized in order to store the results. Finally, the computation of the Hadamard product is performed, between 2, 3 and 4 matrices.

Note

- The $.*$ symbol denotes the Element-wise operation.

- Because `CwiseProd` is a variadic function, there is no limitation on how many Matrices can be passed as input arguments.

1.3.2 Khatri-Rao Product

The Khatri-Rao product is defined as the column-wise Kronecker product. In other words, given an MN matrix $\mathbf{A}$ and a PN matrix $\mathbf{B}$, the Khatri-Rao product is defined as

$$\mathbf{A} \odot \mathbf{B} = [\mathbf{a_1} \otimes \mathbf{b_1}\ \mathbf{a_2} \otimes \mathbf{b_2} \cdots \mathbf{a_n} \otimes \mathbf{b_n}]$$

which is an MPN matrix.

The library provides an implementation of the `Khatri-Rao` product between 2 matrices, but also expands the operation for more than 2 matrices. In `KhatriRao.hpp`, the implementation is described. The mathematical formula to compute the `Khatri-Rao` product follows,

$$\mathbf{Krao} = \mathbf{A_1} \odot \mathbf{A_2} \odot \cdots \odot \mathbf{A_n}$$

1.3.2.1 How to use this function

A simple example follows, that shows the use of this operation. The example is described in more detail in the next section.

```
#include "PARTENSOR.hpp"
using namespace partensor;
int main()
{
 const int row = 10;
 const int col = 4;
 Matrix A(row, col);
 Matrix B(row, col);
 Matrix C(row, col);
 Matrix D(row, col);
 generateRandomMatrix(A);
 generateRandomMatrix(B);
 generateRandomMatrix(C);
 generateRandomMatrix(D);
 Matrix result_2(row*row, col);
 Matrix result_3(row*row*row, col);
 Matrix result_4(row*row*row*row, col);
 result_2 = KhatriRao(A, B);
 result_3 = KhatriRao(A, B, C);
 result_4 = KhatriRao(A, B, C, D);
 return 0;
}
```

1.3.2.2 Comments on the Example

First of all, the rows and the columns of all four matrices are initialized. Then, these matrices are generated randomly using with `generateRandomMatrix` function. Then 3 more matrices initialized in order to store the results. Finally, the computation of Khatri-Rao product is performed, between 2, 3 and 4 matrices.

Note

- The $\otimes$ symbol denotes the Kronecker operation, while $\odot$ symbol denotes the Khatri-Rao operation.

- Because `KhatriRao` is a variadic function, there is no limitation on how many Matrices can be passed as input arguments

1.3.3 Kronecker Product

Given an MN matrix $\mathbf{A}$ and a PQ matrix $\mathbf{B}$, the Kronecker product is defined as

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & a_{12} & \mathbf{B} & \cdots & a_{1N} & \mathbf{B} \\ a_{21}\mathbf{B} & a_{22} & \mathbf{B} & \cdots & a_{2N} & \mathbf{B} \\ \vdots & & \vdots & \ddots & & \vdots \\ a_{M1}\mathbf{B} & a_{M2} & \mathbf{B} & \cdots & a_{MN} & \mathbf{B} \end{bmatrix}$$

which is an $MPNQ$ matrix.

The library provides an implementation of the `Kronecker` product between 2 matrices, but also expands the operation for more than 2 matrices. In `Kronecker.hpp` the implementation is described, according to the mathematical formula

$$\mathbf{Kr} = \mathbf{A_1} \otimes \mathbf{A_2} \otimes \cdots \otimes \mathbf{A_n}$$

1.3.3.1 How to use this function

A simple example follows, that shows the use of this operation. The example is described in more detail in the next section.

```
#include "PARTENSOR.hpp"
using namespace partensor;
int main()
{
 const int row = 5;
 const int col = 4;
 Matrix A(row, col);
 Matrix B(row, col);
 Matrix C(row, col);
 Matrix D(row, col);
 generateRandomMatrix(A);
 generateRandomMatrix(B);
 generateRandomMatrix(C);
 generateRandomMatrix(D);
 Matrix result_2(row*row, col*col);
 Matrix result_3(row*row*row, col*col*col);
 Matrix result_4(row*row*row*row, col*col*col*col);
 result_2 = Kronecker(A, B);
 result_3 = Kronecker(A, B, C);
 result_4 = Kronecker(A, B, C, D);
 return 0;
}
```

1.3.3.2 Comments on the Example

First of all, the rows and the columns of all four matrices are initialized. Then, these matrices are generated randomly using with `generateRandomMatrix` function. Then 3 more matrices initialized in order to store the results. Finally, the computation of Kronecker product is performed, between 2, 3 and 4 matrices.

Note

- The $\otimes$ symbol denotes the Kronecker operation.

- Because `Kronecker` is a variadic function, there is no limitation on how many Matrices can be passed as input arguments.

1.3.4 Tensor Matricization

A tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_n \times \cdots \times I_N}$ can be matricized with respect to the n-th `mode`, namely, the n-mode matricization is the matrix $\mathrm{X}_{(n)} \in \mathbb{R}^{I_n \times (\prod_{k \neq n}^{N} I_k)}$.

The library provides an implementation of this operation in `Matricization.hpp`.

1.3.4.1 How to use this function

```
#include "PARTENSOR.hpp"
using namespace partensor;
int main()
{
 constexpr std::size_t tensor_order = 3;
 constexpr int dim0 = 3;
 constexpr int dim1 = 4;
 constexpr int dim2 = 5;
 std::array<int,tensor_order> tensor_dims = {dim0, dim1, dim2};
 Tensor<tensor_order> tnsX;
 tnsX.resize(tensor_dims);
 generateRandomTensor(tnsX);
 Matrix mat1(dim0, dim1*dim2);
 Matrix mat2(dim1, dim0*dim2);
 Matrix mat3(dim2, dim0*dim1);
 mat1 = Matricization(tnsX, 0);
 mat2 = Matricization(tnsX, 1);
 mat3 = Matricization(tnsX, 2);
 return 0;
}
```

1.3.4.2 Comments on the Example

First of all, the tensor order `TnsSize` is initialized. Then, in lines `9-12`, a 3D `Tensor` is defined, with dimensions `tnsDims`. Then, its entries are drawn from a `Uniform` distribution in [0,1], with function `generateRandomTensor`. This function is implemented in `DataGeneration.hpp`. In the sequel, 3 matrices are initialized in order to contain the matricizations. Afterwards the operation is performed, in modes 0, 1, and 2 respectively.

Warning

The implementation is supported **ONLY** for Tensors with `order` in the range of [3-8].

1.3.5 Data Generation

In `DataGeneration.hpp`, there are implementations for both `Matrix` and `Tensor` modules of `Eigen` for data generation.

1.3.5.1 Randomly Generated Data for Initialized Matrix

If the user has already initialized a `Matrix` then, this matrix is filled with data generated from a `Uniform` distribution in [0,1].

```
#include "PARTENSOR.hpp"
#include <iostream>
using namespace partensor;
int main()
{
 Matrix mat(4,5);
 generateRandomMatrix(mat);
 std::cout « "Matrix\n" « mat « std::endl;
 return 0;
}
```

1.3.5.2 Randomly Generated Data for Initialized Tensor

If the user has already initialized a `Tensor` then, this tensor is filled with data generated from either

- `Uniform` distribution in [0,1] or,
- `Normal` distribution with mean() = 0 and standard deviation() = 1.

```
#include "PARTENSOR.hpp"
#include <iostream>
using namespace partensor;
int main()
{
 constexpr std::size_t tensor_order = 3;
 std::array<int,tensor_order> tnsDims = {10, 11, 12};
 Tensor<tensor_order> tnsX;
 tnsX.resize(tnsDims);
 // zero variable can also be ignored
 generateRandomTensor(tnsX, 0);
 std::cout « "Uniform distribution\n" « tnsX « std::endl;
 generateRandomTensor(tnsX, 1);
 std::cout « "\nNormal distribution\n" « tnsX « std::endl;
 return 0;
}
```

1.3.5.3 Matricization of a Tensor from Factors

In case the factors are available and stored in an `stl` array, then a Tensor can be created using the `generate↔Tensor` function. There are two implementations available.

1.3.5.3.1 Matricized Tensor If the factors are stored as `Matrix` type then, use the version that is used in the following example.

```
#include "PARTENSOR.hpp"
#include <iostream>
using namespace partensor;
int main()
{
 const int rank = 5;
 const int rowA = 10;
 const int rowB = 12;
 const int rowC = 15;
 Matrix A(rowA, rank);
 Matrix B(rowB, rank);
 Matrix C(rowC, rank);
 generateRandomMatrix(A);
 generateRandomMatrix(B);
 generateRandomMatrix(C);
 std::array<Matrix, 3> factors = {A, B, C};
 Matrix matricized_tensor = generateTensor(0, factors);
 std::cout « "Matricization of first mode\n" « matricized_tensor « std::endl;
 return 0;
}
```

1.3.5.3.2 Tensor If the factors are stored as `Tensor<2>` type then, use the alternative version of the function as follows.

```
#include "PARTENSOR.hpp"
#include <iostream>
using namespace partensor;
int main()
{
 constexpr std::size_t tensor_order = 3;
 constexpr std::size_t matrix_order = 2;
 using Tensor_2d = Tensor<matrix_order>;
 const int rank = 5;
 const int rowA = 10;
 const int rowB = 12;
 const int rowC = 15;
 Tensor_2d A(rowA, rank);
 Tensor_2d B(rowB, rank);
 Tensor_2d C(rowC, rank);
 generateRandomTensor(A);
 generateRandomTensor(B);
 generateRandomTensor(C);
 std::array<Tensor_2d, tensor_order> factors = {A, B, C};
 Tensor<tensor_order> tnsX = generateTensor(factors);
 std::cout « "Tensor\n" « tnsX « std::endl;
 return 0;
}
```

In both examples, three matrices are initialized. In the first case the matrices are of `Matrix` type and are filled with random (`double`) values, using `generateRandomMatrix`. In the second case the matrices are of `Tensor<2>` type and are also filled with `double` values, using `generateRandomTensor`. In the first case `generateTensor` is being called with `mode` of matricization equal to 0, meaning that the matricization is created with respect to the first dimension of the tensor. In this case, only the `stl` array needs to be passed as input argument, and the `Tensor` is returned.

1.3.5.4 Generation of an Array with Factors

The library provides a mechanism to generate an `stl` array with factors, of either `Matrix`, `Tensor<2>` or even `FactoDimTree` type. In order to generate the factors, the user must provide the desirable constraint (see `Constants.hpp`), but also row and column dimensions for each factor. The row dimensions should be passed in a `stl` array, where at each index each factor's first dimension is specified.

```
#include "PARTENSOR.hpp"
#include <iostream>
using namespace partensor;
int main()
{
 constexpr std::size_t tensor_order = 3;
 constexpr std::size_t rank = 5;
 std::array<int, tensor_order> tensor_dimensions = {12, 10, 20};
 std::array<Constraint, tensor_order> constraints;
 std::fill(constraints.begin(), constraints.end(), Constraint::unconstrained);
 // In case of Matrix Module
 std::array<Matrix, tensor_order> matrix_factors;
 makeFactors(tensor_dimensions, constraints, rank, matrix_factors);
 std::cout « "matrix_factors[0]\n" « matrix_factors[0] « std::endl;
 // In case of Tensor<2> Module
 std::array<Tensor<2>, tensor_order> tensor_factors;
 makeFactors(tensor_dimensions, constraints, rank, tensor_factors);
 std::cout « "tensor_factors[1]\n" « tensor_factors[1] « std::endl;
 // In case of FactorDimTree Module
 std::array<FactorDimTree, tensor_order> factors_DimTrees;
 makeFactors(tensor_dimensions, constraints, rank, factors_DimTrees);
 std::cout « "factors_DimTrees[2]\n" « factors_DimTrees[2].factor « std::endl;
 return 0;
}
```

Note

`FactorDimTree` can be found in in `DimTrees.hpp` and `Tensor.hpp`.

Warning

- Also, the `constraints` in all cases cannot take value `constant`.

1.3.5.5 Generation of a Tensor with Constraints Applied

In case of synthetic data, a tensor can be created with arbitrary dimensions and constraints.

```
#include "PARTENSOR.hpp"
#include <iostream>
int main()
{
 constexpr std::size_t tensor_order = 3;
 constexpr std::size_t rank = 2;
 using Tensor = partensor::Tensor<tensor_order>;
 using Constraint = partensor::Constraint;
 std::array<int,tensor_order> tnsDims = {10, 11, 12};
 std::array<Constraint,tensor_order> constraints;
 Tensor tnsX;
 std::fill(constraints.begin(), constraints.end(), Constraint::unconstrained);
 partensor::makeTensor(tnsDims, constraints, rank, tnsX);
 std::cout « "Tensor \n" « tnsX « std::endl;
 return 0;
}
```

Warning

- Also, the `constraints` in all cases cannot take value `constant`.

1.4 Read/Write Data from/in Files

Within the library there are many functions in order to read or write from/in a file, of either `Matrix` or `Tensor` variable. There are also implementations for reading a `Tensor` from a file using the Message Passing Interface (MPI).

1.4.1 Write To a File

`Partensor` library provides a function, that can be used to save data in a file. It can be used to save data stored in either `Matrix` or `Tensor` type variable.

```
#include "PARTENSOR.hpp"
using namespace partensor;
int main()
{
 constexpr std::size_t tensor_order = 3;
 constexpr int dim0 = 2;
 constexpr int dim1 = 5;
 std::array<int,tensor_order> tnsDims = {10, 11, 12};
 int tnsDims_prod = std::accumulate(tnsDims.begin(), tnsDims.end(), 1, std::multiplies<int>());
 Matrix mtx(dim0,dim1);
 Tensor<tensor_order> tnsX;
 tnsX.resize(tnsDims);
 generateRandomMatrix(mtx);
 generateRandomTensor(tnsX);
 std::cout « "matrix\n" « mtx « std::endl;
 std::cout « "\ntensor\n" « tnsX « std::endl;
 write(mtx,"../data/matrix.bin",dim0*dim1);
 write(tnsX,"../data/tensor.bin",tnsDims_prod);
 return 0;
}
```

1.4.2 Read From a File

Following the previous section, a sequential function is also provided, in order to read from a whole file or only from a part of it. Also, the data can be stored in either `Matrix` or `Tensor` type variable.

Note

The following example reads the data written in the previous example (of the `write` function.)

```
#include "PARTENSOR.hpp"
using namespace partensor;
int main()
{
 constexpr std::size_t tensor_order = 3;
 constexpr int dim0 = 2;
 constexpr int dim1 = 5;
 std::array<int,tensor_order> tnsDims = {10, 11, 12};
 int tnsDims_prod = std::accumulate(tnsDims.begin(), tnsDims.end(), 1, std::multiplies<int>());
 Matrix mtx(dim0,dim1);
 Tensor<tensor_order> tnsX;
 tnsX.resize(tnsDims);
 read("../data/matrix.bin", dim0*dim1, 0, mtx);
 read("../data/tensor.bin", tnsDims_prod, 0, tnsX);
 std::cout « "matrix\n" « mtx « std::endl;
 std::cout « "\ntensor\n" « tnsX « std::endl;
 return 0;
}
```

1.5 Timers

The library provides implementations of different timers. There is a struct `Timers` in Timers.hpp containing clocks from *time.h*, `stl` *chrono* library and `MPI` timer.

An object called `timer` can be used to call either a function that starts measuring time or getting the time interval that passed since calling the starting timer.

The list of timers follows,

- stl clock,
- stl chrono high resolution,
- stl chrono steady resolution,
- MPI Wtime.

Chapter 2

Getting Started

This is a very short guide on how to get started with **PARTENSOR** library. In our repository page of our `site` a pdf of this manual can be downloaded.

2.1 How to "install" Partensor?

Partensor is a header-only library. It does not need to be installed. However, the following libraries are required to build Partensor.

Policies	Libraries	versions
Sequential or Parallel with **OpenMP**		
	`gcc`	>= *8.3.0*
	`Eigen`	>= *1.4.1*
	`spdlog`	>= *1.4.1*
Parallel with **MPI**		
	`Boost`	>= *1.↩ 71.0*
	`OpenMPI`, or	>= *4.0.1*
	`MPICH`	>= *3.2.0*

After downloading and installing the necessary libraries, then simply clone the project from our github repository `here`.
The **include** directory contains the functionality of the Partensor toolbox. Tests are available in the **test** directory that can be built and then executed.
The *CMakeLists.txt* file in the **test** directory can be used for further information.

Note

> The user should first specify the exact location of the aforementioned libraries in the project's *CMakeLists.txt* file, in order to compile either the implemented tests or user defined tests.

Finally, there is a header file called `PARTENSOR.hpp` that contains the functionality of the library, which should be included in the user's tests.

2.2 A Simple Example

Here we present a simple example to get started.

```
#include "PARTENSOR.hpp"
int main()
{
 constexpr std::size_t tensor_order = 3;
 constexpr std::size_t rank = 2;
 std::array<int,tensor_order> tensor_dims = {10, 10, 10};
 using Tensor = partensor::Tensor<tensor_order>;
 using Status = partensor::Status<Tensor>;
 Tensor tnsX;
 tnsX.resize(tensor_dims);
 partensor::generateRandomTensor(tnsX);
 Status status = partensor::cpd(tnsX, rank);
 for(std::size_t i=0; i<tensor_order; ++i)
 std::cout « "\n factor " « i « "\n" « status.factors[i] « std::endl;
 return 0;
}
```

2.3 Important Notes

2.3.1 Log File

For logging in this project, `spdlog` library is being used as mentioned in the previous section. For every call of a `cpd` or `cpdDimTree` function, the cost function value and other information are written in a file located in the **log** directory, called `partensor.txt`.

Warning

A directory called **log** must be created before the execution of any test in the main directory of the library.

Note

In case more than one executions of a `Partensor` function take place, the results will be appended.

2.3.2 Python Utility

`Partensor` provides a utility in order to print the results of the function that was called, using `Python` and two Python libraries `numpy` and `matplotlib`.

Note

The `plot_cost_function.py` is directly connected with the Log File in order to print the data.

Chapter 3

Overview

3.1 Abstract

The scientific and research interest of the Partensor project is the development of efficient algorithms for processing tensors of very large dimensions, their optimal implementation in parallel environments through the development of an integrated software toolbox, and their use in major special applications, such as fMRI.

3.2 Funding

The project is funded by the European Regional Development Fund under the Operational Program "↩ Competitiveness – Entrepreneurship – Innovation" (NSRF 2014-2020). It is implemented under the action "Research-Develop-Innovate", and running under the Special Management and Implementation Authority for Research, Technological Development and Innovation Actions by the Ministry of Education, Research and Religious Affairs (MIA-RTDI).

The title of this project is "Parallel algorithms and implementations for large scale tensors", with code T1EDK - 03360.

3.3 Contributors

- Technical University of Crete (TUC)
  - Liavas P. Athanasios
  - Karakasis Paris
  - Kolomvakis Christos
  - Papagiannakos John
  - Siaminou Nina
- Neurocom SA
  - Tsalidis Christos
  - Lourakis George
  - Lykoudis George

3.4 References

- A. P. Liavas, G. Kostoulas, G. Lourakis, K. Huang, and N. D. Sidiropoulos, "Nesterov-based Alternating Optimization for Nonnegative Tensor Factorization: Algorithm and Parallel Implementations," IEEE Transactions on Signal Processing, vol. 66, no. 4, Febr. 2018
- G. Lourakis, A. P. Liavas "Nesterov-based Alternating Optimization for Nonnegative Tensor Completion↩ : Algorithm and Parallel Implementation," Proc. IEEE 19th International Workshop on Signal Processing dvances in Wireless Communications (SPAWC), Kalamata, June 2018
- P. A. Karakasis and A. P. Liavas "Alternating Optimization for Tensor Factorization with Orthogonality Constraints: Algorithm and Parallel Implementation," International Conference on High Performance Computing & Simulation (HPCS), Orleans, July 2018

3.5 Links

- For more information about **PARTENSOR** toolbox visit our `site`
- In order to use the toolbox, our open source code can be downloaded from the **Github** repository `here`.

Chapter 4

Tensor Operations

In case the user wishes to explore more with different tensor operations, Partensor library provides the following,

- Data Generation
- Element-wise Product
- Khatri-Rao Product
- Kronecker Product
- Tensor Matricization

4.1 Data Generation

In DataGeneration.hpp, there are implementations for both Matrix and Tensor modules of Eigen for data generation.

4.1.1 Randomly Generated Data for Initialized Matrix

If the user has already initialized a Matrix then, this matrix is filled with data generated from a Uniform distribution in [0,1].

```
#include "PARTENSOR.hpp"
#include <iostream>
using namespace partensor;
int main()
{
 Matrix mat(4,5);
 generateRandomMatrix(mat);
 std::cout « "Matrix\n" « mat « std::endl;
 return 0;
}
```

4.1.2 Randomly Generated Data for Initialized Tensor

If the user has already initialized a Tensor then, this tensor is filled with data generated from either

- `Uniform` distribution in [0,1] or,
- `Normal` distribution with mean() = 0 and standard deviation() = 1.

```
#include "PARTENSOR.hpp"
#include <iostream>
using namespace partensor;
int main()
{
 constexpr std::size_t tensor_order = 3;
 std::array<int,tensor_order> tnsDims = {10, 11, 12};
 Tensor<tensor_order> tnsX;
 tnsX.resize(tnsDims);
 // zero variable can also be ignored
 generateRandomTensor(tnsX, 0);
 std::cout « "Uniform distribution\n" « tnsX « std::endl;
 generateRandomTensor(tnsX, 1);
 std::cout « "\nNormal distribution\n" « tnsX « std::endl;
 return 0;
}
```

4.1.3 Matricization of a Tensor from Factors

In case the factors are available and stored in an `stl` array, then a Tensor can be created using the `generate↩ Tensor` function. There are two implementations available.

4.1.3.1 Matricized Tensor

If the factors are stored as `Matrix` type then, use the version that is used in the following example.

```
#include "PARTENSOR.hpp"
#include <iostream>
using namespace partensor;
int main()
{
 const int rank = 5;
 const int rowA = 10;
 const int rowB = 12;
 const int rowC = 15;
 Matrix A(rowA, rank);
 Matrix B(rowB, rank);
 Matrix C(rowC, rank);
 generateRandomMatrix(A);
 generateRandomMatrix(B);
 generateRandomMatrix(C);
 std::array<Matrix, 3> factors = {A, B, C};
 Matrix matricized_tensor = generateTensor(0, factors);
 std::cout « "Matricization of first mode\n" « matricized_tensor « std::endl;
 return 0;
}
```

4.1.3.2 Tensor

If the factors are stored as `Tensor<2>` type then, use the alternative version of the function as follows.

```
#include "PARTENSOR.hpp"
#include <iostream>
using namespace partensor;
int main()
{
 constexpr std::size_t tensor_order = 3;
 constexpr std::size_t matrix_order = 2;
 using Tensor_2d = Tensor<matrix_order>;
 const int rank = 5;
 const int rowA = 10;
 const int rowB = 12;
 const int rowC = 15;
 Tensor_2d A(rowA, rank);
 Tensor_2d B(rowB, rank);
 Tensor_2d C(rowC, rank);
 generateRandomTensor(A);
 generateRandomTensor(B);
 generateRandomTensor(C);
 std::array<Tensor_2d, tensor_order> factors = {A, B, C};
 Tensor<tensor_order> tnsX = generateTensor(factors);
 std::cout « "Tensor\n" « tnsX « std::endl;
 return 0;
}
```

In both examples, three matrices are initialized. In the first case the matrices are of `Matrix` type and are filled with random (`double`) values, using `generateRandomMatrix`. In the second case the matrices are of `Tensor<2>` type and are also filled with `double` values, using `generateRandomTensor`. In the first case `generateTensor` is being called with `mode` of matricization equal to 0, meaning that the matricization is created with respect to the first dimension of the tensor. In this case, only the `stl` array needs to be passed as input argument, and the `Tensor` is returned.

4.1.4 Generation of an Array with Factors

The library provides a mechanism to generate an `stl` array with factors, of either `Matrix`, `Tensor<2>` or even `FactoDimTree` type. In order to generate the factors, the user must provide the desirable constraint (see `Constants.hpp`), but also row and column dimensions for each factor. The row dimensions should be passed in a `stl` array, where at each index each factor's first dimension is specified.

```
#include "PARTENSOR.hpp"
#include <iostream>
using namespace partensor;
int main()
{
 constexpr std::size_t tensor_order = 3;
 constexpr std::size_t rank = 5;
 std::array<int, tensor_order> tensor_dimensions = {12, 10, 20};
 std::array<Constraint, tensor_order> constraints;
 std::fill(constraints.begin(), constraints.end(), Constraint::unconstrained);
 // In case of Matrix Module
 std::array<Matrix, tensor_order> matrix_factors;
 makeFactors(tensor_dimensions, constraints, rank, matrix_factors);
 std::cout « "matrix_factors[0]\n" « matrix_factors[0] « std::endl;
 // In case of Tensor<2> Module
 std::array<Tensor<2>, tensor_order> tensor_factors;
 makeFactors(tensor_dimensions, constraints, rank, tensor_factors);
 std::cout « "tensor_factors[1]\n" « tensor_factors[1] « std::endl;
 // In case of FactorDimTree Module
 std::array<FactorDimTree, tensor_order> factors_DimTrees;
 makeFactors(tensor_dimensions, constraints, rank, factors_DimTrees);
 std::cout « "factors_DimTrees[2]\n" « factors_DimTrees[2].factor « std::endl;
 return 0;
}
```

Note

`FactorDimTree` can be found in in `DimTrees.hpp` and `Tensor.hpp`.

Warning

- Also, the `constraints` in all cases cannot take value `constant`.

4.1.5 Generation of a Tensor with Constraints Applied

In case of synthetic data, a tensor can be created with arbitrary dimensions and constraints.

```
#include "PARTENSOR.hpp"
#include <iostream>
int main()
{
 constexpr std::size_t tensor_order = 3;
 constexpr std::size_t rank = 2;
 using Tensor = partensor::Tensor<tensor_order>;
 using Constraint = partensor::Constraint;
 std::array<int,tensor_order> tnsDims = {10, 11, 12};
 std::array<Constraint,tensor_order> constraints;
 Tensor tnsX;
 std::fill(constraints.begin(), constraints.end(), Constraint::unconstrained);
 partensor::makeTensor(tnsDims, constraints, rank, tnsX);
 std::cout « "Tensor \n" « tnsX « std::endl;
 return 0;
}
```

Warning

- Also, the `constraints` in all cases cannot take value `constant`.

4.2 Element-wise Product

Given two 33 matrices $\mathbf{A}$ and $\mathbf{B}$, the Hadamard or Element-wise product is defined as

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} .* \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{11}\,b_{11} & a_{12}\,b_{12} & a_{13}\,b_{13} \\ a_{21}\,b_{21} & a_{22}\,b_{22} & a_{23}\,b_{23} \\ a_{31}\,b_{31} & a_{32}\,b_{32} & a_{33}\,b_{33} \end{bmatrix}.$$

The library provides an implementation of the `Hadamard` product between 2 matrices, but also expands the operation for more than 2 matrices. In `CwiseProd.hpp` we describe the implementation, which is computed according to the formula

$$\mathbf{H} = \mathbf{A_1} \ .* \ \mathbf{A_2} \ .* \ \cdots \ .* \ \mathbf{A_n}$$

4.2.1 How to use this function

A simple example follows, that shows the use of this operation. The example is described in more detail in the next section.

```
#include "PARTENSOR.hpp"
using namespace partensor;
int main()
{
 const int row = 5;
 const int col = 5;
 Matrix A(row, col);
 Matrix B(row, col);
 Matrix C(row, col);
 Matrix D(row, col);
 generateRandomMatrix(A);
 generateRandomMatrix(B);
 generateRandomMatrix(C);
 generateRandomMatrix(D);
 Matrix result_2(row, col);
 Matrix result_3(row, col);
 Matrix result_4(row, col);
 result_2 = CwiseProd(A, B);
 result_3 = CwiseProd(A, B, C);
 result_4 = CwiseProd(A, B, C, D);
 return 0;
}
```

4.2.2 Comments on the Example

First of all, the rows and the columns of all four matrices are initialized. Then, these matrices are generated randomly, using the `generateRandomMatrix` function. Then 3 more matrices are initialized in order to store the results. Finally, the computation of the Hadamard product is performed, between 2, 3 and 4 matrices.

Note

- The $.*$ symbol denotes the Element-wise operation.

- Because `CwiseProd` is a variadic function, there is no limitation on how many Matrices can be passed as input arguments.

4.3 Khatri-Rao Product

The Khatri-Rao product is defined as the column-wise Kronecker product. In other words, given an MN matrix $\mathbf{A}$ and a PN matrix $\mathbf{B}$, the Khatri-Rao product is defined as

$$\mathbf{A} \odot \mathbf{B} = [\mathbf{a_1} \otimes \mathbf{b_1}\ \mathbf{a_2} \otimes \mathbf{b_2} \cdots \mathbf{a_n} \otimes \mathbf{b_n}]$$

which is an MPN matrix.

The library provides an implementation of the `Khatri-Rao` product between 2 matrices, but also expands the operation for more than 2 matrices. In `KhatriRao.hpp`, the implementation is described. The mathematical formula to compute the `Khatri-Rao` product follows,

$$\mathbf{Krao} = \mathbf{A_1} \odot \mathbf{A_2} \odot \cdots \odot \mathbf{A_n}$$

4.3.1 How to use this function

A simple example follows, that shows the use of this operation. The example is described in more detail in the next section.

```
#include "PARTENSOR.hpp"
using namespace partensor;
int main()
{
 const int row = 10;
 const int col = 4;
 Matrix A(row, col);
 Matrix B(row, col);
 Matrix C(row, col);
 Matrix D(row, col);
 generateRandomMatrix(A);
 generateRandomMatrix(B);
 generateRandomMatrix(C);
 generateRandomMatrix(D);
 Matrix result_2(row*row, col);
 Matrix result_3(row*row*row, col);
 Matrix result_4(row*row*row*row, col);
 result_2 = KhatriRao(A, B);
 result_3 = KhatriRao(A, B, C);
 result_4 = KhatriRao(A, B, C, D);
 return 0;
}
```

4.3.2 Comments on the Example

First of all, the rows and the columns of all four matrices are initialized. Then, these matrices are generated randomly using with `generateRandomMatrix` function. Then 3 more matrices initialized in order to store the results. Finally, the computation of Khatri-Rao product is performed, between 2, 3 and 4 matrices.

Note

- The $\otimes$ symbol denotes the Kronecker operation, while $\odot$ symbol denotes the Khatri-Rao operation.

- Because `KhatriRao` is a variadic function, there is no limitation on how many Matrices can be passed as input arguments

4.4 Kronecker Product

Given an MN matrix $\mathbf{A}$ and a PQ matrix $\mathbf{B}$, the Kronecker product is defined as

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & a_{12} & \mathbf{B} & \cdots & a_{1N} & \mathbf{B} \\ a_{21}\mathbf{B} & a_{22} & \mathbf{B} & \cdots & a_{2N} & \mathbf{B} \\ \vdots & & \vdots & \ddots & & \vdots \\ a_{M1}\mathbf{B} & a_{M2} & \mathbf{B} & \cdots & a_{MN} & \mathbf{B} \end{bmatrix}$$

which is an $MPNQ$ matrix.

The library provides an implementation of the `Kronecker` product between 2 matrices, but also expands the operation for more than 2 matrices. In `Kronecker.hpp` the implementation is described, according to the mathematical formula

$$\mathbf{Kr} = \mathbf{A_1} \otimes \mathbf{A_2} \otimes \cdots \otimes \mathbf{A_n}$$

4.4.1 How to use this function

A simple example follows, that shows the use of this operation. The example is described in more detail in the next section.

```
#include "PARTENSOR.hpp"
using namespace partensor;
int main()
{
 const int row = 5;
 const int col = 4;
 Matrix A(row, col);
 Matrix B(row, col);
 Matrix C(row, col);
 Matrix D(row, col);
 generateRandomMatrix(A);
 generateRandomMatrix(B);
 generateRandomMatrix(C);
 generateRandomMatrix(D);
 Matrix result_2(row*row, col*col);
 Matrix result_3(row*row*row, col*col*col);
 Matrix result_4(row*row*row*row, col*col*col*col);
 result_2 = Kronecker(A, B);
 result_3 = Kronecker(A, B, C);
 result_4 = Kronecker(A, B, C, D);
 return 0;
}
```

4.4.2 Comments on the Example

First of all, the rows and the columns of all four matrices are initialized. Then, these matrices are generated randomly using with `generateRandomMatrix` function. Then 3 more matrices initialized in order to store the results. Finally, the computation of Kronecker product is performed, between 2, 3 and 4 matrices.

Note

- The $\otimes$ symbol denotes the Kronecker operation.

- Because `Kronecker` is a variadic function, there is no limitation on how many Matrices can be passed as input arguments.

4.5 Tensor Matricization

A tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_n \times \cdots \times I_N}$ can be matricized with respect to the n-th `mode`, namely, the n-mode matricization is the matrix $\mathrm{X}_{(n)} \in \mathbb{R}^{I_n \times (\prod_{k \neq n}^{N} I_k)}$.

The library provides an implementation of this operation in `Matricization.hpp`.

4.5.1 How to use this function

```
#include "PARTENSOR.hpp"
using namespace partensor;
int main()
{
 constexpr std::size_t tensor_order = 3;
 constexpr int dim0 = 3;
 constexpr int dim1 = 4;
 constexpr int dim2 = 5;
 std::array<int,tensor_order> tensor_dims = {dim0, dim1, dim2};
 Tensor<tensor_order> tnsX;
 tnsX.resize(tensor_dims);
 generateRandomTensor(tnsX);
 Matrix mat1(dim0, dim1*dim2);
 Matrix mat2(dim1, dim0*dim2);
 Matrix mat3(dim2, dim0*dim1);
 mat1 = Matricization(tnsX, 0);
 mat2 = Matricization(tnsX, 1);
 mat3 = Matricization(tnsX, 2);
 return 0;
}
```

4.5.2 Comments on the Example

First of all, the tensor order `TnsSize` is initialized. Then, in lines `9-12`, a 3D `Tensor` is defined, with dimensions `tnsDims`. Then, its entries are drawn from a `Uniform` distribution in [0,1], with function `generateRandomTensor`. This function is implemented in `DataGeneration.hpp`. In the sequel, 3 matrices are initialized in order to contain the matricizations. Afterwards the operation is performed, in modes 0, 1, and 2 respectively.

Warning

The implementation is supported **ONLY** for Tensors with `order` in the range of [3-8].

Chapter 5

Class Index

5.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 6

File Index

6.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 7

Class Documentation

7.1 cartesian_communicator Struct Reference

```
#include <ParallelWrapper.hpp>
```

7.1.1 Detailed Description

An MPI communicator with a cartesian topology.

A `cartesian_communicator` is a communicator whose topology is expressed as a grid. Cartesian communicators have the same functionality as communicators, but also allow one to query the relationships among processes and the properties of the grid.

Inherits Boost_CartCommunicator.

Public Member Functions

- cartesian_communicator (cartesian_communicator const &comm, std::vector< int > const &keep)
- cartesian_communicator (communicator const &comm, cartesian_topology const &dims, bool reorder=true)

7.1.2 Constructor & Destructor Documentation

7.1.2.1 cartesian_communicator() [1/2]

```
cartesian_communicator (
            communicator const & comm,
            cartesian_topology const & dims,
            bool reorder = true )  [inline]
```

Create a new communicator whose topology is described by the given cartesian. The indices of the vertices in the cartesian will be assumed to be the ranks of the processes within the communicator. There may be fewer vertices in the cartesian than there are processes in the communicator; in this case, the resulting communicator will be a NULL communicator.

Parameters

comm	The communicator that the new, cartesian communicator will be based on.
dims	the cartesian dimension of the new communicator. The size indicate the number of dimension. Some dimensions be set to zero, in which case the corresponding dimension value is left to the system.
reorder	Whether MPI is permitted to re-order the process ranks within the returned communicator, to better optimize communication. If true, the ranks of each process in the returned process will be new starting from zero.

7.1.2.2 cartesian_communicator() [2/2]

```
cartesian_communicator (
            cartesian_communicator const & comm,
            std::vector< int > const & keep )  [inline]
```

Create a new cartesian communicator whose topology is a subset of an existing cartesian communicator.

Parameters

comm	the original communicator.
keep	and array containing the dimension to keep from the existing communicator.

The documentation for this struct was generated from the following file:

- ParallelWrapper.hpp

7.2 cartesian_dimension Struct Reference

```
#include <ParallelWrapper.hpp>
```

7.2.1 Detailed Description

Specify the size and periodicity of the grid in a single dimension.

Inherits Boost_CartDimension.

Public Member Functions

- cartesian_dimension (int sz=0, bool p=true)

7.2.2 Constructor & Destructor Documentation

7.2.2.1 cartesian_dimension()

```
cartesian_dimension (
            int sz = 0,
            bool p = true )  [inline]
```

Parameters

sz	The size of the grid n this dimension.
p	Is the grid periodic in this dimension.

The documentation for this struct was generated from the following file:

- ParallelWrapper.hpp

7.3 cartesian_topology Struct Reference

```
#include <ParallelWrapper.hpp>
```

7.3.1 Detailed Description

Describe the topology of a cartesian grid.

Behave mostly like a sequence of `cartesian_dimension` with the notable exception that its size is fixed.

Inherits Boost_CartTopology.

Public Member Functions

- template<class InitArr_ >
  cartesian_topology (InitArr_ dims)

  Use dimensions specification provided in the sequence container as initial values.

7.3.2 Constructor & Destructor Documentation

7.3.2.1 cartesian_topology()

```
cartesian_topology (
            InitArr_ dims )  [inline]
```

Use dimensions specification provided in the sequence container as initial values.

Parameters

dims	must be a sequence container.

The documentation for this struct was generated from the following file:

- ParallelWrapper.hpp

7.4 communicator Struct Reference

`#include <ParallelWrapper.hpp>`

7.4.1 Detailed Description

A communicator that permits communication and synchronization among a set of processes.

The `communicator` class abstracts a set of communicating processes in MPI. All of the processes that belong to a certain communicator can determine the size of the communicator, their rank within the communicator, and communicate with any other processes in the communicator.

Inherits Boost_Communicator.

Public Member Functions

- communicator ()

7.4.2 Constructor & Destructor Documentation

7.4.2.1 communicator()

`communicator ( ) [inline]`

Build a new MPI communicator for `MPI_COMM_WORLD`.

Constructs a MPI communicator that attaches to `MPI_COMM_WORLD`, using boost::mpi::communicator.

The documentation for this struct was generated from the following file:

- ParallelWrapper.hpp

7.5 Conditions Struct Reference

`#include <TerminationConditions.hpp>`

7.5.1 Detailed Description

Struct containing default values, for the termination conditions.

The documentation for this struct was generated from the following file:

- TerminationConditions.hpp

7.6 CPD< Tensor_, execution::openmp_policy > Struct Template Reference

```
#include <CpdOpenMP.hpp>
```

7.6.1 Detailed Description

template<typename Tensor_>
struct partensor::v1::internal::CPD< Tensor_, execution::openmp_policy >

Includes the implementation of CPDOPENMP factorization. Based on the given parameters one of the four overloaded operators will be called.

Template Parameters

*Tensor_*	The Type of The given `Tensor` to be factorized.

Inherits CPD_Base< Tensor_ >.

Public Member Functions

- Status operator() (std::array< int, TnsSize > const &tnsDims, std::size_t const R, std::array< std::string, TnsSize > const &true_paths, std::array< std::string, TnsSize > const &init_paths, Options const &options)
- Status operator() (std::array< int, TnsSize > const &tnsDims, std::size_t const R, std::array< std::string, TnsSize+1 > const &paths)
- Status operator() (std::array< int, TnsSize > const &tnsDims, std::size_t const R, std::array< std::string, TnsSize+1 > const &paths, Options const &options)
- Status operator() (std::array< int, TnsSize > const &tnsDims, std::size_t const R, std::string const &path)
- Status operator() (std::array< int, TnsSize > const &tnsDims, std::size_t const R, std::string const &path, Options const &options)
- Status operator() (Tensor_ const &tnsX, std::size_t const R)
- Status operator() (Tensor_ const &tnsX, std::size_t const R, MatrixArray const &factorsInit)
- Status operator() (Tensor_ const &tnsX, std::size_t const R, Options const &options)
- Status operator() (Tensor_ const &tnsX, std::size_t const R, Options const &options, MatrixArray const &factorsInit)

7.6.2 Member Function Documentation

7.6.2.1 operator()() [1/9]

```
Status operator() (
            std::array< int, TnsSize > const & tnsDims,
            std::size_t const R,
            std::array< std::string, TnsSize > const & true_paths,
            std::array< std::string, TnsSize > const & init_paths,
            Options const & options )  [inline]
```

Implementation of CP Decomposition with user's changed values in Options struct and no initialized factors and using OpenMP. In this implementation the TRUE factors can be read from files. The Tensor is computed internally. Also, initialized factors can be read from a file, given the `paths` to the location in the disk, where they are stored.

With this version of `cpd` the true factors, that their outer product produce the Tensor and the initialized points-factors can be read from files.

Parameters

tnsDims	[in] `Stl` array containing the Tensor dimensions, whose length must be same as the Tensor order.
R	[in] The rank of decomposition.
true_paths	[in] An `stl` array containing paths for the true factors.
init_paths	[in] An `stl` array containing paths for initialized factors.
options	[in] User's `options`, other than the default. It must be of `partensor::Options<partensor::Tensor<order>>` type, where `order` must be in range of [3-8].

Returns

An object of type `Status` with the results of the algorithm.

7.6.2.2 operator()() [2/9]

```
Status operator() (
            std::array< int, TnsSize > const & tnsDims,
            std::size_t const R,
            std::array< std::string, TnsSize+1 > const & paths )  [inline]
```

Implementation of CP Decomposition with default values in Options Struct and initialized factors and using OpenMP. In this implementation the Tensor and the factors can be read from a file, given the `paths` to the location in the disk, where the Tensor is stored.

Parameters

tnsDims	[in] `Stl` array containing the Tensor dimensions, whose length must be same as the Tensor order.
R	[in] The rank of decomposition.
paths	[in] An `stl` array containing paths for the Tensor to be factorized and after that the paths for the initialized factors.

Returns

An object of type `Status` with the results of the algorithm.

7.6.2.3 operator()() [3/9]

```
Status operator() (
            std::array< int, TnsSize > const & tnsDims,
            std::size_t const R,
            std::array< std::string, TnsSize+1 > const & paths,
            Options const & options ) [inline]
```

Implementation of CP Decomposition with user's changed values in Options struct and initialized factors and using OpenMP. In this implementation the Tensor and the factors can be read from a file, given the `paths` to the location in the disk, where the Tensor is stored.

With this version of `cpd` the Tensor can be read from a file, specified in `path` variable.

Parameters

tnsDims	[in] `Stl` array containing the Tensor dimensions, whose length must be same as the Tensor order.
R	[in] The rank of decomposition.
paths	[in] An `stl` array containing paths for the Tensor to be factorized and after that the paths for the initialized factors.
options	[in] User's `options`, other than the default. It must be of `partensor::Options<partensor::Tensor<order>>` type, where `order` must be in range of [3-8].

Returns

An object of type `Status` with the results of the algorithm.

7.6.2.4 operator()() [4/9]

```
Status operator() (
            std::array< int, TnsSize > const & tnsDims,
            std::size_t const R,
            std::string const & path ) [inline]
```

Implementation of CP Decomposition with default values in Options Struct and randomly generated initial factors and using OpenMP. In this implementation the Tensor can be read from a file, given the `path` where the Tensor is located.

Parameters

tnsDims	[in] `Stl` array containing the Tensor dimensions, whose length must be same as the Tensor order.
R	[in] The rank of decomposition.
path	[in] The path where the tensor is located.

Returns

An object of type `Status` with the results of the algorithm.

7.6.2.5 operator()() [5/9]

```
Status operator() (
            std::array< int, TnsSize > const & tnsDims,
            std::size_t const R,
            std::string const & path,
            Options const & options ) [inline]
```

Implementation of CP Decomposition with user's changed values in Options struct and randomly generated initial factors and using OpenMP. In this implementation the Tensor can be read from a file, given the `path` where the Tensor is located.

Parameters

tnsDims	[in] `Stl` array containing the Tensor dimensions, whose length must be same as the Tensor order.
R	[in] The rank of decomposition.
path	[in] The path where the tensor is located.
options	[in] User's `options`, other than the default. It must be of `partensor::Options<partensor::Tensor<order>>` type, where `order` must be in range of [3-8].

Returns

An object of type `Status` with the results of the algorithm.

7.6.2.6 operator()() [6/9]

```
Status operator() (
            Tensor_ const & tnsX,
            std::size_t const R ) [inline]
```

Implementation of CP Decomposition with default values in Options struct and randomly generated initial factors and using OpenMP.

Parameters

tnsX	[in] The given Tensor to be factorized of `Tensor_` type, with `double` data.
R	[in] The rank of decomposition.

Returns

An object of type `Status` with the results of the algorithm.

7.6.2.7 operator()() [7/9]

```
Status operator() (
            Tensor_ const & tnsX,
            std::size_t const R,
            MatrixArray const & factorsInit )  [inline]
```

Implementation of CP Decomposition with default values in Options struct, but with initialized factors and using OpenMP.

Parameters

tnsX	[in] The given Tensor to be factorized of `Tensor_` type, with `double` data.
R	[in] The rank of decomposition.
factorsInit	[in] Uses initialized factors instead of randomly generated. The data must be of `partensor::Matrix` type and stored in an `stl` array with size same as the `order` of `tnsX`.

Returns

An object of type `Status` with the results of the algorithm.

7.6.2.8 operator()() [8/9]

```
Status operator() (
            Tensor_ const & tnsX,
            std::size_t const R,
            Options const & options )  [inline]
```

Implementation of CP Decomposition with user's changed values in Options struct, but with randomly generated initial factors and using OpenMP.

Parameters

tnsX	[in] The given Tensor to be factorized of `Tensor_` type, with `double` data.
R	[in] The rank of decomposition.
options	[in] User's `options`, other than the default. It must be of `partensor::Options<partensor::Tensor<order>>` type, where `order` must be in range of [3-8].

Returns

An object of type `Status` with the results of the algorithm.

7.6.2.9 operator()() [9/9]

```
Status operator() (
            Tensor_ const & tnsX,
            std::size_t const R,
            Options const & options,
            MatrixArray const & factorsInit ) [inline]
```

Implementation of CP Decomposition with user's changed values in Options struct, and also initialized factors and using OpenMP.

Parameters

tnsX	[in] The given Tensor to be factorized of `Tensor_` type, with `double` data.
R	[in] The rank of decomposition.
options	[in] User's `options`, other than the default. It must be of `partensor::Options<partensor::Tensor<order>>` type, where `order` must be in range of [3-8].
factorsInit	[in] Uses initialized factors instead of randomly generated. The data must be of `partensor::Matrix` type and stored in an `stl` array with size same as the `order` of `tnsX`.

Returns

An object of type `Status` with the results of the algorithm.

The documentation for this struct was generated from the following file:

- CpdOpenMP.hpp

7.7 CPD< Tensor_, execution::openmpi_policy > Struct Template Reference

`#include <CpdMpi.hpp>`

7.7.1 Detailed Description

template<typename Tensor_>
struct partensor::v1::internal::CPD< Tensor_, execution::openmpi_policy >

Includes the implementation of CPDMPI factorization. Based on the given parameters one of the four overloaded operators will be called.

Template Parameters

*Tensor_*	The Type of The given `Tensor` to be factorized.

Inherits CPD_Base< Tensor_ >.

Public Types

- using IntArray = typename TensorTraits< Tensor_ >::IntArray

Public Member Functions

- template<std::size_t TnsSize>
  Status operator() (std::array< int, TnsSize > const &tnsDims, std::size_t const R, std::array< std::string, TnsSize > const &true_paths, std::array< std::string, TnsSize > const &init_paths, Options const &options)
- template<std::size_t TnsSize>
  Status operator() (std::array< int, TnsSize > const &tnsDims, std::size_t const R, std::array< std::string, TnsSize+1 > const &paths)
- template<std::size_t TnsSize>
  Status operator() (std::array< int, TnsSize > const &tnsDims, std::size_t const R, std::array< std::string, TnsSize+1 > const &paths, Options const &options)
- template<std::size_t TnsSize>
  Status operator() (std::array< int, TnsSize > const &tnsDims, std::size_t const R, std::string const &path)
- template<std::size_t TnsSize>
  Status operator() (std::array< int, TnsSize > const &tnsDims, std::size_t const R, std::string const &path, Options const &options)
- Status operator() (Tensor_ const &tnsX, std::size_t const R)
- Status operator() (Tensor_ const &tnsX, std::size_t const R, MatrixArray const &factorsInit)
- Status operator() (Tensor_ const &tnsX, std::size_t const R, Options const &options)
- Status operator() (Tensor_ const &tnsX, std::size_t const R, Options const &options, MatrixArray const &factorsInit)

7.7.2 Member Typedef Documentation

7.7.2.1 IntArray

```
using IntArray = typename TensorTraits<Tensor_>::IntArray
```

Stl array of size TnsSize and containing int type.

7.7.3 Member Function Documentation

7.7.3.1 operator()() [1/9]

```
Status operator() (
            std::array< int, TnsSize > const & tnsDims,
            std::size_t const R,
            std::array< std::string, TnsSize > const & true_paths,
            std::array< std::string, TnsSize > const & init_paths,
            Options const & options )  [inline]
```

Implementation of CPDMPI factorization with default values in Options and initialized factors.

Template Parameters

TnsSize	Order of input Tensor.

Parameters

tnsDims	[in] `Stl` array containing the Tensor dimensions, whose length must be same as the Tensor order.
R	[in] The rank of decomposition.
true_paths	[in] An `stl` array containing paths for the true factors.
init_paths	[in] An `stl` array containing paths for initialized factors.
options	[in] User's `options`, other than the default. It must be of `partensor::Options<partensor::Tensor<order>>` type, where `order` must be in range of [3-8].

Returns

An object of type `Status` with the results of the algorithm.

7.7.3.2 operator()() [2/9]

```
Status operator() (
            std::array< int, TnsSize > const & tnsDims,
            std::size_t const R,
            std::array< std::string, TnsSize+1 > const & paths )  [inline]
```

Implementation of CPDMPI factorization with default values in Options and no initialized factors.

Template Parameters

TnsSize	Order of input Tensor.

Parameters

tnsDims	[in] `Stl` array containing the Tensor dimensions, whose length must be same as the Tensor order.
R	[in] The rank of decomposition.
paths	[in] An `stl` array containing paths for the Tensor to be factorized and after that the paths for the initialized factors.

Returns

An object of type `Status` with the results of the algorithm.

7.7.3.3 operator()() [3/9]

```
Status operator() (
            std::array< int, TnsSize > const & tnsDims,
            std::size_t const R,
            std::array< std::string, TnsSize+1 > const & paths,
            Options const & options ) [inline]
```

Implementation of CPDMPI factorization with default values in Options and initialized factors.

Template Parameters

TnsSize	Order of input Tensor.

Parameters

tnsDims	[in] `Stl` array containing the Tensor dimensions, whose length must be same as the Tensor order.
R	[in] The rank of decomposition.
paths	[in] An `stl` array containing paths for the Tensor to be factorized and after that the paths for the initialized factors.
options	[in] The options that the user wishes to use.

Returns

An object of type `Status` with the results of the algorithm.

7.7.3.4 operator()() [4/9]

```
Status operator() (
            std::array< int, TnsSize > const & tnsDims,
            std::size_t const R,
            std::string const & path ) [inline]
```

Implementation of CPDMPI factorization with default values in Options and no initialized factors.

Template Parameters

TnsSize	Order of input Tensor.

Parameters

tnsDims	[in] `Stl` array containing the Tensor dimensions, whose length must be same as the Tensor order.
R	[in] The rank of decomposition.
path	[in] The path where the tensor is located.

Returns

An object of type `Status` with the results of the algorithm.

7.7.3.5 operator()() [5/9]

```
Status operator() (
            std::array< int, TnsSize > const & tnsDims,
            std::size_t const R,
            std::string const & path,
            Options const & options ) [inline]
```

Implementation of CPDMPI factorization with default values in Options and no initialized factors.

Template Parameters

TnsSize	Order of input Tensor.

Parameters

tnsDims	[in] `Stl` array containing the Tensor dimensions, whose length must be same as the Tensor order.
R	[in] The rank of decomposition.
path	[in] The path where the tensor is located.
options	[in] The options that the user wishes to use.

Returns

An object of type `Status` with the results of the algorithm.

7.7.3.6 operator()() [6/9]

```
Status operator() (
            Tensor_ const & tnsX,
            std::size_t const R ) [inline]
```

Implementation of CPDMPI factorization with default values in Options and no initialized factors.

Parameters

tnsX	[in] The given Tensor to be factorized of `Tensor_` type, with `double` data.
R	[in] The rank of decomposition.

Returns

An object of type `Status` with the results of the algorithm.

7.7.3.7 operator()() [7/9]

```
Status operator() (
            Tensor_ const & tnsX,
            std::size_t const R,
            MatrixArray const & factorsInit ) [inline]
```

Implementation of CPDMPI factorization with default values in Options and no initialized factors.

Parameters

tnsX	[in] The given Tensor to be factorized of `Tensor_` type, with `double` data.
R	[in] The rank of decomposition.
factorsInit	[in] Uses initialized factors instead of randomly generated. The data must be of `partensor::Matrix` type and stored in an `stl` array with size same as the `order` of `tnsX`.

Returns

An object of type `Status` with the results of the algorithm.

7.7.3.8 operator()() [8/9]

```
Status operator() (
            Tensor_ const & tnsX,
            std::size_t const R,
            Options const & options ) [inline]
```

Implementation of CPDMPI factorization with default values in Options and no initialized factors.

Parameters

tnsX	[in] The given Tensor to be factorized of `Tensor_` type, with `double` data.
R	[in] The rank of decomposition.
options	[in] User's `options`, other than the default. It must be of `partensor::Options<partensor::Tensor<order>>` type, where `order` must be in range of [3-8].

Returns

An object of type `Status` with the results of the algorithm.

7.7.3.9 operator()() [9/9]

```
Status operator() (
            Tensor_ const & tnsX,
            std::size_t const R,
            Options const & options,
            MatrixArray const & factorsInit ) [inline]
```

Implementation of CPDMPI factorization with default values in Options and no initialized factors.

Parameters

tnsX	[in] The given Tensor to be factorized of `Tensor_` type, with `double` data.
R	[in] The rank of decomposition.
options	[in] User's `options`, other than the default. It must be of `partensor::Options<partensor::Tensor<order>>` type, where `order` must be in range of [3-8].
factorsInit	[in] Uses initialized factors instead of randomly generated. The data must be of `partensor::Matrix` type and stored in an `stl` array with size same as the `order` of `tnsX`.

Returns

An object of type `Status` with the results of the algorithm.

The documentation for this struct was generated from the following file:

- CpdMpi.hpp

7.8 CPD_DIMTREE< Tensor_, execution::openmpi_policy > Struct Template Reference

```
#include <CpdDimTreeMpi.hpp>
```

7.8.1 Detailed Description

template<typename Tensor_>
struct partensor::v1::internal::CPD_DIMTREE< Tensor_, execution::openmpi_policy >

Includes the implementation of CPDMPI factorization. Based on the given parameters one of the four overloaded operators will be called.

Template Parameters

Tensor←_	The Type of the given Eigen Tensor to be factorized.

Inherits CPD_DIMTREE_Base< Tensor_ >.

Public Member Functions

- template<std::size_t TnsSize>
  Status operator() (std::array< int, TnsSize > const &tnsDims, std::size_t const R, std::array< std::string, TnsSize+1 > const &paths)
- template<std::size_t TnsSize>
  Status operator() (std::array< int, TnsSize > const &tnsDims, std::size_t const R, std::array< std::string, TnsSize+1 > const &paths, Options const &options)
- template<std::size_t TnsSize>
  Status operator() (std::array< int, TnsSize > const &tnsDims, std::size_t const R, std::string const &path)
- template<std::size_t TnsSize>
  Status operator() (std::array< int, TnsSize > const &tnsDims, std::size_t const R, std::string const &path, Options const &options)
- Status operator() (Tensor_ const &tnsX, std::size_t const R)
- template<typename MatrixArray_ >
  Status operator() (Tensor_ const &tnsX, std::size_t const R, MatrixArray_ const &factorsInit)
- Status operator() (Tensor_ const &tnsX, std::size_t const R, Options const &options)
- template<typename MatrixArray_ >
  Status operator() (Tensor_ const &tnsX, std::size_t const R, Options const &options, MatrixArray_ const &factorsInit)

7.8.2 Member Function Documentation

7.8.2.1 operator()() [1/8]

```
Status operator() (
            std::array< int, TnsSize > const & tnsDims,
            std::size_t const R,
            std::array< std::string, TnsSize+1 > const & paths )  [inline]
```

Implementation of CPDMPI factorization with default values in Options and no initialized factors.

Template Parameters

TnsSize	Order of the input Tensor.

Parameters

tnsDims	[in] `Stl` array containing the Tensor dimensions, whose length must be same as the Tensor order.
R	[in] The rank of decomposition.
paths	[in] An `stl` array containing paths for the Tensor to be factorized and after that the paths for the initialized factors.

Returns

An object of type `Status` with the results of the algorithm.

7.8.2.2 operator()() [2/8]

```
Status operator() (
            std::array< int, TnsSize > const & tnsDims,
            std::size_t const R,
            std::array< std::string, TnsSize+1 > const & paths,
            Options const & options ) [inline]
```

Implementation of CPDMPI factorization with default values in Options and no initialized factors.

Template Parameters

TnsSize	Order of the input Tensor.

Parameters

tnsDims	[in] `Stl` array containing the Tensor dimensions, whose length must be same as the Tensor order.
R	[in] The rank of decomposition.
paths	[in] An `stl` array containing paths for the Tensor to be factorized and after that the paths for the initialized factors.
options	[in] The options that the user wishes to use.

Returns

An object of type `Status` with the results of the algorithm.

7.8.2.3 operator()() [3/8]

```
Status operator() (
            std::array< int, TnsSize > const & tnsDims,
            std::size_t const R,
            std::string const & path ) [inline]
```

Implementation of CPDMPI factorization with default values in Options and no initialized factors.

Template Parameters

TnsSize	Order of the input Tensor.

Parameters

tnsDims	[in] `Stl` array containing the Tensor dimensions, whose length must be same as the Tensor order.
R	[in] The rank of decomposition.
path	[in] The path where the tensor is located.

Returns

An object of type `Status` with the results of the algorithm.

7.8.2.4 operator()() [4/8]

```
Status operator() (
            std::array< int, TnsSize > const & tnsDims,
            std::size_t const R,
            std::string const & path,
            Options const & options )  [inline]
```

Implementation of CPDMPI factorization with default values in Options and no initialized factors.

Template Parameters

TnsSize	Order of the input Tensor.

Parameters

tnsDims	[in] `Stl` array containing the Tensor dimensions, whose length must be same as the Tensor order.
R	[in] The rank of decomposition.
path	[in] The path where the tensor is located.
options	[in] The options that the user wishes to use.

Returns

An object of type `Status` with the results of the algorithm.

7.8.2.5 operator()() [5/8]

```
Status operator() (
            Tensor_ const & tnsX,
            std::size_t const R )  [inline]
```

Implementation of CPDMPI factorization with default values in Options and no initialized factors.

Parameters

tnsX	[in] The given Eigen Tensor to be factorized.
R	[in] The rank of decomposition.

Returns

If `spdlog` provoke no exception, returns an object of type `Status` with the results of the algorithm.

7.8.2.6 operator()() [6/8]

```
Status operator() (
            Tensor_ const & tnsX,
            std::size_t const R,
            MatrixArray_ const & factorsInit ) [inline]
```

Implementation of CPDMPI factorization with default values in Options and no initialized factors.

Parameters

tnsX	[in] The given Eigen Tensor to be factorized.
R	[in] The rank of decomposition.
factorsInit	[in] Uses initialized factors instead of randomly generated.

Returns

If `spdlog` provoke no exception, returns an object of type `Status` with the results of the algorithm.

7.8.2.7 operator()() [7/8]

```
Status operator() (
            Tensor_ const & tnsX,
            std::size_t const R,
            Options const & options ) [inline]
```

Implementation of CPDMPI factorization with default values in Options and no initialized factors.

Parameters

tnsX	[in] The given Eigen Tensor to be factorized.
R	[in] The rank of decomposition.
options	[in] The options that the user wishes to use.

Returns

If `spdlog` provoke no exception, returns an object of type `Status` with the results of the algorithm.

7.8.2.8 operator()() [8/8]

```
Status operator() (
            Tensor_ const & tnsX,
            std::size_t const R,
            Options const & options,
            MatrixArray_ const & factorsInit )  [inline]
```

Implementation of CPDMPI factorization with default values in Options and no initialized factors.

Parameters

tnsX	[in] The given Eigen Tensor to be factorized.
R	[in] The rank of decomposition.
options	[in] The options that the user wishes to use.
factorsInit	[in] Uses initialized factors instead of randomly generated.

Returns

If `spdlog` provoke no exception, returns an object of type `Status` with the results of the algorithm.

The documentation for this struct was generated from the following file:

- CpdDimTreeMpi.hpp

7.9 DefaultValues< Tensor_ > Struct Template Reference

```
#include <PARTENSOR_basic.hpp>
```

7.9.1 Detailed Description

template<typename Tensor_>
struct partensor::DefaultValues< Tensor_ >

Default Values for CPD algorithm.

Contains default values for either constraints, error threshholders or termination conditions parameters. These values can be changed using `Options` struct and the appropriate `cpd` or `cpdDimTree` call.

Template Parameters

Tensor←_	Type(data type and order) of input Tensor.

Static Public Attributes

- static bool constexpr DefaultAcceleration

- static int constexpr DefaultAccelerationCoefficient
- static int constexpr DefaultAccelerationFail
- static Constraint constexpr DefaultConstraint
- static double constexpr DefaultLambda
- static Duration constexpr DefaultMaxDuration
- static unsigned constexpr DefaultMaxIter
- static Method constexpr DefaultMethod
- static double constexpr DefaultNesterovTolerance
- static bool constexpr DefaultNormalization
- static double constexpr DefaultProcessorPerMode
- static double constexpr DefaultThresholdError
- static bool constexpr DefaultWriteToFile

7.9.2 Member Data Documentation

7.9.2.1 DefaultAcceleration

```
bool constexpr DefaultAcceleration  [static], [constexpr]
```

Default value for acceleration.

7.9.2.2 DefaultAccelerationCoefficient

```
int constexpr DefaultAccelerationCoefficient  [static], [constexpr]
```

Default value for acceleration coefficient.

7.9.2.3 DefaultAccelerationFail

```
int constexpr DefaultAccelerationFail  [static], [constexpr]
```

Default value for acceleration fail.

7.9.2.4 DefaultConstraint

```
Constraint constexpr DefaultConstraint  [static], [constexpr]
```

Default value for Constraint is nonnegativity.

7.9.2.5 DefaultLambda

```
double constexpr DefaultLambda  [static], [constexpr]
```

Default value for lambda.

7.9.2.6 DefaultMaxDuration

```
Duration constexpr DefaultMaxDuration [static], [constexpr]
```

Default value outer loop maximum duration.

7.9.2.7 DefaultMaxIter

```
unsigned constexpr DefaultMaxIter [static], [constexpr]
```

Default value outer loop maximum iterations.

7.9.2.8 DefaultMethod

```
Method constexpr DefaultMethod [static], [constexpr]
```

Default value for Method is als.

7.9.2.9 DefaultNesterovTolerance

```
double constexpr DefaultNesterovTolerance [static], [constexpr]
```

Default value for Nesterov's tolerance.

7.9.2.10 DefaultNormalization

```
bool constexpr DefaultNormalization [static], [constexpr]
```

Default value for normalization.

7.9.2.11 DefaultProcessorPerMode

```
double constexpr DefaultProcessorPerMode [static], [constexpr]
```

Default value for number of processors per tensor mode.

7.9.2.12 DefaultThresholdError

```
double constexpr DefaultThresholdError [static], [constexpr]
```

Default value for cost function's threshold.

7.9.2.13 DefaultWriteToFile

```
bool constexpr DefaultWriteToFile [static], [constexpr]
```

Default value for write final factors to files.

The documentation for this struct was generated from the following file:

- PARTENSOR_basic.hpp

7.10 environment Struct Reference

```
#include <ParallelWrapper.hpp>
```

7.10.1 Detailed Description

Initialize, finalize, and query the `MPI` environment.

The `environment` class is used to initialize, finalize, and query the MPI environment.

The instance of `environment` will initialize MPI (by calling `MPI_Init`) in its constructor and finalize MPI (by calling `MPI_Finalize` for normal termination or `MPI_Abort` for an uncaught exception) in its destructor.

The use of `environment` is not mandatory. Users may choose to invoke `MPI_Init` and `MPI_Finalize` manually. In this case, no `environment` object is needed. If one is created, however, it will do nothing on either construction or destruction.

Inherits Boost_Environment.

Public Member Functions

- environment (int &argc, char **&argv, bool abort_on_exception=true)
- ∼environment ()=default

7.10.2 Constructor & Destructor Documentation

7.10.2.1 environment()

```
environment (
        int & argc,
        char **& argv,
        bool abort_on_exception = true ) [inline]
```

Initialize the MPI environment.

If the MPI environment has not already been initialized, initializes MPI with a call to `boost::mpi::environment`.

Parameters

argc	Number of arguments provided in `argv`, as passed into the program's `main` function.
argv	Array of argument strings passed to the program via `main` function.
abort_on_exception	When true, this object will abort the program if it is destructed due to an uncaught exception.

7.10.2.2 ∼environment()

`∼environment ( ) [default]`

Shuts down the MPI environment.

If this `environment` object was used to initialize the MPI environment, and the MPI environment has not already been shut down (finalized), this destructor will shut down the MPI environment. Under normal circumstances, this only involves invoking `MPI_Finalize`. However, if destruction is the result of an uncaught exception and the `abort_on_exception` parameter of the constructor had the value `true`, this destructor will invoke `MPI_↩ Abort` with `MPI_COMM_WORLD` to abort the entire MPI program with a result code of -1.

The documentation for this struct was generated from the following file:

- ParallelWrapper.hpp

7.11 ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize > Struct Template Reference

`#include <DimTrees.hpp>`

7.11.1 Detailed Description

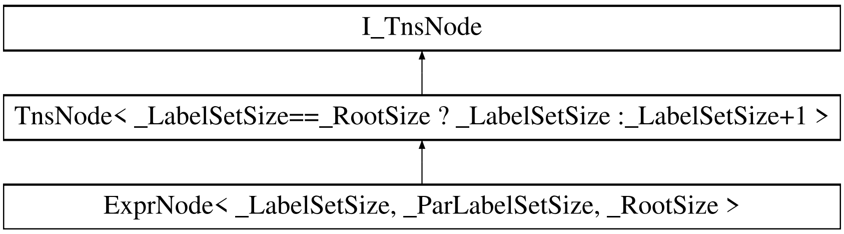
template<std::size_t _LabelSetSize, std::size_t _ParLabelSetSize, std::size_t _RootSize>
struct partensor::ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >

Configuration for `TnsNode`. Representation of the tree struture.

Template Parameters

_LabelSetSize	Size of the LabelSet.
_ParLabelSetSize	Size of the LabelSet of the parent node.
_RootSize	Size of the LabelSet of the root node.

Inheritance diagram for ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >:



Public Member Functions

- void * DeltaSet ()
- ExprNode ()
- void * LabelSet ()
- I_TnsNode * Left ()
- I_TnsNode * Parent ()
- I_TnsNode * Right ()
- I_TnsNode * SearchKey (int const aKey)
- void * TnsDims ()
- Parent_Tensor_Type TreeMode_N_Product (FactorDimTree *const aFactor, int const aNumFactors, int const id, std::array< int, ParTnsSize > const &aTnsDims, Hessian_Type &aGramian, std::array< int, BrotherLabelSetSize > &aDeltaSet)
- Tensor_Type TTVs (FactorDimTree *const aFactor, int const aNumFactors, int const id, std::array< int, BrotherLabelSetSize > const &aDeltaSet, Parent_Tensor_Type const &aX_partial, std::array< int, ParTnsSize > const &aTnsDims, Hessian_Type &aGramian)
- template<std::size_t DeltaSetSize, std::size_t ResTnsSize, std::size_t ResParTnsSize>
  void TTVs_util (FactorDimTree *const it, Tensor< static_cast< int >(ResParTnsSize)> const &aX_partial, int const aContractDim, std::array< int, ResParTnsSize > const &aTnsDims, Hessian_Type &aGramian, Tensor< static_cast< int >(ResTnsSize)> &aX_result)
- void UpdateTree (int const aNumFactors, int const id, FactorDimTree *aFactor) override

Public Attributes

- Left_Node_Type left
- std::array< int, BrotherLabelSetSize > mDeltaSet
- Hessian_Type mGramian
- int mKey
- std::array< int, LabelSetSize > mLabelSet
- std::array< int, TnsSize > mTnsDims
- Tensor_Type mTnsX
- bool mUpdated
- I_TnsNode * parent
- Right_Node_Type right

Static Public Attributes

- static constexpr std::size_t BrotherLabelSetSize
- static constexpr std::size_t DIM_HALF_SIZE
- static constexpr std::size_t DIM_LEFT_SIZE
- static constexpr std::size_t DIM_RIGHT_SIZE
- static constexpr bool IsFirstChild
- static constexpr bool IsLeaf
- static constexpr bool IsRoot
- static constexpr std::size_t LabelSetSize
- static constexpr std::size_t ParLabelSetSize
- static constexpr std::size_t ParTnsSize
- static constexpr std::size_t RootSize
- static constexpr std::size_t TnsSize

Protected Member Functions

- template< std::size_t _ParLabelSetSize2, std::size_t _ParParLabelSetSize, std::size_t _RootSize2> ExprNode (ExprNode< _ParLabelSetSize2, _ParParLabelSetSize, _RootSize2 > *parent_)

7.11.2 Constructor & Destructor Documentation

7.11.2.1 ExprNode() [1/2]

`ExprNode ( ) [inline]`

Default Constructor

7.11.2.2 ExprNode() [2/2]

```
ExprNode (
            ExprNode< _ParLabelSetSize2, _ParParLabelSetSize, _RootSize2 > * parent_ ) [inline],
[protected]
```

Protected Constructor.

7.11.3 Member Function Documentation

7.11.3.1 DeltaSet()

`void * DeltaSet ( ) [inline], [virtual]`

Returns

The `mDeltaSet` member variable of `ExprNode` with the set of identification for the neighboor- brother `TnsNode`. Needs explicit specification for the `stl` size `BrotherLabelSetSize`.

Implements I_TnsNode.

7.11.3.2 LabelSet()

`void * LabelSet ( ) [inline], [virtual]`

Returns

The `mTnsDims` member variable of `ExprNode` with the set of identification for `TnsNode`. Needs explicit specification for the `stl array` size `LabelSetSize` .

Implements I_TnsNode.

7.11.3.3 Left()

```
I_TnsNode * Left ( ) [inline], [virtual]
```

Returns

If the `Exprode` that calls the function is not a leaf node, then the Left `ExprNode` of the `this` node is returned.

Implements I_TnsNode.

7.11.3.4 Parent()

```
I_TnsNode * Parent ( ) [inline], [virtual]
```

Returns

If the `Exprode` that calls the function is not the root node, then the Parent `ExprNode` is returned.

Implements I_TnsNode.

7.11.3.5 Right()

```
I_TnsNode * Right ( ) [inline], [virtual]
```

Returns

If the `Exprode` that calls the function is not a leaf node, then the Right `ExprNode` of the `this` node is returned.

Implements I_TnsNode.

7.11.3.6 SearchKey()

```
I_TnsNode * SearchKey (
             int const aKey ) [inline], [virtual]
```

Search the `ExprTree` in order to find the `ExprNode` with `akey`.

Parameters

aKey	[in] Searching key value.

Returns

The ExprNode that has the searched key.

Implements I_TnsNode.

7.11.3.7 TnsDims()

```
void * TnsDims ( ) [inline], [virtual]
```

Returns

The mLabelSet member variable of ExprNode with the length of each the Tensor 's dimensions. Needs explicit specification for the stl array size TnsSize.

The mLabelSet of ExprNode.

Implements I_TnsNode.

7.11.3.8 TreeMode_N_Product()

```
ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >::Parent_Tensor_Type TreeMode_N_Product
(
            FactorDimTree *const aFactor,
            int const aNumFactors,
            int const id,
            std::array< int, ParTnsSize > const & aTnsDims,
            Hessian_Type & aGramian,
            std::array< int, BrotherLabelSetSize > & aDeltaSet )
```

Computes the N mode product of a tensor with a matrix.

Template Parameters

_LabelSetSize	Size of the LabelSet of this node.
_ParLabelSetSize	Size of the LabelSet of the parent node.
_RootSize	Size of the LabelSet of the root node.

Parameters

aFactor	[in] Factor (of type FactorDimTree) to use for tree mode N product.
aNumFactors	[in] Total number of factors.
id	[in] Indexing for the updating factor, aFactor.
aGramian	[in,out] Gramian matrix of the node.
aDeltaSet	[in,out] Label set of the brother node after the N-mode product.
aTnsDims	[in,out] stl with the dimensions of the final Tensor.

Returns

A TnsNode with size equal to ParTnsSize.

7.11.3.9 TTVs()

```
ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >::Tensor_Type TTVs (
            FactorDimTree *const aFactor,
            int const aNumFactors,
            int const id,
            std::array< int, BrotherLabelSetSize > const & aDeltaSet,
            Parent_Tensor_Type const & aX_partial,
            std::array< int, ParTnsSize > const & aTnsDims,
            Hessian_Type & aGramian )
```

Interface of TTV product computation, between a tensor and a matrix.

Template Parameters

_LabelSetSize	Size of the LabelSet of this node.
_ParLabelSetSize	Size of the LabelSet of the parent node.
_RootSize	Size of the LabelSet of the root node.

Parameters

aFactor	[in] Factor (of type FactorDimTree) to use for TTV product.
aNumFactors	[in] Total number of factors.
id	[in] Identification of the updating factor.
aDeltaSet	[in] stl with the Label set of the brother ExprNode after the TTV product of size BrotherLabelSetSize.
aX_partial	[in] Eigen Tensor used for TTV product.
aTnsDims	[in,out] stl with the dimensions of the computed Tensor of size ParTnsSize.
aGramian	[in,out] Gramian matrix of the ExprNode.

Returns

An TnsNode with size equal to TnsSize.

7.11.3.10 TTVs_util()

```
void TTVs_util (
            FactorDimTree *const it,
            Tensor< static_cast< int >(ResParTnsSize)> const & aX_partial,
            int const aContractDim,
            std::array< int, ResParTnsSize > const & aTnsDims,
            Hessian_Type & aGramian,
            Tensor< static_cast< int >(ResTnsSize)> & aX_result )
```

Computes the TTV product of a tensor with a matrix, using recursion.

Template Parameters

_LabelSetSize	Size of the LabelSet of `this` node.
_ParLabelSetSize	Size of the LabelSet of the parent node.
_RootSize	Size of the LabelSet of the root node.
DeltaSetSize	Size of the DeltaSset of this node.
ResTnsSize	Order of the resulting `Tensor`.
ResParTnsSize	Order of the parent's `Tensor`.

Parameters

it	[in] Factor (of `FactorDimTree` type) to use for `TTV` product.
aX_partial	[in] `Tensor` for `TTV` product.
aContractDim	[in] Dimension for `TTV` product, based on being a Left or Right child.
aTnsDims	[in,out] `stl array` with the dimensions of the computed Tensor of size `ResParTnsSize`.
aGramian	[in,out] Gramian matrix of the `ExprNode`.
aX_result	[in,out] The result of TTV `Tensor` of size `ResTnsSize`.

7.11.3.11 UpdateTree()

```
void UpdateTree (
            int const aNumFactors,
            int const id,
            FactorDimTree * aFactor )  [override], [virtual]
```

Updates the factors in each node until computing the leaf nodes and their `Tensors`. Based on the position of the node chooses to execute `TreeMode_N_Product` or `TTV`. Works in recursive way.

Template Parameters

_LabelSetSize	Size of the LabelSet.
_ParLabelSetSize	Size of the LabelSet of the parent node.
_RootSize	Size of the LabelSet of the root node.

Parameters

aNumFactors	[in] Total number of factors.
id	[in] Identification of the updating factor.
aFactor	[in,out] The factor to be updated.

Implements I_TnsNode.

7.11.4 Member Data Documentation

7.11.4.1 BrotherLabelSetSize

```
constexpr std::size_t BrotherLabelSetSize  [static], [constexpr]
```

Size of the LabelSet of the brother node.

7.11.4.2 DIM_HALF_SIZE

```
constexpr std::size_t DIM_HALF_SIZE  [static], [constexpr]
```

The last index of the left child.

7.11.4.3 DIM_LEFT_SIZE

```
constexpr std::size_t DIM_LEFT_SIZE  [static], [constexpr]
```

Size of the LabelSet of the left child node.

7.11.4.4 DIM_RIGHT_SIZE

```
constexpr std::size_t DIM_RIGHT_SIZE  [static], [constexpr]
```

Size of the LabelSet of the right child node.

7.11.4.5 IsFirstChild

```
constexpr bool IsFirstChild  [static], [constexpr]
```

Checks if node is a child of the the root node.

7.11.4.6 IsLeaf

```
constexpr bool IsLeaf  [static], [constexpr]
```

Checks if node is a leaf node.

7.11.4.7 IsRoot

```
constexpr bool IsRoot  [static], [constexpr]
```

Checks if node is the root node.

7.11.4.8 LabelSetSize

```
constexpr std::size_t LabelSetSize  [static], [constexpr]
```

Size of the LabelSet.

7.11.4.9 left

```
Left_Node_Type left
```

Left child node.

7.11.4.10 mDeltaSet

```
std::array<int,BrotherLabelSetSize> mDeltaSet
```

Array of size BrotherLabelSetSize, with set of indices used for identification of the brother of TnsNode.

7.11.4.11 mGramian

```
Hessian_Type mGramian
```

Gramian Matrix.

7.11.4.12 mKey

```
int mKey
```

Used for mapping Factors with leafs.

7.11.4.13 mLabelSet

```
std::array<int,LabelSetSize> mLabelSet
```

Array of size LabelSetSize, with set of indices used for identification of TnsNode.

7.11.4.14 mTnsDims

```
std::array<int,TnsSize> mTnsDims
```

Array of size TnsSize, with the size of every dimension of the Tensor mTnsX.

7.11.4.15 mTnsX

```
Tensor_Type mTnsX
```

Tensor of TnsNode.

7.11.4.16 mUpdated

```
bool mUpdated
```

Checks if node has updated data.

7.11.4.17 parent

```
I_TnsNode* parent
```

Pointer for the parent node.

7.11.4.18 ParLabelSetSize

```
constexpr std::size_t ParLabelSetSize [static], [constexpr]
```

Size of the LabelSet of the parent node.

7.11.4.19 ParTnsSize

```
constexpr std::size_t ParTnsSize [static], [constexpr]
```

Tensor order of the parent node.

7.11.4.20 right

```
Right_Node_Type right
```

Right child node.

7.11.4.21 RootSize

```
constexpr std::size_t RootSize [static], [constexpr]
```

Size of the LabelSet of the root node.

7.11.4.22 TnsSize

```
constexpr std::size_t TnsSize [static], [constexpr]
```

Tensor order.

The documentation for this struct was generated from the following file:

- DimTrees.hpp

7.12 ExprTree< _TnsSize > Struct Template Reference

```
#include <DimTrees.hpp>
```

7.12.1 Detailed Description

template<std::size_t _TnsSize>
struct partensor::ExprTree< _TnsSize >

Container of the Dimension Tree.

Template Parameters

TnsSize	Tensor order.

Public Member Functions

- template< typename Array , typename ExprNode >
  void Create (Array const &aTnsDims, int const R, ExprNode &expr)
- template< typename Array >
  void Create (std::array< int, RootExprNode::LabelSetSize > &aLabelSet, Array const &aTnsDims, int const R, Tensor< static_cast< int >(TnsSize)> const &aTnsX)

Public Attributes

- RootExprNode root

Static Public Attributes

- static constexpr bool IsNull

7.12.2 Member Function Documentation

7.12.2.1 Create() [1/2]

```
void Create (
            Array const & aTnsDims,
            int const R,
            ExprNode & expr )  [inline]
```

Creates the `ExprNodes`, expcept the root and its children `ExprNodes`.

Parameters

aTnsDims	[in] Dimensions of the initial Tensor.
R	[in] Rank of the factorization.
expr	[in] Newly Created `ExprNode`.

7.12.2.2 Create() [2/2]

```
void Create (
            std::array< int, RootExprNode::LabelSetSize > & aLabelSet,
```

```
Array const & aTnsDims,
int const R,
Tensor< static_cast< int >(TnsSize)> const & aTnsX ) [inline]
```

Creates the root `ExprNode`, the left-right childs `ExprNodes` and calls the `overloaded Create` if needed for more `ExprNodes`.

Parameters

aLabelSet	[in] LabelSet of the root `ExprNode`.
aTnsDims	[in] Dimensions of the initial Tensor.
R	[in] Rank of the factorization.
aTnsX	[in] Initial Tensor of `Tensor` type.

7.12.3 Member Data Documentation

7.12.3.1 IsNull

```
constexpr bool IsNull [static], [constexpr]
```

Checks if TnsSize is greater than zero.

7.12.3.2 root

```
RootExprNode root
```

Root node of the Dimension tree.

The documentation for this struct was generated from the following file:

- DimTrees.hpp

7.13 Factor< FactorType > Struct Template Reference

```
#include <Tensor.hpp>
```

7.13.1 Detailed Description

template<typename FactorType>
struct partensor::Factor< FactorType >

Templated struct, which contains information about a factor. It is used in factorization algorithms.

Template Parameters

FactorType	Either `FactorDimTree` or 2-dimension `Tensor`.

The documentation for this struct was generated from the following file:

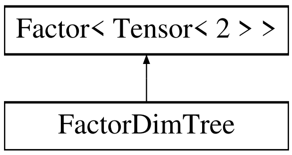
- Tensor.hpp

7.14 FactorDimTree Struct Reference

```
#include <DimTrees.hpp>
```

7.14.1 Detailed Description

Information about Factors and the associated leafs. `Factor` is created in `Tensor.hpp`. Inheritance diagram for FactorDimTree:



The documentation for this struct was generated from the following file:

- DimTrees.hpp

7.15 GTC< TnsSize_, execution::openmpi_policy > Struct Template Reference

```
#include <GtcMpi.hpp>
```

7.15.1 Detailed Description

template< std::size_t TnsSize_ >
struct partensor::v1::internal::GTC< TnsSize_, execution::openmpi_policy >

Includes the implementation of GTCMPI factorization. Based on the given parameters one of the four overloaded operators will be called.

Template Parameters

Tensor←_	The Type of The given `Tensor` to be factorized.

Inherits GTC_Base< TnsSize_ >.

Public Types

- using IntArray = typename SparseTensorTraits< SparseTensor >::IntArray

Public Member Functions

- void initialize_factors (Member_Variables &mv, Status &status)
- Status operator() (Matrix const &Ratings_Base_T, Options const &options)
- Status operator() (Options const &options)

7.15.2 Member Typedef Documentation

7.15.2.1 IntArray

```
using IntArray = typename SparseTensorTraits<SparseTensor>::IntArray
```

Stl array of size TnsSize and containing int type.

7.15.3 Member Function Documentation

7.15.3.1 initialize_factors()

```
void initialize_factors (
            Member_Variables & mv,
            Status & status )  [inline]
```

Initialization of factors.

7.15.3.2 operator()() [1/2]

```
Status operator() (
            Matrix const & Ratings_Base_T,
            Options const & options )  [inline]
```

Implementation of CP Decomposition with default values in Options struct and randomly generated initial factors.

Parameters

tnsX	[in] The given Tensor to be factorized of `Tensor_` type, with `double` data.
R	[in] The rank of decomposition.

Returns

An object of type `Status` with the results of the algorithm.

7.15.3.3 operator()() [2/2]

```
Status operator() (
            Options const & options )  [inline]
```

Implementation of CP Decomposition with default values in Options struct and randomly generated initial factors.

Parameters

tnsX	[in] The given Tensor to be factorized of `Tensor_` type, with `double` data.
R	[in] The rank of decomposition.

Returns

An object of type `Status` with the results of the algorithm.

The documentation for this struct was generated from the following file:

- GtcMpi.hpp

7.16 GTC_STOCHASTIC< TnsSize_, execution::openmpi_policy > Struct Template Reference

```
#include <GtcStochasticMpi.hpp>
```

7.16.1 Detailed Description

template<std::size_t TnsSize_>
struct partensor::v1::internal::GTC_STOCHASTIC< TnsSize_, execution::openmpi_policy >

Includes the implementation of GTCSTOCHASTICMPI factorization. Based on the given parameters one of the four overloaded operators will be called.

Template Parameters

Tensor←_	The Type of The given `Tensor` to be factorized.

Inherits GTC_STOCHASTIC_Base< TnsSize_ >.

Public Types

- using IntArray = typename SparseTensorTraits< SparseTensor >::IntArray

Public Member Functions

- void initialize_factors (Member_Variables &mv, Status &status)
- Status operator() (Matrix const &Ratings_Base_T, Options const &options)
- Status operator() (Options const &options)

7.16.2 Member Typedef Documentation

7.16.2.1 IntArray

```
using IntArray = typename SparseTensorTraits<SparseTensor>::IntArray
```

Stl array of size TnsSize and containing int type.

7.16.3 Member Function Documentation

7.16.3.1 initialize_factors()

```
void initialize_factors (
            Member_Variables & mv,
            Status & status )  [inline]
```

Initialization of factors.

7.16.3.2 operator()() [1/2]

```
Status operator() (
            Matrix const & Ratings_Base_T,
            Options const & options )  [inline]
```

Implementation of CP Decomposition with default values in Options struct and randomly generated initial factors.

Parameters

tnsX	[in] The given Tensor to be factorized of `Tensor_` type, with `double` data.
R	[in] The rank of decomposition.

Returns

An object of type `Status` with the results of the algorithm.

7.16.3.3 operator()() [2/2]

```
Status operator() (
            Options const & options )  [inline]
```

Implementation of CP Decomposition with default values in Options struct and randomly generated initial factors.

Parameters

tnsX	[in] The given Tensor to be factorized of `Tensor_` type, with `double` data.
R	[in] The rank of decomposition.

Returns

An object of type `Status` with the results of the algorithm.

The documentation for this struct was generated from the following file:

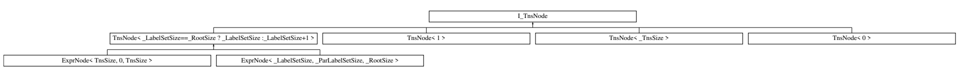
- GtcStochasticMpi.hpp

7.17 I_TnsNode Struct Reference

`#include <DimTrees.hpp>`

7.17.1 Detailed Description

Interface for each Node of the dimension Tree. Contains Tensor order and virtual methods for static and dynamic elements. Inheritance diagram for I_TnsNode:



Public Member Functions

- virtual void * DeltaSet ()=0
- virtual Tensor< 2 > & Gramian ()=0
- I_TnsNode (std::size_t _TnsSize)
- virtual int Key ()=0
- virtual void * LabelSet ()=0
- virtual I_TnsNode * Left ()=0
- virtual I_TnsNode * Parent ()=0
- virtual I_TnsNode * Right ()=0
- virtual I_TnsNode * SearchKey (int const key)=0
- virtual void SetOutdated ()=0
- virtual void * TensorX ()=0
- virtual void * TnsDims ()=0
- virtual bool Updated ()=0
- virtual void UpdateTree (int const num_factors, int const id, FactorDimTree *factors)=0

Public Attributes

- std::size_t TnsSize

7.17.2 Constructor & Destructor Documentation

7.17.2.1 I_TnsNode()

```
I_TnsNode (
        std::size_t _TnsSize ) [inline]
```

Initialize tensor order.

Parameters

_TnsSize	[in] Tensor Order.

7.17.3 Member Function Documentation

7.17.3.1 DeltaSet()

```
virtual void * DeltaSet ( ) [pure virtual]
```

A pure virtual member.

Implemented in ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >, ExprNode< TnsSize, 0, TnsSize >, and TnsNode< 0 >.

7.17.3.2 Gramian()

`virtual Tensor< 2 > & Gramian ( ) [pure virtual]`

A pure virtual member.

Implemented in TnsNode< _TnsSize >, TnsNode< _LabelSetSize==_RootSize ? _LabelSetSize :_LabelSetSize+1 >, TnsNode< 1 >, and TnsNode< 0 >.

7.17.3.3 Key()

`virtual int Key ( ) [pure virtual]`

A pure virtual member.

Implemented in TnsNode< _TnsSize >, TnsNode< _LabelSetSize==_RootSize ? _LabelSetSize :_LabelSetSize+1 >, TnsNode< 1 >, and TnsNode< 0 >.

7.17.3.4 LabelSet()

`virtual void * LabelSet ( ) [pure virtual]`

A pure virtual member.

Implemented in ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >, ExprNode< TnsSize, 0, TnsSize >, and TnsNode< 0 >.

7.17.3.5 Left()

`virtual I_TnsNode * Left ( ) [pure virtual]`

A pure virtual member.

Implemented in ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >, ExprNode< TnsSize, 0, TnsSize >, and TnsNode< 0 >.

7.17.3.6 Parent()

`virtual I_TnsNode * Parent ( ) [pure virtual]`

A pure virtual member.

Implemented in ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >, ExprNode< TnsSize, 0, TnsSize >, and TnsNode< 0 >.

7.17.3.7 Right()

```
virtual I_TnsNode * Right ( )  [pure virtual]
```

A pure virtual member.

Implemented in ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >, ExprNode< TnsSize, 0, TnsSize >, and TnsNode< 0 >.

7.17.3.8 SearchKey()

```
virtual I_TnsNode * SearchKey (
            int const key )  [pure virtual]
```

A pure virtual member.

Parameters

key	[in] Searching key value.

Implemented in ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >, ExprNode< TnsSize, 0, TnsSize >, and TnsNode< 0 >.

7.17.3.9 SetOutdated()

```
virtual void SetOutdated ( )  [pure virtual]
```

A pure virtual member.

Implemented in TnsNode< _TnsSize >, TnsNode< _LabelSetSize==_RootSize ? _LabelSetSize :_LabelSetSize+1 >, TnsNode< 1 >, and TnsNode< 0 >.

7.17.3.10 TensorX()

```
virtual void * TensorX ( )  [pure virtual]
```

A pure virtual member.

Implemented in TnsNode< _TnsSize >, TnsNode< _LabelSetSize==_RootSize ? _LabelSetSize :_LabelSetSize+1 >, TnsNode< 1 >, and TnsNode< 0 >.

7.17.3.11 TnsDims()

```
virtual void * TnsDims ( )  [pure virtual]
```

A pure virtual member.

Implemented in ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >, ExprNode< TnsSize, 0, TnsSize >, and TnsNode< 0 >.

7.17.3.12 Updated()

```
virtual bool Updated ( )  [pure virtual]
```

A pure virtual member.

Implemented in TnsNode< _TnsSize >, TnsNode< _LabelSetSize==_RootSize ? _LabelSetSize :_LabelSetSize+1 >, TnsNode< 1 >, and TnsNode< 0 >.

7.17.3.13 UpdateTree()

```
virtual void UpdateTree (
            int const num_factors,
            int const id,
            FactorDimTree * factors )  [pure virtual]
```

A pure virtual member.

Parameters

num_factors	[in] Number of Factors to update.
id	[in] Indexing for Factor to update.
factors	[in,out] Pointer to the array factors of FactorDimTree type.

Implemented in ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >, ExprNode< TnsSize, 0, TnsSize >, and TnsNode< 0 >.

7.17.4 Member Data Documentation

7.17.4.1 TnsSize

```
std::size_t TnsSize
```

Tensor Order

The documentation for this struct was generated from the following file:

- DimTrees.hpp

7.18 MatrixArrayTraits< MA > Struct Template Reference

```
#include <Tensor.hpp>
```

7.18.1 Detailed Description

template<typename MA>
struct partensor::MatrixArrayTraits< MA >

Initialization of a templated struct. It is being used to hold information about a container of `Eigen` Matrix type.

Template Parameters

MA	An array container type.

The documentation for this struct was generated from the following file:

- Tensor.hpp

7.19 MatrixArrayTraits< std::array< T, _Size > > Struct Template Reference

```
#include <Tensor.hpp>
```

7.19.1 Detailed Description

template<typename T, std::size_t _Size>
struct partensor::MatrixArrayTraits< std::array< T, _Size > >

Specialization of templated struct `MatrixArrayTraits`, for
`stl` array.

Template Parameters

T	Type of `Eigen` Data.
_Size	Size of the `stl array`.

The documentation for this struct was generated from the following file:

- Tensor.hpp

7.20 MatrixTraits< Matrix > Struct Template Reference

7.20.1 Detailed Description

template<typename Matrix>
struct partensor::MatrixTraits< Matrix >

Initialization of a templated struct. It is being used to hold information about `Eigen` Matrix.

Template Parameters

Matrix	`Eigen` Matrix Type.

The documentation for this struct was generated from the following file:

- Tensor.hpp

7.21 MatrixTraits< Matrix > Struct Reference

```
#include <Tensor.hpp>
```

7.21.1 Detailed Description

Specialization of templated struct `MatrixTraits`.

The documentation for this struct was generated from the following file:

- Tensor.hpp

7.22 Options< Tensor_, ExecutionPolicy_, DefaultValues_ > Struct Template Reference

```
#include <PARTENSOR_basic.hpp>
```

7.22.1 Detailed Description

template<typename Tensor_, typename ExecutionPolicy_ = execution::sequenced_policy, template< typename T > class DefaultValues_ = DefaultValues>
struct partensor::Options< Tensor_, ExecutionPolicy_, DefaultValues_ >

Manage defaults parameters for CPD algorithm.

In case different parameters values need to be used, an `Option` object must be created. After changing the default values, then this object can be passed in the appropriate `cpd` operation.

Template Parameters

*Tensor_*	Type(data type and order) of input Tensor.
*Execution↩ Policy_*	The policy that is used, either sequential or parallel with mpi.

Inherited by Options< Tensor_, execution::openmpi_policy, DefaultValues >.

Public Types

- using DataType = typename TensorTraits< Tensor_ >::DataType
- using MatrixType = typename TensorTraits< Tensor_ >::MatrixType

Public Member Functions

- Options ()

Static Public Attributes

- static constexpr std::size_t TnsSize

7.22.2 Member Typedef Documentation

7.22.2.1 DataType

```
using DataType = typename TensorTraits<Tensor_>::DataType
```

Tensor Data type.

7.22.2.2 MatrixType

```
using MatrixType = typename TensorTraits<Tensor_>::MatrixType
```

`Eigen` Matrix with the same Data type with the Tensor.

7.22.3 Constructor & Destructor Documentation

7.22.3.1 Options()

`Options ( ) [inline]`

< Default value for Method is als.
< Default value for Constraint is no negative. < Default value for cost function's threshold. < Default value for Nesterov's tolerance.
< Default value for Nesterov's tolerance.
< Default value for lambda.
< Default value outer loop maximum iterations. < Default value outer loop maximum duration.
< Default value for acceleration coefficient. < Default value for acceleration fail. < Default value for acceleration.
< Default value for normalization.

7.22.4 Member Data Documentation

7.22.4.1 TnsSize

`constexpr std::size_t TnsSize [static], [constexpr]`

Tensor Order.

The documentation for this struct was generated from the following file:

- PARTENSOR_basic.hpp

7.23 SparseStatus< _TnsSize, ExecutionPolicy_, DefaultValues_ > Struct Template Reference

```
#include <PARTENSOR_basic.hpp>
```

7.23.1 Detailed Description

template<std::size_t _TnsSize, typename ExecutionPolicy_ = execution::sequenced_policy, template< typename T > class DefaultValues_ = SparseDefaultValues>
struct partensor::SparseStatus< _TnsSize, ExecutionPolicy_, DefaultValues_ >

Sparse Status

Public Types

- using MatrixArray = typename SparseTensorTraits< SparseTensor >::MatrixArray

Public Attributes

- unsigned ao_iter
- double f_value
- MatrixArray factors
- double frob_tns
- double rel_costFunction

7.23.2 Member Typedef Documentation

7.23.2.1 MatrixArray

```
using MatrixArray = typename SparseTensorTraits<SparseTensor>::MatrixArray
```

`Eigen` Matrix with the same Data type with the Tensor.

7.23.3 Member Data Documentation

7.23.3.1 ao_iter

```
unsigned ao_iter
```

Stores the iteration where the cost function reached the wanted threshold.

7.23.3.2 f_value

```
double f_value
```

Stores the cost function.

7.23.3.3 factors

```
MatrixArray factors
```

An stl array with the resulting Factors from CPD factorization of the Eigen Tensor.

7.23.3.4 frob_tns

```
double frob_tns
```

Stores the Frobenius norm of an `Eigen` Tensor.

7.23.3.5 rel_costFunction

`double rel_costFunction`

Stores the relative cost function.

The documentation for this struct was generated from the following file:

- PARTENSOR_basic.hpp

7.24 SparseTensorTraits< SparseTensor > Struct Template Reference

7.24.1 Detailed Description

template<typename SparseTensor>
struct partensor::SparseTensorTraits< SparseTensor >

Initialization of a templated struct with information about an `Sparse Eigen`.

Template Parameters

c Sparse `Eigen` Tensor Type.

The documentation for this struct was generated from the following file:

- Tensor.hpp

7.25 SparseTensorTraits< SparseTensor< _TnsSize > > Struct Template Reference

`#include <Tensor.hpp>`

7.25.1 Detailed Description

template<std::size_t _TnsSize>
struct partensor::SparseTensorTraits< SparseTensor< _TnsSize > >

Specialization of templated struct `SparseMatrixTraits`.

Template Parameters

_TnsSize	Tensor Order.

Public Types

- using Constraints = std::array< Constraint, _TnsSize >
- using DoubleArray = std::array< double, _TnsSize >
- using IntArray = std::array< int, _TnsSize >
- using MatrixArray = std::array< MatrixType, _TnsSize >

7.25.2 Member Typedef Documentation

7.25.2.1 Constraints

```
using Constraints = std::array<Constraint,_TnsSize>
```

Stl array of size TnsSize and containing Constraint type.

7.25.2.2 DoubleArray

```
using DoubleArray = std::array<double,_TnsSize>
```

Stl array of size TnsSize and containing double type.

7.25.2.3 IntArray

```
using IntArray = std::array<int,_TnsSize>
```

Stl array of size TnsSize and containing int type.

7.25.2.4 MatrixArray

```
using MatrixArray = std::array<MatrixType,_TnsSize>
```

Stl array of size TnsSize and containing MatrixType type.

The documentation for this struct was generated from the following file:

- Tensor.hpp

7.26 Status< Tensor_, ExecutionPolicy_, DefaultValues_ > Struct Template Reference

```
#include <PARTENSOR_basic.hpp>
```

7.26.1 Detailed Description

template<typename Tensor_, typename ExecutionPolicy_ = execution::sequenced_policy, template< typename T > class DefaultValues_ = DefaultValues>
struct partensor::Status< Tensor_, ExecutionPolicy_, DefaultValues_ >

Returned Type of CPD algorithm.

`Status` is the returned type of `cpd` operations. In this struct exist the returned values, such as the `cost function` at the end of the algorithm, or at what `iteration` the operation has ended. Also, includes the factors produced from `cpd` operation in an `stl` array of `Matrix` type and size same as the input Tensor order.

Template Parameters

*Tensor_*	Type(data type and order) of input Tensor.
ExecutionPolicy	The policy that is used, either sequential or with mpi.

Public Types

- using MatrixArray = typename TensorTraits< Tensor_ >::MatrixArray

Public Attributes

- unsigned ao_iter
- double f_value
- MatrixArray factors
- double frob_tns
- double rel_costFunction

7.26.2 Member Typedef Documentation

7.26.2.1 MatrixArray

```
using MatrixArray = typename TensorTraits<Tensor_>::MatrixArray
```

`Eigen` Matrix with the same Data type with the Tensor.

7.26.3 Member Data Documentation

7.26.3.1 ao_iter

```
unsigned ao_iter
```

Stores the iteration where the cost function reached the wanted threshold.

7.26.3.2 f_value

```
double f_value
```

Stores the cost function.

7.26.3.3 factors

```
MatrixArray factors
```

An stl array with the resulting Factors from CPD factorization of the Eigen Tensor.

7.26.3.4 frob_tns

```
double frob_tns
```

Stores the Frobenius norm of an `Eigen` Tensor.

7.26.3.5 rel_costFunction

```
double rel_costFunction
```

Stores the relative cost function.

The documentation for this struct was generated from the following file:

- PARTENSOR_basic.hpp

7.27 TensorTraits< Tensor > Struct Template Reference

7.27.1 Detailed Description

template<typename Tensor>
struct partensor::TensorTraits< Tensor >

Initialization of a templated struct with information about an `Eigen` Tensor.

Template Parameters

Tensor	`Eigen` Tensor Type.

The documentation for this struct was generated from the following file:

- Tensor.hpp

7.28 TensorTraits< Tensor< _TnsSize > > Struct Template Reference

```
#include <Tensor.hpp>
```

7.28.1 Detailed Description

template< int _TnsSize >
struct partensor::TensorTraits< Tensor< _TnsSize > >

Specialization of templated struct `TensorTraits`.

Template Parameters

_TnsSize	Tensor Order.

Public Types

- using Constraints = std::array< Constraint, _TnsSize >
- using DoubleArray = std::array< double, _TnsSize >
- using IntArray = std::array< int, _TnsSize >
- using MatrixArray = std::array< MatrixType, _TnsSize >

7.28.2 Member Typedef Documentation

7.28.2.1 Constraints

```
using Constraints = std::array<Constraint,_TnsSize>
```

Stl array of size TnsSize and containing Constraint type.

7.28.2.2 DoubleArray

```
using DoubleArray = std::array<double,_TnsSize>
```

Stl array of size TnsSize and containing double type.

7.28.2.3 IntArray

```
using IntArray = std::array<int,_TnsSize>
```

Stl array of size TnsSize and containing int type.

7.28.2.4 MatrixArray

```
using MatrixArray = std::array<MatrixType,_TnsSize>
```

Stl array of size TnsSize and containing MatrixType type.

The documentation for this struct was generated from the following file:

- Tensor.hpp

7.29 Timers Struct Reference

```
#include <Timers.hpp>
```

7.29.1 Detailed Description

Struct with implementations for measuring time from libraries as `time`, `stl` chrono, and `MPI`.

Public Types

- using ClockHigh = std::chrono::time_point< std::chrono::high_resolution_clock >
- using ClockSteady = std::chrono::time_point< std::chrono::steady_clock >

Public Member Functions

- double endChronoHighTimer ()
- double endChronoSteadyTimer ()
- double endCpuTimer ()
- double endMpiTimer ()
- void startChronoHighTimer ()
- void startChronoSteadyTimer ()
- void startCpuTimer ()
- void startMpiTimer ()

7.29.2 Member Typedef Documentation

7.29.2.1 ClockHigh

```
using ClockHigh = std::chrono::time_point<std::chrono::high_resolution_clock>
```

Typdef for chrono high resolution clock.

7.29.2.2 ClockSteady

```
using ClockSteady = std::chrono::time_point<std::chrono::steady_clock>
```

Typdef for chrono steady clock.

7.29.3 Member Function Documentation

7.29.3.1 endChronoHighTimer()

```
double endChronoHighTimer ( ) [inline]
```

Stores in a variable of `double` type, the seconds passed since `startChronoHighTimer` was called.

Returns

The measured time passed.

7.29.3.2 endChronoSteadyTimer()

```
double endChronoSteadyTimer ( ) [inline]
```

Stores in a variable of `double` type, the seconds passed since `startChronoSteadyTimer` was called.

Returns

The measured time passed.

7.29.3.3 endCpuTimer()

```
double endCpuTimer ( ) [inline]
```

Stores in a variable of `double` type, the seconds passed since `startCpuTimer` was called.

Returns

The measured time passed.

7.29.3.4 endMpiTimer()

```
double endMpiTimer ( ) [inline]
```

Stores in a variable of `double` type, the seconds passed since `startMpiTimer` was called.

Returns

The measured time passed.

7.29.3.5 startChronoHighTimer()

```
void startChronoHighTimer ( ) [inline]
```

Stores in a variable of `chrono high_resolution_clock` type, the current time.

7.29.3.6 startChronoSteadyTimer()

```
void startChronoSteadyTimer ( ) [inline]
```

Stores in a variable of `chrono steady_clock` type, the current point in time.

7.29.3.7 startCpuTimer()

```
void startCpuTimer ( ) [inline]
```

Stores in a variable of `clock_t` type, the processor's time.

7.29.3.8 startMpiTimer()

```
void startMpiTimer ( ) [inline]
```

Stores in a variable of `double` type, the elapsed time on the calling `MPI` processor.

The documentation for this struct was generated from the following file:

- Timers.hpp

7.30 TnsNode< _TnsSize > Struct Template Reference

```
#include <DimTrees.hpp>
```

7.30.1 Detailed Description

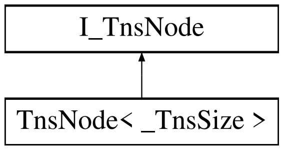
template<std::size_t _TnsSize>
struct partensor::TnsNode< _TnsSize >

Implementation of the `I_TnsNode` interface.

Template Parameters

_TnsSize	Tensor order

Inheritance diagram for TnsNode< _TnsSize >:



Public Member Functions

- Hessian_Type & Gramian () override
- int Key () override
- void SetOutdated () override
- void * TensorX () override
- TnsNode ()
- bool Updated () override

Public Attributes

- Hessian_Type mGramian
- int mKey
- Tensor_Type mTnsX
- bool mUpdated

Static Public Attributes

- static constexpr bool IsNull
- static constexpr std::size_t TnsSize

7.30.2 Constructor & Destructor Documentation

7.30.2.1 TnsNode()

`TnsNode ( ) [inline]`

Initialize tensor order. Set `mKey` to zero, and `mUpdated` to false.

7.30.3 Member Function Documentation

7.30.3.1 Gramian()

```
Hessian_Type & Gramian ( ) [inline], [override], [virtual]
```

Returns

The `mGramian` member variable of `TnsNode`.

Implements I_TnsNode.

7.30.3.2 Key()

```
int Key ( ) [inline], [override], [virtual]
```

Returns

The `mKey` member variable of `TnsNode`.

Implements I_TnsNode.

7.30.3.3 SetOutdated()

```
void SetOutdated ( ) [inline], [override], [virtual]
```

Set the `TnsNode` and its children as outdated.

Implements I_TnsNode.

7.30.3.4 TensorX()

```
void * TensorX ( ) [inline], [override], [virtual]
```

Returns

The `mTnsX` member variable of `TnsNode`. No type is included, in order to specify the `TnsSize` of `mTnsX` explicitly.

Implements I_TnsNode.

7.30.3.5 Updated()

```
bool Updated ( ) [inline], [override], [virtual]
```

Returns

If the `ExprNode` has updated data then `true` is returned, otherwise `false`.

Implements I_TnsNode.

7.30.4 Member Data Documentation

7.30.4.1 IsNull

```
constexpr bool IsNull [static], [constexpr]
```

Checks if TnsSize_ is greater than zero.

7.30.4.2 mGramian

```
Hessian_Type mGramian
```

Gramian Matrix.

7.30.4.3 mKey

```
int mKey
```

Used for mapping Factors with leafs.

7.30.4.4 mTnsX

```
Tensor_Type mTnsX
```

Tensor of TnsNode.

7.30.4.5 mUpdated

```
bool mUpdated
```

Checks if node has updated data.

7.30.4.6 TnsSize

```
constexpr std::size_t TnsSize  [static], [constexpr]
```

Tensor Order.

The documentation for this struct was generated from the following file:

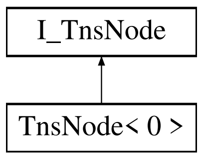
- DimTrees.hpp

7.31 TnsNode< 0 > Struct Reference

```
#include <DimTrees.hpp>
```

7.31.1 Detailed Description

TnsNode with TnsSize set as zero. Used for leaf nodes. Inheritance diagram for TnsNode< 0 >:



Public Types

- using Tensor_Type = Tensor< 0 >

Public Member Functions

- void ∗ DeltaSet () override
- Tensor< 2 > & Gramian () override
- int Key () override
- void ∗ LabelSet () override
- I_TnsNode ∗ Left () override
- I_TnsNode ∗ Parent () override
- I_TnsNode ∗ Right () override
- I_TnsNode ∗ SearchKey (int const) override
- void SetOutdated () override
- void ∗ TensorX () override
- void ∗ TnsDims () override
- template<typename N >
  TnsNode (N par=nullptr)
- bool Updated () override
- void UpdateTree (int const, int const, FactorDimTree ∗) override

Static Public Attributes

- static constexpr bool IsNull
- static constexpr std::size_t TnsSize

Additional Inherited Members

7.31.2 Member Typedef Documentation

7.31.2.1 Tensor_Type

```
using Tensor_Type = Tensor<0>
```

Scalar value.

7.31.3 Constructor & Destructor Documentation

7.31.3.1 TnsNode()

```
TnsNode (
            N par = nullptr )  [inline]
```

Initialize tensor order to zero.

7.31.4 Member Function Documentation

7.31.4.1 DeltaSet()

```
void * DeltaSet ( )  [inline], [override], [virtual]
```

If it is called throws a runtime error.

Implements I_TnsNode.

7.31.4.2 Gramian()

`Tensor< 2 > & Gramian ( ) [inline], [override], [virtual]`

If it is called throws a runtime error.

Implements I_TnsNode.

7.31.4.3 Key()

`int Key ( ) [inline], [override], [virtual]`

If it is called throws a runtime error.

Implements I_TnsNode.

7.31.4.4 LabelSet()

`void * LabelSet ( ) [inline], [override], [virtual]`

If it is called throws a runtime error.

Implements I_TnsNode.

7.31.4.5 Left()

`I_TnsNode * Left ( ) [inline], [override], [virtual]`

Returns

If called from zero- `TnsNode` , there is no Left Child `TnsNode` so it returns `nullptr`.

Implements I_TnsNode.

7.31.4.6 Parent()

`I_TnsNode * Parent ( ) [inline], [override], [virtual]`

Returns

If called from zero- `TnsNode` , there is no Parent `TnsNode` so it returns `nullptr`.

Implements I_TnsNode.

7.31.4.7 Right()

```
I_TnsNode * Right ( ) [inline], [override], [virtual]
```

Returns

If called from zero- `TnsNode` , there is no Right Child `TnsNode` so it returns `nullptr`.

Implements I_TnsNode.

7.31.4.8 SearchKey()

```
I_TnsNode * SearchKey (
            int const ) [inline], [override], [virtual]
```

If called from zero- `TnsNode` then throws a `runtime_error`.

Parameters

key	[in] Searching key value. If it is called throws a runtime error.

Implements I_TnsNode.

7.31.4.9 SetOutdated()

```
void SetOutdated ( ) [inline], [override], [virtual]
```

If it is called throws a runtime error.

Implements I_TnsNode.

7.31.4.10 TensorX()

```
void * TensorX ( ) [inline], [override], [virtual]
```

If it is called throws a runtime error.

Implements I_TnsNode.

7.31.4.11 TnsDims()

```
void * TnsDims ( )  [inline], [override], [virtual]
```

If it is called throws a runtime error.

Implements I_TnsNode.

7.31.4.12 Updated()

```
bool Updated ( )  [inline], [override], [virtual]
```

If it is called throws a runtime error.

Implements I_TnsNode.

7.31.4.13 UpdateTree()

```
void UpdateTree (
            int const ,
            int const ,
            FactorDimTree * )  [inline], [override], [virtual]
```

If called from zero- `TnsNode` then throws a `runtime_error`.

Parameters

num_factors	[in] Number of Factors to update.
id	[in] Indexing for Factor to update.
factors	[in,out] Pointer to the array factors of FactorDimTree type. If it is called throws a runtime error.

Implements I_TnsNode.

7.31.5 Member Data Documentation

7.31.5.1 IsNull

```
constexpr bool IsNull  [static], [constexpr]
```

Indicates that there are no data in this TnsNode.

7.31.5.2 TnsSize

```
constexpr std::size_t TnsSize [static], [constexpr]
```

TnsSize set as zero.

The documentation for this struct was generated from the following file:

- DimTrees.hpp

Chapter 8

File Documentation

8.1 Config.hpp File Reference

8.1.1 Detailed Description

Configurations for PARTENSOR.

8.2 Config.hpp

Go to the documentation of this file.

```

#if !defined (USE_MPI)
#define USE_MPI   0
#endif /* !defined (USE_MPI) */

#if !defined (USE_OPENMP)
#define USE_OPENMP   0
#endif /* !defined (USE_OPENMP) */

#if !defined (USE_CUDA)
#define USE_CUDA   0
#endif /* !defined (USE_CUDA) */
```

8.3 Constants.hpp File Reference

Enumerations

- enum class Constraint : uint8_t {
  unconstrained ,
  nonnegativity ,
  orthogonality ,
  sparsity ,
  constant ,
  symmetric_nonnegativity ,
  symmetric }
- enum class Distribution : uint8_t
- enum class Method : uint8_t
- enum class ProblemType : uint8_t

8.3.1 Detailed Description

Implementation for various enumerations.

8.3.2 Enumeration Type Documentation

8.3.2.1 Constraint

```
enum class Constraint : uint8_t [strong]
```

Possible implementation for each Factor.

Enumerator

unconstrained	unconstrained
nonnegativity	nonnegativity
orthogonality	orthogonality
sparsity	sparsity
constant	constant
symmetric_nonnegativity	symmetric_nonnegativity
symmetric	symmetric

8.3.2.2 Distribution

```
enum class Distribution : uint8_t [strong]
```

In case of not initialized Tensor or Factors, choose one of the following distribution in order to produce synthetic data.

8.3.2.3 Method

```
enum class Method : uint8_t [strong]
```

Based on which Method the factorization will be computed.

8.3.2.4 ProblemType

```
enum class ProblemType : uint8_t [strong]
```

Based on the format of the Tensor, different implementations of the algorithms are used.

8.4 Constants.hpp

Go to the documentation of this file.

```
#ifndef DOXYGEN_SHOULD_SKIP_THIS
#endif  // DOXYGEN_SHOULD_SKIP_THIS
/*********************************************************************/
#ifndef PARTENSOR_CONSTANTS_HPP
#define PARTENSOR_CONSTANTS_HPP

namespace partensor {

        enum class ProblemType : uint8_t {
          dense      = 0,
          sparse     = 1,
          incomplete = 2
        };

        enum class Method : uint8_t {
                als = 0,  // alternating least squares
                rnd = 1,  // randomized
                bc  = 2   // block coordinate descent
        };

        enum class Constraint : uint8_t {
                unconstrained = 0,
                nonnegativity = 1,
                orthogonality = 2,
                sparsity      = 3,
                constant      = 4,
                symmetric_nonnegativity = 5,
                symmetric = 6
        };

        enum class Distribution : uint8_t {
          uniform  = 0,
          gaussian = 1
        };

} // end namespace partensor

#endif // PARTENSOR_CONSTANTS_HPP
```

8.5 Cpd.hpp File Reference

```
#include "PARTENSOR_basic.hpp"
#include "Matricization.hpp"
#include "PartialCwiseProd.hpp"
#include "MTTKRP.hpp"
#include "NesterovMNLS.hpp"
#include "Normalize.hpp"
#include "Timers.hpp"
#include "ReadWrite.hpp"
```

Functions

- template< std::size_t TnsSize, typename ExecutionPolicy >
  execution::internal::enable_if_execution_policy< ExecutionPolicy, Status< Tensor< static_cast< int >(TnsSize)>, execution::execution_policy_t< ExecutionPolicy >, DefaultValues > > cpd (ExecutionPolicy &&, std::array< int, TnsSize > const &tnsDims, std::size_t const R, std::array< std::string, TnsSize > const &true_paths, std::array< std::string, TnsSize > const &init_paths, Options< Tensor< static_cast< int >(TnsSize)>, execution::execution_policy_t< ExecutionPolicy >, DefaultValues > const &options)
- template<typename ExecutionPolicy , std::size_t TnsSize>
  execution::internal::enable_if_execution_policy< ExecutionPolicy, Status< Tensor< static_cast< int >(TnsSize)>, execution::execution_policy_t< ExecutionPolicy >, DefaultValues > > cpd (ExecutionPolicy &&, std::array< int, TnsSize > const &tnsDims, std::size_t const R, std::array< std::string, TnsSize+1 > const &paths)

- template<std::size_t TnsSize, typename ExecutionPolicy >
execution::internal::enable_if_execution_policy< ExecutionPolicy, Status< Tensor< static_cast< int >(TnsSize)>, execution::execution_policy_t< ExecutionPolicy >, DefaultValues > > cpd (Execution↩ Policy &&, std::array< int, TnsSize > const &tnsDims, std::size_t const R, std::array< std::string, Tns↩ Size+1 > const &paths, Options< Tensor< static_cast< int >(TnsSize)>, execution::execution_policy_t< ExecutionPolicy >, DefaultValues > const &options)
- template<typename ExecutionPolicy , std::size_t TnsSize>
execution::internal::enable_if_execution_policy< ExecutionPolicy, Status< Tensor< static_cast< int >(TnsSize)>, execution::execution_policy_t< ExecutionPolicy >, DefaultValues > > cpd (Execution↩ Policy &&, std::array< int, TnsSize > const &tnsDims, std::size_t const R, std::string const &path)
- template<typename ExecutionPolicy , std::size_t TnsSize>
execution::internal::enable_if_execution_policy< ExecutionPolicy, Status< Tensor< static_cast< int >(TnsSize)>, execution::execution_policy_t< ExecutionPolicy >, DefaultValues > > cpd (Execution↩ Policy &&, std::array< int, TnsSize > const &tnsDims, std::size_t const R, std::string const &path, Options< Tensor< static_cast< int >(TnsSize)>, execution::execution_policy_t< ExecutionPolicy >, DefaultValues > const &options)
- template<typename Tensor_ , typename ExecutionPolicy >
execution::internal::enable_if_execution_policy< ExecutionPolicy, Status< Tensor_, execution::execution_↩ policy_t< ExecutionPolicy >, DefaultValues > > cpd (ExecutionPolicy &&, Tensor_ const &tnsX, std::size_t const R)
- template<typename ExecutionPolicy , typename Tensor_ >
execution::internal::enable_if_execution_policy< ExecutionPolicy, Status< Tensor_, execution::execution_↩ policy_t< ExecutionPolicy >, DefaultValues > > cpd (ExecutionPolicy &&, Tensor_ const &tnsX, std::size_t const R, MatrixArray< Tensor_ > const &factorsInit)
- template<typename Tensor_ , typename ExecutionPolicy >
execution::internal::enable_if_execution_policy< ExecutionPolicy, Status< Tensor_, execution::execution_↩ policy_t< ExecutionPolicy >, DefaultValues > > cpd (ExecutionPolicy &&, Tensor_ const &tnsX, std::size↩ _t const R, Options< Tensor_, execution::execution_policy_t< ExecutionPolicy >, DefaultValues > const &options)
- template<typename ExecutionPolicy , typename Tensor_ >
execution::internal::enable_if_execution_policy< ExecutionPolicy, Status< Tensor_, execution::execution_↩ policy_t< ExecutionPolicy >, DefaultValues > > cpd (ExecutionPolicy &&, Tensor_ const &tnsX, std::size↩ _t const R, Options< Tensor_, execution::execution_policy_t< ExecutionPolicy >, DefaultValues > const &options, MatrixArray< Tensor_ > const &factorsInit)

8.5.1 Detailed Description

Implements the Canonical Polyadic Decomposition(cpd). Make use of `spdlog` library in order to write output in a log file in `"../log"`. In case of using parallelism with mpi, then the functions from `CpdMpi.hpp` will be called.

8.5.2 Function Documentation

8.5.2.1 cpd() [1/9]

```
execution::internal::enable_if_execution_policy< ExecutionPolicy, Status< Tensor< static_↩
cast< int >(TnsSize)>, execution::execution_policy_t< ExecutionPolicy >, DefaultValues > >
partensor::cpd (
            ExecutionPolicy && ,
            std::array< int, TnsSize > const & tnsDims,
            std::size_t const R,
```

```
            std::array< std::string, TnsSize > const & true_paths,
            std::array< std::string, TnsSize > const & init_paths,
            Options< Tensor< static_cast< int >(TnsSize)>, execution::execution_policy_t<
ExecutionPolicy >, DefaultValues > const & options )
```

Interface of Canonical Polyadic Decomposition(cpd), with the use of an `Execution Policy`, which can be either `sequential`, `parallel` with the use of `MPI`, or `parallel` with the use of `OpenMP`. In order to choose a policy, type `execution::seq`, `execution::mpi` or `execution::omp`. Default value is `sequential`, in case no `ExecutionPolicy` is passed.

With this version of `cpd`, the Tensor can be read from a file, specified in `path` variable.

Template Parameters

ExecutionPolicy	Type of `stl Execution Policy` (sequential, parallel-mpi).
TnsSize	Order of input Tensor.

Parameters

tnsDims	[in] `Stl` array containing the Tensor dimensions, whose length must be same as the Tensor order.
R	[in] The rank of decomposition.
true_paths	[in] An `stl` array containing paths for the true factors.
init_paths	[in] An `stl` array containing paths for initialized factors.
options	[in] User's `options`, other than the default. It must be of `partensor::Options<partensor::Tensor<order>>` type, where `order` must be in range of [3-8].

Returns

An object of type `Status` with the results of the algorithm.

8.5.2.2 cpd() [2/9]

```
execution::internal::enable_if_execution_policy< ExecutionPolicy, Status< Tensor< static_↩
cast< int >(TnsSize)>, execution::execution_policy_t< ExecutionPolicy >, DefaultValues > >
partensor::cpd (
            ExecutionPolicy && ,
            std::array< int, TnsSize > const & tnsDims,
            std::size_t const R,
            std::array< std::string, TnsSize+1 > const & paths )
```

Interface of Canonical Polyadic Decomposition(cpd), with the use of an `Execution Policy`, which can be either `sequential`, `parallel` with the use of `MPI`, or `parallel` with the use of `OpenMP`. In order to choose a policy, type `execution::seq`, `execution::mpi` or `execution::omp`. Default value is `sequential`, in case no `ExecutionPolicy` is passed.

With this version of `cpd`, the Tensor and the initialized factors can be read from files, specified in `paths` variable.

Template Parameters

ExecutionPolicy	Type of `stl Execution Policy` (sequential, parallel-mpi).
TnsSize	Order of input Tensor.

Parameters

tnsDims	[in] `Stl` array containing the Tensor dimensions, whose length must be same as the Tensor order.
R	[in] The rank of decomposition.
paths	[in] An `stl` array containing paths for the Tensor to be factorized and after that the paths for the initialized factors.

Returns

An object of type `Status` with the results of the algorithm.

8.5.2.3 cpd() [3/9]

```
execution::internal::enable_if_execution_policy< ExecutionPolicy, Status< Tensor< static_↩
cast< int >(TnsSize)>, execution::execution_policy_t< ExecutionPolicy >, DefaultValues > >
partensor::cpd (
            ExecutionPolicy && ,
            std::array< int, TnsSize > const & tnsDims,
            std::size_t const R,
            std::array< std::string, TnsSize+1 > const & paths,
            Options< Tensor< static_cast< int >(TnsSize)>, execution::execution_policy_t<
ExecutionPolicy >, DefaultValues > const & options )
```

Interface of Canonical Polyadic Decomposition(cpd), with the use of an `Execution Policy`, which can be either `sequential`, `parallel` with the use of `MPI`, or `parallel` with the use of `OpenMP`. In order to choose a policy, type `execution::seq`, `execution::mpi` or `execution::omp`. Default value is `sequential`, in case no `ExecutionPolicy` is passed.

With this version of `cpd`, the Tensor and the initialized factors can be read from files, specified in `paths` variable.

Template Parameters

ExecutionPolicy	Type of `stl Execution Policy` (sequential, parallel-mpi).
TnsSize	Order of input Tensor.

Parameters

tnsDims	[in] `Stl` array containing the Tensor dimensions, whose length must be same as the Tensor order.
R	[in] The rank of decomposition.
paths	[in] An `stl` array containing paths for the Tensor to be factorized and after that the paths for the initialized factors.

Parameters

options	[in] User's `options`, other than the default. It must be of `partensor::Options<partensor::Tensor<order>>` type, where `order` must be in range of [3-8].

Returns

An object of type `Status` with the results of the algorithm.

8.5.2.4 cpd() [4/9]

```
execution::internal::enable_if_execution_policy< ExecutionPolicy, Status< Tensor< static_↩
cast< int >(TnsSize)>, execution::execution_policy_t< ExecutionPolicy >, DefaultValues > >
partensor::cpd (
            ExecutionPolicy && ,
            std::array< int, TnsSize > const & tnsDims,
            std::size_t const R,
            std::string const & path )
```

Interface of Canonical Polyadic Decomposition(cpd), with the use of an `Execution Policy`, which can be either `sequential`, `parallel` with the use of `MPI`, or `parallel` with the use of `OpenMP`. In order to choose a policy, type `execution::seq`, `execution::mpi` or `execution::omp`. Default value is `sequential`, in case no `ExecutionPolicy` is passed.

With this version of `cpd` the Tensor can be read from a file, specified in `path` variable.

Template Parameters

ExecutionPolicy	Type of `stl Execution Policy` (sequential, parallel-mpi).
TnsSize	Order of input Tensor.

Parameters

tnsDims	[in] `Stl` array containing the Tensor dimensions, whose length must be same as the Tensor order.
R	[in] The rank of decomposition.
path	[in] The path where the tensor is located.

Returns

An object of type `Status` with the results of the algorithm.

8.5.2.5 cpd() [5/9]

```
execution::internal::enable_if_execution_policy< ExecutionPolicy, Status< Tensor< static_↩
cast< int >(TnsSize)>, execution::execution_policy_t< ExecutionPolicy >, DefaultValues > >
```

```
partensor::cpd (
            ExecutionPolicy && ,
            std::array< int, TnsSize > const & tnsDims,
            std::size_t const R,
            std::string const & path,
            Options< Tensor< static_cast< int >(TnsSize)>, execution::execution_policy_t<
ExecutionPolicy >, DefaultValues > const & options )
```

Interface of Canonical Polyadic Decomposition(cpd), with the use of an `Execution Policy`, which can be either `sequential`, `parallel` with the use of `MPI`, or `parallel` with the use of `OpenMP`. In order to choose a policy, type `execution::seq`, `execution::mpi` or `execution::omp`. Default value is `sequential`, in case no `ExecutionPolicy` is passed.

With this version of `cpd` the Tensor can be read from a file, specified in `path` variable.

Template Parameters

ExecutionPolicy	Type of `stl Execution Policy` (sequential, parallel-mpi).
TnsSize	Order of input Tensor.

Parameters

tnsDims	[in] `Stl` array containing the Tensor dimensions, whose length must be same as the Tensor order.
R	[in] The rank of decomposition.
path	[in] The path where the tensor is located.
options	[in] User's `options`, other than the default. It must be of `partensor::Options<partensor::Tensor<order>>` type, where `order` must be in range of [3-8].

Returns

An object of type `Status` with the results of the algorithm.

8.5.2.6 cpd() [6/9]

```
execution::internal::enable_if_execution_policy< ExecutionPolicy, Status< Tensor_, execution↩
::execution_policy_t< ExecutionPolicy >, DefaultValues > > partensor::cpd (
            ExecutionPolicy && ,
            Tensor_ const & tnsX,
            std::size_t const R )
```

Interface of Canonical Polyadic Decomposition(cpd), with the use of an `Execution Policy`, which can be either `sequential`, `parallel` with the use of `MPI`, or `parallel` with the use of `OpenMP`. In order to choose a policy, type `execution::seq`, `execution::mpi` or `execution::omp`. Default value is `sequential`, in case no `ExecutionPolicy` is passed.

Template Parameters

ExecutionPolicy	Type of `stl Execution Policy` (sequential, parallel-mpi).
*Tensor_*	Type(data type and order) of input Tensor. `Tensor_` must be `partensor::Tensor<order>`, where `order` must be in range of [3-8].

Parameters

tnsX	[in] The given Tensor to be factorized of `Tensor_` type, with `double` data.
R	[in] The rank of decomposition.

Returns

An object of type `Status`, containing the results of the algorithm.

8.5.2.7 cpd() [7/9]

```
execution::internal::enable_if_execution_policy< ExecutionPolicy, Status< Tensor_, execution↩
::execution_policy_t< ExecutionPolicy >, DefaultValues > > partensor::cpd (
            ExecutionPolicy && ,
            Tensor_ const & tnsX,
            std::size_t const R,
            MatrixArray< Tensor_ > const & factorsInit )
```

Interface of Canonical Polyadic Decomposition(cpd), with the use of an `Execution Policy`, which can be either `sequential`, `parallel` with the use of `MPI`, or `parallel` with the use of `OpenMP`. In order to choose a policy, type `execution::seq`, `execution::mpi` or `execution::omp`. Default value is `sequential`, in case no `ExecutionPolicy` is passed.

Template Parameters

ExecutionPolicy	Type of `stl Execution Policy` (sequential, parallel-mpi).
*Tensor_*	Type(data type and order) of input Tensor. `Tensor_` must be `partensor::Tensor<order>`, where `order` must be in range of [3-8].

Parameters

tnsX	[in] The given Tensor to be factorized of `Tensor_` type, with `double` data.
R	[in] The rank of decomposition.
factorsInit	[in] Uses initialized factors instead of randomly generated. The data must be of `partensor::Matrix` type and stored in an `stl` array with size same as the `order` of `tnsX`.

Returns

An object of type `Status` with the results of the algorithm.

8.5.2.8 cpd() [8/9]

```
execution::internal::enable_if_execution_policy< ExecutionPolicy, Status< Tensor_, execution↩
::execution_policy_t< ExecutionPolicy >, DefaultValues > > partensor::cpd (
```

```
            ExecutionPolicy && ,
            Tensor_ const & tnsX,
            std::size_t const R,
            Options< Tensor_, execution::execution_policy_t< ExecutionPolicy >, DefaultValues
> const & options )
```

Interface of Canonical Polyadic Decomposition(cpd), with the use of an `Execution Policy`, which can be either `sequential`, `parallel` with the use of `MPI`, or `parallel` with the use of `OpenMP`. In order to choose a policy, type `execution::seq`, `execution::mpi` or `execution::omp`. Default value is `sequential`, in case no `ExecutionPolicy` is passed.

Template Parameters

ExecutionPolicy	Type of `stl Execution Policy` (sequential, parallel-mpi).
*Tensor_*	Type(data type and order) of input Tensor. `Tensor_` must be `partensor::Tensor<order>`, where `order` must be in range of [3-8].

Parameters

tnsX	[in] The given Tensor to be factorized of `Tensor_` type, with `double` data.
R	[in] The rank of decomposition.
options	[in] User's `options`, other than the default. It must be of `partensor::Options<partensor::Tensor<order>>` type, where `order` must be in range of [3-8].

Returns

An object of type `Status` with the results of the algorithm.

8.5.2.9 cpd() [9/9]

```
execution::internal::enable_if_execution_policy< ExecutionPolicy, Status< Tensor_, execution↩
::execution_policy_t< ExecutionPolicy >, DefaultValues > > partensor::cpd (
            ExecutionPolicy && ,
            Tensor_ const & tnsX,
            std::size_t const R,
            Options< Tensor_, execution::execution_policy_t< ExecutionPolicy >, DefaultValues
> const & options,
            MatrixArray< Tensor_ > const & factorsInit )
```

Interface of Canonical Polyadic Decomposition(cpd), with the use of an `Execution Policy`, which can be either `sequential`, `parallel` with the use of `MPI`, or `parallel` with the use of `OpenMP`. In order to choose a policy, type `execution::seq`, `execution::mpi` or `execution::omp`. Default value is `sequential`, in case no `ExecutionPolicy` is passed.

Template Parameters

ExecutionPolicy	Type of `stl Execution Policy` (sequential, parallel-mpi).
*Tensor_*	Type(data type and order) of input Tensor. `Tensor_` must be `partensor::Tensor<order>`, where `order` must be in range of [3-8].

Parameters

tnsX	[in] The given Tensor to be factorized of `Tensor_` type, with `double` data.
R	[in] The rank of decomposition.
options	[in] User's `options`, other than the default. It must be of `partensor::Options<partensor::Tensor<order>>` type, where `order` must be in range of [3-8].
factorsInit	[in] Uses initialized factors instead of randomly generated. The data must be of `partensor::Matrix` type and stored in an `stl` array with size same as the `order` of `tnsX`.

Returns

An object of type `Status` with the results of the algorithm.

8.6 Cpd.hpp

Go to the documentation of this file.

```
#ifndef DOXYGEN_SHOULD_SKIP_THIS
#endif // DOXYGEN_SHOULD_SKIP_THIS
/*************************************************************************/
#ifndef PARTENSOR_CPD_HPP
#define PARTENSOR_CPD_HPP

#include "PARTENSOR_basic.hpp"
#include "Matricization.hpp"
#include "PartialCwiseProd.hpp"
#include "MTTKRP.hpp"
#include "NesterovMNLS.hpp"
#include "Normalize.hpp"
#include "Timers.hpp"
#include "ReadWrite.hpp"

namespace partensor
{
  inline namespace v1
  {
    namespace internal
    {
      //template <typename ExecutionPolicy, typename Tensor>
      //execution::internal::enable_if_execution_policy<ExecutionPolicy,Tensor>
      //Status cpd_f(ExecutionPolicy &&, Tensor const &tnsX, std::size_t rank);

      /*
       * Includes the implementation of CP Decomposition. Based on the given
       * parameters one of the overloaded operators will be called.
       */
      template <typename Tensor_>
      struct CPD_Base {
        static constexpr std::size_t TnsSize    = TensorTraits<Tensor_>::TnsSize;
        static constexpr std::size_t lastFactor = TnsSize - 1;
        using DataType   = typename TensorTraits<Tensor_>::DataType;
        using MatrixType = typename TensorTraits<Tensor_>::MatrixType;
        using Dimensions = typename TensorTraits<Tensor_>::Dimensions;
        using Constraints = std::array<Constraint, TnsSize>;
        using MatrixArray = std::array<MatrixType, TnsSize>;
        using DoubleArray = std::array<double, TnsSize>;
      };

      template <typename Tensor_, typename ExecutionPolicy = execution::sequenced_policy>
      struct CPD : public CPD_Base<Tensor_>
      {
        using          CPD_Base<Tensor_>::TnsSize;
        using          CPD_Base<Tensor_>::lastFactor;
        using typename CPD_Base<Tensor_>::Dimensions;
        using typename CPD_Base<Tensor_>::MatrixArray;
        using typename CPD_Base<Tensor_>::DataType;

        using Options = partensor::Options<Tensor_,execution::sequenced_policy,DefaultValues>;
        using Status  = partensor::Status<Tensor_,execution::sequenced_policy,DefaultValues>;

        // Variables that will be used in cpd implementations.
        struct Member_Variables {
```

```
        MatrixArray  krao;
        MatrixArray  factor_T_factor;
        MatrixArray  mttkrp;
        MatrixArray  tns_mat;
        MatrixArray  norm_factors;
        MatrixArray  old_factors;
        MatrixArray  true_factors;

        Matrix       cwise_factor_product;
        Matrix       tnsX_mat_lastFactor_T;
        Matrix       temp_matrix;

        Tensor_      tnsX;

        bool         all_orthogonal = true;
        int          weight_factor;

        Member_Variables() = default;
        Member_Variables(Member_Variables const &) = default;
        Member_Variables(Member_Variables      &&) = default;

         Member_Variables &operator=(Member_Variables const &) = default;
         Member_Variables &operator=(Member_Variables      &&) = default;
      };

      /*
       * In case option variable @c writeToFile is enabled, then, before the end
       * of the algorithm, it writes the resulted factors in files, whose
       * paths are specified before compiling in @ options.final_factors_path.
       *
       * @param  st  [in] Struct where the returned values of @c Cpd are stored.
       */
      void writeFactorsToFile(Status const &st)
      {
        std::size_t size;
        for(std::size_t i=0; i<TnsSize; ++i)
        {
          size = st.factors[i].rows() * st.factors[i].cols();
          partensor::write(st.factors[i],
                           st.options.final_factors_paths[i],
                           size);
        }
      }

      /*
       * Compute the cost function value based on the initial factors.
       *
       * @param  mv  [in]     Struct where ALS variables are stored.
       * @param  st  [in,out] Struct where the returned values of @c Cpd are stored.
       *                      In this case the cost function value is updated.
       */
      void cost_function_init(Member_Variables const &mv,
                              Status                 &st)
      {
        st.f_value = sqrt( ( mv.tns_mat[lastFactor] - st.factors[lastFactor] * PartialKhatriRao(st.factors, lastFactor).transpose() ).squaredNorm() );
      }

      /*
       * Compute the cost function value at the end of each outer iteration
       * based on the last factor.
       *
       * @param  mv  [in]     Struct where ALS variables are stored.
       * @param  st  [in,out] Struct where the returned values of @c Cpd are stored.
       *                      In this case the cost function value is updated.
       */
      void cost_function(Member_Variables const &mv,
                         Status                 &st)
      {
        st.f_value = sqrt( ( mv.tns_mat[lastFactor] - st.factors[lastFactor] * mv.krao[lastFactor].transpose() ).squaredNorm() );
      }

      /*
       * Compute the cost function value at the end of each outer iteration
       * based on the last accelerated factor.
       *
       * @param  mv                [in] Struct where ALS variables are stored.
       * @param  st                [in] Struct where the returned values of @c Cpd are stored.
       * @param  factors           [in] Accelerated factors.
       * @param  factors_T_factors [in] Gramian matrices of factors.
       *
       * @returns The cost function calculated with the accelerated factors.
       */
      double accel_cost_function(Member_Variables const &mv,
                                 Status           const &st,
                                 MatrixArray      const &factors,
```

```
                                   MatrixArray      const &factors_T_factors)
        {
          return sqrt( st.frob_tns + (PartialCwiseProd(factors_T_factors, lastFactor) * factors_T_factors[lastFactor]).trace()
                 - 2 * ((PartialKhatriRao(factors, lastFactor).transpose() * mv.tnsX_mat_lastFactor_T) * factors[lastFactor]).trace() );
        }

        void cost_function2(Member_Variables const &mv,
                            Status                 &st)
        {
          st.f_value = sqrt( st.frob_tns -2 * (mv.mttkrp.cwiseProduct(st.factors[lastFactor])).sum() +
                 (mv.cwise_factor_product.cwiseProduct(mv.factor_T_factor[lastFactor])).sum() );
        }

        /*
         * Based on each factor's constraint, a different
         * update function is used at every outer iteration.
         *
         * Computes also factor^T * factor at the end.
         *
         * @param  idx [in]     Factor to be updated.
         * @param  mv  [in]     Struct where ALS variables are stored.
         * @param  st  [in,out] Struct where the returned values of @c Cpd are stored.
         *                      Updates the @c stl array with the factors.
         */
        void update_factor(int              const  idx,
                           Member_Variables       &mv,
                           Status                 &st    )
        {
          // Update factor
          switch ( st.options.constraints[idx] )
          {
            case Constraint::unconstrained:
            {
              st.factors[idx].noalias() = mv.mttkrp[idx] * mv.cwise_factor_product.inverse();
              break;
            }
            case Constraint::nonnegativity:
            {
              mv.temp_matrix = st.factors[idx];
              NesterovMNLS(mv.cwise_factor_product, mv.mttkrp[idx], st.options.nesterov_delta_1,
                        st.options.nesterov_delta_2, st.factors[idx]);
              if(st.factors[idx].cwiseAbs().colwise().sum().minCoeff() == 0)
                st.factors[idx] = 0.9 * st.factors[idx] + 0.1 * mv.temp_matrix;
              break;
            }
            case Constraint::orthogonality:
            {
              mv.temp_matrix = mv.mttkrp[idx].transpose() * mv.mttkrp[idx];
              Eigen::SelfAdjointEigenSolver<Matrix> eigensolver(mv.temp_matrix);
              mv.temp_matrix.noalias() = (eigensolver.eigenvectors())
                                         * (eigensolver.eigenvalues().cwiseInverse().cwiseSqrt().asDiagonal())
                                         * (eigensolver.eigenvectors().transpose());
              st.factors[idx].noalias() = mv.mttkrp[idx] * mv.temp_matrix;
              break;
            }
            case Constraint::sparsity:
              break;
            default: // in case of Constraint::constant
              break;
          }

          // Compute A^T * A + B^T * B + ...
          mv.factor_T_factor[idx].noalias() = st.factors[idx].transpose() * st.factors[idx];
        }

        /*
         * @brief Line Search Acceleration
         *
         * Performs an acceleration step on the updated factors, and keeps the accelerated factors
         * when the step succeeds. Otherwise, the acceleration step is ignored.
         * Line Search Acceleration reduces the number of outer iterations in the ALS algorithm.
         *
         * @note This implementation ONLY, if factors are of @c Matrix type.
         *
         * @param  mv  [in,out] Struct where ALS variables are stored.
         *                      In case the acceleration step is successful the Gramian
         *                      matrices of factors are updated.
         * @param  st  [in,out] Struct where the returned values of @c Cpd are stored.
         *                      If the acceleration succeeds updates @c factors
         *                      and cost function value.
         *
         */
        void line_search_accel(Member_Variables &mv,
                               Status           &st)
```

```
    {
      double       f_accel    = 0.0; // Objective Value after the acceleration step
      double const accel_step = pow(st.ao_iter+1,(1.0/(st.options.accel_coeff)));

      MatrixArray  accel_factors;
      MatrixArray  accel_gramians;

      for(std::size_t i=0; i<TnsSize; ++i)
      {
        accel_factors[i]  = mv.old_factors[i] + accel_step * (st.factors[i] - mv.old_factors[i]);
        accel_gramians[i] = accel_factors[i].transpose() * accel_factors[i];
      }

      f_accel = accel_cost_function(mv, st, accel_factors, accel_gramians);
      if (st.f_value > f_accel)
      {
        st.factors          = accel_factors;
        mv.factor_T_factor = accel_gramians;
        st.f_value          = f_accel;
        Partensor()->Logger()->info("Acceleration Step SUCCEEDED! at iter: {}", st.ao_iter);
      }
      else
        st.options.accel_fail++;

      if (st.options.accel_fail==5)
      {
        st.options.accel_fail=0;
        st.options.accel_coeff++;
      }
    }

    /*
     * Sequential implementation of Alternating Least Squares (ALS) method.
     *
     * @param  R    [in]     The rank of decomposition.
     * @param  mv   [in]     Struct where ALS variables are stored and being updated
     *                       until a termination condition is true.
     * @param  st   [in,out] Struct where the returned values of @c Cpd are stored.
     */
    void als(std::size_t      const  R,
             Member_Variables       &mv,
             Status                 &status)
    {
      for (std::size_t i=0; i<TnsSize; i++)
      {
        mv.factor_T_factor[i].noalias() = status.factors[i].transpose() * status.factors[i];
        mv.tns_mat[i]                   = Matricization(mv.tnsX, i);
      }

      if(status.options.acceleration)
      {
        mv.tnsX_mat_lastFactor_T = mv.tns_mat[lastFactor].transpose();
      }

      if(status.options.normalization)
      {
        choose_normilization_factor(status, mv.all_orthogonal, mv.weight_factor);
      }

      // Normalize(static_cast<int>(R), mv.factor_T_factor, status.factors);
      status.frob_tns          = square_norm(mv.tnsX);
      cost_function_init(mv, status);
      status.rel_costFunction = status.f_value/sqrt(status.frob_tns);

      // ---- Loop until ALS converges ----
      while(1)
      {
        status.ao_iter++;
        Partensor()->Logger()->info("iter: {} -- fvalue: {} -- relative_costFunction: {}",
status.ao_iter,
                                                     status.f_value, status.rel_costFunction);

        for (std::size_t i=0; i<TnsSize; i++)
        {
          mttkrp(status.factors, mv.tns_mat[i], i, mv.krao[i], mv.mttkrp[i]);
          mv.cwise_factor_product = PartialCwiseProd(mv.factor_T_factor, i);

          // Update factor
          update_factor(i, mv, status);

          // Cost function Computation
          if(i == lastFactor)
            cost_function(mv, status);
        }

        status.rel_costFunction = status.f_value/sqrt(status.frob_tns);
        if(status.options.normalization && !mv.all_orthogonal)
```

```
            Normalize(mv.weight_factor, static_cast<int>(R), mv.factor_T_factor, status.factors);


          // ---- Terminating condition ----
          if (status.rel_costFunction < status.options.threshold_error || status.ao_iter >=
status.options.max_iter)
          {
            if(status.options.writeToFile)
              writeFactorsToFile(status);
            break;
          }

          if (status.options.acceleration)
          {
            mv.norm_factors = status.factors;
            // ---- Acceleration Step ----
            if (status.ao_iter > 1)
              line_search_accel(mv, status);

            mv.old_factors = mv.norm_factors;
          }
        } // end of while
      }

      /*
       * Sequential implementation of Alternating Least Squares (ALS) method.
       * Make use of true factors read from files instead of the Tensor.
       *
       * @param  R   [in]      The rank of decomposition.
       * @param  mv  [in]      Struct where ALS variables are stored and being updated
       *                       until a termination condition is true.
       * @param  st  [in,out] Struct where the returned values of @c Cpd are stored.
       */
      void als_true_factors( std::size_t      const  R,
                             Member_Variables       &mv,
                             Status                 &status )
      {
        for (std::size_t i=0; i<TnsSize; i++)
        {
          mv.factor_T_factor[i].noalias() = status.factors[i].transpose() * status.factors[i];
          mv.tns_mat[i]                   = generateTensor(i, mv.true_factors);
        }

        if(status.options.acceleration)
        {
          mv.tnsX_mat_lastFactor_T = mv.tns_mat[lastFactor].transpose();
        }

        if(status.options.normalization)
        {
          choose_normilization_factor(status, mv.all_orthogonal, mv.weight_factor);
        }

        // Normalize(static_cast<int>(R), mv.factor_T_factor, status.factors);
        status.frob_tns          = (mv.tns_mat[lastFactor]).squaredNorm();
        cost_function_init(mv, status);
        status.rel_costFunction = status.f_value/sqrt(status.frob_tns);

        // ---- Loop until ALS converges ----
        while(1)
        {
          status.ao_iter++;
          Partensor()->Logger()->info("iter: {} -- fvalue: {} -- relative_costFunction: {}",
status.ao_iter,
                                                      status.f_value, status.rel_costFunction);

          for (std::size_t i=0; i<TnsSize; i++)
          {
            mv.krao[i]               = PartialKhatriRao(status.factors,  i);
            mv.cwise_factor_product = PartialCwiseProd(mv.factor_T_factor, i);
            mv.mttkrp[i].noalias()   = mv.tns_mat[i] * mv.krao[i];

            // Update factor
            update_factor(i, mv, status);

            // Cost function Computation
            if(i == lastFactor)
              cost_function(mv, status);
          }

          status.rel_costFunction = status.f_value/sqrt(status.frob_tns);
          if(status.options.normalization && !mv.all_orthogonal)
            Normalize(mv.weight_factor, static_cast<int>(R), mv.factor_T_factor, status.factors);

          // ---- Terminating condition ----
          if (status.rel_costFunction < status.options.threshold_error || status.ao_iter >=
status.options.max_iter)
```

```
        {
          if(status.options.writeToFile)
            writeFactorsToFile(status);
          break;
        }

        if (status.options.acceleration)
        {
          mv.norm_factors = status.factors;
          // ---- Acceleration Step ----
          if (status.ao_iter > 1)
            line_search_accel(mv, status);

          mv.old_factors = mv.norm_factors;
        }
      } // end of while
    }

    Status operator()(Tensor_     const &tnsX,
                      std::size_t const  R)
    {
      Status           status = MakeStatus<Tensor_>();
      Member_Variables mv;

      // extract dimensions from tensor
      Dimensions const &tnsDims = tnsX.dimensions();
      // produce estimate factors using uniform distribution with entries in [0,1].
      makeFactors(tnsDims, status.options.constraints, R, status.factors);
      mv.tnsX = tnsX;
      als(R, mv, status);

      return status;
    }

    Status operator()(Tensor_     const &tnsX,
                      std::size_t const  R,
                      Options     const &options)
    {
      Status           status(options);
      Member_Variables mv;

      // extract dimensions from tensor
      Dimensions const &tnsDims = tnsX.dimensions();
      // produce estimate factors using uniform distribution with entries in [0,1].
      makeFactors(tnsDims, status.options.constraints, R, status.factors);
      mv.tnsX = tnsX;

      switch ( status.options.method )
      {
        case Method::als:
        {
          als(R, mv, status);
          break;
        }
        case Method::rnd:
          break;
        case Method::bc:
          break;
        default:
          break;
      }

      return status;
    };

    Status operator()(Tensor_     const &tnsX,
                      std::size_t const  R,
                      MatrixArray const &factorsInit)
    {
      Status status = MakeStatus<Tensor_>();
      Member_Variables mv;

      status.factors = factorsInit;
      mv.tnsX        = tnsX;
      als(R, mv, status);

      return status;
    }

    Status operator()(Tensor_     const &tnsX,
                      std::size_t const R,
                      Options     const &options,
                      MatrixArray const &factorsInit)
    {
      Status status(options);
      Member_Variables mv;

```

```
      status.factors = factorsInit;
      mv.tnsX        = tnsX;

      switch ( status.options.method )
      {
        case Method::als:
        {
          als(R, mv, status);
          break;
        }
        case Method::rnd:
          break;
        case Method::bc:
          break;
        default:
          break;
      }

      return status;
    }

    Status operator()(std::array<int, TnsSize> const &tnsDims,
                      std::size_t              const  R,
                      std::string              const &path)
    {
      using  Tensor = Tensor<static_cast<int>(TnsSize)>;

      Status           status = MakeStatus<Tensor>();
      Member_Variables mv;

      long long int fileSize = 1;
      for(auto &dim : tnsDims)
        fileSize *= static_cast<long long int>(dim);

      mv.tnsX.resize(tnsDims);
      // Read the whole Tensor from a file
      read( path,
            fileSize,
            0,
            mv.tnsX   );
      // produce estimate factors using uniform distribution with entries in [0,1].
      makeFactors(tnsDims, status.options.constraints, R, status.factors);
      als(R, mv, status);

      return status;
    }

    Status operator()(std::array<int, TnsSize> const &tnsDims,
                      std::size_t              const  R,
                      std::string              const &path,
                      Options                  const &options)
    {
      Status           status(options);
      Member_Variables mv;

      long long int fileSize = 1;
      for(auto &dim : tnsDims)
        fileSize *= static_cast<long long int>(dim);

      mv.tnsX.resize(tnsDims);
      // Read the whole Tensor from a file
      read( path,
            fileSize,
            0,
            mv.tnsX   );
      // produce estimate factors using uniform distribution with entries in [0,1].
      makeFactors(tnsDims, status.options.constraints, R, status.factors);

      switch ( status.options.method )
      {
        case Method::als:
        {
          als(R, mv, status);
          break;
        }
        case Method::rnd:
          break;
        case Method::bc:
          break;
        default:
          break;
      }

      return status;
    }

    Status operator()(std::array<int, TnsSize>          const &tnsDims,
```

```
                  std::size_t                       const  R,
                  std::array<std::string,TnsSize+1> const &paths)
{
  using  Tensor = Tensor<static_cast<int>(TnsSize)>;
  Status status = MakeStatus<Tensor>();
  Member_Variables mv;

  long long int fileSize = 1;
  for(auto &dim : tnsDims)
    fileSize *= static_cast<long long int>(dim);

  mv.tnsX.resize(tnsDims);
  // Read the whole Tensor from a file
  read( paths.front(),
        fileSize,
        0,
        mv.tnsX   );

  // Read initialized factors from files
  for(std::size_t i=0; i<TnsSize; ++i)
  {
    status.factors[i] = Matrix(tnsDims[i],static_cast<int>(R));
    read( paths[i+1],
          tnsDims[i]*R,
          0,
          status.factors[i] );
  }

  als(R, mv, status);

  return status;
}

Status operator()(std::array<int, TnsSize>           const &tnsDims,
                  std::size_t                        const  R,
                  std::array<std::string, TnsSize+1> const &paths,
                  Options                            const &options)
{
  Status status(options);
  Member_Variables mv;

  long long int fileSize = 1;
  for(auto &dim : tnsDims)
    fileSize *= static_cast<long long int>(dim);

  mv.tnsX.resize(tnsDims);
  // Read the whole Tensor from a file
  read( paths.front(),
        fileSize,
        0,
        mv.tnsX   );

  // Read initialized factors from files
  for(std::size_t i=0; i<TnsSize; ++i)
  {
    status.factors[i] = Matrix(tnsDims[i],static_cast<int>(R));
    read( paths[i+1],
          tnsDims[i]*R,
          0,
          status.factors[i] );
  }

  switch ( status.options.method )
  {
    case Method::als:
    {
      als(R, mv, status);
      break;
    }
    case Method::rnd:
      break;
    case Method::bc:
      break;
    default:
      break;
  }

  return status;
}

Status operator()(std::array<int, TnsSize>         const &tnsDims,
                  std::size_t                      const  R,
                  std::array<std::string, TnsSize> const &true_paths,
                  std::array<std::string, TnsSize> const &init_paths,
                  Options                          const &options)
{
  Status status(options);
```

```
          Member_Variables mv;

          for(std::size_t i=0; i<TnsSize; ++i)
          {
            mv.true_factors[i] = Matrix(tnsDims[i],static_cast<int>(R));
            status.factors[i]  = Matrix(tnsDims[i],static_cast<int>(R));

            // Read initialized factors from files
            read( true_paths[i],
                  tnsDims[i]*R,
                  0,
                  mv.true_factors[i] );
            // Read initialized factors from files
            read( init_paths[i],
                  tnsDims[i]*R,
                  0,
                  status.factors[i] );
          }

          switch ( status.options.method )
          {
            case Method::als:
            {
              als_true_factors(R, mv, status);
              break;
            }
            case Method::rnd:
              break;
            case Method::bc:
              break;
            default:
              break;
          }

          return status;
        }

      };
    }  // namespace internal
  }    // namespace v1
} // end namespace partensor

#if USE_MPI

#include "CpdMpi.hpp"
#endif /* USE_MPI */

#if USE_OPENMP

#include "CpdOpenMP.hpp"
#endif /* USE_OPENMP */

#if USE_CUDA

#include "CUDA/CpdCUDA.hpp"
#endif /* USE_CUDA */

namespace partensor
{
  template <typename Tensor_, typename ExecutionPolicy>

   execution::internal::enable_if_execution_policy<ExecutionPolicy,Status<Tensor_,execution::execution_policy_t<Execution
  cpd( ExecutionPolicy       &&,
       Tensor_         const &tnsX,
       std::size_t     const  R )
  {
    using  ExPolicy = execution::execution_policy_t<ExecutionPolicy>;

    if constexpr (std::is_same_v<ExPolicy,execution::sequenced_policy>)
    {
      return internal::CPD<Tensor_>()(tnsX,R);
    }
    else if constexpr (std::is_same_v<ExPolicy,execution::openmpi_policy>)
    {
      return internal::CPD<Tensor_,execution::openmpi_policy>()(tnsX,R);
    }
    else if constexpr (std::is_same_v<ExPolicy,execution::openmp_policy>)
    {
      return internal::CPD<Tensor_,execution::openmp_policy>()(tnsX,R);
    }
    else if constexpr (std::is_same_v<ExPolicy, execution::cuda_policy>)
    {
      return internal::CPD<Tensor_, execution::cuda_policy>()(tnsX,R);
    }
    else
      return internal::CPD<Tensor_>()(tnsX,R);
  }
```

```
/*
 * Interface of Canonical Polyadic Decomposition(cpd). Sequential Policy.
 *
 * @tparam Tensor_      Type(data type and order) of input Tensor.
 *                      @c Tensor_ must be @c partensor::Tensor<order>, where
 *                      @c order must be in range of @c [3-8].
 * @param  tnsX    [in] The given Tensor to be factorized of @c Tensor_ type,
 *                      with @c double data.
 * @param  R       [in] The rank of decomposition.
 *
 * @returns An object of type @c Status, containing the results of the algorithm.
 */
template<typename Tensor_>
auto cpd(Tensor_     const &tnsX,
         std::size_t const  R)
{
  return internal::CPD<Tensor_,execution::sequenced_policy>()(tnsX,R);
}

template <typename Tensor_, typename ExecutionPolicy>

 execution::internal::enable_if_execution_policy<ExecutionPolicy,Status<Tensor_,execution::execution_policy_t<Execution
cpd( ExecutionPolicy       &&,
     Tensor_         const &tnsX,
     std::size_t     const  R,
     Options<Tensor_,execution::execution_policy_t<ExecutionPolicy>,DefaultValues> const &options )
{
  using  ExPolicy = execution::execution_policy_t<ExecutionPolicy>;

  if constexpr (std::is_same_v<ExPolicy,execution::sequenced_policy>)
  {
    return internal::CPD<Tensor_>()(tnsX,R,options);
  }
  else if constexpr (std::is_same_v<ExPolicy,execution::openmpi_policy>)
  {
    return internal::CPD<Tensor_,execution::openmpi_policy>()(tnsX,R,options);
  }
  else if constexpr (std::is_same_v<ExPolicy,execution::openmp_policy>)
  {
    return internal::CPD<Tensor_,execution::openmp_policy>()(tnsX,R,options);
  }
  else if constexpr (std::is_same_v<ExPolicy, execution::cuda_policy>)
  {
    return internal::CPD<Tensor_,execution::cuda_policy>()(tnsX,R,options);
  }
  else
    return internal::CPD<Tensor_>()(tnsX,R,options);
}

/*
 * Interface of Canonical Polyadic Decomposition(cpd). Sequential Policy.
 *
 * @tparam Tensor_      Type(data type and order) of input Tensor.
 *                      @c Tensor_ must be @c partensor::Tensor<order>, where
 *                      @c order must be in range of @c [3-8].
 * @param  tnsX    [in] The given Tensor to be factorized of @c Tensor_ type,
 *                      with @c double data.
  * @param  R       [in] The rank of decomposition.
  * @param  options [in] User’s @c options, other than the default. It must be of
  *                      @c partensor::Options<partensor::Tensor<order» type,
  *                      where @c order must be in range of @c [3-8].
  *
  * @returns An object of type @c Status with the results of the algorithm.
  */
 template<typename Tensor_>
 auto cpd(Tensor_          const &tnsX,
          std::size_t      const  R,
          Options<Tensor_> const &options)
 {
   return internal::CPD<Tensor_,execution::sequenced_policy>()(tnsX,R,options);
 }

 template <typename ExecutionPolicy, typename Tensor_>

 execution::internal::enable_if_execution_policy<ExecutionPolicy,Status<Tensor_,execution::execution_policy_t<Execution
 cpd( ExecutionPolicy            &&,
      Tensor_              const &tnsX,
      std::size_t          const  R,
      MatrixArray<Tensor_> const &factorsInit )
 {
   using  ExPolicy = execution::execution_policy_t<ExecutionPolicy>;

   if constexpr (std::is_same_v<ExPolicy,execution::sequenced_policy>)
   {
     return internal::CPD<Tensor_>()(tnsX,R,factorsInit);
   }
```

```
  else if constexpr (std::is_same_v<ExPolicy,execution::openmpi_policy>)
  {
    return internal::CPD<Tensor_,execution::openmpi_policy>()(tnsX,R,factorsInit);
  }
  else if constexpr (std::is_same_v<ExPolicy,execution::openmp_policy>)
  {
    return internal::CPD<Tensor_,execution::openmp_policy>()(tnsX,R,factorsInit);
  }
  else if constexpr (std::is_same_v<ExPolicy,execution::cuda_policy>)
  {
    return internal::CPD<Tensor_,execution::cuda_policy>()(tnsX,R,factorsInit);
  }
  else
    return internal::CPD<Tensor_>()(tnsX,R,factorsInit);
}

/*
 * Interface of Canonical Polyadic Decomposition(cpd). Sequential Policy.
 *
 * @tparam Tensor_          Type(data type and order) of input Tensor.
 *                          @c Tensor_ must be @c partensor::Tensor<order>, where
 *                          @c order must be in range of @c [3-8].
 * @param  tnsX        [in] The given Tensor to be factorized of @c Tensor_ type,
 *                          with @c double data.
 * @param  R           [in] The rank of decomposition.
 * @param  factorsInit [in] Uses initialized factors instead of randomly generated. The
 *                          data must be of @c partensor::Matrix type and stored in an
 *                          @c stl array with size same as the @c order of @c tnsX.
 *
 * @returns An object of type @c Status with the results of the algorithm.
 */
template<typename Tensor_>
auto cpd(Tensor_                const &tnsX,
         std::size_t            const  R,
         MatrixArray<Tensor_> const &factorsInit)
{
  return internal::CPD<Tensor_,execution::sequenced_policy>()(tnsX,R,factorsInit);
}

template <typename ExecutionPolicy, typename Tensor_>

execution::internal::enable_if_execution_policy<ExecutionPolicy,Status<Tensor_,execution::execution_policy_t<Execution
cpd( ExecutionPolicy            &&,
     Tensor_              const &tnsX,
     std::size_t          const  R,
     Options<Tensor_,execution::execution_policy_t<ExecutionPolicy>,DefaultValues> const &options,
     MatrixArray<Tensor_> const &factorsInit )
{
  using  ExPolicy = execution::execution_policy_t<ExecutionPolicy>;

  if constexpr (std::is_same_v<ExPolicy,execution::sequenced_policy>)
  {
    return internal::CPD<Tensor_>()(tnsX,R,options,factorsInit);
  }
  else if constexpr (std::is_same_v<ExPolicy,execution::openmpi_policy>)
  {
    return internal::CPD<Tensor_,execution::openmpi_policy>()(tnsX,R,options,factorsInit);
  }
  else if constexpr (std::is_same_v<ExPolicy,execution::openmp_policy>)
  {
    return internal::CPD<Tensor_,execution::openmp_policy>()(tnsX,R,options,factorsInit);
  }
  else if constexpr (std::is_same_v<ExPolicy,execution::cuda_policy>)
  {
    return internal::CPD<Tensor_,execution::cuda_policy>()(tnsX,R,options,factorsInit);
  }
  else
    return internal::CPD<Tensor_>()(tnsX,R,options,factorsInit);
}

/*
 * Interface of Canonical Polyadic Decomposition(cpd). Sequential Policy.
 *
 * @tparam Tensor_          Type(data type and order) of input Tensor.
 *                          @c Tensor_ must be @c partensor::Tensor<order>, where
 *                          @c order must be in range of @c [3-8].
 * @param  tnsX        [in] The given Tensor to be factorized of @c Tensor_ type,
 *                          with @c double data.
 * @param  R           [in] The rank of decomposition.
 * @param  options     [in] User's @c options, other than the default. It must be of
 *                          @c partensor::Options<partensor::Tensor<order>> type,
 *                          where @c order must be in range of @c [3-8].
 * @param  factorsInit [in] Uses initialized factors instead of randomly generated. The
 *                          data must be of @c partensor::Matrix type and stored in an
 *                          @c stl array with size same as the @c order of @c tnsX.
 *
 * @returns An object of type @c Status with the results of the algorithm.
```

```
 */
template<typename Tensor_>
auto cpd(Tensor_              const &tnsX,
         std::size_t          const  R,
         Options<Tensor_>     const &options,
         MatrixArray<Tensor_> const &factorsInit)
{
  return internal::CPD<Tensor_,execution::sequenced_policy>()(tnsX,R,options,factorsInit);
}

template <typename ExecutionPolicy, std::size_t TnsSize>

execution::internal::enable_if_execution_policy<ExecutionPolicy,Status<Tensor<static_cast<int>(TnsSize)>,execution::exe
cpd( ExecutionPolicy                &&,
     std::array<int, TnsSize> const &tnsDims,
     std::size_t              const  R,
     std::string              const &path )
{
  using  ExPolicy = execution::execution_policy_t<ExecutionPolicy>;
  using  Tensor_  = Tensor<static_cast<int>(TnsSize)>;

  if constexpr (std::is_same_v<ExPolicy,execution::sequenced_policy>)
  {
    return internal::CPD<Tensor_>()(tnsDims,R,path);
  }
  else if constexpr (std::is_same_v<ExPolicy,execution::openmpi_policy>)
  {
    return internal::CPD<Tensor_,execution::openmpi_policy>()(tnsDims,R,path);
  }
  else if constexpr (std::is_same_v<ExPolicy,execution::openmp_policy>)
  {
    return internal::CPD<Tensor_,execution::openmp_policy>()(tnsDims,R,path);
  }
  else if constexpr (std::is_same_v<ExPolicy, execution::cuda_policy>)
  {
    return internal::CPD<Tensor_, execution::cuda_policy>()(tnsDims,R,path);
  }
  else
    return internal::CPD<Tensor_>()(tnsDims,R,path);
}

/*
 * Interface of Canonical Polyadic Decomposition(cpd). Sequential Policy.
 * With this version of @c cpd, the Tensor can be read from a file, specified in
 * @c path variable.
 *
 * @tparam TnsSize         Order of input Tensor.
 *
 * @param  tnsDims    [in] @c Stl array containing the Tensor dimensions, whose
 *                         length must be same as the Tensor order.
 * @param  R          [in] The rank of decomposition.
 * @param  path       [in] The path where the tensor is located.
 *
 * @returns An object of type @c Status with the results of the algorithm.
 */
template<std::size_t TnsSize>
auto cpd(std::array<int, TnsSize> const &tnsDims,
         std::size_t              const  R,
         std::string              const &path )
{
  using Tensor_ = Tensor<static_cast<int>(TnsSize)>;
  return internal::CPD<Tensor_,execution::sequenced_policy>()(tnsDims,R,path);
}

template <typename ExecutionPolicy, std::size_t TnsSize>

execution::internal::enable_if_execution_policy<ExecutionPolicy,Status<Tensor<static_cast<int>(TnsSize)>,execution::exe
cpd( ExecutionPolicy                &&,
     std::array<int,TnsSize>  const &tnsDims,
     std::size_t              const  R,
     std::string              const &path,

Options<Tensor<static_cast<int>(TnsSize)>,execution::execution_policy_t<ExecutionPolicy>,DefaultValues>
const &options )
{
  using  ExPolicy = execution::execution_policy_t<ExecutionPolicy>;
  using  Tensor_  = Tensor<static_cast<int>(TnsSize)>;

  if constexpr (std::is_same_v<ExPolicy,execution::sequenced_policy>)
  {
    return internal::CPD<Tensor_>()(tnsDims,R,path,options);
  }
  else if constexpr (std::is_same_v<ExPolicy,execution::openmpi_policy>)
  {
    return internal::CPD<Tensor_,execution::openmpi_policy>()(tnsDims,R,path,options);
  }
  else if constexpr (std::is_same_v<ExPolicy,execution::openmp_policy>)
```

```
    {
      return internal::CPD<Tensor_,execution::openmp_policy>()(tnsDims,R,path,options);
    }
    else if constexpr (std::is_same_v<ExPolicy,execution::cuda_policy>)
    {
      return internal::CPD<Tensor_,execution::cuda_policy>()(tnsDims,R,path,options);
    }
    else
      return internal::CPD<Tensor_>()(tnsDims,R,path,options);
  }

  /*
   * Interface of Canonical Polyadic Decomposition(cpd). Sequential Policy.
   * With this version of @c cpd, the Tensor can be read from a file, specified in
   * @c path variable.
   *
   * @tparam TnsSize         Order of input Tensor.
   *
   * @param  tnsDims     [in] @c Stl array containing the Tensor dimensions, whose
   *                          length must be same as the Tensor order.
   * @param  R           [in] The rank of decomposition.
   * @param  path        [in] The path where the tensor is located.
   * @param  options     [in] User's @c options, other than the default. It must be of
   *                          @c partensor::Options<partensor::Tensor<order» type,
   *                          where @c order must be in range of @c [3-8].
   *
   * @returns An object of type @c Status with the results of the algorithm.
   */
  template<std::size_t TnsSize>
  auto cpd(std::array<int, TnsSize>                    const &tnsDims,
           std::size_t                                 const  R,
           std::string                                 const &path,
           Options<Tensor<static_cast<int>(TnsSize)»  const &options )
  {
    using Tensor_ = Tensor<static_cast<int>(TnsSize)>;
    return internal::CPD<Tensor_,execution::sequenced_policy>()(tnsDims,R,path,options);
  }


  template <typename ExecutionPolicy, std::size_t TnsSize>

  execution::internal::enable_if_execution_policy<ExecutionPolicy,Status<Tensor<static_cast<int>(TnsSize)>,execution::exe
  cpd( ExecutionPolicy                           &&,
       std::array<int, TnsSize>           const &tnsDims,
       std::size_t                        const  R,
       std::array<std::string, TnsSize+1> const &paths   )
  {
    using  ExPolicy = execution::execution_policy_t<ExecutionPolicy>;
    using  Tensor_  = Tensor<static_cast<int>(TnsSize)>;

    if constexpr (std::is_same_v<ExPolicy,execution::sequenced_policy>)
    {
      return internal::CPD<Tensor_>()(tnsDims,R,paths);
    }
    else if constexpr (std::is_same_v<ExPolicy,execution::openmpi_policy>)
    {
      return internal::CPD<Tensor_,execution::openmpi_policy>()(tnsDims,R,paths);
    }
    else if constexpr (std::is_same_v<ExPolicy,execution::openmp_policy>)
    {
      return internal::CPD<Tensor_,execution::openmp_policy>()(tnsDims,R,paths);
    }
    else if constexpr (std::is_same_v<ExPolicy,execution::cuda_policy>)
    {
      return internal::CPD<Tensor_,execution::cuda_policy>()(tnsDims,R,paths);
    }
    else
      return internal::CPD<Tensor_>()(tnsDims,R,paths);
  }

  /*
   * Interface of Canonical Polyadic Decomposition(cpd). Sequential Policy.
   * With this version of @c cpd, the Tensor can be read from a file, specified in
   * @c path variable.
   *
   * @tparam TnsSize         Order of input Tensor.
   *
   * @param  tnsDims     [in] @c Stl array containing the Tensor dimensions, whose
   *                          length must be same as the Tensor order.
   * @param  R           [in] The rank of decomposition.
   * @param  paths       [in] An @c stl array containing paths for the Tensor to be
   *                          factorized and after that the paths for the initialized
   *                          factors.
   *
   * @returns An object of type @c Status with the results of the algorithm.
   */
  template<std::size_t TnsSize>
```

```
auto cpd(std::array<int, TnsSize>            const &tnsDims,
         std::size_t                         const  R,
         std::array<std::string, TnsSize+1> const &paths  )
{
  using Tensor_ = Tensor<static_cast<int>(TnsSize)>;
  return internal::CPD<Tensor_,execution::sequenced_policy>()(tnsDims,R,paths);
}

template <std::size_t TnsSize, typename ExecutionPolicy>

execution::internal::enable_if_execution_policy<ExecutionPolicy,Status<Tensor<static_cast<int>(TnsSize)>,execution::exe
cpd( ExecutionPolicy                               &&,
     std::array<int, TnsSize>            const &tnsDims,
     std::size_t                         const  R,
     std::array<std::string, TnsSize+1> const &paths,

Options<Tensor<static_cast<int>(TnsSize)>,execution::execution_policy_t<ExecutionPolicy>,DefaultValues>
const &options )
{
  using  ExPolicy = execution::execution_policy_t<ExecutionPolicy>;
  using  Tensor_  = Tensor<static_cast<int>(TnsSize)>;

  if constexpr (std::is_same_v<ExPolicy,execution::sequenced_policy>)
  {
    return internal::CPD<Tensor_>()(tnsDims,R,paths,options);
  }
  else if constexpr (std::is_same_v<ExPolicy,execution::openmpi_policy>)
  {
    return internal::CPD<Tensor_,execution::openmpi_policy>()(tnsDims,R,paths,options);
  }
  else if constexpr (std::is_same_v<ExPolicy,execution::openmp_policy>)
  {
    return internal::CPD<Tensor_,execution::openmp_policy>()(tnsDims,R,paths,options);
  }
  else if constexpr (std::is_same_v<ExPolicy,execution::cuda_policy>)
  {
    return internal::CPD<Tensor_,execution::cuda_policy>()(tnsDims,R,paths,options);
  }
  else
    return internal::CPD<Tensor_>()(tnsDims,R,paths,options);
}

/*
 * Interface of Canonical Polyadic Decomposition(cpd). Sequential Policy.
 * With this version of @c cpd, the Tensor can be read from a file, specified in
 * @c path variable.
 *
 * @tparam TnsSize          Order of input Tensor.
 *
 * @param  tnsDims     [in] @c Stl array containing the Tensor dimensions, whose
 *                          length must be same as the Tensor order.
 * @param  R           [in] The rank of decomposition.
 * @param  paths       [in] An @c stl array containing paths for the Tensor to be
 *                          factorized and after that the paths for the initialized
 *                          factors.
 * @param  options     [in] User's @c options, other than the default. It must be of
 *                          @c partensor::Options<partensor::Tensor<order» type,
 *                          where @c order must be in range of @c [3-8].
 *
 * @returns An object of type @c Status with the results of the algorithm.
 */
template<std::size_t TnsSize>
auto cpd(std::array<int, TnsSize>                    const &tnsDims,
         std::size_t                                 const  R,
         std::array<std::string,TnsSize+1>           const &paths,
         Options<Tensor<static_cast<int>(TnsSize)» const &options )
{
  using Tensor_ = Tensor<static_cast<int>(TnsSize)>;
  return internal::CPD<Tensor_,execution::sequenced_policy>()(tnsDims,R,paths,options);
}

template <std::size_t TnsSize, typename ExecutionPolicy>

execution::internal::enable_if_execution_policy<ExecutionPolicy,Status<Tensor<static_cast<int>(TnsSize)>,execution::exe
cpd( ExecutionPolicy                             &&,
     std::array<int, TnsSize>          const &tnsDims,
     std::size_t                       const  R,
     std::array<std::string, TnsSize> const &true_paths,
     std::array<std::string, TnsSize> const &init_paths,

Options<Tensor<static_cast<int>(TnsSize)>,execution::execution_policy_t<ExecutionPolicy>,DefaultValues>
const &options )
{
  using  ExPolicy = execution::execution_policy_t<ExecutionPolicy>;
  using  Tensor_  = Tensor<static_cast<int>(TnsSize)>;

  if constexpr (std::is_same_v<ExPolicy,execution::sequenced_policy>)
```

```
    {
      return internal::CPD<Tensor_>()(tnsDims,R,true_paths,init_paths,options);
    }
    else if constexpr (std::is_same_v<ExPolicy,execution::openmpi_policy>)
    {
      return internal::CPD<Tensor_,execution::openmpi_policy>()(tnsDims,R,true_paths,init_paths,options);
    }
    else if constexpr (std::is_same_v<ExPolicy,execution::openmp_policy>)
    {
      return internal::CPD<Tensor_,execution::openmp_policy>()(tnsDims,R,true_paths,init_paths,options);
    }
    else if constexpr (std::is_same_v<ExPolicy,execution::cuda_policy>)
    {
      return internal::CPD<Tensor_,execution::cuda_policy>()(tnsDims,R,true_paths,init_paths,options);
    }
    else
      return internal::CPD<Tensor_>()(tnsDims,R,true_paths,init_paths,options);
  }

  /*
   * Interface of Canonical Polyadic Decomposition(cpd). Sequential Policy.
   * With this version of @c cpd, the Tensor can be read from a file, specified in
   * @c path variable.
   *
   * @tparam TnsSize         Order of input Tensor.
   *
   * @param  tnsDims    [in] @c Stl array containing the Tensor dimensions, whose
   *                         length must be same as the Tensor order.
   * @param  R          [in] The rank of decomposition.
   * @param  true_paths [in] An @c stl array containing paths for the true factors.
   * @param  init_paths [in] An @c stl array containing paths for initialized
   *                         factors.
   * @param  options    [in] User's @c options, other than the default. It must be of
   *                         @c partensor::Options<partensor::Tensor<order>> type,
   *                         where @c order must be in range of @c [3-8].
   *
   * @returns An object of type @c Status with the results of the algorithm.
   */
  template<std::size_t TnsSize>
  auto cpd(std::array<int, TnsSize>                    const &tnsDims,
           std::size_t                                 const  R,
           std::array<std::string, TnsSize>            const &true_paths,
           std::array<std::string, TnsSize>            const &init_paths,
           Options<Tensor<static_cast<int>(TnsSize)>> const &options )
  {
    using Tensor_ = Tensor<static_cast<int>(TnsSize)>;
    return internal::CPD<Tensor_,execution::sequenced_policy>()(tnsDims,R,true_paths,init_paths,options);
  }

}

#endif // PARTENSOR_CPD_HPPP
```

8.7 CpdDimTree.hpp File Reference

```
#include "PARTENSOR_basic.hpp"
#include <unsupported/Eigen/MatrixFunctions>
#include "execution.hpp"
#include "DataGeneration.hpp"
#include "DimTrees.hpp"
#include "Normalize.hpp"
#include "NesterovMNLS.hpp"
#include "Constants.hpp"
#include "Timers.hpp"
#include "ReadWrite.hpp"
#include "Matricization.hpp"
#include "PartialCwiseProd.hpp"
#include "PartialKhatriRao.hpp"
```

Functions

- template<typename ExecutionPolicy , std::size_t TnsSize>
  execution::internal::enable_if_execution_policy< ExecutionPolicy, Status< Tensor< static_cast< int >(TnsSize)>, execution::execution_policy_t< ExecutionPolicy >, DefaultValues > > cpdDimTree (ExecutionPolicy &&, std::array< int, TnsSize > const &tnsDims, std::size_t const R, std::array< std::string, TnsSize+1 > const &paths)
- template<typename ExecutionPolicy , std::size_t TnsSize>
  execution::internal::enable_if_execution_policy< ExecutionPolicy, Status< Tensor< static_cast< int >(TnsSize)>, execution::execution_policy_t< ExecutionPolicy >, DefaultValues > > cpdDimTree (ExecutionPolicy &&, std::array< int, TnsSize > const &tnsDims, std::size_t const R, std::array< std::string, TnsSize+1 > const &paths, Options< Tensor< static_cast< int >(TnsSize)>, execution::execution_policy_t< ExecutionPolicy >, DefaultValues > const &options)
- template<typename ExecutionPolicy , std::size_t TnsSize>
  execution::internal::enable_if_execution_policy< ExecutionPolicy, Status< Tensor< static_cast< int >(TnsSize)>, execution::execution_policy_t< ExecutionPolicy >, DefaultValues > > cpdDimTree (ExecutionPolicy &&, std::array< int, TnsSize > const &tnsDims, std::size_t const R, std::string const &path)
- template<typename ExecutionPolicy , std::size_t TnsSize>
  execution::internal::enable_if_execution_policy< ExecutionPolicy, Status< Tensor< static_cast< int >(TnsSize)>, execution::execution_policy_t< ExecutionPolicy >, DefaultValues > > cpdDimTree (ExecutionPolicy &&, std::array< int, TnsSize > const &tnsDims, std::size_t const R, std::string const &path, Options< Tensor< static_cast< int >(TnsSize)>, execution::execution_policy_t< ExecutionPolicy >, DefaultValues > const &options)
- template<typename ExecutionPolicy , typename Tensor_ >
  execution::internal::enable_if_execution_policy< ExecutionPolicy, Status< Tensor_, execution::execution_policy_t< ExecutionPolicy >, DefaultValues > > cpdDimTree (ExecutionPolicy &&, Tensor_ const &tnsX, std::size_t const R)
- template<typename Tensor_ , typename MatrixArray_ , typename ExecutionPolicy >
  execution::internal::enable_if_execution_policy< ExecutionPolicy, Status< Tensor_, execution::execution_policy_t< ExecutionPolicy >, DefaultValues > > cpdDimTree (ExecutionPolicy &&, Tensor_ const &tnsX, std::size_t const R, MatrixArray_ const &factorsInit)
- template<typename Tensor_ , typename ExecutionPolicy >
  execution::internal::enable_if_execution_policy< ExecutionPolicy, Status< Tensor_, execution::execution_policy_t< ExecutionPolicy >, DefaultValues > > cpdDimTree (ExecutionPolicy &&, Tensor_ const &tnsX, std::size_t const R, Options< Tensor_, execution::execution_policy_t< ExecutionPolicy >, DefaultValues > const &options)
- template<typename Tensor_ , typename MatrixArray_ , typename ExecutionPolicy >
  execution::internal::enable_if_execution_policy< ExecutionPolicy, Status< Tensor_, execution::execution_policy_t< ExecutionPolicy >, DefaultValues > > cpdDimTree (ExecutionPolicy &&, Tensor_ const &tnsX, std::size_t const R, Options< Tensor_, execution::execution_policy_t< ExecutionPolicy >, DefaultValues > const &options, MatrixArray_ const &factorsInit)

8.7.1 Detailed Description

Implements the Canonical Polyadic Decomposition using Dimension Trees. Make use of `spdlog` library in order to write output in a log file in `"../log"`. In case of using parallelism with mpi, then the functions from `CpdDimTreeMpi.hpp` will be called.

8.7.2 Function Documentation

8.7.2.1 cpdDimTree() [1/8]

```
execution::internal::enable_if_execution_policy< ExecutionPolicy, Status< Tensor< static_↩
cast< int >(TnsSize)>, execution::execution_policy_t< ExecutionPolicy >, DefaultValues > >
partensor::cpdDimTree (
            ExecutionPolicy && ,
            std::array< int, TnsSize > const & tnsDims,
            std::size_t const R,
            std::array< std::string, TnsSize+1 > const & paths )
```

Interface of Canonical Polyadic Decomposition(cpd) with the use of Dimension Trees. Also, an `Execution Policy` can be used, which can be either `sequential` or `parallel` with the use of `MPI`. In order to choose a policy, type `execution::seq` or `execution::mpi`. Default value is `sequential`, in case no `ExecutionPolicy` is passed.

With this version of `cpdDimTree`, the Tensor can be read from a file, specified in `path` variable.

Template Parameters

ExecutionPolicy	Type of `stl Execution Policy` (sequential, parallel-mpi).
TnsSize	Order of the input Tensor.

Parameters

tnsDims	[in] `Stl` array containing the Tensor dimensions, whose length must be same as the Tensor order.
R	[in] The rank of decomposition.
paths	[in] An `stl` array containing paths for the Tensor to be factorized and after that the paths for the initialized factors.

Returns

An object of type `Status` with the results of the algorithm.

8.7.2.2 cpdDimTree() [2/8]

```
execution::internal::enable_if_execution_policy< ExecutionPolicy, Status< Tensor< static_↩
cast< int >(TnsSize)>, execution::execution_policy_t< ExecutionPolicy >, DefaultValues > >
partensor::cpdDimTree (
            ExecutionPolicy && ,
            std::array< int, TnsSize > const & tnsDims,
            std::size_t const R,
            std::array< std::string, TnsSize+1 > const & paths,
            Options< Tensor< static_cast< int >(TnsSize)>, execution::execution_policy_t<
ExecutionPolicy >, DefaultValues > const & options )
```

Interface of Canonical Polyadic Decomposition(cpd) with the use of Dimension Trees. Also, an `Execution Policy` can be used, which can be either `sequential` or `parallel` with the use of `MPI`. In order to choose a policy, type `execution::seq` or `execution::mpi`. Default value is `sequential`, in case no `ExecutionPolicy` is passed.

With this version of `cpdDimTree`, the Tensor can be read from a file, specified in `path` variable.

Template Parameters

ExecutionPolicy	Type of `stl Execution Policy` (sequential, parallel-mpi).
TnsSize	Order of the input Tensor.

Parameters

tnsDims	[in] `Stl` array containing the Tensor dimensions, whose length must be same as the Tensor order.
R	[in] The rank of decomposition.
paths	[in] An `stl` array containing paths for the Tensor to be factorized and after that the paths for the initialized factors.
options	[in] User's `options`, other than the default. It must be of `partensor::Options<partensor::Tensor<order>>` type, where `order` must be in range of [3-8].

Returns

An object of type `Status` with the results of the algorithm.

8.7.2.3 cpdDimTree() [3/8]

```
execution::internal::enable_if_execution_policy< ExecutionPolicy, Status< Tensor< static_↩
cast< int >(TnsSize)>, execution::execution_policy_t< ExecutionPolicy >, DefaultValues > >
partensor::cpdDimTree (
            ExecutionPolicy && ,
            std::array< int, TnsSize > const & tnsDims,
            std::size_t const R,
            std::string const & path )
```

Interface of Canonical Polyadic Decomposition(cpd) with the use of Dimension Trees. Also, an `Execution Policy` can be used, which can be either `sequential` or `parallel` with the use of `MPI`. In order to choose a policy, type `execution::seq` or `execution::mpi`. Default value is `sequential`, in case no `ExecutionPolicy` is passed.

With this version of `cpdDimTree`, the Tensor can be read from a file, specified in `path` variable.

Template Parameters

ExecutionPolicy	Type of `stl Execution Policy` (sequential, parallel-mpi).
TnsSize	Order of the input Tensor.

Parameters

tnsDims	[in] `Stl` array containing the Tensor dimensions, whose length must be same as the Tensor order.
R	[in] The rank of decomposition.
path	[in] The path where the tensor is located.

Returns

An object of type `Status` with the results of the algorithm.

8.7.2.4 cpdDimTree() [4/8]

```
execution::internal::enable_if_execution_policy< ExecutionPolicy, Status< Tensor< static_↩
cast< int >(TnsSize)>, execution::execution_policy_t< ExecutionPolicy >, DefaultValues > >
partensor::cpdDimTree (
            ExecutionPolicy && ,
            std::array< int, TnsSize > const & tnsDims,
            std::size_t const R,
            std::string const & path,
            Options< Tensor< static_cast< int >(TnsSize)>, execution::execution_policy_t<
ExecutionPolicy >, DefaultValues > const & options )
```

Interface of Canonical Polyadic Decomposition(cpd) with the use of Dimension Trees. Also, an `Execution Policy` can be used, which can be either `sequential` or `parallel` with the use of `MPI`. In order to choose a policy, type `execution::seq` or `execution::mpi`. Default value is `sequential`, in case no `ExecutionPolicy` is passed.

With this version of `cpdDimTree`, the Tensor can be read from a file, specified in `path` variable.

Template Parameters

ExecutionPolicy	Type of `stl Execution Policy` (sequential, parallel-mpi).
TnsSize	Order of the input Tensor.

Parameters

tnsDims	[in] `Stl` array containing the Tensor dimensions, whose length must be same as the Tensor order.
R	[in] The rank of decomposition.
path	[in] The path where the tensor is located.
options	[in] User's `options`, other than the default. It must be of `partensor::Options<partensor::Tensor<order>>` type, where `order` must be in range of [3-8].

Returns

An object of type `Status` with the results of the algorithm.

8.7.2.5 cpdDimTree() [5/8]

```
execution::internal::enable_if_execution_policy< ExecutionPolicy, Status< Tensor_, execution↩
::execution_policy_t< ExecutionPolicy >, DefaultValues > > partensor::cpdDimTree (
```

```
ExecutionPolicy && ,
Tensor_ const & tnsX,
std::size_t const R )
```

Interface of Canonical Polyadic Decomposition(cpd) with the use of Dimension Trees. Also, an `Execution Policy` can be used, which can be either `sequential` or `parallel` with the use of `MPI`. In order to choose a policy, type `execution::seq`
or `execution::mpi`. Default value is `sequential`, in case no `ExecutionPolicy` is passed.

Template Parameters

ExecutionPolicy	Type of `stl Execution Policy` (sequential, parallel-mpi).
*Tensor_*	Type(data type and order) of input Tensor. `Tensor_` must be `partensor::Tensor<order>`, where `order` must be in range of [3-8].

Parameters

tnsX	[in] The given Tensor to be factorized of `Tensor_` type, with `double` data.
R	[in] The rank of decomposition.

Returns

An object of type `Status` with the results of the algorithm.

8.7.2.6 cpdDimTree() [6/8]

```
execution::internal::enable_if_execution_policy< ExecutionPolicy, Status< Tensor_, execution↩
::execution_policy_t< ExecutionPolicy >, DefaultValues > > partensor::cpdDimTree (
            ExecutionPolicy && ,
            Tensor_ const & tnsX,
            std::size_t const R,
            MatrixArray_ const & factorsInit )
```

Interface of Canonical Polyadic Decomposition(cpd) with the use of Dimension Trees. Also, an `Execution Policy` can be used, which can be either `sequential` or `parallel` with the use of `MPI`. In order to choose a policy, type `execution::seq` or `execution::mpi`. Default value is `sequential`, in case no `ExecutionPolicy` is passed.

Template Parameters

ExecutionPolicy	Type of `stl Execution Policy` (sequential, parallel-mpi).
*Tensor_*	Type(data type and order) of input Tensor.
*MatrixArray_*	An `stl` array, where the initialized factors will be stored. Its size must be equal to the Tensor's `tnsX order`. The type can be either `partensor::Matrix`, or `partensor::Tensor<2>`.

Parameters

tnsX	[in] The given Tensor to be factorized of `Tensor_` type, with `double` data.

Parameters

R	[in] The rank of decomposition.
factorsInit	[in] Uses initialized factors instead of randomly generated. The data can be either `partensor::Matrix`, or `partensor::Tensor<2>` type and stored in an `stl` array with size same as the `order` of `tnsX`.

Returns

An object of type `Status` with the results of the algorithm.

8.7.2.7 cpdDimTree() [7/8]

```
execution::internal::enable_if_execution_policy< ExecutionPolicy, Status< Tensor_, execution↩
::execution_policy_t< ExecutionPolicy >, DefaultValues > > partensor::cpdDimTree (
            ExecutionPolicy && ,
            Tensor_ const & tnsX,
            std::size_t const R,
            Options< Tensor_, execution::execution_policy_t< ExecutionPolicy >, DefaultValues
> const & options )
```

Interface of Canonical Polyadic Decomposition(cpd) with the use of Dimension Trees. Also, an `Execution Policy` can be used, which can be either `sequential` or `parallel` with the use of `MPI`. In order to choose a policy, type `execution::seq` or `execution::mpi`. Default value is `sequential`, in case no `ExecutionPolicy` is passed.

Template Parameters

ExecutionPolicy	Type of `stl Execution Policy` (sequential, parallel-mpi).
*Tensor_*	Type(data type and order) of input Tensor.

Parameters

tnsX	[in] The given Tensor to be factorized of `Tensor_` type, with `double` data.
R	[in] The rank of decomposition.
options	[in] User's `options`, other than the default. It must be of `partensor::Options<partensor::Tensor<order>>` type, where `order` must be in range of [3-8].

Returns

An object of type `Status` with the results of the algorithm.

8.7.2.8 cpdDimTree() [8/8]

```
execution::internal::enable_if_execution_policy< ExecutionPolicy, Status< Tensor_, execution↩
::execution_policy_t< ExecutionPolicy >, DefaultValues > > partensor::cpdDimTree (
```

```
            ExecutionPolicy && ,
            Tensor_ const & tnsX,
            std::size_t const R,
            Options< Tensor_, execution::execution_policy_t< ExecutionPolicy >, DefaultValues
> const & options,
            MatrixArray_ const & factorsInit )
```

Interface of Canonical Polyadic Decomposition(cpd) with the use of Dimension Trees. Interface of Canonical Polyadic Decomposition(cpd) with the use of Dimension Trees. Also, an Execution Policy can be used, which can be either sequential or parallel with the use of MPI. In order to choose a policy, type execution::seq or execution::mpi. Default value is sequential, in case no ExecutionPolicy is passed.

Template Parameters

ExecutionPolicy	Type of stl Execution Policy (sequential, parallel-mpi).
*Tensor_*	Type(data type and order) of input Tensor.
*MatrixArray_*	An stl array, where the initialized factors will be stored. Its size must be equal to the Tensor's tnsX order. The type can be either partensor::Matrix, or partensor::Tensor<2>.

Parameters

tnsX	[in] The given Tensor to be factorized of Tensor_ type, with double data.
R	[in] The rank of decomposition.
options	[in] User's options, other than the default. It must be of partensor::Options<partensor::Tensor<order>> type, where order must be in range of [3-8].
factorsInit	[in] Uses initialized factors instead of randomly generated. The data must be of partensor::Matrix type and stored in an stl array with size same as the order of tnsX.

Returns

An object of type Status with the results of the algorithm.

8.8 CpdDimTree.hpp

Go to the documentation of this file.

```
#ifndef DOXYGEN_SHOULD_SKIP_THIS
#endif // DOXYGEN_SHOULD_SKIP_THIS
/*************************************************************************/
#ifndef PARTENSOR_CPD_DIMTREE_DIM_TREE_HPP
#define PARTENSOR_CPD_DIMTREE_DIM_TREE_HPP

#include "PARTENSOR_basic.hpp"
#include <unsupported/Eigen/MatrixFunctions>
#include "execution.hpp"
#include "DataGeneration.hpp"
#include "DimTrees.hpp"
#include "Normalize.hpp"
#include "NesterovMNLS.hpp"
#include "Constants.hpp"
#include "Timers.hpp"
#include "ReadWrite.hpp"
#include "Matricization.hpp"
#include "PartialCwiseProd.hpp"
#include "PartialKhatriRao.hpp"

namespace partensor
{
  inline namespace v1
```

```
{
  namespace internal
  {

    //template <typename ExecutionPolicy, typename Tensor>
    //execution::internal::enable_if_execution_policy<ExecutionPolicy,Tensor>
    //Status cpd_f(ExecutionPolicy &&, Tensor const &tnsX, std::size_t R);

    /*
     * Includes the implementation of CP Decomposition with Dimension Trees.
     * Based on the given parameters one of the overloaded operators will
     * be called.
     * @tparam Tensor_   Type(data type and order) of input Tensor.
     */
    template <typename Tensor_>
    struct CPD_DIMTREE_Base {
      static constexpr std::size_t TnsSize    = TensorTraits<Tensor_>::TnsSize;
      static constexpr std::size_t lastFactor = TnsSize - 1;
      using DataType         = typename TensorTraits<Tensor_>::DataType;
      using MatrixType       = typename TensorTraits<Tensor_>::MatrixType;
      using Dimensions       = typename TensorTraits<Tensor_>::Dimensions;
      using IntArray         = typename TensorTraits<Tensor_>::IntArray;
      using TensorMatrixType = Tensor<2>;
      using Constraints      = std::array<Constraint, TnsSize>;
      using MatrixArray      = std::array<MatrixType, TnsSize>;
      using DoubleArray      = std::array<double, TnsSize>;
      using FactorArray      = std::array<FactorDimTree,TnsSize>;
      using IndexPair        = typename std::array<Eigen::IndexPair<int>, 1>;
    };

    template <typename Tensor_, typename ExecutionPolicy = execution::sequenced_policy>
    struct CPD_DIMTREE : public CPD_DIMTREE_Base<Tensor_>
    {
      using          CPD_DIMTREE_Base<Tensor_>::TnsSize;
      using          CPD_DIMTREE_Base<Tensor_>::lastFactor;
      using typename CPD_DIMTREE_Base<Tensor_>::TensorMatrixType;
      using typename CPD_DIMTREE_Base<Tensor_>::Dimensions;
      using typename CPD_DIMTREE_Base<Tensor_>::MatrixArray;
      using typename CPD_DIMTREE_Base<Tensor_>::DataType;
      using typename CPD_DIMTREE_Base<Tensor_>::IntArray;
      using typename CPD_DIMTREE_Base<Tensor_>::FactorArray;
      using typename CPD_DIMTREE_Base<Tensor_>::IndexPair;

      using Options = partensor::Options<Tensor_,execution::sequenced_policy,DefaultValues>;
      using Status  = partensor::Status<Tensor_,execution::sequenced_policy,DefaultValues>;

      // Variables that will be used in cpd with
      // the Dimension Trees implementations.
      struct Member_Variables {
        Matrix      last_gramian;
        Matrix      cwise_factor_product;
        Matrix      mttkrp;
        Matrix      currentFactor;
        Matrix      temp_matrix;
        Matrix      tnsX_mat_lastFactor_T;

        FactorArray factors;
        FactorArray norm_factors;
        FactorArray old_factors;

        typename FactorArray::iterator it_factor;
        typename FactorArray::iterator it_old_factor;

        Tensor_           tnsX;
        Tensor_           tnsX_approx;
        IntArray          labelSet;    // starting label set for root
        const IndexPair   product_dims = { Eigen::IndexPair<int>(0, 0) }; // used for tensor contractions

        bool              all_orthogonal = true;
        int               weight_factor;

        Member_Variables() = default;
        Member_Variables(Member_Variables const &) = default;
        Member_Variables(Member_Variables      &&) = default;

        Member_Variables &operator=(Member_Variables const &) = default;
        Member_Variables &operator=(Member_Variables      &&) = default;
      };

      /*
       * In case option variable @c writeToFile is enabled, then, before the end
       * of the algorithm, it writes the resulted factors in files, whose
       * paths are specified before compiling in @ options.final_factors_path.
       *
       * @param  st  [in] Struct where the returned values of @c CpdDimTree are stored.
       */
```

```
      void writeFactorsToFile(Status const &st)
      {
        std::size_t size;
        for(std::size_t i=0; i<TnsSize; ++i)
        {
          size = st.factors[i].rows() * st.factors[i].cols();
          partensor::write(st.factors[i],
                           st.options.final_factors_paths[i],
                           size);
        }
      }

      /*
       * Compute the cost function value at the end of each outer iteration
       * based on the last factor.
       *
       * @param  mv  [in]     Struct where ALS variables are stored.
       * @param  st  [in,out] Struct where the returned values of @c CpdDimTree are stored.
       *                      In this case the cost function value is updated.
       */
      void cost_function ( Member_Variables const &mv,
                           Status                 &st)
      {
        st.f_value = sqrt( st.frob_tns -2 * ((mv.mttkrp * mv.currentFactor.transpose()).trace()) +
(mv.cwise_factor_product * mv.last_gramian).trace() );
      }

      /*
       * Compute the cost function value at the end of each outer iteration
       * based on the last accelerated factor.
       *
       * @param  mv                 [in] Struct where ALS variables are stored.
       * @param  st                 [in] Struct where the returned values of @c CpdDimTree
       *                                 are stored.
       * @param  factors            [in] Accelerated factors.
       * @param  factors_T_factors  [in] Gramian matrices of factors.
       *
       * @returns The cost function calculated with the accelerated factors.
       */
      double accel_cost_function(Member_Variables const &mv,
                                 Status           const &st,
                                 MatrixArray      const &factors,
                                 MatrixArray      const &factors_T_factors)
      {
        return sqrt (st.frob_tns + (PartialCwiseProd(factors_T_factors, lastFactor) *
factors_T_factors[lastFactor]).trace()
               - 2 * ((PartialKhatriRao(factors, lastFactor).transpose() * mv.tnsX_mat_lastFactor_T) *
factors[lastFactor]).trace() );
      }

      /*
       * Based on each factor's constraint, a different
       * update function is used at every outer iteration.
       *
       * Computes also factor^T * factor at the end.
       *
       * @tparam Dimensions       Array type containing the Tensor dimensions.
       *
       * @param  idx     [in]     Factor to be updated.
       * @param  R       [in]     The rank of decomposition.
       * @param  tnsDims [in]     Tensor Dimensions. Each index contains the corresponding factor's
rows length.
       * @param  st      [in]     Struct where the returned values of @c CpdDimTree are stored.
       * @param  mv      [in,out] Struct where ALS variables are stored.
       *                          Updates the current factor (@c Matrix type) and then updates
       *                          the same factor of @c FactorDimTree type.
       */
      template<typename Dimensions>
      void update_factor(int                 const  idx,
                         std::size_t         const  R,
                         Dimensions          const &tnsDims,
                         Status              const &st,
                         Member_Variables          &mv )
      {
        switch ( st.options.constraints[idx] )
        {
          case Constraint::unconstrained:
          {
            mv.currentFactor = mv.mttkrp * mv.cwise_factor_product.inverse();
            break;
          }
          case Constraint::nonnegativity:
          {
            mv.temp_matrix = mv.currentFactor;
            NesterovMNLS(mv.cwise_factor_product, mv.mttkrp, st.options.nesterov_delta_1,
                                   st.options.nesterov_delta_2, mv.currentFactor);
            if(mv.currentFactor.cwiseAbs().colwise().sum().minCoeff() == 0)
```

```
                mv.currentFactor = 0.9 * mv.currentFactor + 0.1 * mv.temp_matrix;
              break;
            }
            case Constraint::orthogonality:
            {
              mv.temp_matrix = mv.mttkrp.transpose() * mv.mttkrp;
              Eigen::SelfAdjointEigenSolver<Matrix> eigensolver(mv.temp_matrix);
              mv.temp_matrix.noalias() = (eigensolver.eigenvectors())
                                        *
(eigensolver.eigenvalues().cwiseInverse().cwiseSqrt().asDiagonal())
                                        * (eigensolver.eigenvectors().transpose());
              mv.currentFactor.noalias() = mv.mttkrp * mv.temp_matrix;
              break;
            }
            case Constraint::sparsity:
              break;
            default: // in case of Constraint::constant
              break;
          }
          mv.it_factor->factor     = matrixToTensor(mv.currentFactor, tnsDims[idx],
static_cast<int>(R)); // Map factor from Eigen Matrix to Eigen Tensor
          mv.it_factor->gramian = (mv.it_factor->factor).contract(mv.it_factor->factor,
mv.product_dims); // Compute Covariance Tensor
        }

        /*
         * @brief Line Search Acceleration
         *
         * Performs an acceleration step in the updated factors, and keeps the accelerated factors when
         * the step succeeds. Otherwise, the acceleration step is ignored.
         * Line Search Acceleration reduces the number outer iterations in the ALS algorithm.
         *
         * @note This implementation ONLY, if factors are of @c Matrix type.
         *
         * @tparam Dimensions   Array type containing the Tensor dimensions.
         *
         * @param  mv  [in,out] Struct where ALS variables are stored.
         *                      In case the acceration is successful factor and
         *                      factor^T * factor of @c FactorDimTree type are updated.
         * @param  st  [in,out] Struct where the returned values of @c CpdDimTree are stored.
         *                      If the acceleration succeeds updates the cost function value.
         *
         */
        template<typename Dimensions>
        void line_search_accel(Dimensions       const &tnsDims,
                               std::size_t      const  R,
                               Member_Variables        &mv,
                               Status                  &st)
        {
          double       f_accel    = 0.0; // Objective Value after the acceleration step
          double const accel_step = pow(st.ao_iter+1,(1.0/(st.options.accel_coeff)));

          Matrix      factor;
          Matrix      old_factor;
          MatrixArray accel_factors;
          MatrixArray accel_gramians;

          for(std::size_t i=0; i<TnsSize; ++i)
          {
            factor             = tensorToMatrix(mv.factors[i].factor, tnsDims[i],static_cast<int>(R));
            old_factor         = tensorToMatrix(mv.old_factors[i].factor,
tnsDims[i],static_cast<int>(R));
            accel_factors[i]  = old_factor + accel_step * (factor - old_factor);
            accel_gramians[i] = accel_factors[i].transpose() * accel_factors[i];

            mv.it_factor++;
            mv.it_old_factor++;
          }

          f_accel = accel_cost_function(mv, st, accel_factors, accel_gramians);
          if (st.f_value > f_accel)
          {
            for(std::size_t i=0; i<TnsSize; ++i)
            {
              mv.factors[i].factor     = matrixToTensor(accel_factors[i], tnsDims[i],
static_cast<int>(R));
              mv.factors[i].gramian = matrixToTensor(accel_gramians[i], static_cast<int>(R),
static_cast<int>(R));
            }
            st.f_value = f_accel;
            Partensor()->Logger()->info("Acceleration Step SUCCEEDED! at iter: {}", st.ao_iter);
          }
          else
            st.options.accel_fail++;

          if (st.options.accel_fail==5)
          {
```

```
        st.options.accel_fail=0;
        st.options.accel_coeff++;
      }
    }

    /*
     * Sequential implementation of Alternating Least Squares (ALS) method
     * with Dimension Trees.
     *
     * @tparam Dimensions       Array type containing the Tensor dimensions.
     *
     * @param  tnsDims [in]     Tensor Dimensions. Each index contains the corresponding factor's rows length.
     * @param  R       [in]     The rank of decomposition.
     * @param  mv      [in]     Struct where ALS variables are stored  and being updated
     *                          until a termination condition is true.
     * @param  st      [in,out] Struct where the returned values of @c CpdDimTree are stored.
     */
    template<typename Dimensions>
    void als(Dimensions       const &tnsDims,
             std::size_t      const  R,
             Member_Variables       &mv,
             Status                 &status)
    {
      if (status.options.acceleration)
      {
        mv.tnsX_mat_lastFactor_T = (Matricization(mv.tnsX, lastFactor)).transpose();
      }

      if(status.options.normalization)
      {
        choose_normilization_factor(status, mv.all_orthogonal, mv.weight_factor);
      }

      mv.tnsX_approx.resize(tnsDims);
      CpdGen(mv.factors, R, mv.tnsX_approx);

      status.frob_tns          = square_norm(mv.tnsX);
      status.f_value           = norm(mv.tnsX - mv.tnsX_approx); // Error_tnsX
      status.rel_costFunction = status.f_value/sqrt(status.frob_tns);
      // increments from 1 to labelSet.size()
      std::iota(mv.labelSet.begin(), mv.labelSet.end(), 1);

      ExprTree<TnsSize> tree;
      tree.Create(mv.labelSet, tnsDims, R, mv.tnsX);

      mv.it_factor = mv.factors.begin();
      for(std::size_t k = 0; k<TnsSize; k++)
      {
          mv.it_factor->leaf = static_cast<TnsNode<1>*>(search_leaf(k+1, tree));
          mv.it_factor++;
      }

      // ---- Loop until ALS converges ----
      while(1)
      {
        status.ao_iter++;
        mv.it_factor = mv.factors.begin();
        Partensor()->Logger()->info("iter: {} -- fvalue: {} -- relative_costFunction: {}", status.ao_iter,
                                                          status.f_value, status.rel_costFunction);

        for(std::size_t i=0; i<TnsSize; i++)
        {
          mv.it_factor->leaf->UpdateTree(TnsSize, i, mv.it_factor);

          // Maps from Eigen Tensor to Eigen Matrix
          mv.temp_matrix          = tensorToMatrix(*reinterpret_cast<TensorMatrixType *>(mv.it_factor->leaf->TensorX()), static_cast<int>(R), tnsDims[i]);
          mv.mttkrp               = mv.temp_matrix.transpose();
          mv.cwise_factor_product = tensorToMatrix(mv.it_factor->leaf->Gramian(), static_cast<int>(R), static_cast<int>(R));
          mv.currentFactor        = tensorToMatrix(mv.it_factor->factor, tnsDims[i], static_cast<int>(R));

          update_factor(i, R, tnsDims, status, mv);
          mv.it_factor++;
        }

        mv.it_factor    = mv.factors.end()-1;
        mv.last_gramian = tensorToMatrix(mv.it_factor->gramian, static_cast<int>(R), static_cast<int>(R));
        cost_function(mv, status);
        status.rel_costFunction = status.f_value / sqrt(status.frob_tns);
        if(status.options.normalization && !mv.all_orthogonal)
          Normalize(mv.weight_factor, static_cast<int>(R), tnsDims, mv.factors);

```

```
          // ---- Terminating condition ----
          if (status.ao_iter >= status.options.max_iter || status.rel_costFunction < status.options.threshold_error)
          {
            for(std::size_t i=0; i<TnsSize; ++i)
              status.factors[i] = tensorToMatrix(mv.factors[i].factor, tnsDims[i], static_cast<int>(R));

            if(status.options.writeToFile)
              writeFactorsToFile(status);
            break;
          }

          if (status.options.acceleration)
          {
            mv.norm_factors = mv.factors;
            if (status.ao_iter > 1)
              line_search_accel(tnsDims, R, mv, status);

            mv.old_factors = mv.norm_factors;
          }
        } // end of while
      }

      Status operator()(Tensor_     const &tnsX,
                        std::size_t const  R)
      {
        Status            status = MakeStatus<Tensor_>();
        Member_Variables mv;

        // extract dimensions from tensor
        Dimensions const &tnsDims = tnsX.dimensions();
        // produce estimate factors using uniform distribution with entries in [0,1].
        makeFactors(tnsDims, status.options.constraints, R, mv.factors);
        // Normalize(static_cast<int>(R), tnsDims, factors);

        mv.tnsX = tnsX;
        als(tnsDims, R, mv, status);

        return status;
      }

      Status operator()(Tensor_     const &tnsX,
                        std::size_t const  R,
                        Options     const &options)
      {
        Status            status(options);
        Member_Variables mv;

        // extract dimensions from tensor
        Dimensions const &tnsDims = tnsX.dimensions();
        // produce estimate factors using uniform distribution with entries in [0,1].
        makeFactors(tnsDims, status.options.constraints, R, mv.factors);
        // Normalize(static_cast<int>(R), tnsDims, factors);
        mv.tnsX = tnsX;

        switch ( status.options.method )
        {
          case Method::als:
          {
            als(tnsDims, R, mv, status);
            break;
          }
          case Method::rnd:
            break;
          case Method::bc:
            break;
          default:
            break;
        }
        return status;
      }

      template<typename MatrixArray_>
      Status operator()(Tensor_      const &tnsX,
                        std::size_t  const  R,
                        MatrixArray_ const &factorsInit)
      {
        Status            status = MakeStatus<Tensor_>();
        Member_Variables mv;

        // extract dimensions from tensor
        Dimensions const &tnsDims = tnsX.dimensions();
        // Copy factorsInit data to factors - FactorDimTree data struct
        fillDimTreeFactors(factorsInit, status.options.constraints, mv.factors);
        // Normalize(static_cast<int>(R), tnsDims, factors);

```

```
      mv.tnsX = tnsX;
      als(tnsDims, R, mv, status);
      return status;
    }

    template<typename MatrixArray_>
    Status operator()(Tensor_      const &tnsX,
                      std::size_t  const  R,
                      Options      const &options,
                      MatrixArray_ const &factorsInit)
    {
      Status status(options);
      Member_Variables mv;

      // extract dimensions from tensor
      Dimensions const &tnsDims = tnsX.dimensions();
      // Copy factorsInit data to factors - FactorDimTree data struct
      fillDimTreeFactors(factorsInit, status.options.constraints, mv.factors);
      // Normalize(static_cast<int>(R), tnsDims, factors);
      mv.tnsX = tnsX;

      switch ( status.options.method )
      {
        case Method::als:
        {
          als(tnsDims, R, mv, status);
          break;
        }
        case Method::rnd:
          break;
        case Method::bc:
          break;
        default:
          break;
      }
      return status;
    }

    Status operator()(std::array<int, TnsSize> const &tnsDims,
                      std::size_t              const  R,
                      std::string              const &path)
    {
      using  Tensor = Tensor<static_cast<int>(TnsSize)>;

      Status           status = MakeStatus<Tensor>();
      Member_Variables mv;

      long long int fileSize = 1;
      for(auto &dim : tnsDims)
        fileSize *= static_cast<long long int>(dim);

      mv.tnsX.resize(tnsDims);
      // Read the whole Tensor from a file
      read( path,
            fileSize,
            0,
            mv.tnsX   );
      // produce estimate factors using uniform distribution with entries in [0,1].
      makeFactors(tnsDims, status.options.constraints, R, mv.factors);
      // Normalize(static_cast<int>(R), tnsDims, factors);

      als(tnsDims, R, mv, status);
      return status;
    }

    Status operator()(std::array<int, TnsSize> const &tnsDims,
                      std::size_t              const  R,
                      std::string              const &path,
                      Options                  const &options)
    {
      Status           status(options);
      Member_Variables mv;

      long long int fileSize = 1;
      for(auto &dim : tnsDims)
        fileSize *= static_cast<long long int>(dim);

      mv.tnsX.resize(tnsDims);
      // Read the whole Tensor from a file
      read( path,
            fileSize,
            0,
            mv.tnsX   );
      // produce estimate factors using uniform distribution with entries in [0,1].
      makeFactors(tnsDims, status.options.constraints, R, mv.factors);
      // Normalize(static_cast<int>(R), tnsDims, mv.factors);

```

```
      switch ( status.options.method )
      {
        case Method::als:
        {
          als(tnsDims, R, mv, status);
          break;
        }
        case Method::rnd:
          break;
        case Method::bc:
          break;
        default:
          break;
      }
      return status;
    }

    Status operator()(std::array<int, TnsSize>          const &tnsDims,
                      std::size_t                       const  R,
                      std::array<std::string,TnsSize+1> const &paths)
    {
      using  Tensor = Tensor<static_cast<int>(TnsSize)>;

      Status           status = MakeStatus<Tensor>();
      Member_Variables mv;

      long long int fileSize = 1;
      for(auto &dim : tnsDims)
        fileSize *= static_cast<long long int>(dim);

      mv.tnsX.resize(tnsDims);
      // Read the whole Tensor from a file
      read( paths.front(),
            fileSize,
            0,
            mv.tnsX   );
      // Read initialized factors from files
      for(std::size_t i=0; i<TnsSize; ++i)
      {
        status.factors[i] = Matrix(tnsDims[i],static_cast<int>(R));
        read( paths[i+1],
              tnsDims[i]*R,
              0,
              status.factors[i] );
      }
      // Copy factorsInit data to factors - FactorDimTree data struct
      fillDimTreeFactors(status.factors, status.options.constraints, mv.factors);
      // Normalize(static_cast<int>(R), tnsDims, mv.factors);

      als(tnsDims, R, mv, status);
      return status;
    }

    Status operator()(std::array<int, TnsSize>           const &tnsDims,
                      std::size_t                        const  R,
                      std::array<std::string, TnsSize+1> const &paths,
                      Options                            const &options)
    {
      Status status(options);
      Member_Variables mv;

      long long int fileSize = 1;
      for(auto &dim : tnsDims)
        fileSize *= static_cast<long long int>(dim);

      mv.tnsX.resize(tnsDims);
      // Read the whole Tensor from a file
      read( paths.front(),
            fileSize,
            0,
            mv.tnsX   );
      // Read initialized factors from files
      for(std::size_t i=0; i<TnsSize; ++i)
      {
        status.factors[i] = Matrix(tnsDims[i],static_cast<int>(R));
        read( paths[i+1],
              tnsDims[i]*R,
              0,
              status.factors[i] );
      }
      // Copy factorsInit data to factors - FactorDimTree data struct
      fillDimTreeFactors(status.factors, status.options.constraints, mv.factors);
      // Normalize(static_cast<int>(R), tnsDims, factors);
      switch ( status.options.method )
      {
        case Method::als:
        {
```

```
              als(tnsDims, R, mv, status);
              break;
            }
            case Method::rnd:
              break;
            case Method::bc:
              break;
            default:
              break;
          }
          return status;
        }

      };

    }  // namespace internal
  }  // namespace v1
} // end namespace partensor

#if USE_MPI

#include "CpdDimTreeMpi.hpp"
#endif /* USE_MPI */

namespace partensor
{

  template <typename ExecutionPolicy, typename Tensor_>
   execution::internal::enable_if_execution_policy<ExecutionPolicy,Status<Tensor_,execution::execution_policy_t<Execution
  cpdDimTree( ExecutionPolicy       &&,
              Tensor_         const &tnsX,
              std::size_t     const  R )
  {
    using  ExPolicy = execution::execution_policy_t<ExecutionPolicy>;

    if constexpr (std::is_same_v<ExPolicy,execution::sequenced_policy>)
    {
      return internal::CPD_DIMTREE<Tensor_>()(tnsX,R);
    }
    else if constexpr (std::is_same_v<ExPolicy,execution::openmpi_policy>)
    {
      return internal::CPD_DIMTREE<Tensor_,execution::openmpi_policy>()(tnsX,R);
    }
    else
      return internal::CPD_DIMTREE<Tensor_>()(tnsX,R);
  }

  /*
   * Interface of Canonical Polyadic Decomposition(cpd) with the use of Dimension
   * Trees. Sequential Policy.
   *
   * @tparam Tensor_      Type(data type and order) of input Tensor.
   *                      @c Tensor_ must be @c partensor::Tensor<order>, where
   *                      @c order must be in range of @c [3-8].
   * @param  tnsX    [in] The given Tensor to be factorized of @c Tensor_ type,
   *                      with @c double data.
   * @param  R       [in] The rank of decomposition.
   *
   * @returns An object of type @c Status with the results of the algorithm.
   */
  template<typename Tensor_>
  auto cpdDimTree(Tensor_     const &tnsX,
                  std::size_t const  R)
  {
    return internal::CPD_DIMTREE<Tensor_>()(tnsX,R);
  }

  template <typename Tensor_, typename ExecutionPolicy>
   execution::internal::enable_if_execution_policy<ExecutionPolicy,Status<Tensor_,execution::execution_policy_t<Execution
  cpdDimTree( ExecutionPolicy       &&,
              Tensor_         const &tnsX,
              std::size_t     const  R,
              Options<Tensor_,execution::execution_policy_t<ExecutionPolicy>,DefaultValues> const
   &options )
  {
    using  ExPolicy = execution::execution_policy_t<ExecutionPolicy>;

    if constexpr (std::is_same_v<ExPolicy,execution::sequenced_policy>)
    {
      return internal::CPD_DIMTREE<Tensor_>()(tnsX,R,options);
    }
    else if constexpr (std::is_same_v<ExPolicy,execution::openmpi_policy>)
    {
      return internal::CPD_DIMTREE<Tensor_,execution::openmpi_policy>()(tnsX,R,options);
    }
```

```
  else
    return internal::CPD_DIMTREE<Tensor_>()(tnsX,R,options);
}

/*
 * Interface of Canonical Polyadic Decomposition(cpd) with the use of Dimension
 * Trees. Sequential Policy.
 *
 * @tparam Tensor_      Type(data type and order) of input Tensor.
 * @param  tnsX    [in] The given Tensor to be factorized of @c Tensor_ type,
 *                      with @c double data.
 * @param  R       [in] The rank of decomposition.
 * @param  options [in] User's @c options, other than the default. It must be of
 *                      @c partensor::Options<partensor::Tensor<order» type,
 *                      where @c order must be in range of @c [3-8].
 *
 * @returns An object of type @c Status with the results of the algorithm.
 */
template<typename Tensor_>
auto cpdDimTree(Tensor_          const &tnsX,
                std::size_t      const  R,
                Options<Tensor_> const &options)
{
  return internal::CPD_DIMTREE<Tensor_,execution::sequenced_policy>()(tnsX,R,options);
}

template <typename Tensor_, typename MatrixArray_, typename ExecutionPolicy>

 execution::internal::enable_if_execution_policy<ExecutionPolicy,Status<Tensor_,execution::execution_policy_t<Execution
cpdDimTree( ExecutionPolicy       &&,
            Tensor_          const &tnsX,
            std::size_t      const  R,
            MatrixArray_     const &factorsInit )
{
  using  ExPolicy = execution::execution_policy_t<ExecutionPolicy>;

  if constexpr (std::is_same_v<ExPolicy,execution::sequenced_policy>)
  {
    return internal::CPD_DIMTREE<Tensor_>()(tnsX,R,factorsInit);
  }
  else if constexpr (std::is_same_v<ExPolicy,execution::openmpi_policy>)
  {
    return internal::CPD_DIMTREE<Tensor_,execution::openmpi_policy>()(tnsX,R,factorsInit);
  }
  else
    return internal::CPD_DIMTREE<Tensor_>()(tnsX,R,factorsInit);
}

/*
 * Interface of Canonical Polyadic Decomposition(cpd) with the use of Dimension
 * Trees. Sequential Policy.
 *
 * @tparam Tensor_          Type(data type and order) of input Tensor.
 * @tparam MatrixArray_     An @c stl array, where the initialized factors will
 *                          be stored. Its size must be equal to the Tensor's @c tnsX
 *                          @c order. The type can be either @c partensor::Matrix,
 *                          or @c partensor::Tensor<2>.
 *
 * @param  tnsX        [in] The given Tensor to be factorized of @c Tensor_ type,
 *                          with @c double data.
 * @param  R           [in] The rank of decomposition.
 * @param  factorsInit [in] Uses initialized factors instead of randomly generated. The
 *                          data can be either @c partensor::Matrix, or
 *                          @c partensor::Tensor<2> type and stored in an @c stl array
 *                          with size same as the @c order of @c tnsX.
 *
 * @returns An object of type @c Status with the results of the algorithm.
 */
template<typename Tensor_, typename MatrixArray_>
auto cpdDimTree(Tensor_      const &tnsX,
                std::size_t  const  R,
                MatrixArray_ const &factorsInit)
{
  return internal::CPD_DIMTREE<Tensor_, execution::sequenced_policy>()(tnsX,R,factorsInit);
}

 template <typename Tensor_, typename MatrixArray_, typename ExecutionPolicy>

 execution::internal::enable_if_execution_policy<ExecutionPolicy,Status<Tensor_,execution::execution_policy_t<Execution
 cpdDimTree( ExecutionPolicy       &&,
             Tensor_          const &tnsX,
             std::size_t      const  R,
             Options<Tensor_,execution::execution_policy_t<ExecutionPolicy>,DefaultValues> const
 &options,
             MatrixArray_     const &factorsInit )
 {
   using  ExPolicy = execution::execution_policy_t<ExecutionPolicy>;
```

```

    if constexpr (std::is_same_v<ExPolicy,execution::sequenced_policy>)
    {
      return internal::CPD_DIMTREE<Tensor_>()(tnsX,R,options,factorsInit);
    }
    else if constexpr (std::is_same_v<ExPolicy,execution::openmpi_policy>)
    {
      return internal::CPD_DIMTREE<Tensor_,execution::openmpi_policy>()(tnsX,R,options,factorsInit);
    }
    else
      return internal::CPD_DIMTREE<Tensor_>()(tnsX,R,options,factorsInit);
  }

  /*
   * Interface of Canonical Polyadic Decomposition(cpd) with the use of Dimension
   * Trees. Sequential Policy.
   *
   * @tparam Tensor_          Type(data type and order) of input Tensor.
   * @tparam MatrixArray_     An @c stl array, where the initialized factors will
   *                          be stored. Its size must be equal to the Tensor's @c tnsX
   *                          @c order. The type can be either @c partensor::Matrix,
   *                          or @c partensor::Tensor<2>.
   *
   * @param  tnsX        [in] The given Tensor to be factorized of @c Tensor_ type,
   *                          with @c double data.
   * @param  R           [in] The rank of decomposition.
   * @param  options     [in] User's @c options, other than the default. It must be of
   *                          @c partensor::Options<partensor::Tensor<order» type,
   *                          where @c order must be in range of @c [3-8].
   * @param  factorsInit [in] Uses initialized factors instead of randomly generated. The
   *                          data must be of @c partensor::Matrix type and stored in an
   *                          @c stl array with size same as the @c order of @c tnsX.
   *
   * @returns An object of type @c Status with the results of the algorithm.
   */
  template<typename Tensor_, typename MatrixArray_>
  auto cpdDimTree(Tensor_          const &tnsX,
                  std::size_t      const  R,
                  Options<Tensor_> const &options,
                  MatrixArray_     const &factorsInit)
  {
    return internal::CPD_DIMTREE<Tensor_,execution::sequenced_policy>()(tnsX,R,options,factorsInit);
  }

  template <typename ExecutionPolicy, std::size_t TnsSize>

  execution::internal::enable_if_execution_policy<ExecutionPolicy,Status<Tensor<static_cast<int>(TnsSize)>,execution::exe
  cpdDimTree( ExecutionPolicy                 &&,
              std::array<int, TnsSize> const &tnsDims,
              std::size_t              const  R,
              std::string              const &path )
  {
    using  ExPolicy = execution::execution_policy_t<ExecutionPolicy>;
    using  Tensor_  = Tensor<static_cast<int>(TnsSize)>;

    if constexpr (std::is_same_v<ExPolicy,execution::sequenced_policy>)
    {
      return internal::CPD_DIMTREE<Tensor_>()(tnsDims,R,path);
    }
    else if constexpr (std::is_same_v<ExPolicy,execution::openmpi_policy>)
    {
      return internal::CPD_DIMTREE<Tensor_,execution::openmpi_policy>()(tnsDims,R,path);
    }
    else
      return internal::CPD_DIMTREE<Tensor_>()(tnsDims,R,path);
  }

  /*
   * Interface of Canonical Polyadic Decomposition(cpd) with the use of Dimension
   * Trees. Sequential Policy.
   *
   * With this version of @c cpdDimTree, the Tensor can be read from a file, specified
   * in @c path variable.
   *
   * @tparam TnsSize         Order of the input Tensor.
   *
   * @param  tnsDims     [in] @c Stl array containing the Tensor dimensions, whose
   *                         length must be same as the Tensor order.
   * @param  R           [in] The rank of decomposition.
   * @param  path        [in] The path where the tensor is located.
   *
   * @returns An object of type @c Status with the results of the algorithm.
   */
  template<std::size_t TnsSize>
  auto cpdDimTree(std::array<int, TnsSize> const &tnsDims,
                  std::size_t              const  R,
                  std::string              const &path)
```

```
{
  using Tensor_ = Tensor<static_cast<int>(TnsSize)>;
  return internal::CPD_DIMTREE<Tensor_,execution::sequenced_policy>()(tnsDims,R,path);
}

template <typename ExecutionPolicy, std::size_t TnsSize>

execution::internal::enable_if_execution_policy<ExecutionPolicy,Status<Tensor<static_cast<int>(TnsSize)>,execution::exe
cpdDimTree( ExecutionPolicy                &&,
            std::array<int, TnsSize> const &tnsDims,
            std::size_t              const  R,
            std::string              const &path,

Options<Tensor<static_cast<int>(TnsSize)>,execution::execution_policy_t<ExecutionPolicy>,DefaultValues>
const &options )
{
  using  ExPolicy = execution::execution_policy_t<ExecutionPolicy>;
  using  Tensor_  = Tensor<static_cast<int>(TnsSize)>;

  if constexpr (std::is_same_v<ExPolicy,execution::sequenced_policy>)
  {
    return internal::CPD_DIMTREE<Tensor_>()(tnsDims,R,path,options);
  }
  else if constexpr (std::is_same_v<ExPolicy,execution::openmpi_policy>)
  {
    return internal::CPD_DIMTREE<Tensor_,execution::openmpi_policy>()(tnsDims,R,path,options);
  }
  else
    return internal::CPD_DIMTREE<Tensor_>()(tnsDims,R,path,options);
}

/*
 * Interface of Canonical Polyadic Decomposition(cpd) with the use of Dimension
 * Trees. Sequential Policy.
 *
 * With this version of @c cpdDimTree, the Tensor can be read from a file, specified
 * in @c path variable.
 *
 * @tparam TnsSize         Order of the input Tensor.
 *
 * @param  tnsDims    [in] @c Stl array containing the Tensor dimensions, whose
 *                         length must be same as the Tensor order.
 * @param  R          [in] The rank of decomposition.
 * @param  path       [in] The path where the tensor is located.
 * @param  options    [in] User's @c options, other than the default. It must be of
 *                         @c partensor::Options<partensor::Tensor<order>> type,
 *                         where @c order must be in range of @c [3-8].
 *
 * @returns An object of type @c Status with the results of the algorithm.
 */
template<std::size_t TnsSize>
auto cpdDimTree(std::array<int, TnsSize>                   const &tnsDims,
                std::size_t                                const  R,
                std::string                                const &path,
                Options<Tensor<static_cast<int>(TnsSize)>> const &options )
{
  using Tensor_ = Tensor<static_cast<int>(TnsSize)>;
  return internal::CPD_DIMTREE<Tensor_,execution::sequenced_policy>()(tnsDims,R,path,options);
}

template <typename ExecutionPolicy, std::size_t TnsSize>

execution::internal::enable_if_execution_policy<ExecutionPolicy,Status<Tensor<static_cast<int>(TnsSize)>,execution::exe
cpdDimTree( ExecutionPolicy                          &&,
            std::array<int, TnsSize>           const &tnsDims,
            std::size_t                        const  R,
            std::array<std::string, TnsSize+1> const &paths )
{
  using  ExPolicy = execution::execution_policy_t<ExecutionPolicy>;
  using  Tensor_  = Tensor<static_cast<int>(TnsSize)>;

  if constexpr (std::is_same_v<ExPolicy,execution::sequenced_policy>)
  {
    return internal::CPD_DIMTREE<Tensor_>()(tnsDims,R,paths);
  }
  else if constexpr (std::is_same_v<ExPolicy,execution::openmpi_policy>)
  {
    return internal::CPD_DIMTREE<Tensor_,execution::openmpi_policy>()(tnsDims,R,paths);
  }
  else
    return internal::CPD_DIMTREE<Tensor_>()(tnsDims,R,paths);
}

/*
 * Interface of Canonical Polyadic Decomposition(cpd) with the use of Dimension
 * Trees. Sequential Policy.
 *
```

```
   * With this version of @c cpdDimTree, the Tensor can be read from a file, specified
   * in @c path variable.
   *
   * @tparam TnsSize         Order of the input Tensor.
   *
   * @param  tnsDims    [in] @c Stl array containing the Tensor dimensions, whose
   *                         length must be same as the Tensor order.
   * @param  R          [in] The rank of decomposition.
   * @param  paths      [in] An @c stl array containing paths for the Tensor to be
   *                         factorized and after that the paths for the initialized
   *                         factors.
   *
   * @returns An object of type @c Status with the results of the algorithm.
   */
  template<std::size_t TnsSize>
  auto cpdDimTree(std::array<int, TnsSize>            const &tnsDims,
                  std::size_t                          const  R,
                  std::array<std::string, TnsSize+1> const &paths  )
  {
    using Tensor_ = Tensor<static_cast<int>(TnsSize)>;
    return internal::CPD_DIMTREE<Tensor_,execution::sequenced_policy>()(tnsDims,R,paths);
  }

  template <typename ExecutionPolicy, std::size_t TnsSize>

  execution::internal::enable_if_execution_policy<ExecutionPolicy,Status<Tensor<static_cast<int>(TnsSize)>,execution::exe
  cpdDimTree( ExecutionPolicy                                 &&,
              std::array<int, TnsSize>            const &tnsDims,
              std::size_t                          const  R,
              std::array<std::string, TnsSize+1> const &paths,

  Options<Tensor<static_cast<int>(TnsSize)>,execution::execution_policy_t<ExecutionPolicy>,DefaultValues>
  const &options )
  {
    using  ExPolicy = execution::execution_policy_t<ExecutionPolicy>;
    using  Tensor_  = Tensor<static_cast<int>(TnsSize)>;

    if constexpr (std::is_same_v<ExPolicy,execution::sequenced_policy>)
    {
      return internal::CPD_DIMTREE<Tensor_>()(tnsDims,R,paths,options);
    }
    else if constexpr (std::is_same_v<ExPolicy,execution::openmpi_policy>)
    {
      return internal::CPD_DIMTREE<Tensor_,execution::openmpi_policy>()(tnsDims,R,paths,options);
    }
    else
      return internal::CPD_DIMTREE<Tensor_>()(tnsDims,R,paths,options);
  }

  /*
   * Interface of Canonical Polyadic Decomposition(cpd) with the use of Dimension
   * Trees. Sequential Policy.
   *
   * With this version of @c cpdDimTree, the Tensor can be read from a file, specified
   * in @c path variable.
   *
   * @tparam TnsSize         Order of the input Tensor.
   *
   * @param  tnsDims    [in] @c Stl array containing the Tensor dimensions, whose
   *                         length must be same as the Tensor order.
   * @param  R          [in] The rank of decomposition.
   * @param  paths      [in] An @c stl array containing paths for the Tensor to be
   *                         factorized and after that the paths for the initialized
   *                         factors.
   * @param  options    [in] User's @c options, other than the default. It must be of
   *                         @c partensor::Options<partensor::Tensor<order» type,
   *                         where @c order must be in range of @c [3-8].
   *
   * @returns An object of type @c Status with the results of the algorithm.
   */
  template<std::size_t TnsSize>
  auto cpdDimTree(std::array<int, TnsSize>                    const &tnsDims,
                  std::size_t                                  const  R,
                  std::array<std::string, TnsSize+1>         const &paths,
                  Options<Tensor<static_cast<int>(TnsSize)>» const &options )
  {
    using Tensor_ = Tensor<static_cast<int>(TnsSize)>;
    return internal::CPD_DIMTREE<Tensor_,execution::sequenced_policy>()(tnsDims,R,paths,options);
  }

} // end namespace partensor

#endif // PARTENSOR_CPD_DIMTREE_DIM_TREE_HPP
```

8.9 CpdDimTreeMpi.hpp File Reference

```
#include <math.h>
#include "PartialCwiseProd.hpp"
#include "TensorOperations.hpp"
#include "unsupported/Eigen/MatrixFunctions"
```

Classes

- struct CPD_DIMTREE< Tensor_, execution::openmpi_policy >

8.9.1 Detailed Description

Implements the Canonical Polyadic Decomposition(cpd) using `MPI`. and Dimensional Trees. Make use of `spdlog` library in order to write output in a log file in `"../log"`.

8.10 CpdDimTreeMpi.hpp

Go to the documentation of this file.

```
#ifndef DOXYGEN_SHOULD_SKIP_THIS
#endif // DOXYGEN_SHOULD_SKIP_THIS
/*********************************************************************/
#if !defined(PARTENSOR_CPD_DIMTREE_DIM_TREE_HPP)
#error "CpdDimTreeMpi can only included inside CpdDimTree"
#endif /* PARTENSOR_CPD_DIMTREE_DIM_TREE_HPP */

#include <math.h>
#include "PartialCwiseProd.hpp"
#include "TensorOperations.hpp"
#include "unsupported/Eigen/MatrixFunctions"

namespace partensor
{

    inline namespace v1 {
        namespace internal {

            template<typename Tensor_>
            struct CPD_DIMTREE<Tensor_,execution::openmpi_policy> : public CPD_DIMTREE_Base<Tensor_>
            {
                using          CPD_DIMTREE_Base<Tensor_>::TnsSize;
                using          CPD_DIMTREE_Base<Tensor_>::lastFactor;
                using typename CPD_DIMTREE_Base<Tensor_>::TensorMatrixType;
                using typename CPD_DIMTREE_Base<Tensor_>::Dimensions;
                using typename CPD_DIMTREE_Base<Tensor_>::MatrixArray;
                using typename CPD_DIMTREE_Base<Tensor_>::DataType;
                using typename CPD_DIMTREE_Base<Tensor_>::IntArray;
                using typename CPD_DIMTREE_Base<Tensor_>::FactorArray;
                using typename CPD_DIMTREE_Base<Tensor_>::IndexPair;

                // For MPI usage
                using CartCommunicator = partensor::cartesian_communicator; // From ParallelWrapper.hpp
                using CartCommVector   = std::vector<CartCommunicator>;
                using IntVector        = std::vector<int>;
                using Int2DVector      = std::vector<std::vector<int»;

                using Options = partensor::Options<Tensor_,execution::openmpi_policy,DefaultValues>;
                using Status  = partensor::Status<Tensor_,execution::openmpi_policy,DefaultValues>;

                // Variables that will be used in cpd with
                // Dimension Trees implementations.
                struct Member_Variables
                {
                    MPI_Communicator &world = Partensor()->MpiCommunicator(); // MPI_COMM_WORLD

                    double          local_f_value;
```

```
            int             RxR;
            int             world_size;
            const IndexPair product_dims = { Eigen::IndexPair<int>(0, 0) }; // used for tensor contractions

            Int2DVector     displs_subTns;       // skipping dimension "rows" for each subtensor
            Int2DVector     displs_subTns_R;     // skipping dimension "rows" for each subtensor times R ( for MPI communication purposes )
            Int2DVector     subTnsDims;          // dimensions of subtensor
            Int2DVector     subTnsDims_R;        // dimensions of subtensor times R ( for MPI communication purposes )
            Int2DVector     displs_local_update; // displacement in the local factor for update rows
            Int2DVector     send_recv_counts;    // rows to be communicated after update times R

            CartCommVector  layer_comm;
            CartCommVector  fiber_comm;

            IntArray        layer_rank;
            IntArray        fiber_rank;
            IntArray        rows_for_update;
            IntArray        subTns_offsets;
            IntArray        subTns_extents;
            IntArray        labelSet;        // starting label set for root

            MatrixArray     local_factors;
            MatrixArray     local_factors_T;
            MatrixArray     layer_factors;
            MatrixArray     layer_factors_T;
            MatrixArray     local_mttkrp;
            MatrixArray     layer_mttkrp;
            MatrixArray     local_mttkrp_T;
            MatrixArray     layer_mttkrp_T;

            Matrix          cwise_factor_product;
            Matrix          factor_T_factor;
            Matrix          last_cov;
            Matrix          temp_matrix;
            Matrix          tnsX_mat_lastFactor_T;
            Matrix          nesterov_old_layer_factor;

            Tensor_         subTns;
            Tensor_         subTnsX_approx;  // tensor in order to compute starting f_value from random generated factors

            FactorArray     factors;
            FactorArray     layer_factors_dimTree;
            FactorArray     norm_factors;
            FactorArray     old_factors;

            std::array<int, 2> subfactor_offsets;
            std::array<int, 2> subfactor_extents;

            typename FactorArray::iterator status_factor_it;
            typename FactorArray::iterator layer_factor_it;

            bool             all_orthogonal = true;
            int              weight_factor;

            /*
             * Calculates if the number of processors given from terminal
             * are equal to the processors in the implementation.
             *
             * @param procs [in] @c stl array with the number of processors per
             *                   dimension of the tensor.
             */
            void check_processor_avaliability(std::array<int, TnsSize> const &procs)
            {
                // MPI_Environment &env = Partensor()->MpiEnvironment();
                world_size = world.size();
                //  numprocs must be product of options.proc_per_mode
                if (std::accumulate(procs.begin(), procs.end(), 1,
                                    std::multiplies<int>()) != world_size && world.rank() == 0) {
                Partensor()->Logger()->error("The product of the processors per mode must be equal to {}\n", world_size);
                // env.abort(-1);
                }
            }

            Member_Variables() = default;
            Member_Variables(int R, std::array<int, TnsSize> &procs) :  local_f_value(0.0),
                                                                        RxR(R*R),
                                                                        displs_subTns(TnsSize),
                                                                        displs_subTns_R(TnsSize),
                                                                        subTnsDims(TnsSize),
```

```
                                                                                subTnsDims_R(TnsSize),
                                                                                displs_local_update(TnsSize),
                                                                                send_recv_counts(TnsSize)
                {
                    check_processor_avaliability(procs);
                    layer_comm.reserve(TnsSize);
                    fiber_comm.reserve(TnsSize);
                }

                Member_Variables(Member_Variables const &) = default;
                Member_Variables(Member_Variables      &&) = default;

                Member_Variables &operator=(Member_Variables const &) = default;
                Member_Variables &operator=(Member_Variables      &&) = default;
            };

            /*
             * In case option variable @c writeToFile is enabled then, before the end
             * of the algorithm writes the resulted factors in files, where their
             * paths are specified before compiling in @ options.final_factors_path.
             *
             * @param  st  [in] Struct where the returned values of @c CpdDimTree are stored.
             */
            void writeFactorsToFile(Status const &st)
            {
                std::size_t size;
                for(std::size_t i=0; i<TnsSize; ++i)
                {
                    size = st.factors[i].rows() * st.factors[i].cols();
                    partensor::write(st.factors[i],
                                     st.options.final_factors_paths[i],
                                     size);
                }
            }

            /*
             * Compute the cost function value based on the initial factors.
             *
             * @param  grid_comm [in]     The communication grid, where the processors
             *                            communicate their cost function.
             * @param  R         [in]     The rank of decomposition.
             * @param  mv        [in]     Struct where ALS variables are stored.
             * @param  st        [in,out] Struct where the returned values of @c CpdDimTree are stored.
             *                            In this case the cost function value  and the Frobenius
             *                            squared norm of the tensor are updated.
             */
            void cost_function_init(CartCommunicator const &grid_comm,
                                    std::size_t       const  R,
                                    Member_Variables        &mv,
                                    Status                  &st )
            {
                mv.subTnsX_approx.resize(mv.subTns_extents);
                CpdGen(mv.layer_factors_dimTree, R, mv.subTnsX_approx);

                // communicate the squared norm of sub tensor, in order to compute frob_tns
                all_reduce( grid_comm,
                            square_norm(mv.subTns),
                            st.frob_tns,
                            std::plus<double>());
                // communication among all processors for f_value
                all_reduce( grid_comm,
                            square_norm(mv.subTns - mv.subTnsX_approx),
                            st.f_value,
                            std::plus<double>());
                st.f_value = sqrt(st.f_value);
            }

            /*
             * Compute the cost function value at the end of each outer iteration
             * based on the last factor.
             *
             * @param  grid_comm [in]     The communication grid, where the processors
             *                            communicate their cost function.
             * @param  mv        [in]     Struct where ALS variables are stored.
             * @param  st        [in,out] Struct where the returned values of @c CpdDimTree are stored.
             *                            In this case the cost function value is updated.
             */
            void cost_function( CartCommunicator const &grid_comm,
                                Member_Variables       &mv,
                                Status                 &st )
            {
                mv.local_f_value = ((mv.layer_mttkrp_T[lastFactor] * mv.layer_factors[lastFactor]).trace());
```

```
                    all_reduce( grid_comm,
                                inplace(&mv.local_f_value),
                                1,
                                std::plus<double>() );
                    st.f_value = sqrt(st.frob_tns - 2 * mv.local_f_value + (mv.cwise_factor_product.cwiseProduct(mv.factor_T_factor).sum()));
                }

                /*
                 * Compute the cost function value at the end of each outer iteration
                 * based on the last accelerated factor.
                 *
                 * @param  grid_comm         [in] The communication grid, where the processors
                 *                                communicate their cost function.
                 * @param  mv                [in] Struct where ALS variables are stored.
                 * @param  st                [in] Struct where the returned values of @c CpdDimTree
                 *                                are stored.
                 * @param  factors           [in] Accelerated factors.
                 * @param  factors_T_factors [in] Gramian matrices of factors.
                 *
                 * @returns The cost function calculated with the accelerated factors.
                 */
                 double accel_cost_function(CartCommunicator const &grid_comm,
                                            Member_Variables const &mv,
                                            Status           const &st,
                                            MatrixArray      const &factors,
                                            MatrixArray      const &factors_T_factors)
                 {
                    double local_f_value =
                        ((PartialKhatriRao(factors, lastFactor).transpose() * mv.tnsX_mat_lastFactor_T) * factors[lastFactor]).trace();
                    all_reduce( grid_comm,
                                inplace(&local_f_value),
                                1,
                                std::plus<double>() );
                    Matrix cwiseFactor_prod = PartialCwiseProd(factors_T_factors, lastFactor) * factors_T_factors[lastFactor];
                    return sqrt(st.frob_tns - 2 * local_f_value + cwiseFactor_prod.trace());
                }

                /*
                 * Make use of the dimensions and the number of processors per dimension
                 * and then calculates the dimensions of the subtensor and subfactor for
                 * each processor.
                 *
                 * Also initialize the FactorDimTree struct for each processor with the
                 * data from factors.
                 *
                 * @tparam Dimensions          Array type containing the Tensor dimensions.
                 *
                 * @param  tnsDims    [in]     Tensor Dimensions. Each index contains the corresponding
                 *                             factor's rows length.
                 * @param  st         [in]     Struct where the returned values of @c CpdDimTree are stored.
                 * @param  R          [in]     The rank of decomposition.
                 * @param  mv         [in,out] Struct where ALS variables are stored.
                 *                             Updates @c stl arrays with dimensions for subtensors and
                 *                             subfactors.
                 */
                template<typename Dimensions>
                void compute_sub_dimensions(Dimensions       const &tnsDims,
                                            Status           const &st,
                                            std::size_t      const  R,
                                            Member_Variables        &mv)
                {
                    mv.status_factor_it = mv.factors.begin();
                    mv.layer_factor_it  = mv.layer_factors_dimTree.begin();
                    for (std::size_t i = 0; i < TnsSize; ++i)
                    {
                        DisCount(mv.displs_subTns[i], mv.subTnsDims[i], st.options.proc_per_mode[i], tnsDims[i], 1);
                        // for fiber communication and Gatherv
                        DisCount(mv.displs_subTns_R[i], mv.subTnsDims_R[i], st.options.proc_per_mode[i], tnsDims[i], static_cast<int>(R));
                        // information per layer
                        DisCount(mv.displs_local_update[i], mv.send_recv_counts[i], mv.world_size / st.options.proc_per_mode[i],
                                                            mv.subTnsDims[i][mv.fiber_rank[i]], static_cast<int>(R));
                        // sizes and skips for sub factor
                        mv.subfactor_offsets = { mv.displs_subTns[i][mv.fiber_rank[i]], 0 };
                        mv.subfactor_extents = { mv.subTnsDims[i][mv.fiber_rank[i]], static_cast<int>(R) };
                        // get sub factor and compute its covariance matrix
                        mv.layer_factor_it->factor     = mv.status_factor_it->factor.slice(mv.subfactor_offsets, mv.subfactor_extents);
                        mv.layer_factor_it->gramian =
```

```
mv.layer_factor_it->factor.contract(mv.layer_factor_it->factor, mv.product_dims);
                        all_reduce( mv.fiber_comm[i],
                                    inplace(mv.layer_factor_it->gramian.data()),
                                    mv.RxR,
                                    std::plus<double>() );

                        mv.rows_for_update[i] = mv.send_recv_counts[i][mv.layer_rank[i]] / static_cast<int>(R);
                        // sizes and skips for sub tensor
                        mv.subTns_offsets[i]  = mv.displs_subTns[i][mv.fiber_rank[i]];
                        mv.subTns_extents[i]  = mv.subTnsDims[i][mv.fiber_rank[i]];
                        mv.local_mttkrp_T[i].resize(R, mv.rows_for_update[i]);
                        mv.layer_factors_T[i].resize(R, mv.subTnsDims[i][mv.fiber_rank[i]]);

                        mv.status_factor_it++;
                        mv.layer_factor_it++;
                    }
                }

                /*
                 * Make use of the dimensions and the number of processors per dimension
                 * and then calculates the dimensions of the subtensor and subfactor for
                 * each processor.
                 *
                 * After reading from files the given factors, then initializes the
                 * FactorDimTree struct for each processor with the data read from files.
                 *
                 * @tparam Dimensions          Array type containing the Tensor dimensions.
                 *
                 * @param  tnsDims    [in]     Tensor Dimensions. Each index contains the corresponding
                 *                             factor's rows length.
                 * @param  st         [in]     Struct where the returned values of @c CpdDimTree are stored.
                 * @param  R          [in]     The rank of decomposition.
                 *
                 * @param  paths      [in]     Paths where the starting point-factors are located.
                 * @param  mv         [in,out] Struct where ALS variables are stored.
                 *                             Updates @c stl arrays with dimensions for subtensors and
                 *                             subfactors.
                 */
                template<typename Dimensions>
                void compute_sub_dimensions(Dimensions                          const &tnsDims,
                                            Status                              const &st,
                                            std::size_t                         const  R,
                                            std::array<std::string, TnsSize+1> const &paths,
                                            Member_Variables                          &mv)
                {
                    mv.layer_factor_it  = mv.layer_factors_dimTree.begin();
                    for (std::size_t i = 0; i < TnsSize; ++i)
                    {
                        DisCount(mv.displs_subTns[i], mv.subTnsDims[i], st.options.proc_per_mode[i], tnsDims[i], 1);
                        // for fiber communication and Gatherv
                        DisCount(mv.displs_subTns_R[i], mv.subTnsDims_R[i], st.options.proc_per_mode[i], tnsDims[i], static_cast<int>(R));
                        // information per layer
                        DisCount(mv.displs_local_update[i], mv.send_recv_counts[i], mv.world_size / st.options.proc_per_mode[i],
                                                                mv.subTnsDims[i][mv.fiber_rank[i]], static_cast<int>(R));
                        // sizes and skips for sub factor
                        mv.subfactor_offsets = { mv.displs_subTns[i][mv.fiber_rank[i]], 0 };
                        mv.subfactor_extents = { mv.subTnsDims[i][mv.fiber_rank[i]], static_cast<int>(R) };

                        mv.temp_matrix = Matrix(mv.subTnsDims[i][mv.fiber_rank[i]],static_cast<int>(R));
                        read( paths[i+1],
                              mv.subTnsDims[i][mv.fiber_rank[i]]*static_cast<int>(R),
                              mv.displs_subTns_R[i][mv.fiber_rank[i]],
                              mv.temp_matrix );

                        // get sub factor and compute its covariance matrix
                        mv.layer_factor_it->factor     = matrixToTensor(mv.temp_matrix, mv.subTnsDims[i][mv.fiber_rank[i]], static_cast<int>(R));
                        mv.layer_factor_it->gramian = mv.layer_factor_it->factor.contract(mv.layer_factor_it->factor, mv.product_dims);
                        all_reduce( mv.fiber_comm[i],
                                    inplace(mv.layer_factor_it->gramian.data()),
                                    mv.RxR,
                                    std::plus<double>() );

                        mv.rows_for_update[i] = mv.send_recv_counts[i][mv.layer_rank[i]] / static_cast<int>(R);
                        // sizes and skips for sub tensor
                        mv.subTns_offsets[i]  = mv.displs_subTns[i][mv.fiber_rank[i]];
                        mv.subTns_extents[i]  = mv.subTnsDims[i][mv.fiber_rank[i]];
                        mv.local_mttkrp_T[i].resize(R, mv.rows_for_update[i]);
```

```
                mv.layer_factors_T[i].resize(R, mv.subTnsDims[i][mv.fiber_rank[i]]);

                mv.layer_factor_it++;
            }
        }

        /*
         * Based on each factor's constraint, a different
         * update function is used at every outer iteration.
         *
         * Computes also factor^T * factor at the end.
         *
         * @param  idx  [in]     Factor to be updated.
         * @param  R    [in]     The rank of decomposition.
         * @param  st   [in]     Struct where the returned values of @c CpdDimTree
         *                       are stored.
         * @param  mv   [in,out] Struct where ALS variables are stored.
         *                       Updates the current layer factor (@c Matrix type) and
         *                       then updates the same factor of @c FactorDimTree type.
         */
        void update_factor( int              const  idx,
                            std::size_t      const  R,
                            Status           const &st,
                            Member_Variables       &mv   )
        {
            switch ( st.options.constraints[idx] )
            {
                case Constraint::unconstrained:
                {
                    // communicate the local mttkrp
                    v2::reduce_scatter( mv.layer_comm[idx],
                                        mv.layer_mttkrp_T[idx],
                                        mv.send_recv_counts[idx][0],
                                        mv.local_mttkrp_T[idx] );

                    mv.local_mttkrp[idx] = mv.local_mttkrp_T[idx].transpose();

                    if(mv.rows_for_update[idx] != 0)
                        mv.local_factors[idx] = mv.local_mttkrp[idx] * mv.cwise_factor_product.inverse(); // Compute new factor

                    break;
                }
                case Constraint::nonnegativity:
                {
                    // communicate the local mttkrp
                    v2::reduce_scatter( mv.layer_comm[idx],
                                        mv.layer_mttkrp_T[idx],
                                        mv.send_recv_counts[idx][0],
                                        mv.local_mttkrp_T[idx] );

                    mv.local_mttkrp[idx]          = mv.local_mttkrp_T[idx].transpose();
                    mv.nesterov_old_layer_factor = mv.layer_factors[idx];
                    if (mv.rows_for_update[idx] != 0)
                    {
                        NesterovMNLS(mv.cwise_factor_product, mv.local_mttkrp[idx], st.options.nesterov_delta_1,
                                        st.options.nesterov_delta_2, mv.local_factors[idx]);
                    }
                    break;
                }
                case Constraint::orthogonality:
                {
                    all_reduce( mv.layer_comm[idx],
                                inplace(mv.layer_mttkrp[idx].data()),
                                mv.subTnsDims_R[idx][mv.fiber_rank[idx]],
                                std::plus<double>() );

                    if (mv.rows_for_update[idx] != 0) {
                        mv.local_mttkrp[idx]      = mv.layer_mttkrp[idx].block(mv.displs_local_update[idx][mv.layer_rank[idx]] / static_cast<int>(R), 0,
                        mv.rows_for_update[idx],                             static_cast<int>(R));
                        mv.temp_matrix.noalias() = mv.layer_mttkrp[idx].transpose() * mv.layer_mttkrp[idx];
                    }

                    all_reduce( mv.fiber_comm[idx],
                                inplace(mv.temp_matrix.data()),
                                mv.RxR,
                                std::plus<double>() );

                    Eigen::SelfAdjointEigenSolver<Matrix> eigensolver(mv.temp_matrix);
                    mv.temp_matrix.noalias() = (eigensolver.eigenvectors())
                                                 * (eigensolver.eigenvalues().cwiseInverse().cwiseSqrt().asDiagonal())
                                                 * (eigensolver.eigenvectors().transpose());
```

```

                        if(mv.rows_for_update[idx] != 0)
                            mv.local_factors[idx].noalias() = mv.local_mttkrp[idx] * mv.temp_matrix;
                            // mv.local_factors[idx].noalias() = mv.local_mttkrp[idx] * (mv.temp_matrix.pow(-0.5));
                        break;
                    }
                    case Constraint::sparsity:
                    break;
                    default: // in case of Constraint::constant
                    break;
                } // end of constraints switch

                if (st.options.constraints[idx] != Constraint::constant)
                {
                    mv.local_factors_T[idx] = mv.local_factors[idx].transpose();
                    v2::all_gatherv(mv.layer_comm[idx],
                                    mv.local_factors_T[idx],
                                    mv.send_recv_counts[idx][mv.layer_rank[idx]],
                                    mv.send_recv_counts[idx][0],
                                    mv.displs_local_update[idx][0],
                                    mv.layer_factors_T[idx] );

                    mv.layer_factors[idx] = mv.layer_factors_T[idx].transpose();
                    mv.factor_T_factor    = mv.layer_factors_T[idx] * mv.layer_factors[idx];
                    all_reduce( mv.fiber_comm[idx],
                                inplace(mv.factor_T_factor.data()),
                                mv.RxR,
                                std::plus<double>() );

                    if(st.options.constraints[idx] == Constraint::nonnegativity)
                    {
                        if ((mv.factor_T_factor.diagonal()).minCoeff()==0)
                        {
                        mv.layer_factors[idx] = 0.9 * mv.layer_factors[idx] + 0.1 * mv.nesterov_old_layer_factor;
                        all_reduce( mv.fiber_comm[idx],
                                    inplace(mv.factor_T_factor.data()),
                                    mv.RxR,
                                    std::plus<double>() );
                        }
                    }

                    mv.layer_factor_it->factor     = matrixToTensor(mv.layer_factors[idx], mv.subTnsDims[idx][mv.fiber_rank[idx]], static_cast<int>(R)); // Map factor from Eigen Matrix to Eigen Tensor
                    mv.layer_factor_it->gramian = matrixToTensor(mv.factor_T_factor, static_cast<int>(R), static_cast<int>(R));   // Map Covariance from Eigen Matrix to Eigen Tensor

                }
            }


            /*
             * At the end of the algorithm processor 0
             * collects each part of the factor that each
             * processor holds and return them in status.factors.
             *
             * @tparam Dimensions       Array type containing the Tensor dimensions.
             *
             * @param  tnsDims [in]     Tensor Dimensions. Each index contains the corresponding
             *                          factor's rows length.
             * @param  R       [in]     The rank of decomposition.
             * @param  mv      [in]     Struct where ALS variables are stored.
             *                          Use variables to compute result factors by gathering each
             *                          part of the factor from processors.
             * @param  st      [in,out] Struct where the returned values of @c CpdDimTree are stored.
             *                          Stores the resulted factors.
             */
            template<typename Dimensions>
            void gather_final_factors(Dimensions       const &tnsDims,
                                      std::size_t      const  R,
                                      Member_Variables       &mv,
                                      Status                 &st)
            {
                for(std::size_t i=0; i<TnsSize; ++i)
                {
                    mv.temp_matrix        = tensorToMatrix(mv.layer_factors_dimTree[i].factor, mv.subTnsDims[i][mv.fiber_rank[i]], static_cast<int>(R));
                    mv.layer_factors_T[i] = mv.temp_matrix.transpose();
                }

                for(std::size_t i=0; i<TnsSize; ++i)
                {
                    mv.temp_matrix.resize(static_cast<int>(R), tnsDims[i]);
                    // Gatherv from all processors to processor with rank 0 the final factors
                    v2::gatherv(mv.fiber_comm[i],
```

```
                            mv.layer_factors_T[i],
                            mv.subTnsDims_R[i][mv.fiber_rank[i]],
                            mv.subTnsDims_R[i][0],
                            mv.displs_subTns_R[i][0],
                            0,
                            mv.temp_matrix );

                    st.factors[i] = mv.temp_matrix.transpose();
                }
            }

            /*
             * @brief Line Search Acceleration
             *
             * Performs an acceleration step in the updated factors, and keeps the accelerated factors when
             * the step succeeds. Otherwise, the acceleration step is ignored.
             * Line Search Acceleration reduces the number outer iterations in the ALS algorithm.
             *
             * @note This implementation ONLY, if factors are of @c Matrix type.
             *
             * @param  grid_comm [in]     MPI communicator where the new cost function value
             *                            will be communicated and computed.
             * @param  R         [in]     Rank of the factorization.
             * @param  mv        [in,out] Struct where ALS variables are stored.
             *                            In case the acceration is successful layer factor^T * factor
             *                            and layer factor variables are updated.
             * @param  st        [in,out] Struct of the returned values of @c CpdDimTree are stored.
             *                            If the acceleration succeeds updates cost function value.

             */
            void line_search_accel(CartCommunicator const &grid_comm,
                                   std::size_t      const  R,
                                   Member_Variables       &mv,
                                   Status                 &st)
            {
                double       f_accel    = 0.0; // Objective Value after the acceleration step
                double const accel_step = pow(st.ao_iter+1,(1.0/(st.options.accel_coeff)));

                Matrix      factor;
                Matrix      old_factor;
                MatrixArray accel_factors;
                MatrixArray accel_gramians;

                for(std::size_t i=0; i<TnsSize; ++i)
                {
                    factor            = tensorToMatrix(mv.layer_factors_dimTree[i].factor, mv.subTnsDims[i][mv.fiber_rank[i]],static_cast<int>(R));
                    old_factor        = tensorToMatrix(mv.old_factors[i].factor, mv.subTnsDims[i][mv.fiber_rank[i]],static_cast<int>(R));
                    accel_factors[i]  = old_factor + accel_step * (factor - old_factor);
                    accel_gramians[i] = accel_factors[i].transpose() * accel_factors[i];
                    all_reduce( mv.fiber_comm[i],
                                inplace(accel_gramians[i].data()),
                                mv.RxR,
                                std::plus<double>() );
                }

                f_accel = accel_cost_function(grid_comm, mv, st, accel_factors, accel_gramians);
                if (st.f_value > f_accel)
                {
                    for(std::size_t i=0; i<TnsSize; ++i)
                    {
                        mv.layer_factors_dimTree[i].factor     = matrixToTensor(accel_factors[i], mv.subTnsDims[i][mv.fiber_rank[i]], static_cast<int>(R));
                        mv.layer_factors_dimTree[i].gramian = matrixToTensor(accel_gramians[i], static_cast<int>(R), static_cast<int>(R));
                    }
                    st.f_value = f_accel;
                    if(grid_comm.rank() == 0)
                        Partensor()->Logger()->info("Acceleration Step SUCCEEDED! at iter: {}", st.ao_iter);
                }
                else
                    st.options.accel_fail++;

                if (st.options.accel_fail==5)
                {
                    st.options.accel_fail=0;
                    st.options.accel_coeff++;
                }
            }

            /*
             * Parallel implementation of als method with MPI.
             *
```

```
             * @tparam Dimensions        Array type containing the Tensor dimensions.
             *
             * @param  grid_comm [in]     MPI communicator where the new cost function value
             *                            will be communicated and computed.
             * @param  tnsDims    [in]     Tensor Dimensions. Each index contains the corresponding
             *                             factor's rows length.
             * @param  R          [in]     The rank of decomposition.
             * @param  mv         [in]     Struct where ALS variables are stored  and being updated
             *                            until a termination condition is true.
             * @param  st         [in,out] Struct where the returned values of @c CpdDimTree are stored.
             */
            template<typename Dimensions>
            void als(CartCommunicator const &grid_comm,
                     Dimensions       const &tnsDims,
                     std::size_t      const  R,
                     Member_Variables        &mv,
                     Status                  &status)
            {
                mv.tnsX_mat_lastFactor_T = (Matricization(mv.subTns, lastFactor)).transpose();
                if(status.options.normalization)
                {
                  choose_normilization_factor(status, mv.all_orthogonal, mv.weight_factor);
                }

                std::iota(mv.labelSet.begin(), mv.labelSet.end(), 1); // increments from 1 to labelSet.size()

                ExprTree<TnsSize> tree;
                tree.Create(mv.labelSet, mv.subTns_extents, R, mv.subTns);

                mv.layer_factor_it = mv.layer_factors_dimTree.begin();
                for(std::size_t k = 0; k<TnsSize; k++)
                {
                    mv.layer_factor_it->leaf = static_cast<TnsNode<1>*>(search_leaf(k+1, tree));
                    mv.layer_factor_it++;
                }

                // Wait for all processors to reach here
                grid_comm.barrier();

                // ---- Loop until ALS converges ----
                while(1)
                {
                    status.ao_iter++;
                    mv.layer_factor_it = mv.layer_factors_dimTree.begin();
                    if (!grid_comm.rank())
                        Partensor()->Logger()->info("iter: {} -- fvalue: {} -- relative_costFunction: {}", status.ao_iter,
                                                  status.f_value, status.rel_costFunction);

                    for(std::size_t i=0; i<TnsSize; i++)
                    {
                        mv.layer_factor_it->leaf->UpdateTree(TnsSize, i, mv.layer_factor_it);


                        // Maps from Eigen Tensor to Eigen Matrix
                        mv.layer_mttkrp_T[i]     = tensorToMatrix(*reinterpret_cast<TensorMatrixType *>(mv.layer_factor_it->leaf->TensorX()),
                                                                  static_cast<int>(R), mv.subTnsDims[i][mv.fiber_rank[i]]);
                        mv.layer_mttkrp[i]       = mv.layer_mttkrp_T[i].transpose();
                        mv.cwise_factor_product = tensorToMatrix(mv.layer_factor_it->leaf->Gramian(), static_cast<int>(R), static_cast<int>(R));
                        mv.layer_factors[i]      = tensorToMatrix(mv.layer_factor_it->factor, mv.subTnsDims[i][mv.fiber_rank[i]], static_cast<int>(R));
                        mv.local_factors[i]      = mv.layer_factors[i].block(mv.displs_local_update[i][mv.layer_rank[i]] / static_cast<int>(R), 0,
                                                                                         mv.rows_for_update[i], static_cast<int>(R));

                        update_factor(i, R, status, mv);
                        mv.layer_factor_it++;
                    }

                    cost_function( grid_comm, mv, status );
                    status.rel_costFunction = status.f_value / sqrt(status.frob_tns);
                    if(status.options.normalization && !mv.all_orthogonal)
                        Normalize(mv.weight_factor, static_cast<int>(R), tnsDims, mv.factors);

                    // ---- Terminating condition ----
                    if (status.ao_iter >= status.options.max_iter || status.rel_costFunction < status.options.threshold_error)
                    {
                        gather_final_factors(tnsDims, R, mv, status);
                        if(grid_comm.rank() == 0)
                        {
```

```
                        Partensor()->Logger()->info("Processor 0 collected all {} factors.\n", TnsSize);
                        if(status.options.writeToFile)
                            writeFactorsToFile(status);
                    }
                    break;
                }

                if (status.options.acceleration)
                {
                    mv.norm_factors = mv.layer_factors_dimTree;
                    // ---- Acceleration Step ----
                    if (status.ao_iter > 1)
                        line_search_accel(grid_comm, R, mv, status);

                    mv.old_factors = mv.norm_factors;
                }
            }
        }

        Status operator()(Tensor_     const &tnsX,
                          std::size_t const  R)
        {
            Options options = MakeOptions<Tensor_>(execution::openmpi_policy());
            Status  status(options);
            Member_Variables mv(R, status.options.proc_per_mode);

            // Communicator with cartesian topology
            CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);

            // Functions that create layer and fiber grids.
            create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
            create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);

            // extract dimensions from tensor
            Dimensions const &tnsDims = tnsX.dimensions();
            // produce estimate factors using uniform distribution with entries in [0,1].
            makeFactors(tnsDims, status.options.constraints, R, mv.factors);

            compute_sub_dimensions(tnsDims, status, R, mv);
            // Normalize each layer_factor, compute status.frob_tns and status.f_value
            // Normalize(R, subTns_extents, layer_factors_dimTree);
            // Each processor takes a subtensor from tnsX
            mv.subTns = tnsX.slice(mv.subTns_offsets, mv.subTns_extents);
            cost_function_init(grid_comm, R, mv, status);
            status.rel_costFunction = status.f_value / sqrt(status.frob_tns);

            als(grid_comm, tnsDims, R, mv, status);

            return status;
        }

        Status operator()(Tensor_     const &tnsX,
                          std::size_t const  R,
                          Options     const &options)
        {
            Status           status(options);
            Member_Variables mv(R, status.options.proc_per_mode);

            // Communicator with cartesian topology
            CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);

            // Functions that create layer and fiber grids.
            create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
            create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);

            // extract dimensions from tensor
            Dimensions const &tnsDims = tnsX.dimensions();
            // produce estimate factors using uniform distribution with entries in [0,1].
            makeFactors(tnsDims, status.options.constraints, R, mv.factors);

            compute_sub_dimensions(tnsDims, status, R, mv);
            // Normalize each layer_factor, compute status.frob_tns and status.f_value
            // Normalize(R, subTns_extents, layer_factors_dimTree);
            // Each processor takes a subtensor from tnsX
            mv.subTns = tnsX.slice(mv.subTns_offsets, mv.subTns_extents);
            cost_function_init(grid_comm, R, mv, status);
            status.rel_costFunction = status.f_value / sqrt(status.frob_tns);
            switch ( status.options.method )
            {
                case Method::als:
                {
                    als(grid_comm, tnsDims, R, mv, status);
                    break;
                }
                case Method::rnd:
                break;
```

```
                case Method::bc:
                break;
                default:
                break;
            }
            return status;
        }

        template <typename MatrixArray_>
        Status operator()(Tensor_      const &tnsX,
                          std::size_t  const  R,
                          MatrixArray_ const &factorsInit)
        {
            Options          options = MakeOptions<Tensor_>(execution::openmpi_policy());
            Status           status(options);
            Member_Variables mv(R, status.options.proc_per_mode);

            // Communicator with cartesian topology
            CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);

            // Functions that create layer and fiber grids.
            create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
            create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);

            // extract dimensions from tensor
            Dimensions const &tnsDims = tnsX.dimensions();
            // Copy factorsInit data to status.factors - FactorDimTree data struct
            fillDimTreeFactors(factorsInit, status.options.constraints, mv.factors);

             compute_sub_dimensions(tnsDims, status, R, mv);
            // Normalize each layer_factor, compute status.frob_tns and status.f_value
            // Normalize(R, subTns_extents, layer_factors_dimTree);
            // Each processor takes a subtensor from tnsX
            mv.subTns = tnsX.slice(mv.subTns_offsets, mv.subTns_extents);
            cost_function_init(grid_comm, R, mv, status);
            status.rel_costFunction = status.f_value / sqrt(status.frob_tns);

            als(grid_comm, tnsDims, R, mv, status);
            return status;
        }

        template <typename MatrixArray_>
        Status operator()(Tensor_      const &tnsX,
                          std::size_t  const  R,
                          Options      const &options,
                          MatrixArray_ const &factorsInit)
        {
            Status           status(options);
            Member_Variables mv(R, status.options.proc_per_mode);

            // Communicator with cartesian topology
            CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);

            // Functions that create layer and fiber grids.
            create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
            create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);

            // extract dimensions from tensor
            Dimensions const &tnsDims = tnsX.dimensions();
            // Copy factorsInit data to status.factors - FactorDimTree data struct
            fillDimTreeFactors(factorsInit, status.options.constraints, mv.factors);

            compute_sub_dimensions(tnsDims, status, R, mv);
            // Normalize each layer_factor, compute status.frob_tns and status.f_value
            // Normalize(R, subTns_extents, layer_factors_dimTree);
            // Each processor takes a subtensor from tnsX
            mv.subTns = tnsX.slice(mv.subTns_offsets, mv.subTns_extents);
            cost_function_init(grid_comm, R, mv, status);
            status.rel_costFunction = status.f_value / sqrt(status.frob_tns);
            switch ( status.options.method )
            {
                case Method::als:
                {
                    als(grid_comm, tnsDims, R, mv, status);
                    break;
                }
                case Method::rnd:
                break;
                case Method::bc:
                break;
                default:
                break;
            }
            return status;
        }

        template <std::size_t TnsSize>
```

```
            Status operator()(std::array<int, TnsSize> const &tnsDims,
                              std::size_t              const  R,
                              std::string              const &path)
            {
                using   TensorType = Tensor<static_cast<int>(TnsSize)>;

                Options          options    = MakeOptions<TensorType>(execution::openmpi_policy());
                Status           status(options);
                Member_Variables mv(R, status.options.proc_per_mode);

                // Communicator with cartesian topology
                CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);

                // Functions that create layer and fiber grids.
                create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
                create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);

                // produce estimate factors using uniform distribution with entries in [0,1].
                makeFactors(tnsDims, status.options.constraints, R, mv.factors);

                compute_sub_dimensions(tnsDims, status, R, mv);
                // Normalize each layer_factor, compute status.frob_tns and status.f_value
                // Normalize(R, subTns_extents, layer_factors_dimTree);
                // Each processor takes a subtensor from tnsX
                mv.subTns.resize(mv.subTns_extents);
                readTensor( path, tnsDims, mv.subTns_extents, mv.subTns_offsets, mv.subTns );
                cost_function_init(grid_comm, R, mv, status);
                status.rel_costFunction = status.f_value / sqrt(status.frob_tns);

                als(grid_comm, tnsDims, R, mv, status);
                return status;
            }

            template <std::size_t TnsSize>
            Status operator()(std::array<int, TnsSize> const &tnsDims,
                              std::size_t              const  R,
                              std::string              const &path,
                              Options                  const &options)
            {
                Status           status(options);
                Member_Variables mv(R, status.options.proc_per_mode);

                // Communicator with cartesian topology
                CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);

                // Functions that create layer and fiber grids.
                create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
                create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);

                // produce estimate factors using uniform distribution with entries in [0,1].
                makeFactors(tnsDims, status.options.constraints, R, mv.factors);

                compute_sub_dimensions(tnsDims, status, R, mv);
                // Normalize each layer_factor, compute status.frob_tns and status.f_value
                // Normalize(R, subTns_extents, layer_factors_dimTree);
                // Each processor takes a subtensor from tnsX
                 mv.subTns.resize(mv.subTns_extents);
                 readTensor( path, tnsDims, mv.subTns_extents, mv.subTns_offsets, mv.subTns );
                 cost_function_init(grid_comm, R, mv, status);
                 status.rel_costFunction = status.f_value / sqrt(status.frob_tns);
                 switch ( status.options.method )
                 {
                     case Method::als:
                     {
                         als(grid_comm, tnsDims, R, mv, status);
                         break;
                     }
                     case Method::rnd:
                     break;
                     case Method::bc:
                     break;
                     default:
                     break;
                 }
                 return status;
             }

             template <std::size_t TnsSize>
             Status operator()(std::array<int, TnsSize>           const &tnsDims,
                               std::size_t                        const  R,
                               std::array<std::string, TnsSize+1> const &paths)
             {
                 using   TensorType = Tensor<static_cast<int>(TnsSize)>;

                 Options          options = MakeOptions<TensorType>(execution::openmpi_policy());
                 Status           status(options);
                 Member_Variables mv(R, status.options.proc_per_mode);
```

```

                    // Communicator with cartesian topology
                    CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);

                    // Functions that create layer and fiber grids.
                    create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
                    create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);

                    compute_sub_dimensions(tnsDims, status, R, paths, mv);
                    // Normalize each layer_factor, compute status.frob_tns and status.f_value
                    // Normalize(R, subTns_extents, layer_factors_dimTree);
                    // Each processor takes a subtensor from tnsX
                    mv.subTns.resize(mv.subTns_extents);
                    readTensor( paths[0], tnsDims, mv.subTns_extents, mv.subTns_offsets, mv.subTns );
                    cost_function_init(grid_comm, R, mv, status);
                    status.rel_costFunction = status.f_value / sqrt(status.frob_tns);
                    als(grid_comm, tnsDims, R, mv, status);

                    return status;
                }

                template <std::size_t TnsSize>
                Status operator()(std::array<int, TnsSize>           const &tnsDims,
                                  std::size_t                        const  R,
                                  std::array<std::string, TnsSize+1> const &paths,
                                  Options                            const &options)
                {
                    Status           status(options);
                    Member_Variables mv(R, status.options.proc_per_mode);

                    // Communicator with cartesian topology
                    CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);

                    // Functions that create layer and fiber grids.
                    create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
                    create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);

                    compute_sub_dimensions(tnsDims, status, R, paths, mv);
                    // Normalize each layer_factor, compute status.frob_tns and status.f_value
                    // Normalize(R, subTns_extents, layer_factors_dimTree);
                    // Each processor takes a subtensor from tnsX
                    mv.subTns.resize(mv.subTns_extents);
                    readTensor( paths[0], tnsDims, mv.subTns_extents, mv.subTns_offsets, mv.subTns );
                    cost_function_init(grid_comm, R, mv, status);
                    status.rel_costFunction = status.f_value / sqrt(status.frob_tns);
                    switch ( status.options.method )
                    {
                        case Method::als:
                        {
                            als(grid_comm, tnsDims, R, mv, status);
                            break;
                        }
                        case Method::rnd:
                        break;
                        case Method::bc:
                        break;
                        default:
                        break;
                    }
                    return status;
                }

            };
      } // end namespace internal
    } // end namespace v1

} //end namespace partensor
```

8.11 CpdMpi.hpp File Reference

Classes

- struct CPD< Tensor_, execution::openmpi_policy >

8.11.1 Detailed Description

Implements the Canonical Polyadic Decomposition(cpd) using `MPI`. Make use of `spdlog` library in order to write output in a log file in `"../log"`.

8.12 CpdMpi.hpp

Go to the documentation of this file.

```
#ifndef DOXYGEN_SHOULD_SKIP_THIS
#endif // DOXYGEN_SHOULD_SKIP_THIS
/************************************************************************/
#if !defined(PARTENSOR_CPD_HPP)
#error "CPD_MPI can only included inside CPD"
#endif /* PARTENSOR_CPD_HPP */

namespace partensor
{

  inline namespace v1 {

    namespace internal {
      template<typename Tensor_>
      struct CPD<Tensor_,execution::openmpi_policy> : public CPD_Base<Tensor_>
      {
        using          CPD_Base<Tensor_>::TnsSize;
        using          CPD_Base<Tensor_>::lastFactor;
        using typename CPD_Base<Tensor_>::Dimensions;
        using typename CPD_Base<Tensor_>::MatrixArray;
        using typename CPD_Base<Tensor_>::DataType;

        using IntArray         = typename TensorTraits<Tensor_>::IntArray;
        using CartCommunicator = partensor::cartesian_communicator; // From ParallelWrapper.hpp
        using CartCommVector   = std::vector<CartCommunicator>;
        using IntVector        = std::vector<int>;
        using Int2DVector      = std::vector <std::vector<int»;

        using Options = partensor::Options<Tensor_,execution::openmpi_policy,DefaultValues>;
        using Status  = partensor::Status<Tensor_,execution::openmpi_policy,DefaultValues>;

        // Variables that will be used in cpd implementations.
        struct Member_Variables
        {
          MPI_Communicator &world = Partensor()->MpiCommunicator(); // MPI_COMM_WORLD

          double          local_f_value;
          int             RxR;
          int             world_size;

          Int2DVector     displs_subTns;       // skipping dimension "rows" for each subtensor
          Int2DVector     displs_subTns_R;     // skipping dimension "rows" for each subtensor times R ( for MPI communication purposes )
          Int2DVector     subTnsDims;          // dimensions of subtensor
          Int2DVector     subTnsDims_R;        // dimensions of subtensor times R ( for MPI communication purposes )
          Int2DVector     displs_local_update; // displacement in the local factor for update rows

          Int2DVector     send_recv_counts;    // rows to be communicated after update times R

          CartCommVector  layer_comm;
          CartCommVector  fiber_comm;

          IntArray        layer_rank;
          IntArray        fiber_rank;
          IntArray        rows_for_update;
          IntArray        subTns_offsets;
          IntArray        subTns_extents;

          MatrixArray     proc_krao;
          MatrixArray     layer_factors;
          MatrixArray     layer_factors_T;
          MatrixArray     factors_T;
          MatrixArray     factor_T_factor;
          MatrixArray     local_mttkrp;
          MatrixArray     layer_mttkrp;
          MatrixArray     local_mttkrp_T;
          MatrixArray     layer_mttkrp_T;
          MatrixArray     subTns_mat;
          MatrixArray     local_factors;
          MatrixArray     local_factors_T;
          MatrixArray     norm_factors;
          MatrixArray     old_factors;
          MatrixArray     true_factors;

          Matrix          cwise_factor_product;
          Matrix          tnsX_mat_lastFactor_T;
          Matrix          temp_matrix;
          Matrix          nesterov_old_layer_factor;

          Tensor_         subTns;

```

```
      bool            all_orthogonal = true;
      int             weight_factor;

      /*
       * Calculates if the number of processors given from terminal
       * are equal to the processors in the implementation.
       *
       * @param procs [in] @c stl array with the number of processors per
       *                   dimension of the tensor.
       */
      void check_processor_avaliability(std::array<int, TnsSize> const &procs)
      {
        // MPI_Environment &env = Partensor()->MpiEnvironment();
        world_size = world.size();
        //  numprocs must be product of options.proc_per_mode
        if (std::accumulate(procs.begin(), procs.end(), 1,
                            std::multiplies<int>()) != world_size && world.rank() == 0) {
          Partensor()->Logger()->error("The product of the processors per mode must be equal to {}\n", world_size);
          // env.abort(-1);
        }
      }

      Member_Variables() = default;
      Member_Variables(int R, std::array<int, TnsSize> &procs) :  local_f_value(0.0),
                                                                  RxR(R*R),
                                                                  displs_subTns(TnsSize),
                                                                  displs_subTns_R(TnsSize),
                                                                  subTnsDims(TnsSize),
                                                                  subTnsDims_R(TnsSize),
                                                                  displs_local_update(TnsSize),
                                                                  send_recv_counts(TnsSize)
      {
        check_processor_avaliability(procs);
        layer_comm.reserve(TnsSize);
        fiber_comm.reserve(TnsSize);
      }

      Member_Variables(Member_Variables const &) = default;
      Member_Variables(Member_Variables      &&) = default;

      Member_Variables &operator=(Member_Variables const &) = default;
      Member_Variables &operator=(Member_Variables      &&) = default;
    };

    /*
     * In case option variable @c writeToFile is enabled then, before the end
     * of the algorithm writes the resulted factors in files, where their
     * paths are specified before compiling in @ options.final_factors_path.
     *
     * @param  st  [in] Struct where the returned values of @c Cpd are stored.
     */
    void writeFactorsToFile(Status const &st)
    {
      std::size_t size;
      for(std::size_t i=0; i<TnsSize; ++i)
      {
        size = st.factors[i].rows() * st.factors[i].cols();
        partensor::write(st.factors[i],
                         st.options.final_factors_paths[i],
                         size);
      }
    }

    /*
     * Compute the cost function value at the end of each outer iteration
     * based on the last factor.
     *
     * @param  grid_comm [in]     MPI communicator where the new cost function value
     *                            will be communicated and computed.
     * @param  mv        [in]     Struct where ALS variables are stored.
     * @param  st        [in,out] Struct where the returned values of @c Cpd are stored.
     *                            In this case the cost function value is updated.
     */
    void cost_function_init( CartCommunicator const &grid_comm,
                             Member_Variables       &mv,
                             Status                 &st )
    {
        mv.local_f_value =
            ((mv.proc_krao[lastFactor].transpose() * mv.tnsX_mat_lastFactor_T) * mv.layer_factors[lastFactor]).trace();
        all_reduce( grid_comm,
                    inplace(&mv.local_f_value),
                    1,
                    std::plus<double>() );

        Matrix cwiseFactor_prod = PartialCwiseProd(mv.factor_T_factor, lastFactor) *
```

```
mv.factor_T_factor[lastFactor];
            st.f_value =
                sqrt(st.frob_tns - 2 * mv.local_f_value + cwiseFactor_prod.trace());
        }

        /*
         * Compute the cost function value at the end of each outer iteration
         * based on the last factor.
         *
         * @param  grid_comm [in]     MPI communicator where the new cost function value
         *                            will be communicated and computed.
         * @param  mv        [in]     Struct where ALS variables are stored.
         * @param  st        [in,out] Struct where the returned values of @c Cpd are stored.
         *                            In this case the cost function value is updated.
         */
        void cost_function( CartCommunicator const &grid_comm,
                            Member_Variables       &mv,
                            Status                 &st )
        {
            mv.local_f_value = ((mv.layer_mttkrp_T[lastFactor] * mv.layer_factors[lastFactor]).trace());
            all_reduce( grid_comm,
                        inplace(&mv.local_f_value),
                        1,
                        std::plus<double>() );

            Matrix cwiseFactor_prod = PartialCwiseProd(mv.factor_T_factor, lastFactor) *
mv.factor_T_factor[lastFactor];
            st.f_value =
                sqrt(st.frob_tns - 2 * mv.local_f_value + cwiseFactor_prod.trace());
        }

        /*
         * Compute the cost function value at the end of each outer iteration
         * based on the last accelerated factor.
         *
         * @param  grid_comm         [in] MPI communicator where the new cost function value
         *                                will be communicated and computed.
         * @param  mv                [in] Struct where ALS variables are stored.
         * @param  st                [in] Struct where the returned values of @c Cpd are stored.
         *                                In this case the cost function value is updated.
         * @param  factors           [in] Accelerated factors.
         * @param  factors_T_factors [in] Gramian matrices of factors.
         *
         * @returns The cost function calculated with the accelerated factors.
         */
        double accel_cost_function(CartCommunicator const &grid_comm,
                                   Member_Variables const &mv,
                                   Status           const &st,
                                   MatrixArray      const &factors,
                                   MatrixArray      const &factors_T_factors)
        {
          double local_f_value =
              ((PartialKhatriRao(factors, lastFactor).transpose() * mv.tnsX_mat_lastFactor_T) *
factors[lastFactor]).trace();
          all_reduce( grid_comm,
                      inplace(&local_f_value),
                      1,
                      std::plus<double>() );
          Matrix cwiseFactor_prod = PartialCwiseProd(factors_T_factors, lastFactor) *
factors_T_factors[lastFactor];
          return sqrt(st.frob_tns - 2 * local_f_value + cwiseFactor_prod.trace());
        }

        /*
         * Make use of the dimensions and the number of processors per dimension
         * and then calculates the dimensions of the subtensor and subfactor for
         * each processor.
         *
         * @tparam Dimensions           Array type containing the length of Tensor's dimensions.
         *
         * @param  tnsDims    [in]      Tensor Dimensions. Each index contains the corresponding
         *                              factor's rows length.
         * @param  st         [in]      Struct where the returned values of @c Cpd are stored.
         * @param  R          [in]      The rank of decomposition.
         * @param  mv         [in,out]  Struct where ALS variables are stored.
         *                              Updates @c stl arrays with dimensions for subtensors and
         *                              subfactors.
         */
        template<typename Dimensions>
        void compute_sub_dimensions(Dimensions       const &tnsDims,
                                    Status           const &st,
                                    std::size_t      const  R,
                                    Member_Variables       &mv)
        {
          for (std::size_t i = 0; i < TnsSize; ++i)
          {
            mv.factor_T_factor[i].noalias() = st.factors[i].transpose() * st.factors[i];
```

```

          DisCount(mv.displs_subTns[i], mv.subTnsDims[i], st.options.proc_per_mode[i], tnsDims[i], 1);
          // for fiber communication and Gatherv
          DisCount(mv.displs_subTns_R[i], mv.subTnsDims_R[i], st.options.proc_per_mode[i], tnsDims[i], static_cast<int>(R));
          // information per layer
          DisCount(mv.displs_local_update[i], mv.send_recv_counts[i], mv.world_size / st.options.proc_per_mode[i],
                                                      mv.subTnsDims[i][mv.fiber_rank[i]], static_cast<int>(R));

          mv.rows_for_update[i] = mv.send_recv_counts[i][mv.layer_rank[i]] / static_cast<int>(R);
          mv.subTns_offsets[i]  = mv.displs_subTns[i][mv.fiber_rank[i]];
          mv.subTns_extents[i]  = mv.subTnsDims[i][mv.fiber_rank[i]];
        }
      }

      /*
       * Based on each factor's constraint, a different
       * update function is used at every outer iteration.
       *
       * Computes also factor^T * factor at the end.
       *
       * @param  idx [in]     Factor to be updated.
       * @param  R   [in]     The rank of decomposition.
       * @param  st  [in]     Struct where the returned values of @c Cpd are stored.
       *                      Here constraints and options variables are needed.
       * @param  mv  [in,out] Struct where ALS variables are stored.
       *                      Updates the factors of each layer.
       */
      void update_factor(int              const  idx,
                         std::size_t      const  R,
                         Status           const &st,
                         Member_Variables       &mv  )
      {
        switch ( st.options.constraints[idx] )
        {
          case Constraint::unconstrained:
          {
            v2::reduce_scatter( mv.layer_comm[idx],
                                mv.layer_mttkrp_T[idx],
                                mv.send_recv_counts[idx][0],
                                mv.local_mttkrp_T[idx] );

            mv.local_mttkrp[idx] = mv.local_mttkrp_T[idx].transpose();
            if (mv.rows_for_update[idx] != 0)
              mv.local_factors[idx].noalias() = mv.local_mttkrp[idx] * mv.cwise_factor_product.inverse();
            break;
          }
          case Constraint::nonnegativity:
          {
            v2::reduce_scatter( mv.layer_comm[idx],
                                mv.layer_mttkrp_T[idx],
                                mv.send_recv_counts[idx][0],
                                mv.local_mttkrp_T[idx] );

            mv.local_mttkrp[idx]         = mv.local_mttkrp_T[idx].transpose();
            mv.nesterov_old_layer_factor = mv.layer_factors[idx];
            if (mv.rows_for_update[idx] != 0)
            {
              NesterovMNLS(mv.cwise_factor_product, mv.local_mttkrp[idx], st.options.nesterov_delta_1,

                               st.options.nesterov_delta_2, mv.local_factors[idx]);
            }
            break;
          }
          case Constraint::orthogonality:
          {
            all_reduce( mv.layer_comm[idx],
                        inplace(mv.layer_mttkrp[idx].data()),
                        mv.subTnsDims_R[idx][mv.fiber_rank[idx]],
                        std::plus<double>() );

            if (mv.rows_for_update[idx] != 0)
            {
              mv.local_mttkrp[idx]     = mv.layer_mttkrp[idx].block(mv.displs_local_update[idx][mv.layer_rank[idx]] / static_cast<int>(R), 0,
                                                                    mv.rows_for_update[idx], static_cast<int>(R));
              mv.temp_matrix.noalias() = mv.layer_mttkrp[idx].transpose() * mv.layer_mttkrp[idx];
            }
            all_reduce( mv.fiber_comm[idx],
                        inplace(mv.temp_matrix.data()),
                        mv.RxR,
                        std::plus<double>() );

```

```
            Eigen::SelfAdjointEigenSolver<Matrix> eigensolver(mv.temp_matrix);
            mv.temp_matrix.noalias() = (eigensolver.eigenvectors())
                                       *
(eigensolver.eigenvalues().cwiseInverse().cwiseSqrt().asDiagonal())
                                       * (eigensolver.eigenvectors().transpose());

            if(mv.rows_for_update[idx] != 0)
              mv.local_factors[idx].noalias() = mv.local_mttkrp[idx] * mv.temp_matrix;
            break;
          }
          case Constraint::sparsity:
            break;
          default: // in case of Constraint::constant
            break;
        } // end of constraints switch

        if (st.options.constraints[idx] != Constraint::constant)
        {
          mv.local_factors_T[idx] = mv.local_factors[idx].transpose();
          v2::all_gatherv( mv.layer_comm[idx],
                           mv.local_factors_T[idx],
                           mv.send_recv_counts[idx][mv.layer_rank[idx]],
                           mv.send_recv_counts[idx][0],
                           mv.displs_local_update[idx][0],
                           mv.layer_factors_T[idx] );

          mv.layer_factors[idx]              = mv.layer_factors_T[idx].transpose();
          mv.factor_T_factor[idx].noalias() = mv.layer_factors_T[idx] * mv.layer_factors[idx];
        }

        all_reduce( mv.fiber_comm[idx],
                    inplace(mv.factor_T_factor[idx].data()),
                    mv.RxR,
                    std::plus<double>() );

        if(st.options.constraints[idx] == Constraint::nonnegativity)
        {
          if ((mv.factor_T_factor[idx].diagonal()).minCoeff()==0)
          {
            mv.layer_factors[idx] = 0.9 * mv.layer_factors[idx] + 0.1 * mv.nesterov_old_layer_factor;
            all_reduce( mv.fiber_comm[idx],
                        inplace(mv.factor_T_factor[idx].data()),
                        mv.RxR,
                        std::plus<double>() );
          }
        }
      }

      /*
       * At the end of the algorithm processor 0
       * collects each part of the factor that each
       * processor holds and return them in status.factors.
       *
       * @tparam Dimensions      Array type containing the Tensor dimensions.
       *
       * @param  tnsDims [in]     Tensor Dimensions. Each index contains the corresponding
       *                          factor's rows length.
       * @param  R       [in]     The rank of decomposition.
       * @param  mv      [in]     Struct where ALS variables are stored.
       *                          Use variables to compute result factors by gathering each
       *                          part of the factor from processors.
       * @param  st      [in,out] Struct where the returned values of @c Cpd are stored.
       *                          Stores the resulted factors.
       */
      template<typename Dimensions>
      void gather_final_factors(Dimensions       const &tnsDims,
                                std::size_t      const  R,
                                Member_Variables       &mv,
                                Status                 &st)
      {
        for(std::size_t i=0; i<TnsSize; ++i)
          mv.layer_factors_T[i] = mv.layer_factors[i].transpose();

        for(std::size_t i=0; i<TnsSize; ++i)
        {
          mv.temp_matrix.resize(static_cast<int>(R), tnsDims[i]);
          // Gatherv from all processors to processor with rank 0 the final factors
          v2::gatherv( mv.fiber_comm[i],
                       mv.layer_factors_T[i],
                       mv.subTnsDims_R[i][mv.fiber_rank[i]],
                       mv.subTnsDims_R[i][0],
                       mv.displs_subTns_R[i][0],
                       0,
                       mv.temp_matrix );

          st.factors[i] = mv.temp_matrix.transpose();
        }
```

```
    }

    /*
     * @brief Line Search Acceleration
     *
     * Performs an acceleration step in the updated factors, and keeps the accelerated factors when
     * the step succeeds. Otherwise, the acceleration step is ignored.
     * Line Search Acceleration reduces the number outer iterations in the ALS algorithm.
     *
     * @note This implementation ONLY, if factors are of @c Matrix type.
     *
     * @param  grid_comm [in]     MPI communicator where the new cost function value
     *                            will be communicated and computed.
     * @param  mv        [in,out] Struct where ALS variables are stored.
     *                            In case the acceration is successful layer factor^T * factor
     *                            and layer factor variables are updated.
     * @param  st        [in,out] Struct where the returned values of @c Cpd are stored.
     *                            If the acceleration succeeds updates cost function value.
     *
     */
    void line_search_accel(CartCommunicator const &grid_comm,
                           Member_Variables       &mv,
                           Status                 &st)
    {
      double        f_accel    = 0.0; // Objective Value after the acceleration step
      double const accel_step = pow(st.ao_iter+1,(1.0/(st.options.accel_coeff)));

      MatrixArray   accel_factors;
      MatrixArray   accel_gramians;

      for(std::size_t i=0; i<TnsSize; ++i)
      {
        accel_factors[i]  = mv.old_factors[i] + accel_step * (mv.layer_factors[i] - mv.old_factors[i]);
        accel_gramians[i] = accel_factors[i].transpose() * accel_factors[i];
        all_reduce( mv.fiber_comm[i],
                    inplace(accel_gramians[i].data()),
                    mv.RxR,
                    std::plus<double>() );
      }

      f_accel = accel_cost_function(grid_comm, mv, st, accel_factors, accel_gramians);
      if (st.f_value > f_accel)
      {
        mv.layer_factors   = accel_factors;
        mv.factor_T_factor = accel_gramians;
        st.f_value         = f_accel;
        if(grid_comm.rank() == 0)
          Partensor()->Logger()->info("Acceleration Step SUCCEEDED! at iter: {}", st.ao_iter);
      }
      else
        st.options.accel_fail++;

      if (st.options.accel_fail==5)
      {
        st.options.accel_fail=0;
        st.options.accel_coeff++;
      }
    }

    /*
     * Parallel implementation of als method with MPI.
     *
     * @tparam Dimensions         Array type containing the Tensor dimensions.
     *
     * @param  grid_comm [in]     The communication grid, where the processors
     *                            communicate their cost function.
     * @param  tnsDims   [in]     Tensor Dimensions. Each index contains the corresponding
     *                            factor's rows length.
     * @param  R         [in]     The rank of decomposition.
     * @param  mv        [in]     Struct where ALS variables are stored and being updated
     *                            until a termination condition is true.
     * @param  status    [in,out] Struct where the returned values of @c Cpd are stored.
     */
    template<typename Dimensions>
    void als(CartCommunicator const &grid_comm,
             Dimensions       const &tnsDims,
             std::size_t      const  R,
             Member_Variables       &mv,
             Status                 &status)
    {
      for (std::size_t i = 0; i < TnsSize; ++i)
      {
        mv.proc_krao[i]   = PartialKhatriRao(mv.layer_factors, i);
        mv.subTns_mat[i] = Matricization(mv.subTns, i);

        mv.factor_T_factor[i].noalias() = mv.layer_factors[i].transpose() * mv.layer_factors[i];
```

```
            all_reduce( mv.fiber_comm[i],
                        inplace(mv.factor_T_factor[i].data()),
                        mv.RxR,
                        std::plus<double>() );

            mv.local_mttkrp_T[i].resize(R, mv.rows_for_update[i]);
            mv.layer_factors_T[i].resize(R, mv.subTnsDims[i][mv.fiber_rank[i]]);
          }

          mv.tnsX_mat_lastFactor_T = mv.subTns_mat[lastFactor].transpose();
          if(status.options.normalization)
          {
            choose_normilization_factor(status, mv.all_orthogonal, mv.weight_factor);
          }

          all_reduce( grid_comm,
                      square_norm(mv.subTns),
                      status.frob_tns,
                      std::plus<double>());
          cost_function_init( grid_comm, mv, status );
          status.rel_costFunction = status.f_value / sqrt(status.frob_tns);

          // Wait for all processors to reach here
          grid_comm.barrier();

          // ---- Loop until ALS converges ----
          while(1)
          {
              status.ao_iter++;
              if (!grid_comm.rank())
                  Partensor()->Logger()->info("iter: {} -- fvalue: {} -- relative_costFunction: {}",
      status.ao_iter,
                                                    status.f_value, status.rel_costFunction);

              for (std::size_t i = 0; i < TnsSize; i++)
              {
                mttkrp(mv.layer_factors, mv.subTns_mat[i], i, mv.proc_krao[i], mv.layer_mttkrp[i]);
                mv.cwise_factor_product = PartialCwiseProd(mv.factor_T_factor, i);
                mv.layer_mttkrp_T[i]     = mv.layer_mttkrp[i].transpose();
                mv.local_factors[i]      =
      mv.layer_factors[i].block(mv.displs_local_update[i][mv.layer_rank[i]] / static_cast<int>(R), 0,
                                                                          mv.rows_for_update[i],
                static_cast<int>(R));

                update_factor(i, R, status, mv);
              }

              // ---- Cost function Computation ----
              cost_function(grid_comm, mv, status);
              status.rel_costFunction = status.f_value / sqrt(status.frob_tns);
              if(status.options.normalization && !mv.all_orthogonal)
                Normalize(mv.weight_factor, R, mv.factor_T_factor, mv.layer_factors);

              // ---- Terminating condition ----
              if (status.rel_costFunction < status.options.threshold_error || status.ao_iter >=
      status.options.max_iter)
              {
                  gather_final_factors(tnsDims, R, mv, status);
                  if(grid_comm.rank() == 0)
                  {
                    Partensor()->Logger()->info("Processor 0 collected all {} factors.\n", TnsSize);
                    if(status.options.writeToFile)
                      writeFactorsToFile(status);
                  }
                  break;
              }

              if (status.options.acceleration)
              {
                mv.norm_factors = mv.layer_factors;
                // ---- Acceleration Step ----
                if (status.ao_iter > 1)
                  line_search_accel(grid_comm, mv, status);

                mv.old_factors = mv.norm_factors;
              }

          } // end of outer while loop
        }

        /*
         * Parallel implementation of als method with MPI.
         *
         * @tparam Dimensions         Array type containing the Tensor dimensions.
         *
         * @param  grid_comm [in]     The communication grid, where the processors
         *                            communicate their cost function.
```

```
         * @param  tnsDims   [in]     Tensor Dimensions. Each index contains the corresponding
         *                            factor's rows length.
         * @param  R         [in]     The rank of decomposition.
         * @param  mv        [in]     Struct where ALS variables are stored and being updated
         *                            until a termination condition is true.
         * @param  status    [in,out] Struct where the returned values of @c Cpd are stored.
         */
        template<typename Dimensions>
        void als_true_factors( CartCommunicator const &grid_comm,
                               Dimensions       const &tnsDims,
                               std::size_t      const  R,
                               Member_Variables       &mv,
                               Status                 &status )
        {
          for (std::size_t i = 0; i < TnsSize; ++i)
          {
            mv.proc_krao[i]   = PartialKhatriRao(mv.layer_factors, i);
            mv.subTns_mat[i] = generateTensor(i, mv.true_factors);

            mv.factor_T_factor[i].noalias() = mv.layer_factors[i].transpose() * mv.layer_factors[i];
            all_reduce( mv.fiber_comm[i],
                        inplace(mv.factor_T_factor[i].data()),
                        mv.RxR,
                        std::plus<double>() );

            mv.local_mttkrp_T[i].resize(R, mv.rows_for_update[i]);
            mv.layer_factors_T[i].resize(R, mv.subTnsDims[i][mv.fiber_rank[i]]);
          }

          mv.tnsX_mat_lastFactor_T = mv.subTns_mat[lastFactor].transpose();
          if(status.options.normalization)
          {
            choose_normilization_factor(status, mv.all_orthogonal, mv.weight_factor);
          }

          all_reduce( grid_comm,
                      (mv.subTns_mat[lastFactor]).squaredNorm(),
                      status.frob_tns,
                      std::plus<double>());
          cost_function_init( grid_comm, mv, status );
          status.rel_costFunction = status.f_value / sqrt(status.frob_tns);

          // Wait for all processors to reach here
          grid_comm.barrier();

          // ---- Loop until ALS converges ----
          while(1)
          {
              status.ao_iter++;
              if (!grid_comm.rank())
                  Partensor()->Logger()->info("iter: {} -- fvalue: {} -- relative_costFunction: {}", status.ao_iter,
                                              status.f_value, status.rel_costFunction);

              for (std::size_t i = 0; i < TnsSize; i++)
              {
                mv.proc_krao[i]               = PartialKhatriRao(mv.layer_factors, i);
                mv.cwise_factor_product       = PartialCwiseProd(mv.factor_T_factor, i);
                mv.layer_mttkrp[i].noalias() = mv.subTns_mat[i] * mv.proc_krao[i];
                mv.layer_mttkrp_T[i]          = mv.layer_mttkrp[i].transpose();
                mv.local_factors[i]           = mv.layer_factors[i].block(mv.displs_local_update[i][mv.layer_rank[i]] / static_cast<int>(R), 0,
                                                                            mv.rows_for_update[i], static_cast<int>(R));

                update_factor(i, R, status, mv);
                all_reduce( mv.fiber_comm[i],
                            inplace(mv.factor_T_factor[i].data()),
                            mv.RxR,
                            std::plus<double>() );
              }

              // ---- Cost function Computation ----
              cost_function(grid_comm, mv, status);
              status.rel_costFunction = status.f_value / sqrt(status.frob_tns);
              if(status.options.normalization && !mv.all_orthogonal)
                Normalize(mv.weight_factor, R, mv.factor_T_factor, mv.layer_factors);

              // ---- Terminating condition ----
              if (status.rel_costFunction < status.options.threshold_error || status.ao_iter >= status.options.max_iter)
              {
                  gather_final_factors(tnsDims, R, mv, status);
                  if(grid_comm.rank() == 0)
                  {
                    Partensor()->Logger()->info("Processor 0 collected all {} factors.\n", TnsSize);
                    if(status.options.writeToFile)
```

```
                    writeFactorsToFile(status);
                }
                break;
            }

            if (status.options.acceleration)
            {
              mv.norm_factors = mv.layer_factors;
              // ---- Acceleration Step ----
              if (status.ao_iter > 1)
                line_search_accel(grid_comm, mv, status);

              mv.old_factors = mv.norm_factors;
            }

        } // end of outer while loop
      }

      Status operator()(Tensor_     const &tnsX,
                        std::size_t const  R)
      {
        Options           options = MakeOptions<Tensor_>(execution::openmpi_policy());
        Status            status(options);
        Member_Variables mv(R, status.options.proc_per_mode);

        // Communicator with cartesian topology
        CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);

        // Functions that create layer and fiber grids.
        create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
        create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);

        // extract dimensions from tensor
        Dimensions const &tnsDims = tnsX.dimensions();
        // produce estimate factors using uniform distribution with entries in [0,1].
        makeFactors(tnsDims, status.options.constraints, R, status.factors);

        compute_sub_dimensions(tnsDims, status, R, mv);
        // Normalize each layer_factor, compute status.frob_tns and status.f_value
        // Normalize(R, factor_T_factor, status.factors);
        // After factor normalization scatter to each processor a part of each factor.
        for (std::size_t i = 0; i < TnsSize; ++i)
        {
          mv.layer_factors[i] = status.factors[i].block(mv.displs_subTns[i][mv.fiber_rank[i]], 0,
                                                        mv.subTnsDims[i][mv.fiber_rank[i]],
static_cast<int>(R));
        }

        // Each processor takes a subtensor from tnsX
        mv.subTns = tnsX.slice(mv.subTns_offsets, mv.subTns_extents);
        als(grid_comm, tnsDims, R, mv, status);

        return status;
      }

      Status operator()(Tensor_     const &tnsX,
                        std::size_t const R,
                        Options     const &options)
      {
        Status            status(options);
        Member_Variables mv(R, status.options.proc_per_mode);

        // Communicator with cartesian topology
        CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);

        // Functions that create layer and fiber grids.
        create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
        create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);

        // extract dimensions from tensor
        Dimensions const &tnsDims = tnsX.dimensions();
        // produce estimate factors using uniform distribution with entries in [0,1].
        makeFactors(tnsDims, status.options.constraints, R, status.factors);

        compute_sub_dimensions(tnsDims, status, R, mv);
        // Normalize each layer_factor, compute status.frob_tns and status.f_value
        // Normalize(R, factor_T_factor, status.factors);
        // After factor normalization scatter to each processor a part of each factor.
        for (std::size_t i = 0; i < TnsSize; ++i)
        {
          mv.layer_factors[i] = status.factors[i].block(mv.displs_subTns[i][mv.fiber_rank[i]], 0,
                                                        mv.subTnsDims[i][mv.fiber_rank[i]],
static_cast<int>(R));
        }

        // Each processor takes a subtensor from tnsX
        mv.subTns = tnsX.slice(mv.subTns_offsets, mv.subTns_extents);
```

```
          switch ( status.options.method )
          {
            case Method::als:
            {
              als(grid_comm, tnsDims, R, mv, status);
              break;
            }
            case Method::rnd:
              break;
            case Method::bc:
              break;
            default:
              break;
          }

          return status;
        }

        Status operator()(Tensor_     const &tnsX,
                          std::size_t const  R,
                          MatrixArray const &factorsInit)
        {
          Options            options = MakeOptions<Tensor_>(execution::openmpi_policy());
          Status             status(options);
          Member_Variables mv(R, status.options.proc_per_mode);

          // Communicator with cartesian topology
          CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);

          // Functions that create layer and fiber grids.
          create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
          create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);

          // extract dimensions from tensor
          Dimensions const &tnsDims = tnsX.dimensions();
          status.factors            = factorsInit;

          compute_sub_dimensions(tnsDims, status, R, mv);
          // Normalize each layer_factor, compute status.frob_tns and status.f_value
          // Normalize(R, factor_T_factor, status.factors);
          // After factor normalization scatter to each processor a part of each factor.
          for (std::size_t i = 0; i < TnsSize; ++i)
          {
            mv.layer_factors[i] = status.factors[i].block(mv.displs_subTns[i][mv.fiber_rank[i]], 0,
                                                           mv.subTnsDims[i][mv.fiber_rank[i]],
      static_cast<int>(R));
          }

          // Each processor takes a subtensor from tnsX
          mv.subTns = tnsX.slice(mv.subTns_offsets, mv.subTns_extents);

          als(grid_comm, tnsDims, R, mv, status);

          return status;
        }

        Status operator()(Tensor_     const &tnsX,
                          std::size_t const  R,
                          Options     const &options,
                          MatrixArray const &factorsInit)
        {
          Status             status(options);
          Member_Variables mv(R, status.options.proc_per_mode);

          // Communicator with cartesian topology
          CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);

          // Functions that create layer and fiber grids.
          create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
          create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);

          // extract dimensions from tensor
          Dimensions const &tnsDims = tnsX.dimensions();
          status.factors            = factorsInit;

          compute_sub_dimensions(tnsDims, status, R, mv);
          // Normalize each layer_factor, compute status.frob_tns and status.f_value
          // Normalize(R, factor_T_factor, status.factors);
          // After factor normalization scatter to each processor a part of each factor.
          for (std::size_t i = 0; i < TnsSize; ++i)
          {
            mv.layer_factors[i] = status.factors[i].block(mv.displs_subTns[i][mv.fiber_rank[i]], 0,
                                                           mv.subTnsDims[i][mv.fiber_rank[i]],
      static_cast<int>(R));
          }

```

```
      // Each processor takes a subtensor from tnsX
      mv.subTns.resize(mv.subTns_extents);
      mv.subTns = tnsX.slice(mv.subTns_offsets, mv.subTns_extents);

      switch ( status.options.method )
      {
        case Method::als:
        {
          als(grid_comm, tnsDims, R, mv, status);
          break;
        }
        case Method::rnd:
          break;
        case Method::bc:
          break;
        default:
          break;
      }

      return status;
    }

    template <std::size_t TnsSize>
    Status operator()(std::array<int, TnsSize> const &tnsDims,
                      std::size_t              const  R,
                      std::string              const &path)
    {
      Options          options = MakeOptions<Tensor_>(execution::openmpi_policy());
      Status           status(options);
      Member_Variables mv(R, status.options.proc_per_mode);

      // Communicator with cartesian topology
      CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);

      // Functions that create layer and fiber grids.
      create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
      create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);

      // produce estimate factors using uniform distribution with entries in [0,1].
      makeFactors(tnsDims, status.options.constraints, R, status.factors);

      compute_sub_dimensions(tnsDims, status, R, mv);
      // Normalize each layer_factor, compute status.frob_tns and status.f_value
      // Normalize(R, factor_T_factor, status.factors);
      // After factor normalization scatter to each processor a part of each factor.
      for (std::size_t i = 0; i < TnsSize; ++i)
      {
        mv.layer_factors[i] = status.factors[i].block(mv.displs_subTns[i][mv.fiber_rank[i]], 0,
                                                      mv.subTnsDims[i][mv.fiber_rank[i]],
  static_cast<int>(R));
      }

      // Each processor takes a subtensor from tnsX
      mv.subTns.resize(mv.subTns_extents);
      readTensor( path, tnsDims, mv.subTns_extents, mv.subTns_offsets, mv.subTns );
      als(grid_comm, tnsDims, R, mv, status);

      return status;
    }

     template <std::size_t TnsSize>
     Status operator()(std::array<int, TnsSize> const &tnsDims,
                       std::size_t              const  R,
                       std::string              const &path,
                       Options                  const &options)
     {
       Status           status(options);
       Member_Variables mv(R, status.options.proc_per_mode);

       // Communicator with cartesian topology
       CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);

       // Functions that create layer and fiber grids.
       create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
       create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);

       // produce estimate factors using uniform distribution with entries in [0,1].
       makeFactors(tnsDims, status.options.constraints, R, status.factors);

       compute_sub_dimensions(tnsDims, status, R, mv);
       // Normalize each layer_factor, compute status.frob_tns and status.f_value
       // Normalize(R, factor_T_factor, status.factors);
       // After factor normalization scatter to each processor a part of each factor.
       for (std::size_t i = 0; i < TnsSize; ++i)
       {
         mv.layer_factors[i] = status.factors[i].block(mv.displs_subTns[i][mv.fiber_rank[i]], 0,
                                                       mv.subTnsDims[i][mv.fiber_rank[i]],
```

```
  static_cast<int>(R));
          }

          mv.subTns.resize(mv.subTns_extents);
          readTensor( path, tnsDims, mv.subTns_extents, mv.subTns_offsets, mv.subTns );

          switch ( status.options.method )
          {
            case Method::als:
            {
              als(grid_comm, tnsDims, R, mv, status);
              break;
            }
            case Method::rnd:
              break;
            case Method::bc:
              break;
            default:
              break;
          }

          return status;
        }

        template <std::size_t TnsSize>
        Status operator()(std::array<int, TnsSize>           const &tnsDims,
                          std::size_t                        const  R,
                          std::array<std::string, TnsSize+1> const &paths)
        {
          Options           options = MakeOptions<Tensor_>(execution::openmpi_policy());
          Status            status(options);
          Member_Variables mv(R, status.options.proc_per_mode);

          // Communicator with cartesian topology
          CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);

          // Functions that create layer and fiber grids.
          create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
          create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);

          compute_sub_dimensions(tnsDims, status, R, mv);
          // Read initialized factors from files
          for (std::size_t i = 0; i < TnsSize; ++i)
          {
            mv.layer_factors[i] = Matrix(mv.subTnsDims[i][mv.fiber_rank[i]],static_cast<int>(R));

            read( paths[i+1],
                  mv.subTnsDims[i][mv.fiber_rank[i]]*static_cast<int>(R),
                  mv.displs_subTns_R[i][mv.fiber_rank[i]],
                  mv.layer_factors[i] );
          }

          // Normalize each layer_factor, compute status.frob_tns and status.f_value
          // Normalize(R, factor_T_factor, status.factors);
          // Each processor takes a subtensor from tnsX
          mv.subTns.resize(mv.subTns_extents);
          readTensor( paths[0], tnsDims, mv.subTns_extents, mv.subTns_offsets, mv.subTns );
          als(grid_comm, tnsDims, R, mv, status);

          return status;
        }

        template <std::size_t TnsSize>
        Status operator()(std::array<int, TnsSize>           const &tnsDims,
                          std::size_t                        const  R,
                          std::array<std::string, TnsSize+1> const &paths,
                          Options                            const &options)
        {
          Status            status(options);
          Member_Variables mv(R, status.options.proc_per_mode);

          // Communicator with cartesian topology
          CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);

          // Functions that create layer and fiber grids.
          create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
          create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);

          compute_sub_dimensions(tnsDims, status, R, mv);
          // Read initialized factors from files
          for (std::size_t i = 0; i < TnsSize; ++i)
          {
            mv.layer_factors[i] = Matrix(mv.subTnsDims[i][mv.fiber_rank[i]],static_cast<int>(R));

            read( paths[i+1],
                  mv.subTnsDims[i][mv.fiber_rank[i]]*static_cast<int>(R),
                  mv.displs_subTns_R[i][mv.fiber_rank[i]],
```

```
                mv.layer_factors[i] );
        }

        // Normalize each layer_factor, compute status.frob_tns and status.f_value
        // Normalize(R, factor_T_factor, status.factors);
        // Each processor takes a subtensor from tnsX
        mv.subTns.resize(mv.subTns_extents);
        readTensor( paths[0], tnsDims, mv.subTns_extents, mv.subTns_offsets, mv.subTns );

        switch ( status.options.method )
        {
          case Method::als:
          {
            als(grid_comm, tnsDims, R, mv, status);
            break;
          }
          case Method::rnd:
            break;
          case Method::bc:
            break;
          default:
            break;
          }

        return status;
      }

      template <std::size_t TnsSize>
      Status operator()(std::array<int, TnsSize>          const &tnsDims,
                        std::size_t                       const  R,
                        std::array<std::string, TnsSize> const &true_paths,
                        std::array<std::string, TnsSize> const &init_paths,
                        Options                           const &options)
      {
        Status           status(options);
        Member_Variables mv(R, status.options.proc_per_mode);

        // Communicator with cartesian topology
        CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);

        // Functions that create layer and fiber grids.
        create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
        create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);

        compute_sub_dimensions(tnsDims, status, R, mv);
        // Read initialized factors from files
        for (std::size_t i = 0; i < TnsSize; ++i)
        {
          mv.layer_factors[i] = Matrix(mv.subTnsDims[i][mv.fiber_rank[i]],static_cast<int>(R));
          mv.true_factors[i]  = Matrix(mv.subTnsDims[i][mv.fiber_rank[i]],static_cast<int>(R));

          read( init_paths[i],
                mv.subTnsDims[i][mv.fiber_rank[i]]*static_cast<int>(R),
                mv.displs_subTns_R[i][mv.fiber_rank[i]],
                mv.layer_factors[i] );

          read( true_paths[i],
                mv.subTnsDims[i][mv.fiber_rank[i]]*static_cast<int>(R),
                mv.displs_subTns_R[i][mv.fiber_rank[i]],
                mv.true_factors[i] );
        }

        // Normalize each layer_factor, compute status.frob_tns and status.f_value
        // Normalize(R, factor_T_factor, status.factors);
        switch ( status.options.method )
        {
          case Method::als:
          {
            als_true_factors(grid_comm, tnsDims, R, mv, status);
            break;
          }
          case Method::rnd:
            break;
          case Method::bc:
            break;
          default:
            break;
        }

        return status;
      }

    };
  }  // namespace internal
 }    // namespace v1

} // end namespace partensor
```

8.13 CpdOpenMP.hpp File Reference

Classes

- struct CPD< Tensor_, execution::openmp_policy >

8.13.1 Detailed Description

Implements the Canonical Polyadic Decomposition(cpd) using Shared memory and `OpenMP`. Make use of `spdlog` library in order to write output in a log file in `"../log"`. In case of using parallelism with mpi, then the functions from `CpdMpi.hpp` will be called.

8.14 CpdOpenMP.hpp

Go to the documentation of this file.

```
#ifndef DOXYGEN_SHOULD_SKIP_THIS
#endif // DOXYGEN_SHOULD_SKIP_THIS
/**********************************************************************/
#if !defined(PARTENSOR_CPD_HPP)
#error "CPD_OPENMP can only included inside CPD"
#endif /* PARTENSOR_CPD_HPP */

namespace partensor {

  inline namespace v1 {

    namespace internal {
        template <typename Tensor_>
        struct CPD<Tensor_,execution::openmp_policy> : public CPD_Base<Tensor_>
        {
            using          CPD_Base<Tensor_>::TnsSize;
            using          CPD_Base<Tensor_>::lastFactor;
            using typename CPD_Base<Tensor_>::Dimensions;
            using typename CPD_Base<Tensor_>::MatrixArray;
            using typename CPD_Base<Tensor_>::DataType;

            using Options = partensor::Options<Tensor_,execution::openmp_policy,DefaultValues>;
            using Status  = partensor::Status<Tensor_,execution::openmp_policy,DefaultValues>;

            // Variables that will be used in cpd implementations.
            struct Member_Variables {
                MatrixArray  krao;
                MatrixArray  factor_T_factor;
                MatrixArray  mttkrp;
                MatrixArray  tns_mat;
                MatrixArray  norm_factors;
                MatrixArray  old_factors;
                MatrixArray  true_factors;

                Matrix       cwise_factor_product;
                Matrix       temp_matrix;

                Tensor_      tnsX;

                Dimensions   tnsDims;

                bool         all_orthogonal = true;
                int          weight_factor;

                Member_Variables() = default;
                Member_Variables(Member_Variables const &) = default;
                Member_Variables(Member_Variables      &&) = default;

                Member_Variables &operator=(Member_Variables const &) = default;
                Member_Variables &operator=(Member_Variables      &&) = default;
            };

            /*
            * In case option variable @c writeToFile is enabled, then, before the end
            * of the algorithm, it writes the resulted factors in files, whose
            * paths are specified before compiling in @ options.final_factors_path.
            *
```

```
        * @param  st  [in] Struct where the returned values of @c Cpd are stored.
        */
        void writeFactorsToFile(Status const &st)
        {
            std::size_t size;
            for(std::size_t i=0; i<TnsSize; ++i)
            {
                size = st.factors[i].rows() * st.factors[i].cols();
                partensor::write(st.factors[i],
                                st.options.final_factors_paths[i],
                                size);
            }
        }

        /*
        * Compute the cost function value based on the initial factors.
        *
        * @param  mv  [in]     Struct where ALS variables are stored.
        * @param  st  [in,out] Struct where the returned values of @c Cpd are stored.
        *                      In this case the cost function value is updated.
        */
        void cost_function_init(Member_Variables const &mv,
                                Status                 &st)
        {
            st.f_value = sqrt( ( mv.tns_mat[lastFactor] - st.factors[lastFactor] * PartialKhatriRao(st.factors, lastFactor).transpose() ).squaredNorm() );
        }

        /*
        * Compute the cost function value at the end of each outer iteration
        * based on the last factor.
        *
        * @param  mv  [in]     Struct where ALS variables are stored.
        * @param  st  [in,out] Struct where the returned values of @c Cpd are stored.
        *                      In this case the cost function value is updated.
        */
        void cost_function(Member_Variables const &mv,
                           Status                 &st)
        {
            st.f_value = sqrt( ( mv.tns_mat[lastFactor] - st.factors[lastFactor] * mv.krao[lastFactor].transpose() ).squaredNorm() );
        }

        /*
        * Compute the cost function value at the end of each outer iteration
        * based on the last accelerated factor.
        *
        * @param  mv                [in] Struct where ALS variables are stored.
        * @param  st                [in] Struct where the returned values of @c Cpd are stored.
        * @param  factors           [in] Accelerated factors.
        * @param  factors_T_factors [in] Gramian matrices of factors.
        *
        * @returns The cost function calculated with the accelerated factors.
        */
        double accel_cost_function(Member_Variables const &mv,
                                   Status           const &st,
                                   MatrixArray      const &factors,
                                   MatrixArray      const &factors_T_factors)
        {
            return sqrt( st.frob_tns + (PartialCwiseProd(factors_T_factors, lastFactor).cwiseProduct(factors_T_factors[lastFactor])).sum()
                    - 2 * (mv.mttkrp[lastFactor].cwiseProduct(factors[lastFactor])).sum() );
        }

        void cost_function2(Member_Variables const &mv,
                            Status                 &st)
        {
            st.f_value = sqrt( st.frob_tns -2 * (mv.mttkrp.cwiseProduct(st.factors[lastFactor])).sum() +
                    (mv.cwise_factor_product.cwiseProduct(mv.factor_T_factor[lastFactor])).sum() );
        }

        /*
        * Based on each factor's constraint, a different
        * update function is used at every outer iteration.
        *
        * Computes also factor^T * factor at the end.
        *
        * @param  idx [in]     Factor to be updated.
        * @param  mv  [in]     Struct where ALS variables are stored.
        * @param  st  [in,out] Struct where the returned values of @c Cpd are stored.
        *                      Updates the @c stl array with the factors.
        */
        void update_factor(int              const  idx,
                           Member_Variables        &mv,
                           Status                  &st    )
        {
```

```
                // Update factor
                switch ( st.options.constraints[idx] )
                {
                    case Constraint::unconstrained:
                    {
                        st.factors[idx].noalias() = mv.mttkrp[idx] * mv.cwise_factor_product.inverse();
                        break;
                    }
                        case Constraint::nonnegativity:
                    {
                        mv.temp_matrix = st.factors[idx];
                        NesterovMNLS(mv.cwise_factor_product, mv.mttkrp[idx], st.options.nesterov_delta_1,
                                              st.options.nesterov_delta_2, st.factors[idx]);
                        if(st.factors[idx].cwiseAbs().colwise().sum().minCoeff() == 0)
                            st.factors[idx] = 0.9 * st.factors[idx] + 0.1 * mv.temp_matrix;
                        break;
                    }
                    case Constraint::orthogonality:
                    {
                        mv.temp_matrix = mv.mttkrp[idx].transpose() * mv.mttkrp[idx];
                        Eigen::SelfAdjointEigenSolver<Matrix> eigensolver(mv.temp_matrix);
                        mv.temp_matrix.noalias() = (eigensolver.eigenvectors())
                                                    * (eigensolver.eigenvalues().cwiseInverse().cwiseSqrt().asDiagonal())
                                                    * (eigensolver.eigenvectors().transpose());
                        st.factors[idx].noalias() = mv.mttkrp[idx] * mv.temp_matrix;
                        break;
                    }
                    case Constraint::sparsity:
                        break;
                    default: // in case of Constraint::constant
                        break;
                }

                // Compute A^T * A + B^T * B + ...
                mv.factor_T_factor[idx].noalias() = st.factors[idx].transpose() * st.factors[idx];
            }

            /*
            * @brief Line Search Acceleration
            *
            * Performs an acceleration step on the updated factors, and keeps the accelerated factors
            * when the step succeeds. Otherwise, the acceleration step is ignored.
            * Line Search Acceleration reduces the number of outer iterations in the ALS algorithm.
            *
            * @note This implementation ONLY, if factors are of @c Matrix type.
            *
            * @param  mv  [in,out] Struct where ALS variables are stored.
            *                      In case the acceleration step is successful the Gramian
            *                      matrices of factors are updated.
            * @param  st  [in,out] Struct where the returned values of @c Cpd are stored.
            *                      If the acceleration succeeds updates @c factors
            *                      and cost function value.
            *
            */
            void line_search_accel(Member_Variables &mv,
                                   Status           &st)
            {
                double       f_accel    = 0.0; // Objective Value after the acceleration step
                double const accel_step = pow(st.ao_iter+1,(1.0/(st.options.accel_coeff)));

                MatrixArray  accel_factors;
                MatrixArray  accel_gramians;

                for(std::size_t i=0; i<TnsSize; ++i)
                {
                    accel_factors[i]  = mv.old_factors[i] + accel_step * (st.factors[i] - mv.old_factors[i]);
                    accel_gramians[i] = accel_factors[i].transpose() * accel_factors[i];
                }

                mttkrp(mv.tnsDims, accel_factors, mv.tns_mat[lastFactor], lastFactor, get_num_threads(), mv.mttkrp[lastFactor]);
                f_accel = accel_cost_function(mv, st, accel_factors, accel_gramians);
                if (st.f_value > f_accel)
                {
                    st.factors         = accel_factors;
                    mv.factor_T_factor = accel_gramians;
                    st.f_value         = f_accel;
                    Partensor()->Logger()->info("Acceleration Step SUCCEEDED! at iter: {}", st.ao_iter);
                }
                else
                    st.options.accel_fail++;

                if (st.options.accel_fail==5)
                {
```

```
                st.options.accel_fail=0;
                st.options.accel_coeff++;
            }
        }

        /*
        * Sequential implementation of Alternating Least Squares (ALS) method,
        * using Shared Memory and OpenMP.
        *
        * @param  R   [in]     The rank of decomposition.
        * @param  mv  [in]     Struct where ALS variables are stored and being updated
        *                      until a termination condition is true.
        * @param  st  [in,out] Struct where the returned values of @c Cpd are stored.
        */
        void als(std::size_t      const  R,
                 Member_Variables       &mv,
                 Status                 &status)
        {
            for (std::size_t i=0; i<TnsSize; i++)
            {
                mv.factor_T_factor[i].noalias() = status.factors[i].transpose() * status.factors[i];
                mv.tns_mat[i]                   = Matricization(mv.tnsX, i);
            }

            if(status.options.normalization)
            {
              choose_normilization_factor(status, mv.all_orthogonal, mv.weight_factor);
            }

            // Normalize(static_cast<int>(R), mv.factor_T_factor, status.factors);
            status.frob_tns          = square_norm(mv.tnsX);
            cost_function_init(mv, status);
            status.rel_costFunction = status.f_value/sqrt(status.frob_tns);

            // ---- Loop until ALS converges ----
            while (1)
            {
                status.ao_iter++;
                Partensor()->Logger()->info("iter: {} -- fvalue: {} -- relative_costFunction: {}",
status.ao_iter,
                                                                              status.f_value,
status.rel_costFunction);

                for (std::size_t i=0; i<TnsSize; i++)
                {
                    mv.krao[i]              = PartialKhatriRao(status.factors,  i);
                    mttkrp(mv.tnsDims, status.factors,  mv.tns_mat[i], i, get_num_threads(),
mv.mttkrp[i]);
                    mv.cwise_factor_product = PartialCwiseProd(mv.factor_T_factor, i);

                    // Update factor
                    update_factor(i, mv, status);

                    // Cost function Computation
                    if(i == lastFactor)
                        cost_function(mv, status);
                }

                status.rel_costFunction = status.f_value/sqrt(status.frob_tns);
                if(status.options.normalization && !mv.all_orthogonal)
                    Normalize(mv.weight_factor, static_cast<int>(R), mv.factor_T_factor,
status.factors);

                // ---- Terminating condition ----
                if (status.rel_costFunction < status.options.threshold_error || status.ao_iter >=
status.options.max_iter)
                {
                    if(status.options.writeToFile)
                        writeFactorsToFile(status);
                    break;
                }

                if (status.options.acceleration)
                {
                    mv.norm_factors = status.factors;
                    // ---- Acceleration Step ----
                    if (status.ao_iter > 1)
                        line_search_accel(mv, status);

                    mv.old_factors = mv.norm_factors;
                }
            } // end of while
        }

        Status operator()(Tensor_     const &tnsX,
                          std::size_t const  R)
        {
```

```
        Options         options = MakeOptions<Tensor_,execution::openmp_policy,DefaultValues>();
        Status          status(options);
        Member_Variables mv;

        // extract dimensions from tensor
        mv.tnsDims = tnsX.dimensions();
        // produce estimate factors using uniform distribution with entries in [0,1].
        makeFactors(mv.tnsDims, status.options.constraints, R, status.factors);
        mv.tnsX = tnsX;
        als(R, mv, status);

        return status;
      }

      Status operator()(Tensor_     const &tnsX,
                        std::size_t const  R,
                        Options     const &options)
      {
        Status          status(options);
        Member_Variables mv;

        // extract dimensions from tensor
        mv.tnsDims = tnsX.dimensions();
        // produce estimate factors using uniform distribution with entries in [0,1].
        makeFactors(mv.tnsDims, status.options.constraints, R, status.factors);
        mv.tnsX = tnsX;

        switch ( status.options.method )
        {
          case Method::als:
          {
            als(R, mv, status);
            break;
          }
          case Method::rnd:
            break;
          case Method::bc:
            break;
          default:
            break;
        }

        return status;
      };

      Status operator()(Tensor_     const &tnsX,
                        std::size_t const  R,
                        MatrixArray const &factorsInit)
      {
        Options         options = MakeOptions<Tensor_,execution::openmp_policy,DefaultValues>();
        Status          status(options);
        Member_Variables mv;

        status.factors = factorsInit;
        mv.tnsX        = tnsX;
        // extract dimensions from tensor
        mv.tnsDims     = tnsX.dimensions();
        als(R, mv, status);

        return status;
      }

      Status operator()(Tensor_     const &tnsX,
                        std::size_t const R,
                        Options     const &options,
                        MatrixArray const &factorsInit)
      {
        Status status(options);
        Member_Variables mv;

        status.factors = factorsInit;
        mv.tnsX        = tnsX;
        // extract dimensions from tensor
        mv.tnsDims     = tnsX.dimensions();

        switch ( status.options.method )
        {
          case Method::als:
          {
            als(R, mv, status);
            break;
          }
          case Method::rnd:
            break;
          case Method::bc:
            break;
          default:
```

```
      break;
  }

  return status;
}

Status operator()(std::array<int, TnsSize> const &tnsDims,
                  std::size_t              const  R,
                  std::string              const &path)
{
  Options           options = MakeOptions<Tensor_,execution::openmp_policy,DefaultValues>();
  Status            status(options);
  Member_Variables mv;

  long long int fileSize = 1;
  for(auto &dim : tnsDims)
    fileSize *= static_cast<long long int>(dim);

  mv.tnsX.resize(tnsDims);
  // Read the whole Tensor from a file
  read( path,
        fileSize,
        0,
        mv.tnsX   );
  // produce estimate factors using uniform distribution with entries in [0,1].
  makeFactors(tnsDims, status.options.constraints, R, status.factors);
  // mv.tnsDims = tnsDims;
  std::copy(tnsDims.begin(),tnsDims.end(),mv.tnsDims.begin());
  als(R, mv, status);

  return status;
}

Status operator()(std::array<int, TnsSize> const &tnsDims,
                  std::size_t              const  R,
                  std::string              const &path,
                  Options                  const &options)
{
  Status            status(options);
  Member_Variables mv;

  long long int fileSize = 1;
  for(auto &dim : tnsDims)
    fileSize *= static_cast<long long int>(dim);

  mv.tnsX.resize(tnsDims);
  // Read the whole Tensor from a file
  read( path,
        fileSize,
        0,
        mv.tnsX   );
  // produce estimate factors using uniform distribution with entries in [0,1].
  makeFactors(tnsDims, status.options.constraints, R, status.factors);
  std::copy(tnsDims.begin(),tnsDims.end(),mv.tnsDims.begin());

  switch ( status.options.method )
  {
    case Method::als:
    {
      als(R, mv, status);
      break;
    }
    case Method::rnd:
      break;
    case Method::bc:
      break;
    default:
      break;
  }

  return status;
}

Status operator()(std::array<int, TnsSize>          const &tnsDims,
                  std::size_t                       const  R,
                  std::array<std::string,TnsSize+1> const &paths)
{
  Options           options = MakeOptions<Tensor_,execution::openmp_policy,DefaultValues>();
  Status            status(options);
  Member_Variables mv;

  long long int fileSize = 1;
  for(auto &dim : tnsDims)
    fileSize *= static_cast<long long int>(dim);

  mv.tnsX.resize(tnsDims);
  // Read the whole Tensor from a file
```

```
          read( paths.front(),
                fileSize,
                0,
                mv.tnsX   );

          // Read initialized factors from files
          for(std::size_t i=0; i<TnsSize; ++i)
          {
            status.factors[i] = Matrix(tnsDims[i],static_cast<int>(R));
            read( paths[i+1],
                  tnsDims[i]*R,
                  0,
                  status.factors[i] );
          }
          std::copy(tnsDims.begin(),tnsDims.end(),mv.tnsDims.begin());

          als(R, mv, status);

          return status;
        }

        Status operator()(std::array<int, TnsSize>           const &tnsDims,
                          std::size_t                        const  R,
                          std::array<std::string, TnsSize+1> const &paths,
                          Options                            const &options)
        {
          Status status(options);
          Member_Variables mv;

          long long int fileSize = 1;
          for(auto &dim : tnsDims)
            fileSize *= static_cast<long long int>(dim);

          mv.tnsX.resize(tnsDims);
          // Read the whole Tensor from a file
          read( paths.front(),
                fileSize,
                0,
                mv.tnsX   );

          // Read initialized factors from files
          for(std::size_t i=0; i<TnsSize; ++i)
          {
            status.factors[i] = Matrix(tnsDims[i],static_cast<int>(R));
            read( paths[i+1],
                  tnsDims[i]*R,
                  0,
                  status.factors[i] );
          }
          std::copy(tnsDims.begin(),tnsDims.end(),mv.tnsDims.begin());

          switch ( status.options.method )
          {
            case Method::als:
            {
              als(R, mv, status);
              break;
            }
            case Method::rnd:
              break;
            case Method::bc:
              break;
            default:
              break;
          }

          return status;
        }

        Status operator()(std::array<int, TnsSize>         const &tnsDims,
                          std::size_t                      const  R,
                          std::array<std::string, TnsSize> const &true_paths,
                          std::array<std::string, TnsSize> const &init_paths,
                          Options                          const &options)
        {
          Status status(options);
          Member_Variables mv;

          for(std::size_t i=0; i<TnsSize; ++i)
          {
            mv.true_factors[i] = Matrix(tnsDims[i],static_cast<int>(R));
            status.factors[i]  = Matrix(tnsDims[i],static_cast<int>(R));

            // Read initialized factors from files
            read( true_paths[i],
                  tnsDims[i]*R,
                  0,
```

```
                    mv.true_factors[i] );
                // Read initialized factors from files
                read( init_paths[i],
                      tnsDims[i]*R,
                      0,
                      status.factors[i] );
              }
              std::copy(tnsDims.begin(),tnsDims.end(),mv.tnsDims.begin());

              switch ( status.options.method )
              {
                case Method::als:
                {
                  als_true_factors(R, mv, status);
                  break;
                }
                case Method::rnd:
                  break;
                case Method::bc:
                  break;
                default:
                  break;
              }

              return status;
            }
        };
      } // end namespace internal
    } // end namespace v1
  } // end namespace partensor
```

8.15 CwiseProd.hpp File Reference

`#include "PARTENSOR_basic.hpp"`

Functions

- template<typename... Matrices>
  Matrix CwiseProd (Matrix const &mat1, Matrix const &mat2, Matrices const &... mats)

  Element Wise product among Matrices.

8.15.1 Detailed Description

Implements the element wise product among two or more Matrices using the `Eigen` function `cwiseProduct`.

8.15.2 Function Documentation

8.15.2.1 CwiseProd()

```
Matrix partensor::v1::CwiseProd (
              Matrix const & mat1,
              Matrix const & mat2,
              Matrices const &...  mats )
```

Element Wise product among Matrices.

Expand implementation of `Eigen` `cwiseProduct` (element wise product) for two or more Matrices.

Template Parameters

Matrices	A template parameter pack (`stl variadic`) type, with possible multiple Matrices.

Parameters

mat1	[in] A `partensor::Matrix`.
mat2	[in] A `partensor::Matrix`.
mats	[in] Possible 0 or more A `partensor` Matrices.

Returns

Returns the result `Matrix` with the element wise product among the given matrices.

8.16 CwiseProd.hpp

Go to the documentation of this file.

```
#ifndef DOXYGEN_SHOULD_SKIP_THIS
#endif  // DOXYGEN_SHOULD_SKIP_THIS
/*************************************************************************/
#ifndef PARTENSOR_CWISE_PROD_HPP
#define PARTENSOR_CWISE_PROD_HPP

#include "PARTENSOR_basic.hpp"

namespace partensor {

  inline namespace v1 {
    template<typename... Matrices>
    Matrix CwiseProd( Matrix   const &mat1,
                      Matrix   const &mat2,
                      Matrices const &... mats )
    {
      if constexpr (sizeof... (mats) == 0)
      {
        return mat1.cwiseProduct(mat2);
      }
      else
      {
        auto _temp = CwiseProd(mat2, mats...);

        return CwiseProd(mat1, _temp);
      }
    }

  } // end namespace v1

} // end namespace partensor

#endif // PARTENSOR_CWISE_PROD_HPP
```

8.17 DataGeneration.hpp File Reference

```
#include "PARTENSOR_basic.hpp"
#include "unsupported/Eigen/CXX11/src/Tensor/TensorRandom.h"
#include "PartialKhatriRao.hpp"
#include "TensorOperations.hpp"
#include "DimTrees.hpp"
#include "Constants.hpp"
#include <time.h>
#include <cassert>
```

Functions

- void generateRandomMatrix (Matrix &mtx)

    Fills a `Matrix` with pseudo random data.
- template< typename Tensor_ >
  void generateRandomTensor (Tensor_ &tnsX, unsigned const distribution=0)

    Fills a `Tensor` with pseudo random data.
- template< std::size_t _TnsSize>
  Tensor< static_cast< int >(_TnsSize)> generateTensor (std::array< Tensor< 2 >, _TnsSize > &factor↩
  Array)

    Computes the Tensor from an array of factors.
- template< std::size_t Size>
  Matrix generateTensor (std::size_t const mode, std::array< Matrix, Size > const &factorArray)

    Computes the matriced Tensor from an array of factors.
- template< std::size_t _TnsSize, typename Dimensions , typename FactorType >
  void makeFactors (Dimensions const &tnsDims, std::array< Constraint, _TnsSize > const &constraints, int const R, std::array< FactorType, _TnsSize > &factorArray)

    Creates an `stl` array, where Matrices-Factors are stored.
- template< std::size_t _TnsSize>
  void makeTensor (std::array< int, _TnsSize > const &tnsDims, std::array< Constraint, _TnsSize > const &constraints, std::size_t const R, Tensor< static_cast< int >(_TnsSize)> &tnsX)

    Initialize a Tensor with constraints applied.

8.17.1 Detailed Description

Includes a variety of functions, that either create synthetic data or reform them.

8.17.2 Function Documentation

8.17.2.1 generateRandomMatrix()

```
void partensor::generateRandomMatrix (
            Matrix & mtx )
```

Fills a `Matrix` with pseudo random data.

Generate synthetic-random Matrix in uniform distribution with numbers in [0,1].

Parameters

mtx	[in,out] The `Matrix`, where the data will be stored.

Note

`mtx` must initiallized before the function call.

8.17.2.2 generateRandomTensor()

```
void partensor::generateRandomTensor (
            Tensor_ & tnsX,
            unsigned const distribution = 0 )
```

Fills a `Tensor` with pseudo random data.

Generates synthetic-random data for a `Tensor` based on a distribution. The distribution can be either uniform or normal.

Template Parameters

Tensor←_	Type(data type and order) of input Tensor `tnsX`. `Tensor_` must be `Tensor<order>`, where `order` must be in range of [3-8].

Parameters

tnsX	[in,out] The given Tensor to be filled with data, based on `distribution`.
distribution	[in] If 0 data the data are chosen from a Uniform distribution in range [0,1], else in Normal distribution with `mean=0` and standard `deviation()`=1. The default is Uniform distribution.

Note

`tnsX` must be initialized before the function call.

8.17.2.3 generateTensor() [1/2]

```
Tensor< static_cast< int >(_TnsSize)> partensor::generateTensor (
            std::array< Tensor< 2 >, _TnsSize > & factorArray )
```

Computes the Tensor from an array of factors.

If there are factors saved in an `stl` array, then `generateTensor`, can be used to produce the matricized Tensor. It computes the matricized Tensor, from contraction of all factors in `factorArray`.

Template Parameters

_TnsSize	Tensor Order of the Tensor.

Parameters

factorArray	[in] An `stl` array with all Factors of type `Tensor<2>`.

Returns

The generated Tensor from the `factorArray`.

8.17.2.4 generateTensor() [2/2]

```
Matrix partensor::generateTensor (
            std::size_t const mode,
            std::array< Matrix, Size > const & factorArray )
```

Computes the matriced Tensor from an array of factors.

If there are factors saved in an `stl` array, then `generateTensor`, can be used to produce the matricized Tensor. It computes the matricized Tensor, from the `Khatri-Rao` product of factors, based on the chosen `mode` for the matricization.

Template Parameters

MatrixArray	An array container with Matrices of `Matrix` type.

Parameters

mode	[in] The dimension, in which the matricized Tensor will be returned.
factorArray	[in] The array with the factors of `Matrix` type, where its size must be equal to Tensor order.

Returns

The Tensor matricization based on the factors.

8.17.2.5 makeFactors()

```
void partensor::makeFactors (
            Dimensions const & tnsDims,
            std::array< Constraint, _TnsSize > const & constraints,
            int const R,
            std::array< FactorType, _TnsSize > & factorArray )
```

Creates an `stl` array, where Matrices-Factors are stored.

Creates a pseudo-random `stl` array with Matrices-Factors. These factors can be of type either `Matrix` or `FactorDimTree`. Also, there can be applied different `constraints` to each factor, specified in `Constraint` enumeration from `Constants.hpp`.

An array container with the dimension per factor is needed and the variable `R`, which indicates the number of columns of each factor.

Template Parameters

_TnsSize	Essentially tensor order, but also the size of `constraints` and `factorArray` arrays.
Dimensions	Array container for `tnsDims`.
FactorType	The type for `factorArray` and the generated Factors, either `Eigen` Matrix or `FactorDimTree`.

Parameters

tnsDims	[in] The row dimension for each factor.
constraints	[in] The `Constraint` to apply to each Factor, check `Constants.hpp`.
R	[in] Rank of factorization (Number of columns in each `Matrix` or `FactorDimTree`).
factorArray	[int,out] An `stl` array containing all factors with type `FactorType`.

8.17.2.6 makeTensor()

```
void partensor::makeTensor (
             std::array< int, _TnsSize > const & tnsDims,
             std::array< Constraint, _TnsSize > const & constraints,
             std::size_t const R,
             Tensor< static_cast< int >(_TnsSize)> & tnsX )
```

Initialize a Tensor with constraints applied.

Creates a tensor `tnsX`, with dimensions specified in `tnsDims`. In order to create `tnsX`, some factors will be created. The number of factors being created is equal to `tnsDims` size. On each factor can be applied a constraint of type `Constraint`. Check `Constants.hpp` for the other constraints. Default value is `nonnegative` constraint. Also, the rank `R`, of these factors is needed.

Template Parameters

_TnsSize	Size of the `tnsDims`, `constraints` arrays and the number of `tnsX` dimensions.

Parameters

tnsDims	[in] `stl` array with each dimension for `tnsX`.
constraints	[in] The constraints to be applied in on each factor that will be used to generate `tnsX`.
R	[in] Essentially, the number of columns for each factor.
tnsX	[in,out] The tensor to be created based on the other parameters. It is a `Tensor` type.

8.18 DataGeneration.hpp

Go to the documentation of this file.

```
#ifndef DOXYGEN_SHOULD_SKIP_THIS
#endif  // DOXYGEN_SHOULD_SKIP_THIS
/*********************************************************************/
#ifndef PARTENSOR_DATA_GENERATION_HPP
#define PARTENSOR_DATA_GENERATION_HPP

#include "PARTENSOR_basic.hpp"
#include "unsupported/Eigen/CXX11/src/Tensor/TensorRandom.h"
#include "PartialKhatriRao.hpp"
#include "TensorOperations.hpp"
#include "DimTrees.hpp"
#include "Constants.hpp"
// #include "Normalize.hpp"
#include <time.h>
#include <cassert>
```

```cpp

// Use (void) to silent unused warnings.
#define assertm(exp, msg) assert(((void)msg, exp))

namespace partensor
{

        void generateRandomMatrix(Matrix &mtx)
        {
                int m = mtx.rows();
                int n = mtx.cols();
                assertm(m>0 && n>0, "Rows and columns must be greater than 1 in generateRandomMatrix\n");

                std::srand((unsigned int) time(NULL)+std::rand());

                mtx = (Matrix::Random(m, n) + Matrix::Ones(m, n))/2;
        }

        template<typename Tensor_>
        void generateRandomTensor( Tensor_        &tnsX,
                                   unsigned const  distribution = 0 )
        {
                using DataType = typename TensorTraits<Tensor_>::DataType;
                std::srand((unsigned int) time(NULL)+std::rand());

                if(distribution == 0)// uniform distribution with numbers in [0,1]
                {
                        tnsX.template setRandom<Eigen::internal::UniformRandomGenerator<DataType»();
                }
                else if (distribution == 1) // normal distribution with mean = 0 and deviation() = 1
                {
                        tnsX.template setRandom<Eigen::internal::NormalRandomGenerator<DataType»();
                }
                else
                {
                        throw std::runtime_error("Choose correct distribution in generateRandomTensor().\n");
                }
        }

        template<std::size_t Size>
        Matrix generateTensor( std::size_t             const  mode,
                                                   std::array<Matrix,Size> const &factorArray)
        {
                if (mode > Size-1) { throw std::runtime_error("Mode must be in [0,factorArray.Size) in generateTensor().\n"); }

                Matrix partial_krao = PartialKhatriRao(factorArray, mode);
                return (factorArray[mode] * partial_krao.transpose());
        }

        template<std::size_t _TnsSize>
        Tensor<static_cast<int>(_TnsSize)> generateTensor( std::array<Tensor<2>,_TnsSize> &factorArray)
        {
                static_assert(_TnsSize>0, "Tensor cannot be scalar in generateTensor()!\n");

                using MatrixArray = std::array<Tensor<2>,_TnsSize>;
                // same for all factors
                const std::size_t R = factorArray[0].dimension(1);
                // Initialize Core Tensor for PARAFAC
                std::array<int,_TnsSize> dim;
                int                     i = 0;
                constexpr int           w = 1;

                Tensor<static_cast<int>(_TnsSize)>    tnsX;
                Tensor<static_cast<int>(_TnsSize)>    Temp_X;

                std::fill(dim.begin(), dim.end(), R);
                IdentityTensorGen(dim, tnsX);

                std::array<Eigen::IndexPair<int>, 1> product_dims;

                for(typename MatrixArray::reverse_iterator it=factorArray.rbegin(); it != factorArray.rend(); ++it)
                {
                        product_dims = { Eigen::IndexPair<int>(w,i) };

                        Temp_X = (*it).contract(tnsX, product_dims);
                        tnsX   = Temp_X;
                        i++;
                }
                return tnsX;
        }

        #ifndef DOXYGEN_SHOULD_SKIP_THIS
        template <std::size_t idx, std::size_t _TnsSize>
        void fillDimTreeFactors_Matrix( std::array<Matrix, _TnsSize>        const &factorArray,
```

```
                                                        std::array<Constraint, _TnsSize> const &constraints,
                                                        std::array<FactorDimTree, _TnsSize>       &factorDimTreeArray )
    {
        const Matrix                        &_factor        = factorArray[idx];
        FactorDimTree                       &_factorDimTree = factorDimTreeArray[idx];
        std::array<Eigen::IndexPair<int>, 1>   product_dims   = { Eigen::IndexPair<int>(0, 0) };

        const int rows = _factor.rows();
        const int cols = _factor.cols();

        _factorDimTree.factor.resize(rows,cols);
        _factorDimTree.gramian.resize(cols,cols);
        _factorDimTree.constraint = constraints[idx];
        _factorDimTree.factor     = matrixToTensor(_factor, rows, cols);
        _factorDimTree.gramian    = _factorDimTree.factor.contract(_factorDimTree.factor, product_dims);

        if constexpr (idx+1 < _TnsSize)
        {
            fillDimTreeFactors_Matrix<idx + 1, _TnsSize>(factorArray, constraints, factorDimTreeArray);
        }
    }
    #endif  // DOXYGEN_SHOULD_SKIP_THIS

    #ifndef DOXYGEN_SHOULD_SKIP_THIS
    template <std::size_t idx, std::size_t _TnsSize>
    void fillDimTreeFactors_Tensor( std::array<Tensor<2>, _TnsSize>     const &factorArray,
                                                        std::array<Constraint, _TnsSize> const &constraints,
                                                        std::array<FactorDimTree, _TnsSize>       &factorDimTreeArray )
    {
        const Tensor<2>                     &_factor        = factorArray[idx];
        FactorDimTree                       &_factorDimTree = factorDimTreeArray[idx];
        std::array<Eigen::IndexPair<int>, 1>   product_dims   = { Eigen::IndexPair<int>(0, 0) };

        const int rows = _factor.dimension(0); // rows
        const int cols = _factor.dimension(1); // cols

        _factorDimTree.factor.resize(rows,cols);
        _factorDimTree.gramian.resize(cols,cols);
        _factorDimTree.constraint = constraints[idx];
        _factorDimTree.factor     = _factor;
        _factorDimTree.gramian    = _factorDimTree.factor.contract(_factorDimTree.factor, product_dims);

        if constexpr (idx+1 < _TnsSize)
        {
            fillDimTreeFactors_Tensor<idx + 1, _TnsSize>(factorArray, constraints, factorDimTreeArray);
        }
    }
    #endif  // DOXYGEN_SHOULD_SKIP_THIS

    #ifndef DOXYGEN_SHOULD_SKIP_THIS
    template <std::size_t _TnsSize, typename FactorType>
    void fillDimTreeFactors( std::array<FactorType, _TnsSize> const &factorArray,
                             std::array<Constraint, _TnsSize> const &constraints,
                                                   std::array<FactorDimTree, _TnsSize> &factorDimTreeArray )
    {
        constexpr bool check = (std::is_same_v<FactorType,Matrix> ||
                                std::is_same_v<FactorType,Tensor<2>>);
        static_assert(check, "Factors must be of type Matrix or Tensor<2>, fillDimTreeFactors()\n");

        if constexpr (std::is_same_v<FactorType,Matrix>)
            fillDimTreeFactors_Matrix<0,_TnsSize>(factorArray, constraints, factorDimTreeArray);
        else
            fillDimTreeFactors_Tensor<0,_TnsSize>(factorArray, constraints, factorDimTreeArray);
    }
    #endif  // DOXYGEN_SHOULD_SKIP_THIS

    #ifndef DOXYGEN_SHOULD_SKIP_THIS
    template<std::size_t _TnsSize>
    void CpdGen( std::array<FactorDimTree,_TnsSize>       &FactorArray,
                             int                                                   const  R,
                             Tensor<static_cast<int>(_TnsSize)>       &tnsX)
    {
        static_assert(_TnsSize>0, "Tensor cannot be scalar in CpdGen()!\n");

        assertm(R>0, "Variable R - factor column must be greater than one in CpdGen()!\n");

```

```
            using MatrixArray = std::array<FactorDimTree,_TnsSize>;
            // Initialize Core Tensor for PARAFAC
            std::array<int,_TnsSize> dim;
            int                  i = 0;
            constexpr int        w = 1;

            Tensor<static_cast<int>(_TnsSize)>   Temp_X;
            std::array<Eigen::IndexPair<int>, 1> product_dims;

            std::fill(dim.begin(), dim.end(), R);
            IdentityTensorGen(dim, tnsX);

            for(typename MatrixArray::reverse_iterator it=FactorArray.rbegin(); it != FactorArray.rend(); ++it)
            {
                    product_dims = { Eigen::IndexPair<int>(w,i) };

                    Temp_X = (it->factor).contract(tnsX, product_dims);
                    tnsX   = Temp_X;
                    i++;
            }
    }
    #endif  // DOXYGEN_SHOULD_SKIP_THIS

    #ifndef DOXYGEN_SHOULD_SKIP_THIS
    template<std::size_t idx, std::size_t _TnsSize, typename Dimensions>
    void makeFactors_Matrix( Dimensions                              const &tnsDims,
                             std::array<Constraint,_TnsSize> const &constraints,
                             int                                     const  R,
                             std::array<Matrix,_TnsSize>            &factorArray  )
    {
            Matrix  &_factor = factorArray[idx];
            _factor.resize(tnsDims[idx], R);
            generateRandomMatrix(_factor);

            switch(constraints[idx])
            {
                    case Constraint::unconstrained:
                    {
                            break;
                    }
                    case Constraint::nonnegativity:
                    {
                            Matrix Zeros = Matrix::Zero(_factor.rows(),_factor.cols());
                            _factor = _factor.cwiseMax(Zeros);
                            break;
                    }
                    case Constraint::orthogonality:
                    {
                            Eigen::JacobiSVD<Matrix> svd(_factor, Eigen::ComputeThinU | Eigen::ComputeThinV);
                            _factor = svd.matrixU();
                            break;
                    }
                    case Constraint::sparsity:
                    {
                            break;
                    }
                    default: // in case of Constraint::constant
                    {
                            break;
                    }
            }

            if constexpr (idx+1 < _TnsSize)
            {
                    makeFactors_Matrix<idx + 1, _TnsSize>(tnsDims, constraints, R, factorArray);
            }
    }
    #endif  // DOXYGEN_SHOULD_SKIP_THIS

    #ifndef DOXYGEN_SHOULD_SKIP_THIS
    template<std::size_t idx, std::size_t _TnsSize, typename Dimensions>
    void makeFactors_Tensor( Dimensions                      const &tnsDims,
                             std::array<Constraint,_TnsSize>  const &constraints,
                             int                              const  R,
                             std::array<Tensor<2>,_TnsSize>         &factorArray )
    {
            Tensor<2>  &_factor = factorArray[idx];

            _factor.resize(tnsDims[idx],R);
            generateRandomTensor(_factor);
```

```

        switch(constraints[idx])
        {
            case Constraint::unconstrained:
            {
                break;
            }
            case Constraint::nonnegativity:
            {
                _factor = (_factor.abs()).eval();
                break;
            }
            case Constraint::orthogonality:
            {
                Matrix _mtx = tensorToMatrix(_factor, tnsDims[idx], R);

                Eigen::JacobiSVD<Matrix> svd(_mtx, Eigen::ComputeThinU | Eigen::ComputeThinV);
                _mtx = svd.matrixU();

                _factor = matrixToTensor(_mtx, tnsDims[idx], R);
                break;
            }
            case Constraint::sparsity:
            {
                break;
            }
            default: // in case of Constraint::constant
            {
                break;
            }
        }

        if constexpr (idx+1 < _TnsSize)
        {
            makeFactors_Tensor<idx + 1, _TnsSize>(tnsDims, constraints, R, factorArray);
        }
    }
    #endif // DOXYGEN_SHOULD_SKIP_THIS

    #ifndef DOXYGEN_SHOULD_SKIP_THIS
    template<std::size_t idx, std::size_t _TnsSize, typename Dimensions>
    void makeFactors_DimTree( Dimensions                                const &tnsDims,
                              std::array<Constraint,_TnsSize>    const &constraints,
                              int                                 const  R,
                              std::array<FactorDimTree,_TnsSize> &factorArray )
    {
        FactorDimTree                         &_factor      = factorArray[idx];
        std::array<Eigen::IndexPair<int>, 1>  product_dims = { Eigen::IndexPair<int>(0, 0) };

        _factor.factor.resize(tnsDims[idx],R);
        _factor.gramian.resize(R,R);
        _factor.constraint = constraints[idx];

        generateRandomTensor(_factor.factor);

        switch(constraints[idx])
        {
            case Constraint::unconstrained:
            {
                break;
            }
            case Constraint::nonnegativity:
            {
            _factor.factor = (_factor.factor.abs()).eval();
            break;
            }
            case Constraint::orthogonality:
            {
                Matrix _mtx = tensorToMatrix(_factor.factor, tnsDims[idx], R);

                Eigen::JacobiSVD<Matrix> svd(_mtx, Eigen::ComputeThinU | Eigen::ComputeThinV);
                _mtx = svd.matrixU();

                _factor.factor = matrixToTensor(_mtx, tnsDims[idx], R);
            break;
            }
            case Constraint::sparsity:
            {
                break;
            }
            default: // in case of Constraint::constant
            {
                break;
```

```
                        }
                }
                _factor.gramian = _factor.factor.contract(_factor.factor, product_dims);

                if constexpr (idx+1 < _TnsSize)
                {
                        makeFactors_DimTree<idx + 1, _TnsSize>(tnsDims, constraints, R, factorArray);
                }
        }
        #endif  // DOXYGEN_SHOULD_SKIP_THIS

        template<std::size_t _TnsSize, typename Dimensions, typename FactorType>
        void makeFactors( Dimensions                    const &tnsDims,
                                        std::array<Constraint,_TnsSize> const &constraints,
                                        int                             const  R,
                                        std::array<FactorType,_TnsSize>       &factorArray )
        {
                constexpr bool check = (std::is_same_v<FactorType,Matrix> ||
                                        std::is_same_v<FactorType,Tensor<2»  ||

    std::is_same_v<FactorType,FactorDimTree>);
                static_assert(check, "Factors must be of type Matrix, Tensor<2> or FactorDimTree,
    makeFactors()\n");

                if constexpr (std::is_same_v<FactorType,Matrix>)
                        makeFactors_Matrix<0, _TnsSize>(tnsDims, constraints, R, factorArray);
                else if constexpr (std::is_same_v<FactorType,Tensor<2»)
                        makeFactors_Tensor<0, _TnsSize>(tnsDims, constraints, R, factorArray);
                else
                        makeFactors_DimTree<0, _TnsSize>(tnsDims, constraints, R, factorArray);

        }

        template <std::size_t _TnsSize>
        void makeTensor( std::array<int,_TnsSize>           const &tnsDims,
                         std::array<Constraint,_TnsSize>    const &constraints,
                         std::size_t                        const  R,
                                        Tensor<static_cast<int>(_TnsSize)>       &tnsX   )
        {
                static_assert(_TnsSize>0, "Tensor cannot be scalar in makeTensor()!\n");

                assertm(R>0, "Variable R - factor column must be greater than one in makeTensor()!\n");

                using MatrixArray = typename TensorTraits<Tensor<_TnsSize>>::MatrixArray;

                MatrixArray  true_factors;
        MatrixArray  gramians;
                Matrix       matricized_tensor;

                tnsX.resize(tnsDims);
                makeFactors(tnsDims, constraints, R, true_factors);
                for (std::size_t i=0; i<_TnsSize; ++i)
        {
                        gramians[i].noalias() = true_factors[i].transpose() * true_factors[i];
        }
                // Normalize(R, gramians, true_factors);
                matricized_tensor = generateTensor(0, true_factors);
                tnsX              = matrixToTensor(matricized_tensor, tnsDims);
        }

        #ifndef DOXYGEN_SHOULD_SKIP_THIS
        template <std::size_t _TnsSize>
        void makeTensor( std::array<int,_TnsSize>           const &tnsDims,
                         std::array<Constraint,_TnsSize>    const &constraints,
                         std::size_t                        const  R,
                                        std::array<Matrix,_TnsSize>              &true_factors,
                                        Tensor<static_cast<int>(_TnsSize)>       &tnsX)
        {
                static_assert(_TnsSize>0, "Tensor cannot be scalar in makeTensor()!\n");

                assertm(R>0, "Variable R - factor column must be greater than one in makeTensor()!\n");

                using MatrixArray = typename TensorTraits<Tensor<_TnsSize»::MatrixArray;

        MatrixArray  gramians;
                Matrix       matricized_tensor;

                tnsX.resize(tnsDims);
                makeFactors(tnsDims, constraints, R, true_factors);

                for (std::size_t i=0; i<_TnsSize; ++i)
        {
                        gramians[i].noalias() = true_factors[i].transpose() * true_factors[i];
        }

                matricized_tensor = generateTensor(0, true_factors);
                tnsX              = matrixToTensor(matricized_tensor, tnsDims);
```

```
    }
    #endif  // DOXYGEN_SHOULD_SKIP_THIS

    #ifndef DOXYGEN_SHOULD_SKIP_THIS
    template<typename Dimensions, typename MatrixArray>
    void generateFactors( std::size_t const  rank,
                          Dimensions  const &tnsDims,
                                      unsigned    const  distribution,
                                      MatrixArray       &factorArray)
    {
        using Matrix = typename MatrixArrayTraits<MatrixArray>::value_type;

        constexpr std::size_t TnsSize = MatrixArrayTraits<MatrixArray>::Size;

        std::srand((unsigned int) time(NULL)+std::rand());
        switch(distribution)
        {
            case 0: // uniform distribution with numbers in [-1,1]
            {
                for(std::size_t i=0; i<TnsSize; i++) { factorArray[i] = Matrix::Random(tnsDims[i], rank); }
                break;
            }
            case 1: // uniform distribution with numbers in [0,1]
            {
                for(std::size_t i=0; i<TnsSize; i++) { factorArray[i] = (Matrix::Random(tnsDims[i], rank) + Matrix::Ones(tnsDims[i], rank))/2; }
                break;
            }
            case 2: // normal distribution with mean = 0 and deviation() = 1
            {
                std::random_device rd;
                std::mt19937 e2(rd());
                std::normal_distribution<> dist(0.0, 1.0);
                for(std::size_t i=0; i<TnsSize; i++)
                {
                    factorArray[i] = Matrix(tnsDims[i],rank);
                for(std::size_t j=0; j<static_cast<std::size_t>(tnsDims[i]); j++)
                    {
                        for(std::size_t k=0; k<rank; k++)
                        {
                    factorArray[i](j,k) = dist(e2);
                        }
                }
            }
                break;
            }
            default:
            {
                break;
            }
        }
    }
    #endif // DOXYGEN_SHOULD_SKIP_THIS

    #ifndef DOXYGEN_SHOULD_SKIP_THIS
    template<typename  Dimensions, typename Tensor_>
    void customTensor( Dimensions const &tns_dim,
                                 Tensor_         &tnsX     )
    {
        static constexpr std::size_t TnsSize = TensorTraits<Tensor_>::TnsSize;

        tnsX.resize(tns_dim);
        int count = 0;

        if constexpr (TnsSize == 3)
        {
            for(int k=0; k<tns_dim[2]; k++)
            {
                for(int j=0; j<tns_dim[1]; j++)
                {
                    for(int i=0; i<tns_dim[0]; i++)
                    {
                            tnsX(i,j,k) = count+1;
                            count++;
                    }
                }
            }
        }
        else if constexpr (TnsSize == 4)
        {
            for(int l=0; l<tns_dim[3]; l++)
   {
     for(int k=0; k<tns_dim[2]; k++)
     {
       for(int j=0; j<tns_dim[1]; j++)
       {
```

```
            for(int i=0; i<tns_dim[0]; i++)
            {
                tnsX(i,j,k,l) = count+1;
                count++;
            }
          }
        }
      }
                }
                else if constexpr (TnsSize == 5)
                {
                        for(int m=0; m<tns_dim[4]; m++)
      {
        for(int l=0; l<tns_dim[3]; l++)
        {
          for(int k=0; k<tns_dim[2]; k++)
          {
            for(int j=0; j<tns_dim[1]; j++)
            {
              for(int i=0; i<tns_dim[0]; i++)
              {
                  tnsX(i,j,k,l,m) = count+1;
                  count++;
              }
            }
          }
        }
      }
        }
        #endif // DOXYGEN_SHOULD_SKIP_THIS

        }
} // end namespace partensor

#endif // end of PARTENSOR_DATA_GENERATION_HPP
```

8.19 DimTrees.hpp File Reference

```
#include "PARTENSOR_basic.hpp"
#include "TensorOperations.hpp"
#include "Constants.hpp"
```

Classes

- struct ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >
- struct ExprTree< _TnsSize >
- struct FactorDimTree
- struct I_TnsNode
- struct TnsNode< _TnsSize >
- struct TnsNode< 0 >

Functions

- template<std::size_t _TreeDim>
  I_TnsNode ∗ search_leaf (int const key, ExprTree< _TreeDim > &tree)

8.19.1 Detailed Description

Implementation of Dimension Trees functionality.

8.19.2 Function Documentation

8.19.2.1 search_leaf()

```
I_TnsNode * partensor::search_leaf (
            int const key,
            ExprTree< _TreeDim > & tree )
```

Searches in whole `tree` to find the `ExprNode` with the `key`. Make use of `SearchKey`.

Parameters

key	[in] Searching key value
tree	[in,out] Expression tree

Returns

The `ExprNode` in the specified `tree`, that has `key`.

8.20 DimTrees.hpp

Go to the documentation of this file.

```
#ifndef DOXYGEN_SHOULD_SKIP_THIS
#endif  // DOXYGEN_SHOULD_SKIP_THIS
/**********************************************************************/
#ifndef DIM_TREES_HPP
#define DIM_TREES_HPP

#include "PARTENSOR_basic.hpp"
#include "TensorOperations.hpp"
#include "Constants.hpp"

namespace partensor
{
  struct FactorDimTree;

  struct  I_TnsNode
  {
    using DataType = DefaultDataType;
    std::size_t   TnsSize;
    I_TnsNode(std::size_t _TnsSize) : TnsSize(_TnsSize)
    {  }
    virtual I_TnsNode *Parent() = 0;
    virtual I_TnsNode *Left()   = 0;
    virtual I_TnsNode *Right()  = 0;

    virtual bool Updated()       = 0;
    virtual void SetOutdated()   = 0;

    virtual int                          Key()      = 0;

    virtual Tensor<2>                   &Gramian()  = 0;

    virtual void                        *TnsDims()  = 0;

    virtual void                        *LabelSet() = 0;

    virtual void                        *DeltaSet() = 0;

     virtual void                        *TensorX()  = 0;

     virtual I_TnsNode *SearchKey  (int const key                                            ) = 0;

```

```
  virtual void      UpdateTree(int const num_factors, int const id, FactorDimTree *factors) = 0;

};

template <std::size_t _TnsSize>
struct TnsNode : I_TnsNode
{
  static constexpr std::size_t TnsSize = _TnsSize;
  static constexpr bool        IsNull  = false;
  using DataType     = I_TnsNode::DataType;
  using Tensor_Type  = Tensor<static_cast<int>(TnsSize)>;
  using Hessian_Type = Tensor<2>;

  Tensor_Type      mTnsX;
  Hessian_Type     mGramian;
  int              mKey;
  bool             mUpdated;
  TnsNode() : I_TnsNode(TnsSize), mKey(0), mUpdated(false)
  {  }

  bool Updated() override
  {
    return mUpdated;
  }

  void SetOutdated() override
  {
    if (mUpdated)
    {
      if ( !Left() && !Right() )
      {
        mUpdated = false;
      }
      else
      {
        Left()->SetOutdated();
        Right()->SetOutdated();
        mUpdated = false;
      }
    }
  }

  int Key() override
  {
    return mKey;
  }

  Hessian_Type &Gramian() override
  {
    return mGramian;
  }

  void *TensorX() override
  {
    return &mTnsX;
  }

};

template <>
struct TnsNode<0> : public I_TnsNode
{
  static constexpr std::size_t  TnsSize = 0;
  static constexpr bool         IsNull  = true;
  using DataType    = I_TnsNode::DataType;
  using Tensor_Type = Tensor<0>;
  template <typename N>
  TnsNode(N par = nullptr) : I_TnsNode(TnsSize)
  {  (void) par; }

  I_TnsNode *Parent()  override { return nullptr; }
  I_TnsNode *Left   ()  override { return nullptr; }
  I_TnsNode *Right  ()  override { return nullptr; }

  void       SetOutdated() override  { throw std::runtime_error("SetOutdated()");  }
  bool       Updated()     override  { throw std::runtime_error("Updated()");      }
  int        Key()         override  { throw std::runtime_error("Key()");          }
  void      *TnsDims()     override  { throw std::runtime_error("TnsDims()");      }
  void      *LabelSet()    override  { throw std::runtime_error("LabelSet()");     }
  void      *DeltaSet()    override  { throw std::runtime_error("DeltaSet()");     }
  Tensor<2> &Gramian()     override  { throw std::runtime_error("Gramian()");      }
  void      *TensorX()     override  { throw std::runtime_error("TensorX()");      }
  void       UpdateTree(int const, int const, FactorDimTree *) override {  throw
 std::runtime_error("UpdateTree()");   }
  I_TnsNode  *SearchKey (int const                              ) override {  throw
 std::runtime_error("SearchKey()");    }
};
```

```

using  NullTensorType = TnsNode<0>::Tensor_Type;

template <std::size_t _LabelSetSize, std::size_t _ParLabelSetSize, std::size_t _RootSize>
struct ExprNode : public TnsNode<_LabelSetSize == _RootSize ? _LabelSetSize : _LabelSetSize + 1>
{
  using DataType = typename I_TnsNode::DataType;

  //static_assert(LabelSetSize != 0, "Error in expansion!");
  static constexpr std::size_t LabelSetSize    = _LabelSetSize;
  static constexpr std::size_t ParLabelSetSize = _ParLabelSetSize;
  static constexpr std::size_t RootSize        = _RootSize;
  static constexpr bool IsRoot       = LabelSetSize == RootSize;
  static constexpr bool IsFirstChild = ParLabelSetSize   == RootSize;
  static constexpr bool IsLeaf       = LabelSetSize == 1;
  static constexpr std::size_t BrotherLabelSetSize = IsRoot ? 0 : ParLabelSetSize - LabelSetSize;

  static constexpr std::size_t TnsSize             = LabelSetSize == RootSize ? LabelSetSize : LabelSetSize + 1;
  static constexpr std::size_t ParTnsSize          = IsRoot ? 0 : (IsFirstChild ? ParLabelSetSize : ParLabelSetSize+1);
  static constexpr std::size_t DIM_HALF_SIZE       = (1+LabelSetSize)/2;

  static constexpr std::size_t DIM_LEFT_SIZE       = IsLeaf ? 0 : DIM_HALF_SIZE;

  static constexpr std::size_t DIM_RIGHT_SIZE      = IsLeaf ? 0 : LabelSetSize - DIM_LEFT_SIZE;

  std::array<int,TnsSize>             mTnsDims;
  std::array<int,LabelSetSize>        mLabelSet;
  std::array<int,BrotherLabelSetSize> mDeltaSet;
  using  Tns_Node_Type        = TnsNode<TnsSize>;
  using  Parent_Tns_Node_Type = TnsNode<ParTnsSize>;
  using  Node_Type            = ExprNode<LabelSetSize,ParLabelSetSize,RootSize>;
  using  Left_Node_Type       = std::conditional_t<IsLeaf, TnsNode<0>, ExprNode<DIM_LEFT_SIZE,LabelSetSize,RootSize>>;
  using  Right_Node_Type      = std::conditional_t<IsLeaf, TnsNode<0>, ExprNode<DIM_RIGHT_SIZE,LabelSetSize,RootSize>>;

  // const int             mParLabelSetSize;
  // const int             mParParLabelSetSize;
  // const std::ptrdiff_t  mParOffset;

  using Tensor_Type        = typename Tns_Node_Type::Tensor_Type;
  using Hessian_Type       = typename Tns_Node_Type::Hessian_Type;
  using Parent_Tensor_Type = typename Parent_Tns_Node_Type::Tensor_Type;

  using Tns_Node_Type::mTnsX;
  using Tns_Node_Type::mGramian;
  using Tns_Node_Type::mKey;
  using Tns_Node_Type::mUpdated;

  I_TnsNode               *parent;
  Left_Node_Type           left;
  Right_Node_Type          right;
  ExprNode() : TnsNode<TnsSize>(),
               // mParLabelSetSize(0),
               // mParParLabelSetSize(0),
               // mParOffset(0),
               parent(nullptr),
               left(this),
               right(this)
  {
    static_assert(IsRoot,"Wrong expansion!");
  }

  template <std::size_t _ParLabelSetSize2, std::size_t _ParParLabelSetSize, std::size_t _RootSize2>
  friend struct ExprNode;

protected:
  template <std::size_t _ParLabelSetSize2, std::size_t _ParParLabelSetSize, std::size_t _RootSize2>
  ExprNode(ExprNode<_ParLabelSetSize2,_ParParLabelSetSize,_RootSize2> *parent_) :
                        TnsNode<TnsSize>(),
                        // mParLabelSetSize(_ParLabelSetSize2),
                        // mParParLabelSetSize(_ParParLabelSetSize),
                        // mParOffset(reinterpret_cast<char*>(parent_) - reinterpret_cast<char*>(this)),
                        parent(parent_),
                        left(this),
                        right(this)
  {
    static_assert(_ParLabelSetSize2 == ParLabelSetSize, "Wrong expansion!");
    static_assert(_RootSize2         == RootSize,        "Wrong expansion!");
    static_assert(_ParLabelSetSize2 != 0,                "Wrong expansion!");
  }

public:

```

```
    void *TnsDims()
    {
      return &mTnsDims;
    }

    void *LabelSet()
    {
      return &mLabelSet;
    }

    void *DeltaSet()
    {
      return &mDeltaSet;
    }

    I_TnsNode *SearchKey (int const aKey)
    {
      auto length = LabelSetSize;
      if (length > 1)
      {
        if (aKey <= mKey)
          return Left()->SearchKey(aKey);
        else
          return Right()->SearchKey(aKey);
      }
      else
      {
        if (aKey == mKey)
          return this;
        else
          return nullptr;
      }
    }

    I_TnsNode *Parent()   { return IsRoot ? nullptr  : parent; }
    I_TnsNode *Left  ()   { return IsLeaf ? nullptr  : &left;  }
    I_TnsNode *Right ()   { return IsLeaf ? nullptr  : &right; }

    /*
     * Computes the N mode product of a tensor with a matrix.
     *
     * @tparam _LabelSetSize        Size of the LabelSet of @c this node.
     * @tparam _ParLabelSetSize     Size of the LabelSet of the parent node.
     * @tparam _RootSize            Size of the LabelSet of the root node.
     * @param  aFactor     [in]     Factor, of type @c FactorDimTree, to use
     *                              for tree mode N product.
     * @param  aNumFactors [in]     Total number of factors.
     * @param  id          [in]     Identification of the updating factor @c aFactor.
     * @param  aGramian    [in,out] Gramian matrix of the node.
     * @param  aDeltaSet   [in,out] Label set of the brother node after the N-mode product.
     * @param  aTnsDims    [in,out] @c stl with the dimensions of the final Tensor.
     *
     * @returns A @c TnsNode of @c ParTnsSize is returned.
     */
    Parent_Tensor_Type TreeMode_N_Product( FactorDimTree                        *const  aFactor,
                                           int                                   const  aNumFactors,
                                           int                                   const  id,
                                           std::array<int,ParTnsSize>            const &aTnsDims,
                                           Hessian_Type                                &aGramian,
                                           std::array<int,BrotherLabelSetSize>         &aDeltaSet     );

    /*
     * Interface of @c TTV product computation, between a tensor and a matrix.
     * @tparam _LabelSetSize        Size of the LabelSet of @c this node.
     * @tparam _ParLabelSetSize     Size of the LabelSet of the parent node.
     * @tparam _RootSize            Size of the LabelSet of the root node.
     * @param aFactor      [in]     Factor (of type @c FactorDimTree) to use for @c TTV product.
     * @param aNumFactors  [in]     Total number of factors.
     * @param id           [in]     Identification of the updating factor.
     * @param aDeltaSet    [in]     @c stl with the Label set of the brother @c ExprNode
     *                              after the @c TTV product of size @c BrotherLabelSetSize.
     * @param aX_partial   [in]     @c Tensor used for @c TTV product.
     * @param aTnsDims     [in,out] @c stl with the dimensions of the computed Tensor
     *                              of size @c ParTnsSize.
     * @param aGramian     [in,out] Gramian matrix of the @c ExprNode.
     *
     * @returns A @c TnsNode with size equal to @c TnsSize.
     */
    Tensor_Type TTVs                     ( FactorDimTree                        *const  aFactor,
                                           int                                   const  aNumFactors,
                                           int                                   const  id,
                                           std::array<int,BrotherLabelSetSize>   const &aDeltaSet,
                                           Parent_Tensor_Type                    const &aX_partial,
                                           std::array<int,ParTnsSize>            const &aTnsDims,
                                           Hessian_Type                                &aGramian     );

    /*
```

```
    * Computes the TTV product of a tensor with a matrix, using recursion.
    *
    * @tparam _LabelSetSize         Size of the LabelSet of @c this node.
    * @tparam _ParLabelSetSize      Size of the LabelSet of the parent node.
    * @tparam _RootSize             Size of the LabelSet of the root node.
    * @tparam DeltaSetSize          Size of the DeltaSset of this node.
    * @tparam ResTnsSize            Order of the resulting @c Tensor.
    * @tparam ResParTnsSize         Order of the parent's @c Tensor.
    * @param  it           [in]     Factor (of @c FactorDimTree type) to use for @c TTV product.
    * @param  aX_partial   [in]     @c Tensor for @c TTV product.
    * @param  aContractDim [in]     Dimension for @c TTV product, based on being a Left
    *                               or Right child.
    * @param  aTnsDims     [in,out] @c stl @c array with the dimensions of the computed Tensor
    *                               of size @c ResParTnsSize.
    * @param  aGramian     [in,out] Gramian matrix of the @c ExprNode.
    * @param  aX_result    [in,out] The result of TTV ( @c Tensor ) of size @c ResTnsSize.
    */
   template <std::size_t DeltaSetSize, std::size_t ResTnsSize, std::size_t ResParTnsSize>
   void       TTVs_util                ( FactorDimTree                          *const  it,
                                         Tensor<static_cast<int>(ResParTnsSize)>  const &aX_partial,
                                         int                                      const
  aContractDim,
                                         std::array<int,ResParTnsSize>            const &aTnsDims,
                                         Hessian_Type                                   &aGramian,
                                         Tensor<static_cast<int>(ResTnsSize)>           &aX_result
  );

   /*
    * Updates the factors in each node until computing the leaf nodes and their
    * @c Eigen @c Tensors. Based on the position of the node chooses to execute
    * @c TreeMode_N_Product or @c TTV. Works in recursive way.
    *
    * @tparam _LabelSetSize        Size of the LabelSet.
    * @tparam _ParLabelSetSize     Size of the LabelSet of the parent node.
    * @tparam _RootSize            Size of the LabelSet of the root node.
    * @param  aNumFactors [in]     Total number of factors.
    * @param  id          [in]     Identification of the updating factor.
    * @param  aFactor     [in,out] The factor to be updated.
    */
   void UpdateTree(int const aNumFactors, int const id, FactorDimTree *aFactor) override;
 };

 /*
  * typedef for zero order tensor.
  * @tparam TreeDim Tensor Order.
  */
 template <std::size_t _TreeDim>
 using NullExprNode = ExprNode<0,1,_TreeDim>;

 template <std::size_t _TnsSize>
 struct ExprTree
 {
   static_assert(_TnsSize >= 1, "Expansion problem in ExprTree!");
   static constexpr std::size_t TnsSize = _TnsSize;
   static constexpr bool         IsNull  = false;
   using  RootExprNode = ExprNode<TnsSize,0,TnsSize>;
   using  DataType     = typename RootExprNode::DataType;

   RootExprNode  root;
   template<typename Array>
   void Create( std::array<int,RootExprNode::LabelSetSize>       &aLabelSet,
                Array                                      const &aTnsDims,
                int                                        const  R,
                Tensor<static_cast<int>(TnsSize)>          const &aTnsX     )
   {
     constexpr bool IsLeaf = RootExprNode::IsLeaf;
     constexpr bool IsRoot = RootExprNode::IsRoot;

     static_assert(! (IsLeaf && IsRoot), "Tree expression with 1 dimension!" );

     constexpr std::size_t vHalf         = RootExprNode::DIM_HALF_SIZE;
     constexpr std::size_t vLabelSetSize = RootExprNode::LabelSetSize;

     root.mUpdated  = true;
     root.mTnsX     = aTnsX;
     std::copy(aTnsDims.begin(),               aTnsDims.begin()+TnsSize, root.mTnsDims.begin());
     root.mLabelSet = aLabelSet;
     root.mGramian.setConstant(1);

     // Create left child
     std::copy(aLabelSet.begin(),              aLabelSet.begin()+vHalf, root.left.mLabelSet.begin());
     std::copy(aLabelSet.end()-(TnsSize-vHalf), aLabelSet.end(),        root.left.mDeltaSet.begin());
     std::copy(aTnsDims.begin(),               aTnsDims.begin()+vHalf,  root.left.mTnsDims.begin()+1);
     root.left.mTnsDims.front() = R;
     RandomTensorGen(root.left.mTnsDims, root.left.mTnsX);

     // Create right child
```

```
    std::copy(aLabelSet.end()-(TnsSize-vHalf), aLabelSet.end(),
 root.right.mLabelSet.begin());
    std::copy(aLabelSet.begin(),              aLabelSet.begin()+vHalf,
 root.right.mDeltaSet.begin());
    std::copy(aTnsDims.begin()+vHalf,         aTnsDims.begin()+vLabelSetSize,
 root.right.mTnsDims.begin()+1);
    root.right.mTnsDims.front() = R;
    RandomTensorGen(root.right.mTnsDims, root.right.mTnsX);

    // Set key in root node
    root.mKey = root.left.mLabelSet[vHalf-1];

    Create(root.left.mTnsDims,  R, root.left);
    Create(root.right.mTnsDims, R, root.right);
  }

  template<typename Array, typename ExprNode>
  void Create( Array                 const &aTnsDims,
               [[maybe_unused]] int const  R,      // not used in leaf nodes.
               ExprNode                    &expr    )
  {
    static_assert(!std::is_same_v<ExprNode,NullExprNode<TnsSize», "Expansion problem!");

    using Expr_Node_Type = ExprNode;

    constexpr  bool IsLeaf              = Expr_Node_Type::IsLeaf;
    constexpr  std::size_t vExprTnsSize = Expr_Node_Type::TnsSize;

    if constexpr (!IsLeaf)
    {
      constexpr std::size_t vHalf         = Expr_Node_Type::DIM_HALF_SIZE;
      constexpr std::size_t vLabelSetSize = Expr_Node_Type::LabelSetSize;

      // Create left child
      std::copy(expr.mLabelSet.begin(),                      expr.mLabelSet.begin()+vHalf,
 expr.left.mLabelSet.begin());
      std::copy(expr.mLabelSet.end()-(vExprTnsSize-vHalf-1), expr.mLabelSet.end(),
 expr.left.mDeltaSet.begin());
      std::copy(aTnsDims.begin(),                            aTnsDims.begin()+vHalf+1,
 expr.left.mTnsDims.begin());
      RandomTensorGen(expr.left.mTnsDims, expr.left.mTnsX);

      // Create right child
      std::copy(expr.mLabelSet.end()-(vExprTnsSize-vHalf-1), expr.mLabelSet.end(),
 expr.right.mLabelSet.begin());
      std::copy(expr.mLabelSet.begin(),                      expr.mLabelSet.begin()+vHalf,
 expr.right.mDeltaSet.begin());
      std::copy(aTnsDims.begin()+vHalf+1,                    aTnsDims.begin()+vLabelSetSize+1,
 expr.right.mTnsDims.begin()+1);
      expr.right.mTnsDims.front() = R;
      RandomTensorGen(expr.right.mTnsDims,   expr.right.mTnsX);

      expr.mKey = expr.left.mLabelSet[vHalf-1];

      Create(expr.left.mTnsDims,  R, expr.left);
      Create(expr.right.mTnsDims, R, expr.right);
    }
    else
    {
      expr.mKey = expr.mLabelSet[0];
    }
  }

};

/*
 * ExprTree with no nodes.
 */
template <>
struct ExprTree<0>
{
  static constexpr bool IsNull = true;
};

struct FactorDimTree : public Factor<Tensor<2»
{
    TnsNode<1>         *leaf;

    FactorDimTree() : leaf(nullptr)
    { }
};

template <std::size_t _LabelSetSize, std::size_t _ParLabelSetSize, std::size_t _RootSize>
typename ExprNode<_LabelSetSize,_ParLabelSetSize,_RootSize>::Parent_Tensor_Type
ExprNode<_LabelSetSize,_ParLabelSetSize,_RootSize>::TreeMode_N_Product( FactorDimTree
     *const  aFactor,
                                                                        int
```

```
        const  aNumFactors,
                                                                            int
        const  id,
                                                                            std::array<int,ParTnsSize>
        const &aTnsDims,
                                                                            Hessian_Type
              &aGramian,
    std::array<int,BrotherLabelSetSize>       &aDeltaSet     )
{
  int                     vContractDim;
  Parent_Tensor_Type      vX_partial;
  FactorDimTree          *it;
  constexpr std::size_t   aDeltaSetSize = BrotherLabelSetSize-1;

  static_assert(!IsRoot, "TreeMode_N_Product() must not be called on root node!");

  const std::size_t                    R = aFactor->gramian.dimension(0);
  std::array<int,ParTnsSize>           vTnsDims;
  std::array<Eigen::IndexPair<int>, 1> product_dims;

  if (this == Parent()->Left())
  {
    it             = aFactor+aNumFactors-id-1;
    vContractDim = RootSize-1;
    std::copy(mDeltaSet.begin(), mDeltaSet.begin()+aDeltaSetSize, aDeltaSet.begin());

    std::copy(aTnsDims.begin(), aTnsDims.end()-1, vTnsDims.begin()+1);
    vTnsDims.front() = R;
  }
  else
  {
    it             = aFactor-id;
    vContractDim = 0;
    std::copy(mDeltaSet.end()-aDeltaSetSize, mDeltaSet.end(), aDeltaSet.begin());
    std::copy(aTnsDims.begin()+1, aTnsDims.end(), vTnsDims.begin()+1);
    vTnsDims.front() = R;
  }

  aGramian     = (*it).gramian;
  product_dims = { Eigen::IndexPair<int>(0, vContractDim) };

  vX_partial.resize(vTnsDims);
  vX_partial = it->factor.contract(*reinterpret_cast<Parent_Tensor_Type *>(Parent()->TensorX()),
 product_dims);

  return vX_partial;
}

template <std::size_t _LabelSetSize, std::size_t _ParLabelSetSize, std::size_t _RootSize>
template <std::size_t DeltaSetSize, std::size_t ResTnsSize, std::size_t ResParTnsSize>
void
ExprNode<_LabelSetSize,_ParLabelSetSize,_RootSize>::TTVs_util(  FactorDimTree
  *const  it,
    Tensor<static_cast<int>(ResParTnsSize)>  const &aX_partial,
                                                                    int
     const  aContractDim,
                                                                    std::array<int,ResParTnsSize>
     const &aTnsDims,
                                                                    Hessian_Type
           &aGramian,
                                                                    Tensor<static_cast<int>(ResTnsSize)>
           &aX_result    )
{
  constexpr int _ResParTnsSize = ResParTnsSize - 1;

  using Result_Tensor_Type = Tensor<_ResParTnsSize>;

  std::array<int,static_cast<std::size_t>(_ResParTnsSize)> vTnsDims;
  const std::size_t R = aGramian.dimension(0);

  aGramian *= (*it).gramian;   // Hadamard Product.

  // Allocate the reduction result
  if (this == Parent()->Right())  // Right Child
  {
    std::copy(aTnsDims.end()-_ResParTnsSize+1, aTnsDims.end(), vTnsDims.begin()+1);
    vTnsDims.front() = R;
  }
  else                            // Left Child
  {
    // 1o TTV
    if constexpr (IsFirstChild)
    {
      // First Child and First TTV
      if constexpr (DeltaSetSize == BrotherLabelSetSize - 1)
```

```
      {
        std::copy(aTnsDims.begin(), aTnsDims.begin()+_LabelSetSize, vTnsDims.begin()+1);
        if constexpr (_ResParTnsSize - _LabelSetSize > 0)
          std::copy(aTnsDims.end() - (_ResParTnsSize - _LabelSetSize), aTnsDims.end(), vTnsDims.begin()+_LabelSetSize+1);
        vTnsDims.front() = R;
      }
      else
      {
        std::copy(aTnsDims.begin(), aTnsDims.begin()+_LabelSetSize+1, vTnsDims.begin());
        if constexpr (_ResParTnsSize - _LabelSetSize > 0)
            std::copy(aTnsDims.end() - (_ResParTnsSize - _LabelSetSize) + 1, aTnsDims.end(), vTnsDims.begin()+_LabelSetSize+1);
      }
    }
    else
    {
      std::copy(aTnsDims.begin(), aTnsDims.begin()+_LabelSetSize+1, vTnsDims.begin());
      if constexpr (_ResParTnsSize - _LabelSetSize > 0)
          std::copy(aTnsDims.end() - (_ResParTnsSize - _LabelSetSize) + 1, aTnsDims.end(), vTnsDims.begin()+_LabelSetSize+1);
    }
  }

  // Apply reduction
  Result_Tensor_Type  vTnsX;
  vTnsX.resize(vTnsDims);

  TensorPartialProduct_R<ResParTnsSize,_ResParTnsSize>(aX_partial, it->factor, 0, aContractDim, &vTnsX);

  // Update tensor orders
  if constexpr (_ResParTnsSize > ResTnsSize)
  {
    TTVs_util<DeltaSetSize-1,ResTnsSize>(it+1, vTnsX, aContractDim, vTnsDims, aGramian, aX_result);
  }
  else
  {
    aX_result = vTnsX;
  }
}

template <std::size_t _LabelSetSize, std::size_t _ParLabelSetSize, std::size_t _RootSize>
typename ExprNode<_LabelSetSize,_ParLabelSetSize,_RootSize>::Tensor_Type
ExprNode<_LabelSetSize,_ParLabelSetSize,_RootSize>::TTVs( FactorDimTree                       *const aFactor,
                                                          int                                 const aNumFactors,
                                                          int                                 const id,
                                                          std::array<int,BrotherLabelSetSize> const &aDeltaSet,
                                                          Parent_Tensor_Type                  const &aX_partial,
                                                          std::array<int,ParTnsSize>          const &aTnsDims,
                                                          Hessian_Type                        &aGramian     )
{
  using Result_Tensor_Type = ExprNode<_LabelSetSize,_ParLabelSetSize,_RootSize>::Tensor_Type;

  constexpr std::size_t vDeltaSetSize = IsFirstChild ? BrotherLabelSetSize - 1 : BrotherLabelSetSize;

  static_assert( (vDeltaSetSize+TnsSize) == ParTnsSize, "Wrong call!" );

  int  vContractDim = (this == Parent()->Right()) ? 0 : LabelSetSize;

  FactorDimTree *it;

  it = aFactor;               // TODO  There is no range check for it !
  assert(aDeltaSet[0] - 1 < aNumFactors);
  it = aFactor + (aDeltaSet[0] - id - 1);

  Result_Tensor_Type vResTensor;

  TTVs_util<vDeltaSetSize,TnsSize>(it, aX_partial, vContractDim, aTnsDims, aGramian, vResTensor);

  return vResTensor;
}

template <std::size_t _LabelSetSize, std::size_t _ParLabelSetSize, std::size_t _RootSize>
void ExprNode<_LabelSetSize,_ParLabelSetSize,_RootSize>::UpdateTree(int const aNumFactors, int const id, FactorDimTree *aFactor)
{
  if (mUpdated)  // Root is always mUpdated == true
  {
    if constexpr (!IsRoot)
```

```
        left.SetOutdated();
      else
      {
        if constexpr (!IsLeaf)
        {
          if (left.mUpdated)
            left.SetOutdated();
          else if (right.mUpdated)
            right.SetOutdated();
        }
      }
    }
    else
    {
      Parent()->UpdateTree(aNumFactors, id, aFactor);
      int R   = mTnsDims.back();

      Hessian_Type                       vGramian(R,R);
      Tensor_Type                        vX_temp;
      Parent_Tensor_Type                 vX_partial;

      std::array<int, ParTnsSize>          vTnsDims;
      std::array<int, BrotherLabelSetSize> vDeltaSet;

      std::size_t vDeltaSetSize = (IsFirstChild) ? BrotherLabelSetSize-1 : BrotherLabelSetSize;

      vTnsDims = *reinterpret_cast<std::array<int, ParTnsSize> *>(Parent()->TnsDims());

      if constexpr (IsFirstChild)
      {  // Case: leaf is a child of the root: mode n product is required
        vX_partial = TreeMode_N_Product(aFactor, aNumFactors, id, vTnsDims, vGramian, vDeltaSet);
      }
      else
      {
        std::copy(mDeltaSet.begin(), mDeltaSet.begin()+vDeltaSetSize, vDeltaSet.begin());
        vX_partial = *reinterpret_cast<Parent_Tensor_Type *>(Parent()->TensorX());
        vGramian   = Parent()->Gramian();
      }

      if constexpr (TnsSize == ParTnsSize)
      {
        vX_temp = vX_partial;
      }
      else if constexpr (!IsRoot)
      {
        if (vDeltaSetSize > 0)
        {
          vX_temp = TTVs(aFactor, aNumFactors, id, vDeltaSet, vX_partial, vTnsDims, vGramian); // TODO check for bad allocation
        }   // TODO check all paths
      }
      else
      {
        // TODO Now???
      }

      // Update the current tree node
      mTnsX    = vX_temp;
      mGramian = vGramian;
      mUpdated = true;
    }
  }

  template<std::size_t _TreeDim>
  I_TnsNode *search_leaf(int const key, ExprTree<_TreeDim> &tree)
  {
    if constexpr (_TreeDim >= 0)
      return tree.root.SearchKey(key);
    else
      return nullptr;
  }

} // end namespace partensor

#endif // end of DIM_TREES_HPP
```

8.21 execution.hpp File Reference

```
#include <type_traits>
```

8.21.1 Detailed Description

execution namespace defines the execution policies that partensor library implements.

8.22 execution.hpp

Go to the documentation of this file.

```
#ifndef DOXYGEN_SHOULD_SKIP_THIS
#endif // DOXYGEN_SHOULD_SKIP_THIS
/********************************************************************/
#ifndef PARTENSOR_EXECUTION_HPP
#define PARTENSOR_EXECUTION_HPP

#include <type_traits>

namespace partensor::execution {
inline namespace v1 {

class sequenced_policy
{
public:
  // For internal use only
  static constexpr std::false_type __allow_unsequenced()
  {
    return std::false_type{};
  }

  static constexpr std::false_type __allow_vector()
  {
    return std::false_type{};
  }

  static constexpr std::false_type __allow_parallel()
  {
    return std::false_type{};
  }

  static constexpr std::false_type __allow_cuda()
  {
    return std::false_type{};
  }

  static constexpr std::false_type __allow_openmpi()
  {
    return std::false_type{};
  }

  static constexpr std::false_type __allow_openmp()
  {
    return std::false_type{};
  }
};

class parallel_policy
{
public:
  // For internal use only
  static constexpr std::false_type __allow_unsequenced()
  {
    return std::false_type{};
  }

  static constexpr std::false_type __allow_vector()
  {
    return std::false_type{};
  }

  static constexpr std::true_type __allow_parallel()
  {
    return std::true_type{};
  }

  static constexpr std::false_type __allow_cuda()
  {
    return std::false_type{};
  }

  static constexpr std::false_type __allow_openmpi()
  {
```

```
    return std::false_type{};
  }

  static constexpr std::false_type __allow_openmp()
  {
    return std::false_type{};
  }
};

class parallel_unsequenced_policy
{
public:
  // For internal use only
  static constexpr std::true_type __allow_unsequenced()
  {
      return std::true_type{};
  }

  static constexpr std::true_type __allow_vector()
  {
    return std::true_type{};
  }

  static constexpr std::true_type __allow_parallel()
  {
    return std::true_type{};
  }

  static constexpr std::false_type __allow_cuda()
  {
    return std::false_type{};
  }

  static constexpr std::false_type __allow_openmpi()
  {
    return std::false_type{};
  }

  static constexpr std::false_type __allow_openmp()
  {
    return std::false_type{};
  }
};

class unsequenced_policy
{
public:
  // For internal use only
  static constexpr std::true_type __allow_unsequenced()
  {
    return std::true_type{};
  }
  static constexpr std::true_type __allow_vector()
  {
    return std::true_type{};
  }
  static constexpr std::false_type __allow_parallel()
  {
    return std::false_type{};
  }

  static constexpr std::false_type __allow_cuda()
  {
    return std::false_type{};
  }

  static constexpr std::false_type __allow_openmpi()
  {
    return std::false_type{};
  }

  static constexpr std::false_type __allow_openmp()
  {
    return std::false_type{};
  }
};

class cuda_policy
{
public:
  // For internal use only
  static constexpr std::false_type __allow_unsequenced()
  {
    return std::false_type{};
  }

  static constexpr std::false_type __allow_vector()
```

```
  {
    return std::false_type{};
  }

  static constexpr std::false_type __allow_parallel()
  {
    return std::false_type{};
  }

  static constexpr std::true_type __allow_cuda()
  {
    return std::true_type{};
  }

  static constexpr std::false_type __allow_openmpi()
  {
    return std::false_type{};
  }

  static constexpr std::false_type __allow_openmp()
  {
    return std::false_type{};
  }
};

class openmpi_policy
{
public:
  // For internal use only
  static constexpr std::false_type __allow_unsequenced()
  {
    return std::false_type{};
  }

  static constexpr std::false_type __allow_vector()
  {
    return std::false_type{};
  }

  static constexpr std::false_type __allow_parallel()
  {
    return std::false_type{};
  }

  static constexpr std::false_type __allow_cuda()
  {
    return std::false_type{};
  }

  static constexpr std::true_type __allow_openmpi()
  {
    return std::true_type{};
  }

  static constexpr std::false_type __allow_openmp()
  {
    return std::false_type{};
  }
};

class openmp_policy
{
public:
  // For internal use only
  static constexpr std::false_type __allow_unsequenced()
  {
    return std::false_type{};
  }

  static constexpr std::false_type __allow_vector()
  {
    return std::false_type{};
  }

  static constexpr std::false_type __allow_parallel()
  {
    return std::false_type{};
  }

  static constexpr std::false_type __allow_cuda()
  {
    return std::false_type{};
  }

  static constexpr std::false_type __allow_openmpi()
  {
    return std::false_type{};
```

```
  }

  static constexpr std::true_type __allow_openmp()
  {
    return std::true_type{};
  }
};

constexpr sequenced_policy            seq{};
constexpr parallel_policy             par{};
constexpr parallel_unsequenced_policy par_unseq{};
constexpr unsequenced_policy          unseq{};
constexpr cuda_policy                 cuda{};
constexpr openmpi_policy              mpi{};
constexpr openmp_policy               omp{};

template <class T>
struct is_execution_policy : std::false_type
{ };

template <>
struct is_execution_policy<sequenced_policy> : std::true_type
{   };

template <>
struct is_execution_policy<parallel_policy> : std::true_type
{ };

template <>
struct is_execution_policy<parallel_unsequenced_policy> : std::true_type
{ };

template <>
struct is_execution_policy<unsequenced_policy> : std::true_type
{ };

template <>
struct is_execution_policy<cuda_policy> : std::true_type
{ };

template <>
struct is_execution_policy<openmpi_policy> : std::true_type
{ };

template <>
struct is_execution_policy<openmp_policy> : std::true_type
{ };

template <class T>
constexpr bool is_execution_policy_v = is_execution_policy<T>::value;

template <typename P>
using execution_policy_t = std::remove_cv_t<std::remove_reference_t<P»;

} // v1

namespace internal
{
template <class ExecPolicy, class T>
using enable_if_execution_policy = typename
    std::enable_if<partensor::execution::is_execution_policy<typename
    std::decay<ExecPolicy>::type>::value,T>::type;
} // namespace internal

} // partensor::execution

#endif //PARTENSOR_EXECUTION_HPP
```

8.23 Gtc.hpp File Reference

```
#include "PARTENSOR_basic.hpp"
#include "PartialCwiseProd.hpp"
#include "MTTKRP.hpp"
#include "NesterovMNLS.hpp"
#include "Normalize.hpp"
#include "Timers.hpp"
#include "ReadWrite.hpp"
```

Functions

- template< std::size_t _TnsSize, typename ExecutionPolicy >
  execution::internal::enable_if_execution_policy< ExecutionPolicy, SparseStatus< _TnsSize, execution::execution_policy_t< ExecutionPolicy >, SparseDefaultValues > > gtc (ExecutionPolicy &&, Matrix const &Ratings_Base_T, SparseOptions< _TnsSize, execution::execution_policy_t< ExecutionPolicy >, SparseDefaultValues > const &options)
- template< std::size_t _TnsSize, typename ExecutionPolicy >
  execution::internal::enable_if_execution_policy< ExecutionPolicy, SparseStatus< _TnsSize, execution::execution_policy_t< ExecutionPolicy >, SparseDefaultValues > > gtc (ExecutionPolicy &&, SparseOptions< _TnsSize, execution::execution_policy_t< ExecutionPolicy >, SparseDefaultValues > const &options)

8.23.1 Detailed Description

Implements the General Tensor Completion(gtc). Make use of `spdlog` library in order to write output in a log file in `"../log"`. In case of using parallelism with mpi, then the functions from `GtcMpi.hpp` will be called.

8.23.2 Function Documentation

8.23.2.1 gtc() [1/2]

```
execution::internal::enable_if_execution_policy< ExecutionPolicy, SparseStatus< _TnsSize,
execution::execution_policy_t< ExecutionPolicy >, SparseDefaultValues > > partensor::gtc (
            ExecutionPolicy && ,
            Matrix const & Ratings_Base_T,
            SparseOptions< _TnsSize, execution::execution_policy_t< ExecutionPolicy >, SparseDefaultValues > const & options )
```

Interface of General Tensor Completion(gtc), with the use of an `Execution Policy`, which can be either `sequential`, `parallel` with the use of `MPI`, or `parallel` with the use of `OpenMP`. In order to choose a policy, type `execution::seq`, `execution::mpi` or `execution::omp`. Default value is `sequential`, in case no `ExecutionPolicy` is passed.

Template Parameters

ExecutionPolicy	Type of `stl Execution Policy` (sequential, parallel-mpi).
*Tensor_*	Type(data type and order) of input Tensor. `Tensor_` must be `partensor::Tensor<order>`, where `order` must be in range of [3-8].

Parameters

tnsX	[in] The given Tensor to be factorized of `Tensor_` type, with `double` data.
R	[in] The rank of decomposition.

Returns

An object of type `Status`, containing the results of the algorithm.

8.23.2.2 gtc() [2/2]

```
execution::internal::enable_if_execution_policy< ExecutionPolicy, SparseStatus< _TnsSize,
execution::execution_policy_t< ExecutionPolicy >, SparseDefaultValues > > partensor::gtc (
            ExecutionPolicy && ,
            SparseOptions< _TnsSize, execution::execution_policy_t< ExecutionPolicy >, Sparse←
DefaultValues > const & options )
```

Interface of General Tensor Completion(gtc), with the use of an `Execution Policy`, which can be either `sequential`, `parallel` with the use of `MPI`, or `parallel` with the use of `OpenMP`. In order to choose a policy, type `execution::seq`, `execution::mpi` or `execution::omp`. Default value is `sequential`, in case no `ExecutionPolicy` is passed.

Template Parameters

ExecutionPolicy	Type of `stl Execution Policy` (sequential, parallel-mpi).
*Tensor_*	Type(data type and order) of input Tensor. `Tensor_` must be `partensor::Tensor<order>`, where `order` must be in range of [3-8].

Parameters

tnsX	[in] The given Tensor to be factorized of `Tensor_` type, with `double` data.
R	[in] The rank of decomposition.

Returns

An object of type `Status`, containing the results of the algorithm.

8.24 Gtc.hpp

Go to the documentation of this file.

```
#ifndef DOXYGEN_SHOULD_SKIP_THIS
#endif // DOXYGEN_SHOULD_SKIP_THIS
/***********************************************************************/
#ifndef PARTENSOR_GTC_HPP
#define PARTENSOR_GTC_HPP

#include "PARTENSOR_basic.hpp"
#include "PartialCwiseProd.hpp"
#include "MTTKRP.hpp"
#include "NesterovMNLS.hpp"
#include "Normalize.hpp"
#include "Timers.hpp"
#include "ReadWrite.hpp"

namespace partensor
{
  inline namespace v1
  {
    namespace internal
    {
```

```
//template <typename ExecutionPolicy, typename Tensor>
//execution::internal::enable_if_execution_policy<ExecutionPolicy,Tensor>
//Status gtc_f(ExecutionPolicy &&, Tensor const &tnsX, std::size_t rank);

/*
 * Includes the implementation of General Tensor Completion. Based on the given
 * parameters one of the overloaded operators will be called.
 */
template <std::size_t TnsSize_>
struct GTC_Base
{
  static constexpr std::size_t TnsSize    = TnsSize_;
  static constexpr std::size_t lastFactor = TnsSize - 1;
  using SparseTensor = typename partensor::SparseTensor<TnsSize_>;
  using DataType     = typename SparseTensorTraits<SparseTensor>::DataType;
  using MatrixType   = typename SparseTensorTraits<SparseTensor>::MatrixType;
  using Dimensions   = typename SparseTensorTraits<SparseTensor>::Dimensions;
  using SparseMatrix = typename SparseTensorTraits<SparseTensor>::SparseMatrixType;
  using LongMatrix   = typename SparseTensorTraits<SparseTensor>::LongMatrixType;
  using Constraints  = typename SparseTensorTraits<SparseTensor>::Constraints;
  using MatrixArray  = typename SparseTensorTraits<SparseTensor>::MatrixArray;
  using DoubleArray  = typename SparseTensorTraits<SparseTensor>::DoubleArray;
  using IntArray     = typename SparseTensorTraits<SparseTensor>::IntArray;
  template<int mode>
  void sort_ratings_base_util(Matrix       const &Ratings_Base_T,
                              SparseTensor       &tnsX,
                              IntArray     const &tnsDims,
                              long int     const  nnz)
   {
     Matrix ratings_base_temp = Ratings_Base_T;
     std::vector<std::vector<double» vectorized_ratings_base;
     vectorized_ratings_base.resize(nnz,std::vector<double>(TnsSize + 1));

     for (int rows = 0; rows < static_cast<int>(TnsSize) + 1; rows++)
     {
       for (int cols = 0; cols < nnz; cols++)
       {
         vectorized_ratings_base[cols][rows] = Ratings_Base_T(rows, cols);
       }
     }
     // Sort
     std::sort(vectorized_ratings_base.begin(), vectorized_ratings_base.end(), SortRows<TnsSize_, mode, double>);

     for (int rows = 0; rows < static_cast<int>(TnsSize) + 1; rows++)
     {
       for (int cols = 0; cols < nnz; cols++)
       {
         ratings_base_temp(rows, cols) = vectorized_ratings_base[cols][rows];
       }
     }

     FillSparseMatricization<TnsSize>(tnsX, nnz, ratings_base_temp, tnsDims, mode);

     if constexpr (mode+1 < TnsSize)
        sort_ratings_base_util<mode+1>(Ratings_Base_T, tnsX, tnsDims, nnz);
    }

    void sort_ratings_base(Matrix       const &Ratings_Base_T,
                           SparseTensor       &tnsX,
                           IntArray     const &tnsDims,
                           long int     const  nnz)
    {
      ReserveSparseTensor<TnsSize>(tnsX, tnsDims, nnz);
      sort_ratings_base_util<0>(Ratings_Base_T, tnsX, tnsDims, nnz);
    }
  };

 template <std::size_t TnsSize_, typename ExecutionPolicy = execution::sequenced_policy>
 struct GTC : public GTC_Base<TnsSize_>
 {
   using          GTC_Base<TnsSize_>::TnsSize;
   using          GTC_Base<TnsSize_>::lastFactor;
   using typename GTC_Base<TnsSize_>::Dimensions;
   using typename GTC_Base<TnsSize_>::MatrixArray;
   using typename GTC_Base<TnsSize_>::DataType;
   using typename GTC_Base<TnsSize_>::SparseTensor;
   using typename GTC_Base<TnsSize_>::IntArray;
   using typename GTC_Base<TnsSize_>::LongMatrix;

   using Options = partensor::SparseOptions<TnsSize_,execution::sequenced_policy,SparseDefaultValues>;
   using Status  = partensor::SparseStatus<TnsSize_,execution::sequenced_policy,SparseDefaultValues>;

   // Variables that will be used in gtc implementations.
   struct Member_Variables
```

```
        {
          MatrixArray  factors_T;
          MatrixArray  factor_T_factor;
          MatrixArray  mttkrp_T;
          IntArray     tnsDims;
          std::array<std::array<int, TnsSize_ -1>, TnsSize_> offsets;

          MatrixArray  norm_factors_T;
          MatrixArray  old_factors_T;

          Matrix       cwise_factor_product;
          SparseTensor tnsX;
          int          rank;

          Member_Variables() = default;

          Member_Variables(int R, IntArray dims) :  tnsDims(dims),
                                                    rank(R)
          {}

          Member_Variables(Member_Variables const &) = default;
          Member_Variables(Member_Variables      &&) = default;

          Member_Variables &operator=(Member_Variables const &) = default;
          Member_Variables &operator=(Member_Variables      &&) = default;
        };

        /*
         * In case option variable @c writeToFile is enabled, then, before the end
         * of the algorithm, it writes the resulted factors in files, whose
         * paths are specified before compiling in @ options.final_factors_path.
         *
         * @param  st  [in] Struct where the returned values of @c Gtc are stored.
         */
        void writeFactorsToFile(Status const &st)
        {
          std::size_t size;
          for(std::size_t i=0; i<TnsSize; ++i)
          {
            size = st.factors[i].rows() * st.factors[i].cols();
            partensor::write(st.factors[i],
                             st.options.final_factors_paths[i],
                             size);
          }
        }

        /*
         * Compute the cost function value at the end of each outer iteration
         * based on the last factor.
         *
         * @param  mv  [in]      Struct where ALS variables are stored.
         * @param  st  [in,out] Struct where the returned values of @c Gtc are stored.
         *                       In this case the cost function value is updated.
         */
        void cost_function(Member_Variables const &mv,
                           Status                 &st)
        {
          Matrix temp_R_1(mv.rank, 1);
          double temp_1_1 = 0;
          st.f_value = 0;

          std::array<int,TnsSize-1> offsets;
          offsets[0] = 1;
          for (int j = 1; j < static_cast<int>(TnsSize) - 1; j++)
          {
            offsets[j] = offsets[j - 1] * mv.tnsDims[j-1];
          }

          for (long int i = 0; i < mv.tnsX[lastFactor].outerSize(); ++i)
          {
              int row = 0;
              for (SparseMatrix::InnerIterator it(mv.tnsX[lastFactor], i); it; ++it)
              {
                  temp_R_1 = mv.factors_T[lastFactor].col(it.col());
                  // Select rows of each factor an compute the Hadamard product of the respective row of
    the Khatri-Rao product, and the row of factor A_N.
                  for (int mode_i = static_cast<int>(TnsSize) - 2; mode_i >= 0; mode_i--)
                  {
                      row      = ((it.row()) / offsets[mode_i]) % (mv.tnsDims[mode_i]);
                      temp_R_1 = temp_R_1.cwiseProduct(mv.factors_T[mode_i].col(row));
                  }
                  temp_1_1 = it.value() - temp_R_1.sum();
                  st.f_value += temp_1_1 * temp_1_1;
              }
          }
        }

```

```
    /*
     * Compute the cost function value at the end of each outer iteration
     * based on the last accelerated factor.
     *
     * @param  mv                [in] Struct where ALS variables are stored.
     * @param  accel_factors     [in] Accelerated factors.
     *
     * @returns The cost function calculated with the accelerated factors.
     */
    double accel_cost_function(Member_Variables       const &mv,
                               MatrixArray            const &accel_factors)
    {
      Matrix temp_R_1(mv.rank, 1);
      double temp_1_1 = 0;
      double f_value = 0;

      std::array<int,TnsSize-1> offsets;
      offsets[0] = 1;
      for (int j = 1; j < static_cast<int>(TnsSize) - 1; j++)
      {
        offsets[j] = offsets[j - 1] * mv.tnsDims[j-1];
      }

      for (long int i = 0; i < mv.tnsX[lastFactor].outerSize(); ++i)
      {
        int row = 0;
        for (SparseMatrix::InnerIterator it(mv.tnsX[lastFactor], i); it; ++it)
        {
            temp_R_1 = accel_factors[lastFactor].col(it.col());
            // Select rows of each factor an compute the Hadamard product of the respective row of the Khatri-Rao product, and the row of factor A_N.
            // temp_R_1  = A_N(i_N,:) .* ... .* A_2(i_2,:) .* A_1(i_1,:)
            for (int mode_i = static_cast<int>(TnsSize) - 2; mode_i >= 0; mode_i--)
            {
                row      = ((it.row()) / offsets[mode_i]) % (mv.tnsDims[mode_i]);
                temp_R_1 = temp_R_1.cwiseProduct(accel_factors[mode_i].col(row));
            }
            temp_1_1 = it.value() - temp_R_1.sum();
            f_value += temp_1_1 * temp_1_1;
        }
      }
      return f_value;
    }

    void calculate_offsets(Member_Variables &mv)
    {
      for (int idx = 0; idx < static_cast<int>(TnsSize); idx++)
      {
        mv.offsets[idx][0] = 1;
        for (int j = 1, mode = 0; j < static_cast<int>(TnsSize) - 1; j++, mode++)
        {
          if (idx == mode)
          {
            mode++;
          }
          mv.offsets[idx][j] = mv.offsets[idx][j - 1] * mv.tnsDims[mode];
        }
      }
    }

    void unconstraint_update(int               const  idx,
                             Member_Variables       &mv,
                             Status                 &st)
    {
        int r = mv.rank;

        Matrix eye = st.options.lambdas[idx] * Matrix::Identity(r, r);

        int last_mode = (idx == static_cast<int>(TnsSize) - 1) ? static_cast<int>(TnsSize) - 2 : static_cast<int>(TnsSize) - 1;

        Matrix MTTKRP_col(r, 1);
        Matrix temp_RxR(r, r);
        Matrix temp_R_1(r, 1);

        // Compute MTTKRP
        for (long int i = 0; i < mv.tnsX[idx].outerSize(); ++i)
        {
            MTTKRP_col.setZero();
            temp_RxR.setZero(); // is the Hadamard product of Grammians of the Factors, that correspond to the nnz elements of the Tensor.
            for (SparseMatrix::InnerIterator it(mv.tnsX[idx], i); it; ++it)
            {
                temp_R_1 = Matrix::Ones(r, 1);
                int row;
                // Select rows of each factor an compute the respective row of the Khatri-Rao product.
```

```
                 for (int mode_i = last_mode, kr_counter = static_cast<int>(TnsSize) - 2; mode_i >= 0 && kr_counter >= 0; mode_i--)
                 {
                     if (mode_i == idx)
                     {
                         continue;
                     }
                     row      = ((it.row()) / mv.offsets[idx][kr_counter]) % (mv.tnsDims[mode_i]);
                     temp_R_1 = temp_R_1.cwiseProduct(mv.factors_T[mode_i].col(row));
                     kr_counter--;
                 }
                 // Subtract from the previous row the respective row of W, according to relation (9).
                 MTTKRP_col.noalias() += it.value() * temp_R_1;
                 temp_RxR.noalias()   += temp_R_1 * temp_R_1.transpose();
             }
             mv.factors_T[idx].col(i) = (temp_RxR + eye).inverse() * MTTKRP_col;
         }
     }

     /*
      * Based on each factor's constraint, a different
      * update function is used at every outer iteration.
      *
      * Computes also factor^T * factor at the end.
      *
      * @param  idx [in]     Factor to be updated.
      * @param  mv  [in]     Struct where ALS variables are stored.
      * @param  st  [in,out] Struct where the returned values of @c Gtc are stored.
      *                      Updates the @c stl array with the factors.
      */
     void update_factor(int              const  idx,
                        Member_Variables       &mv,
                        Status                 &st    )
     {
       // Update factor
       switch ( st.options.constraints[idx] )
       {
         case Constraint::unconstrained:
         case Constraint::symmetric:
         {
           unconstraint_update(idx, mv, st);
           break;
         }
         case Constraint::nonnegativity:
         case Constraint::symmetric_nonnegativity:
         {
           int last_mode = (idx == static_cast<int>(TnsSize) - 1) ? static_cast<int>(TnsSize) - 2 : static_cast<int>(TnsSize) - 1;
           SparseMTTKRP(mv.tnsDims, mv.tnsX[idx], mv.factors_T, mv.rank, mv.offsets[idx], last_mode, idx, mv.mttkrp_T[idx]);

           // NesterovMNLS(mv.cwise_factor_product, mv.factors_T, mv.tnsDims, mv.tnsX[idx], mv.offsets[idx], st.options.max_nesterov_iter,
           //              st.options.lambdas[idx], idx, st.options.constraints[idx], mv.mttkrp_T[idx]);
           local_L::NesterovMNLS(mv.factors_T, mv.tnsDims, mv.tnsX[idx], mv.offsets[idx], st.options.max_nesterov_iter,
                        st.options.lambdas[idx], idx, mv.mttkrp_T[idx]);
           break;
         }
         default: // in case of Constraint::constant
           break;
       }

       // Compute A^T * A + B^T * B + ...
       st.factors[idx] = mv.factors_T[idx].transpose();
       if (st.options.constraints[idx] == Constraint::symmetric_nonnegativity || st.options.constraints[idx] == Constraint::symmetric)
       {
         for (std::size_t i=0; i<TnsSize; i++)
         {
           if (i != static_cast<std::size_t>(idx))
           {
             mv.factors_T[i] = mv.factors_T[idx];
           }
         }
       }
       mv.factor_T_factor[idx].noalias() = mv.factors_T[idx] * st.factors[idx];
     }

     /*
      * @brief Line Search Acceleration
      *
      * Performs an acceleration step on the updated factors, and keeps the accelerated factors
      * when the step succeeds. Otherwise, the acceleration step is ignored.
      * Line Search Acceleration reduces the number of outer iterations in the ALS algorithm.
```

```
     *
     * @note This implementation ONLY, if factors are of @c Matrix type.
     *
     * @param  mv  [in,out] Struct where ALS variables are stored.
     *                      In case the acceleration step is successful the Gramian
     *                      matrices of factors are updated.
     * @param  st  [in,out] Struct where the returned values of @c Gtc are stored.
     *                      If the acceleration succeeds updates @c factors
     *                      and cost function value.
     *
     */
    void line_search_accel(Member_Variables &mv,
                           Status           &st)
    {
      double       f_accel    = 0.0; // Objective Value after the acceleration step
      double const accel_step = pow(st.ao_iter+1,(1.0/(st.options.accel_coeff)));

      MatrixArray  accel_factors_T;
      MatrixArray  accel_gramians;

      for(std::size_t i=0; i<TnsSize; ++i)
      {
        accel_factors_T[i] = mv.old_factors_T[i] + accel_step * (mv.factors_T[i] - mv.old_factors_T[i]);
        accel_gramians[i]  = accel_factors_T[i] * accel_factors_T[i].transpose();
      }

      f_accel = accel_cost_function(mv, accel_factors_T);
      if (st.f_value > f_accel)
      {
        mv.factors_T        = accel_factors_T;
        mv.factor_T_factor = accel_gramians;
        st.f_value          = f_accel;
        Partensor()->Logger()->info("Acceleration Step SUCCEEDED! at iter: {}", st.ao_iter);
      }
      else
        st.options.accel_fail++;

      if (st.options.accel_fail==5)
      {
        st.options.accel_fail=0;
        st.options.accel_coeff++;
      }
    }

    /*
     * Sequential implementation of Alternating Least Squares (ALS) method.
     *
     * @param  R   [in]     The rank of decomposition.
     * @param  mv  [in]     Struct where ALS variables are stored and being updated
     *                      until a termination condition is true.
     * @param  st  [in,out] Struct where the returned values of @c Gtc are stored.
     */
    void aogtc(Member_Variables       &mv,
               Status                 &status)
    {
      for (std::size_t i=0; i<TnsSize; i++)
      {
        // mv.factors_T[i] = status.factors[i].transpose();
        // mv.factor_T_factor[i].noalias() = mv.factors_T[i] * status.factors[i];
        mv.factor_T_factor[i].noalias() = mv.factors_T[i] * mv.factors_T[i].transpose();
        mv.mttkrp_T[i] = Matrix(mv.rank, mv.tnsDims[i]);
      }

      // if(status.options.normalization)
      // {
      //   choose_normilization_factor(status, mv.all_orthogonal, mv.weight_factor);
      // }
      // Normalize(static_cast<int>(R), mv.factor_T_factor, status.factors);

      status.frob_tns          = (mv.tnsX[0]).squaredNorm();
      cost_function(mv, status);
      status.rel_costFunction = status.f_value/status.frob_tns;

      // ---- Loop until ALS converges ----
      while(1)
      {
        status.ao_iter++;
        Partensor()->Logger()->info("iter: {} -- fvalue: {} -- relative_costFunction: {}", status.ao_iter,
                                                    status.f_value, status.rel_costFunction);

        for (std::size_t i=0; i<TnsSize; i++)
        {
          mv.cwise_factor_product = PartialCwiseProd(mv.factor_T_factor, i);

          // Update factor
```

```
          update_factor(i, mv, status);
        }

        cost_function(mv, status);
        status.rel_costFunction = status.f_value/status.frob_tns;

        // if(status.options.normalization && !mv.all_orthogonal)
        //   Normalize(mv.weight_factor, static_cast<int>(R), mv.factor_T_factor, status.factors);

        // ---- Terminating condition ----
        if (status.rel_costFunction < status.options.threshold_error || status.ao_iter >= status.options.max_iter)
        {
          if(status.options.writeToFile)
            writeFactorsToFile(status);
          break;
        }

        if (status.options.acceleration)
        {
          mv.norm_factors_T = mv.factors_T;
          // ---- Acceleration Step ----
          if (status.ao_iter > 1)
            line_search_accel(mv, status);

          mv.old_factors_T = mv.norm_factors_T;
        }
      } // end of while
    }

    void initialize_factors(Member_Variables &mv,
                            Status           &status)
    {
      if(status.options.initialized_factors)
      {
        if(status.options.read_factors_from_file)
        {
          for(std::size_t i=0; i<TnsSize; ++i)
          {
            status.factors[i] = Matrix(mv.tnsDims[i], static_cast<int>(mv.rank));
            read( status.options.initial_factors_paths[i],
                  mv.tnsDims[i] * mv.rank,
                  0,
                  status.factors[i] );
          }
        }
        else
          status.factors = status.options.factorsInit;
      }
      else // produce estimate factors using uniform distribution with entries in [0,1].
        makeFactors(mv.tnsDims, status.options.constraints, mv.rank, status.factors);
    }

    Status operator()(Options const &options)
    {
      Status           status(options);
      Member_Variables mv(options.rank, options.tnsDims);

      long long int fileSize = (TnsSize + 1) * options.nonZeros;

      // Begin Load Balancing
            Matrix                                    Balanced_Ratings_Base_T(TnsSize + 1, options.nonZeros);
            std::array<std::vector<long int>, TnsSize> perm_tns_indices;

      Matrix Ratings_Base_T = Matrix(static_cast<int>(TnsSize+1), options.nonZeros);
      // Read the whole Tensor from a file
      read( options.ratings_path,
            fileSize,
            0,
            Ratings_Base_T );

            BalanceDataset<TnsSize>(options.nonZeros, options.tnsDims, Ratings_Base_T, perm_tns_indices, Balanced_Ratings_Base_T);

      // GTC_Base<TnsSize>::sort_ratings_base(Ratings_Base_T, mv.tnsX, options.tnsDims, options.nonZeros);
      GTC_Base<TnsSize>::sort_ratings_base(Balanced_Ratings_Base_T, mv.tnsX, options.tnsDims, options.nonZeros);
      Ratings_Base_T.resize(0,0);
      Balanced_Ratings_Base_T.resize(0,0);

      for (std::size_t i=0; i<TnsSize; i++)
      {
        mv.tnsX[i].makeCompressed();
      }

```

```
        calculate_offsets(mv);

        initialize_factors(mv, status);

        PermuteFactors<TnsSize>(status.factors, perm_tns_indices, mv.factors_T);

        aogtc(mv, status);

        // IF Depermute ....

        return status;
      }

      Status operator()(Matrix       const &Ratings_Base_T,
                        Options      const &options)
      {
        Status           status(options);
        Member_Variables mv(options.rank, options.tnsDims);

        // Begin Load Balancing
              Matrix                                    Balanced_Ratings_Base_T(TnsSize + 1,
   options.nonZeros);
              std::array<std::vector<long int>, TnsSize> perm_tns_indices;

              BalanceDataset<TnsSize>(options.nonZeros, options.tnsDims, Ratings_Base_T,
   perm_tns_indices, Balanced_Ratings_Base_T);

        ReserveSparseTensor<TnsSize>(mv.tnsX, options.tnsDims, options.nonZeros);
        // FillSparseTensor<TnsSize>(mv.tnsX, options.nonZeros, Ratings_Base_T, options.tnsDims);
        FillSparseTensor<TnsSize>(mv.tnsX, options.nonZeros, Balanced_Ratings_Base_T,
   options.tnsDims);
        // Ratings_Base_T.resize(0,0);
        Balanced_Ratings_Base_T.resize(0,0);

        for (std::size_t i=0; i<TnsSize; i++)
        {
          mv.tnsX[i].makeCompressed();
        }

        calculate_offsets(mv);

        // produce estimate factors using uniform distribution with entries in [0,1].
        initialize_factors(mv, status);

        PermuteFactors<TnsSize>(status.factors, perm_tns_indices, mv.factors_T);

        aogtc(mv, status);

        // IF Depermute ....

        return status;
      }
    };
  }  // namespace internal
}    // namespace v1
} // end namespace partensor

#if USE_MPI

#include "GtcMpi.hpp"
#endif /* USE_MPI */

#if USE_OPENMP

#include "GtcOpenMP.hpp"
#endif /* USE_OPENMP */

namespace partensor
{
  template <std::size_t _TnsSize, typename ExecutionPolicy>

   execution::internal::enable_if_execution_policy<ExecutionPolicy,SparseStatus<_TnsSize,execution::execution_policy_t<Exe
  gtc( ExecutionPolicy
     &&,
       SparseOptions<_TnsSize,execution::execution_policy_t<ExecutionPolicy>,SparseDefaultValues>
   const &options  )
  {
    using  ExPolicy = execution::execution_policy_t<ExecutionPolicy>;

    if constexpr (std::is_same_v<ExPolicy,execution::sequenced_policy>)
    {
      return internal::GTC<_TnsSize>()(options);
    }
    else if constexpr (std::is_same_v<ExPolicy,execution::openmpi_policy>)
    {
      return internal::GTC<_TnsSize,execution::openmpi_policy>()(options);
    }
```

```
    else if constexpr (std::is_same_v<ExPolicy,execution::openmp_policy>)
    {
      return internal::GTC<_TnsSize,execution::openmp_policy>()(options);
    }
    else
      return internal::GTC<_TnsSize>()(options);
  }

  /*
   * Interface of General Tensor Completion(gtc). Sequential Policy.
   *
   * @tparam Tensor_       Type(data type and order) of input Tensor.
   *                       @c Tensor_ must be @c partensor::Tensor<order>, where
   *                       @c order must be in range of @c [3-8].
   * @param  tnsX    [in] The given Tensor to be factorized of @c Tensor_ type,
   *                       with @c double data.
   * @param  R       [in] The rank of decomposition.
   *
   * @returns An object of type @c Status, containing the results of the algorithm.
   */
  template<std::size_t _TnsSize>
  auto gtc(SparseOptions<_TnsSize> const &options )
  {
    return internal::GTC<_TnsSize,execution::sequenced_policy>()(options);
  }

  template <std::size_t _TnsSize, typename ExecutionPolicy>

   execution::internal::enable_if_execution_policy<ExecutionPolicy,SparseStatus<_TnsSize,execution::execution_policy_t<Exe
  gtc( ExecutionPolicy
     &&,
       Matrix
   const &Ratings_Base_T,
       SparseOptions<_TnsSize,execution::execution_policy_t<ExecutionPolicy>,SparseDefaultValues>
   const &options  )
  {
    using  ExPolicy = execution::execution_policy_t<ExecutionPolicy>;

    if constexpr (std::is_same_v<ExPolicy,execution::sequenced_policy>)
    {
      return internal::GTC<_TnsSize>()(Ratings_Base_T,options);
    }
    else if constexpr (std::is_same_v<ExPolicy,execution::openmpi_policy>)
    {
      return internal::GTC<_TnsSize,execution::openmpi_policy>()(Ratings_Base_T,options);
    }
    else if constexpr (std::is_same_v<ExPolicy,execution::openmp_policy>)
    {
      return internal::GTC<_TnsSize,execution::openmp_policy>()(Ratings_Base_T,options);
    }
    else
      return internal::GTC<_TnsSize>()(Ratings_Base_T,options);
  }

  /*
   * Interface of General Tensor Completion(gtc). Sequential Policy.
   *
   * @tparam Tensor_       Type(data type and order) of input Tensor.
   *                       @c Tensor_ must be @c partensor::Tensor<order>, where
   *                       @c order must be in range of @c [3-8].
   * @param  tnsX    [in] The given Tensor to be factorized of @c Tensor_ type,
   *                       with @c double data.
   * @param  R       [in] The rank of decomposition.
   *
   * @returns An object of type @c Status, containing the results of the algorithm.
   */
  template<std::size_t _TnsSize>
  auto gtc(Matrix                             const &Ratings_Base_T,
           SparseOptions<_TnsSize>            const &options )
  {
    return internal::GTC<_TnsSize,execution::sequenced_policy>()(Ratings_Base_T,options);
  }

}

#endif // PARTENSOR_GTC_HPPP
```

8.25 GtcMpi.hpp File Reference

Classes

- struct GTC< TnsSize_, execution::openmpi_policy >

8.25.1 Detailed Description

Implements the Canonical Polyadic Decomposition(gtc) using `MPI`. Make use of `spdlog` library in order to write output in a log file in `"../log"`.

8.26 GtcMpi.hpp

Go to the documentation of this file.

```
#ifndef DOXYGEN_SHOULD_SKIP_THIS
#endif // DOXYGEN_SHOULD_SKIP_THIS
/*****************************************************************/
#if !defined(PARTENSOR_GTC_HPP)
#error "GTC_MPI can only included inside GTC"
#endif /* PARTENSOR_GTC_HPP */

namespace partensor
{

  inline namespace v1 {

    namespace internal {
      template<std::size_t TnsSize_>
      struct GTC<TnsSize_,execution::openmpi_policy> : public GTC_Base<TnsSize_>
      {
        using          GTC_Base<TnsSize_>::TnsSize;
        using          GTC_Base<TnsSize_>::lastFactor;
        using typename GTC_Base<TnsSize_>::Dimensions;
        using typename GTC_Base<TnsSize_>::MatrixArray;
        using typename GTC_Base<TnsSize_>::DataType;
        using typename GTC_Base<TnsSize_>::SparseTensor;
        using typename GTC_Base<TnsSize_>::LongMatrix;

        using IntArray         = typename SparseTensorTraits<SparseTensor>::IntArray;
        using CartCommunicator = partensor::cartesian_communicator; // From ParallelWrapper.hpp
        using CartCommVector   = std::vector<CartCommunicator>;
        using IntVector        = std::vector<int>;
        using Int2DVector      = std::vector <std::vector<int»;

        using Options = partensor::SparseOptions<TnsSize_,execution::openmpi_policy,SparseDefaultValues>;
        using Status  = partensor::SparseStatus<TnsSize_,execution::openmpi_policy,SparseDefaultValues>;

        // Variables that will be used in gtc implementations.
        struct Member_Variables
        {
          MPI_Communicator &world = Partensor()->MpiCommunicator(); // MPI_COMM_WORLD

          double          local_f_value;
          int             RxR;
          int             world_size;

          Int2DVector     displs_subTns;       // skipping dimension "rows" for each subtensor
          Int2DVector     displs_subTns_R;     // skipping dimension "rows" for each subtensor times R (
    for MPI communication purposes )
          Int2DVector     subTnsDims;          // dimensions of subtensor
          Int2DVector     subTnsDims_R;        // dimensions of subtensor times R ( for MPI communication
    purposes )
          Int2DVector     displs_local_update; // displacement in the local factor for update rows

          Int2DVector     send_recv_counts;    // rows to be communicated after update times R

          CartCommVector  layer_comm;
          CartCommVector  fiber_comm;

          IntArray        layer_rank;
          IntArray        fiber_rank;
          IntArray        rows_for_update;
          IntArray        tnsDims;

          MatrixArray     layer_factors;
          MatrixArray     layer_factors_T;
          MatrixArray     factors;
          MatrixArray     factors_T;
          MatrixArray     factor_T_factor;
          MatrixArray     local_factors_T;
          MatrixArray     norm_factors_T;
          MatrixArray     old_factors_T;

          Matrix          cwise_factor_product;
          Matrix          temp_matrix;
```

```
      Matrix          Ratings_Base_T;
      SparseTensor    subTns;

      int             rank;
      std::array<std::array<int, TnsSize-1>, TnsSize> offsets;

      /*
       * Calculates if the number of processors given from terminal
       * are equal to the processors in the implementation.
       *
       * @param procs [in] @c stl array with the number of processors per
       *                   dimension of the tensor.
       */
      void check_processor_avaliability(std::array<int, TnsSize> const &procs)
      {
        // MPI_Environment &env = Partensor()->MpiEnvironment();
        world_size = world.size();
        //  numprocs must be product of options.proc_per_mode
        if (std::accumulate(procs.begin(), procs.end(), 1,
                            std::multiplies<int>()) != world_size && world.rank() == 0) {
          Partensor()->Logger()->error("The product of the processors per mode must be equal to
{}\n", world_size);
          // env.abort(-1);
        }
      }

      Member_Variables() = default;
      Member_Variables(int R, IntArray dims, std::array<int, TnsSize> &procs) :  local_f_value(0.0),
                                                                                RxR(R*R),
displs_subTns(TnsSize),
displs_subTns_R(TnsSize),
subTnsDims(TnsSize),
subTnsDims_R(TnsSize),
displs_local_update(TnsSize),
send_recv_counts(TnsSize),
                                                                                tnsDims(dims),
                                                                                rank(R)
      {
        check_processor_avaliability(procs);
        layer_comm.reserve(TnsSize);
        fiber_comm.reserve(TnsSize);
      }

      Member_Variables(Member_Variables const &) = default;
      Member_Variables(Member_Variables      &&) = default;

      Member_Variables &operator=(Member_Variables const &) = default;
      Member_Variables &operator=(Member_Variables      &&) = default;
    };

    template<int mode>
    void sort_ratings_base_util(Matrix           const &Ratings_Base_T,
                                long int         const  nnz,
                                Member_Variables       &mv)
    {
      Matrix ratings_base_temp = Ratings_Base_T;
      std::vector<std::vector<double>> vectorized_ratings_base;
      vectorized_ratings_base.resize(nnz, std::vector<double>(TnsSize + 1));

      for (int rows = 0; rows < static_cast<int>(TnsSize) + 1; rows++)
      {
        for (int cols = 0; cols < nnz; cols++)
        {
          vectorized_ratings_base[cols][rows] = Ratings_Base_T(rows, cols);
        }
      }

      // Sort
      std::sort(vectorized_ratings_base.begin(), vectorized_ratings_base.end(), SortRows<TnsSize_,
mode, double>);

      for (int rows = 0; rows < static_cast<int>(TnsSize) + 1; rows++)
      {
        for (int cols = 0; cols < nnz; cols++)
        {
          ratings_base_temp(rows, cols) = vectorized_ratings_base[cols][rows];
        }
      }

      Dist_NNZ_sorted<TnsSize>(mv.subTns, nnz, mv.displs_subTns, mv.fiber_rank, ratings_base_temp,
mv.subTnsDims, mode);
```

```
      mv.subTns[mode].makeCompressed();

      if constexpr (mode+1 < TnsSize)
        sort_ratings_base_util<mode+1>(Ratings_Base_T, nnz, mv);
    }

    void sort_ratings_base(Matrix           const &Ratings_Base_T,
                           long int         const  nnz,
                           Member_Variables       &mv)
    {
      ReserveSparseTensor<TnsSize>(mv.subTns, mv.subTnsDims, mv.fiber_rank, mv.world_size, nnz);
      sort_ratings_base_util<0>(Ratings_Base_T, nnz, mv);
    }

    void NesterovMNLS(Member_Variables                   &mv,
                      Status                       const  &st,
                      int     const                        idx,
                      Matrix                             &MTTKRP_T)
    {
      double L, mu, q, alpha, new_alpha, beta, lambda;
      int iter = 0;

      Matrix grad_Y_T(mv.rank, mv.subTnsDims[idx][mv.fiber_rank[idx]]);
      Matrix grad_Y_local_T(mv.rank, mv.rows_for_update[idx]);

      Matrix Y_T(mv.rank, mv.subTnsDims[idx][mv.fiber_rank[idx]]);
      Matrix Y_local_T(mv.rank, mv.rows_for_update[idx]);

      Matrix new_A(mv.rank, mv.rows_for_update[idx]);
      Matrix A(mv.rank, mv.rows_for_update[idx]);
      Matrix Zero_Matrix = Matrix::Zero(mv.rank, mv.rows_for_update[idx]);

      ComputeEIG(mv.cwise_factor_product, L, mu);

      lambda = st.options.lambdas[idx];
      L      = L + lambda;
      q      = lambda / L;
      alpha  = 1;

      A         = mv.local_factors_T[idx];
      Y_T       = mv.layer_factors_T[idx]; // layer_factor
      Y_local_T = A;

      int last_mode = (idx == static_cast<int>(TnsSize) - 1) ? static_cast<int>(TnsSize) - 2 : static_cast<int>(TnsSize) - 1;

      Matrix temp_R_1(mv.rank, 1);
      while (1)
      {
        grad_Y_T.setZero();

        if (iter >= st.options.max_nesterov_iter)
        {
          break;
        }

        // Compute grad_Y
        Matrix temp_col = Matrix::Zero(mv.rank, 1);
        for (long int i = 0; i < mv.subTns[idx].outerSize(); ++i)
        {
          temp_col.setZero();
          for (SparseMatrix::InnerIterator it(mv.subTns[idx], i); it; ++it)
          {
            temp_R_1 = Matrix::Ones(mv.rank, 1);
            // Select rows of each factor an compute the respective row of the Khatri-Rao product.
            for (int mode_i = last_mode, kr_counter = static_cast<int>(TnsSize) - 2; mode_i >= 0 && kr_counter >= 0; mode_i--)
            {
              if (mode_i == idx)
              {
                continue;
              }
              int row;
              row      = ((it.row()) / mv.offsets[idx][kr_counter]) % mv.subTnsDims[mode_i][mv.fiber_rank[mode_i]];
              temp_R_1 = temp_R_1.cwiseProduct(mv.layer_factors_T[mode_i].col(row));
              kr_counter--;
            }
            // Computation of row of Z according the relation (10) of the paper.
            temp_col += (temp_R_1.transpose() * Y_T.col(i))(0) * temp_R_1;
          }
          grad_Y_T.col(i) = temp_col;
        }

        // Add each process' results and scatter the block rows among the processes in the layer.
        // MPI_Reduce_scatter(grad_Y_T.data(), grad_Y_local_T.data(), send_recv_counts, MPI_DOUBLE, MPI_SUM, mode_layer_comm);
```

```
        v2::reduce_scatter( mv.layer_comm[idx],
                            grad_Y_T,
                            mv.send_recv_counts[idx][0],
                            grad_Y_local_T );

        // Add proximal term.
        grad_Y_local_T += MTTKRP_T + lambda * Y_local_T;
        new_A          = (Y_local_T - grad_Y_local_T / L).cwiseMax(Zero_Matrix);

        new_alpha = UpdateAlpha(alpha, q);
        beta      = alpha * (1 - alpha) / (alpha * alpha + new_alpha);

        Y_local_T = (1 + beta) * new_A - beta * A;

        // The updated block rows of Y are all gathered, and we have the whole updated Y of the layer.
        // MPI_Allgatherv(Y_local_T.data(), send_recv_counts_layer, MPI_DOUBLE, Y_T.data(), send_recv_counts, displs, MPI_DOUBLE, mode_layer_comm); // Communication through layer
        v2::all_gatherv( mv.layer_comm[idx],
                        Y_local_T,
                        mv.send_recv_counts[idx][mv.layer_rank[idx]],
                        mv.send_recv_counts[idx][0],
                        mv.displs_local_update[idx][0],
                        Y_T );

        A     = new_A;
        alpha = new_alpha;
        iter++;
      }
      mv.local_factors_T[idx] = A;

    }

    void NesterovMNLS_localL(Member_Variables                    &mv,
                             Status                        const &st,
                             int    const                         idx,
                             Matrix                              &MTTKRP_T)
    {
      int iter = 0;
                              double L2;
                              double sqrt_q = 0, beta = 0;
      double lambda = st.options.lambdas[idx];

                            Matrix inv_L2(mv.subTns[idx].outerSize(), 1); // rows_layer

      Matrix grad_Y_T(mv.rank, mv.subTnsDims[idx][mv.fiber_rank[idx]]);
      Matrix grad_Y_local_T(mv.rank, mv.rows_for_update[idx]);

      Matrix Y_T(mv.rank, mv.subTnsDims[idx][mv.fiber_rank[idx]]);
      Matrix Y_local_T(mv.rank, mv.rows_for_update[idx]);

      Matrix new_A(mv.rank, mv.rows_for_update[idx]);
      Matrix A(mv.rank, mv.rows_for_update[idx]);

                             const Matrix zero_vec = Matrix::Zero(mv.rank, 1);

      A         = mv.local_factors_T[idx];
      Y_T       = mv.layer_factors_T[idx]; // layer_factor
      Y_local_T = A;

      int last_mode = (idx == static_cast<int>(TnsSize) - 1) ? static_cast<int>(TnsSize) - 2 : static_cast<int>(TnsSize) - 1;

      Matrix temp_R_1(mv.rank, 1);
                              Matrix temp_RxR(mv.rank, mv.rank);

      while (1)
      {
        grad_Y_T.setZero();

        if (iter >= st.options.max_nesterov_iter)
        {
          break;
        }

        // Compute grad_Y
        Matrix temp_col = Matrix::Zero(mv.rank, 1);
        for (long int i = 0; i < mv.subTns[idx].outerSize(); ++i)
        {
          temp_col.setZero();
          if (iter < 1)
          {
            temp_RxR.setZero();
          }
          for (SparseMatrix::InnerIterator it(mv.subTns[idx], i); it; ++it)
          {
            temp_R_1 = Matrix::Ones(mv.rank, 1);
```

```
          // Select rows of each factor an compute the respective row of the Khatri-Rao product.
          for (int mode_i = last_mode, kr_counter = static_cast<int>(TnsSize) - 2; mode_i >= 0 && kr_counter >= 0; mode_i--)
          {
            if (mode_i == idx)
            {
              continue;
            }
            long long int row;
            row      = ((it.row()) / mv.offsets[idx][kr_counter]) % mv.subTnsDims[mode_i][mv.fiber_rank[mode_i]];
            temp_R_1 = temp_R_1.cwiseProduct(mv.layer_factors_T[mode_i].col(row));
            kr_counter--;
          }
          // Computation of row of Z according the relation (10) of the paper.
          temp_col += (temp_R_1.transpose() * Y_T.col(i))(0) * temp_R_1;

          // Compute only once!
          if (iter < 1)
          {
            temp_RxR.noalias() += (temp_R_1 * temp_R_1.transpose());
          }
        }
        grad_Y_T.col(i) = temp_col;

        if (iter < 1)
        {
          // Communicate only once!
          all_reduce( mv.layer_comm[idx],
                      inplace(temp_RxR.data()),
                      mv.RxR,
                      std::plus<double>() );

          L2 = PowerMethod(temp_RxR, 1e-3);
          L2 += lambda;
          inv_L2(i) = 1 / L2;
        }
      }

      // Add each process' results and scatter the block rows among the processes in the layer.
      // MPI_Reduce_scatter(grad_Y_T.data(), grad_Y_local_T.data(), send_recv_counts, MPI_DOUBLE, MPI_SUM, mode_layer_comm);
      v2::reduce_scatter( mv.layer_comm[idx],
                          grad_Y_T,
                          mv.send_recv_counts[idx][0],
                          grad_Y_local_T );

      // Add proximal term.
      grad_Y_local_T += MTTKRP_T + lambda * Y_local_T;

      for (long int i=0; i<mv.rows_for_update[idx]; i++)
      {
        long int translate_i = i + mv.displs_local_update[idx][mv.layer_rank[idx]]/mv.rank;

        new_A.col(i) = (Y_local_T.col(i) - grad_Y_local_T.col(i) * inv_L2(translate_i)).cwiseMax(zero_vec);

        sqrt_q = sqrt( lambda * inv_L2(translate_i) );
        beta = (1 - sqrt_q) / (1 + sqrt_q);

        // Update Y
        Y_local_T.col(i) = (1 + beta) * new_A.col(i) - beta * A.col(i);
      }

      // The updated block rows of Y are all gathered, and we have the whole updated Y of the layer.
      v2::all_gatherv( mv.layer_comm[idx],
                      Y_local_T,
                      mv.send_recv_counts[idx][mv.layer_rank[idx]],
                      mv.send_recv_counts[idx][0],
                      mv.displs_local_update[idx][0],
                      Y_T );

      A = new_A;

      iter++;
    }
    mv.local_factors_T[idx] = A;
  }

  /*
   * In case option variable @c writeToFile is enabled then, before the end
   * of the algorithm writes the resulted factors in files, where their
   * paths are specified before compiling in @ options.final_factors_path.
   *
   * @param  st  [in] Struct where the returned values of @c Gtc are stored.
   */
```

```
    void writeFactorsToFile(Status const &st)
    {
      std::size_t size;
      for(std::size_t i=0; i<TnsSize; ++i)
      {
        size = st.factors[i].rows() * st.factors[i].cols();
        partensor::write(st.factors[i],
                         st.options.final_factors_paths[i],
                         size);
      }
    }

    /*
     * Compute the cost function value at the end of each outer iteration
     * based on the last factor.
     *
     * @param  grid_comm [in]     MPI communicator where the new cost function value
     *                            will be communicated and computed.
     * @param  mv        [in]     Struct where ALS variables are stored.
     * @param  st        [in,out] Struct where the returned values of @c Gtc are stored.
     *                            In this case the cost function value is updated.
     */
    void cost_function( CartCommunicator const &grid_comm,
                        Member_Variables       &mv,
                        Status                 &st )
    {
        Matrix temp_R_1(mv.rank, 1);
        double temp_1_1  = 0;
        mv.local_f_value = 0;
        std::array<int, TnsSize-1> offsets;
        offsets[0] = 1;
        for (int j = 1; j < static_cast<int>(TnsSize) - 1; j++)
        {
          offsets[j] = offsets[j - 1] * mv.subTnsDims[j-1][mv.fiber_rank[j-1]]; // mv.layer_factors_T[j - 1].cols()
        }

        for (long int i = 0; i < mv.subTns[lastFactor].outerSize(); ++i)
        {
            int row;
            for (SparseMatrix::InnerIterator it(mv.subTns[lastFactor], i); it; ++it)
            {
                temp_R_1 = mv.layer_factors_T[lastFactor].col(it.col());
                // Select rows of each factor an compute the Hadamard product of the respective row of the Khatri-Rao product, and the row of factor A_N.
                // temp_R_1  = A_N(i_N,:) .* ... .* A_2(i_2,:) .* A_1(i_1,:)
                for (int mode_i = static_cast<int>(TnsSize) - 2; mode_i >= 0; mode_i--)
                {
                    row = ((it.row()) / offsets[mode_i]) % (mv.subTnsDims[mode_i][mv.fiber_rank[mode_i]]);
                    temp_R_1.noalias() = temp_R_1.cwiseProduct(mv.layer_factors_T[mode_i].col(row));
                }
                temp_1_1 = it.value() - temp_R_1.sum();
                mv.local_f_value += temp_1_1 * temp_1_1;
            }
        }

        all_reduce( grid_comm,
                    mv.local_f_value,
                    st.f_value,
                    std::plus<double>() );
    }

    /*
     * Compute the cost function value at the end of each outer iteration
     * based on the last accelerated factor.
     *
     * @param  grid_comm         [in] MPI communicator where the new cost function value
     *                                will be communicated and computed.
     * @param  mv                [in] Struct where ALS variables are stored.
     * @param  st                [in] Struct where the returned values of @c Gtc are stored.
     *                                In this case the cost function value is updated.
     * @param  factors           [in] Accelerated factors.
     * @param  factors_T_factors [in] Gramian matrices of factors.
     *
     * @returns The cost function calculated with the accelerated factors.
     */
    double accel_cost_function(CartCommunicator const &grid_comm,
                               Member_Variables const &mv,
                               MatrixArray      const &layer_factors_T)
    {
      Matrix temp_R_1(mv.rank, 1);
      double temp_1_1 = 0;
      double f_value = 0;

      std::array<int, TnsSize-1> offsets;
      offsets[0] = 1;
```

```
          for (int j = 1; j < static_cast<int>(TnsSize) - 1; j++)
          {
            offsets[j] = offsets[j - 1] * mv.subTnsDims[j-1][mv.fiber_rank[j-1]];
          }

          for (long int i = 0; i < mv.subTns[lastFactor].outerSize(); ++i)
          {
              int row;
              for (SparseMatrix::InnerIterator it(mv.subTns[lastFactor], i); it; ++it)
              {
                  temp_R_1 = layer_factors_T[lastFactor].col(it.col());
                  // Select rows of each factor an compute the Hadamard product of the respective row of the Khatri-Rao product, and the row of factor A_N.
                  // temp_R_1  = A_N(i_N,:) .* ... .* A_2(i_2,:) .* A_1(i_1,:)
                  for (int mode_i = static_cast<int>(TnsSize) - 2; mode_i >= 0; mode_i--)
                  {
                      row = ((it.row()) / offsets[mode_i]) % (mv.subTnsDims[mode_i][mv.fiber_rank[mode_i]]);
                      temp_R_1.noalias() = temp_R_1.cwiseProduct(layer_factors_T[mode_i].col(row));
                  }
                  temp_1_1 = it.value() - temp_R_1.sum();
                  f_value += temp_1_1 * temp_1_1;
              }
          }

          all_reduce( grid_comm,
                      inplace(&f_value),
                      1,
                      std::plus<double>() );

          return f_value;
        }

        /*
         * Make use of the dimensions and the number of processors per dimension
         * and then calculates the dimensions of the subtensor and subfactor for
         * each processor.
         *
         * @tparam Dimensions          Array type containing the length of Tensor's dimensions.
         *
         * @param  tnsDims    [in]     Tensor Dimensions. Each index contains the corresponding
         *                             factor's rows length.
         * @param  st         [in]     Struct where the returned values of @c Gtc are stored.
         * @param  R          [in]     The rank of decomposition.
         * @param  mv         [in,out] Struct where ALS variables are stored.
         *                             Updates @c stl arrays with dimensions for subtensors and
         *                             subfactors.
         */
        void compute_sub_dimensions(Status           const &st,
                                    Member_Variables       &mv)
        {
          for (std::size_t i = 0; i < TnsSize; ++i)
          {
            mv.factor_T_factor[i].noalias() = st.factors[i].transpose() * st.factors[i];

            DisCount(mv.displs_subTns[i], mv.subTnsDims[i], st.options.proc_per_mode[i], mv.tnsDims[i], 1);
            // for fiber communication and Gatherv
            DisCount(mv.displs_subTns_R[i], mv.subTnsDims_R[i], st.options.proc_per_mode[i], mv.tnsDims[i], static_cast<int>(mv.rank));
            // information per layer
            DisCount(mv.displs_local_update[i], mv.send_recv_counts[i], mv.world_size / st.options.proc_per_mode[i],
                                                       mv.subTnsDims[i][mv.fiber_rank[i]], static_cast<int>(mv.rank));

            mv.rows_for_update[i] = mv.send_recv_counts[i][mv.layer_rank[i]] / static_cast<int>(mv.rank);
          }

          calculate_offsets(mv);
        }

        void calculate_offsets(Member_Variables &mv)
        {
          for (int idx = 0; idx < static_cast<int>(TnsSize); idx++)
          {
            mv.offsets[idx][0] = 1;
            for (int j = 1, mode = 0; j < static_cast<int>(TnsSize) - 1; j++, mode++)
            {
              if (idx == mode)
              {
                mode++;
              }
              mv.offsets[idx][j] = mv.offsets[idx][j - 1] * mv.subTnsDims[mode][mv.fiber_rank[mode]];
            }
          }
```

```
    }

    /*
     * Based on each factor's constraint, a different
     * update function is used at every outer iteration.
     *
     * Computes also factor^T * factor at the end.
     *
     * @param  idx [in]     Factor to be updated.
     * @param  R   [in]     The rank of decomposition.
     * @param  st  [in]     Struct where the returned values of @c Gtc are stored.
     *                      Here constraints and options variables are needed.
     * @param  mv  [in,out] Struct where ALS variables are stored.
     *                      Updates the factors of each layer.
     */
    void update_factor(int              const  idx,
                       Status           const &st,
                       Member_Variables       &mv  )
    {
      switch ( st.options.constraints[idx] )
      {
        case Constraint::unconstrained:
        case Constraint::symmetric:
        {
          // std::cout « " Inside symmetric update factor ... " « std::endl;
          Matrix A(mv.rank, mv.subTnsDims[idx][mv.fiber_rank[idx]]);
          Matrix A_local(mv.rank, mv.rows_for_update[idx]);
          Matrix eye = st.options.lambdas[idx] * Matrix::Identity(mv.rank, mv.rank);

          // int first_mode = (idx == 0) ? 1 : 0;
          int last_mode = (idx == static_cast<int>(TnsSize) - 1) ? static_cast<int>(TnsSize) - 2 : static_cast<int>(TnsSize) - 1;

          Matrix MTTKRP_col(mv.rank, 1);
          Matrix temp_RxR(mv.rank, mv.rank);
          Matrix temp_R_1(mv.rank, 1);

          // Compute MTTKRP
          for (long long int i = 0; i < mv.subTns[idx].outerSize(); ++i)
          {
            MTTKRP_col.setZero();
            temp_RxR.setZero(); // temp_RxR : is the Hadamard product of Grammians of the Factors, that correspond to the nnz elements of the Tensor.
            for (SparseMatrix::InnerIterator it(mv.subTns[idx], i); it; ++it)
            {
              temp_R_1 = Matrix::Ones(mv.rank, 1);
              long long int row;
              // Select rows of each factor an compute the respective row of the Khatri-Rao product.
              for (int mode_i = last_mode, kr_counter = static_cast<int>(TnsSize) - 2; mode_i >= 0 && kr_counter >= 0; mode_i--)
              {
                if (mode_i == idx)
                {
                  continue;
                }
                row     = ((it.row()) / mv.offsets[idx][kr_counter]) % (mv.subTnsDims[mode_i][mv.fiber_rank[mode_i]]);

                temp_R_1 = temp_R_1.cwiseProduct(mv.layer_factors_T[mode_i].col(row));
                kr_counter--;
              }
              // Subtract from the previous row the respective row of W, according to relation (9).
              MTTKRP_col.noalias() += it.value() * temp_R_1;

              temp_RxR.noalias()   += temp_R_1 * temp_R_1.transpose();
            }

            all_reduce( mv.layer_comm[idx],
                        inplace(MTTKRP_col.data()),
                        1 * mv.rank,
                        std::plus<double>() );

            all_reduce( mv.layer_comm[idx],
                        inplace(temp_RxR.data()),
                        mv.RxR,
                        std::plus<double>() );

            A.col(i) = ((temp_RxR + eye).inverse()) * MTTKRP_col;
          }
          // std::cout « " Inside symmetric update factor After loop  ... " « std::endl;

          mv.local_factors_T[idx] = A.block(0, mv.displs_local_update[idx][mv.layer_rank[idx]] / mv.rank, mv.rank,  mv.rows_for_update[idx]);

          break;
        }
        case Constraint::nonnegativity:
```

```
          case Constraint::symmetric_nonnegativity:
          {
            int last_mode = (idx == static_cast<int>(TnsSize) - 1) ? static_cast<int>(TnsSize) - 2 :
 static_cast<int>(TnsSize) - 1;

            Matrix local_MTTKRP_T = Matrix::Zero(mv.rank, mv.rows_for_update[idx]);
            Matrix MTTKRP_T       = Matrix::Zero(mv.rank, mv.subTnsDims[idx][mv.fiber_rank[idx]]);

            // Compute MTTKRP
            SparseMTTKRP<TnsSize>(mv.subTnsDims, mv.fiber_rank, mv.subTns[idx], mv.layer_factors_T,
 mv.rank, mv.offsets[idx], last_mode, idx, MTTKRP_T);

            // Add each process' results and scatter the block rows among the processes in the layer.
            // MPI_Reduce_scatter(MTTKRP_T.data(), local_MTTKRP_T.data(), send_recv_counts,
 MPI_DOUBLE, MPI_SUM, mode_layer_comm);
            v2::reduce_scatter( mv.layer_comm[idx],
                                MTTKRP_T,
                                mv.send_recv_counts[idx][0],
                                local_MTTKRP_T );

            // NesterovMNLS(mv, st, idx, local_MTTKRP_T);
            NesterovMNLS_localL(mv, st, idx, local_MTTKRP_T);

            break;
          }
          case Constraint::sparsity:
            break;
          default: // in case of Constraint::constant
            break;
        } // end of constraints switch

        // std::cout « " Inside update factor After switch  ... " « std::endl;
        if (st.options.constraints[idx] != Constraint::constant)
        {
          v2::all_gatherv( mv.layer_comm[idx],
                           mv.local_factors_T[idx],
                           mv.send_recv_counts[idx][mv.layer_rank[idx]],
                           mv.send_recv_counts[idx][0],
                           mv.displs_local_update[idx][0],
                           mv.layer_factors_T[idx] );

          mv.layer_factors[idx]             = mv.layer_factors_T[idx].transpose();

          if (st.options.constraints[idx] == Constraint::symmetric_nonnegativity ||
 st.options.constraints[idx] == Constraint::symmetric)
          {
            for (std::size_t i=0; i<TnsSize; i++)
            {
              if (i != static_cast<std::size_t>(idx))
              {
                mv.layer_factors_T[i] = mv.layer_factors_T[idx];
              }
            }
          }

          mv.factor_T_factor[idx].noalias() = mv.layer_factors_T[idx] * mv.layer_factors[idx];
        }


        // std::cout « " Inside update factor .... switch  ... " « std::endl;

        all_reduce( mv.fiber_comm[idx],
                    inplace(mv.factor_T_factor[idx].data()),
                    mv.RxR,
                    std::plus<double>() );
      }

      /*
       * At the end of the algorithm processor 0
       * collects each part of the factor that each
       * processor holds and return them in status.factors.
       *
       * @param  mv      [in]     Struct where ALS variables are stored.
       *                          Use variables to compute result factors by gathering each
       *                          part of the factor from processors.
       * @param  st      [in,out] Struct where the returned values of @c Gtc are stored.
       *                          Stores the resulted factors.
       */
      void gather_final_factors(Member_Variables       &mv,
                                Status                 &st)
      {
        for(std::size_t i=0; i<TnsSize; ++i)
        {
          mv.temp_matrix.resize(static_cast<int>(mv.rank), mv.tnsDims[i]);
          // Gatherv from all processors to processor with rank 0 the final factors
          v2::gatherv( mv.fiber_comm[i],
                       mv.layer_factors_T[i],
```

```
                        mv.subTnsDims_R[i][mv.fiber_rank[i]],
                        mv.subTnsDims_R[i][0],
                        mv.displs_subTns_R[i][0],
                        0,
                        mv.temp_matrix );

          st.factors[i] = mv.temp_matrix.transpose();
        }
      }

      /*
       * @brief Line Search Acceleration
       *
       * Performs an acceleration step in the updated factors, and keeps the accelerated factors when
       * the step succeeds. Otherwise, the acceleration step is ignored.
       * Line Search Acceleration reduces the number outer iterations in the ALS algorithm.
       *
       * @note This implementation ONLY, if factors are of @c Matrix type.
       *
       * @param  grid_comm [in]     MPI communicator where the new cost function value
       *                            will be communicated and computed.
       * @param  mv        [in,out] Struct where ALS variables are stored.
       *                            In case the acceration is successful layer factor^T * factor
       *                            and layer factor variables are updated.
       * @param  st        [in,out] Struct where the returned values of @c Gtc are stored.
       *                            If the acceleration succeeds updates cost function value.
       *
       */
      void line_search_accel(CartCommunicator const &grid_comm,
                             Member_Variables       &mv,
                             Status                 &st)
      {
        double       f_accel    = 0.0; // Objective Value after the acceleration step
        double const accel_step = pow(st.ao_iter+1,(1.0/(st.options.accel_coeff)));

        MatrixArray   accel_factors_T;
        MatrixArray   accel_gramians;

        for(std::size_t i=0; i<TnsSize; ++i)
        {
          accel_factors_T[i] = mv.old_factors_T[i] + accel_step * (mv.layer_factors_T[i] -
      mv.old_factors_T[i]);
          accel_gramians[i]  = accel_factors_T[i] * accel_factors_T[i].transpose();
          all_reduce( mv.fiber_comm[i],
                      inplace(accel_gramians[i].data()),
                      mv.RxR,
                      std::plus<double>() );
        }

        f_accel = accel_cost_function(grid_comm, mv, accel_factors_T);
        if (st.f_value > f_accel)
        {
          mv.layer_factors_T = accel_factors_T;
          mv.factor_T_factor = accel_gramians;
          st.f_value         = f_accel;
          if(grid_comm.rank() == 0)
            Partensor()->Logger()->info("Acceleration Step SUCCEEDED! at iter: {}", st.ao_iter);
        }
        else
          st.options.accel_fail++;

        if (st.options.accel_fail==5)
        {
          st.options.accel_fail=0;
          st.options.accel_coeff++;
        }
      }

      /*
       * Parallel implementation of als method with MPI.
       *
       * @tparam Dimensions         Array type containing the Tensor dimensions.
       *
       * @param  grid_comm [in]     The communication grid, where the processors
       *                            communicate their cost function.
       * @param  tnsDims   [in]     Tensor Dimensions. Each index contains the corresponding
       *                            factor's rows length.
       * @param  R         [in]     The rank of decomposition.
       * @param  mv        [in]     Struct where ALS variables are stored and being updated
       *                            until a termination condition is true.
       * @param  status    [in,out] Struct where the returned values of @c Gtc are stored.
       */
      void aogtc(CartCommunicator const &grid_comm,
                 Member_Variables       &mv,
                 Status                 &status)
      {
        status.frob_tns = (mv.subTns[0]).squaredNorm();
```

```
            all_reduce( grid_comm,
                  inplace(&status.frob_tns),
                  1,
                  std::plus<double>() );

        cost_function(grid_comm, mv, status);
        status.rel_costFunction = status.f_value / status.frob_tns;

        for (int i=0; i< static_cast<int>(TnsSize); i++)
        {
          mv.factor_T_factor[i].noalias() = mv.layer_factors_T[i] * mv.layer_factors[i];
          all_reduce( mv.fiber_comm[i],
                      inplace(mv.factor_T_factor[i].data()),
                      mv.RxR,
                      std::plus<double>() );
        }

        // Wait for all processors to reach here
        grid_comm.barrier();

        // ---- Loop until ALS converges ----
        while(1)
        {
            status.ao_iter++;
            if (!grid_comm.rank())
            {
                Partensor()->Logger()->info("iter: {} -- fvalue: {} -- relative_costFunction: {}", status.ao_iter,
                                                    status.f_value, status.rel_costFunction);

                std::cout « "iter: " « status.ao_iter « " - status.f_value: " « status.f_value « " - status.rel_costFunction: " « status.rel_costFunction « std::endl;
            }

            //        <------------------------        loop for every mode ---------------------------->    //
            for (std::size_t i = 0; i < TnsSize; i++)
            {
              // Compute hadamard of grammians to compute L.
              mv.cwise_factor_product = partensor::PartialCwiseProd(mv.factor_T_factor, i);


              // Partition rows of subfactor to the processes in the respective layer.
              mv.local_factors_T[i] = mv.layer_factors_T[i].block(0, mv.displs_local_update[i][mv.layer_rank[i]] / mv.rank, mv.rank, mv.rows_for_update[i]);


              update_factor(i, status, mv);

            }

            // ---- Cost function Computation ----
            cost_function(grid_comm, mv, status);

            status.rel_costFunction = status.f_value / status.frob_tns;

            // if(status.options.normalization && !mv.all_orthogonal)
            //   Normalize(mv.weight_factor, mv.rank, mv.factor_T_factor, mv.layer_factors);

            // ---- Terminating condition ----
            if (status.ao_iter >= status.options.max_iter)
            {
                gather_final_factors(mv, status);
                if(grid_comm.rank() == 0)
                {
                  std::cout « "status.rel_costFunction : " « status.rel_costFunction « std::endl;
                  Partensor()->Logger()->info("Processor 0 collected all {} factors.\n", TnsSize);
                  if(status.options.writeToFile)
                    writeFactorsToFile(status);
                }
                break;
            }

            if (status.options.acceleration)
            {
              mv.norm_factors_T = mv.layer_factors_T;
              // ---- Acceleration Step ----
              if (status.ao_iter > 1)
                line_search_accel(grid_comm, mv, status);

              mv.old_factors_T = mv.norm_factors_T;
            }

        } // end of outer while loop
      }

      void initialize_factors(Member_Variables &mv,
```

```
                                Status           &status)
        {
          if(status.options.initialized_factors)
          {
            if(status.options.read_factors_from_file)
            {
              for(std::size_t i=0; i<TnsSize; ++i)
              {
                status.factors[i] = Matrix(mv.tnsDims[i], static_cast<int>(mv.rank));
                read( status.options.initial_factors_paths[i],
                      mv.tnsDims[i] * mv.rank,
                      0,
                      status.factors[i] );
              }
            }
            else
              status.factors = status.options.factorsInit;
          }
          else
            makeFactors(mv.tnsDims, status.options.constraints, mv.rank, status.factors);
        }

        Status operator()(Options const &options)
        {
          Status           status(options);
          Member_Variables mv(options.rank, options.tnsDims, status.options.proc_per_mode);

         // Communicator with cartesian topology
          CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);

          // Functions that create layer and fiber grids.
          create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
          create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);

          // produce estimate factors using uniform distribution with entries in [0,1].
          initialize_factors(mv, status);

          compute_sub_dimensions(status, mv);

          for (std::size_t i = 0; i < TnsSize; ++i)
          {
            mv.layer_factors[i]           =
  status.factors[i].block(mv.displs_subTns[i][mv.fiber_rank[i]], 0,
                                                           mv.subTnsDims[i][mv.fiber_rank[i]],  mv.rank);
            mv.layer_factors_T[i]           = mv.layer_factors[i].transpose();
            mv.factor_T_factor[i].noalias() = mv.layer_factors_T[i] * mv.layer_factors[i];
          }

          long long int fileSize = (TnsSize + 1) * options.nonZeros;

          // Matrix Ratings_Base = Matrix(options.nonZeros, static_cast<int>(TnsSize+1));
          Matrix Ratings_Base_T = Matrix(static_cast<int>(TnsSize+1), options.nonZeros);

          // Read the whole Tensor from a file
          read( options.ratings_path,
                fileSize,
                0,
                Ratings_Base_T );

          // Matrix Ratings_Base_T = Ratings_Base.transpose();

          sort_ratings_base(Ratings_Base_T, options.nonZeros, mv);
                Ratings_Base_T.resize(0,0);
          // Ratings_Base.resize(0,0);

          aogtc(grid_comm, mv, status);

           return status;
         }

         Status operator()(Matrix       const &Ratings_Base_T,
                           Options      const &options)
         {
           Status           status(options);
           Member_Variables mv(options.rank, options.tnsDims, status.options.proc_per_mode);

          // Communicator with cartesian topology
           CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);

           // Functions that create layer and fiber grids.
           create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
           create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);

           // produce estimate factors using uniform distribution with entries in [0,1].
           initialize_factors(mv, status);

           compute_sub_dimensions(status, mv);
```

```
        // Begin Load Balancing
              // Matrix                                     Balanced_Ratings_Base_T(TnsSize + 1, options.nonZeros);
              // std::array<std::vector<long int>, TnsSize> perm_tns_indices;

        // BalanceDataset<TnsSize>(options.nonZeros, options.tnsDims, Ratings_Base_T, perm_tns_indices, Balanced_Ratings_Base_T);

        // PermuteFactors<TnsSize>(status.factors, perm_tns_indices, mv.factors_T);

        std::cout « "After Balance........" « std::endl;
        for (std::size_t i = 0; i < TnsSize; ++i)
        {
          mv.layer_factors[i]             = status.factors[i].block(mv.displs_subTns[i][mv.fiber_rank[i]], 0,
                                                      mv.subTnsDims[i][mv.fiber_rank[i]], mv.rank);
          mv.layer_factors_T[i]           = mv.layer_factors[i].transpose();
          mv.factor_T_factor[i].noalias() = mv.layer_factors_T[i] * mv.layer_factors[i];

          // mv.layer_factors_T[i] = mv.factors_T[i].block(0, mv.displs_subTns[i][mv.fiber_rank[i]],
          //                                             mv.rank, mv.subTnsDims[i][mv.fiber_rank[i]]);
          // mv.layer_factors[i]            = mv.layer_factors_T[i].transpose();
          // mv.factor_T_factor[i].noalias() = mv.layer_factors_T[i] * mv.layer_factors[i];
        }

        // Each processor takes a subtensor from tnsX
        ReserveSparseTensor<TnsSize>(mv.subTns, mv.subTnsDims, mv.fiber_rank, mv.world_size, options.nonZeros);

        Dist_NNZ<TnsSize>(mv.subTns, options.nonZeros, mv.displs_subTns, mv.fiber_rank, Ratings_Base_T, mv.subTnsDims);
        // Dist_NNZ<TnsSize>(mv.subTns, options.nonZeros, mv.displs_subTns, mv.fiber_rank, Balanced_Ratings_Base_T, mv.subTnsDims);
              // Ratings_Base_T.resize(0,0);
              // Balanced_Ratings_Base_T.resize(0,0);

        for(int mode_i = 0; mode_i < static_cast<int>(TnsSize); mode_i++)
        {
          mv.subTns[mode_i].makeCompressed();
        }

        aogtc(grid_comm, mv, status);

        return status;
      }

    };
  }  // namespace internal
}    // namespace v1

} // end namespace partensor
```

8.27 GtcOpenMP.hpp File Reference

8.27.1 Detailed Description

Implements the Canonical Polyadic Decomposition(gtc) using `OpenMP`. Make use of `spdlog` library in order to write output in a log file in "`../log`".

8.28 GtcOpenMP.hpp

Go to the documentation of this file.

```
#ifndef DOXYGEN_SHOULD_SKIP_THIS
#endif // DOXYGEN_SHOULD_SKIP_THIS
/*****************************************************************/
#if !defined(PARTENSOR_GTC_HPP)
#error "GTC_OMP can only included inside GTC"
#endif /* PARTENSOR_GTC_HPP */

namespace partensor
```

```
{
  inline namespace v1
  {
    namespace internal
    {
      template <std::size_t TnsSize_>
      struct GTC<TnsSize_, execution::openmp_policy> : public GTC_Base<TnsSize_>
      {
        using          GTC_Base<TnsSize_>::TnsSize;
        using          GTC_Base<TnsSize_>::lastFactor;
        using typename GTC_Base<TnsSize_>::Dimensions;
        using typename GTC_Base<TnsSize_>::MatrixArray;
        using typename GTC_Base<TnsSize_>::DataType;
        using typename GTC_Base<TnsSize_>::SparseTensor;
        using typename GTC_Base<TnsSize_>::IntArray;
        using typename GTC_Base<TnsSize_>::LongMatrix;

        using Options = partensor::SparseOptions<TnsSize_,execution::openmp_policy,SparseDefaultValues>;
        using Status  = partensor::SparseStatus<TnsSize_,execution::openmp_policy,SparseDefaultValues>;

        // Variables that will be used in gtc implementations.
        struct Member_Variables
        {
          MatrixArray  factors_T;
          MatrixArray  factor_T_factor;
          MatrixArray  mttkrp_T;
          IntArray     tnsDims;
          std::array<std::array<int, TnsSize_ - 1>, TnsSize_> offsets;

          MatrixArray  norm_factors_T;
          MatrixArray  old_factors_T;

          Matrix       cwise_factor_product;
          Matrix       Ratings_Base_T;
          SparseTensor tnsX;

          // bool          all_orthogonal = true;
          // int           weight_factor;
          int          rank;

          MatrixArray grad;
          MatrixArray Y;
          MatrixArray invL;

          Member_Variables() = default;

          Member_Variables(int R, IntArray dims) :  tnsDims(dims),
                                                    rank(R)
          {}

          Member_Variables(Member_Variables const &) = default;
          Member_Variables(Member_Variables      &&) = default;

          Member_Variables &operator=(Member_Variables const &) = default;
          Member_Variables &operator=(Member_Variables      &&) = default;
        };

        /*
         * In case option variable @c writeToFile is enabled, then, before the end
         * of the algorithm, it writes the resulted factors in files, whose
         * paths are specified before compiling in @ options.final_factors_path.
         *
         * @param  st  [in] Struct where the returned values of @c Gtc are stored.
         */
        void writeFactorsToFile(Status const &st)
        {
          std::size_t size;
          for(std::size_t i=0; i<TnsSize; ++i)
          {
            size = st.factors[i].rows() * st.factors[i].cols();
             partensor::write(st.factors[i],
                              st.options.final_factors_paths[i],
                              size);
           }
         }

         /*
          * Compute the cost function value at the end of each outer iteration
          * based on the last factor.
          *
          * @param  mv  [in]      Struct where ALS variables are stored.
          * @param  st  [in,out] Struct where the returned values of @c Gtc are stored.
          *                      In this case the cost function value is updated.
          */
         void cost_function(Member_Variables const &mv,
                            Status                 &st)
         {
```

```
      Matrix temp_R_1(mv.rank, 1);
      double temp_1_1 = 0;
      double f_value_loc = 0;

      #pragma omp master
      st.f_value = 0;

      #pragma omp barrier

      std::array<int,TnsSize-1> offsets;
      offsets[0] = 1;
      for (int j = 1; j < static_cast<int>(TnsSize) - 1; j++)
      {
        offsets[j] = offsets[j - 1] * mv.tnsDims[j-1];
      }

      #pragma omp for schedule(static)
      for (long int i = 0; i < mv.tnsX[lastFactor].outerSize(); ++i)
      {
          int row = 0;
          for (SparseMatrix::InnerIterator it(mv.tnsX[lastFactor], i); it; ++it)
          {
              temp_R_1 = mv.factors_T[lastFactor].col(it.col());
              // Select rows of each factor an compute the Hadamard product of the respective row of the Khatri-Rao product, and the row of factor A_N.
              for (int mode_i = static_cast<int>(TnsSize) - 2; mode_i >= 0; mode_i--)
              {
                  row                  = ((it.row()) / offsets[mode_i]) % (mv.tnsDims[mode_i]);
                  temp_R_1.noalias() = temp_R_1.cwiseProduct(mv.factors_T[mode_i].col(row));
              }
              temp_1_1 = it.value() - temp_R_1.sum();
              f_value_loc += temp_1_1 * temp_1_1;
          }
      }
      #pragma omp atomic
      st.f_value += f_value_loc;

      #pragma omp barrier

    }

    /*
     * Compute the cost function value at the end of each outer iteration
     * based on the last accelerated factor.
     *
     * @param  mv                 [in] Struct where ALS variables are stored.
     * @param  accel_factors      [in] Accelerated factors.
     *
     * @returns The cost function calculated with the accelerated factors.
     */
    double accel_cost_function(Member_Variables       const &mv,
                               MatrixArray            const &accel_factors)
    {
      Matrix temp_R_1(mv.rank, 1);
      double temp_1_1 = 0;
      double f_value = 0;

      std::array<int,TnsSize-1> offsets;
      offsets[0] = 1;
      for (int j = 1; j < static_cast<int>(TnsSize) - 1; j++)
      {
        offsets[j] = offsets[j - 1] * mv.tnsDims[j-1];
      }

      #pragma omp for schedule(static)
      for (long int i = 0; i < mv.tnsX[lastFactor].outerSize(); ++i)
      {
        int row = 0;
        for (SparseMatrix::InnerIterator it(mv.tnsX[lastFactor], i); it; ++it)
        {
            temp_R_1 = accel_factors[lastFactor].col(it.col());
            // Select rows of each factor an compute the Hadamard product of the respective row of the Khatri-Rao product, and the row of factor A_N.
            // temp_R_1  = A_N(i_N,:) .* ... .* A_2(i_2,:) .* A_1(i_1,:)
            for (int mode_i = static_cast<int>(TnsSize) - 2; mode_i >= 0; mode_i--)
            {
                row                  = ((it.row()) / offsets[mode_i]) % (mv.tnsDims[mode_i]);
                temp_R_1.noalias() = temp_R_1.cwiseProduct(accel_factors[mode_i].col(row));
            }
            temp_1_1 = it.value() - temp_R_1.sum();
            f_value += temp_1_1 * temp_1_1;
        }
      }
      return f_value;
    }

    void calculate_offsets(Member_Variables &mv)
```

```
     {
       for (int idx = 0; idx < static_cast<int>(TnsSize); idx++)
       {
         mv.offsets[idx][0] = 1;
         for (int j = 1, mode = 0; j < static_cast<int>(TnsSize) - 1; j++, mode++)
         {
           if (idx == mode)
           {
             mode++;
           }
           mv.offsets[idx][j] = mv.offsets[idx][j - 1] * mv.tnsDims[mode];
         }
       }
     }

     void unconstraint_update(int              const  idx,
                              Member_Variables       &mv,
                              Status                 &st)
     {
         int r = mv.rank;

         Matrix eye = st.options.lambdas[idx] * Matrix::Identity(r, r);

         int last_mode = (idx == static_cast<int>(TnsSize) - 1) ? static_cast<int>(TnsSize) - 2 : static_cast<int>(TnsSize) - 1;

         Matrix MTTKRP_col(r, 1);
         Matrix temp_RxR(r, r);
         Matrix temp_R_1(r, 1);

         // Compute MTTKRP
         #pragma omp for schedule(dynamic) //nowait
         for (long int i = 0; i < mv.tnsX[idx].outerSize(); ++i)
         {
             MTTKRP_col.setZero();
             temp_RxR.setZero(); // is the Hadamard product of Grammians of the Factors, that correspond to the nnz elements of the Tensor.
             for (SparseMatrix::InnerIterator it(mv.tnsX[idx], i); it; ++it)
             {
                 temp_R_1 = Matrix::Ones(r, 1);
                 int row;
                 // Select rows of each factor an compute the respective row of the Khatri-Rao product.
                 for (int mode_i = last_mode, kr_counter = static_cast<int>(TnsSize) - 2; mode_i >= 0 && kr_counter >= 0; mode_i--)
                 {
                     if (mode_i == idx)
                     {
                         continue;
                     }
                     row      = ((it.row()) / mv.offsets[idx][kr_counter]) % (mv.tnsDims[mode_i]);
                     temp_R_1 = temp_R_1.cwiseProduct(mv.factors_T[mode_i].col(row));
                     kr_counter--;
                 }
                 // Subtract from the previous row the respective row of W, according to relation (9).
                 MTTKRP_col.noalias() += it.value() * temp_R_1;
                 temp_RxR.noalias()   += temp_R_1 * temp_R_1.transpose();
             }
             mv.factors_T[idx].col(i) = (temp_RxR + eye).inverse() * MTTKRP_col;
         }
     }

     /*
      * Based on each factor's constraint, a different
      * update function is used at every outer iteration.
      *
      * Computes also factor^T * factor at the end.
      *
      * @param  idx [in]     Factor to be updated.
      * @param  mv  [in]     Struct where ALS variables are stored.
      * @param  st  [in,out] Struct where the returned values of @c Gtc are stored.
      *                      Updates the @c stl array with the factors.
      */
     void update_factor(int              const  idx,
                        Member_Variables       &mv,
                        Status                 &st    )
     {
       // Update factor
       switch ( st.options.constraints[idx] )
       {
         case Constraint::unconstrained:
         {
           unconstraint_update(idx, mv, st);
           break;
         }
         case Constraint::nonnegativity:
```

```
        {
          int last_mode = (idx == static_cast<int>(TnsSize) - 1) ? static_cast<int>(TnsSize) - 2 : static_cast<int>(TnsSize) - 1;
          SparseMTTKRP_omp(mv.tnsDims, mv.tnsX[idx], mv.factors_T, mv.rank, mv.offsets[idx], last_mode, idx, mv.mttkrp_T[idx]);
          #pragma omp barrier

          // NesterovMNLS(mv.cwise_factor_product, mv.factors_T, mv.tnsDims, mv.tnsX[idx], mv.offsets[idx], mv.Y[idx],
          //                st.options.max_nesterov_iter, st.options.lambdas[idx], idx, st.options.constraints[idx], mv.mttkrp_T[idx]);

          local_L::NesterovMNLS(mv.invL[idx], mv.factors_T, mv.tnsDims, mv.tnsX[idx], mv.offsets[idx], mv.Y[idx],
                                 st.options.max_nesterov_iter, st.options.lambdas[idx], idx, mv.mttkrp_T[idx]);

          break;
        }
        default: // in case of Constraint::constant
          break;
      }

      // Compute A^T * A + B^T * B + ...
      #pragma omp master
      {
        st.factors[idx] = mv.factors_T[idx].transpose();
        mv.factor_T_factor[idx].noalias() = mv.factors_T[idx] * mv.factors_T[idx].transpose();
      }
    }

    /*
     * @brief Line Search Acceleration
     *
     * Performs an acceleration step on the updated factors, and keeps the accelerated factors
     * when the step succeeds. Otherwise, the acceleration step is ignored.
     * Line Search Acceleration reduces the number of outer iterations in the ALS algorithm.
     *
     * @note This implementation ONLY, if factors are of @c Matrix type.
     *
     * @param  mv  [in,out] Struct where ALS variables are stored.
     *                      In case the acceleration step is successful the Gramian
     *                      matrices of factors are updated.
     * @param  st  [in,out] Struct where the returned values of @c Gtc are stored.
     *                      If the acceleration succeeds updates @c factors
     *                      and cost function value.
     *
     */
    void line_search_accel(Member_Variables &mv,
                           Status           &st,
                           double           &f_accel,
                           double           &accel_step,
                           MatrixArray      &accel_factors_T,
                           MatrixArray      &accel_gramians)
    {
      #pragma omp master
      {
        for(std::size_t i=0; i<TnsSize; ++i)
        {
            accel_factors_T[i] = mv.old_factors_T[i] + accel_step * (mv.factors_T[i] - mv.old_factors_T[i]);
            accel_gramians[i]  = accel_factors_T[i] * accel_factors_T[i].transpose();
        }

        f_accel = 0;
      }

      #pragma omp barrier

      double f_accel_loc = accel_cost_function(mv, accel_factors_T);

      #pragma omp atomic
      f_accel += f_accel_loc;

      #pragma omp barrier

      #pragma omp master
      {
        if (st.f_value > f_accel)
        {
            mv.factors_T       = accel_factors_T;
            mv.factor_T_factor = accel_gramians;
            st.f_value         = f_accel;
            Partensor()->Logger()->info("Acceleration Step SUCCEEDED! at iter: {}", st.ao_iter);
        }
        else
            st.options.accel_fail++;
```

```
            if (st.options.accel_fail==5)
            {
                st.options.accel_fail=0;
                st.options.accel_coeff++;
            }
          }
        }

        /*
         * Sequential implementation of Alternating Least Squares (ALS) method.
         *
         * @param  R   [in]     The rank of decomposition.
         * @param  mv  [in]     Struct where ALS variables are stored and being updated
         *                      until a termination condition is true.
         * @param  st  [in,out] Struct where the returned values of @c Gtc are stored.
         */
        void aogtc(Member_Variables       &mv,
                   Status                 &status)
        {
          double f_accel    = 0.0; // Objective Value after the acceleration step
          double accel_step = 0.0;

          MatrixArray  accel_factors_T;
          MatrixArray  accel_gramians;

          for (std::size_t i=0; i<TnsSize; i++)
          {
            mv.Y[i]             = Matrix::Zero(mv.rank, mv.tnsDims[i]);
            mv.factors_T[i]     = status.factors[i].transpose();
            mv.mttkrp_T[i]      = Matrix(mv.rank, mv.tnsDims[i]);
            mv.factor_T_factor[i].noalias() = mv.factors_T[i] * status.factors[i];
            accel_factors_T[i] = mv.factors_T[i];
            accel_gramians[i]  = Matrix::Zero(mv.rank, mv.rank);
            mv.invL[i]          = Matrix::Zero(mv.tnsDims[i], 1);
          }

          // if(status.options.normalization)
          // {
          //   choose_normilization_factor(status, mv.all_orthogonal, mv.weight_factor);
          // }
          // Normalize(static_cast<int>(R), mv.factor_T_factor, status.factors);

          const int total_num_threads = get_num_threads();
          omp_set_nested(0);

          status.frob_tns          = (mv.tnsX[0]).squaredNorm();

          #pragma omp parallel \
                  num_threads(total_num_threads) \
                  proc_bind(spread)\
                  default(shared)\
                  shared(status, mv)
          {
            cost_function(mv, status);
            #pragma omp barrier

            #pragma omp master
            {
              status.rel_costFunction = status.f_value / status.frob_tns;
            }
            #pragma omp barrier


            // ---- Loop until ALS converges ----
            while(1)
            {
                #pragma omp master
                {
                    status.ao_iter++;
                    Partensor()->Logger()->info("iter: {} -- fvalue: {} -- relative_costFunction: {}",
      status.ao_iter,
                                                            status.f_value, status.rel_costFunction);

                    std::cout « "iter : " « status.ao_iter « " - fvalue : " « status.f_value « " -
      relative_costFunction : " « status.rel_costFunction « std::endl;
                    // status.f_value = 0;
                }
                #pragma omp barrier
                for (std::size_t i=0; i<TnsSize; i++)
                {
                    #pragma omp master
                    {
                        mv.cwise_factor_product = PartialCwiseProd(mv.factor_T_factor, i);
                    }
                    #pragma omp barrier
                    // Update factor
```

```
                    update_factor(i, mv, status);
                    #pragma omp barrier
                }

                #pragma omp barrier
                // Avg Factors if symmetric tensor
                if (status.options.averaging && status.ao_iter >= 1)
                {
                  #pragma omp master
                  {
                    for (int i = 1; i < static_cast<int>(TnsSize); i++)
                    {
                      mv.factors_T[0].noalias() += mv.factors_T[i];
                    }

                    mv.factors_T[0].noalias() = mv.factors_T[0] / TnsSize;

                    for (int i = 0; i < static_cast<int>(TnsSize); i++)
                    {
                      mv.factors_T[i] = mv.factors_T[0];

                      status.factors[i] = mv.factors_T[i].transpose();
                      mv.factor_T_factor[i].noalias() = mv.factors_T[i] * mv.factors_T[i].transpose();
                    }
                  }
                }
                #pragma omp barrier

                cost_function(mv, status);
                #pragma omp master
                {
                  status.rel_costFunction = status.f_value / status.frob_tns;
                }
                #pragma omp barrier

                // ------------------------------------------------------------------------------------ //
                // if(status.options.normalization && !mv.all_orthogonal)
                //   Normalize(mv.weight_factor, static_cast<int>(R), mv.factor_T_factor, status.factors);

                // ---- Terminating condition ----
                if (status.rel_costFunction < status.options.threshold_error || status.ao_iter >= status.options.max_iter)
                {
                    #pragma omp master
                    {
                        if (status.options.writeToFile)
                            writeFactorsToFile(status);
                    }
                    break;
                }
                #pragma omp barrier // DON'T REMOVE!

                if (status.options.acceleration)
                {
                    // ---- Acceleration Step ----
                    if (status.ao_iter > 1)
                    {
                        #pragma omp master
                        accel_step = pow(status.ao_iter+1,(1.0/(status.options.accel_coeff)));

                        line_search_accel(mv, status, f_accel, accel_step, accel_factors_T, accel_gramians);
                        #pragma omp barrier
                    }
                    // Averaging
                    #pragma omp master
                    {
                      for (int i = 1; i < static_cast<int>(TnsSize); i++)
                      {
                        mv.factors_T[0].noalias() += mv.factors_T[i];
                      }

                      mv.factors_T[0].noalias() = mv.factors_T[0] / TnsSize;

                      for (int i = 0; i < static_cast<int>(TnsSize); i++)
                      {
                        mv.factors_T[i] = mv.factors_T[0];

                        status.factors[i] = mv.factors_T[i].transpose();
                        mv.factor_T_factor[i].noalias() = mv.factors_T[i] * mv.factors_T[i].transpose();

                        if (status.options.acceleration)
                        {
                          mv.old_factors_T[i] = mv.factors_T[i];
                        }
```

```
                  }
                }
                #pragma omp master
                {
                    for (int i = 0; i < static_cast<int>(TnsSize); i++)
                        mv.old_factors_T[i] = mv.factors_T[i];
                }
                #pragma omp barrier
            }
        } // end of while
      } // end of pragma
    }

    void initialize_factors(Member_Variables &mv,
                            Status           &status)
    {
      if(status.options.initialized_factors)
      {
        if(status.options.read_factors_from_file)
        {
          for(std::size_t i=0; i<TnsSize; ++i)
          {
            status.factors[i] = Matrix(mv.tnsDims[i], static_cast<int>(mv.rank));
            read( status.options.initial_factors_paths[i],
                  mv.tnsDims[i] * mv.rank,
                  0,
                  status.factors[i] );
          }
        }
        else
          status.factors = status.options.factorsInit;
      }
      else // produce estimate factors using uniform distribution with entries in [0,1].
        makeFactors(mv.tnsDims, status.options.constraints, mv.rank, status.factors);
    }

    Status operator()(Options const &options)
    {
      Status           status(options);
      Member_Variables mv(options.rank, options.tnsDims);

      long long int fileSize = (TnsSize + 1) * options.nonZeros;

      Matrix Ratings_Base_T = Matrix(static_cast<int>(TnsSize+1), options.nonZeros);
      // Read the whole Tensor from a file
      read( options.ratings_path,
            fileSize,
            0,
            Ratings_Base_T );

      // GTC_Base<TnsSize>::sort_ratings_base(mv.Ratings_Base_T, options.nonZeros);
      // ReserveSparseTensor<TnsSize>(mv.tnsX, options.tnsDims, options.nonZeros);
      // FillSparseTensor<TnsSize>(mv.tnsX, options.nonZeros, mv.Ratings_Base_T, options.tnsDims);
      // mv.Ratings_Base_T.resize(0,0);
      GTC_Base<TnsSize>::sort_ratings_base(Ratings_Base_T, mv.tnsX, options.tnsDims,
options.nonZeros);
      Ratings_Base_T.resize(0,0);

      for (std::size_t i=0; i<TnsSize; i++)
      {
        mv.tnsX[i].makeCompressed();
      }

      calculate_offsets(mv);

      initialize_factors(mv, status);

      aogtc(mv, status);

      return status;
    }

    Status operator()(Matrix       const &Ratings_Base_T,
                      Options      const &options)
    {
      Status           status(options);
      Member_Variables mv(options.rank, options.tnsDims);

      // ReserveSparseTensor<TnsSize>(mv.tnsX, options.tnsDims, options.nonZeros);
      // FillSparseTensor<TnsSize>(mv.tnsX, options.nonZeros, Ratings_Base_T, options.tnsDims);
      // Ratings_Base_T.resize(0,0);
      GTC_Base<TnsSize>::sort_ratings_base(Ratings_Base_T, mv.tnsX, options.tnsDims,
options.nonZeros);
      // Ratings_Base_T.resize(0,0);

      for (std::size_t i=0; i<TnsSize; i++)
      {
```

```
                mv.tnsX[i].makeCompressed();
            }

            calculate_offsets(mv);

            // produce estimate factors using uniform distribution with entries in [0,1].
            initialize_factors(mv, status);

            partensor::timer.startChronoHighTimer();
            aogtc(mv, status);
            double end_gtc_time_omp = partensor::timer.endChronoHighTimer();
            std::cout « "GtcOpenMP took " « end_gtc_time_omp « " sec." « std::endl;
            return status;
          }
        };
      }  // namespace internal
    }    // namespace v1
} // end namespace partensor
```

8.29 GtcStochastic.hpp File Reference

```
#include "PARTENSOR_basic.hpp"
#include "PartialCwiseProd.hpp"
#include "NesterovMNLS.hpp"
#include "Normalize.hpp"
#include "Timers.hpp"
#include "ReadWrite.hpp"
```

Functions

- template< std::size_t _TnsSize, typename ExecutionPolicy >
  execution::internal::enable_if_execution_policy< ExecutionPolicy, SparseStatus< _TnsSize, execution::execution_policy_t< ExecutionPolicy >, SparseDefaultValues > > gtc_stochastic (ExecutionPolicy &&, Matrix const &Ratings_Base_T, SparseOptions< _TnsSize, execution::execution_policy_t< ExecutionPolicy >, SparseDefaultValues > const &options)
- template< std::size_t _TnsSize, typename ExecutionPolicy >
  execution::internal::enable_if_execution_policy< ExecutionPolicy, SparseStatus< _TnsSize, execution::execution_policy_t< ExecutionPolicy >, SparseDefaultValues > > gtc_stochastic (ExecutionPolicy &&, SparseOptions< _TnsSize, execution::execution_policy_t< ExecutionPolicy >, SparseDefaultValues > const &options)

8.29.1 Detailed Description

Implements the Stochastic General Tensor Completion(gtc stochastic). Make use of `spdlog` library in order to write output in a log file in `"../log"`. In case of using parallelism with mpi, then the functions from `GtcStochasticMpi.hpp` will be called.

8.29.2 Function Documentation

8.29.2.1 gtc_stochastic() [1/2]

```
execution::internal::enable_if_execution_policy< ExecutionPolicy, SparseStatus< _TnsSize,
execution::execution_policy_t< ExecutionPolicy >, SparseDefaultValues > > partensor::gtc_←
stochastic (
            ExecutionPolicy && ,
            Matrix const & Ratings_Base_T,
            SparseOptions< _TnsSize, execution::execution_policy_t< ExecutionPolicy >, Sparse←
DefaultValues > const & options )
```

Interface of General Tensor Completion(gtc_stochastic), with the use of an `Execution Policy`, which can be either `sequential`, `parallel` with the use of `MPI`, or `parallel` with the use of `OpenMP`. In order to choose a policy, type `execution::seq`, `execution::mpi` or `execution::omp`. Default value is `sequential`, in case no `ExecutionPolicy` is passed.

Template Parameters

ExecutionPolicy	Type of `stl Execution Policy` (sequential, parallel-mpi).
*Tensor_*	Type(data type and order) of input Tensor. `Tensor_` must be `partensor::Tensor<order>`, where `order` must be in range of [3-8].

Parameters

tnsX	[in] The given Tensor to be factorized of `Tensor_` type, with `double` data.
R	[in] The rank of decomposition.

Returns

An object of type `Status`, containing the results of the algorithm.

8.29.2.2 gtc_stochastic() [2/2]

```
execution::internal::enable_if_execution_policy< ExecutionPolicy, SparseStatus< _TnsSize,
execution::execution_policy_t< ExecutionPolicy >, SparseDefaultValues > > partensor::gtc_←
stochastic (
            ExecutionPolicy && ,
            SparseOptions< _TnsSize, execution::execution_policy_t< ExecutionPolicy >, Sparse←
DefaultValues > const & options )
```

Interface of Stochastic General Tensor Completion(gtc_stochastic), with the use of an `Execution Policy`, which can be either `sequential`, `parallel` with the use of `MPI`, or `parallel` with the use of `OpenMP`. In order to choose a policy, type `execution::seq`, `execution::mpi` or `execution::omp`. Default value is `sequential`, in case no `ExecutionPolicy` is passed.

Template Parameters

ExecutionPolicy	Type of `stl Execution Policy` (sequential, parallel-mpi).
*Tensor_*	Type(data type and order) of input Tensor. `Tensor_` must be `partensor::Tensor<order>`, where `order` must be in range of [3-8].

Parameters

tnsX	[in] The given Tensor to be factorized of `Tensor_` type, with `double` data.
R	[in] The rank of decomposition.

Returns

An object of type `Status`, containing the results of the algorithm.

8.30 GtcStochastic.hpp

Go to the documentation of this file.

```
#ifndef DOXYGEN_SHOULD_SKIP_THIS
#endif // DOXYGEN_SHOULD_SKIP_THIS
/**********************************************************************/
#ifndef PARTENSOR_GTC_STOCHASTIC_HPP
#define PARTENSOR_GTC_STOCHASTIC_HPP

#include "PARTENSOR_basic.hpp"
#include "PartialCwiseProd.hpp"
#include "NesterovMNLS.hpp"
#include "Normalize.hpp"
#include "Timers.hpp"
#include "ReadWrite.hpp"

namespace partensor
{
  inline namespace v1
  {
    namespace internal
    {
      /*
       * Includes the implementation of Stochastic General Tensor Completion. Based on the given
       * parameters one of the overloaded operators will be called.
       */
      template <std::size_t TnsSize_>
      struct GTC_STOCHASTIC_Base
      {
        static constexpr std::size_t TnsSize    = TnsSize_;
        static constexpr std::size_t lastFactor = TnsSize - 1;
        using SparseTensor = typename partensor::SparseTensor<TnsSize_>;
        using DataType     = typename SparseTensorTraits<SparseTensor>::DataType;
        using MatrixType   = typename SparseTensorTraits<SparseTensor>::MatrixType;
        using Dimensions   = typename SparseTensorTraits<SparseTensor>::Dimensions;
        using SparseMatrix = typename SparseTensorTraits<SparseTensor>::SparseMatrixType;
        using LongMatrix   = typename SparseTensorTraits<SparseTensor>::LongMatrixType;
        using Constraints  = typename SparseTensorTraits<SparseTensor>::Constraints;
        using MatrixArray  = typename SparseTensorTraits<SparseTensor>::MatrixArray;
        using DoubleArray  = typename SparseTensorTraits<SparseTensor>::DoubleArray;
        using IntArray     = typename SparseTensorTraits<SparseTensor>::IntArray;
        template<int mode>
        void sort_ratings_base_util(Matrix       const &Ratings_Base_T,
                                    SparseTensor       &tnsX,
                                    IntArray     const &tnsDims,
                                    long int     const  nnz)
        {
          Matrix ratings_base_temp = Ratings_Base_T;
          std::vector<std::vector<double» vectorized_ratings_base;
          vectorized_ratings_base.resize(nnz,std::vector<double>(TnsSize + 1));

          for (int rows = 0; rows < static_cast<int>(TnsSize) + 1; rows++)
          {
            for (int cols = 0; cols < nnz; cols++)
            {
              vectorized_ratings_base[cols][rows] = Ratings_Base_T(rows, cols);
            }
          }
          // Sort
          std::sort(vectorized_ratings_base.begin(), vectorized_ratings_base.end(), SortRows<TnsSize_, mode, double>);

          for (int rows = 0; rows < static_cast<int>(TnsSize) + 1; rows++)
          {
            for (int cols = 0; cols < nnz; cols++)
            {
              ratings_base_temp(rows, cols) = vectorized_ratings_base[cols][rows];
```

```
            }
          }

          FillSparseMatricization<TnsSize>(tnsX, nnz, ratings_base_temp, tnsDims, mode);

          if constexpr (mode+1 < TnsSize)
            sort_ratings_base_util<mode+1>(Ratings_Base_T, tnsX, tnsDims, nnz);
        }

        void sort_ratings_base(Matrix       const &Ratings_Base_T,
                               SparseTensor       &tnsX,
                                IntArray     const &tnsDims,
                                long int     const  nnz)
        {
          ReserveSparseTensor<TnsSize>(tnsX, tnsDims, nnz);
          sort_ratings_base_util<0>(Ratings_Base_T, tnsX, tnsDims, nnz);
        }
      };

      template <std::size_t TnsSize_, typename ExecutionPolicy = execution::sequenced_policy>
      struct GTC_STOCHASTIC : public GTC_STOCHASTIC_Base<TnsSize_>
      {
        using          GTC_STOCHASTIC_Base<TnsSize_>::TnsSize;
        using          GTC_STOCHASTIC_Base<TnsSize_>::lastFactor;
        using typename GTC_STOCHASTIC_Base<TnsSize_>::Dimensions;
        using typename GTC_STOCHASTIC_Base<TnsSize_>::MatrixArray;
        using typename GTC_STOCHASTIC_Base<TnsSize_>::DataType;
        using typename GTC_STOCHASTIC_Base<TnsSize_>::SparseTensor;
        using typename GTC_STOCHASTIC_Base<TnsSize_>::IntArray;
        using typename GTC_STOCHASTIC_Base<TnsSize_>::LongMatrix;

        using Options = partensor::SparseOptions<TnsSize_,execution::sequenced_policy,SparseDefaultValues>;
        using Status  = partensor::SparseStatus<TnsSize_,execution::sequenced_policy,SparseDefaultValues>;

        // Variables that will be used in gtc stochastic implementations.
        struct Member_Variables
        {
          MatrixArray  factors_T;
          MatrixArray  factor_T_factor;
          MatrixArray  mttkrp_T;
          IntArray     tnsDims;
          IntArray     blocksize;
          std::array<std::array<int, TnsSize_ -1>, TnsSize_> offsets;

          MatrixArray  norm_factors_T;
          MatrixArray  old_factors_T;

          Matrix       cwise_factor_product;
          SparseTensor tnsX;
          int          rank;
          double       c_stochastic_perc;

          Member_Variables() = default;

          Member_Variables(int R, IntArray dims) :  tnsDims(dims),
                                                    rank(R)
          {}

          Member_Variables(Member_Variables const &) = default;
          Member_Variables(Member_Variables      &&) = default;

          Member_Variables &operator=(Member_Variables const &) = default;
          Member_Variables &operator=(Member_Variables      &&) = default;
        };

        /*
         * In case option variable @c writeToFile is enabled, then, before the end
         * of the algorithm, it writes the resulted factors in files, whose
         * paths are specified before compiling in @ options.final_factors_path.
         *
         * @param  st  [in] Struct where the returned values of @c GtcStochastic are stored.
         */
        void writeFactorsToFile(Status const &st)
        {
          std::size_t size;
          for(std::size_t i=0; i<TnsSize; ++i)
          {
            size = st.factors[i].rows() * st.factors[i].cols();
            partensor::write(st.factors[i],
                             st.options.final_factors_paths[i],
                             size);
          }
        }

        /*
```

```
  * Compute the cost function value at the end of each outer iteration
  * based on the last factor.
  *
  * @param  mv   [in]     Struct where ALS variables are stored.
  * @param  st   [in,out] Struct where the returned values of @c GtcStochastic are stored.
  *                       In this case the cost function value is updated.
  */
void cost_function(Member_Variables const &mv,
                   Status                 &st)
{
  Matrix temp_R_1(mv.rank, 1);
  double temp_1_1 = 0;
  st.f_value = 0;

  std::array<int,TnsSize-1> offsets;
  offsets[0] = 1;
  for (int j = 1; j < static_cast<int>(TnsSize) - 1; j++)
  {
    offsets[j] = offsets[j - 1] * mv.tnsDims[j-1];
  }

  for (long int i = 0; i < mv.tnsX[lastFactor].outerSize(); ++i)
  {
      int row = 0;
      for (SparseMatrix::InnerIterator it(mv.tnsX[lastFactor], i); it; ++it)
      {
          temp_R_1 = mv.factors_T[lastFactor].col(it.col());
          // Select rows of each factor an compute the Hadamard product of the respective row of the Khatri-Rao product, and the row of factor A_N.
          for (int mode_i = static_cast<int>(TnsSize) - 2; mode_i >= 0; mode_i--)
          {
              row      = ((it.row()) / offsets[mode_i]) % (mv.tnsDims[mode_i]);
              temp_R_1 = temp_R_1.cwiseProduct(mv.factors_T[mode_i].col(row));
          }
          temp_1_1 = it.value() - temp_R_1.sum();
          st.f_value += temp_1_1 * temp_1_1;
      }
  }
}

/*
 * Compute the cost function value at the end of each outer iteration
 * based on the last accelerated factor.
 *
 * @param  mv                [in] Struct where ALS variables are stored.
 * @param  accel_factors     [in] Accelerated factors.
 *
 * @returns The cost function calculated with the accelerated factors.
 */
double accel_cost_function(Member_Variables       const &mv,
                           MatrixArray            const &accel_factors)
{
  Matrix temp_R_1(mv.rank, 1);
  double temp_1_1 = 0;
  double f_value = 0;

  std::array<int,TnsSize-1> offsets;
  offsets[0] = 1;
  for (int j = 1; j < static_cast<int>(TnsSize) - 1; j++)
  {
    offsets[j] = offsets[j - 1] * mv.tnsDims[j-1];
  }

  for (long int i = 0; i < mv.tnsX[lastFactor].outerSize(); ++i)
  {
    int row = 0;
    for (SparseMatrix::InnerIterator it(mv.tnsX[lastFactor], i); it; ++it)
    {
        temp_R_1 = accel_factors[lastFactor].col(it.col());
        // Select rows of each factor an compute the Hadamard product of the respective row of the Khatri-Rao product, and the row of factor A_N.
        // temp_R_1  = A_N(i_N,:) .* ... .* A_2(i_2,:) .* A_1(i_1,:)
        for (int mode_i = static_cast<int>(TnsSize) - 2; mode_i >= 0; mode_i--)
        {
            row      = ((it.row()) / offsets[mode_i]) % (mv.tnsDims[mode_i]);
            temp_R_1 = temp_R_1.cwiseProduct(accel_factors[mode_i].col(row));
        }
        temp_1_1 = it.value() - temp_R_1.sum();
        f_value += temp_1_1 * temp_1_1;
    }
  }
  return f_value;
}

void calculate_offsets(Member_Variables &mv)
{
  for (int idx = 0; idx < static_cast<int>(TnsSize); idx++)
```

```
      {
        mv.offsets[idx][0] = 1;
        for (int j = 1, mode = 0; j < static_cast<int>(TnsSize) - 1; j++, mode++)
        {
          if (idx == mode)
          {
            mode++;
          }
          mv.offsets[idx][j] = mv.offsets[idx][j - 1] * mv.tnsDims[mode];
        }
      }
    }

    void unconstraint_update(int              const  idx,
                             Member_Variables       &mv,
                             Status                 &st)
    {
        int r = mv.rank;

        Matrix eye = st.options.lambdas[idx] * Matrix::Identity(r, r);

        int last_mode = (idx == static_cast<int>(TnsSize) - 1) ? static_cast<int>(TnsSize) - 2 : static_cast<int>(TnsSize) - 1;

        Matrix MTTKRP_col(r, 1);
        Matrix temp_RxR(r, r);
        Matrix temp_R_1(r, 1);

        // Compute MTTKRP
        for (long int i = 0; i < mv.tnsX[idx].outerSize(); ++i)
        {
            MTTKRP_col.setZero();
            temp_RxR.setZero(); // is the Hadamard product of Grammians of the Factors, that correspond to the nnz elements of the Tensor.
            for (SparseMatrix::InnerIterator it(mv.tnsX[idx], i); it; ++it)
            {
                temp_R_1 = Matrix::Ones(r, 1);
                int row;
                // Select rows of each factor an compute the respective row of the Khatri-Rao product.
                for (int mode_i = last_mode, kr_counter = static_cast<int>(TnsSize) - 2; mode_i >= 0 && kr_counter >= 0; mode_i--)
                {
                    if (mode_i == idx)
                    {
                        continue;
                    }
                    row      = ((it.row()) / mv.offsets[idx][kr_counter]) % (mv.tnsDims[mode_i]);
                    temp_R_1 = temp_R_1.cwiseProduct(mv.factors_T[mode_i].col(row));
                    kr_counter--;
                }
                // Subtract from the previous row the respective row of W, according to relation (9).
                MTTKRP_col.noalias() += it.value() * temp_R_1;
                temp_RxR.noalias()   += temp_R_1 * temp_R_1.transpose();
            }
            mv.factors_T[idx].col(i) = (temp_RxR + eye).inverse() * MTTKRP_col;
        }
    }

    /*
     * Based on each factor's constraint, a different
     * update function is used at every outer iteration.
     *
     * Computes also factor^T * factor at the end.
     *
     * @param  idx [in]     Factor to be updated.
     * @param  mv  [in]     Struct where ALS variables are stored.
     * @param  st  [in,out] Struct where the returned values of @c GtcStochastic are stored.
     *                      Updates the @c stl array with the factors.
     */
    void update_factor(int              const  idx,
                       Member_Variables       &mv,
                       Status                 &st    )
    {
      // Update factor
      switch ( st.options.constraints[idx] )
      {
        case Constraint::unconstrained:
        {
          break;
        }
        case Constraint::nonnegativity:
        {
          dynamic_blocksize::StochasticNesterovMNLS(mv.factors_T, mv.tnsDims, mv.tnsX[idx], mv.offsets[idx],
                                                    mv.c_stochastic_perc,
```

```
st.options.max_nesterov_iter, st.options.lambdas[idx], idx);
      break;
    }
    default: // in case of Constraint::constant
      break;
  }

  // Compute A^T * A + B^T * B + ...
  st.factors[idx] = mv.factors_T[idx].transpose();
  mv.factor_T_factor[idx].noalias() = mv.factors_T[idx] * st.factors[idx];
}

/*
 * @brief Line Search Acceleration
 *
 * Performs an acceleration step on the updated factors, and keeps the accelerated factors
 * when the step succeeds. Otherwise, the acceleration step is ignored.
 * Line Search Acceleration reduces the number of outer iterations in the ALS algorithm.
 *
 * @note This implementation ONLY, if factors are of @c Matrix type.
 *
 * @param  mv  [in,out] Struct where ALS variables are stored.
 *                      In case the acceleration step is successful the Gramian
 *                      matrices of factors are updated.
 * @param  st  [in,out] Struct where the returned values of @c GtcStochastic are stored.
 *                      If the acceleration succeeds updates @c factors
 *                      and cost function value.
 *
 */
void line_search_accel(Member_Variables &mv,
                       Status           &st)
{
  double       f_accel    = 0.0; // Objective Value after the acceleration step
  double const accel_step = pow(st.ao_iter+1,(1.0/(st.options.accel_coeff)));

  MatrixArray  accel_factors_T;
  MatrixArray  accel_gramians;

  for(std::size_t i=0; i<TnsSize; ++i)
  {
    accel_factors_T[i] = mv.old_factors_T[i] + accel_step * (mv.factors_T[i] -
mv.old_factors_T[i]);
    accel_gramians[i]  = accel_factors_T[i] * accel_factors_T[i].transpose();
  }

  f_accel = accel_cost_function(mv, accel_factors_T);
  if (st.f_value > f_accel)
  {
    mv.factors_T       = accel_factors_T;
    mv.factor_T_factor = accel_gramians;
    st.f_value         = f_accel;
    Partensor()->Logger()->info("Acceleration Step SUCCEEDED! at iter: {}", st.ao_iter);
  }
  else
    st.options.accel_fail++;

  if (st.options.accel_fail==5)
  {
    st.options.accel_fail=0;
    st.options.accel_coeff++;
  }
}

/*
 * Sequential implementation of Alternating Least Squares (ALS) method.
 *
 * @param  R   [in]     The rank of decomposition.
 * @param  mv  [in]     Struct where ALS variables are stored and being updated
 *                      until a termination condition is true.
 * @param  st  [in,out] Struct where the returned values of @c GtcStochastic are stored.
 */
void aogtc_stochastic(Member_Variables       &mv,
                      Status                 &status)
{

  for (std::size_t i=0; i<TnsSize; i++)
  {
    mv.factors_T[i] = status.factors[i].transpose();
    mv.factor_T_factor[i].noalias() = mv.factors_T[i] * status.factors[i];
    mv.mttkrp_T[i] = Matrix(mv.rank, mv.tnsDims[i]);
  }

  // if(status.options.normalization)
  // {
  //   choose_normilization_factor(status, mv.all_orthogonal, mv.weight_factor);
  // }
  // Normalize(static_cast<int>(R), mv.factor_T_factor, status.factors);
```

```
      status.frob_tns          = (mv.tnsX[0]).squaredNorm();
      cost_function(mv, status);
      status.rel_costFunction = status.f_value/status.frob_tns;

      // ---- Loop until ALS converges ----
      int epoch = int(mv.tnsX[0].nonZeros() / (mv.blocksize[0] * mv.tnsDims[0])) + 1;
      std::cout « "epoch = " « epoch « std::endl;
      std::size_t epoch_counter = 0;

      while(1)
      {
        status.ao_iter++;
        std::cout « "iter: " «  status.ao_iter « " -- fvalue: " « status.f_value « " -- relative_costFunction: " « status.rel_costFunction « std::endl;
        Partensor()->Logger()->info("iter: {} -- fvalue: {} -- relative_costFunction: {}", status.ao_iter,
                                                      status.f_value, status.rel_costFunction);

        for (std::size_t i=0; i<TnsSize; i++)
        {
          mv.cwise_factor_product = PartialCwiseProd(mv.factor_T_factor, i);

          // Update factor
          update_factor(i, mv, status);
        }

                    //      <---------------------  End loop for every mode -------------------------->      //
        // Cost function computation.
        if (status.ao_iter % epoch == 0)
        {
            epoch_counter++;
            cost_function(mv, status);
            status.rel_costFunction = status.f_value/status.frob_tns;
            // std::cout « epoch_counter « " - " « f_value « " - " « f_value / frob_tns « " - " « frob_tns « "\n";
        }

        //  <-----------------------        Terminating condition           ----------------------->      //
        // if (epoch_counter > max_iter || f_value < ao_tol)

        // ---- Terminating condition ----
        if (status.rel_costFunction < status.options.threshold_error || epoch_counter >= status.options.max_iter )
        {
          if(status.options.writeToFile)
            writeFactorsToFile(status);
          break;
        }

        // if (status.options.acceleration)
        // {
        //   mv.norm_factors_T = mv.factors_T;
        //   // ---- Acceleration Step ----
        //   if (status.ao_iter > 1)
        //     line_search_accel(mv, status);

        //   mv.old_factors_T = mv.norm_factors_T;
        // }
      } // end of while
    }

    void initialize_factors(Member_Variables &mv,
                            Status           &status)
    {
      if(status.options.initialized_factors)
      {
        if(status.options.read_factors_from_file)
        {
          for(std::size_t i=0; i<TnsSize; ++i)
          {
            status.factors[i] = Matrix(mv.tnsDims[i], static_cast<int>(mv.rank));
            read( status.options.initial_factors_paths[i],
                  mv.tnsDims[i] * mv.rank,
                  0,
                  status.factors[i] );
          }
        }
        else
          status.factors = status.options.factorsInit;
      }
      else // produce estimate factors using uniform distribution with entries in [0,1].
        makeFactors(mv.tnsDims, status.options.constraints, mv.rank, status.factors);
    }

```

```
        Status operator()(Options const &options)
        {
          Status           status(options);
          Member_Variables mv(options.rank, options.tnsDims);

          long long int fileSize = (TnsSize + 1) * options.nonZeros;

          Matrix Ratings_Base_T = Matrix(static_cast<int>(TnsSize+1), options.nonZeros);
          // Read the whole Tensor from a file
          read( options.ratings_path,
                fileSize,
                0,
                Ratings_Base_T );

          GTC_STOCHASTIC_Base<TnsSize>::sort_ratings_base(Ratings_Base_T, mv.tnsX, options.tnsDims, options.nonZeros);
          Ratings_Base_T.resize(0,0);

          for (std::size_t i=0; i<TnsSize; i++)
          {
            mv.tnsX[i].makeCompressed();
          }

          mv.c_stochastic_perc = options.c_stochastic_perc;

          // c := percentage (%) of dataset to be sampled.
          // blocksize(mode) := c * ( nnz /  tns_dimensions(mode) )
          for (std::size_t i = 0; i < TnsSize; i++)
          {
            mv.blocksize[i] = int(mv.c_stochastic_perc * options.nnz / mv.tnsDims[i]) + 1;
          }

          calculate_offsets(mv);

          initialize_factors(mv, status);

          aogtc_stochastic(mv, status);

          return status;
        }

        Status operator()(Matrix       const &Ratings_Base_T,
                          Options      const &options)
        {
          Status           status(options);
          Member_Variables mv(options.rank, options.tnsDims);

          ReserveSparseTensor<TnsSize>(mv.tnsX, options.tnsDims, options.nonZeros);
          FillSparseTensor<TnsSize>(mv.tnsX, options.nonZeros, Ratings_Base_T, options.tnsDims);
          // Ratings_Base_T.resize(0,0);

          for (std::size_t i=0; i<TnsSize; i++)
          {
            mv.tnsX[i].makeCompressed();
          }

          mv.c_stochastic_perc = options.c_stochastic_perc;

          // c := percentage (%) of dataset to be sampled.
          // blocksize(mode) := c * ( nnz /  tns_dimensions(mode) )
          for (std::size_t i = 0; i < TnsSize; i++)
          {
            mv.blocksize[i] = int(mv.c_stochastic_perc * options.nonZeros / mv.tnsDims[i]) + 1;
          }

          calculate_offsets(mv);

          // produce estimate factors using uniform distribution with entries in [0,1].
          initialize_factors(mv, status);

          aogtc_stochastic(mv, status);

          return status;
        }
      };
    }  // namespace internal
  }    // namespace v1
} // end namespace partensor

#if USE_MPI

#include "GtcStochasticMpi.hpp"
#endif /* USE_MPI */

#if USE_OPENMP

#include "GtcStochasticOpenMP.hpp"
```

```
#endif /* USE_OPENMP */

namespace partensor
{
  template <std::size_t _TnsSize, typename ExecutionPolicy>

   execution::internal::enable_if_execution_policy<ExecutionPolicy,SparseStatus<_TnsSize,execution::execution_policy_t<Exe
  gtc_stochastic( ExecutionPolicy
                 &&,
       SparseOptions<_TnsSize,execution::execution_policy_t<ExecutionPolicy>,SparseDefaultValues>
   const &options  )
  {
    using  ExPolicy = execution::execution_policy_t<ExecutionPolicy>;

    if constexpr (std::is_same_v<ExPolicy,execution::sequenced_policy>)
    {
      return internal::GTC_STOCHASTIC<_TnsSize>()(options);
    }
    else if constexpr (std::is_same_v<ExPolicy,execution::openmpi_policy>)
    {
      return internal::GTC_STOCHASTIC<_TnsSize,execution::openmpi_policy>()(options);
    }
    else if constexpr (std::is_same_v<ExPolicy,execution::openmp_policy>)
    {
      return internal::GTC_STOCHASTIC<_TnsSize,execution::openmp_policy>()(options);
    }
    else
      return internal::GTC_STOCHASTIC<_TnsSize>()(options);
  }

  /*
   * Interface of Stochastic General Tensor Completion(gtc_stochastic). Sequential Policy.
   *
   * @tparam Tensor_      Type(data type and order) of input Tensor.
   *                      @c Tensor_ must be @c partensor::Tensor<order>, where
   *                      @c order must be in range of @c [3-8].
   * @param  tnsX    [in] The given Tensor to be factorized of @c Tensor_ type,
   *                      with @c double data.
   * @param  R       [in] The rank of decomposition.
   *
   * @returns An object of type @c Status, containing the results of the algorithm.
   */
  template<std::size_t _TnsSize>
  auto gtc_stochastic(SparseOptions<_TnsSize> const &options )
  {
    return internal::GTC_STOCHASTIC<_TnsSize,execution::sequenced_policy>()(options);
  }

  template <std::size_t _TnsSize, typename ExecutionPolicy>

   execution::internal::enable_if_execution_policy<ExecutionPolicy,SparseStatus<_TnsSize,execution::execution_policy_t<Exe
  gtc_stochastic( ExecutionPolicy
                 &&,
       Matrix
   const &Ratings_Base_T,
       SparseOptions<_TnsSize,execution::execution_policy_t<ExecutionPolicy>,SparseDefaultValues>
   const &options  )
  {
    using  ExPolicy = execution::execution_policy_t<ExecutionPolicy>;

    if constexpr (std::is_same_v<ExPolicy,execution::sequenced_policy>)
    {
      return internal::GTC_STOCHASTIC<_TnsSize>()(Ratings_Base_T,options);
    }
    else if constexpr (std::is_same_v<ExPolicy,execution::openmpi_policy>)
    {
      return internal::GTC_STOCHASTIC<_TnsSize,execution::openmpi_policy>()(Ratings_Base_T,options);
    }
    else if constexpr (std::is_same_v<ExPolicy,execution::openmp_policy>)
    {
      return internal::GTC_STOCHASTIC<_TnsSize,execution::openmp_policy>()(Ratings_Base_T,options);
    }
    else
      return internal::GTC_STOCHASTIC<_TnsSize>()(Ratings_Base_T,options);
  }

  /*
   * Interface of Stochastic General Tensor Completion(gtc_stochastic). Sequential Policy.
   *
   * @tparam Tensor_      Type(data type and order) of input Tensor.
   *                      @c Tensor_ must be @c partensor::Tensor<order>, where
   *                      @c order must be in range of @c [3-8].
   * @param  tnsX    [in] The given Tensor to be factorized of @c Tensor_ type,
   *                      with @c double data.
   * @param  R       [in] The rank of decomposition.
   *
   * @returns An object of type @c Status, containing the results of the algorithm.
```

```
  */
template<std::size_t _TnsSize>
auto gtc_stochastic(Matrix                              const &Ratings_Base_T,
                    SparseOptions<_TnsSize>             const &options )
{
  return internal::GTC_STOCHASTIC<_TnsSize,execution::sequenced_policy>()(Ratings_Base_T,options);
}

}

#endif // PARTENSOR_GTC_STOCHASTIC_HPPP
```

8.31 GtcStochasticMpi.hpp File Reference

Classes

- struct GTC_STOCHASTIC< TnsSize_, execution::openmpi_policy >

8.31.1 Detailed Description

Implements the Canonical Polyadic Decomposition(gtc) using `MPI`. Make use of `spdlog` library in order to write output in a log file in `"../log"`.

8.32 GtcStochasticMpi.hpp

Go to the documentation of this file.

```
#ifndef DOXYGEN_SHOULD_SKIP_THIS
#endif // DOXYGEN_SHOULD_SKIP_THIS
/**********************************************************************/
#if !defined(PARTENSOR_GTC_STOCHASTIC_HPP)
#error "GTC_STOCHASTIC_MPI can only included inside GTC"
#endif /* PARTENSOR_GTC_STOCHASTIC_HPP */

namespace partensor
{

  inline namespace v1 {

    namespace internal {
      template<std::size_t TnsSize_>
      struct GTC_STOCHASTIC<TnsSize_,execution::openmpi_policy> : public GTC_STOCHASTIC_Base<TnsSize_>
      {
        using          GTC_STOCHASTIC_Base<TnsSize_>::TnsSize;
        using          GTC_STOCHASTIC_Base<TnsSize_>::lastFactor;
        using typename GTC_STOCHASTIC_Base<TnsSize_>::Dimensions;
        using typename GTC_STOCHASTIC_Base<TnsSize_>::MatrixArray;
        using typename GTC_STOCHASTIC_Base<TnsSize_>::DataType;
        using typename GTC_STOCHASTIC_Base<TnsSize_>::SparseTensor;
        using typename GTC_STOCHASTIC_Base<TnsSize_>::LongMatrix;

        using IntArray         = typename SparseTensorTraits<SparseTensor>::IntArray;
        using CartCommunicator = partensor::cartesian_communicator; // From ParallelWrapper.hpp
        using CartCommVector   = std::vector<CartCommunicator>;
        using IntVector        = std::vector<int>;
        using Int2DVector      = std::vector <std::vector<int»;

        using Options = partensor::SparseOptions<TnsSize_,execution::openmpi_policy,SparseDefaultValues>;
        using Status  = partensor::SparseStatus<TnsSize_,execution::openmpi_policy,SparseDefaultValues>;

        // Variables that will be used in gtc implementations.
        struct Member_Variables
        {
          MPI_Communicator &world = Partensor()->MpiCommunicator(); // MPI_COMM_WORLD

          double          local_f_value;
          int             RxR;
          int             world_size;
          double          c_stochastic_perc;

```

```
        Int2DVector     displs_subTns;       // skipping dimension "rows" for each subtensor
        Int2DVector     displs_subTns_R;     // skipping dimension "rows" for each subtensor times R ( for MPI communication purposes )
        Int2DVector     subTnsDims;          // dimensions of subtensor
        Int2DVector     subTnsDims_R;        // dimensions of subtensor times R ( for MPI communication purposes )
        Int2DVector     displs_local_update; // displacement in the local factor for update rows
        Int2DVector     send_recv_counts;    // rows to be communicated after update times R

        CartCommVector  layer_comm;
        CartCommVector  fiber_comm;

        IntArray        layer_rank;
        IntArray        fiber_rank;
        IntArray        rows_for_update;
        IntArray        tnsDims;

        MatrixArray     layer_factors;
        MatrixArray     layer_factors_T;
        MatrixArray     factors_T;
        MatrixArray     factor_T_factor;
        MatrixArray     local_factors_T;
        MatrixArray     norm_factors_T;
        MatrixArray     old_factors_T;

        Matrix          temp_matrix;
        Matrix          Ratings_Base_T;
        SparseTensor    subTns;

        int             rank;
        std::array<std::array<int, TnsSize-1>, TnsSize> offsets;

         /*
          * Calculates if the number of processors given from terminal
          * are equal to the processors in the implementation.
          *
          * @param procs [in] @c stl array with the number of processors per
          *                   dimension of the tensor.
          */
         void check_processor_avaliability(std::array<int, TnsSize> const &procs)
         {
           // MPI_Environment &env = Partensor()->MpiEnvironment();
           world_size = world.size();
           //  numprocs must be product of options.proc_per_mode
           if (std::accumulate(procs.begin(), procs.end(), 1,
                               std::multiplies<int>()) != world_size && world.rank() == 0) {
             Partensor()->Logger()->error("The product of the processors per mode must be equal to {}\n", world_size);
             // env.abort(-1);
           }
         }

         Member_Variables() = default;
         Member_Variables(int R, IntArray dims, std::array<int, TnsSize> &procs) :  local_f_value(0.0),
                                                                                    RxR(R*R),
                                                                                    displs_subTns(TnsSize),
                                                                                    displs_subTns_R(TnsSize),
                                                                                    subTnsDims(TnsSize),
                                                                                    subTnsDims_R(TnsSize),
                                                                                    displs_local_update(TnsSize),
                                                                                    send_recv_counts(TnsSize),
                                                                                    tnsDims(dims),
                                                                                    rank(R)
         {
           check_processor_avaliability(procs);
           layer_comm.reserve(TnsSize);
           fiber_comm.reserve(TnsSize);
         }

         Member_Variables(Member_Variables const &) = default;
         Member_Variables(Member_Variables      &&) = default;

         Member_Variables &operator=(Member_Variables const &) = default;
         Member_Variables &operator=(Member_Variables      &&) = default;
       };

       template<int mode>
       void sort_ratings_base_util(Matrix           const &Ratings_Base_T,
                                   long int         const  nnz,
                                   Member_Variables       &mv)
```

```
    {
      Matrix ratings_base_temp = Ratings_Base_T;
      std::vector<std::vector<double» vectorized_ratings_base;
      vectorized_ratings_base.resize(nnz, std::vector<double>(TnsSize + 1));

      for (int rows = 0; rows < static_cast<int>(TnsSize) + 1; rows++)
      {
        for (int cols = 0; cols < nnz; cols++)
        {
          vectorized_ratings_base[cols][rows] = Ratings_Base_T(rows, cols);
        }
      }

      // Sort
      std::sort(vectorized_ratings_base.begin(), vectorized_ratings_base.end(), SortRows<TnsSize_, mode, double>);

      for (int rows = 0; rows < static_cast<int>(TnsSize) + 1; rows++)
      {
        for (int cols = 0; cols < nnz; cols++)
        {
          ratings_base_temp(rows, cols) = vectorized_ratings_base[cols][rows];
        }
      }

      Dist_NNZ_sorted<TnsSize>(mv.subTns, nnz, mv.displs_subTns, mv.fiber_rank, ratings_base_temp, mv.subTnsDims, mode);
      mv.subTns[mode].makeCompressed();

      if constexpr (mode+1 < TnsSize)
        sort_ratings_base_util<mode+1>(Ratings_Base_T, nnz, mv);
    }

    void sort_ratings_base(Matrix           const &Ratings_Base_T,
                           long int         const  nnz,
                           Member_Variables       &mv)
    {
      ReserveSparseTensor<TnsSize>(mv.subTns, mv.subTnsDims, mv.fiber_rank, mv.world_size, nnz);
      sort_ratings_base_util<0>(Ratings_Base_T, nnz, mv);
    }


    void NesterovMNLS_stochastic( Member_Variables                   &mv,
                                  Status                       const &st,
                                  int    const                        idx)
    {
      int iter = 0;
                             double L2;
                             double sqrt_q = 0, beta = 0;
      double lambda = st.options.lambdas[idx];
      int rows_layer = mv.subTnsDims[idx][mv.fiber_rank[idx]];

                           Matrix inv_L2(rows_layer, 1); // rows_layer

      Matrix grad_Y_T(mv.rank, rows_layer);
      Matrix grad_Y_local_T(mv.rank, mv.rows_for_update[idx]);

      Matrix Y_T(mv.rank, rows_layer);
      Matrix Y_local_T(mv.rank, mv.rows_for_update[idx]);

      Matrix new_A(mv.rank, mv.rows_for_update[idx]);
      Matrix A(mv.rank, mv.rows_for_update[idx]);

                            const Matrix zero_mat = Matrix::Zero(mv.rank, mv.rank);
                            const Matrix zero_vec = Matrix::Zero(mv.rank, 1);

      A         = mv.local_factors_T[idx];
      Y_T       = mv.layer_factors_T[idx]; // layer_factor
      Y_local_T = A;

      int last_mode = (idx == static_cast<int>(TnsSize) - 1) ? static_cast<int>(TnsSize) - 2 : static_cast<int>(TnsSize) - 1;

      Matrix temp_R_1(mv.rank, 1);
                             Matrix temp_RxR(mv.rank, mv.rank);
      Matrix temp_col(mv.rank, 1);

                             std::srand(std::time(nullptr));

      while (1)
      {
        grad_Y_T.setZero();

        if (iter >= st.options.max_nesterov_iter)
        {
          break;
        }
```

```

          // Compute grad_Y
          for (long int i = 0; i < mv.subTns[idx].outerSize(); ++i)
          {
            temp_col.setZero();

            SparseMatrix::InnerIterator it(mv.subTns[idx], i);

            // Get the number of nnz per row of matricization.
            long int nnzs_per_col = mv.subTns[idx].innerVector(i).nonZeros();

                                                long int var_blocksize_i = static_cast<long int>(mv.c_stochastic_perc * nnzs_per_col);

            if (var_blocksize_i > 0)
            {
              // Choose a pivot from [0, nnzs_per_col - blocksize].
              long int pivot = (var_blocksize_i > nnzs_per_col) ? 0 : (std::rand() % (nnzs_per_col - var_blocksize_i + 1));

              SparseMatrix::InnerIterator it(mv.subTns[idx], i);

              // Iterate over [pivot, pivot + blocksize] nnz elements per row.
              // it += pivot;
              for (long int acuum = 0; acuum < pivot; acuum++)
                ++it;

              temp_RxR.setZero();

              for (long int sample = 0; sample < var_blocksize_i; sample++, ++it)
              {
                temp_R_1 = Matrix::Ones(mv.rank, 1);
                // Select rows of each factor an compute the respective row of the Khatri-Rao product.
                for (int mode_i = last_mode, kr_counter = static_cast<int>(TnsSize) - 2; mode_i >= 0 && kr_counter >= 0; mode_i--)
                {
                  if (mode_i == idx)
                  {
                    continue;
                  }
                  long int row;
                  row      = ((it.row()) / mv.offsets[idx][kr_counter]) % mv.subTnsDims[mode_i][mv.fiber_rank[mode_i]];
                  temp_R_1 = temp_R_1.cwiseProduct(mv.layer_factors_T[mode_i].col(row));
                  kr_counter--;
                }
                // Computation of row of Z according the relation (10) of the paper.
                temp_col += ((temp_R_1.transpose() * Y_T.col(i))(0)  - it.value()) * temp_R_1;

                temp_RxR.noalias() += (temp_R_1 * temp_R_1.transpose());
              }

              grad_Y_T.col(i) = temp_col;

              all_reduce( mv.layer_comm[idx],
                          inplace(temp_RxR.data()),
                          mv.RxR,
                          std::plus<double>() );
            }
            else
            {
              all_reduce( mv.layer_comm[idx],
                          zero_mat.data(),
                          mv.RxR,
                          temp_RxR.data(),
                          std::plus<double>() );

            }

            L2 = PowerMethod(temp_RxR, 1e-3);
            L2 += lambda;
            inv_L2(i) = 1 / L2;
          } // end of for loop on layer_rows

          // Add each process' results and scatter the block rows among the processes in the layer.
          // MPI_Reduce_scatter(grad_Y_T.data(), grad_Y_local_T.data(), send_recv_counts, MPI_DOUBLE, MPI_SUM, mode_layer_comm);
          v2::reduce_scatter( mv.layer_comm[idx],
                              grad_Y_T,
                              mv.send_recv_counts[idx][0],
                              grad_Y_local_T );

          // Add proximal term.
          grad_Y_local_T += lambda * (Y_local_T + A);

          for (long int i=0; i<mv.rows_for_update[idx]; i++)
          {
```

```
           long int translate_i = i + mv.displs_local_update[idx][mv.layer_rank[idx]]/mv.rank;

           new_A.col(i) = (Y_local_T.col(i) - grad_Y_local_T.col(i) * inv_L2(translate_i)).cwiseMax(zero_vec);

           sqrt_q = sqrt( lambda * inv_L2(translate_i) );
           beta = (1 - sqrt_q) / (1 + sqrt_q);

           // Update Y
           Y_local_T.col(i) = (1 + beta) * new_A.col(i) - beta * A.col(i);
         }

         // The updated block rows of Y are all gathered, and we have the whole updated Y of the layer.
         v2::all_gatherv( mv.layer_comm[idx],
                         Y_local_T,
                         mv.send_recv_counts[idx][mv.layer_rank[idx]],
                         mv.send_recv_counts[idx][0],
                         mv.displs_local_update[idx][0],
                         Y_T );

         A = new_A;

         iter++;
       }
       mv.local_factors_T[idx] = A;
     }

     /*
      * In case option variable @c writeToFile is enabled then, before the end
      * of the algorithm writes the resulted factors in files, where their
      * paths are specified before compiling in @ options.final_factors_path.
      *
      * @param  st  [in] Struct where the returned values of @c Gtc are stored.
      */
     void writeFactorsToFile(Status const &st)
     {
       std::size_t size;
       for(std::size_t i=0; i<TnsSize; ++i)
       {
         size = st.factors[i].rows() * st.factors[i].cols();
         partensor::write(st.factors[i],
                          st.options.final_factors_paths[i],
                          size);
       }
     }

     /*
      * Compute the cost function value at the end of each outer iteration
      * based on the last factor.
      *
      * @param  grid_comm [in]     MPI communicator where the new cost function value
      *                            will be communicated and computed.
      * @param  mv        [in]     Struct where ALS variables are stored.
      * @param  st        [in,out] Struct where the returned values of @c Gtc are stored.
      *                            In this case the cost function value is updated.
      */
     void cost_function( CartCommunicator const &grid_comm,
                         Member_Variables       &mv,
                         Status                 &st )
     {
         Matrix temp_R_1(mv.rank, 1);
         double temp_1_1  = 0;
         mv.local_f_value = 0;
         std::array<int, TnsSize-1> offsets;
         offsets[0] = 1;
         for (int j = 1; j < static_cast<int>(TnsSize) - 1; j++)
         {
           offsets[j] = offsets[j - 1] * mv.subTnsDims[j-1][mv.fiber_rank[j-1]]; // mv.layer_factors_T[j - 1].cols()
         }

         for (long int i = 0; i < mv.subTns[lastFactor].outerSize(); ++i)
         {
             int row;
             for (SparseMatrix::InnerIterator it(mv.subTns[lastFactor], i); it; ++it)
             {
                 temp_R_1 = mv.layer_factors_T[lastFactor].col(it.col());
                 // Select rows of each factor an compute the Hadamard product of the respective row of the Khatri-Rao product, and the row of factor A_N.
                 // temp_R_1  = A_N(i_N,:) .* ... .* A_2(i_2,:) .* A_1(i_1,:)
                 for (int mode_i = static_cast<int>(TnsSize) - 2; mode_i >= 0; mode_i--)
                 {
                     row = ((it.row()) / offsets[mode_i]) % (mv.subTnsDims[mode_i][mv.fiber_rank[mode_i]]);
                     temp_R_1.noalias() = temp_R_1.cwiseProduct(mv.layer_factors_T[mode_i].col(row));
                 }
```

```
                    temp_1_1 = it.value() - temp_R_1.sum();
                    mv.local_f_value += temp_1_1 * temp_1_1;
                }
            }

            all_reduce( grid_comm,
                        mv.local_f_value,
                        st.f_value,
                        std::plus<double>() );
        }

        /*
         * Compute the cost function value at the end of each outer iteration
         * based on the last accelerated factor.
         *
         * @param  grid_comm         [in] MPI communicator where the new cost function value
         *                                will be communicated and computed.
         * @param  mv                [in] Struct where ALS variables are stored.
         * @param  st                [in] Struct where the returned values of @c Gtc are stored.
         *                                In this case the cost function value is updated.
         * @param  factors           [in] Accelerated factors.
         * @param  factors_T_factors [in] Gramian matrices of factors.
         *
         * @returns The cost function calculated with the accelerated factors.
         */
        double accel_cost_function(CartCommunicator const &grid_comm,
                                   Member_Variables const &mv,
                                   MatrixArray      const &layer_factors_T)
        {
          Matrix temp_R_1(mv.rank, 1);
          double temp_1_1 = 0;
          double f_value = 0;

          std::array<int, TnsSize-1> offsets;
          offsets[0] = 1;
          for (int j = 1; j < static_cast<int>(TnsSize) - 1; j++)
          {
            offsets[j] = offsets[j - 1] * mv.subTnsDims[j-1][mv.fiber_rank[j-1]];
          }

          for (long int i = 0; i < mv.subTns[lastFactor].outerSize(); ++i)
          {
              int row;
              for (SparseMatrix::InnerIterator it(mv.subTns[lastFactor], i); it; ++it)
              {
                  temp_R_1 = layer_factors_T[lastFactor].col(it.col());
                  // Select rows of each factor an compute the Hadamard product of the respective row of the Khatri-Rao product, and the row of factor A_N.
                  // temp_R_1  = A_N(i_N,:) .* ... .* A_2(i_2,:) .* A_1(i_1,:)
                  for (int mode_i = static_cast<int>(TnsSize) - 2; mode_i >= 0; mode_i--)
                  {
                      row = ((it.row()) / offsets[mode_i]) % (mv.subTnsDims[mode_i][mv.fiber_rank[mode_i]]);
                      temp_R_1.noalias() = temp_R_1.cwiseProduct(layer_factors_T[mode_i].col(row));
                  }
                  temp_1_1 = it.value() - temp_R_1.sum();
                  f_value += temp_1_1 * temp_1_1;
              }
          }

          all_reduce( grid_comm,
                      inplace(&f_value),
                      1,
                      std::plus<double>() );

          return f_value;
        }

        /*
         * Make use of the dimensions and the number of processors per dimension
         * and then calculates the dimensions of the subtensor and subfactor for
         * each processor.
         *
         * @tparam Dimensions          Array type containing the length of Tensor's dimensions.
         *
         * @param  tnsDims    [in]     Tensor Dimensions. Each index contains the corresponding
         *                             factor's rows length.
         * @param  st         [in]     Struct where the returned values of @c Gtc are stored.
         * @param  R          [in]     The rank of decomposition.
         * @param  mv         [in,out] Struct where ALS variables are stored.
         *                             Updates @c stl arrays with dimensions for subtensors and
         *                             subfactors.
         */
        void compute_sub_dimensions(Status           const &st,
                                    Member_Variables       &mv)
        {
          for (std::size_t i = 0; i < TnsSize; ++i)
```

```
      {
        mv.factor_T_factor[i].noalias() = st.factors[i].transpose() * st.factors[i];

        DisCount(mv.displs_subTns[i], mv.subTnsDims[i], st.options.proc_per_mode[i], mv.tnsDims[i], 1);
        // for fiber communication and Gatherv
        DisCount(mv.displs_subTns_R[i], mv.subTnsDims_R[i], st.options.proc_per_mode[i], mv.tnsDims[i], static_cast<int>(mv.rank));
        // information per layer
        DisCount(mv.displs_local_update[i], mv.send_recv_counts[i], mv.world_size / st.options.proc_per_mode[i],
                                                   mv.subTnsDims[i][mv.fiber_rank[i]], static_cast<int>(mv.rank));

        mv.rows_for_update[i] = mv.send_recv_counts[i][mv.layer_rank[i]] / static_cast<int>(mv.rank);
      }

      calculate_offsets(mv);
    }

    void calculate_offsets(Member_Variables &mv)
    {
      for (int idx = 0; idx < static_cast<int>(TnsSize); idx++)
      {
        mv.offsets[idx][0] = 1;
        for (int j = 1, mode = 0; j < static_cast<int>(TnsSize) - 1; j++, mode++)
        {
          if (idx == mode)
          {
            mode++;
          }
          mv.offsets[idx][j] = mv.offsets[idx][j - 1] * mv.subTnsDims[mode][mv.fiber_rank[mode]];
        }
      }
    }

    /*
     * Based on each factor's constraint, a different
     * update function is used at every outer iteration.
     *
     * Computes also factor^T * factor at the end.
     *
     * @param  idx [in]     Factor to be updated.
     * @param  R   [in]     The rank of decomposition.
     * @param  st  [in]     Struct where the returned values of @c Gtc are stored.
     *                      Here constraints and options variables are needed.
     * @param  mv  [in,out] Struct where ALS variables are stored.
     *                      Updates the factors of each layer.
     */
    void update_factor(int              const  idx,
                       Status           const &st,
                       Member_Variables       &mv  )
    {
      switch ( st.options.constraints[idx] )
      {
        case Constraint::unconstrained:
        {
          break;
        }
        case Constraint::nonnegativity:
        {
          NesterovMNLS_stochastic(mv, st, idx);
          break;
        }
        case Constraint::sparsity:
          break;
        default: // in case of Constraint::constant
          break;
      } // end of constraints switch

      if (st.options.constraints[idx] != Constraint::constant)
      {
        v2::all_gatherv( mv.layer_comm[idx],
                         mv.local_factors_T[idx],
                         mv.send_recv_counts[idx][mv.layer_rank[idx]],
                         mv.send_recv_counts[idx][0],
                         mv.displs_local_update[idx][0],
                         mv.layer_factors_T[idx] );

        mv.layer_factors[idx]             = mv.layer_factors_T[idx].transpose();
        mv.factor_T_factor[idx].noalias() = mv.layer_factors_T[idx] * mv.layer_factors[idx];
      }
    }

    /*
     * At the end of the algorithm processor 0
```

```
         * collects each part of the factor that each
         * processor holds and return them in status.factors.
         *
         * @param  mv      [in]     Struct where ALS variables are stored.
         *                          Use variables to compute result factors by gathering each
         *                          part of the factor from processors.
         * @param  st      [in,out] Struct where the returned values of @c Gtc are stored.
         *                          Stores the resulted factors.
         */
        void gather_final_factors(Member_Variables       &mv,
                                  Status                 &st)
        {
          for(std::size_t i=0; i<TnsSize; ++i)
          {
            mv.temp_matrix.resize(static_cast<int>(mv.rank), mv.tnsDims[i]);
            // Gatherv from all processors to processor with rank 0 the final factors
            v2::gatherv( mv.fiber_comm[i],
                         mv.layer_factors_T[i],
                         mv.subTnsDims_R[i][mv.fiber_rank[i]],
                         mv.subTnsDims_R[i][0],
                         mv.displs_subTns_R[i][0],
                         0,
                         mv.temp_matrix );

            st.factors[i] = mv.temp_matrix.transpose();
          }
        }

        /*
         * @brief Line Search Acceleration
         *
         * Performs an acceleration step in the updated factors, and keeps the accelerated factors when
         * the step succeeds. Otherwise, the acceleration step is ignored.
         * Line Search Acceleration reduces the number outer iterations in the ALS algorithm.
         *
         * @note This implementation ONLY, if factors are of @c Matrix type.
         *
         * @param  grid_comm [in]     MPI communicator where the new cost function value
         *                            will be communicated and computed.
         * @param  mv        [in,out] Struct where ALS variables are stored.
         *                            In case the acceration is successful layer factor^T * factor
         *                            and layer factor variables are updated.
         * @param  st        [in,out] Struct where the returned values of @c Gtc are stored.
         *                            If the acceleration succeeds updates cost function value.
         *
         */
        void line_search_accel(CartCommunicator const &grid_comm,
                               Member_Variables       &mv,
                               Status                 &st)
        {
          double       f_accel    = 0.0; // Objective Value after the acceleration step
          double const accel_step = pow(st.ao_iter+1,(1.0/(st.options.accel_coeff)));

          MatrixArray   accel_factors_T;
          MatrixArray   accel_gramians;

          for(std::size_t i=0; i<TnsSize; ++i)
          {
            accel_factors_T[i] = mv.old_factors_T[i] + accel_step * (mv.layer_factors_T[i] - mv.old_factors_T[i]);
            accel_gramians[i]   = accel_factors_T[i] * accel_factors_T[i].transpose();
            all_reduce( mv.fiber_comm[i],
                        inplace(accel_gramians[i].data()),
                        mv.RxR,
                        std::plus<double>() );
          }

          f_accel = accel_cost_function(grid_comm, mv, accel_factors_T);
          if (st.f_value > f_accel)
          {
            mv.layer_factors_T = accel_factors_T;
            mv.factor_T_factor = accel_gramians;
            st.f_value          = f_accel;
            if(grid_comm.rank() == 0)
              Partensor()->Logger()->info("Acceleration Step SUCCEEDED! at iter: {}", st.ao_iter);
          }
          else
            st.options.accel_fail++;

          if (st.options.accel_fail==5)
          {
            st.options.accel_fail=0;
            st.options.accel_coeff++;
          }
        }

        /*
```

```
        * Parallel implementation of als method with MPI.
        *
        * @tparam Dimensions         Array type containing the Tensor dimensions.
        *
        * @param  grid_comm [in]     The communication grid, where the processors
        *                            communicate their cost function.
        * @param  tnsDims   [in]     Tensor Dimensions. Each index contains the corresponding
        *                            factor's rows length.
        * @param  R         [in]     The rank of decomposition.
        * @param  mv        [in]     Struct where ALS variables are stored and being updated
        *                            until a termination condition is true.
        * @param  status    [in,out] Struct where the returned values of @c Gtc are stored.
        */
       void aogtc_stochastic(CartCommunicator const &grid_comm,
                 Member_Variables       &mv,
                 Status                 &status)
       {
         status.frob_tns = (mv.subTns[0]).squaredNorm();
               all_reduce( grid_comm,
                     inplace(&status.frob_tns),
                     1,
                     std::plus<double>() );

         cost_function(grid_comm, mv, status);
         status.rel_costFunction = status.f_value / status.frob_tns;

           std::size_t epoch = static_cast<std::size_t> (1/mv.c_stochastic_perc);
           std::size_t epoch_counter = 0;

         for (int i=0; i< static_cast<int>(TnsSize); i++)
         {
           mv.factor_T_factor[i].noalias() = mv.layer_factors_T[i] * mv.layer_factors[i];
           all_reduce( mv.fiber_comm[i],
                       inplace(mv.factor_T_factor[i].data()),
                       mv.RxR,
                       std::plus<double>() );
         }

         // Wait for all processors to reach here
         grid_comm.barrier();

         // ---- Loop until ALS converges ----
         while(1)
         {
             status.ao_iter++;
             if (!grid_comm.rank())
             {
                 Partensor()->Logger()->info("iter: {} -- fvalue: {} -- relative_costFunction: {}",
status.ao_iter,
                                                status.f_value, status.rel_costFunction);
             }

             //        <-----------------------        loop for every mode
--------------------------->    //
             for (std::size_t i = 0; i < TnsSize; i++)
             {
               // Partition rows of subfactor to the processes in the respective layer.
               mv.local_factors_T[i] = mv.layer_factors_T[i].block(0,
mv.displs_local_update[i][mv.layer_rank[i]] / mv.rank, mv.rank, mv.rows_for_update[i]);

               update_factor(i, status, mv);
             }

             // ---- Cost function Computation ----
             if (status.ao_iter % epoch == 0)
             {
                 epoch_counter++;
                 cost_function(grid_comm, mv, status);
                 status.rel_costFunction = status.f_value / status.frob_tns;
             }

             // if(status.options.normalization && !mv.all_orthogonal)
             //   Normalize(mv.weight_factor, mv.rank, mv.factor_T_factor, mv.layer_factors);

             // ---- Terminating condition ----
             if (status.rel_costFunction < status.options.threshold_error || epoch_counter >=
status.options.max_iter)
             {
                 gather_final_factors(mv, status);
                 if(grid_comm.rank() == 0)
                 {
                   Partensor()->Logger()->info("Processor 0 collected all {} factors.\n", TnsSize);
                   if(status.options.writeToFile)
                     writeFactorsToFile(status);
                 }
                 break;
             }
```

```

            if (status.options.acceleration)
            {
              mv.norm_factors_T = mv.layer_factors_T;
              // ---- Acceleration Step ----
              if (status.ao_iter > 1)
                line_search_accel(grid_comm, mv, status);

              mv.old_factors_T = mv.norm_factors_T;
            }

        } // end of outer while loop
      }

      void initialize_factors(Member_Variables &mv,
                              Status           &status)
      {
        if(status.options.initialized_factors)
        {
          if(status.options.read_factors_from_file)
          {
            for(std::size_t i=0; i<TnsSize; ++i)
            {
              status.factors[i] = Matrix(mv.tnsDims[i], static_cast<int>(mv.rank));
              read( status.options.initial_factors_paths[i],
                    mv.tnsDims[i] * mv.rank,
                    0,
                    status.factors[i] );
            }
          }
          else
            status.factors = status.options.factorsInit;
        }
        else
          makeFactors(mv.tnsDims, status.options.constraints, mv.rank, status.factors);
      }

      Status operator()(Options const &options)
      {
        Status           status(options);
        Member_Variables mv(options.rank, options.tnsDims, status.options.proc_per_mode);

       // Communicator with cartesian topology
        CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);

        // Functions that create layer and fiber grids.
        create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
        create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);

        mv.c_stochastic_perc = options.c_stochastic_perc;

        // produce estimate factors using uniform distribution with entries in [0,1].
        initialize_factors(mv, status);

        compute_sub_dimensions(status, mv);

        for (std::size_t i = 0; i < TnsSize; ++i)
        {
          mv.layer_factors[i]             =
    status.factors[i].block(mv.displs_subTns[i][mv.fiber_rank[i]], 0,
                                                mv.subTnsDims[i][mv.fiber_rank[i]],  mv.rank);
          mv.layer_factors_T[i]           = mv.layer_factors[i].transpose();
          mv.factor_T_factor[i].noalias() = mv.layer_factors_T[i] * mv.layer_factors[i];
        }

        long long int fileSize = (TnsSize + 1) * options.nonZeros;

        // Matrix Ratings_Base = Matrix(options.nonZeros, static_cast<int>(TnsSize+1));
        Matrix Ratings_Base_T = Matrix(static_cast<int>(TnsSize+1), options.nonZeros);

        // Read the whole Tensor from a file
        read( options.ratings_path,
              fileSize,
              0,
              Ratings_Base_T );

        // Matrix Ratings_Base_T = Ratings_Base.transpose();

        sort_ratings_base(Ratings_Base_T, options.nonZeros, mv);
              Ratings_Base_T.resize(0,0);
        // Ratings_Base.resize(0,0);

        aogtc_stochastic(grid_comm, mv, status);

        return status;
      }

```

```
        Status operator()(Matrix        const &Ratings_Base_T,
                          Options       const &options)
        {
          Status           status(options);
          Member_Variables mv(options.rank, options.tnsDims, status.options.proc_per_mode);

         // Communicator with cartesian topology
          CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);

          // Functions that create layer and fiber grids.
          create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
          create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);

          mv.c_stochastic_perc = options.c_stochastic_perc;

          // produce estimate factors using uniform distribution with entries in [0,1].
          initialize_factors(mv, status);

          compute_sub_dimensions(status, mv);

          for (std::size_t i = 0; i < TnsSize; ++i)
          {
            mv.layer_factors[i]              =
status.factors[i].block(mv.displs_subTns[i][mv.fiber_rank[i]], 0,
                                                                  mv.subTnsDims[i][mv.fiber_rank[i]],  mv.rank);
            mv.layer_factors_T[i]            = mv.layer_factors[i].transpose();
            mv.factor_T_factor[i].noalias() = mv.layer_factors_T[i] * mv.layer_factors[i];
          }

          // Each processor takes a subtensor from tnsX
          ReserveSparseTensor<TnsSize>(mv.subTns, mv.subTnsDims, mv.fiber_rank, mv.world_size,
options.nonZeros);

          Dist_NNZ<TnsSize>(mv.subTns, options.nonZeros, mv.displs_subTns, mv.fiber_rank,
Ratings_Base_T, mv.subTnsDims);
                // Ratings_Base_T.resize(0,0);

          for(int mode_i = 0; mode_i < static_cast<int>(TnsSize); mode_i++)
          {
            mv.subTns[mode_i].makeCompressed();
          }

          aogtc_stochastic(grid_comm, mv, status);

          return status;
        }

      };
    }  // namespace internal
  }    // namespace v1

} // end namespace partensor
```

8.33 GtcStochasticOpenMP.hpp File Reference

8.33.1 Detailed Description

Implements the Canonical Polyadic Decomposition(gtc) using `OpenMP`. Make use of `spdlog` library in order to write output in a log file in `"../log"`.

8.34 GtcStochasticOpenMP.hpp

Go to the documentation of this file.

```
#ifndef DOXYGEN_SHOULD_SKIP_THIS
#endif // DOXYGEN_SHOULD_SKIP_THIS
/*************************************************************************/
#if !defined(PARTENSOR_GTC_STOCHASTIC_HPP)
#error "GTC_STOCHASTIC_OMP can only included inside GTC_STOCHASTIC"
#endif /* PARTENSOR_GTC_STOCHASTIC_HPP */

namespace partensor
{
  inline namespace v1
```

```
{
  namespace internal
  {
    template <std::size_t TnsSize_>
    struct GTC_STOCHASTIC<TnsSize_, execution::openmp_policy> : public GTC_STOCHASTIC_Base<TnsSize_>
    {
      using          GTC_STOCHASTIC_Base<TnsSize_>::TnsSize;
      using          GTC_STOCHASTIC_Base<TnsSize_>::lastFactor;
      using typename GTC_STOCHASTIC_Base<TnsSize_>::Dimensions;
      using typename GTC_STOCHASTIC_Base<TnsSize_>::MatrixArray;
      using typename GTC_STOCHASTIC_Base<TnsSize_>::DataType;
      using typename GTC_STOCHASTIC_Base<TnsSize_>::SparseTensor;
      using typename GTC_STOCHASTIC_Base<TnsSize_>::IntArray;
      using typename GTC_STOCHASTIC_Base<TnsSize_>::LongMatrix;

      using Options = partensor::SparseOptions<TnsSize_,execution::openmp_policy,SparseDefaultValues>;
      using Status  = partensor::SparseStatus<TnsSize_,execution::openmp_policy,SparseDefaultValues>;

      // Variables that will be used in gtc implementations.
      struct Member_Variables
      {
        MatrixArray  factors_T;
        MatrixArray  factor_T_factor;
        MatrixArray  mttkrp_T;
        IntArray     tnsDims;
        std::array<std::array<int, TnsSize_ - 1>, TnsSize_> offsets;

        MatrixArray  norm_factors_T;
        MatrixArray  old_factors_T;

        Matrix       cwise_factor_product;
        Matrix       Ratings_Base_T;
        SparseTensor tnsX;

        // bool         all_orthogonal = true;
        // int          weight_factor;
        int          rank;
        double       c_stochastic_perc;

        MatrixArray grad;
        MatrixArray Y;
        MatrixArray invL;

        Member_Variables() = default;

        Member_Variables(int R, IntArray dims) :  tnsDims(dims),
                                                  rank(R)
        {}

        Member_Variables(Member_Variables const &) = default;
        Member_Variables(Member_Variables      &&) = default;

        Member_Variables &operator=(Member_Variables const &) = default;
        Member_Variables &operator=(Member_Variables      &&) = default;
      };

      /*
       * In case option variable @c writeToFile is enabled, then, before the end
       * of the algorithm, it writes the resulted factors in files, whose
       * paths are specified before compiling in @ options.final_factors_path.
       *
       * @param  st   [in] Struct where the returned values of @c Gtc are stored.
       */
      void writeFactorsToFile(Status const &st)
      {
        std::size_t size;
        for(std::size_t i=0; i<TnsSize; ++i)
        {
           size = st.factors[i].rows() * st.factors[i].cols();
           partensor::write(st.factors[i],
                            st.options.final_factors_paths[i],
                            size);
         }
       }

       /*
        * Compute the cost function value at the end of each outer iteration
        * based on the last factor.
        *
        * @param  mv   [in]     Struct where ALS variables are stored.
        * @param  st   [in,out] Struct where the returned values of @c Gtc are stored.
        *                       In this case the cost function value is updated.
        */
       void cost_function(Member_Variables const &mv,
                          Status                 &st)
       {
         Matrix temp_R_1(mv.rank, 1);
```

```
          double temp_1_1 = 0;
          double f_value_loc = 0;

          #pragma omp master
          st.f_value = 0;

          #pragma omp barrier

          std::array<int,TnsSize-1> offsets;
          offsets[0] = 1;
          for (int j = 1; j < static_cast<int>(TnsSize) - 1; j++)
          {
            offsets[j] = offsets[j - 1] * mv.tnsDims[j-1];
          }

          #pragma omp for schedule(static)
          for (long int i = 0; i < mv.tnsX[lastFactor].outerSize(); ++i)
          {
              int row = 0;
              for (SparseMatrix::InnerIterator it(mv.tnsX[lastFactor], i); it; ++it)
              {
                  temp_R_1 = mv.factors_T[lastFactor].col(it.col());
                  // Select rows of each factor an compute the Hadamard product of the respective row of the Khatri-Rao product, and the row of factor A_N.
                  for (int mode_i = static_cast<int>(TnsSize) - 2; mode_i >= 0; mode_i--)
                  {
                      row                = ((it.row()) / offsets[mode_i]) % (mv.tnsDims[mode_i]);
                      temp_R_1.noalias() = temp_R_1.cwiseProduct(mv.factors_T[mode_i].col(row));
                  }
                  temp_1_1 = it.value() - temp_R_1.sum();
                  f_value_loc += temp_1_1 * temp_1_1;
              }
          }
          #pragma omp atomic
          st.f_value += f_value_loc;

        }

        /*
         * Compute the cost function value at the end of each outer iteration
         * based on the last accelerated factor.
         *
         * @param  mv                [in] Struct where ALS variables are stored.
         * @param  accel_factors     [in] Accelerated factors.
         *
         * @returns The cost function calculated with the accelerated factors.
         */
        double accel_cost_function(Member_Variables       const &mv,
                                   MatrixArray            const &accel_factors)
        {
          Matrix temp_R_1(mv.rank, 1);
          double temp_1_1 = 0;
          double f_value = 0;

          std::array<int,TnsSize-1> offsets;
          offsets[0] = 1;
          for (int j = 1; j < static_cast<int>(TnsSize) - 1; j++)
          {
            offsets[j] = offsets[j - 1] * mv.tnsDims[j-1];
          }

          #pragma omp for schedule(static)
          for (long int i = 0; i < mv.tnsX[lastFactor].outerSize(); ++i)
          {
            int row = 0;
            for (SparseMatrix::InnerIterator it(mv.tnsX[lastFactor], i); it; ++it)
            {
                temp_R_1 = accel_factors[lastFactor].col(it.col());
                // Select rows of each factor an compute the Hadamard product of the respective row of the Khatri-Rao product, and the row of factor A_N.
                // temp_R_1  = A_N(i_N,:) .* ... .* A_2(i_2,:) .* A_1(i_1,:)
                for (int mode_i = static_cast<int>(TnsSize) - 2; mode_i >= 0; mode_i--)
                {
                    row                = ((it.row()) / offsets[mode_i]) % (mv.tnsDims[mode_i]);
                    temp_R_1.noalias() = temp_R_1.cwiseProduct(accel_factors[mode_i].col(row));
                }
                temp_1_1 = it.value() - temp_R_1.sum();
                f_value += temp_1_1 * temp_1_1;
            }
          }
          return f_value;
        }

        void calculate_offsets(Member_Variables &mv)
        {
          for (int idx = 0; idx < static_cast<int>(TnsSize); idx++)
          {
```

```
            mv.offsets[idx][0] = 1;
            for (int j = 1, mode = 0; j < static_cast<int>(TnsSize) - 1; j++, mode++)
            {
              if (idx == mode)
              {
                mode++;
              }
              mv.offsets[idx][j] = mv.offsets[idx][j - 1] * mv.tnsDims[mode];
            }
          }
        }

        void unconstraint_update(int              const  idx,
                                 Member_Variables       &mv,
                                 Status                 &st)
        {
            int r = mv.rank;

            Matrix eye = st.options.lambdas[idx] * Matrix::Identity(r, r);

            int last_mode = (idx == static_cast<int>(TnsSize) - 1) ? static_cast<int>(TnsSize) - 2 : static_cast<int>(TnsSize) - 1;

            Matrix MTTKRP_col(r, 1);
            Matrix temp_RxR(r, r);
            Matrix temp_R_1(r, 1);

            // Compute MTTKRP
            #pragma omp for schedule(guided) nowait
            for (long int i = 0; i < mv.tnsX[idx].outerSize(); ++i)
            {
                MTTKRP_col.setZero();
                temp_RxR.setZero(); // is the Hadamard product of Grammians of the Factors, that correspond to the nnz elements of the Tensor.
                for (SparseMatrix::InnerIterator it(mv.tnsX[idx], i); it; ++it)
                {
                    temp_R_1 = Matrix::Ones(r, 1);
                    int row;
                    // Select rows of each factor an compute the respective row of the Khatri-Rao product.
                    for (int mode_i = last_mode, kr_counter = static_cast<int>(TnsSize) - 2; mode_i >= 0 && kr_counter >= 0; mode_i--)
                    {
                        if (mode_i == idx)
                        {
                            continue;
                        }
                        row      = ((it.row()) / mv.offsets[idx][kr_counter]) % (mv.tnsDims[mode_i]);
                        temp_R_1 = temp_R_1.cwiseProduct(mv.factors_T[mode_i].col(row));
                        kr_counter--;
                    }
                    // Subtract from the previous row the respective row of W, according to relation (9).
                    MTTKRP_col.noalias() += it.value() * temp_R_1;
                    temp_RxR.noalias()   += temp_R_1 * temp_R_1.transpose();
                }
                mv.factors_T[idx].col(i) = (temp_RxR + eye).inverse() * MTTKRP_col;
            }
        }

        /*
         * Based on each factor's constraint, a different
         * update function is used at every outer iteration.
         *
         * Computes also factor^T * factor at the end.
         *
         * @param  idx [in]     Factor to be updated.
         * @param  mv  [in]     Struct where ALS variables are stored.
         * @param  st  [in,out] Struct where the returned values of @c Gtc are stored.
         *                      Updates the @c stl array with the factors.
         */
        void update_factor(int              const  idx,
                           Member_Variables       &mv,
                           Status                 &st    )
        {
          // Update factor
          switch ( st.options.constraints[idx] )
          {
            case Constraint::unconstrained:
            {
              // unconstraint_update(idx, mv, st);
              break;
            }
            case Constraint::nonnegativity:
            {
              dynamic_blocksize::local_L::StochasticNesterovMNLS(mv.factors_T, mv.tnsDims, mv.tnsX[idx], mv.offsets[idx], mv.c_stochastic_perc, mv.Y[idx],
```

```
                       st.options.max_nesterov_iter, st.options.lambdas[idx], idx);
          break;
        }
        default: // in case of Constraint::constant
          break;
      }

      // Compute A^T * A + B^T * B + ...
      #pragma omp master
      {
        st.factors[idx] = mv.factors_T[idx].transpose();
        mv.factor_T_factor[idx].noalias() = mv.factors_T[idx] * st.factors[idx];
      }
    }

    /*
     * @brief Line Search Acceleration
     *
     * Performs an acceleration step on the updated factors, and keeps the accelerated factors
     * when the step succeeds. Otherwise, the acceleration step is ignored.
     * Line Search Acceleration reduces the number of outer iterations in the ALS algorithm.
     *
     * @note This implementation ONLY, if factors are of @c Matrix type.
     *
     * @param  mv  [in,out] Struct where ALS variables are stored.
     *                      In case the acceleration step is successful the Gramian
     *                      matrices of factors are updated.
     * @param  st  [in,out] Struct where the returned values of @c Gtc are stored.
     *                      If the acceleration succeeds updates @c factors
     *                      and cost function value.
     *
     */
    void line_search_accel(Member_Variables &mv,
                           Status           &st,
                           double           &f_accel,
                           double           &accel_step,
                           MatrixArray      &accel_factors_T,
                           MatrixArray      &accel_gramians)
    {
      #pragma omp master
      {
        for(std::size_t i=0; i<TnsSize; ++i)
        {
            accel_factors_T[i] = mv.old_factors_T[i] + accel_step * (mv.factors_T[i] - mv.old_factors_T[i]);
            accel_gramians[i]  = accel_factors_T[i] * accel_factors_T[i].transpose();
        }

        f_accel = 0;
      }

      #pragma omp barrier

      double f_accel_loc = accel_cost_function(mv, accel_factors_T);

      #pragma omp atomic
      f_accel += f_accel_loc;

      #pragma omp barrier

      #pragma omp master
      {
        if (st.f_value > f_accel)
        {
            mv.factors_T       = accel_factors_T;
            mv.factor_T_factor = accel_gramians;
            st.f_value         = f_accel;
            Partensor()->Logger()->info("Acceleration Step SUCCEEDED! at iter: {}", st.ao_iter);
        }
        else
            st.options.accel_fail++;

        if (st.options.accel_fail==5)
        {
            st.options.accel_fail=0;
            st.options.accel_coeff++;
        }
      }
    }

    /*
     * Sequential implementation of Alternating Least Squares (ALS) method.
     *
     * @param  R   [in]     The rank of decomposition.
     * @param  mv  [in]     Struct where ALS variables are stored and being updated
     *                      until a termination condition is true.
     * @param  st  [in,out] Struct where the returned values of @c Gtc are stored.
```

```
        */
       void aogtc_stochastic(Member_Variables       &mv,
                   Status                   &status)
       {
         double f_accel    = 0.0; // Objective Value after the acceleration step
         double accel_step = 0.0;

         MatrixArray  accel_factors_T;
         MatrixArray  accel_gramians;

         for (std::size_t i=0; i<TnsSize; i++)
         {
           mv.Y[i]              = Matrix::Zero(mv.rank, mv.tnsDims[i]);
           mv.factors_T[i]      = status.factors[i].transpose();
           mv.mttkrp_T[i]       = Matrix(mv.rank, mv.tnsDims[i]);
           mv.factor_T_factor[i].noalias() = mv.factors_T[i] * status.factors[i];
           accel_factors_T[i] = mv.factors_T[i];
           accel_gramians[i]  = Matrix::Zero(mv.rank, mv.rank);
         }

         // if(status.options.normalization)
         // {
         //   choose_normilization_factor(status, mv.all_orthogonal, mv.weight_factor);
         // }
         // Normalize(static_cast<int>(R), mv.factor_T_factor, status.factors);

         std::size_t epoch = static_cast<std::size_t> (1/mv.c_stochastic_perc);
         std::size_t epoch_counter = 0;

         const int total_num_threads = get_num_threads();
         omp_set_nested(0);

         status.frob_tns          = (mv.tnsX[0]).squaredNorm();

         #pragma omp parallel \
                 num_threads(total_num_threads) \
                 proc_bind(spread)\
                 default(shared)
         {
           cost_function(mv, status);
           #pragma omp barrier

           #pragma omp master
           {
             status.rel_costFunction = status.f_value / status.frob_tns;
           }
           #pragma omp barrier

           // ---- Loop until ALS converges ----
           while(1)
           {
               #pragma omp master
               {
                   status.ao_iter++;
                   Partensor()->Logger()->info("iter: {} -- fvalue: {} -- relative_costFunction: {}", status.ao_iter,
                                                                           status.f_value, status.rel_costFunction);
               }

               for (std::size_t i=0; i<TnsSize; i++)
               {
                   // Update factor
                   update_factor(i, mv, status);
               }

               //      <---------------------  End loop for every mode -------------------------->      //
               // Cost function computation.
               if (status.ao_iter % epoch == 0)
               {
                 #pragma omp master
                 {
                   epoch_counter++;
                 }

                 #pragma omp barrier

                 cost_function(mv, status);

                 #pragma omp barrier

                 #pragma omp master
                 {
                   status.rel_costFunction = status.f_value/status.frob_tns;
                 }

               }
```

```
            #pragma omp barrier

            // ---- Terminating condition ----
            if (status.rel_costFunction < status.options.threshold_error || epoch_counter >= status.options.max_iter)
            {
                #pragma omp master
                {
                    if (status.options.writeToFile)
                        writeFactorsToFile(status);
                }
                break;
            }

            // if (status.options.acceleration)
            // {
            //     // ---- Acceleration Step ----
            //     if (status.ao_iter > 1)
            //     {
            //         #pragma omp master
            //         accel_step = pow(status.ao_iter+1,(1.0/(status.options.accel_coeff)));

            //         line_search_accel(mv, status, f_accel, accel_step, accel_factors_T, accel_gramians);
            //         #pragma omp barrier
            //     }
            //     #pragma omp master
            //     {
            //         for (int i = 0; i < static_cast<int>(TnsSize); i++)
            //             mv.old_factors_T[i] = mv.factors_T[i];
            //     }
            // }
        } // end of while
      } // end of pragma
    }

    void initialize_factors(Member_Variables &mv,
                            Status           &status)
    {
      if(status.options.initialized_factors)
      {
        if(status.options.read_factors_from_file)
        {
          for(std::size_t i=0; i<TnsSize; ++i)
          {
            status.factors[i] = Matrix(mv.tnsDims[i], static_cast<int>(mv.rank));
            read( status.options.initial_factors_paths[i],
                  mv.tnsDims[i] * mv.rank,
                  0,
                  status.factors[i] );
          }
        }
        else
          status.factors = status.options.factorsInit;
      }
      else // produce estimate factors using uniform distribution with entries in [0,1].
        makeFactors(mv.tnsDims, status.options.constraints, mv.rank, status.factors);
    }

    Status operator()(Options const &options)
    {
      Status           status(options);
      Member_Variables mv(options.rank, options.tnsDims);

      long long int fileSize = (TnsSize + 1) * options.nonZeros;

      Matrix Ratings_Base_T = Matrix(static_cast<int>(TnsSize+1), options.nonZeros);
      // Read the whole Tensor from a file
      read( options.ratings_path,
            fileSize,
            0,
            Ratings_Base_T );

      // GTC_STOCHASTIC_Base<TnsSize>::sort_ratings_base(mv.Ratings_Base_T, options.nonZeros);
      // ReserveSparseTensor<TnsSize>(mv.tnsX, options.tnsDims, options.nonZeros);
      // FillSparseTensor<TnsSize>(mv.tnsX, options.nonZeros, mv.Ratings_Base_T, options.tnsDims);
      // mv.Ratings_Base_T.resize(0,0);
      GTC_STOCHASTIC_Base<TnsSize>::sort_ratings_base(Ratings_Base_T, mv.tnsX, options.tnsDims, options.nonZeros);
      Ratings_Base_T.resize(0,0);

      for (std::size_t i=0; i<TnsSize; i++)
      {
        mv.tnsX[i].makeCompressed();
      }

```

```
          mv.c_stochastic_perc = options.c_stochastic_perc;

          calculate_offsets(mv);

          initialize_factors(mv, status);

          aogtc_stochastic(mv, status);

          return status;
        }

        Status operator()(Matrix       const &Ratings_Base_T,
                          Options      const &options)
        {
          Status           status(options);
          Member_Variables mv(options.rank, options.tnsDims);

          // ReserveSparseTensor<TnsSize>(mv.tnsX, options.tnsDims, options.nonZeros);
          // FillSparseTensor<TnsSize>(mv.tnsX, options.nonZeros, Ratings_Base_T, options.tnsDims);
          // Ratings_Base_T.resize(0,0);
          GTC_STOCHASTIC_Base<TnsSize>::sort_ratings_base(Ratings_Base_T, mv.tnsX, options.tnsDims, options.nonZeros);
          // Ratings_Base_T.resize(0,0);

          for (std::size_t i=0; i<TnsSize; i++)
          {
            mv.tnsX[i].makeCompressed();
          }

          mv.c_stochastic_perc = options.c_stochastic_perc;

          calculate_offsets(mv);

          // produce estimate factors using uniform distribution with entries in [0,1].
          initialize_factors(mv, status);

          partensor::timer.startChronoHighTimer();
          aogtc_stochastic(mv, status);
          double end_gtc_time_omp = partensor::timer.endChronoHighTimer();
          std::cout « "GtcStochasticOpenMP took " « end_gtc_time_omp « " sec." « std::endl;
          return status;
        }
      };
    }  // namespace internal
  }    // namespace v1
} // end namespace partensor
```

8.35 KhatriRao.hpp File Reference

```
#include "PARTENSOR_basic.hpp"
#include "unsupported/Eigen/KroneckerProduct"
```

Functions

- template<typename ... Matrices>
  Matrix KhatriRao (Matrix const &mat1, Matrix const &mat2, Matrices const &...mats)

8.35.1 Detailed Description

Implementations of the Khatri-Rao Product for two or more matrices of `Matrix` type, using the `kroneckerProduct` function from `Eigen`.

8.35.2 Function Documentation

8.35.2.1 KhatriRao()

```
Matrix partensor::v1::KhatriRao (
            Matrix const & mat1,
            Matrix const & mat2,
            Matrices const &...  mats )
```

Computes the `Khatri-Rao` Product among 2 or more Matrices, with the use of `Eigen::kronecker↩ Product`.

Template Parameters

Matrices	A `variadic` type in case of more than 2 matrices.

Parameters

mat1	[in] A `Matrix`.
mat2	[in] A `Matrix`.
mats	[in] Possible 0 or more Matrices of `Matrix` type.

Returns

An `Eigen` Matrix is returned of type `DT`.

8.36 KhatriRao.hpp

Go to the documentation of this file.

```
#ifndef DOXYGEN_SHOULD_SKIP_THIS
#endif  // DOXYGEN_SHOULD_SKIP_THIS
/***********************************************************************/
#ifndef PARTENSOR_KHATRI_RAO_HPP
#define PARTENSOR_KHATRI_RAO_HPP

#include "PARTENSOR_basic.hpp"
#include "unsupported/Eigen/KroneckerProduct"

#if __has_include("tbb/parallel_for.h")
        #include "boost/range/irange.hpp"
        #include "tbb/parallel_for.h"
#endif /* __has_include("tbb/parallel_for.h") */

// #if USE_TBB
//      #include "boost/range/irange.hpp"
//      #include "tbb/parallel_for.h"
// #endif /* USE_TBB */

namespace partensor
{
        inline namespace v1 {

                #ifndef DOXYGEN_SHOULD_SKIP_THIS

                namespace internal {

                        template <typename ...Matrices>
                        Matrix KhatriRao_seq( Matrix   const &mat1,
                                              Matrix   const &mat2,
                                              Matrices const &...mats )
                        {
                                if constexpr (sizeof... (mats) == 0)
                                {
                                int R   = mat1.cols();
                                int I1  = mat1.rows();
                                int I2  = mat2.rows();

```

```
                        Matrix res(I1*I2,R);

                        for (int i=0; i < R; i++)
                        {
                                res.block(0,i,I1*I2,1) = kroneckerProduct(mat1.block(0,i,I1,1),mat2.block(0,i,I2,1));
                        }

                        return res;
                        }
                        else
                        {
                        auto _temp = KhatriRao_seq(mat2,mats...);

                        return KhatriRao_seq(mat1,_temp);
                        }
                }
        } // end namespace internal

        #endif // DOXYGEN_SHOULD_SKIP_THIS

        #ifndef DOXYGEN_SHOULD_SKIP_THIS
        template <typename ExecutionPolicy, typename ...Matrices>
        execution::internal::enable_if_execution_policy<ExecutionPolicy,Matrix>
        KhatriRao( ExecutionPolicy      &&,
                   Matrix          const &mat1,
                   Matrix          const &mat2,
                   Matrices        const &...mats )
        {
                return internal::KhatriRao_seq(mat1,mat2,mats...);
        }

        #endif // DOXYGEN_SHOULD_SKIP_THIS

        template <typename ...Matrices>
        Matrix KhatriRao( Matrix   const &mat1,
                                   Matrix   const &mat2,
                                   Matrices const &...mats )
        {
                return KhatriRao(execution::seq,mat1,mat2,mats...);
        }

} // namespace v1

#if __has_include("tbb/parallel_for.h")
#ifndef DOXYGEN_SHOULD_SKIP_THIS
namespace experimental {

        inline namespace v1{

                namespace internal {

                        template <typename ...Matrices>
                        Matrix KhatriRao_par( Matrix   const &mat1,
                                                        Matrix   const &mat2,
                                                        Matrices const &...mats )
                        {
                                if constexpr (sizeof... (mats) == 0)
                                {
                                        int R   = mat1.cols();
                                        int I1  = mat1.rows();
                                        int I2  = mat2.rows();

                                        Matrix res(I1*I2,R);
                                        // auto      r = boost::irange(0,R);

                                        tbb::parallel_for(tbb::blocked_range<std::size_t>(0, R),
                                                          [&](const tbb::blocked_range<size_t>& r) {
                                                                for (std::size_t i = r.begin(); i != r.end(); ++i)
                                                                        res.block(0,i,I1*I2,1) = kroneckerProduct(mat1.block(0,i,I1,1),mat2.block(0,i,I2,1));
                                                          } );

                                        return res;
                                }
                                else
                                {
                                        //int R         = mat1.cols();
                                        //int I         = (mats.rows() * ... * mat2.rows());

                                        auto _temp = KhatriRao_par(mat2,mats...);

                                        return KhatriRao_par(mat1,_temp);
                                }
                        }
```

```
        }  // namespace internal

        template <typename ExecutionPolicy, typename ...Matrices>
        execution::internal::enable_if_execution_policy<ExecutionPolicy,Matrix>
        KhatriRao( ExecutionPolicy       &&,
                   Matrix          const &mat1,
                   Matrix          const &mat2,
                   Matrices        const &...mats )
        {
            using  ExPolicy =
    std::remove_cv_t<std::remove_reference_t<ExecutionPolicy>>;

            if constexpr (std::is_same_v<ExPolicy,execution::sequenced_policy>)
            {
            return partensor::v1::internal::KhatriRao_seq(mat1,mat2,mats...);
            }
            else if constexpr (std::is_same_v<ExPolicy,execution::parallel_policy>)
            {
            return internal::KhatriRao_par(mat1,mat2,mats...);
            }
            else
            {
            return partensor::v1::internal::KhatriRao_seq(mat1,mat2,mats...);
            }
        }

    } // end namespace

    namespace v2 {

        namespace internal {

            template <typename ...Matrices>
            int KhatriRao_seq( Matrix          &res,
                               int                  I1,
                                       Matrix   const &mat2,
                                       Matrices const &...mats )
            {
                const int R  = mat2.cols();
                int       I2 = mat2.rows();

                [[maybe_unused]] int I  = I1 * I2 * (mats.rows()*...*1);

                for (int i=0; i < R; i++)
                {
                    res.block(0,i,I1*I2,1) =
    kroneckerProduct(res.block(0,i,I1,1),mat2.block(0,i,I2,1)).eval();
                }

                I1 *= I2;

                if constexpr (sizeof... (mats) == 0)
                {
                    return I1;
                }
                else
                {
                    return KhatriRao_seq(res, I1, mats...);
                }
            }

            template <typename ...Matrices>
            Matrix KhatriRao_seq( Matrix   const &mat1,
                                          Matrix   const &mat2,
                                          Matrices const &...mats )
            {
                int R = mat1.cols();
                int I = mat1.rows() * mat2.rows() * (mats.rows()*...*1);

                Matrix res(I,R);

                res.block(0,0,mat1.rows(),mat1.cols()) = mat1;

                int  I_KR = KhatriRao_seq(res,mat1.rows(),mat2,mats...);

                assert(I == I_KR);

                return res;
            }

            template <typename DT = DefaultDataType, typename ...Matrices>
            int KhatriRao_par( Matrix          &res,
                               int              I1,
                                       Matrix   const &mat2,
                                       Matrices const &...mats )
            {
                const int R  = mat2.cols();
```

```
                int       I2 = mat2.rows();

                // Matrix  res(I1*I2,R);
                // auto      r = boost::irange(0,R);

                tbb::parallel_for(tbb::blocked_range<std::size_t>(0, R),
                        [&](const tbb::blocked_range<size_t>& r) {
                                Eigen::VectorXd temp = Eigen::VectorXd::Zero(I2,1);
                                for (std::size_t j = r.begin(); j != r.end(); j++)
                                {
                                        temp = mat2.col(j);
                                        for (auto i = 0; i < I1; i++)//std::size_t
                                        {
                                        res.block(i*I2, j, I2, 1).noalias() = res(i,j) * temp;
                                        }
                                }
                        } );

                I1 *= I2;

                if constexpr (sizeof... (mats) == 0)
                {
                        return I1;
                }
                else
                {
                        return KhatriRao_par(res, I1, mats...);
                }
        }

        template <typename DT = DefaultDataType, typename ...Matrices>
        Matrix KhatriRao_par( Matrix   const &mat1,
                                                  Matrix   const &mat2,
                                                  Matrices const &...mats )
        {
                int R = mat1.cols();
                int I = mat1.rows() * mat2.rows() * (mats.rows()*...*1);

                Matrix res(I,R);

                res.block(I-mat1.rows(),0,mat1.rows(),mat1.cols()) = mat1;

                int I_KR = KhatriRao_par(res,mat1.rows(),mat2,mats...);

                assert(I == I_KR);

                return res;
        }
}   // namespace internal

template <typename ExecutionPolicy, typename ...Matrices>
execution::internal::enable_if_execution_policy<ExecutionPolicy,Matrix>
KhatriRao( ExecutionPolicy             &&,
           Matrix          const &mat1,
                Matrix          const &mat2,
                Matrices        const &...mats )
{
  using  ExPolicy = std::remove_cv_t<std::remove_reference_t<ExecutionPolicy»;

  if constexpr (std::is_same_v<ExPolicy,execution::sequenced_policy>)
  {
    return internal::KhatriRao_seq(mat1,mat2,mats...);
  }
  else if constexpr (std::is_same_v<ExPolicy,execution::parallel_policy>)
  {
    return internal::KhatriRao_par(mat1,mat2,mats...);
  }
  else
    return internal::KhatriRao_seq(mat1,mat2,mats...);
}

template <typename ...Matrices>
Matrix KhatriRao( Matrix const &mat1,
                                                  Matrix const &mat2,
                                                  Matrices       const &...mats )
{
  return KhatriRao(execution::seq,mat1,mat2,mats...);
}
} // end namespace v2

namespace v3 {
```

```
        namespace internal {

            template <typename DT = DefaultDataType, typename ...Matrices>
            int KhatriRao_seq( Matrix           &res,
                                                 int              I1,
                                                 Matrix   const &mat2,
                                                 Matrices const &...mats )
            {
                const int R  = mat2.cols();
                int       I2 = mat2.rows();
                int       I  = I1 * I2 * (mats.rows()*...*1);

                for (int i=0; i < R; i++)
                {
                    Eigen::Map<Matrix, 0, Eigen::InnerStride<»(res.data()+i*I, I1*I2, 1, Eigen::InnerStride<>( res.innerStride()*(I/(I1*I2)))) =
                    kroneckerProduct( Eigen::Map<Matrix, 0, Eigen::InnerStride<»(res.data()+i*I, I1, 1, Eigen::InnerStride<>(res.innerStride()*(I/I1))), mat2.block(0,i,I2,1));
                }

                I1 *= I2;

                if constexpr (sizeof... (mats) == 0)
                {
                    return I1;
                }
                else
                {
                    return KhatriRao_seq(res, I1, mats...);
                }
            }

            template <typename ...Matrices>
            Matrix KhatriRao_seq( Matrix   const &mat1,
                                         Matrix   const &mat2,
                                         Matrices const &...mats)
            {
                int R = mat1.cols();
                int I = mat1.rows() * mat2.rows() * (mats.rows()*...*1);

                Matrix res(I,R);

                Eigen::Map<Matrix, 0, Eigen::InnerStride<» (res.data(), mat1.rows(), R, Eigen::InnerStride<>(res.innerStride()*(I/mat1.rows()))) = mat1;

                int  I_KR = KhatriRao_seq(res,mat1.rows(),mat2,mats...);

                assert(I == I_KR);

                return res;
            }

            template <typename ...Matrices>
            int KhatriRao_par( Matrix           &res,
                               int                I1,
                                                 Matrix   const &mat2,
                                                 Matrices const &...mats )
            {
                const int R  = mat2.cols();
                int       I2 = mat2.rows();

                // Matrix  res(I1*I2,R);
                // auto            r = boost::irange(0,R);

                tbb::parallel_for(tbb::blocked_range<std::size_t>(0, R),
                        [&](const tbb::blocked_range<size_t>& r) {
                            Eigen::VectorXd temp = Eigen::VectorXd::Zero(I2,1);
                            for (std::size_t j = r.begin(); j != r.end(); j++)
                            {
                                temp = mat2.col(j);
                                for (auto i = 0; i < I1; i++)
                                {
                                    res.block(i*I2, j, I2, 1).noalias() =  res(i,j) * temp;
                                }
                            }
                        } );

                I1 *= I2;

                if constexpr (sizeof... (mats) == 0)
                {
                    return I1;
```

```
                                        }
                                        else
                                        {
                                                return KhatriRao_par(res, I1, mats...);
                                        }
                                }

                                template <typename ...Matrices>
                                Matrix KhatriRao_par( Matrix   const &mat1,
                                                                          Matrix   const &mat2,
                                                                          Matrices const &...mats )
                                {
                                        int R = mat1.cols();
                                        int I = mat1.rows() * mat2.rows() * (mats.rows()*...*1);

                                        Matrix res(I,R);

                                        res.block(I-mat1.rows(),0,mat1.rows(),mat1.cols()) = mat1;

                                        int I_KR = KhatriRao_par(res,mat1.rows(),mat2,mats...);

                                        assert(I == I_KR);

                                        return res;
                                }
                }   // namespace internal

                        template <typename ExecutionPolicy, typename ...Matrices>
                        execution::internal::enable_if_execution_policy<ExecutionPolicy,Matrix>
                        KhatriRao( ExecutionPolicy              &&,
                                           Matrix         const &mat1,
                                           Matrix         const &mat2,
                                           Matrices       const &...mats)
                        {
                                using  ExPolicy = std::remove_cv_t<std::remove_reference_t<ExecutionPolicy»;

                                if constexpr (std::is_same_v<ExPolicy,execution::sequenced_policy>)
                                {
                                        return internal::KhatriRao_seq(mat1,mat2,mats...);
                                }
                                else if constexpr (std::is_same_v<ExPolicy,execution::parallel_policy>)
                                {
                                        return internal::KhatriRao_par(mat1,mat2,mats...);
                                }
                                else
                                return internal::KhatriRao_seq(mat1,mat2,mats...);
                        }

                        template <typename ...Matrices>
                        Matrix KhatriRao( Matrix   const &mat1,
                                                          Matrix   const &mat2,
                                                          Matrices const &...mats )
                        {
                                return KhatriRao(execution::seq,mat1,mat2,mats...);
                        }
                } // end namespace v3

                // namespace v4 {
                //
                //      namespace internal {

              // template <typename DT = DefaultDataType, typename ...Matrices>
              // int KhatriRao_seq(Matrix &res, int I1, Matrix const &mat1, Matrix const &mat2, Matrices const &...mats)
              // {
              //    // const int R = mat2.cols();
                                //      //
              //    // int I2    = mat2.rows();
                                //      //
              //    // int I     = I1 * I2 * (mats.rows()*...*1);
                                //      //
              //    // for (int i=0; i < R; i++)
              //    // {
              //    //   res.block(0,i,I1*I2,1) = kroneckerProduct(res.block(0,i,I1,1),mat2.block(0,i,I2,1)).eval();
              //    // }
                                //      //
              //    // I1 *= I2;
                                //
              //    if constexpr (sizeof... (mats) == 0)
              //    {
                                //              for (int i=0; i < R; i++)
                                //              {
                                //                      res.block(0,i,I1*I2,1) = kroneckerProduct(mat1.block(0,i,I1,1),mat2.block(0,i,I2,1)).eval();
                                //              }
```

```
                        //
            //     return I1;
            //   }
            //   else
            //   {
            //    return KhatriRao_seq(res, I1, mats...);
                        //
            //   for (int i=0; i < R; i++)
            //   {
            //     Eigen::Map<MatrixType<dataType>, 0, Eigen::InnerStride<> >  (res.data()+i*I, I1*I2, 1, Eigen::InnerStride<>( res.innerStride()*(I/(I1*I2)))) =
            //     kroneckerProduct( Eigen::Map<MatrixType<dataType>, 0, Eigen::InnerStride<> > (res.data()+i*I, I1, 1, Eigen::InnerStride<>( res.innerStride()*(I/I1))), mat2.block(0,i,I2,1));
            //   }
                        //
            //   }
            // }
                        //
            // template <typename DT = DefaultDataType, typename ...Matrices>
                        // Matrix KhatriRao_seq(Matrix const &mat1, Matrix const &mat2, Matrices const &...mats)
                        // {
            //   int    R = mat1.cols();
            //   int    I = mat1.rows() * mat2.rows() * (mats.rows()*...*1);
            //   Matrix res(I,R);
                        //
                        //   res.block(0,0,mat1.rows(),mat1.cols()) = mat1;
                        //
            //   int  I_KR = KhatriRao_seq(res,mat1,mat2,mats...);
                        //
            //   assert(I == I_KR);
                        //
            //   return res;
                        // }

         //     template <typename DT = DefaultDataType, typename ...Matrices>
         //     int KhatriRao_par(Matrix &res, int I1, Matrix const &mat2, Matrices const &...mats)
             //             {
         //       const int R = mat2.cols();
             //
         //       int I2       = mat2.rows();
             //
         //       // Matrix  res(I1*I2,R);
         //       // auto      r = boost::irange(0,R);
             //
         //       tbb::parallel_for(tbb::blocked_range<std::size_t>(0, R),
         //                          [&](const tbb::blocked_range<size_t>& r) {
         //                             Eigen::VectorXd temp = Eigen::VectorXd::Zero(I2,1);
         //                             for (std::size_t j = r.begin(); j != r.end(); j++)
         //                             {
         //                               temp = mat2.col(j);
         //                               for (std::size_t i = 0; i < I1; i++)
         //                               {
         //                                 res.block(i*I2, j, I2, 1).noalias() =  res(i,j) * temp;
         //                               }
         //                             }
         //                          } );
             //
         //       I1 *= I2;
             //
         //       if constexpr (sizeof... (mats) == 0)
             //              {
         //        return I1;
             //              }
             //              else
             //              {
         //        return KhatriRao_par(res, I1, mats...);
             //              }
             //             }
             //
         //     template <typename DT = DefaultDataType, typename ...Matrices>
         //     Matrix KhatriRao_par(Matrix const &mat1, Matrix const &mat2, Matrices const &...mats)
         //     {
         //       int    R = mat1.cols();
         //       int    I = mat1.rows() * mat2.rows() * (mats.rows()*...*1);
         //       Matrix res(I,R);
             //
         //       res.block(I-mat1.rows(),0,mat1.rows(),mat1.cols()) = mat1;
             //
         //       int  I_KR = KhatriRao_par(res,mat1.rows(),mat2,mats...);
             //
         //       assert(I == I_KR);
             //
         //       return res;
         //     }
         //   }   // namespace internal
             //
```

```
            //      template <typename ExecutionPolicy, typename DT = DefaultDataType, typename
    ...Matrices>
            //      execution::internal::enable_if_execution_policy<ExecutionPolicy,Matrix>
            //      KhatriRao(ExecutionPolicy &&, Matrix const &mat1, Matrix const &mat2, Matrices
    const &...mats)
            //      {
            //        using  ExPolicy = std::remove_cv_t<std::remove_reference_t<ExecutionPolicy»;
            //
            //        if constexpr (std::is_same_v<ExPolicy,execution::sequenced_policy>)
            //        {
            //          return internal::KhatriRao_seq(mat1,mat2,mats...);
            //        }
            //        else if constexpr (std::is_same_v<ExPolicy,execution::parallel_policy>)
            //        {
            //          return internal::KhatriRao_par(mat1,mat2,mats...);
            //        }
            //        else
            //          return internal::KhatriRao_seq(mat1,mat2,mats...);
            //      }
            //
            //      template <typename DT = DefaultDataType, typename ...Matrices>
            //      Matrix KhatriRao(Matrix const &mat1, Matrix const &mat2, Matrices const
    &...mats)
            //      {
            //        return KhatriRao(execution::seq,mat1,mat2,mats...);
            //      }
            // } // end namespace v4

        }  // end namespace experimental

        #endif // end of DOXYGEN_SHOULD_SKIP_THIS
        #endif // end of #if __has_include("tbb/parallel_for.h")
} // end namespace partensor

#endif // end of PARTENSOR_KHATRI_RAO_HPP
```

8.37 Kronecker.hpp File Reference

```
#include "PARTENSOR_basic.hpp"
#include "unsupported/Eigen/KroneckerProduct"
```

Functions

- template<typename ... Matrices>
  Matrix Kronecker (Matrix const &mat1, Matrix const &mat2, Matrices const &...mats)

8.37.1 Detailed Description

Implementations of the Kronecker Product for two or more matrices of `Matrix` type, using the `kronecker↩ Product` function from `Eigen`.

See also

Kronecker Product

8.37.2 Function Documentation

8.37.2.1 Kronecker()

```
Matrix partensor::v1::Kronecker (
            Matrix const & mat1,
            Matrix const & mat2,
            Matrices const &... mats )
```

Computes the `Kronecker` Product among 2 or more Matrices, with the use of `Eigen::kroneckerProduct`.

Template Parameters

Matrices	A `variadic` type in case of more than 2 matrices.

Parameters

mat1	[in] A `Matrix`.
mat2	[in] A `Matrix`.
mats	[in] Possible 0 or more Matrices of `Matrix` type.

Returns

The result of the `Kronecker` product, stored in a `Matrix` variable.

8.38 Kronecker.hpp

Go to the documentation of this file.

```
#ifndef DOXYGEN_SHOULD_SKIP_THIS
#endif  // DOXYGEN_SHOULD_SKIP_THIS
/*********************************************************************/
#ifndef PARTENSOR_KRONECKER_HPP
#define PARTENSOR_KRONECKER_HPP

#include "PARTENSOR_basic.hpp"
#include "unsupported/Eigen/KroneckerProduct"

namespace partensor {

  inline namespace v1 {

    template <typename ...Matrices>
    Matrix Kronecker( Matrix   const &mat1,
                      Matrix   const &mat2,
                      Matrices const &...mats )
    {
      if constexpr (sizeof... (mats) == 0)
      {
        int I1  = mat1.rows();
        int I2  = mat1.cols();
        int I3  = mat2.rows();
        int I4  = mat2.cols();

        Matrix  res(I1*I3,I2*I4);
        res = Eigen::kroneckerProduct(mat1,mat2);

        return res;
      }
      else
      {
        auto _temp = Kronecker(mat2,mats...);

        return Kronecker(mat1,_temp);
      }
    }

  } // namespace v1

}// end namespace partensor

#endif // end of PARTENSOR_KRONECKER_HPP
```

8.39 Matricization.hpp File Reference

```
#include "PARTENSOR_basic.hpp"
#include "TensorOperations.hpp"
```

Functions

- template<int _TnsSize>
  Matrix Matricization (Tensor< _TnsSize > const &tnsX, std::size_t const mode)
  Implementation of matricization operation over a Tensor.

8.39.1 Detailed Description

Implements the Tensor Matricization operation.

Warning

The Tensor Order must be in [3,8].

Possible examples with Matrices of `Matrix` type from `Tensor.hpp`, are the following.

- If Tensor order is 3, with Tensor `tnsX(IxJxK)`,

  Returns

  A `Matrix`, but the size depends on the matricization `mode`. If it is on the first mode then the Matrix size is (IxJK), if it happened on the second, its size is (JxIK), and if the matricization happened on the 3rd mode then it has size (KxIJ).

- If Tensor order is 4, with Tensor `tnsX(IxJxKxL)`,

  Returns

  A `Matrix`, but the size depends on the matricization `mode`. If it is on the first mode then the Matrix size is (IxJKL), if it happened on the second, its size is (JxIKL), if the matricization happened on the 3rd mode then it has size (KxIJL), and in case of the 4th mode it has size (LxIJK).

8.39.2 Function Documentation

8.39.2.1 Matricization()

```
Matrix partensor::v1::Matricization (
            Tensor< _TnsSize > const & tnsX,
            std::size_t const mode )
```

Implementation of matricization operation over a Tensor.

Takes as input a Tensor with order equal to `_TnsSize`, and performs a matricization, more specifically a shuffling of the data, based on a `mode-dimension`.

Template Parameters

_TnsSize	Tensor Order of `tnsX`.

Parameters

tnsX	[in] The Tensor to be matricized.
mode	[in] The dimension where the matricization will be performed. If `mode=0`, then the matricization will be performed in rows dimensions.

Returns

A `Matrix` with the `tnsX` data permuted based on `mode`.

Warning

`mode` variable takes values from range [0,_TnsSize-1].

The result column dimension of the matricized tensor must have value up until `LONG_MAX`.

8.40 Matricization.hpp

Go to the documentation of this file.

```
#ifndef DOXYGEN_SHOULD_SKIP_THIS
#endif  // DOXYGEN_SHOULD_SKIP_THIS
/**********************************************************************/
#ifndef PARTENSOR_MATRICIZATION_HPP
#define PARTENSOR_MATRICIZATION_HPP

#include "PARTENSOR_basic.hpp"
#include "TensorOperations.hpp"

namespace partensor {

        #ifndef DOXYGEN_SHOULD_SKIP_THIS
        template<std::size_t _TnsSize>
        Eigen::array<int,_TnsSize> permute(std::size_t mode)
        {
                Eigen::array<int, _TnsSize> permutation;
                std::iota(permutation.begin(), permutation.end(), 0); // {0, 1, 2, ..., N}
            permutation[0] = mode;                                       // {n, 0, 1, ..., n-1, n+1, ... , N}
            for(std::size_t i=1; i <= mode; i++) {
                permutation[i] = i - 1;
            }
            return permutation;
        }
        #endif // DOXYGEN_SHOULD_SKIP_THIS

        inline namespace v1 {

                template<int _TnsSize>
                Matrix Matricization( Tensor<_TnsSize> const &tnsX,
                                                    std::size_t      const  mode )
                {
                        using Dimensions                  = typename Tensor<_TnsSize>::Dimensions; // Type of @c Eigen Tensor Dimensions.
                        const Dimensions& tnsDims = tnsX.dimensions();                       // Eigen Array with the lengths of each of Tensor Dimension.

                        Tensor<2> matricedTns; // Temporary @c Eigen Tensor in order to keep the matricized Tensor.
                        auto      permutation = partensor::permute<_TnsSize>(mode);
                        // Compute the column dimension for the matricized tensor.
                        long int newColDim = 1;
                        for(int i=0; i<_TnsSize; ++i)
                        {
                                if(i!=static_cast<int>(mode))
                                        newColDim *= tnsDims[i];
                        }

                        // reshape: View of the input tensor that has been reshaped to the specified new dimensions.
                        // shuffle: A copy of the input tensor whose dimensions have been reordered according to the specified permutation.
```

```
            Eigen::array<long int, 2> reshape({static_cast<long int>(tnsDims[mode]), newColDim});
            matricedTns = tnsX.shuffle(permutation).reshape(reshape);
            // Map the @c Eigen Tensor to @c Eigen Matrix type.
            return tensorToMatrix(matricedTns, tnsDims[mode], newColDim);
        }

    }  // end namespace v1

    #ifndef DOXYGEN_SHOULD_SKIP_THIS
    namespace experimental {

        inline namespace v1 {

            template<int _TnsSize>
            Matrix Matricization( Tensor<_TnsSize> const &tnsX,
                                  std::size_t      const  mode )
            {
                using Dimensions          = typename Tensor<_TnsSize>::Dimensions; // Type of @c Eigen Tensor Dimensions.
                const Dimensions& tnsDims = tnsX.dimensions();                      // Eigen Array with the lengths of each of Tensor Dimension.

                Tensor<2> matricedTns; // Temporary @c Eigen Tensor in order to keep the matricized Tensor.
                auto      permutation = partensor::permute<_TnsSize>(mode);

                // Compute the column dimension for the matricized tensor.
                long int newColDim = 1;
                for(int i=0; i<_TnsSize; ++i)
                {
                    if(i!=static_cast<int>(mode))
                        newColDim *= tnsDims[i];
                }

                if constexpr (_TnsSize<4)
                {
                    // reshape: View of the input tensor that has been reshaped to the specified new dimensions.
                    // shuffle: A copy of the input tensor whose dimensions have been reordered according to the specified permutation.
                    Eigen::array<long int, 2> newshape({tnsDims[mode], newColDim});
                    matricedTns = tnsX.shuffle(permutation).reshape(newshape);

                // Map the @c Eigen Tensor to @c Eigen Matrix type.
                    return tensorToMatrix(matricedTns, tnsDims[mode], newColDim);
                }
                else
                {
                    long int matricized_dim = 1;
                    for(std::size_t i=1; i < permutation.size()-1; ++i)
                    {
                        matricized_dim *= tnsDims[permutation[i]];
                    }
                    Tensor<_TnsSize-1> cubeTns;
                    Matrix             matricedTns(tnsDims[mode], newColDim);
                    // Recursive Call for each cube, based on the mode if it is on the last dimension or not.
                    if(mode<_TnsSize-1)
                    {
                        for(int cubeId=0; cubeId<tnsDims[_TnsSize-1]; cubeId++)
                        {
                            // Get a _TnsSize-1 order hypercube from whole tensor.
                            cubeTns = tnsX.chip(cubeId,_TnsSize-1);
                            // Save the matriced 3D Tensor to matricedTns, after returning from the recursive calls.
                            matricedTns.block(0, cubeId * matricized_dim, tnsDims[mode], matricized_dim) = Matricization(cubeTns, mode);
                        }
                    }
                    else
                    {
                        for(int cubeId=0; cubeId<tnsDims[_TnsSize-2]; cubeId++)
                        {
                            // Get a _TnsSize-1 order hypercube from whole tensor.
                            cubeTns = tnsX.chip(cubeId,_TnsSize-2);
                            // Save the matriced 3D Tensor to matricedTns, after returning from the recursive calls.
                            matricedTns.block(0, cubeId * matricized_dim, tnsDims[mode], matricized_dim) = Matricization(cubeTns, _TnsSize-2);
                        }
                    }
                    return matricedTns;
                }

```

```
        }

            template<int _TnsSize>
            Matrix Matricization_3( Tensor<_TnsSize> const &tnsX,
                                    std::size_t      const   mode )
            {
                using Dimensions          = typename Tensor<_TnsSize>::Dimensions;
                const Dimensions& tnsDims = tnsX.dimensions();
                Tensor<2> matricedTns; // Temporary @c Eigen Tensor in order to keep the matricized Tensor.
                auto      permutation = partensor::permute<_TnsSize>(mode);

                long int newColDim = 1;
                for(int i=0; i<_TnsSize; ++i)
                {
                    if(i!=static_cast<int>(mode))
                        newColDim *= tnsDims[i];
                }
                Eigen::array<long int, 2> newshape({tnsDims[mode], newColDim});
                matricedTns = tnsX.shuffle(permutation).reshape(newshape);

                // Map the @c Eigen Tensor to @c Eigen Matrix type.
                return tensorToMatrix(matricedTns, tnsDims[mode], newColDim);
            }

            template<int _TnsSize>
            Matrix Matricization_4( Tensor<_TnsSize> const &tnsX,
                                    std::size_t      const   mode )
            {
                using Dimensions                = typename Tensor<_TnsSize>::Dimensions;
                const Dimensions& tnsDims = tnsX.dimensions();

                Tensor<_TnsSize-1> cubeTns; // Temporary @c Eigen Tensor in order to keep the matricized Tensor.
                auto               permutation = partensor::permute<_TnsSize>(mode);

                long int newColDim = 1;
                for(int i=0; i<_TnsSize; ++i)
                {
                    if(i!=static_cast<int>(mode))
                        newColDim *= tnsDims[i];
                }
                long int matricized_dim = 1;
                for(std::size_t i=1; i < permutation.size()-1; ++i)
                {
                    matricized_dim *= tnsDims[permutation[i]];
                }
                Matrix matricedTns(tnsDims[mode], newColDim);
                // Recursive Call for each cube, based on the mode if it is on the last dimension or not.
                if(mode<_TnsSize-1)
                {
                    for(int cubeId=0; cubeId<tnsDims[_TnsSize-1]; cubeId++)
                    {
                        // Get a _TnsSize-1 order hypercube from whole tensor.
                        cubeTns = tnsX.chip(cubeId,_TnsSize-1);
                        // Save the matriced 3D Tensor to matricedTns, after returning from the recursive calls.
                        matricedTns.block(0, cubeId * matricized_dim, tnsDims[mode], matricized_dim) = Matricization_3(cubeTns, mode);
                    }
                }
                else
                {
                    for(int cubeId=0; cubeId<tnsDims[_TnsSize-2]; cubeId++)
                    {
                        // Get a _TnsSize-1 order hypercube from whole tensor.
                        cubeTns = tnsX.chip(cubeId,_TnsSize-2);
                        // Save the matriced 3D Tensor to matricedTns, after returning from the recursive calls.
                        matricedTns.block(0, cubeId * matricized_dim, tnsDims[mode], matricized_dim) = Matricization_3(cubeTns, _TnsSize-2);
                    }
                }
                return matricedTns;
        }

            template<int _TnsSize>
            Matrix Matricization_5( Tensor<_TnsSize> const &tnsX,
                                    std::size_t      const   mode )
            {
                using Dimensions          = typename Tensor<_TnsSize>::Dimensions;
                const Dimensions& tnsDims = tnsX.dimensions();

                Tensor<_TnsSize-1> cubeTns;
                auto               permutation = partensor::permute<_TnsSize>(mode);
```

```

                long int newColDim = std::accumulate(tnsDims.begin(), tnsDims.end(), 1, std::multiplies<long int>());
                newColDim            /= tnsDims[mode];

                long int matricized_dim = 1;
                for(std::size_t i=1; i < permutation.size()-1; ++i)
                {
                    matricized_dim *= tnsDims[permutation[i]];
                }
                Matrix matricedTns(tnsDims[mode], newColDim);
                // Recursive Call for each cube, based on the mode if it is on the last dimension or not.
                if(mode<_TnsSize-1)
                {
                    for(int cubeId=0; cubeId<tnsDims[_TnsSize-1]; cubeId++)
                    {
                        // Get a _TnsSize-1 order hypercube from whole tensor.
                        cubeTns = tnsX.chip(cubeId,_TnsSize-1);
                        // Save the matriced 3D Tensor to matricedTns, after returning from the recursive calls.
                        matricedTns.block(0, cubeId * matricized_dim, tnsDims[mode], matricized_dim) = Matricization_4(cubeTns, mode);
                    }
                }
                else
                {
                    for(int cubeId=0; cubeId<tnsDims[_TnsSize-2]; cubeId++)
                    {
                        // Get a _TnsSize-1 order hypercube from whole tensor.
                        cubeTns = tnsX.chip(cubeId,_TnsSize-2);
                        // Save the matriced 3D Tensor to matricedTns, after returning from the recursive calls.
                        matricedTns.block(0, cubeId * matricized_dim, tnsDims[mode], matricized_dim) = Matricization_4(cubeTns, _TnsSize-2);
                    }
                }
                return matricedTns;
        }

            template<int _TnsSize>
            Matrix Matricization_6( Tensor<_TnsSize> const &tnsX,
                                    std::size_t   const  mode )
            {
                using Dimensions        = typename Tensor<_TnsSize>::Dimensions;
                const Dimensions& tnsDims = tnsX.dimensions();

                Tensor<_TnsSize-1> cubeTns;
                auto          permutation = partensor::permute<_TnsSize>(mode);

                long int newColDim = std::accumulate(tnsDims.begin(), tnsDims.end(), 1, std::multiplies<long int>());
                newColDim       /= tnsDims[mode];

                long int matricized_dim = 1;
                for(std::size_t i=1; i < permutation.size()-1; ++i)
                {
                    matricized_dim *= tnsDims[permutation[i]];
                }
                Matrix matricedTns(tnsDims[mode], newColDim);
                // Recursive Call for each cube, based on the mode if it is on the last dimension or not.
                if(mode<_TnsSize-1)
                {
                    for(int cubeId=0; cubeId<tnsDims[_TnsSize-1]; cubeId++)
                    {
                        // Get a _TnsSize-1 order hypercube from whole tensor.
                        cubeTns = tnsX.chip(cubeId,_TnsSize-1);
                        // Save the matriced 3D Tensor to matricedTns, after returning from the recursive calls.
                        matricedTns.block(0, cubeId * matricized_dim, tnsDims[mode], matricized_dim) = Matricization_5(cubeTns, mode);
                    }
                }
                else
                {
                    for(int cubeId=0; cubeId<tnsDims[_TnsSize-2]; cubeId++)
                    {
                        // Get a _TnsSize-1 order hypercube from whole tensor.
                        cubeTns = tnsX.chip(cubeId,_TnsSize-2);
                        // Save the matriced 3D Tensor to matricedTns, after returning from the recursive calls.
                        matricedTns.block(0, cubeId * matricized_dim, tnsDims[mode], matricized_dim) = Matricization_5(cubeTns, _TnsSize-2);
                    }
                }
                return matricedTns;
```

```
        }

            template<int _TnsSize>
            Matrix Matricization_7( Tensor<_TnsSize> const &tnsX,
                                                           std::size_t      const   mode )
            {
                using Dimensions           = typename Tensor<_TnsSize>::Dimensions;
                const Dimensions& tnsDims = tnsX.dimensions();

                Tensor<_TnsSize-1> cubeTns;
                auto                 permutation = partensor::permute<_TnsSize>(mode);

                long int newColDim = std::accumulate(tnsDims.begin(), tnsDims.end(), 1, std::multiplies<long int>());
                newColDim                 /= tnsDims[mode];

                long int matricized_dim = 1;
                for(std::size_t i=1; i < permutation.size()-1; ++i)
                {
                    matricized_dim *= tnsDims[permutation[i]];
                }
                Matrix matricedTns(tnsDims[mode], newColDim);
                // Recursive Call for each cube, based on the mode if it is on the last dimension or not.
                if(mode<_TnsSize-1)
                {
                    for(int cubeId=0; cubeId<tnsDims[_TnsSize-1]; cubeId++)
                    {
                        // Get a _TnsSize-1 order hypercube from whole tensor.
                        cubeTns = tnsX.chip(cubeId,_TnsSize-1);
                        // Save the matriced 3D Tensor to matricedTns, after returning from the recursive calls.
                        matricedTns.block(0, cubeId * matricized_dim, tnsDims[mode], matricized_dim) = Matricization_6(cubeTns, mode);
                    }
                }
                else
                {
                    for(int cubeId=0; cubeId<tnsDims[_TnsSize-2]; cubeId++)
                    {
                        // Get a _TnsSize-1 order hypercube from whole tensor.
                        cubeTns = tnsX.chip(cubeId,_TnsSize-2);
                        // Save the matriced 3D Tensor to matricedTns, after returning from the recursive calls.
                        matricedTns.block(0, cubeId * matricized_dim, tnsDims[mode], matricized_dim) = Matricization_6(cubeTns, _TnsSize-2);
                    }
                }
                return matricedTns;
        }

            template<int _TnsSize>
            Matrix Matricization_8( Tensor<_TnsSize> const &tnsX,
                                                           std::size_t      const   mode )
            {
                using Dimensions           = typename Tensor<_TnsSize>::Dimensions;
                const Dimensions& tnsDims = tnsX.dimensions();

                Tensor<_TnsSize-1> cubeTns;
                auto                 permutation = partensor::permute<_TnsSize>(mode);

                long int newColDim = std::accumulate(tnsDims.begin(), tnsDims.end(), 1, std::multiplies<long int>());
                newColDim                 /= tnsDims[mode];

                long int matricized_dim = 1;
                for(std::size_t i=1; i < permutation.size()-1; ++i)
                {
                    matricized_dim *= tnsDims[permutation[i]];
                }
                Matrix matricedTns(tnsDims[mode], newColDim);
                // Recursive Call for each cube, based on the mode if it is on the last dimension or not.
                if(mode<_TnsSize-1)
                {
                    for(int cubeId=0; cubeId<tnsDims[_TnsSize-1]; cubeId++)
                    {
                        // Get a _TnsSize-1 order hypercube from whole tensor.
                        cubeTns = tnsX.chip(cubeId,_TnsSize-1);
                        // Save the matriced 3D Tensor to matricedTns, after returning from the recursive calls.
                        matricedTns.block(0, cubeId * matricized_dim, tnsDims[mode], matricized_dim) = Matricization_7(cubeTns, mode);
                    }
                }
                else
                {
```

```
                    for(int cubeId=0; cubeId<tnsDims[_TnsSize-2]; cubeId++)
                    {
                        // Get a _TnsSize-1 order hypercube from whole tensor.
                        cubeTns = tnsX.chip(cubeId,_TnsSize-2);
                        // Save the matriced 3D Tensor to matricedTns, after returning from the recursive calls.
                        matricedTns.block(0, cubeId * matricized_dim, tnsDims[mode], matricized_dim) = Matricization_7(cubeTns, _TnsSize-2);
                    }
                }
                return matricedTns;
        }

        } // end namespace v1

        namespace v2 {

            template<typename Tensor_>
            typename TensorTraits<Tensor_>::MatrixType Matricization( Tensor_    const &tnsX,
                                                                     std::size_t const  mode )
            {
                // using DataType    = typename TensorTraits<Tensor_>::DataType;    // Type of @c Eigen Tensor Data (e.g double, float, etc.).
                constexpr std::size_t TnsSize = TensorTraits<Tensor_>::TnsSize;
                using Dimensions              = typename TensorTraits<Tensor_>::Dimensions;  // Type of @c Eigen Tensor Dimensions.

                const Dimensions& tnsDims = tnsX.dimensions(); // Eigen Array with the lengths of each of Tensor Dimension.
                Tensor<2>         matricedTns;                 // Temporary @c Eigen Tensor in order to keep the matricized Tensor.
                auto              permutation = partensor::permute<TnsSize>(mode);

                // Compute the column dimension for the matricized tensor.
                long int newColDim = 1;
                for(std::size_t i=0; i<TnsSize; ++i)
                {
                    if(i!=mode)
                        newColDim *= tnsDims[i];
                }

                // reshape: View of the input tensor that has been reshaped to the specified new dimensions.
                // shuffle: A copy of the input tensor whose dimensions have been reordered according to the specified permutation.
                Eigen::array<long int, 2> reshape({tnsDims[mode], newColDim});
                matricedTns = tnsX.shuffle(permutation).reshape(reshape);
                // Map the @c Eigen Tensor to @c Eigen Matrix type.
                return tensorToMatrix(matricedTns, tnsDims[mode], newColDim);
            }

            template<typename Tensor_>
            typename TensorTraits<Tensor_>::MatrixType Matricization_rec( Tensor_    const &tnsX,
                                                                         std::size_t const  mode )
            {
                // using DataType    = typename TensorTraits<Tensor_>::DataType;    // Type of @c Eigen Tensor Data (e.g double, float, etc.).
            using MatrixType  = typename TensorTraits<Tensor_>::MatrixType;    // Type of @c Eigen Matrix based on Tensor Type.
                using Dimensions  = typename TensorTraits<Tensor_>::Dimensions;        // Type of @c Eigen Tensor Dimensions.

                static constexpr std::size_t TnsSize = TensorTraits<Tensor_>::TnsSize;    // Tensor Order.

                const Dimensions&  tnsDims     = tnsX.dimensions(); // Eigen Array with the lengths of each of Tensor Dimension.
                Tensor<2>          matricedTns;                     // Temporary @c Eigen Tensor in order to keep the matricized Tensor.
                auto               permutation = partensor::permute<TnsSize>(mode);

                // Compute the column dimension for the matricized tensor.
                long int newColDim = std::accumulate(tnsDims.begin(), tnsDims.end(), 1, std::multiplies<long int>());
                newColDim                 /= tnsDims[mode];

                if constexpr (TnsSize<4)
                {
                    // reshape: View of the input tensor that has been reshaped to the specified new dimensions.
                    // shuffle: A copy of the input tensor whose dimensions have been reordered according to the specified permutation.
                    Eigen::array<long int, 2> newshape({tnsDims[mode], newColDim});
```

```
                                        matricedTns = tnsX.shuffle(permutation).reshape(newshape);

                                // Map the @c Eigen Tensor to @c Eigen Matrix type.
                                        return tensorToMatrix(matricedTns, tnsDims[mode], newColDim);
                                }
                                else
                                {
                                        long int matricized_dim = 1;
                                        for(std::size_t i=1; i < permutation.size()-1; ++i)
                                        {
                                                matricized_dim *= tnsDims[permutation[i]];
                                        }
                                        Tensor<TnsSize-1> cubeTns;
                                        MatrixType        matricedTns(tnsDims[mode], newColDim);
                                        // Recursive Call for each cube, based on the mode if it is on the last dimension or not.
                                        if(mode<TnsSize-1)
                                        {
                                                for(int cubeId=0; cubeId<tnsDims[TnsSize-1]; cubeId++)
                                                {
                                                        // Get a TnsSize-1 order hypercube from whole tensor.
                                                        cubeTns = tnsX.chip(cubeId,TnsSize-1);
                                                        // Save the matriced 3D Tensor to matricedTns, after returning from the recursive calls.
                                                        matricedTns.block(0, cubeId * matricized_dim, tnsDims[mode], matricized_dim) = Matricization_rec(cubeTns, mode);
                                                }
                                        }
                                        else
                                        {
                                                for(int cubeId=0; cubeId<tnsDims[TnsSize-2]; cubeId++)
                                                {
                                                        // Get a TnsSize-1 order hypercube from whole tensor.
                                                        cubeTns = tnsX.chip(cubeId,TnsSize-2);
                                                        // Save the matriced 3D Tensor to matricedTns, after returning from the recursive calls.
                                                        matricedTns.block(0, cubeId * matricized_dim, tnsDims[mode], matricized_dim) = Matricization_rec(cubeTns, TnsSize-2);
                                                }
                                        }
                                        return matricedTns;
                                }

                }

                } // end namespace v2

  } // end namespace experimental
  #endif // DOXYGEN_SHOULD_SKIP_THIS

} // end namespace partensor

#endif // PARTENSOR_MATRICIZATION_HPP
```

8.41 MTTKRP.hpp File Reference

```
#include "PARTENSOR_basic.hpp"
#include "PartialKhatriRao.hpp"
#include <omp.h>
```

Functions

- template< std::size_t _TnsSize, typename Dimensions >
  void mttkrp (Dimensions const &tnsDims, std::array< Matrix, _TnsSize > const &factors, Matrix const &tns_mat, std::size_t const mode, int const num_threads, Matrix &result)
- template< std::size_t _TnsSize>
  void mttkrp (std::array< Matrix, _TnsSize > const &factors, Matrix const &tns_mat, int const &mode, Matrix &krao, Matrix &result)

8.41.1 Detailed Description

Implements the Matricized Tensor times Khatri Rao Product.

8.41.2 Function Documentation

8.41.2.1 mttkrp() [1/2]

```
void partensor::v1::mttkrp (
            Dimensions const & tnsDims,
            std::array< Matrix, _TnsSize > const & factors,
            Matrix const & tns_mat,
            std::size_t const mode,
            int const num_threads,
            Matrix & result )
```

Computes Matricized Tensor Times Khatri-Rao Product with the use of `OpenMP`.

Template Parameters

_TnsSize	Tensor Order.
Dimensions	Array type containing the Tensor dimensions.

Parameters

tnsDims	[in] `Stl` array containing the Tensor dimensions, whose length must be same as the Tensor order.
factors	[in] An `stl` array with the factors.
tns_mat	[in] Matricization of the Tensor based on `mode`.
mode	[in] The dimension where the tensor was matricized and the MTTKRP will be computed.
num_threads	[in] The number of available threads, defined by the environmental variable `OMP_NUM_THREADS`.
result	[in/out] The result matrix of the multiplication of the matricized tensor and the Khatri-Rao product.

8.41.2.2 mttkrp() [2/2]

```
void partensor::v1::mttkrp (
            std::array< Matrix, _TnsSize > const & factors,
            Matrix const & tns_mat,
            int const & mode,
            Matrix & krao,
            Matrix & result )
```

Computes the "Partial" Khatri-Rao product and the Matricized Tensor Khatri-Rao Product (MTTKRP). More specifically, computes the Khatri-Rao product for all `factors` apart from factors[`mode`] and use it to finally calculate the MTTKRP.

Template Parameters

_TnsSize	Size of the `factors` array.

Parameters

factors	[in] An `stl` array with the factors.
tns_mat	[in] Matricization of the Tensor based on `mode`.
mode	[in] The dimension where the tensor was matricized and the MTTKRP will be computed.
krao	[in,out] The result Khatri-Rao product for the `factors` excluding the `mode` factor.
result	[in/out] The result matrix of the multiplication of the matricized tensor and the Khatri-Rao product.

8.42 MTTKRP.hpp

Go to the documentation of this file.

```
#ifndef DOXYGEN_SHOULD_SKIP_THIS
#endif  // DOXYGEN_SHOULD_SKIP_THIS
/*********************************************************************/
#ifndef PARTENSOR_TENSOR_KHATRIRAO_PRODUCT_HPP
#define PARTENSOR_TENSOR_KHATRIRAO_PRODUCT_HPP

#include "PARTENSOR_basic.hpp"
#include "PartialKhatriRao.hpp"
#include <omp.h>

// #define EIGEN_DONT_PARALLELIZE
// #define NUM_SOCKETS                1
#define STATIC_SCHEDULE_CHUNK_SIZE 1
/* -- Declare custom reduction of variable_type : Matrix -- */
#pragma omp declare reduction(sum                            \
                              : Eigen::MatrixXd             \
                              : omp_out = omp_out + omp_in) \
    initializer(omp_priv = Eigen::MatrixXd::Zero(omp_orig.rows(), omp_orig.cols()))

namespace partensor {

  inline namespace v1 {

    template<std::size_t _TnsSize>
    void mttkrp(std::array<Matrix,_TnsSize> const &factors,
                Matrix                      const &tns_mat,
                int                         const &mode,
                Matrix                            &krao,
                Matrix                            &result)
    {
      krao             = PartialKhatriRao(factors, mode);
      result.noalias() = tns_mat * krao;
    }

    /*
     * Get number of threads, defined by the environmental variable $(OMP_NUM_THREADS).
     */
    inline int get_num_threads()
    {
      int threads;
      #pragma omp parallel
      {
        threads = omp_get_num_threads();
      }
      return threads;
    }

     template <std::size_t _TnsSize, typename Dimensions>
     void mttkrp(Dimensions                  const &tnsDims,
                 std::array<Matrix,_TnsSize> const &factors,
                 Matrix                      const &tns_mat,
                 std::size_t                 const  mode,
                 int                         const  num_threads,
                 Matrix                             &result)
     {
       #ifndef EIGEN_DONT_PARALLELIZE
```

```
        Eigen::setNbThreads(1);
    #endif

    constexpr std::size_t NUM_SOCKETS = 1;
    std::size_t           inner_num_threads = num_threads / NUM_SOCKETS;
    if (inner_num_threads < 1)
      inner_num_threads = 1;

    const int   rank       = factors[0].cols();
    std::size_t last_mode  = (mode == _TnsSize-1)  ? (_TnsSize - 2) : (_TnsSize - 1);
    std::size_t first_mode = (mode == 0) ? 1 : 0;

    result = Matrix::Zero(tnsDims[mode], rank);

    // dim = I_(1) * ... * I_(mode-1) * I_(mode+1) * ... * I_(N)
    long int dim = 1;
    for(std::size_t i=0; i<_TnsSize; ++i)
    {
      if(i!=mode)
        dim *= tnsDims[i];
    }
    // I_(first_mode+1) x I_(first_mode+2) x ... x I_(last_mode), where <I_(first_mode)> #rows of the starting factor.
    int num_of_blocks              = static_cast<int>(dim / static_cast<long int>(tnsDims[first_mode]));
    int numOfBlocks_div_NumSockets = num_of_blocks / NUM_SOCKETS;
    std::array<int, _TnsSize-2> rows_offset;

    for (int i = static_cast<int>(_TnsSize - 3), j = last_mode; i >= 0; i--, j--)
    {
      if (j == static_cast<int>(mode))
      {
        j--;
      }
      if (i == static_cast<int>(_TnsSize - 3))
      {
        rows_offset[i] = num_of_blocks / tnsDims[j];
      }
      else
      {
        rows_offset[i] = rows_offset[i + 1] / tnsDims[j];
      }
    }

    // --- If Factors are Transposed ---
    // omp_set_nested(1);
    // #pragma omp parallel for num_threads(NUM_SOCKETS) proc_bind(spread)
    // for (std::size_t sock_id=0; sock_id<NUM_SOCKETS; sock_id++)
    // {
    //   #pragma omp parallel for reduction(sum: result) schedule(static, STATIC_SCHEDULE_CHUNK_SIZE) num_threads(inner_num_threads) proc_bind(close)
    //   for (std::size_t block_idx = sock_id * numOfBlocks_div_NumSockets; block_idx < (sock_id + 1) * numOfBlocks_div_NumSockets + (sock_id + 1 == NUM_SOCKETS) * (num_of_blocks % NUM_SOCKETS); block_idx++)
    //   {
    //     // Compute Kr = KhatriRao(A_(last_mode)(l,:), A_(last_mode-1)(k,:), ..., A_(2)(j,:))
    //     // Initialize vector Kr as Kr = A_(last_mode)(l,:)
    //     Matrix Kr(rank,1);
    //     Kr = factors[last_mode].col((block_idx / rows_offset[_TnsSize - 3]) % tnsDims[last_mode]);
    //     Matrix PartialKR(rank, tnsDims[first_mode]);

    //     // compute "partial" KhatriRao as a recursive Hadamard Product : Kr = Kr .* A_(j)(...,:)
    //     for (std::size_t i = _TnsSize - 4, j = last_mode - 1; i >= 0; i--, j--)
    //     {
    //       if (j == mode)
    //       {
    //         j--;
    //       }
    //       Kr = (factors[j].col((block_idx / rows_offset[i]) % tnsDims[j])).cwiseProduct(Kr);
    //     }

    //     // Compute block of KhatriRao, using "partial" vector Kr and first Factor A_(first_mode), as : KhatriRao(Kr, A_(first_mode)(:,:))
    //     for (int col = 0; col < tnsDims[first_mode]; col++)
    //     {
    //       PartialKR.col(col) = ((factors[first_mode].col(col)).cwiseProduct(Kr));
    //     }

    //     result.noalias() += tns_mat.block(0, block_idx * tnsDims[first_mode], tnsDims[mode], tnsDims[first_mode]) * PartialKR.transpose();
    //   }
    // }
    // #ifndef EIGEN_DONT_PARALLELIZE
    //     Eigen::setNbThreads(num_threads);
    // #endif

    omp_set_nested(1);
```

```
    #pragma omp parallel for num_threads(NUM_SOCKETS) proc_bind(spread)
    for (std::size_t sock_id=0; sock_id<NUM_SOCKETS; sock_id++)
    {
      #pragma omp parallel for reduction(sum: result) schedule(static, STATIC_SCHEDULE_CHUNK_SIZE) num_threads(inner_num_threads) proc_bind(close)
      for (std::size_t block_idx = sock_id * numOfBlocks_div_NumSockets; block_idx < (sock_id + 1) * numOfBlocks_div_NumSockets + (sock_id + 1 == NUM_SOCKETS) * (num_of_blocks % NUM_SOCKETS); block_idx++)
      // for (int block_idx = 0; block_idx < num_of_blocks; block_idx++)
      {
        // Compute Kr = KhatriRao(A_(last_mode)(l,:), A_(last_mode-1)(k,:), ..., A_(2)(j,:))
        // Initialize vector Kr as Kr = A_(last_mode)(l,:)
        Matrix Kr(1, rank);
        Kr = factors[last_mode].row((block_idx / rows_offset[_TnsSize - 3]) % tnsDims[last_mode]);
        Matrix PartialKR(tnsDims[first_mode], rank);
        // compute "partial" KhatriRao as a recursive Hadamard Product : Kr = Kr .* A_(j)(...,:)
        for (int i = static_cast<int>(_TnsSize - 4), j = last_mode - 1; i >= 0; i--, j--)
        {
          if (j == static_cast<int>(mode))
          {
            j--;
          }
          Kr = (factors[j].row((block_idx / rows_offset[i]) % tnsDims[j])).cwiseProduct(Kr);
        }

        // Compute block of KhatriRao, using "partial" vector Kr and first Factor A_(first_mode), as : KhatriRao(Kr, A_(first_mode)(:,:))
        for (int row = 0; row < tnsDims[first_mode]; row++)
        {
          PartialKR.row(row) = ((factors[first_mode].row(row)).cwiseProduct(Kr));
        }

        result.noalias() += tns_mat.block(0, block_idx * tnsDims[first_mode], tnsDims[mode], tnsDims[first_mode]) * PartialKR;
      }
    }
    #ifndef EIGEN_DONT_PARALLELIZE
        Eigen::setNbThreads(num_threads);
    #endif
  }

  // trasposed_v
  // Serial (using std::array for tns_dimensions)
  template<std::size_t _TnsSize>
  void SparseMTTKRP(const std::array<int, _TnsSize>     &tns_dimensions,
                    const SparseMatrix                  &sparse_tns,
                    const std::array<Matrix, _TnsSize> &factors,
                    const int                            rank,
                    const std::array<int, _TnsSize-1>     &offsets,
                    const int                            last_mode,
                    const int                            cur_mode,
                    Matrix                              &MTTKRP)
  {
    Matrix temp_R_1(rank, 1);
    Matrix MTTKRP_col(rank, 1);

    for (long int i = 0; i < sparse_tns.outerSize(); ++i)
    {
      MTTKRP_col = Matrix::Zero(rank, 1);
      for (typename SparseMatrix::InnerIterator it(sparse_tns, i); it; ++it)
      {
        temp_R_1 = Matrix::Ones(rank, 1);
        int row;
        // Select rows of each factor an compute the respective row of the Khatri-Rao product.
        for (int mode_i = last_mode, kr_counter = static_cast<int>(_TnsSize) - 2; mode_i >= 0 && kr_counter >= 0; mode_i--)
        {

          if (mode_i == cur_mode)
          {
            continue;
          }
          row      = ((it.row()) / offsets[kr_counter]) % (tns_dimensions[mode_i]);
          temp_R_1 = temp_R_1.cwiseProduct(factors[mode_i].col(row));
          kr_counter--;
        }
        // Subtract from the previous row the respective row of W, according to relation (9).
        // MTTKRP.row(it.col()) -= it.value() * temp_R_1;
        MTTKRP_col -= it.value() * temp_R_1;
      }
      MTTKRP.col(i) = MTTKRP_col;
    }
  }

  // Parallel OpenMP (using std::array for tns_dimensions)
  template<std::size_t _TnsSize>
  void SparseMTTKRP_omp(const std::array<int, _TnsSize>     &tns_dimensions,
```

```
                        const SparseMatrix                 &sparse_tns,
                        const std::array<Matrix, _TnsSize> &factors,
                        const int                           rank,
                        const std::array<int, _TnsSize-1>    &offsets,
                        const int                           last_mode,
                        const int                           cur_mode,
                        Matrix                             &MTTKRP)
  {
    Matrix temp_R_1(rank, 1);
    Matrix MTTKRP_col(rank, 1);

    #pragma omp for schedule(dynamic) nowait
    for (long int i = 0; i < sparse_tns.outerSize(); ++i)
    {
      MTTKRP_col = Matrix::Zero(rank, 1);
      for (typename SparseMatrix::InnerIterator it(sparse_tns, i); it; ++it)
      {
        temp_R_1 = Matrix::Ones(rank, 1);
        long int row;
        // Select rows of each factor an compute the respective row of the Khatri-Rao product.
        for (int mode_i = last_mode, kr_counter = static_cast<int>(_TnsSize) - 2; mode_i >= 0 && kr_counter >= 0; mode_i--)
        {

          if (mode_i == cur_mode)
          {
            continue;
          }
          row      = ((it.row()) / offsets[kr_counter]) % (tns_dimensions[mode_i]);
          temp_R_1 = temp_R_1.cwiseProduct(factors[mode_i].col(row));
          kr_counter--;
        }
        // Subtract from the previous row the respective row of W, according to relation (9).
        // MTTKRP.row(it.col()) -= it.value() * temp_R_1;
        MTTKRP_col -= it.value() * temp_R_1;
      }
      MTTKRP.col(i) = MTTKRP_col;
    }
  }

  // Parallel (using std::vector for tns_dimensions)
  template<std::size_t _TnsSize>
  void SparseMTTKRP(const std::vector<std::vector<int>> &tns_dimensions,
                    const std::array<int, _TnsSize>      &fiber_rank,
                    const SparseMatrix                   &sparse_tns,
                    const std::array<Matrix, _TnsSize>   &factors,
                    const int                             rank,
                    const std::array<int, _TnsSize-1>    &offsets,
                    const int                             last_mode,
                    const int                             cur_mode,
                    Matrix                               &MTTKRP)
  {
    Matrix temp_R_1(rank, 1);
    Matrix MTTKRP_col(rank, 1);

    for (long long int i = 0; i < sparse_tns.outerSize(); ++i)
    {
      MTTKRP_col.setZero();
      for (typename SparseMatrix::InnerIterator it(sparse_tns, i); it; ++it)
      {
        temp_R_1 = Matrix::Ones(rank, 1);
        long long int row;
        // Select rows of each factor an compute the respective row of the Khatri-Rao product.
        for (int mode_i = last_mode, kr_counter = static_cast<int>(_TnsSize) - 2; mode_i >= 0 && kr_counter >= 0; mode_i--)
        {
          if (mode_i == cur_mode)
          {
            continue;
          }
          row      = ((it.row()) / offsets[kr_counter]) % (tns_dimensions[mode_i][fiber_rank[mode_i]]);

          temp_R_1 = temp_R_1.cwiseProduct(factors[mode_i].col(row));
          kr_counter--;
        }
        // Subtract from the previous row the respective row of W, according to relation (9).
        MTTKRP_col -= it.value() * temp_R_1;
      }
      MTTKRP.col(i) = MTTKRP_col;
    }
  }

} // end namespace v1

namespace v2 // std_v
{
```

```
    // Serial (using std::array for tns_dimensions)
    template<std::size_t _TnsSize>
    void SparseMTTKRP(const std::array<int, _TnsSize>     &tns_dimensions,
                      const SparseMatrix                  &sparse_tns,
                      const std::array<Matrix, _TnsSize> &factors,
                      const int                            rank,
                      const std::array<int, _TnsSize>     &offsets,
                      const int                            last_mode,
                      const int                            cur_mode,
                      Matrix                              &MTTKRP)
    {
      Matrix temp_1_R(1, rank);
      Matrix MTTKRP_row(1, rank);

      MTTKRP.setZero();

      for (long int i = 0; i < sparse_tns.outerSize(); ++i)
      {
        MTTKRP_row = Matrix::Zero(1, rank);
        for (typename SparseMatrix::InnerIterator it(sparse_tns, i); it; ++it)
        {
          temp_1_R = Matrix::Ones(1, rank);
          int row;
          // Select rows of each factor an compute the respective row of the Khatri-Rao product.
          for (int mode_i = last_mode, kr_counter = static_cast<int>(_TnsSize) - 2; mode_i >= 0 &&
   kr_counter >= 0; mode_i--)
          {

            if (mode_i == cur_mode)
            {
              continue;
            }
            row      = ((it.row()) / offsets[kr_counter]) % (tns_dimensions[mode_i]);
            temp_1_R = temp_1_R.cwiseProduct(factors[mode_i].row(row));
            kr_counter--;
          }
          // Subtract from the previous row the respective row of W, according to relation (9).
          MTTKRP_row -= it.value() * temp_1_R;
        }
        MTTKRP.row(i) = MTTKRP_row;
      }
    }

    // Parallel (using std::vector for tns_dimensions)
    template<std::size_t _TnsSize>
    void SparseMTTKRP(const std::vector<int>              &tns_dimensions,
                      const std::array<int, _TnsSize>     &fiber_rank,
                      const SparseMatrix                  &sparse_tns,
                      const std::array<Matrix, _TnsSize> &factors,
                      const int                            rank,
                      const std::array<int, _TnsSize>     &offsets,
                      const int                            last_mode,
                      const int                            cur_mode,
                      Matrix                              &MTTKRP)
    {
      Matrix temp_1_R(1, rank);
      Matrix MTTKRP_row(1, rank);

      MTTKRP.setZero();

      for (long int i = 0; i < sparse_tns.outerSize(); ++i)
      {
        MTTKRP_row = Matrix::Zero(1, rank);
        for (typename SparseMatrix::InnerIterator it(sparse_tns, i); it; ++it)
        {
          temp_1_R = Matrix::Ones(1, rank);
          int row;
          // Select rows of each factor an compute the respective row of the Khatri-Rao product.
          for (int mode_i = last_mode, kr_counter = static_cast<int>(_TnsSize) - 2; mode_i >= 0 &&
   kr_counter >= 0; mode_i--)
          {

            if (mode_i == cur_mode)
            {
              continue;
            }
            row      = ((it.row()) / offsets[kr_counter]) % (tns_dimensions[fiber_rank[mode_i]]);
            temp_1_R = temp_1_R.cwiseProduct(factors[mode_i].row(row));
            kr_counter--;
          }
          // Subtract from the previous row the respective row of W, according to relation (9).
          MTTKRP_row -= it.value() * temp_1_R;
        }
        MTTKRP.row(i) = MTTKRP_row;
      }
    }

```

```
  } // end namespace v2

} // end namespace partensor

#endif // PARTENSOR_TENSOR_KHATRIRAO_PRODUCT_HPP
```

8.43 NesterovMNLS.hpp File Reference

```
#include "PARTENSOR_basic.hpp"
#include "MTTKRP.hpp"
```

Functions

- void ComputeSVD (Matrix const &mat, double &L, double &mu)

    `SVD` decomposition of a `Matrix`.
- void GLambda (double mu, double &L, double &lambda, double &q)
- void NesterovMNLS (Matrix const &mat1, Matrix const &mat2, double const delta_1, double const delta_2, Matrix &res)
- double UpdateAlpha (double const alpha, double const q)

8.43.1 Detailed Description

Implementation of the Nesterov's (Accelerated Gradient) algorithm with nonnegative constraints. It also provides, supporting functions for Nesterov's algorithm.

8.43.2 Function Documentation

8.43.2.1 ComputeSVD()

```
void partensor::v1::ComputeSVD (
            Matrix const & mat,
            double & L,
            double & mu )
```

`SVD` decomposition of a `Matrix`.

Computes the `svd` decomposition of an `Matrix` with `Jacobi's` implementation from `Eigen`.

Parameters

mat	[in] The `Matrix` to be decomposed.
L	[in,out] The maximum singular value from the `svd` (returned).
mu	[in,out] The minimum singular value from the `svd` (returned).

8.43.2.2 GLambda()

```
void partensor::v1::GLambda (
            double mu,
            double & L,
            double & lambda,
            double & q ) [inline]
```

Computes necessary quantities for `NesterovMNLS` (Nesterov matrix-nonnegative- least-squares) algorithm.

Parameters

mu	[in] The minimum singular value from the `svd` (computed in `ComputeSVD`).
L	[in,out] The maximum singular value from the `svd` (computed in `ComputeSVD`).
lambda	[in,out] Normalization parameter.
q	[in,out] Inverse of condition number of the problem used in `UpdateAlpha`.

8.43.2.3 NesterovMNLS()

```
void partensor::v1::NesterovMNLS (
            Matrix const & mat1,
            Matrix const & mat2,
            double const delta_1,
            double const delta_2,
            Matrix & res )
```

Nesterov's algorithm with no-negative constraints and proximal term.

Let `X` belongs in `R` with dimensions `m` x `n`, `A` belongs in `R` with dimensions `m` x `r`, `B` belongs in `R` with dimensions `n` x `r` and consider the minimization problem `0.5*Frobenius_norm`(X-AB')^2. We can use the following function in order to solve this problem.

Parameters

mat1	[in] The covariance `Matrix` B'∗B.
mat2	[in] A `Matrix` containing ther resultt of -(X∗B).
delta← _1	[in] If `Y` is the updated matrix at each iteration, then `delta_1` is the maximum tolerance for the value `abs`(gradient(Y).∗Y).
delta← _2	[in] If `Y` is the updated matrix at each iteration, then `delta_2` is the minimum tolerance for the value `gradient(Y).`
res	[in,out] The result `Matrix` from Nesterov's algorithm.

8.43.2.4 UpdateAlpha()

```
double partensor::v1::UpdateAlpha (
              double const alpha,
              double const q )  [inline]
```

Computes the interpolation quantity for `NesterovMNLS` (Nesterov minimum-nonnegative-least-squares) algorithm.

Parameters

alpha	[in] Starting value for interpolation.
q	[in] Inverse of condition number of the problem used in `NesterovMNLS`.

Returns

The interpolation quantity.

8.44 NesterovMNLS.hpp

Go to the documentation of this file.

```
#ifndef DOXYGEN_SHOULD_SKIP_THIS
#endif  // DOXYGEN_SHOULD_SKIP_THIS
/*********************************************************************/
#ifndef PARTENSOR_NESTEROV_MNLS_HPP
#define PARTENSOR_NESTEROV_MNLS_HPP

#include "PARTENSOR_basic.hpp"
#include "MTTKRP.hpp"

namespace partensor
{

        inline namespace v1 {
                void ComputeSVD( Matrix const &mat,
                                                double       &L,
                                                double       &mu )
                {
                        Eigen::JacobiSVD<Matrix> svd(mat, Eigen::ComputeThinU | Eigen::ComputeThinV);
                        L = svd.singularValues().maxCoeff();
                        mu = svd.singularValues().minCoeff();
                }

                void ComputeEIG(Matrix const &mat,
                                double      &L,
                                                double       &mu)
                {
                        Eigen::EigenSolver<Matrix> eig(mat);
                        L  = (eig.eigenvalues().real()).maxCoeff();
                        mu = (eig.eigenvalues().real()).minCoeff();
                }

                inline void GLambda( double  mu,
                                                        double &L,
                                                        double &lambda,
                                                        double &q     )
                {
                        q = mu/L;
                        if (1/q>1e6)
                                lambda = 10 * mu;
                        else if (1/q>1e3)
                                lambda = mu;
                        else
                                lambda = mu/10;

                        L += lambda;
                        mu += lambda;
                        q = mu/L;
                }

```

```
inline double UpdateAlpha( double const alpha,
                           double const q     )
{
        double a, b, c, D;
        a = 1;
        b = alpha*alpha - q;
        c = -alpha*alpha;
        D = b*b - 4*a*c;

        return (-b+sqrt(D))/2;
}

void TuneLambda(const double &L,
        double       &lambda,
                const double &ratio)
{
        // if(ratio < 1)
        // {
        //      lambda = 0.01;
        // }
        // else
        // {
        //      lambda = (L * ratio) / (1 - ratio);
        // }
        lambda = L/ratio;
}

// Returns maximum eigenvalue L of square matrix mat
inline double PowerMethod( Matrix &mat, const double epsilon)
{
        Matrix x_init = Matrix::Random(mat.cols(),1);
        Matrix x_new = x_init;
        Matrix Ax = x_init;
        double norm_Ax;
        double lambda_max = 0;;

        int iter = 0;
        int MAX_ITER = 1e+4;

        while (1)
        {
                Ax.noalias() = mat * x_init;
                norm_Ax = Ax.norm();
                x_new.noalias() = 1/(norm_Ax + 1e-12) * Ax;


                if ((x_new - x_init).norm() <= epsilon || iter >= MAX_ITER)
                {
                        lambda_max = (x_new.transpose() * (mat * x_new))(0);
                        break;
                }

                x_init = x_new;
                iter++;
        }

        return lambda_max;
}

void NesterovMNLS( Matrix const &mat1,
                   Matrix const &mat2,
                   double const  delta_1,
                   double const  delta_2,
                   Matrix       &res     )
{
        const int max_inner = 50;

        int m    = res.rows();
        int r    = res.cols();
        int iter = 0;
        double L, mu, lambda, q, alpha, new_alpha, beta;

        Matrix A(m, r);
        Matrix Y(m, r);
        Matrix new_A(m, r);
        Matrix grad_Y(m, r);
        Matrix _mat1 = mat1;
        Matrix _mat2 = mat2;
        Matrix Zero_Matrix = Matrix::Zero(m, r);

        ComputeSVD(mat1, L, mu);
        GLambda(mu, L, lambda, q);

        _mat1 += lambda * Matrix::Identity(r, r);
        _mat2 += lambda * res;
        // q      = mu/L;
        alpha  = 1;
```

```
                        A           = res;
                        Y           = res;

                        while(1)
                        {
                                grad_Y                  = -_mat2;               // |
                                grad_Y.noalias() += Y * _mat1;          // | grad_Y = W + Y * Z.transpose();

                                if ((grad_Y.cwiseProduct(Y).cwiseAbs().maxCoeff() <= delta_1 && grad_Y.minCoeff() >= -delta_2)  || (iter >= max_inner))
                                                break;

                                new_A     = (Y - grad_Y/L).cwiseMax(Zero_Matrix);

                                // if ( ( (new_A-Y).norm()/Y.norm()<=delta_1 ) || (iter >= max_inner))
                                //      break;

                                new_alpha = UpdateAlpha(alpha, q);
                                beta      = alpha * (1 - alpha) / (alpha*alpha + new_alpha);

                                Y     = (1 + beta) * new_A - beta * A;
                                A     = new_A;
                                alpha = new_alpha;
                                iter++;
                        }
                        res  = A;
                }

                // GTC Serial
                // transposed_v
                template<std::size_t _TnsSize>
                void NesterovMNLS(Matrix                                        &mat1,
                                        std::array<Matrix, _TnsSize>       &factors,
                                        std::array<int, _TnsSize>     const &tns_dimensions,
                                        SparseMatrix                  const &tns_spMat,
                        std::array<int, _TnsSize-1>  const &offsets,
                                        int     const                         max_nest_iter,
                                        double const                          ratio,
                                        int     const                         cur_mode,
                                        Constraint const                      constraint_i,
                                        Matrix                               &MTTKRP_T)
                {
                        int m = factors[cur_mode].cols();
                        int r = factors[cur_mode].rows();
                        double L, mu, q, alpha, new_alpha, beta, lambda;
                        int iter = 0;

                        Matrix grad_Y(r, m);
                        Matrix Y(r, m);
                        Matrix new_A(r, m);
                        Matrix A(r, m);
                        Matrix Zero_Matrix = Matrix::Zero(r, m);

                        ComputeEIG(mat1, L, mu);

                        lambda = ratio;
                        L = L + lambda;
                        q = lambda / L;
                        alpha = 1;

                        A = factors[cur_mode];
                        Y = factors[cur_mode];

                        int last_mode = (cur_mode == static_cast<int>(_TnsSize) - 1) ? static_cast<int>(_TnsSize) - 2 : static_cast<int>(_TnsSize) - 1;

                        Matrix temp_R_1(r, 1);
                        Matrix temp_col = Matrix::Zero(r, 1);
                        while (1)
                        {
                                grad_Y.setZero();

                                if (iter >= max_nest_iter)
                                {
                                        break;
                                }

                                // Compute grad_Y
                                for (long int i = 0; i < tns_spMat.outerSize(); ++i)
                                {
                                        temp_col.setZero();
                                        for (SparseMatrix::InnerIterator it(tns_spMat, i); it; ++it)
                                        {
                                                temp_R_1 = Matrix::Ones(r, 1);
                                                // Select rows of each factor an compute the respective row of the Khatri-Rao product.
```

```
                                                for (int mode_i = last_mode, kr_counter = static_cast<int>(_TnsSize) - 2; mode_i >= 0 && kr_counter >= 0; mode_i--)
                                                {
                                                        if (mode_i == cur_mode)
                                                        {
                                                                continue;
                                                        }
                                                        long int row;
                                                        row      = ((it.row()) / offsets[kr_counter]) % (tns_dimensions[mode_i]);
                                                        temp_R_1 = temp_R_1.cwiseProduct(factors[mode_i].col(row));
                                                        kr_counter--;
                                                }
                                                // Computation of row of Z according the relation (10) of the paper.
                                                temp_col += (temp_R_1.transpose() * Y.col(i))(0) * temp_R_1;
                                        }
                                        grad_Y.col(i) = temp_col;
                                }

                                // Add proximal term.
                                grad_Y += MTTKRP_T + lambda * Y;
                                if (constraint_i == Constraint::unconstrained)
                                {
                                        new_A = (Y - grad_Y / L);
                                }
                                else // Use projection
                                {
                                        new_A = (Y - grad_Y / L).cwiseMax(Zero_Matrix);
                                }

                                new_alpha = UpdateAlpha(alpha, q);
                                beta      = alpha * (1 - alpha) / (alpha * alpha + new_alpha);

                                Y = (1 + beta) * new_A - beta * A;

                                // Update Y

                                A     = new_A;
                                alpha = new_alpha;
                                iter++;
                        }
                        factors[cur_mode] = A;

                }

                // OpenMP
                template<std::size_t _TnsSize>
                void NesterovMNLS(Matrix                              &mat1,
                                  std::array<Matrix, _TnsSize>        &factors,
                                  std::array<int, _TnsSize>     const &tns_dimensions,
                                  SparseMatrix                  const &tns_spMat,
                        const std::array<int, _TnsSize-1>     &offsets,
                                  Matrix                               &Y,
                                  int     const                        max_nest_iter,
                                  double const                         ratio,
                                  int     const                        cur_mode,
                                  Constraint const                     constraint_i,
                                  Matrix                               &MTTKRP_T)
                {
                        int r = factors[cur_mode].rows();
                        double L, mu, q, alpha, new_alpha, beta, lambda;
                        int iter = 0;
                        long int row;

                        Matrix new_A_vec(r, 1);
                        Matrix Zero_Vec = Matrix::Zero(r, 1);

                        ComputeEIG(mat1, L, mu);

                        lambda = ratio;
                        L = L + lambda;
                        q = lambda / L;
                        alpha = 1;

                        #pragma omp master
                        Y = factors[cur_mode];
                        #pragma omp barrier

                        int last_mode = (cur_mode == static_cast<int>(_TnsSize) - 1) ? static_cast<int>(_TnsSize) - 2 : static_cast<int>(_TnsSize) - 1;

                        Matrix temp_R_1(r, 1);

                        #pragma omp barrier
```

```
            while (1)
            {
                if (iter >= max_nest_iter)
                {
                    break;
                }

                Matrix temp_col = Matrix::Zero(r, 1);
                new_alpha       = UpdateAlpha(alpha, q);
                beta            = alpha * (1 - alpha) / (alpha * alpha + new_alpha);

                // Compute grad_Y
                #pragma omp for schedule(dynamic) //ordered
                for (long int i = 0; i < tns_spMat.outerSize(); ++i)
                {
                    temp_col.setZero();
                    for (SparseMatrix::InnerIterator it(tns_spMat, i); it; ++it)
                    {
                        temp_R_1 = Matrix::Ones(r, 1);
                        // Select rows of each factor an compute the respective row of the Khatri-Rao product.
                        for (int mode_i = last_mode, kr_counter = static_cast<int>(_TnsSize) - 2; mode_i >= 0 && kr_counter >= 0; mode_i--)
                        {
                            if (mode_i == cur_mode)
                            {
                                continue;
                            }
                            row      = ((it.row()) / offsets[kr_counter]) % (tns_dimensions[mode_i]);
                            temp_R_1 = temp_R_1.cwiseProduct(factors[mode_i].col(row));
                            kr_counter--;
                        }
                        // Computation of row of Z according the relation (10) of the paper.
                        temp_col += (temp_R_1.transpose() * Y.col(i))(0) * temp_R_1;
                    }

                    temp_col.noalias() += MTTKRP_T.col(i) + (lambda * Y.col(i));

                    // Add proximal term.
                    if (constraint_i == Constraint::unconstrained)
                    {
                        new_A_vec = (Y.col(i) - temp_col / L);
                    }
                    else // Use projection
                    {
                        new_A_vec = (Y.col(i) - temp_col / L).cwiseMax(Zero_Vec);
                    }

                    Y.col(i) = (1 + beta) * new_A_vec - beta * factors[cur_mode].col(i);
                    factors[cur_mode].col(i) = new_A_vec;
                }
                alpha = new_alpha;
                iter++;
            }

            #pragma omp barrier
        }

        namespace dynamic_blocksize
        {
            template <std::size_t _TnsSize>
            void StochasticNesterovMNLS(std::array<Matrix, _TnsSize>      &factors,
                                        std::array<int, _TnsSize>    const &tns_dimensions,
                                        SparseMatrix                 const &tns_spMat,
                                        std::array<int, _TnsSize-1> const &offsets,
                                        double                       const  c_stochastic_perc,
                                        int                          const  max_nest_iter,
                                        double                       const  lambda,
                                        int                          const  cur_mode)
            {
                int r = factors[cur_mode].rows();
                double L2, inv_L2;
                double sqrt_q = 0, beta = 0;
                int iter = 0;
```

```
long int row;

const Matrix zero_vec = Matrix::Zero(r, 1);
Matrix new_A_vec = Matrix::Zero(r, 1);

Matrix Y = factors[cur_mode];

// int first_mode = (cur_mode == 0) ? 1 : 0;
int last_mode = (cur_mode == static_cast<int>(_TnsSize) - 1) ? static_cast<int>(_TnsSize) - 2 : static_cast<int>(_TnsSize) - 1;

Matrix temp_R_1 = Matrix::Ones(r, 1);

Matrix temp_RxR(r, r);
Matrix temp_col = Matrix::Zero(r, 1);

std::srand(std::time(nullptr));

while (1)
{

        if (iter >= max_nest_iter)
        {
                break;
        }

        // Compute grad_Y
        for (long int i = 0; i < tns_spMat.outerSize(); ++i)
        {
                temp_col.setZero();
                // -- Pseudocode --
                // SparseMatrix::InnerIterator it(tns_spMat, i);
                // int length = tns_spMat.innerNonZeros(); // ???? check if innerNonzeros, we want to count the nonzeros entries in the row
                // int random_pivot = rand(length - blocksize)
                // it += random_pivot;

                // Get the number of nnz per row of matricization.
                long int nnzs_per_col = tns_spMat.innerVector(i).nonZeros();

                long int var_blocksize_i = static_cast<long int>(c_stochastic_perc * nnzs_per_col);

                if (var_blocksize_i > 0)
                {
                        // Choose a pivot from [0, nnzs_per_col - blocksize].
                        long int pivot = (std::rand() % (nnzs_per_col - var_blocksize_i + 1));

                        SparseMatrix::InnerIterator it(tns_spMat, i);
                        // Iterate over [pivot, pivot + var_blocksize_i] nnz elements per row.
                        // it += pivot;
                        for (long int acuum = 0; acuum < pivot; acuum++)
                                ++it;

                        // std::cout « pivot « "\t" « nnzs_per_col « std::endl;
                        temp_RxR.setZero();

                        for (long int sample = 0; sample < var_blocksize_i; sample++, ++it)
                        {
                                temp_R_1 = Matrix::Ones(r, 1);
                                // Select rows of each factor an compute the respective row of the Khatri-Rao product.
                                for (int mode_i = last_mode, kr_counter = static_cast<int>(_TnsSize) - 2; mode_i >= 0 && kr_counter >= 0; mode_i--)
                                {
                                        if (mode_i == cur_mode)
                                        {
                                                continue;
                                        }
                                        row = ((it.row()) / offsets[kr_counter]) % (tns_dimensions[mode_i]);
                                        // temp_R_1 = temp_R_1.cwiseProduct(factors[mode_i].row(row));
                                        temp_R_1 = temp_R_1.cwiseProduct(factors[mode_i].col(row));
                                        kr_counter--;
                                }
                                // Computation of row of Z according the relation (10) of the paper.
                                // Subtract each term of MTTKRP's row.
                                temp_col.noalias() +=
```

```
((temp_R_1.transpose() * Y.col(i))(0) - it.value()) * temp_R_1;

                        // Estimate Hessian for each row.
                        temp_RxR.noalias() += (temp_R_1 * temp_R_1.transpose());
                    }

                    // Solve with l2-regularization term
                    // temp_col.noalias() += (lambda * Y.col(i));
                    // Solve with Proximal term
                    temp_col.noalias() += lambda * (Y.col(i) + factors[cur_mode].col(i));

                    L2 = PowerMethod(temp_RxR, 1e-3);
                    L2 += lambda;
                    inv_L2 = 1 / L2;

                    new_A_vec.noalias() = (Y.col(i) - inv_L2 * temp_col);
                    new_A_vec           = (new_A_vec).cwiseMax(zero_vec);

                    sqrt_q = sqrt(lambda * inv_L2);

                    beta = (1 - sqrt_q) / (1 + sqrt_q);

                    // Update Y
                    Y.col(i) = (1 + beta) * new_A_vec - beta * factors[cur_mode].col(i);
                    // Update i-th column of current factor
                    factors[cur_mode].col(i) = new_A_vec;
                }
            }
            // alpha = new_alpha;
            iter++;
        }
    }


    namespace local_L
    {
        // OpenMP
        template<std::size_t _TnsSize>
        void StochasticNesterovMNLS(std::array<Matrix, _TnsSize>       &factors,
                                    std::array<int, _TnsSize>     const &tns_dimensions,
                                    SparseMatrix                  const &tns_spMat,
                                    const std::array<int, _TnsSize-1>  &offsets,
                                    double                        const  c_stochastic_perc,
                                    Matrix                              &Y,
                                    int     const                        max_nest_iter,
                                    double  const                        lambda,
                                    int     const                        cur_mode)
        {
            int r = factors[cur_mode].rows();
            double sqrt_q = 0, beta = 0, L2 = 0, inv_L2 = 0;
            int iter = 0;
            long int row;

            Matrix new_A_vec = Matrix::Zero(r, 1);
            Matrix zero_vec  = Matrix::Zero(r, 1);

            #pragma omp master
            Y = factors[cur_mode];

            int last_mode = (cur_mode == static_cast<int>(_TnsSize) - 1) ? static_cast<int>(_TnsSize) - 2 : static_cast<int>(_TnsSize) - 1;

            Matrix temp_R_1 = Matrix::Ones(r, 1);
            Matrix temp_RxR(r, r);
            Matrix temp_col(r, 1);

            std::srand(std::time(nullptr));

            #pragma omp barrier
            while (1)
            {
                if (iter >= max_nest_iter)
                {
```

```
                break;
            }

            // Compute grad_Y
            // #pragma omp for schedule(dynamic, 8) ordered
            // #pragma omp for schedule(guided) nowait
            #pragma omp for schedule(dynamic) nowait
            for (long int i = 0; i < tns_spMat.outerSize(); ++i)
            {
                // Get the number of nnz per row of matricization.
                long int nnzs_per_col = tns_spMat.innerVector(i).nonZeros();

                long int var_blocksize_i = static_cast<long int>(c_stochastic_perc * nnzs_per_col);

                if (var_blocksize_i > 0)
                {
                    temp_col.setZero();

                    // Choose a pivot from [0, nnzs_per_col - blocksize].
                    long int pivot = (std::rand() % (nnzs_per_col - var_blocksize_i + 1));

                    SparseMatrix::InnerIterator it(tns_spMat, i);
                    // Iterate over [pivot, pivot + var_blocksize_i] nnz elements per row.
                    // it += pivot;
                    for (long int acuum = 0; acuum < pivot; acuum++)
                        ++it;

                    temp_RxR.setZero();

                    for (long int sample = 0; sample < var_blocksize_i; sample++, ++it)
                    {
                        temp_R_1 = Matrix::Ones(r, 1);

                        // Select rows of each factor an compute the respective row of the Khatri-Rao product.
                        for (int mode_i = last_mode, kr_counter = static_cast<int>(_TnsSize) - 2; mode_i >= 0 && kr_counter >= 0; mode_i--)
                        {
                            if (mode_i == cur_mode)
                            {
                                continue;
                            }
                            row      = ((it.row()) / offsets[kr_counter]) % (tns_dimensions[mode_i]);
                            temp_R_1 = temp_R_1.cwiseProduct(factors[mode_i].col(row));
                            kr_counter--;
                        }
                        // Computation of row of Z according the relation (10) of the paper.
                        temp_col.noalias() += ((temp_R_1.transpose() * Y.col(i))(0) - it.value()) * temp_R_1;

                        // Estimate Hessian for each row.
                        temp_RxR.noalias() += (temp_R_1 * temp_R_1.transpose());
                    }

                    temp_col.noalias() += lambda * (Y.col(i) + factors[cur_mode].col(i));

                    L2 = PowerMethod(temp_RxR, 1e-3);
                    L2 += lambda;
                    inv_L2 = 1 / L2;

                    new_A_vec.noalias() = (Y.col(i) - inv_L2 * temp_col);
                    new_A_vec           = (new_A_vec).cwiseMax(zero_vec);

                    sqrt_q = sqrt( lambda * inv_L2 );

                    beta = (1 - sqrt_q) / (1 + sqrt_q);

                    // Update Y
                    Y.col(i) = (1 + beta) * new_A_vec - beta
```

```
* factors[cur_mode].col(i);

                                                        // Update i-th column of current factor
                                                        factors[cur_mode].col(i) = new_A_vec;

                                                }
                                        }
                                        iter++;
                                }
                                #pragma omp barrier
                        }

                }// end namespace local_L

        }// end namespace dynamic_blocksize

} // end namespace v1

namespace std_V
{
        // GTC Serial
        template<std::size_t _TnsSize>
        void NesterovMNLS(Matrix                                &mat1,
                          std::array<Matrix, _TnsSize>          &factors,
                          std::array<int, _TnsSize>     const &tns_dimensions,
                          SparseMatrix                  const &tns_spMat,
                          const std::array<int, _TnsSize-1>    &offsets,
                          int         const                    max_nest_iter,
                          double      const                    ratio,
                          int         const                    cur_mode,
                          Constraint  const                    constraint_i,
                          Matrix      const                    &MTTKRP)
        {
                int m = factors[cur_mode].rows();
                int r = factors[cur_mode].cols();
                double L, mu, q, alpha, new_alpha, beta, lambda;
                int iter = 0;

                Matrix grad_Y(m, r);
                Matrix Y(m, r);
                Matrix new_A(m, r);
                Matrix A(m, r);
                Matrix Zero_Matrix = Matrix::Zero(m, r);

                ComputeEIG(mat1, L, mu);

                lambda = ratio; // TuneLambda(L, lambda, ratio);
                L = L + lambda;
                q = lambda / L;
                alpha = 1;

                A = factors[cur_mode];
                Y = factors[cur_mode]; // layer_factor

                int last_mode = (cur_mode == static_cast<int>(_TnsSize) - 1) ?
static_cast<int>(_TnsSize) - 2 : static_cast<int>(_TnsSize) - 1;

                Matrix temp_1_R(1, r);
                while (1)
                {
                        grad_Y.setZero();

                        if (iter >= max_nest_iter)
                        {
                                break;
                        }

                        // Compute grad_Y
                        Matrix temp_row = Matrix::Zero(1, r);
                        for (long int i = 0; i < tns_spMat.outerSize(); ++i)
                        {
                                temp_row.setZero();
                                for (SparseMatrix::InnerIterator it(tns_spMat, i); it; ++it)
                                {
                                        temp_1_R = Matrix::Ones(1, r);
                                        // Select rows of each factor an compute the respective
row of the Khatri-Rao product.
                                        for (int mode_i = last_mode, kr_counter =
static_cast<int>(_TnsSize) - 2; mode_i >= 0 && kr_counter >= 0; mode_i--)
                                        {
                                                if (mode_i == cur_mode)
                                                {
                                                        continue;
                                                }
                                                int row;
                                                row      = ((it.row()) / offsets[kr_counter]) %
(tns_dimensions[mode_i]);
```

```
                                                        temp_1_R = temp_1_R.cwiseProduct(factors[mode_i].row(row));
                                                        kr_counter--;
                                                }
                                                // Computation of row of Z according the relation (10) of the paper.
                                                temp_row += (Y.row(i) * temp_1_R.transpose()) * temp_1_R;
                                        }
                                        grad_Y.row(i) = temp_row;
                                }

                                // Add proximal term.
                                grad_Y += MTTKRP + lambda * Y;
                                if (constraint_i == Constraint::unconstrained)
                                {
                                        new_A = (Y - grad_Y / L);
                                }
                                else // Use projection
                                {
                                        new_A = (Y - grad_Y / L).cwiseMax(Zero_Matrix);
                                }

                                new_alpha = UpdateAlpha(alpha, q);
                                beta      = alpha * (1 - alpha) / (alpha * alpha + new_alpha);

                                Y = (1 + beta) * new_A - beta * A;

                                // Update Y

                                A     = new_A;
                                alpha = new_alpha;
                                iter++;
                        }
                        factors[cur_mode] = A;
                }
        } // end of namespace std_V


        namespace local_L
        {
                // GTC Serial
                // transposed_v
                template<std::size_t _TnsSize>
                void NesterovMNLS(std::array<Matrix, _TnsSize>       &factors,
                                                   std::array<int, _TnsSize>     const &tns_dimensions,
                                                   SparseMatrix                  const &tns_spMat,
                          const std::array<int, _TnsSize-1>    &offsets,
                                                   int     const                       max_nest_iter,
                                                   double  const                       lambda,
                                                   int     const                       cur_mode,
                                                   Matrix                             &MTTKRP_T)
                {
                        int m = factors[cur_mode].cols();
                        int r = factors[cur_mode].rows();

                        Matrix inv_L2(tns_spMat.outerSize(),1);

                        double sqrt_q = 0, beta = 0;
                        double L2;
                        int iter = 0;
                        Matrix Y(r, m);
                        Matrix A(r, m);

                        const Matrix zero_vec    = Matrix::Zero(r, 1);
                        Matrix       new_A_vec   = Matrix::Zero(r, 1);

                        A = factors[cur_mode];
                        Y = factors[cur_mode];

                        int last_mode = (cur_mode == static_cast<int>(_TnsSize) - 1) ? static_cast<int>(_TnsSize) - 2 : static_cast<int>(_TnsSize) - 1;

                        Matrix temp_R_1 = Matrix::Ones(r, 1);
                        Matrix temp_RxR(r, r);

                        while (1)
                        {
                                if (iter >= max_nest_iter)
                                {
                                        break;
                                }

                                // Compute grad_Y
                                Matrix temp_col = Matrix::Zero(r, 1);
                                for (long int i = 0; i < tns_spMat.outerSize(); ++i)
                                {
```

```
                                        temp_col.setZero();

                                        if (iter < 1)
                                        {
                                                temp_RxR.setZero();
                                        }

                                        for (SparseMatrix::InnerIterator it(tns_spMat, i); it; ++it)
                                        {
                                                temp_R_1 = Matrix::Ones(r, 1);
                                                // Select rows of each factor an compute the respective row of the Khatri-Rao product.
                                                for (int mode_i = last_mode, kr_counter = static_cast<int>(_TnsSize) - 2; mode_i >= 0 && kr_counter >= 0; mode_i--)
                                                {
                                                        if (mode_i == cur_mode)
                                                        {
                                                                continue;
                                                        }
                                                        long long int row  = ((it.row()) / offsets[kr_counter]) % (tns_dimensions[mode_i]);
                                                        temp_R_1           = temp_R_1.cwiseProduct(factors[mode_i].col(row));
                                                        kr_counter--;
                                                }
                                                // Computation of row of Z according the relation (10) of the paper.
                                                temp_col.noalias() += (temp_R_1.transpose() * Y.col(i))(0) * temp_R_1;
                                                // Estimate Hessian for each row.
                                                if (iter < 1)
                                                {
                                                        temp_RxR.noalias() += (temp_R_1 * temp_R_1.transpose());
                                                }
                                        }
                                        temp_col.noalias() += MTTKRP_T.col(i) + (lambda * Y.col(i));

                                        if (iter < 1)
                                        {
                                                L2 = PowerMethod(temp_RxR, 1e-3);
                                                L2 += lambda;
                                                inv_L2(i) = 1 / L2;
                                        }

                                        new_A_vec = (Y.col(i) - inv_L2(i) * temp_col).cwiseMax(zero_vec);

                                        sqrt_q = sqrt( lambda * inv_L2(i) );

                                        beta = (1 - sqrt_q) / (1 + sqrt_q);

                                        // Update Y
                                        Y.col(i) = (1 + beta) * new_A_vec - beta * factors[cur_mode].col(i);

                                        // Update i-th column of current factor
                                        factors[cur_mode].col(i) = new_A_vec;
                                }
                                iter++;

                        }

                }



                // OpenMP
                template<std::size_t _TnsSize>
                void NesterovMNLS(Matrix                                 &inv_L2,
                                  std::array<Matrix, _TnsSize>        &factors,
                                  std::array<int, _TnsSize>     const &tns_dimensions,
                                  SparseMatrix                  const &tns_spMat,
                        const std::array<int, _TnsSize-1>    &offsets,
                                  Matrix                              &Y,
                                  int    const                         max_nest_iter,
                                  double const                         lambda,
                                  int    const                         cur_mode,
                                  Matrix                              &MTTKRP_T)
                {
                        int r = factors[cur_mode].rows();
                        double sqrt_q = 0, beta = 0;
                        double L2;
                        int iter = 0;
                        long int row;

                        Matrix new_A_vec = Matrix::Zero(r, 1);
```

```
Matrix zero_vec  = Matrix::Zero(r, 1);

#pragma omp master
Y = factors[cur_mode];
#pragma omp barrier

int last_mode = (cur_mode == static_cast<int>(_TnsSize) - 1) ? static_cast<int>(_TnsSize) - 2 : static_cast<int>(_TnsSize) - 1;

Matrix temp_R_1 = Matrix::Ones(r, 1);
Matrix temp_RxR(r, r);
Matrix temp_col(r, 1);

#pragma omp barrier
while (1)
{
        if (iter >= max_nest_iter)
        {
                break;
        }

        // Compute grad_Y
        #pragma omp for schedule(dynamic) nowait
        for (long int i = 0; i < tns_spMat.outerSize(); ++i)
        {
                temp_col.setZero();
                if (iter < 1)
                {
                        temp_RxR.setZero();
                }
                for (SparseMatrix::InnerIterator it(tns_spMat, i); it; ++it)
                {
                        temp_R_1 = Matrix::Ones(r, 1);
                        // Select rows of each factor an compute the respective row of the Khatri-Rao product.
                        for (int mode_i = last_mode, kr_counter = static_cast<int>(_TnsSize) - 2; mode_i >= 0 && kr_counter >= 0; mode_i--)
                        {
                                if (mode_i == cur_mode)
                                {
                                        continue;
                                }
                                row      = ((it.row()) / offsets[kr_counter]) % (tns_dimensions[mode_i]);
                                temp_R_1 = temp_R_1.cwiseProduct(factors[mode_i].col(row));
                                kr_counter--;
                        }
                        // Computation of row of Z according the relation (10) of the paper.
                        temp_col.noalias() += (temp_R_1.transpose() * Y.col(i))(0) * temp_R_1;

                        // Estimate Hessian for each row.
                        // Compute only once!
                        if (iter < 1)
                        {
                                temp_RxR.noalias() += (temp_R_1 * temp_R_1.transpose());
                        }
                }

                temp_col.noalias() += MTTKRP_T.col(i) + (lambda * Y.col(i));

                if (iter < 1)
                {
                        L2 = PowerMethod(temp_RxR, 1e-3);
                        L2 += lambda;
                        inv_L2(i) = 1 / L2;
                }

                new_A_vec = (Y.col(i) - inv_L2(i) * temp_col).cwiseMax(zero_vec);

                sqrt_q = sqrt( lambda * inv_L2(i) );

                beta = (1 - sqrt_q) / (1 + sqrt_q);

                // Update Y
                Y.col(i) = (1 + beta) * new_A_vec - beta * factors[cur_mode].col(i);

                // Update i-th column of current factor
                factors[cur_mode].col(i) = new_A_vec;

        }
        iter++;
```

```
                        }
                        #pragma omp barrier
                }

        } // end of namespace local_L

} // end namespace partensor

#endif // end of PARTENSOR_NESTEROV_MNLS_HPP
```

8.45 Normalize.hpp File Reference

```
#include <math.h>
#include "PARTENSOR_basic.hpp"
#include "DimTrees.hpp"
#include "TensorOperations.hpp"
#include "Eigen/Core"
```

Functions

- template< typename Status >
  void choose_normilization_factor (Status const &st, bool &all_orthogonal=true, int &weight_factor=0)
- template< std::size_t _TnsSize, typename DimensionType >
  void Normalize (int const weight_factor, int const R, DimensionType const &tnsDims, std::array< Factor↩ DimTree, _TnsSize > &factors)

  Factors normalization.
- template< std::size_t _TnsSize>
  void Normalize (int const weight_factor, int const R, std::array< Matrix, _TnsSize > &gramian, std::array< Matrix, _TnsSize > &factors)

  Factors normalization.

8.45.1 Detailed Description

Implementations for the normalization of the factors. The factors can be either `Eigen` Matrix or `FactorDimTree`.

8.45.2 Function Documentation

8.45.2.1 choose_normilization_factor()

```
void partensor::v1::choose_normilization_factor (
            Status const & st,
            bool & all_orthogonal = true,
            int & weight_factor = 0 )
```

Checks if all factors have orthogonal constraint, in order to avoid normalization after each factor update. After that if a non-orthogonal constraint being applied on a factor, then the `the` id of this factor is returned, to be used for normalization function.

Template Parameters

Status	Status class based on `cpd` algorithm chosen.

Parameters

st	[in] Reference to the `Status` class. Used in order to extract the the factors' `constraints` array.
all_orthogonal	[in,out] If all factors have `orthogonal` constraint then the algorithm will not normalize factors after each update. Otherwise, uses `weight_factor` factor to load the weights of the other factors.
weight_factor	[in,out] The first factor that has no orthogonal constraint.

8.45.2.2 Normalize() [1/2]

```
void partensor::v1::Normalize (
            int const weight_factor,
            int const R,
            DimensionType const & tnsDims,
            std::array< FactorDimTree, _TnsSize > & factors )
```

Factors normalization.

Normalizes the columns of all factors based on the last factor from the `stl` array `factors`.

Template Parameters

_TnsSize	Size of the `factors` array.
DimensionType	Array container for `tnsDims`.

Parameters

weight_factor	[in] The facotr that the weights of the other factors will be loaded.
R	[in] Rank of factorization. (Number of columns in each `FactorDimTree`).
tnsDims	[in] The row dimension for each factor.
factors	[in,out] An `stl` array containing all factors of type `FactorDimTree`.

Note

This implementation ONLY, if factors are of `FactorDimTree` type.

NO `orthogonal` constraints must be applied on the `weight_factor` factor.

8.45.2.3 Normalize() [2/2]

```
void partensor::v1::Normalize (
            int const weight_factor,
            int const R,
            std::array< Matrix, _TnsSize > & gramian,
            std::array< Matrix, _TnsSize > & factors )
```

Factors normalization.

Normalizes the columns of all factors based on the last factor from the `stl` array `factors`.

Template Parameters

_TnsSize	Size of the `factors` and `gramian` arrays.

Parameters

weight_factor	[in] The facotr that the weights of the other factors will be loaded.
R	[in] Rank of factorization. (Number of columns in each `Matrix`).
gramian	[in,out] Quantity of `factor^T * factor`.
factors	[in,out] An `stl` array containing all `Matrix` factors.

Note

This implementation ONLY, if factors are of `Matrix` type.

NO `orthogonal` constraints must be applied on the `weight_factor` factor.

8.46 Normalize.hpp

Go to the documentation of this file.

```
#ifndef DOXYGEN_SHOULD_SKIP_THIS
#endif  // DOXYGEN_SHOULD_SKIP_THIS
/**********************************************************************/
#ifndef NORMALIZE_HPP
#define NORMALIZE_HPP

#include <math.h>
#include "PARTENSOR_basic.hpp"
#include "DimTrees.hpp"
#include "TensorOperations.hpp"
#include "Eigen/Core"

namespace partensor
{

  inline namespace v1 {

    template<typename Status>
    void choose_normilization_factor(Status const &st,
                                     bool &all_orthogonal=true,
                                     int &weight_factor=0)
    {
      for(std::size_t i=0; i<st.options.constraints.size(); ++i)
      {
        if(st.options.constraints[i] != Constraint::orthogonality)
        {
          all_orthogonal = false;
          weight_factor  = i;
          break;
        }
      }
```

```
}

template<std::size_t _TnsSize>
void Normalize( int                              const  weight_factor,
                int                              const  R,
                std::array<Matrix,_TnsSize>       &gramian,
                std::array<Matrix,_TnsSize>       &factors)
{
  using Vector = Eigen::VectorXd;

   constexpr std::size_t         lastFactor  = _TnsSize - 1;

   int                           pass_flag   = 0;
   double                        cumul_power = 1;
   bool                          nonZeroFlag = true;

   std::array<Matrix,_TnsSize>   normMatrixList;

   std::array<Vector,lastFactor> lambda_fac;
   std::array<double,lastFactor> norm_factor;

   for(int i=0; i<static_cast<int>(_TnsSize); ++i)
   {
     if(i != weight_factor)
     {
       norm_factor[i-pass_flag] = 1;
       lambda_fac[i-pass_flag]  = Vector(static_cast<int>(R));
       lambda_fac[i-pass_flag]  = gramian[i].diagonal();
       normMatrixList[i]        = Matrix::Zero(static_cast<int>(R),static_cast<int>(R));
     }
     else
     {
       normMatrixList[i] = Matrix::Ones(static_cast<int>(R),static_cast<int>(R));
       pass_flag         = 1;
     }
   }

   for(int i=0; i<static_cast<int>(R); ++i)
   {
     for(int j=0; j<static_cast<int>(lastFactor); ++j)
     {
       norm_factor[j] = sqrt((lambda_fac[j])(i));
       nonZeroFlag    = !(norm_factor[j] == 0);
     }

     if(!nonZeroFlag)
     {
         for(int j=0; j<static_cast<int>(lastFactor); ++j)
           (lambda_fac[j])(i) = 1;
     }
     else
     {
         pass_flag = 0;
         for(int j=0; j<static_cast<int>(lastFactor); ++j)
         {
             if(j==weight_factor)
             {
               pass_flag = 1;
               continue;
             }
             factors[j].col(i)            *= 1/norm_factor[j-pass_flag];
             cumul_power                  *= norm_factor[j-pass_flag];
             (lambda_fac[j-pass_flag])(i) = norm_factor[j-pass_flag];
         }
         factors[weight_factor].col(i)    *= cumul_power;
         cumul_power                       = 1;
     }
   }

   pass_flag = 0;
   for(int i=0; i<static_cast<int>(lastFactor); ++i)
   {
       if(i==weight_factor)
       {
         pass_flag = 1;
         continue;
       }
       normMatrixList[i].noalias()    = lambda_fac[i-pass_flag] * lambda_fac[i-pass_flag].transpose();
       normMatrixList[weight_factor] = normMatrixList[weight_factor].cwiseProduct(normMatrixList[i]);
       gramian[i]                    = gramian[i].cwiseQuotient(normMatrixList[i]);
   }

   gramian[weight_factor] = gramian[weight_factor].cwiseProduct(normMatrixList[weight_factor]);

 }

 template<std::size_t _TnsSize, typename DimensionType>
```

```
void Normalize( int                                const  weight_factor,
                int                                const  R,
                DimensionType                      const &tnsDims,
                std::array<FactorDimTree,_TnsSize>       &factors)
{
  using MatrixArray    = std::array<Matrix,_TnsSize>;
  using FactorIterator = typename std::array<FactorDimTree,_TnsSize>::iterator;
  using Vector         = Eigen::VectorXd;

  constexpr std::size_t         lastFactor  = _TnsSize - 1;

  int                           pass_flag   = 0;
  double                        cumul_power = 1;
  bool                          nonZeroFlag = true;

  MatrixArray                   factorsList;
  MatrixArray                   gramMatrixList;
  MatrixArray                   normMatrixList;

  std::array<Vector,lastFactor> lambda_fac;
  std::array<double,lastFactor> norm_factor;

  FactorIterator factorPtr = factors.begin();

  for(int i=0; i<static_cast<int>(_TnsSize); ++i)
  {
    factorsList[i]    = Matrix(tnsDims[i],R);
    factorsList[i]    = tensorToMatrix(factorPtr->factor,tnsDims[i],static_cast<int>(R));

    gramMatrixList[i] = Matrix(R,R);
    gramMatrixList[i] = tensorToMatrix(factorPtr->gramian,static_cast<int>(R),static_cast<int>(R));

    if(i != weight_factor)
    {
        norm_factor[i-pass_flag] = 1;
        lambda_fac[i-pass_flag]  = Vector(static_cast<int>(R));
        lambda_fac[i-pass_flag]  = gramMatrixList[i].diagonal();
        normMatrixList[i]        = Matrix::Zero(static_cast<int>(R),static_cast<int>(R));
    }
    else
    {
        normMatrixList[i] = Matrix::Ones(static_cast<int>(R),static_cast<int>(R));
        pass_flag         = 1;
    }
    factorPtr++;
  }

  for(int i=0; i<static_cast<int>(R); ++i)
  {
    for(int j=0; j<static_cast<int>(lastFactor); ++j)
    {
        norm_factor[j] = sqrt((lambda_fac[j])(i));
        nonZeroFlag    = !(norm_factor[j] == 0);
    }

    if(!nonZeroFlag)
    {
        for(int j=0; j<static_cast<int>(lastFactor); ++j)
          (lambda_fac[j])(i) = 1;
    }
    else
    {
        pass_flag = 0;
        for(int j=0; j<static_cast<int>(lastFactor); ++j)
        {
            if(j==weight_factor)
            {
              pass_flag = 1;
              continue;
            }
            factorsList[j].col(i)         *= 1/norm_factor[j-pass_flag];
            cumul_power                   *= norm_factor[j-pass_flag];
            (lambda_fac[j-pass_flag])(i)   = norm_factor[j-pass_flag];
        }
        factorsList[weight_factor].col(i) *= cumul_power;
        cumul_power                        = 1;
    }
  }

  pass_flag = 0;
  for(int i=0; i<static_cast<int>(lastFactor); ++i)
  {
    if(i==weight_factor)
    {
      pass_flag = 1;
      continue;
    }
```

```
          normMatrixList[i].noalias()    = lambda_fac[i-pass_flag] * lambda_fac[i-pass_flag].transpose();
          normMatrixList[weight_factor] = normMatrixList[weight_factor].cwiseProduct(normMatrixList[i]);
          gramMatrixList[i]             = gramMatrixList[i].cwiseQuotient(normMatrixList[i]);
        }

        gramMatrixList[weight_factor] =
    gramMatrixList[weight_factor].cwiseProduct(normMatrixList[weight_factor]);
        factorPtr                     = factors.begin();

        for(int i=0; i<static_cast<int>(_TnsSize); ++i)
        {
            // Fill with the normalized factors.
            factorPtr->factor  = matrixToTensor(factorsList[i], tnsDims[i], static_cast<int>(R));
            factorPtr->gramian = matrixToTensor(gramMatrixList[i], static_cast<int>(R),
    static_cast<int>(R));
            factorPtr++;
        }
      }

    } // end namespace v1

  } // end namespace partensor

  #endif // end of NORMALIZE_HPP
```

8.47 ParallelWrapper.hpp File Reference

```
#include <vector>
#include "boost/mpi/communicator.hpp"
#include "boost/mpi/collectives.hpp"
#include "boost/mpi/environment.hpp"
#include "boost/mpi/cartesian_communicator.hpp"
```

Classes

- struct cartesian_communicator

  An MPI communicator with a cartesian topology.
- struct cartesian_dimension

  Specify the size and periodicity of the grid in a single dimension.
- struct cartesian_topology

  Describe the topology of a cartesian grid.
- struct communicator

  A communicator that permits communication and synchronization among a set of processes.
- struct environment

  Initialize, finalize, and query the `MPI` environment.

Typedefs

- using Boost_CartCommunicator = boost::mpi::cartesian_communicator
- using Boost_CartDimension = boost::mpi::cartesian_dimension
- using Boost_CartTopology = boost::mpi::cartesian_topology
- using Boost_Communicator = boost::mpi::communicator
- using Boost_Environment = boost::mpi::environment
- template< typename T >

  using **inplace_t** = typename boost::mpi::inplace_t< T >

  Wrapper type to explicitly indicate that a input data can be overriden with an output value.
- template< typename T >

  using **maximum** = typename boost::mpi::maximum< T >

  Compute the maximum of two values.
- template< typename T >

  using **minimum** = typename boost::mpi::minimum< T >

  Compute the minimum of two values.

Functions

- template< typename T , typename Op >
  void all_reduce (const cartesian_communicator &comm, inplace_t< T * > value, int n, Op op)

  Combine the values stored by each process into a single value available to all processes.
- template<std::size_t _TnsSize>
  void create_fiber_grid (cartesian_communicator const &grid, std::vector< cartesian_communicator > &fiber_comm, std::array< int, _TnsSize > &fiber_rank)
- template<std::size_t _TnsSize>
  void create_layer_grid (cartesian_communicator &grid, std::vector< cartesian_communicator > &layer_↩ comm, std::array< int, _TnsSize > &layer_rank)
- void DisCount (std::vector< int > &dis, std::vector< int > &count, int const size, int const dim, std::size_t const rank)
- template<typename T >
  inplace_t< T * > inplace (T *inout)

  Wrap an input data to indicate that it can be overriden with an ouput value.

8.47.1 Detailed Description

Implements wrapper functions from `Boost mpi` library and `OpenMPI`, necessary for this project.

8.47.2 Typedef Documentation

8.47.2.1 Boost_CartCommunicator

```
using Boost_CartCommunicator = boost::mpi::cartesian_communicator
```

Typdef for `cartesian_communicator` class from boost.

8.47.2.2 Boost_CartDimension

```
using Boost_CartDimension = boost::mpi::cartesian_dimension
```

Typdef for `cartesian_dimension` class from boost.

8.47.2.3 Boost_CartTopology

```
using Boost_CartTopology = boost::mpi::cartesian_topology
```

Typdef for `cartesian_topology` class from boost.

8.47.2.4 Boost_Communicator

```
using Boost_Communicator = boost::mpi::communicator
```

Typdef for `communicator` class from boost.

8.47.2.5 Boost_Environment

```
using Boost_Environment = boost::mpi::environment
```

Typdef for `environment` class from boost.

8.47.3 Function Documentation

8.47.3.1 all_reduce()

```
void partensor::v1::all_reduce (
            const cartesian_communicator & comm,
            inplace_t< T * > value,
            int n,
            Op op ) [inline]
```

Combine the values stored by each process into a single value available to all processes.

`all_reduce` is a collective algorithm that combines the values stored by each process into a single value available to all processes. The values are combined in a user-defined way, specified via a function object. The type `T` of the values may be any type that is serializable or has an associated MPI data type.

When the type `T` has an associated MPI data type, this routine invokes `MPI_Allreduce` to perform the reduction. If possible, built-in MPI operations will be used; otherwise, `all_reduce()` will create a custom MPI_Op for the call to MPI_Allreduce.

Parameters

comm	[in] The communicator over which the reduction will occur.
value	[in] The local value to be combined with the local values of every other process. For reducing arrays, `in_values` is a pointer to the local values to be reduced and `n` is the number of values to reduce. See `reduce` for more information.

If wrapped in a `inplace_t` object, combine the usage of both input and $c out_value and the local value will be overwritten (a convenience function `inplace` is provided for the wrapping).

Parameters

out_value	[in,out] Will receive the result of the reduction operation. If this parameter is omitted, the outgoing value will instead be returned.
n	[in] Indicated the size of the buffers of array type.

Returns

If no `out_value` parameter is supplied, returns the result of the reduction operation.

Parameters

op	[in] The binary operation that combines two values of type `T` and returns a third value of type `T`.

8.47.3.2 create_fiber_grid()

```
void partensor::v1::create_fiber_grid (
            cartesian_communicator const & grid,
            std::vector< cartesian_communicator > & fiber_comm,
            std::array< int, _TnsSize > & fiber_rank )
```

Creates a fiber grid in a `cartesian_communicator`.

Template Parameters

_TnsSize	Order of the Tensor.

Parameters

grid	[in] The MPI_COMM_WORLD implemented in a `cartesian communicator`.
fiber_comm	[in,out] An `stl` vector containing the newly created layer communicator, that still belong in `grid`.
fiber_rank	[in,out] An `stl` array containing the ranks of each processor in each `fiber_comm`.

8.47.3.3 create_layer_grid()

```
void partensor::v1::create_layer_grid (
            cartesian_communicator & grid,
            std::vector< cartesian_communicator > & layer_comm,
            std::array< int, _TnsSize > & layer_rank )
```

Creates a layer grid in a `cartesian_communicator`.

Template Parameters

_TnsSize	Order of the Tensor.

Parameters

grid	[in] The MPI_COMM_WORLD implemented in a `cartesian communicator`.
layer_comm	[in,out] An `stl` vector containing the newly created layer communicator, that still belong in `grid`.
layer_rank	[in,out] An `stl` array containing the ranks of each processor in each `layer_comm`.

8.47.3.4 DisCount()

```
void partensor::v1::DisCount (
            std::vector< int > & dis,
            std::vector< int > & count,
            int const size,
            int const dim,
            std::size_t const rank )
```

Computes two arrays (`dis`, `count`) with the number of "lines" from Tensor to skip and how many to read, based on tensor dimensions `dim`, number of processors `size` and tensor `rank`.

Parameters

dis	[in,out] The number of "lines" to skip per processor.
count	[in,out] The number of "lines" to read per processor.
size	[in] Number of processors.
dim	[in] Tensor dimensions.
rank	[in] Tensor rank.

8.47.3.5 inplace()

```
inplace_t< T * > partensor::v1::inplace (
            T * inout )
```

Wrap an input data to indicate that it can be overriden with an ouput value.

Parameters

inout	the contributing input value, it will be overriden with the output value where one is expected. If it is a pointer, the number of elements will be provided separately.

Returns

The wrapped value or pointer.

8.48 ParallelWrapper.hpp

Go to the documentation of this file.

```
#ifndef DOXYGEN_SHOULD_SKIP_THIS
#endif // DOXYGEN_SHOULD_SKIP_THIS
/*********************************************************************/
#ifndef PARTENSOR_PARALLEL_WRAPPER_HPP
#define PARTENSOR_PARALLEL_WRAPPER_HPP

#include <vector>
#include "boost/mpi/communicator.hpp"
#include "boost/mpi/collectives.hpp"
#include "boost/mpi/environment.hpp"
#include "boost/mpi/cartesian_communicator.hpp"

namespace partensor {
```

```cpp

inline namespace v1 {
  using Boost_Environment       = boost::mpi::environment;
  using Boost_Communicator      = boost::mpi::communicator;
  using Boost_CartDimension     = boost::mpi::cartesian_dimension;
  using Boost_CartTopology      = boost::mpi::cartesian_topology;
  using Boost_CartCommunicator = boost::mpi::cartesian_communicator;
  struct environment : public Boost_Environment
  {
    environment(int &argc, char** &argv, bool abort_on_exception = true) : Boost_Environment(argc, argv, abort_on_exception) {};

    ~environment() = default;

    using Boost_Environment::abort;
  };

   struct communicator : public Boost_Communicator
   {
     communicator() : Boost_Communicator() { }

     using Boost_Communicator::rank;
     using Boost_Communicator::size;
     using Boost_Communicator::barrier;
   };

   struct cartesian_dimension : public Boost_CartDimension
   {
     cartesian_dimension(int sz = 0, bool p = true) : Boost_CartDimension(sz,p) {}
   };

   struct cartesian_topology : public Boost_CartTopology
   {
     template<class InitArr_>
     cartesian_topology(InitArr_ dims) : Boost_CartTopology(dims) {};
   };

   struct cartesian_communicator : public Boost_CartCommunicator
   {
     friend struct communicator;
     using communicator::rank;

     cartesian_communicator( communicator       const &comm,
                             cartesian_topology const &dims,
                             bool               reorder = true) : Boost_CartCommunicator(comm, cartesian_topology(dims), reorder) {}

     cartesian_communicator( cartesian_communicator const &comm,
                             std::vector<int>       const &keep) : Boost_CartCommunicator(comm, keep) {}
   };

   template<typename T>
   using inplace_t = typename boost::mpi::inplace_t<T>;

   // template<typename T>
   // inplace_t<T> inplace(T& inout) {
   //   return inplace_t<T>(inout);
   // }

   template<typename T>
   inplace_t<T*> inplace(T* inout) {
     return inplace_t<T*>(inout);
   }

   template<typename T>
   using maximum = typename boost::mpi::maximum<T>;

   template<typename T>
   using minimum = typename boost::mpi::minimum<T>;

   template<typename T, typename Op>
   inline void all_reduce( const cartesian_communicator &comm,
                           inplace_t<T*>                  value,
                           int                            n,
                           Op                             op    )
   {
     boost::mpi::all_reduce(static_cast<const Boost_CartCommunicator &>(comm), value, n, op);
   }

   template<typename T, typename Op>
   inline void all_reduce( const cartesian_communicator &comm,
                           T                             &value,
                           T                             &out_value,
                           Op                             op        )
   {
     boost::mpi::all_reduce(static_cast<const Boost_CartCommunicator &>(comm), value, out_value, op);
```

```
  }

  template<typename T, typename Op>
  inline void all_reduce( const cartesian_communicator &comm,
                          const T                       *value,
                          int                            n,
                          T                             *out_value,
                          Op                             op)
  {
    boost::mpi::all_reduce(static_cast<const Boost_CartCommunicator &>(comm), value, n, out_value, op);
  }

  /*
   * @brief Combine the values stored by each process into a single
   * value at the root.
   *
   * @c reduce is a collective algorithm that combines the values
   * stored by each process into a single value at the @c root. The
   * values can be combined arbitrarily, specified via a function
   * object. The type @c T of the values may be any type that is
   * serializable or has an associated MPI data type. One can think of
   * this operation as a @c gather to the @p root, followed by an @c
   * std::accumulate() over the gathered values and using the operation
   * @c op.
   *
   * When the type @c T has an associated MPI data type, this routine
   * invokes @c MPI_Reduce to perform the reduction. If possible,
   * built-in MPI operations will be used; otherwise, @c reduce() will
   * create a custom MPI_Op for the call to MPI_Reduce.
   *
   *   @param comm [in] The communicator over which the reduction will
   *   occur.
   *
   *   @param in_values [in] The local value to be combined with the local
   *   values of every other process. For reducing arrays, @c in_values
   *   contains a pointer to the local values. In this case, @c n is
   *   the number of values that will be reduced. Reduction occurs
   *   independently for each of the @p n values referenced by @p
   *   in_values, e.g., calling reduce on an array of @p n values is
   *   like calling @c reduce @p n separate times, one for each
   *   location in @p in_values and @p out_values.
   *
   *   @param out_values [in,out] Will receive the result of the reduction
   *   operation, but only for the @p root process. Non-root processes
   *   may omit if parameter; if they choose to supply the parameter,
   *   it will be unchanged. For reducing arrays, @c out_values
   *   contains a pointer to the storage for the output values.
   *
   *   @param op [in] The binary operation that combines two values of type
   *   @c T into a third value of type @c T. For types @c T that has
   *   associated MPI data types, @c op will either be translated into
   *   an @c MPI_Op (via @c MPI_Op_create) or, if possible, mapped
   *   directly to a built-in MPI operation. See @c is_mpi_op in the @c
   *   operations.hpp header for more details on this mapping. For any
   *   non-built-in operation, commutativity will be determined by the
   *   @c is_commmutative trait (also in @c operations.hpp): users are
   *   encouraged to mark commutative operations as such, because it
   *   gives the implementation additional latitude to optimize the
   *   reduction operation.
   *
   *   @param root [in] The process ID number that will receive the final,
   *   combined value. This value must be the same on all processes.
   */
  template<typename T, typename Op>
  void reduce(const cartesian_communicator &comm,
              const T*                      in_values,
              int                           n,
              T*                            out_values,
              Op                            op,
              int                           root      )
  {
    boost::mpi::reduce(static_cast<const Boost_CartCommunicator &>(comm), in_values, n, out_values, op, root );
  }

  /*
   * @brief Similar to boost::mpi::scatter with the difference that the number
   * of values stored at the root process does not need to be a multiple of
   * the communicator's size.
   *
   *   @param comm [in] The communicator over which the scatter will occur.
   *
   *   @param in_values [in] A vector or pointer to storage that will contain
   *   the values to send to each process, indexed by the process rank.
   *   For non-root processes, this parameter may be omitted. If it is
   *   still provided, however, it will be unchanged.
```

```
     *
     *    @param sizes [in] A vector containing the number of elements each non-root
     *    process will receive.
     *
     *    @param out_values [in,out] The array of values received by each process.
     *
     *    @param root [in] The process ID number that will scatter the
     *    values. This value must be the same on all processes.
     */
    template<typename T>
    void scatterv(const cartesian_communicator &comm,
                  const T*                     in_values,
                  const std::vector<int>      &sizes,
                  T*                           out_values,
                  int                          root      )
    {
      boost::mpi::scatterv(static_cast<const Boost_CartCommunicator &>(comm), in_values, sizes,
   out_values, root);
    }

    /*
     *  @brief Gather the values stored at every process into a vector of
     *  values from each process.
     *
     *  @c all_gatherv is a collective algorithm that collects the values
     *  stored at each process into a vector of values at each
     *  process. This vector is indexed by the process number that the
     *  value came from. The type @c T of the values may be any type that
     *  is serializable or has an associated MPI data type.
     *
     *    @param comm The communicator over which the gather will occur.
     *
     *    @param in_values The array of values to be transmitted by each process.
     *
     *    @param in_size For each process this specifies the size of @p in_values.
     *
     *    @param out_values A pointer to storage that will be populated with
     *    the values from each process.
     *
     *    @param sizes A vector containing the number of elements each
     *    process will send.
     *
     *    @param displs A vector such that the i-th entry specifies the
     *    displacement (relative to @p out_values) from which to take the ingoing
     *    data at the @p root process. Overloaded versions for which @p displs is
     *    omitted assume that the data is to be placed contiguously at each process.
     *
     */
    template<typename T>
    void all_gatherv( const cartesian_communicator &comm,
                      const T*                      in_values,
                      int                           in_size,
                      T*                            out_values,
                      const std::vector<int>       &sizes,
                      const std::vector<int>       &displs    )
    {
      for(int layerRank = 0; layerRank<comm.size(); layerRank++)
        boost::mpi::gatherv(static_cast<const Boost_CartCommunicator &>(comm), in_values, in_size,
   out_values, sizes, displs, layerRank);
    }

    template<std::size_t _TnsSize>
    void create_layer_grid( cartesian_communicator              &grid,
                            std::vector<cartesian_communicator> &layer_comm,
                            std::array<int, _TnsSize>           &layer_rank )
    {
        std::vector<int>          layer_dims(_TnsSize-1);
        std::array<int, _TnsSize> free_coords;

        for (std::size_t i = 0; i < _TnsSize; ++i)
        {
            std::fill(free_coords.begin(), free_coords.end(), 1);
            free_coords[i] = 0;
            int pos = 0;
            for (std::size_t j = 0; j < _TnsSize; ++j) {
                if (free_coords[j]) {
                    layer_dims[pos++] = free_coords[j] * j;
                }
            }
            // create the sub communicator in the cartesian communicator for layers
            layer_comm.push_back(cartesian_communicator(grid, layer_dims));
            // ID for each processor in layers sub communicator
            layer_rank[i] = layer_comm[i].rank();
        }
    }

    template<std::size_t _TnsSize>
```

```
    void create_fiber_grid( cartesian_communicator const        &grid,
                            std::vector<cartesian_communicator> &fiber_comm,
                            std::array<int, _TnsSize>           &fiber_rank )
    {
        std::vector<int> fiber_dims(1);

        for (std::size_t i = 0; i < _TnsSize; ++i)
        {
            fiber_dims[0] = i;
            // create the sub communicator in the cartesian communicator for fibers
            fiber_comm.push_back(cartesian_communicator(grid, fiber_dims));
            // ID for each processor in fibers sub communicator
            fiber_rank[i] = fiber_comm[i].rank();
        }
    }

    void DisCount(int *dis, int *count, int const size, int const dim, std::size_t const rank)
    {
      int x = dim / size;
      int y = dim % size;
      for (int i=0; i<size; i++)
      {
        count[i] = (i >= y) ? x*rank : (x+1)*rank;
        dis[i]   = 0;

        for (int j=0; j<i; j++)
          dis[i] += count[j];
      }
    }

    void DisCount(std::vector<int> &dis, std::vector<int> &count, int const size, int const dim,
   std::size_t const rank)
    {
      int x = dim / size;
      int y = dim % size;

      for (int i=0; i<size; i++)
      {
        count.push_back((i >= y) ? x*rank : (x+1)*rank);
        dis.push_back(0);

        for (int j=0; j<i; j++)
          dis[i] += count[j];
      }
    }

  } // end namespace v1

  #ifndef DOXYGEN_SHOULD_SKIP_THIS
  namespace v2 {

    /*
     * Wrapper for MPI_Init, which initializes the MPI execution environment.
     *
     * @param argc [in] Pointer to the number of arguments.
     * @param argv [in] Argument vector.
     */
    void Init(int argc, char **argv)
    {
      MPI_Init(&argc, &argv);
    }

    /*
     * Wrapper for MPI_Comm_rank, which determines the rank of the calling
     * process in the communicator.
     *
     * @param comm [in]     Communicator.
     * @param rank [in,out] Rank of the calling process in group of comm.
     */
    void Comm_Rank( MPI_Comm const &comm,
                    int            &rank   )
    {
      MPI_Comm_rank(comm, &rank);
    }

    /*
     * Wrapper for MPI_Comm_size, which returns the size of the group
     * associated with a communicator.
     *
     * @param comm [in]      Communicator.
     * @param size [int,out] Number of processes in the group of comm.
     */
    void Comm_Size( MPI_Comm const &comm,
                    int            &size   )
    {
      MPI_Comm_size(comm, &size);
    }
```

```
    /*
     * Wrapper for MPI_Abort, which terminates MPI execution environment.
     *
     * @param comm      [in] Communicator.
     * @param errorCode [in] Error code to return to invoking environment.
     */
    void Abort( MPI_Comm const &comm,
                int      const  errorCode )
    {
      MPI_Abort(comm, errorCode);
    }

    template <typename DataType_>
    void all_reduce( MPI_Comm  const &comm,
                     double    const  size,
                     DataType_       &dt    )
    {
      // In case of @c Eigen Matrix
      MPI_Allreduce(MPI_IN_PLACE, dt.data(), size, MPI_DOUBLE, MPI_SUM, comm);
    }

    template <typename DataType_>
    void all_reduce ( MPI_Comm  const &comm,
                      DataType_       &value,
                      DataType_       &out_value,
                      int       const  size )
    {
      MPI_Allreduce(value.data(), out_value.data(), size, MPI_DOUBLE, MPI_SUM, comm);
    }

    template <typename DataType_>
    void gatherv( MPI_Comm  const &comm,
                  DataType_ const &sendBuf,
                  int       const  sendSize,
                  int       const &recvSize,
                  int       const &displs,
                  int       const  root,
                  DataType_       &recvBuf)
    {
      MPI_Gatherv(sendBuf.data(), sendSize, MPI_DOUBLE, recvBuf.data(), &recvSize, &displs, MPI_DOUBLE, root, comm);
    }

    template <typename DataType_>
    void all_gatherv( MPI_Comm  const &comm,
                      DataType_ const &sendBuf,
                      int       const  sendSize,
                      int       const &recvSize,
                      int       const &displs,
                      DataType_       &recvBuf   )
    {
      MPI_Allgatherv(sendBuf.data(), sendSize, MPI_DOUBLE, recvBuf.data(), &recvSize, &displs, MPI_DOUBLE, comm);
    }

    template <typename DataType_>
    void reduce_scatter( MPI_Comm  const &comm,
                         DataType_ const &sendBuf,
                         int       const &recvCounts,
                         DataType_       &recvBuf    )
    {
                        MPI_Reduce_scatter(sendBuf.data(), recvBuf.data(), &recvCounts, MPI_DOUBLE, MPI_SUM, comm);
    }

    /*
     * Wrapper for MPI_Cart_create, which makes a new communicator to which
     * Cartesian topology information has been attached.
     * @tparam _Size            Number of dimensions of Cartesian grid.
     * @param  dims    [in]     Array specifying the number of processes in each dimension.
     * @param  periods [in]     Logical array specifying whether the grid is periodic (1 = true) or not (0 = false) in each dimension.
     * @param  reorder [in]     Ranking may be reordered (1 = true) or not (0 = false).
     * @param  comm    [in,out] Communicator with new Cartesian topology.
     */
    template<std::size_t _Size>
    void Cart_Create( std::array<int,_Size> const &dims,
                      std::array<int,_Size> const &periods,
                      int                   const  reorder,
                      MPI_Comm                    &comm     )
    {
      MPI_Cart_create(MPI_COMM_WORLD, static_cast<int>(_Size), dims.data(), periods.data(), reorder, &comm);
    }

    /*
```

```
      * Wrapper for MPI_Cart_coords, which determines process coords in
      * Cartesian topology given rank in group.
      * @tparam _Size            Number of dimensions of Cartesian grid.
      * @param  comm    [in]     Communicator with Cartesian structure.
      * @param  rank    [in]     Rank of a process within group of comm.
      * @param  coords  [in,out] Array containing the Cartesian coordinates of specified process.
      */
     template<std::size_t _Size>
     void Cart_Coords( MPI_Comm               const &comm,
                       int                    const  rank,
                       std::array<int, _Size>       &coords)
     {
       MPI_Cart_coords(comm, rank, static_cast<int>(_Size), coords.data());
     }

     /*
      * Wrapper for MPI_Cart_sub, which partitions a communicator into subgroups,
      * and form lower-dimensional Cartesian subgrids.
      * @tparam _Size            Number of dimensions of Cartesian grid.
      * @param  comm    [in]     Communicator with Cartesian structure.
      * @param  rDims   [in]     The ith entry of rDims specifies whether the ith dimension is kept in
    the subgrid (1 = true) or is dropped (0 = false).
      * @param  subComm [in,out] Communicator containing the subgrid that includes the calling process.
      */
     template<std::size_t _Size>
     void Cart_Sub( MPI_Comm               const &comm,
                    std::array<int, _Size> const &rDims,
                    MPI_Comm                     &subComm )
     {
       MPI_Cart_sub(comm, rDims.data(), &subComm);
     }

     /*
      * Wrapper for MPI_Barrier, for synchronization between MPI processes
      * @param comm [in] Communicator.
      */
     void Barrier(MPI_Comm const &comm)
     {
       MPI_Barrier(comm);
     }

     /*
      * Wrapper for Comm_Free, which marks a communicator object for deallocation.
      * @param comm [in] Communicator.
      */
     void Comm_Free(MPI_Comm &comm)
     {
       MPI_Comm_free(&comm);
     }

     /*
      * Wrapper for MPI_Finalize, that checks whether MPI has been finalized.
      */
     void Finalize()
     {
       MPI_Finalize();
     }

   } // end namespace v2
   #endif // DOXYGEN_SHOULD_SKIP_THIS

 } // end namespace partensor

 #endif // PARTENSOR_PARALLEL_WRAPPER_HPP
```

8.49 PARTENSOR.hpp File Reference

```
#include "PARTENSOR_basic.hpp"
#include "Constants.hpp"
#include "CwiseProd.hpp"
#include "KhatriRao.hpp"
#include "Kronecker.hpp"
#include "DataGeneration.hpp"
#include "ReadWrite.hpp"
#include "Matricization.hpp"
#include "Tensor.hpp"
#include "TensorOperations.hpp"
```

```
#include "Timers.hpp"
#include "DimTrees.hpp"
#include "Cpd.hpp"
#include "Gtc.hpp"
#include "GtcStochastic.hpp"
#include "CpdDimTree.hpp"
#include "ParallelWrapper.hpp"
```

8.49.1 Detailed Description

Containts all the header files created for `PARTENSOR` project.

8.50 PARTENSOR.hpp

Go to the documentation of this file.

```
#ifndef DOXYGEN_SHOULD_SKIP_THIS
#endif // DOXYGEN_SHOULD_SKIP_THIS
/************************************************************************/
#ifndef PARTENSOR_HPP
#define PARTENSOR_HPP

#include "PARTENSOR_basic.hpp"
#include "Constants.hpp"
#include "CwiseProd.hpp"
#include "KhatriRao.hpp"
#include "Kronecker.hpp"
#include "DataGeneration.hpp"
#include "ReadWrite.hpp"
#include "Matricization.hpp"
#include "Tensor.hpp"
#include "TensorOperations.hpp"
#include "Timers.hpp"
#include "DimTrees.hpp"
#include "Cpd.hpp"
#include "Gtc.hpp"
#include "GtcStochastic.hpp"
#include "CpdDimTree.hpp"
#include "ParallelWrapper.hpp"

#endif // PARTENSOR_HPP
```

8.51 PARTENSOR_basic.hpp File Reference

```
#include "Config.hpp"
#include <mutex>
#include <Eigen/Dense>
#include <Eigen/Sparse>
#include <unsupported/Eigen/CXX11/Tensor>
#include "spdlog/spdlog.h"
#include "spdlog/sinks/basic_file_sink.h"
#include "execution.hpp"
#include "Constants.hpp"
#include "Tensor.hpp"
```

Classes

- struct DefaultValues< Tensor_ >

    Default Values for CPD algorithm.
- struct Options< Tensor_, ExecutionPolicy_, DefaultValues_ >

    Manage defaults parameters for CPD algorithm.
- struct SparseStatus< _TnsSize, ExecutionPolicy_, DefaultValues_ >
- struct Status< Tensor_, ExecutionPolicy_, DefaultValues_ >

    Returned Type of CPD algorithm.

Typedefs

- using Clock = std::chrono::high_resolution_clock
- using Duration = std::chrono::nanoseconds

Functions

- template<typename Tensor_ , typename ExecutionPolicy_ = execution::sequenced_policy, template< typename T > class DefaultValues_ = DefaultValues>
  Options< Tensor_, ExecutionPolicy_, DefaultValues_ > MakeOptions ()
- template<typename Tensor_ , typename ExecutionPolicy_ = execution::sequenced_policy, template< typename T > class DefaultValues_ = DefaultValues>
  Options< Tensor_, ExecutionPolicy_, DefaultValues_ > MakeOptions (DefaultValues_< Tensor_ > &&dv, ExecutionPolicy_ &&xp)
- template<std::size_t _TnsSize, typename ExecutionPolicy_ = execution::sequenced_policy, template< typename T > class DefaultValues_ = SparseDefaultValues>
  SparseOptions< _TnsSize, ExecutionPolicy_, DefaultValues_ > MakeSparseOptions ()
- template<std::size_t _TnsSize, typename ExecutionPolicy_ = execution::sequenced_policy, template< typename T > class DefaultValues_ = SparseDefaultValues>
  SparseOptions< _TnsSize, ExecutionPolicy_, DefaultValues_ > MakeSparseOptions (DefaultValues_< partensor::SparseTensor< _TnsSize > > &&dv, ExecutionPolicy_ &&xp)

8.51.1 Detailed Description

Containts the the most basic information for `PARTENSOR` project.

8.51.2 Typedef Documentation

8.51.2.1 Clock

```
using Clock = std::chrono::high_resolution_clock
```

Chrono type for measuring time.

8.51.2.2 Duration

```
using Duration = std::chrono::nanoseconds
```

Type for chrono and duration time.

8.51.3 Function Documentation

8.51.3.1 MakeOptions() [1/2]

```
Options< Tensor_, ExecutionPolicy_, DefaultValues_ > partensor::MakeOptions ( )
```

< Tensor Order.

8.51.3.2 MakeOptions() [2/2]

```
Options< Tensor_, ExecutionPolicy_, DefaultValues_ > partensor::MakeOptions (
            DefaultValues_< Tensor_ > && dv,
            ExecutionPolicy_ && xp )
```

< Tensor Order.

8.51.3.3 MakeSparseOptions() [1/2]

```
SparseOptions< _TnsSize, ExecutionPolicy_, DefaultValues_ > partensor::MakeSparseOptions ( )
```

< Tensor Order.

8.51.3.4 MakeSparseOptions() [2/2]

```
SparseOptions< _TnsSize, ExecutionPolicy_, DefaultValues_ > partensor::MakeSparseOptions (
            DefaultValues_< partensor::SparseTensor< _TnsSize > > && dv,
            ExecutionPolicy_ && xp )
```

< Tensor Order.

8.52 PARTENSOR_basic.hpp

Go to the documentation of this file.

```
#ifndef DOXYGEN_SHOULD_SKIP_THIS
#endif // DOXYGEN_SHOULD_SKIP_THIS
/***********************************************************************/
#ifndef PARTENSOR_BASIC_HPP
#define PARTENSOR_BASIC_HPP

#define EIGEN_PERMANENTLY_DISABLE_STUPID_WARNINGS 1

#if defined __GNUC__ && __GNUC__>=6
# pragma GCC diagnostic push
# pragma GCC diagnostic ignored "-Wignored-attributes"
# pragma GCC diagnostic ignored "-Wunknown-pragmas"
#endif

#include "Config.hpp"

#include <mutex>

#if USE_MPI
#include "ParallelWrapper.hpp"
#endif /* USE_MPI */

#include <Eigen/Dense>
#include <Eigen/Sparse>
#include <unsupported/Eigen/CXX11/Tensor>
#include "spdlog/spdlog.h"
#include "spdlog/sinks/basic_file_sink.h"
#include "execution.hpp"
#include "Constants.hpp"
#include "Tensor.hpp"

namespace partensor
{
#if USE_MPI
  using MPI_Environment  = partensor::environment;
  using MPI_Communicator = partensor::communicator;
#endif /* USE_MPI */

  using Clock    = std::chrono::high_resolution_clock;
  using Duration = std::chrono::nanoseconds;             // mircoseconds??

  class Environment
  {
  public:
    Environment(int argc, char *argv[], char *envp[])
  #if USE_MPI
      : mMpiEnv(argc,argv,true)
  #endif /* USE_MPI */
    {
      (void) argc;
      (void) argv;
      (void) envp;

      // create an spdlog object with pattern:
      // %l       = The log level of the message (eg info, warn)
      // %Y-%m-%d = Year in 4 digits - Month 01-12 - Day of month 01-31 (eg 2019-09-19)
      // %r       = 12 hour clock (eg 02:55:02 pm)
      // %F       = Nanosecond part of the current second 000000000-999999999 (eg 256789123)
      // %n       = Logger's name (eg some logger name)
      // %v       = The actual text to log (eg "some user text")
      spdlog::set_pattern("<%l> : [%Y-%m-%d %r] [%F] [%n] %v");
      mLogger = spdlog::basic_logger_mt("Partensor", "../log/partensor.txt");
    }

    Environment(int argc, char *argv[]) : Environment(argc,argv,nullptr)
    { }

    Environment() : Environment(0, nullptr,nullptr)
    { }

    ~Environment()
    {
      mLogger->flush();
      // TODO spdlog::drop("Partensor");
    }

    static Environment *Partensor(int argc, char **argv, char **envp)
    {
      static std::mutex                   l_mutex;
      static std::unique_ptr<Environment> l_partensor(nullptr);

       if (!l_partensor)
```

```
      {
        std::lock_guard<std::mutex> lock(l_mutex);

        if (!l_partensor)
        {
          if (argc == 0)
            l_partensor.reset(new Environment());
          else if (envp == nullptr)
            l_partensor.reset(new Environment(argc,argv));
          else
            l_partensor.reset(new Environment(argc,argv,envp));
        }
      }

      return l_partensor.get();
    }

    std::shared_ptr<spdlog::logger> Logger()
    {
      return mLogger;
    }

#if USE_MPI
    MPI_Environment &MpiEnvironment()
    {
      return mMpiEnv;
    }

    MPI_Communicator &MpiCommunicator()
    {
      return mMpiCom;
    }

#endif /* USE_MPI */
  private:
    std::shared_ptr<spdlog::logger>  mLogger;

  #if USE_MPI
    MPI_Environment   mMpiEnv;
    MPI_Communicator  mMpiCom;
  #endif  /* USE_MPI */
  };

  inline Environment *Partensor(int argc=0, char **argv=nullptr, char **envp=nullptr)
  {
    return Environment::Partensor(argc,argv,envp);
  }

  inline void Init(int argc, char **argv, char **envp)
  {
    Partensor(argc,argv,envp);
  }

  inline void Init(int argc, char **argv)
  {
    Partensor(argc,argv);
  }

  inline void Init()
  {
    Partensor();
  }

  template<typename Tensor_>
  struct DefaultValues {

    static std::size_t constexpr TnsSize = TensorTraits<Tensor_>::TnsSize;

    using DoubleArray = typename TensorTraits<Tensor_>::DoubleArray;
    using Constraints = typename TensorTraits<Tensor_>::Constraints;
    using IntArray    = typename TensorTraits<Tensor_>::IntArray;

    static Method      constexpr DefaultMethod = Method::als;
    // static Constraint  constexpr DefaultConstraint = Constraint::unconstrained;   /**< Default value for Constraint is unconstrained. */
    static Constraint  constexpr DefaultConstraint = Constraint::nonnegativity;
    static double      constexpr DefaultThresholdError = 1e-3;
    static double      constexpr DefaultNesterovTolerance = 1e-2;
    static unsigned    constexpr DefaultMaxIter = 20;
    static Duration    constexpr DefaultMaxDuration = Duration(10000);
    static double      constexpr DefaultLambda = 0.1;
    static double      constexpr DefaultProcessorPerMode = 2;
    static int         constexpr DefaultAccelerationCoefficient = 3;
    static int         constexpr DefaultAccelerationFail = 0;
    static bool        constexpr DefaultAcceleration = true;
    static bool        constexpr DefaultNormalization = true;
    static bool        constexpr DefaultWriteToFile = false;
```

```
    static DoubleArray constexpr DefaultLambdas = []() constexpr -> auto {
      DoubleArray c{};
      for (auto &e : c) e = DefaultLambda;
      return c;
    } ();

    static Constraints constexpr DefaultConstraints = []() constexpr -> auto {
      Constraints c{};
      for (auto &e : c) e = DefaultConstraint;
      return c;
    } ();

    static IntArray constexpr DefaultProcessorsPerMode = []() constexpr -> auto {
      IntArray c{};
      for(auto &e : c) e = DefaultProcessorPerMode;
      return c;
    } ();

  };

  template<typename SparseTensor_>
  struct SparseDefaultValues {

    static std::size_t constexpr TnsSize = SparseTensorTraits<SparseTensor_>::TnsSize;

    using DoubleArray = typename SparseTensorTraits<SparseTensor_>::DoubleArray;
    using Constraints = typename SparseTensorTraits<SparseTensor_>::Constraints;
    using IntArray    = typename SparseTensorTraits<SparseTensor_>::IntArray;

    static Method       constexpr DefaultMethod = Method::als;
    // static Constraint  constexpr DefaultConstraint = Constraint::unconstrained;   /**< Default value for Constraint is unconstrained. */
    static Constraint  constexpr DefaultConstraint = Constraint::nonnegativity;
    static double      constexpr DefaultThresholdError = 1e-3;
    static double      constexpr DefaultNesterovTolerance = 1e-2;
    static int         constexpr DefaultMaxNesterovIter   = 20;
    static unsigned    constexpr DefaultMaxIter = 20;
    static Duration    constexpr DefaultMaxDuration = Duration(10000);
    static double      constexpr DefaultC_stochastic_perc = 0.5;
    static double      constexpr DefaultLambda = 0.01;
    static double      constexpr DefaultProcessorPerMode = 2;
    static int         constexpr DefaultAccelerationCoefficient = 3;
    static int         constexpr DefaultAccelerationFail = 0;
    static bool        constexpr DefaultAcceleration = false;
    static bool        constexpr DefaultAveraging = false;
    static bool        constexpr DefaultNormalization = false;
    static bool        constexpr DefaultInitializeFactors = false;
    static bool        constexpr DefaultReadFactorsFromFile = false;
    static bool        constexpr DefaultWriteToFile = false;
    static int        constexpr DefaultNonZeros     = 1000;
    static DoubleArray constexpr DefaultLambdas = []() constexpr -> auto {
      DoubleArray c{};
      for (auto &e : c) e = DefaultLambda;
      return c;
    } ();

    static Constraints constexpr DefaultConstraints = []() constexpr -> auto {
      Constraints c{};
      for (auto &e : c) e = DefaultConstraint;
      return c;
    } ();

    static IntArray constexpr DefaultProcessorsPerMode = []() constexpr -> auto {
      IntArray c{};
      for(auto &e : c) e = DefaultProcessorPerMode;
      return c;
    } ();

  };

  template < typename Tensor_,
             typename ExecutionPolicy_ = execution::sequenced_policy,
             template <typename T> class DefaultValues_ = DefaultValues    >
  struct Options
  {
      static constexpr std::size_t TnsSize = TensorTraits<Tensor_>::TnsSize;
      using DataType    = typename TensorTraits<Tensor_>::DataType;
      using MatrixType  = typename TensorTraits<Tensor_>::MatrixType;
      using Constraints = typename TensorTraits<Tensor_>::Constraints;
      using DoubleArray = typename TensorTraits<Tensor_>::DoubleArray;
      using StringArray = std::array<std::string, TnsSize>;

      Method         method;
      Constraints    constraints;
      double         threshold_error;
      double         nesterov_delta_1;
      double         nesterov_delta_2;
```

```
    DoubleArray   lambdas;
    unsigned      max_iter;
    Duration      max_duration;
    int           accel_coeff;
    int           accel_fail;
    bool          acceleration;
    bool          normalization;
    bool          writeToFile;
    StringArray   final_factors_paths;

    Options() : method(DefaultValues_<Tensor_>::DefaultMethod),
                constraints(DefaultValues_<Tensor_>::DefaultConstraints),
                threshold_error(DefaultValues_<Tensor_>::DefaultThresholdError),
                nesterov_delta_1(DefaultValues_<Tensor_>::DefaultNesterovTolerance),
                nesterov_delta_2(DefaultValues_<Tensor_>::DefaultNesterovTolerance),
                lambdas(DefaultValues_<Tensor_>::DefaultLambdas),
                max_iter(DefaultValues_<Tensor_>::DefaultMaxIter),
                max_duration(DefaultValues_<Tensor_>::DefaultMaxDuration),
                accel_coeff(DefaultValues_<Tensor_>::DefaultAccelerationCoefficient),
                accel_fail(DefaultValues_<Tensor_>::DefaultAccelerationFail),
                acceleration(DefaultValues_<Tensor_>::DefaultAcceleration),
                normalization(DefaultValues_<Tensor_>::DefaultNormalization),
                writeToFile(DefaultValues_<Tensor_>::DefaultWriteToFile)//,              /**< Default value for write final factors to files. */
                // final_factors_paths(DefaultValues_<Tensor_>::DefaultFinalFactorsPaths) /**< Default value for path for factors at the end of the algorithm. */
    {
      for(std::size_t i=0; i<TnsSize; ++i)
      {
        final_factors_paths[i] = "final_" + std::to_string(i) + ".bin";
      }
    }
    Options(Options const &) = default;
    Options(Options       &&) = default;

    Options &operator=(Options const &) = default;
    Options &operator=(Options       &&) = default;
  };

/*
    Sparse Options
*/
template < std::size_t _TnsSize,
           typename ExecutionPolicy_ = execution::sequenced_policy,
           template <typename T> class DefaultValues_ = SparseDefaultValues    >
struct SparseOptions
{
    using SparseTenor = typename partensor::SparseTensor<_TnsSize>;
    static constexpr std::size_t TnsSize = SparseTensorTraits<SparseTenor>::TnsSize;
    using DataType         = typename SparseTensorTraits<SparseTenor>::DataType;
    using MatrixType       = typename SparseTensorTraits<SparseTenor>::MatrixType;
    using Constraints      = typename SparseTensorTraits<SparseTenor>::Constraints;
    using DoubleArray      = typename SparseTensorTraits<SparseTenor>::DoubleArray;
    using SparseMatrixType = typename SparseTensorTraits<SparseTenor>::SparseMatrixType;
    using LongMatrixType   = typename SparseTensorTraits<SparseTenor>::LongMatrixType;
    using Dimensions       = typename SparseTensorTraits<SparseTenor>::Dimensions;
    using SparseTensor     = typename SparseTensorTraits<SparseTenor>::SparseTensor;
    using MatrixArray      = typename SparseTensorTraits<SparseTenor>::MatrixArray;
    using StringArray      = std::array<std::string, TnsSize>;

    int                         rank;
    std::array<int, TnsSize> tnsDims;
    int                         nonZeros;
    bool                        initialized_factors;
    bool                        read_factors_from_file;
    MatrixArray                 factorsInit;

    Method        method;
    Constraints   constraints;
    double        threshold_error;
    double        nesterov_delta_1;
    double        nesterov_delta_2;
    int           max_nesterov_iter;
    double        c_stochastic_perc;
    DoubleArray   lambdas;
    unsigned      max_iter;
    Duration      max_duration;
    int           accel_coeff;
    int           accel_fail;
    bool          acceleration;
    bool          averaging;
    bool          normalization;
    bool          writeToFile;

    std::string   ratings_path;
    StringArray   initial_factors_paths;
    StringArray   final_factors_paths;
```

```cpp

    SparseOptions() : initialized_factors(DefaultValues_<SparseTenor>::DefaultInitializeFactors),
                      read_factors_from_file(DefaultValues_<SparseTenor>::DefaultReadFactorsFromFile),
                      method(DefaultValues_<SparseTenor>::DefaultMethod),
                      constraints(DefaultValues_<SparseTenor>::DefaultConstraints),
                      threshold_error(DefaultValues_<SparseTenor>::DefaultThresholdError),
                      nesterov_delta_1(DefaultValues_<SparseTenor>::DefaultNesterovTolerance),
                      nesterov_delta_2(DefaultValues_<SparseTenor>::DefaultNesterovTolerance),
                      max_nesterov_iter(DefaultValues_<SparseTenor>::DefaultMaxNesterovIter),
                      c_stochastic_perc(DefaultValues_<SparseTenor>::DefaultC_stochastic_perc),
                      lambdas(DefaultValues_<SparseTenor>::DefaultLambdas),
                      max_iter(DefaultValues_<SparseTenor>::DefaultMaxIter),
                      max_duration(DefaultValues_<SparseTenor>::DefaultMaxDuration),
                      accel_coeff(DefaultValues_<SparseTenor>::DefaultAccelerationCoefficient),
                      accel_fail(DefaultValues_<SparseTenor>::DefaultAccelerationFail),
                      acceleration(DefaultValues_<SparseTenor>::DefaultAcceleration),
                      averaging(DefaultValues_<SparseTenor>::DefaultAveraging),
                      normalization(DefaultValues_<SparseTenor>::DefaultNormalization),
                      writeToFile(DefaultValues_<SparseTenor>::DefaultWriteToFile)
                      // final_factors_paths(DefaultValues_<SparseTenor>::DefaultFinalFactorsPaths) /**< Default value for path for factors at the end of the algorithm. */
    {
      for(std::size_t i=0; i<TnsSize; ++i)
      {
        final_factors_paths[i] = "final_" + std::to_string(i) + ".bin";
      }
    }
    SparseOptions(SparseOptions const &) = default;
    SparseOptions(SparseOptions      &&) = default;

    SparseOptions &operator=(SparseOptions const &) = default;
    SparseOptions &operator=(SparseOptions      &&) = default;
  };

  template < typename Tensor_,
             typename ExecutionPolicy_ = execution::sequenced_policy,
             template <typename T> class DefaultValues_ = DefaultValues   >
  Options<Tensor_, ExecutionPolicy_, DefaultValues_> MakeOptions()
  {
    Options<Tensor_,ExecutionPolicy_,DefaultValues_> options;
    static constexpr std::size_t TnsSize = TensorTraits<Tensor_>::TnsSize;
    options.method              = DefaultValues_<Tensor_>::DefaultMethod;                   // Default value for Method is als.
    options.constraints         = DefaultValues_<Tensor_>::DefaultConstraints;              // Default value for Constraint is no negative.
    options.threshold_error     = DefaultValues_<Tensor_>::DefaultThresholdError;           // Default value for cost function's threshold.
    options.nesterov_delta_1    = DefaultValues_<Tensor_>::DefaultNesterovTolerance;        // Default value for Nesterov's tolerance.
    options.nesterov_delta_2    = DefaultValues_<Tensor_>::DefaultNesterovTolerance;        // Default value for Nesterov's tolerance.
    options.lambdas             = DefaultValues_<Tensor_>::DefaultLambdas;                  // Default value for lambda.
    options.max_iter            = DefaultValues_<Tensor_>::DefaultMaxIter;                  // Default value outer loop maximum iterations.
    options.max_duration        = DefaultValues_<Tensor_>::DefaultMaxDuration;              // Default value outer loop maximum duration.
    options.accel_coeff         = DefaultValues_<Tensor_>::DefaultAccelerationCoefficient; // Default value for acceleration coefficient.
    options.accel_fail          = DefaultValues_<Tensor_>::DefaultAccelerationFail;         // Default value for acceleration fail.
    options.acceleration        = DefaultValues_<Tensor_>::DefaultAcceleration;             // Default value for acceleration.
    // options.averaging            = DefaultValues_<Tensor_>::DefaultAveraging;               // Default value for averaging.
    options.normalization       = DefaultValues_<Tensor_>::DefaultNormalization;            // Default value for normalization.
    options.writeToFile         = DefaultValues_<Tensor_>::DefaultWriteToFile;              // Default value for write final factors to files.
    // options.final_factors_paths = DefaultValues_<Tensor_>::DefaultFinalFactorsPaths;       // Default value for path for factors at the end of the algorithm.
    for(std::size_t i=0; i<TnsSize; ++i)
    {
      options.final_factors_paths[i] = "final_" + std::to_string(i) + ".bin";
    }
    return options;
  }

  template < std::size_t _TnsSize,
             typename ExecutionPolicy_ = execution::sequenced_policy,
             template <typename T> class DefaultValues_ = SparseDefaultValues   >
  SparseOptions<_TnsSize, ExecutionPolicy_, DefaultValues_> MakeSparseOptions()
  {
    SparseOptions<_TnsSize,ExecutionPolicy_,DefaultValues_> options;
    using SparseTensor          = typename partensor::SparseTensor<_TnsSize>;
    static constexpr std::size_t TnsSize = SparseTensorTraits<SparseTensor>::TnsSize;
    options.method              = DefaultValues_<SparseTensor>::DefaultMethod;                //
```

```
Default value for Method is als.
    options.constraints         = DefaultValues_<SparseTensor>::DefaultConstraints;            //
Default value for Constraint is no negative.
    options.threshold_error     = DefaultValues_<SparseTensor>::DefaultThresholdError;         //
Default value for cost function's threshold.
    options.nesterov_delta_1    = DefaultValues_<SparseTensor>::DefaultNesterovTolerance;      //
Default value for Nesterov's tolerance.
    options.nesterov_delta_2    = DefaultValues_<SparseTensor>::DefaultNesterovTolerance;      //
Default value for Nesterov's tolerance.
    options.c_stochastic_perc   = DefaultValues_<SparseTensor>::DefaultC_stochastic_perc;
    options.lambdas             = DefaultValues_<SparseTensor>::DefaultLambdas;                //
Default value for lambda.
    options.max_iter            = DefaultValues_<SparseTensor>::DefaultMaxIter;                //
Default value outer loop maximum iterations.
    options.max_duration        = DefaultValues_<SparseTensor>::DefaultMaxDuration;            //
Default value outer loop maximum duration.
    options.accel_coeff         = DefaultValues_<SparseTensor>::DefaultAccelerationCoefficient; //
Default value for acceleration coefficient.
    options.accel_fail          = DefaultValues_<SparseTensor>::DefaultAccelerationFail;       //
Default value for acceleration fail.
    options.acceleration        = DefaultValues_<SparseTensor>::DefaultAcceleration;           //
Default value for acceleration.
    options.averaging           = DefaultValues_<SparseTensor>::DefaultAveraging;              //
Default value for averaging.
    options.normalization       = DefaultValues_<SparseTensor>::DefaultNormalization;          //
Default value for normalization.
    options.writeToFile         = DefaultValues_<SparseTensor>::DefaultWriteToFile;            //
Default value for write final factors to files.
    // options.final_factors_paths = DefaultValues_<SparseTensor>::DefaultFinalFactorsPaths;       //
Default value for path for factors at the end of the algorithm.
    for(std::size_t i=0; i<TnsSize; ++i)
    {
      options.final_factors_paths[i] = "final_" + std::to_string(i) + ".bin";
    }
    return options;
  }

  template < typename Tensor_,
             typename ExecutionPolicy_ = execution::sequenced_policy,
             template <typename T> class DefaultValues_ = DefaultValues     >
  Options<Tensor_,ExecutionPolicy_,DefaultValues_> MakeOptions(DefaultValues_<Tensor_> &&dv,
ExecutionPolicy_ &&xp)
  {
    Options<Tensor_,ExecutionPolicy_,DefaultValues_> options;
    static constexpr std::size_t TnsSize = TensorTraits<Tensor_>::TnsSize;
    options.method              = DefaultValues_<Tensor_>::DefaultMethod;                  // Default
value for Method is als.
    options.constraints         = DefaultValues_<Tensor_>::DefaultConstraints;             // Default
value for Constraint is no negative.
    options.threshold_error     = DefaultValues_<Tensor_>::DefaultThresholdError;          // Default
value for cost function's threshold.
    options.nesterov_delta_1    = DefaultValues_<Tensor_>::DefaultNesterovTolerance;       // Default
value for Nesterov's tolerance.
    options.nesterov_delta_2    = DefaultValues_<Tensor_>::DefaultNesterovTolerance;       // Default
value for Nesterov's tolerance.
    options.lambdas             = DefaultValues_<Tensor_>::DefaultLambdas;                 // Default
value for lambda.
    options.max_iter            = DefaultValues_<Tensor_>::DefaultMaxIter;                 // Default
value outer loop maximum iterations.
    options.max_duration        = DefaultValues_<Tensor_>::DefaultMaxDuration;             // Default
value outer loop maximum duration.
    options.accel_coeff         = DefaultValues_<Tensor_>::DefaultAccelerationCoefficient; // Default
value for acceleration coefficient.
    options.accel_fail          = DefaultValues_<Tensor_>::DefaultAccelerationFail;        // Default
value for acceleration fail.
    options.acceleration        = DefaultValues_<Tensor_>::DefaultAcceleration;            // Default
value for acceleration.
    // options.averaging           = DefaultValues_<Tensor_>::DefaultAveraging;               //
Default value for averaging.
    options.normalization       = DefaultValues_<Tensor_>::DefaultNormalization;           // Default
value for normalization.
    options.writeToFile         = DefaultValues_<Tensor_>::DefaultWriteToFile;             // Default
value for write final factors to files.
    // options.final_factors_paths = DefaultValues_<Tensor_>::DefaultFinalFactorsPaths;       //
Default value for path for factors at the end of the algorithm.
    for(std::size_t i=0; i<TnsSize; ++i)
    {
      options.final_factors_paths[i] = "final_" + std::to_string(i) + ".bin";
    }
    return options;
  }

  template < std::size_t _TnsSize,
             typename ExecutionPolicy_ = execution::sequenced_policy,
             template <typename T> class DefaultValues_ = SparseDefaultValues    >
  SparseOptions<_TnsSize, ExecutionPolicy_, DefaultValues_>
MakeSparseOptions(DefaultValues_<partensor::SparseTensor<_TnsSize» &&dv, ExecutionPolicy_ &&xp)
```

```
  {
    SparseOptions<_TnsSize,ExecutionPolicy_,DefaultValues_> options;
    using SparseTensor           = typename partensor::SparseTensor<_TnsSize>;
    static constexpr std::size_t TnsSize = SparseTensorTraits<SparseTensor>::TnsSize;
    options.method               = DefaultValues_<SparseTensor>::DefaultMethod;                   // Default value for Method is als.
    options.constraints          = DefaultValues_<SparseTensor>::DefaultConstraints;              // Default value for Constraint is no negative.
    options.threshold_error      = DefaultValues_<SparseTensor>::DefaultThresholdError;           // Default value for cost function's threshold.
    options.nesterov_delta_1     = DefaultValues_<SparseTensor>::DefaultNesterovTolerance;        // Default value for Nesterov's tolerance.
    options.nesterov_delta_2     = DefaultValues_<SparseTensor>::DefaultNesterovTolerance;        // Default value for Nesterov's tolerance.
    options.c_stochastic_perc    = DefaultValues_<SparseTensor>::DefaultC_stochastic_perc;
    options.lambdas              = DefaultValues_<SparseTensor>::DefaultLambdas;                  // Default value for lambda.
    options.max_iter             = DefaultValues_<SparseTensor>::DefaultMaxIter;                  // Default value outer loop maximum iterations.
    options.max_duration         = DefaultValues_<SparseTensor>::DefaultMaxDuration;              // Default value outer loop maximum duration.
    options.accel_coeff          = DefaultValues_<SparseTensor>::DefaultAccelerationCoefficient; // Default value for acceleration coefficient.
    options.accel_fail           = DefaultValues_<SparseTensor>::DefaultAccelerationFail;         // Default value for acceleration fail.
    options.acceleration         = DefaultValues_<SparseTensor>::DefaultAcceleration;             // Default value for acceleration.
    options.averaging            = DefaultValues_<SparseTensor>::DefaultAveraging;
  // Default value for averaging.
    options.normalization        = DefaultValues_<SparseTensor>::DefaultNormalization;            // Default value for normalization.
    options.writeToFile          = DefaultValues_<SparseTensor>::DefaultWriteToFile;              // Default value for write final factors to files.
    // options.final_factors_paths = DefaultValues_<SparseTensor>::DefaultFinalFactorsPaths;       // Default value for path for factors at the end of the algorithm.
    for(std::size_t i=0; i<TnsSize; ++i)
    {
      options.final_factors_paths[i] = "final_" + std::to_string(i) + ".bin";
    }
    return options;
  }

  template < typename Tensor_,
             typename ExecutionPolicy_ = execution::sequenced_policy,
             template <typename T> class DefaultValues_ = DefaultValues  >
  struct Status
  {
    using MatrixArray  = typename TensorTraits<Tensor_>::MatrixArray;
    Options<Tensor_,ExecutionPolicy_,DefaultValues_> options;
    double                                           frob_tns          = 0.0;
    double                                           f_value           = 0.0;
    double                                           rel_costFunction  = 0.0;
    unsigned                                         ao_iter           = 0;
    MatrixArray                                      factors;
    Status()                = default;
    Status(Status const &) = default;
    Status(Status      &&) = default;

    Status(Options<Tensor_,ExecutionPolicy_,DefaultValues_> const &opt) : options(opt)
    {  }

    Status &operator=(Status const &) = default;
    Status &operator=(Status      &&) = default;

    /*
    Constructor called, in case the user decides to change one or more
    of the options and use them in the factorization.
    */
    Status &operator=(Options<Tensor_,ExecutionPolicy_,DefaultValues_> const &opts)
    {
    options = opts;

    return *this;
    }

    explicit operator Options<Tensor_,ExecutionPolicy_,DefaultValues_>& ()
    {
    return options;
    }
  };

  template < std::size_t _TnsSize,
             typename ExecutionPolicy_ = execution::sequenced_policy,
             template <typename T> class DefaultValues_ = SparseDefaultValues  >
  struct SparseStatus
  {
    using SparseTensor = typename partensor::SparseTensor<_TnsSize>;
```

```
    using MatrixArray  = typename SparseTensorTraits<SparseTensor>::MatrixArray;
    SparseOptions<_TnsSize,ExecutionPolicy_,DefaultValues_> options;
    double                                              frob_tns          = 0.0;
    double                                              f_value           = 0.0;
    double                                              rel_costFunction  = 0.0;
    unsigned                                            ao_iter           = 0;
    MatrixArray                                         factors;
    SparseStatus()              = default;
    SparseStatus(SparseStatus const &) = default;
    SparseStatus(SparseStatus      &&) = default;

    SparseStatus(SparseOptions<_TnsSize,ExecutionPolicy_,DefaultValues_> const &opt) : options(opt)
    {  }

    SparseStatus &operator=(SparseStatus const &) = default;
    SparseStatus &operator=(SparseStatus      &&) = default;

    /*
    Constructor called, in case the user decides to change one or more
    of the options and use them in the factorization.
    */
    SparseStatus &operator=(SparseOptions<_TnsSize,ExecutionPolicy_,DefaultValues_> const &opts)
    {
      options = opts;

      return *this;
    }

    explicit operator SparseOptions<_TnsSize,ExecutionPolicy_,DefaultValues_>& ()
    {
      return options;
    }
  };

  template < typename Tensor_,
             typename ExecutionPolicy_ = execution::sequenced_policy,
             template <typename T> class DefaultValues_ = DefaultValues   >
  Status<Tensor_> MakeStatus()
  {
    Status<Tensor_,ExecutionPolicy_,DefaultValues_>  status;

    status.options = MakeOptions<Tensor_,ExecutionPolicy_,DefaultValues_>();

    return status;
  }

  template <typename Tensor_, typename ExecutionPolicy_, template <typename T> class DefaultValues_>
  Status<Tensor_,std::remove_cv_t<std::remove_reference_t<ExecutionPolicy_»,DefaultValues_>
 MakeStatus(DefaultValues_<Tensor_> &&dv, ExecutionPolicy_ &&xp)
  {
    using ExecutionPolicy_t = std::remove_cv_t<std::remove_reference_t<ExecutionPolicy_»;

    Status<Tensor_,ExecutionPolicy_t,DefaultValues_>  status;

    status.options = MakeOptions<Tensor_>(std::forward<ExecutionPolicy_>(xp));

    return status;
  }

  template < std::size_t _TnsSize,
             typename ExecutionPolicy_ = execution::sequenced_policy,
             template <typename T> class DefaultValues_ = SparseDefaultValues   >
  SparseStatus<_TnsSize> MakeSparseStatus()
  {
    SparseStatus<_TnsSize,ExecutionPolicy_,DefaultValues_>  status;

    status.options = MakeSparseOptions<_TnsSize,ExecutionPolicy_,DefaultValues_>();

    return status;
  }

  template < std::size_t _TnsSize,
             typename ExecutionPolicy_,
             template <typename T> class DefaultValues_   >
  SparseStatus<_TnsSize,std::remove_cv_t<std::remove_reference_t<ExecutionPolicy_»,DefaultValues_>
 MakeSparseStatus(DefaultValues_<partensor::SparseTensor<_TnsSize» &&dv, ExecutionPolicy_ &&xp)
  {
    SparseStatus<_TnsSize,ExecutionPolicy_,DefaultValues_>  status;

    status.options = MakeSparseOptions<_TnsSize,ExecutionPolicy_,DefaultValues_>();

    return status;
  }

  template <typename Tensor_>
  struct Options<Tensor_,execution::openmpi_policy,DefaultValues> : public Options<Tensor_>
  {
```

```
    using Options<Tensor_,execution::sequenced_policy,DefaultValues>::constraints;

    using IntArray = typename TensorTraits<Tensor_>::IntArray;

    IntArray proc_per_mode;

    Options() : proc_per_mode(DefaultValues<Tensor_>::DefaultProcessorsPerMode)
    { }
    Options(Options const &) = default;
    Options(Options      &&) = default;

    Options &operator=(Options const &) = default;
    Options &operator=(Options      &&) = default;
  };

  template <typename Tensor_>
  Options<Tensor_,execution::openmpi_policy,DefaultValues> MakeOptions(execution::openmpi_policy &&)
  {
    Options<Tensor_,execution::openmpi_policy,DefaultValues>  options;

    static_cast<Options<Tensor_>&>(options) = MakeOptions<Tensor_>();

    options.proc_per_mode = DefaultValues<Tensor_>::DefaultProcessorsPerMode;

    return options;
  }

  template <std::size_t _TnsSize>
  struct SparseOptions<_TnsSize,execution::openmpi_policy,SparseDefaultValues> : public
 SparseOptions<_TnsSize>
  {
    using SparseOptions<_TnsSize,execution::sequenced_policy,SparseDefaultValues>::constraints;

    using SparseTensor = typename partensor::SparseTensor<_TnsSize>;
    using IntArray     = typename SparseTensorTraits<SparseTensor>::IntArray;

    IntArray proc_per_mode;

    SparseOptions() : proc_per_mode(SparseDefaultValues<SparseTensor>::DefaultProcessorsPerMode)
    { }
    SparseOptions(SparseOptions const &) = default;
    SparseOptions(SparseOptions      &&) = default;

    SparseOptions &operator=(SparseOptions const &) = default;
    SparseOptions &operator=(SparseOptions      &&) = default;
  };

  template <std::size_t _TnsSize>
  SparseOptions<_TnsSize,execution::openmpi_policy,SparseDefaultValues>
 MakeSparseOptions(execution::openmpi_policy &&)
  {
    using SparseTensor = typename partensor::SparseTensor<_TnsSize>;

    SparseOptions<_TnsSize,execution::openmpi_policy,SparseDefaultValues>  options;

    static_cast<SparseOptions<_TnsSize>&>(options) = MakeSparseOptions<_TnsSize>();

    options.proc_per_mode = SparseDefaultValues<SparseTensor>::DefaultProcessorsPerMode;

    return options;
  }

  template <typename Tensor_>
  using MpiOptions = Options<Tensor_,execution::openmpi_policy,DefaultValues>;

  template <typename Tensor_>
  using MpiStatus = Status<Tensor_,execution::openmpi_policy,DefaultValues>;

  template < std::size_t _TnsSize>
  using MpiSparseOptions = SparseOptions<_TnsSize,execution::openmpi_policy,SparseDefaultValues>;

  template < std::size_t _TnsSize>
  using MpiSparseStatus = SparseStatus<_TnsSize,execution::openmpi_policy,SparseDefaultValues>;

  template <typename Tensor_>
  using OmpOptions = Options<Tensor_,execution::openmp_policy,DefaultValues>;

  template <typename Tensor_>
  using OmpStatus = Status<Tensor_,execution::openmp_policy,DefaultValues>;

  template <typename Tensor_>
  using CudaOptions = Options<Tensor_, execution::cuda_policy, DefaultValues>;

  template <typename Tensor_>
  using CudaStatus = Status<Tensor_, execution::cuda_policy, DefaultValues>;

  template <std::size_t _TnsSize>
```

```
    using OmpSparseOptions = SparseOptions<_TnsSize,execution::openmp_policy,SparseDefaultValues>;

    template <std::size_t _TnsSize>
    using OmpSparseStatus = SparseStatus<_TnsSize,execution::openmp_policy,SparseDefaultValues>;

} // namespace partensor

namespace ptl = partensor;

#endif // PARTENSOR_BASIC_HPP
```

8.53 PartialCwiseProd.hpp File Reference

`#include "CwiseProd.hpp"`

Functions

- template<typename MatrixArray_ >
  auto PartialCwiseProd (MatrixArray_ const &matArray, std::size_t const mode)

  Partial `CwiseProd` implementation.

8.53.1 Detailed Description

Implementation for partial computation of an element wise product. Makes use of `CwiseProd` function from `CwiseProd.hpp`.

Warning

The implementation supports only this operation, when the number of Matrices of `Matrix` type are in range of [3-8].

8.53.2 Function Documentation

8.53.2.1 PartialCwiseProd()

```
auto partensor::v1::PartialCwiseProd (
            MatrixArray_ const & matArray,
            std::size_t const mode )
```

Partial `CwiseProd` implementation.

Computes the element wise product of matrices of `Matrix` type contained in an array container, excluding the one specified in `mode`.

Template Parameters

*Matrix↩Array_*	An array container type.

Parameters

matArray	[in] An array containing matrices to use in this operation.
mode	[in] Id of matrix to exclude from the element wise product.

Returns

A `Matrix` with the result.

Note

The function is supported only, when the size of `matArray` is in range of [3-8].

8.54 PartialCwiseProd.hpp

Go to the documentation of this file.

```
#ifndef DOXYGEN_SHOULD_SKIP_THIS
#endif  // DOXYGEN_SHOULD_SKIP_THIS
/********************************************************************/
#ifndef PARTENSOR_PARTIAL_CWISE_PROD_HPP
#define PARTENSOR_PARTIAL_CWISE_PROD_HPP

#include "CwiseProd.hpp"

namespace partensor
{
        inline namespace v1 {
                template <typename MatrixArray_>
                auto PartialCwiseProd( MatrixArray_ const &matArray,
                                           std::size_t  const  mode     )
                {
                        using Matrix = typename MatrixArrayTraits<MatrixArray_>::value_type;   // Type of data of @c Eigen Matrix (e.g double, float, ..).

                        constexpr std::size_t TnsSize = MatrixArrayTraits<MatrixArray_>::Size; // Size of matArray.

                        if constexpr ( TnsSize == 3 )
                        {
                                switch ( mode )
                                {
                                        case 0:
                                                return CwiseProd(matArray[2], matArray[1]);
                                                break;
                                        case 1:
                                                return CwiseProd(matArray[2], matArray[0]);
                                                break;
                                        case 2:
                                                return CwiseProd(matArray[1], matArray[0]);
                                                break;
                                        default:
                                                Matrix resMat;
                                                return resMat.setZero();
                                                break;
                                }
                        }
                        else if constexpr ( TnsSize == 4 )
                        {
                                switch ( mode )
                                {
                                        case 0:
                                                return CwiseProd(matArray[3], matArray[2], matArray[1]);
                                                break;
                                        case 1:
                                                return CwiseProd(matArray[3], matArray[2], matArray[0]);
                                                break;
                                        case 2:
                                                return CwiseProd(matArray[3], matArray[1], matArray[0]);
                                                break;
                                        case 3:
                                                return CwiseProd(matArray[2], matArray[1], matArray[0]);
                                                break;
                                        default:
```

```
						Matrix resMat;
						return resMat.setZero();
						break;
				}
			}
			else if constexpr ( TnsSize == 5 )
			{
				switch ( mode )
				{
					case 0:
						return CwiseProd(matArray[4], matArray[3], matArray[2], matArray[1]);
						break;
					case 1:
						return CwiseProd(matArray[4], matArray[3], matArray[2], matArray[0]);
						break;
					case 2:
						return CwiseProd(matArray[4], matArray[3], matArray[1], matArray[0]);
						break;
					case 3:
						return CwiseProd(matArray[4], matArray[2], matArray[1], matArray[0]);
						break;
					case 4:
						return CwiseProd(matArray[3], matArray[2], matArray[1], matArray[0]);
						break;
					default:
						Matrix resMat;
						return resMat.setZero();
						break;
				}
			}
			else if constexpr ( TnsSize == 6 )
			{
				switch ( mode )
				{
					case 0:
						return CwiseProd(matArray[5], matArray[4], matArray[3], matArray[2], matArray[1]);
						break;
					case 1:
						return CwiseProd(matArray[5], matArray[4], matArray[3], matArray[2], matArray[0]);
						break;
					case 2:
						return CwiseProd(matArray[5], matArray[4], matArray[3], matArray[1], matArray[0]);
						break;
					case 3:
						return CwiseProd(matArray[5], matArray[4], matArray[2], matArray[1], matArray[0]);
						break;
					case 4:
						return CwiseProd(matArray[5], matArray[3], matArray[2], matArray[1], matArray[0]);
						break;
					case 5:
						return CwiseProd(matArray[4], matArray[3], matArray[2], matArray[1], matArray[0]);
						break;
					default:
						Matrix resMat;
						return resMat.setZero();
						break;
				}
			}
			else if constexpr ( TnsSize == 7 )
			{
				switch ( mode )
				{
					case 0:
						return CwiseProd(matArray[6], matArray[5], matArray[4], matArray[3], matArray[2], matArray[1]);
						break;
					case 1:
						return CwiseProd(matArray[6], matArray[5], matArray[4], matArray[3], matArray[2], matArray[0]);
						break;
					case 2:
						return CwiseProd(matArray[6], matArray[5], matArray[4], matArray[3], matArray[1], matArray[0]);
						break;
					case 3:
						return CwiseProd(matArray[6], matArray[5], matArray[4],
```

```
matArray[2], matArray[1], matArray[0]);
                                        break;
                                case 4:
                                        return CwiseProd(matArray[6], matArray[5], matArray[3],
matArray[2], matArray[1], matArray[0]);
                                        break;
                                case 5:
                                        return CwiseProd(matArray[6], matArray[4], matArray[3],
matArray[2], matArray[1], matArray[0]);
                                        break;
                                case 6:
                                        return CwiseProd(matArray[5], matArray[4], matArray[3],
matArray[2], matArray[1], matArray[0]);
                                        break;
                                default:
                                        Matrix resMat;
                                        return resMat.setZero();
                                        break;
                        }
                }
                else if constexpr ( TnsSize == 8 )
                {
                        switch ( mode )
                        {
                                case 0:
                                        return CwiseProd(matArray[7], matArray[6], matArray[5],
matArray[4], matArray[3], matArray[2], matArray[1]);
                                        break;
                                case 1:
                                        return CwiseProd(matArray[7], matArray[6], matArray[5],
matArray[4], matArray[3], matArray[2], matArray[0]);
                                        break;
                                case 2:
                                        return CwiseProd(matArray[7], matArray[6], matArray[5],
matArray[4], matArray[3], matArray[1], matArray[0]);
                                        break;
                                case 3:
                                        return CwiseProd(matArray[7], matArray[6], matArray[5],
matArray[4], matArray[2], matArray[1], matArray[0]);
                                        break;
                                case 4:
                                        return CwiseProd(matArray[7], matArray[6], matArray[5],
matArray[3], matArray[2], matArray[1], matArray[0]);
                                        break;
                                case 5:
                                        return CwiseProd(matArray[7], matArray[6], matArray[4],
matArray[3], matArray[2], matArray[1], matArray[0]);
                                        break;
                                case 6:
                                        return CwiseProd(matArray[7], matArray[5], matArray[4],
matArray[3], matArray[2], matArray[1], matArray[0]);
                                        break;
                                case 7:
                                        return CwiseProd(matArray[6], matArray[5], matArray[4],
matArray[3], matArray[2], matArray[1], matArray[0]);
                                        break;
                                default:
                                        Matrix resMat;
                                        return resMat.setZero();
                                        break;
                        }
                }

        }

    } // end namespace v1

} // end namespace partensor

#endif // end of PARTENSOR_PARTIAL_CWISE_PROD_HPP
```

8.55 PartialKhatriRao.hpp File Reference

```
#include "KhatriRao.hpp"
```

Functions

- template<typename Policy , typename MatrixArray_ >
  auto PartialKhatriRao (Policy const &execution_policy, MatrixArray_ const &matArray, std::size_t const mode)

Partial `KhatriRao` implementation.

8.55.1 Detailed Description

Implementation for partial computation of Khatri-Rao product. Makes use of `KhatriRao` function from `KhatriRao.hpp`.

Warning

The implementation supports only this operation, when the number of Matrices of `Matrix` type are in range of [3-8].

8.55.2 Function Documentation

8.55.2.1 PartialKhatriRao()

```
auto partensor::v1::PartialKhatriRao (
            Policy const & execution_policy,
            MatrixArray_ const & matArray,
            std::size_t const mode )
```

Partial `KhatriRao` implementation.

Computes the Khatri-Rao product of matrices of `Matrix` type contained in an array container, excluding the one specified in `mode`.

An `execution_policy` can be applied as in `KhatriRao` function. Either `execution::seq` (sequential) or `execution::par` (parallel). The default value is sequential.

Template Parameters

Policy	Type of `Execution Policy` (sequential, parallel).
MatrixArray	An array container type.

Parameters

execution_policy	[in] Which `Policy` to execute.
matArray	[in] An array containing matrices to use in this operation.
mode	[in] Id of matrix to exclude from the Khatri-Rao product.

Returns

An `Eigen` Matrix with the result.

Note

The function is supported only, when the size of `matArray` is in range of [3-8].

8.56 PartialKhatriRao.hpp

Go to the documentation of this file.

```
#ifndef DOXYGEN_SHOULD_SKIP_THIS
#endif  // DOXYGEN_SHOULD_SKIP_THIS
/*************************************************************************/
#ifndef PARTENSOR_PARTIAL_KHATRI_RAO_HPP
#define PARTENSOR_PARTIAL_KHATRI_RAO_HPP

#include "KhatriRao.hpp"

namespace partensor
{
        inline namespace v1 {
                template <typename Policy, typename MatrixArray_>
                auto PartialKhatriRao( Policy       const &execution_policy,
                                              MatrixArray_ const &matArray,
                                              std::size_t  const  mode            )
                {
                        using Matrix = typename MatrixArrayTraits<MatrixArray_>::value_type;   // Type of data of @c Eigen Matrix (e.g double, float, ..).

                        constexpr std::size_t TnsSize = MatrixArrayTraits<MatrixArray_>::Size; // Size of matArray.

                        if constexpr ( TnsSize == 3 )
                        {
                                switch ( mode )
                                {
                                        case 0:
                                                return KhatriRao(execution_policy, matArray[2], matArray[1]);
                                                break;
                                        case 1:
                                                return KhatriRao(execution_policy, matArray[2], matArray[0]);
                                                break;
                                        case 2:
                                                return KhatriRao(execution_policy, matArray[1], matArray[0]);
                                                break;
                                        default:
                                                Matrix resMat;
                                                return resMat.setZero();
                                                break;
                                }
                        }
                        else if constexpr ( TnsSize == 4 )
                        {
                                switch ( mode )
                                {
                                        case 0:
                                                return KhatriRao(execution_policy, matArray[3], matArray[2], matArray[1]);
                                                break;
                                        case 1:
                                                return KhatriRao(execution_policy, matArray[3], matArray[2], matArray[0]);
                                                break;
                                        case 2:
                                                return KhatriRao(execution_policy, matArray[3], matArray[1], matArray[0]);
                                                break;
                                         case 3:
                                                 return KhatriRao(execution_policy, matArray[2], matArray[1], matArray[0]);
                                                 break;
                                         default:
                                                 Matrix resMat;
                                                 return resMat.setZero();
                                                 break;
                                 }
                         }
                         else if constexpr ( TnsSize == 5 )
                         {
                                 switch ( mode )
                                 {
                                         case 0:
                                                 return KhatriRao(execution_policy, matArray[4], matArray[3], matArray[2], matArray[1]);
                                                 break;
                                         case 1:
                                                 return KhatriRao(execution_policy, matArray[4], matArray[3], matArray[2], matArray[0]);
                                                 break;
```

```
                        case 2:
                            return KhatriRao(execution_policy, matArray[4], matArray[3], matArray[1], matArray[0]);
                            break;
                        case 3:
                            return KhatriRao(execution_policy, matArray[4], matArray[2], matArray[1], matArray[0]);
                            break;
                        case 4:
                            return KhatriRao(execution_policy, matArray[3], matArray[2], matArray[1], matArray[0]);
                            break;
                        default:
                            Matrix resMat;
                            return resMat.setZero();
                            break;
                    }
                }
                else if constexpr ( TnsSize == 6 )
                {
                    switch ( mode )
                    {
                        case 0:
                            return KhatriRao(execution_policy, matArray[5], matArray[4], matArray[3], matArray[2], matArray[1]);
                            break;
                        case 1:
                            return KhatriRao(execution_policy, matArray[5], matArray[4], matArray[3], matArray[2], matArray[0]);
                            break;
                        case 2:
                            return KhatriRao(execution_policy, matArray[5], matArray[4], matArray[3], matArray[1], matArray[0]);
                            break;
                        case 3:
                            return KhatriRao(execution_policy, matArray[5], matArray[4], matArray[2], matArray[1], matArray[0]);
                            break;
                        case 4:
                            return KhatriRao(execution_policy, matArray[5], matArray[3], matArray[2], matArray[1], matArray[0]);
                            break;
                        case 5:
                            return KhatriRao(execution_policy, matArray[4], matArray[3], matArray[2], matArray[1], matArray[0]);
                            break;
                        default:
                            Matrix resMat;
                            return resMat.setZero();
                            break;
                    }
                }
                else if constexpr ( TnsSize == 7 )
                {
                    switch ( mode )
                    {
                        case 0:
                            return KhatriRao(execution_policy, matArray[6], matArray[5], matArray[4], matArray[3], matArray[2], matArray[1]);
                            break;
                        case 1:
                            return KhatriRao(execution_policy, matArray[6], matArray[5], matArray[4], matArray[3], matArray[2], matArray[0]);
                            break;
                        case 2:
                            return KhatriRao(execution_policy, matArray[6], matArray[5], matArray[4], matArray[3], matArray[1], matArray[0]);
                            break;
                        case 3:
                            return KhatriRao(execution_policy, matArray[6], matArray[5], matArray[4], matArray[2], matArray[1], matArray[0]);
                            break;
                        case 4:
                            return KhatriRao(execution_policy, matArray[6], matArray[5], matArray[3], matArray[2], matArray[1], matArray[0]);
                            break;
                        case 5:
                            return KhatriRao(execution_policy, matArray[6], matArray[4], matArray[3], matArray[2], matArray[1], matArray[0]);
                            break;
                        case 6:
                            return KhatriRao(execution_policy, matArray[5], matArray[4], matArray[3], matArray[2], matArray[1], matArray[0]);
                            break;
                        default:
                            Matrix resMat;
                            return resMat.setZero();
```

```
                                        break;
                                }
                        }
                        else if constexpr ( TnsSize == 8 )
                        {
                                switch ( mode )
                                {
                                        case 0:
                                                return KhatriRao(execution_policy, matArray[7], matArray[6], matArray[5], matArray[4], matArray[3], matArray[2], matArray[1]);
                                                break;
                                        case 1:
                                                return KhatriRao(execution_policy, matArray[7], matArray[6], matArray[5], matArray[4], matArray[3], matArray[2], matArray[0]);
                                                break;
                                        case 2:
                                                return KhatriRao(execution_policy, matArray[7], matArray[6], matArray[5], matArray[4], matArray[3], matArray[1], matArray[0]);
                                                break;
                                        case 3:
                                                return KhatriRao(execution_policy, matArray[7], matArray[6], matArray[5], matArray[4], matArray[2], matArray[1], matArray[0]);
                                                break;
                                        case 4:
                                                return KhatriRao(execution_policy, matArray[7], matArray[6], matArray[5], matArray[3], matArray[2], matArray[1], matArray[0]);
                                                break;
                                        case 5:
                                                return KhatriRao(execution_policy, matArray[7], matArray[6], matArray[4], matArray[3], matArray[2], matArray[1], matArray[0]);
                                                break;
                                        case 6:
                                                return KhatriRao(execution_policy, matArray[7], matArray[5], matArray[4], matArray[3], matArray[2], matArray[1], matArray[0]);
                                                break;
                                        case 7:
                                                return KhatriRao(execution_policy, matArray[6], matArray[5], matArray[4], matArray[3], matArray[2], matArray[1], matArray[0]);
                                                break;
                                        default:
                                                Matrix resMat;
                                                return resMat.setZero();
                                                break;
                                }
                        }
                }

                template <typename MatrixArray_>
                auto PartialKhatriRao(MatrixArray_ const &matArray, std::size_t const mode)
                {
                        return PartialKhatriRao(execution::seq, matArray, mode);
                }

        } // end namespace v1

} // end namespace partensor

#endif // end of PARTENSOR_PARTIAL_KHATRI_RAO_HPP
```

8.57 ReadWrite.hpp File Reference

```
#include "PARTENSOR_basic.hpp"
#include "boost/interprocess/file_mapping.hpp"
#include "boost/interprocess/mapped_region.hpp"
#include <iostream>
#include <fstream>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
```

Functions

- template<typename Data , typename FileName >
  void read (FileName const &fileName, std::size_t const size, std::size_t const skip, Data &dat)

Read from a file a Struct, in a serial manner.

- template<typename FileName , typename Dimensions >
  void readFMRI_matrix (FileName const &fileName, Dimensions const tnsDims, std::size_t const skip, Matrix &dat)

  Read `FMRI` data and store in a `Matrix` struct.
- template<std::size_t TnsSize, typename FileName >
  void readFMRI_mpi (FileName const &fileName, std::array< int, TnsSize > const &tnsDims, std::array< int, TnsSize > const &local_tnsDims, std::array< int, TnsSize > const &local_skip, Tensor< static_cast< int >(TnsSize)> &dat)

  Read `FMRI` data and store in a `Tensor` struct with the use of MPI.
- template<typename Data , typename FileName >
  void readFMRI_tensor (FileName const &fileName, std::size_t const skip, Data &dat)

  Read `FMRI` data and store in a `Tensor` struct.
- template<std::size_t TnsSize, typename FileName >
  void readTensor (FileName const &fileName, std::array< int, TnsSize > const &tnsDims, std::array< int, TnsSize > const &local_tnsDims, std::array< int, TnsSize > const &local_skip, Tensor< static_cast< int >(TnsSize)> &dat)

  Read and store an `Tensor` struct with the use of MPI.
- template<typename Data , typename FileName >
  void write (Data const &dat, FileName const &fileName, std::size_t const size)

  Write a Struct of data in a file, in a serial manner.
- template<typename FileName >
  void writeToFile_append (double const fvalue, FileName const &fileName)

  Write function, where data written in appending mode.

8.57.1 Detailed Description

A variety of functions for reading/writing from/to files. There are also `read` implementations, in case an `MPI` environment has been established, to be used and distribute the data.

8.57.2 Function Documentation

8.57.2.1 read()

```
void partensor::read (
            FileName const & fileName,
            std::size_t const size,
            std::size_t const skip,
            Data & dat )
```

Read from a file a Struct, in a serial manner.

Use this functions to read from `fileName`, and store `Eigen` data of `Matrix` or `Tensor` type.

Template Parameters

Data	Input data type. It can be either `Matrix` or `Tensor`.
FileName	Input file type. Either `std::string` or `char*`.

Parameters

fileName	[in] Specify the path to the file, where the data are located. Variable can be of `FileName` type.
size	[in] Number of elements to read. (e.g. For a `Matrix` with rows and columns, then the size should be rows∗columns)
skip	[in] Number of elements to skip.
dat	[in,out] Struct where the data will read from file and stored.

Note

`dat` MUST be initialized before the function call called.

8.57.2.2 readFMRI_matrix()

```
void partensor::readFMRI_matrix (
            FileName const & fileName,
            Dimensions const tnsDims,
            std::size_t const skip,
            Matrix & dat )
```

Read `FMRI` data and store in a `Matrix` struct.

Implementation for `FMRI` data, that are stored in files, in a specific way. In each file,a single row is stored, from the 3D data and the pattern of the file names are like : fileName_00000, fileName_00001, etc. Then use `readFMRI_matrix` to read from all files and store the data in a `Matrix`, equivalent to the third order Tensor matricization in first mode.

Template Parameters

FileName	Type of input file, either `std::string` or `char*`.
Dimensions	Array type for Tensor dimensions.

Parameters

fileName	[in] Specify the path to the file, where the data are located. Variable can be of `FileName` type.
tnsDims	[in] `Eigen` or `stl Array` with the lengths of each of Tensor dimension.
skip	[in] Number of elements to skip in `fileName`.
dat	[in,out] Data read from file.

Note

`dat` MUST be initialized before the function call called.

8.57.2.3 readFMRI_mpi()

```
void partensor::readFMRI_mpi (
            FileName const & fileName,
            std::array< int, TnsSize > const & tnsDims,
            std::array< int, TnsSize > const & local_tnsDims,
            std::array< int, TnsSize > const & local_skip,
            Tensor< static_cast< int >(TnsSize)> & dat )
```

Read `FMRI` data and store in a `Tensor` struct with the use of MPI.

Implementation for `FMRI` data, that are stored in files, in a specific way. In each file,a single row is stored, from the 3D data and the pattern of the file names are like : fileName_00000, fileName_00001, etc. Then use `read↩ FMRIFromFiles` in an `MPI` environment, to read from all files and store the data in an `Tensor` with order equal to 3.

Template Parameters

TnsSize	Order of tensor `dat`. Also, the size of the `stl` arrays `tnsDims`, `local_tnsDims` and `local_skip`.
FileName	Input file type. Either `std::string` or `char*`.

Parameters

fileName	[in] Specify the path to the file, where the data are located. Variable can be of `FileName` type.
tnsDims	[in] `stl` array with initial lengths for each Tensor dimension.
local_tnsDims	[in] `stl` array with lengths for each Tensor dimension per processor.
local_skip	[in] `stl` array with number of skip elements per Tensor dimension.
dat	[in,out] Data read from file.

Note

`dat` MUST be initialized before the function call called.

8.57.2.4 readFMRI_tensor()

```
void partensor::readFMRI_tensor (
            FileName const & fileName,
            std::size_t const skip,
            Data & dat )
```

Read `FMRI` data and store in a `Tensor` struct.

Implementation for `FMRI` data, that are stored in files, in a specific way. In each file,a single row is stored, from the 3D data and the pattern of the file names are like : fileName_00000, fileName_00001, etc. Then use `readFMRI↩ _tensor` to read from all files and store the data in a `Tensor` with order
equal to 3.

Template Parameters

Data	Tensor type. Inside function used to extract Tensor DataType and Dimensions.
FileName	Input file type. Either std::string or char*.

Parameters

fileName	[in] Specify the path to the file, where the data are located. Variable can be of FileName type.
skip	[in] Number of elements to skip.
dat	[in,out] Data read from file.

Note

dat MUST be initialized before the function call called.

8.57.2.5 readTensor()

```
void partensor::readTensor (
            FileName const & fileName,
            std::array< int, TnsSize > const & tnsDims,
            std::array< int, TnsSize > const & local_tnsDims,
            std::array< int, TnsSize > const & local_skip,
            Tensor< static_cast< int >(TnsSize)> & dat )
```

Read and store an Tensor struct with the use of MPI.

When the data of a Tensor are stored in a fileName, and an MPI environent has been set, then readTensor can be used to extract those data from the file. Passing the correct values in arguments : local_tnsDims and local_skip for each processor, then a sub-tensor is distributed to each one, in dat.

Template Parameters

TnsSize	Order of tensor dat. Also, the size of the stl arrays tnsDims, local_tnsDims and local_skip.
FileName	Type of input file, either std::string or char*.

Parameters

fileName	[in] Specify the path to the file, where the data are located. Variable can be of FileName type.
tnsDims	[in] stl array with initial lengths for each Tensor dimension.
local_tnsDims	[in] stl array with lengths for each Tensor dimension per processor.
local_skip	[in] stl array containing where to start for each Tensor dimension, computed in offset_calculation.
dat	[in,out] Data read from file.

Note

`dat` MUST be initialized before the function call called.

8.57.2.6 write()

```
void partensor::write (
            Data const & dat,
            FileName const & fileName,
            std::size_t const size )
```

Write a Struct of data in a file, in a serial manner.

If the data are stored in a struct of data, then this function can be used to write them in a file.

Template Parameters

Data	Input data type. It can be either `Matrix` or `Tensor`.
FileName	Input file type. Either `std::string` or `char*`.

Parameters

dat	[in] `Data` struct that will be written in file.
fileName	[in] Specify the path to the file, where the data will be written.
size	[in] Number of elements to write. (e.g. For a Matrix with rows and columns then the size should be rows*columns)

8.57.2.7 writeToFile_append()

```
void partensor::writeToFile_append (
            double const fvalue,
            FileName const & fileName )
```

Write function, where data written in appending mode.

Use `stl ofstream` to write to a file a single quantity.

Template Parameters

FileName	Type of input file, either `std::string` or `char*`.

Parameters

fvalue	[in] Value that will be written in file.
fileName	[in] Path to the file.

8.58 ReadWrite.hpp

Go to the documentation of this file.

```
#ifndef DOXYGEN_SHOULD_SKIP_THIS
#endif  // DOXYGEN_SHOULD_SKIP_THIS
/**********************************************************************/
#ifndef PARTENSOR_READ_WRITE_HPP
#define PARTENSOR_READ_WRITE_HPP

#include "PARTENSOR_basic.hpp"
#include "boost/interprocess/file_mapping.hpp"
#include "boost/interprocess/mapped_region.hpp"

#include <iostream>
#include <fstream>
// Open Function Sys Call
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
// Read-Write Functions Sys Call
#include <unistd.h>

namespace partensor {

        #ifndef DOXYGEN_SHOULD_SKIP_THIS
        template<typename T>
        inline constexpr bool is_c_str = std::is_same<char const *, typename std::decay<T>::type>::value
    ||
                                                                  std::is_same<char *, typename
    std::decay<T>::type>::value;
        #endif // end of DOXYGEN_SHOULD_SKIP_THIS

        // =====================================================
        // ================== WRITE FUNCTIONS ==================
        // =====================================================

        template<typename FileName>
        void writeToFile_append( double   const  fvalue,
                                 FileName const &fileName )
        {
        std::ofstream os;
                os.exceptions(std::ofstream::badbit | std::ofstream::failbit);
                try
                {
                        os.open(fileName, std::ios::app);
                        os « fvalue «"\n";
                        os.close();
                }
        catch (std::ofstream::failure const &ex)
                {
                        std::cerr « "Exception opening/writing/closing file: " « fileName « std::endl;
                }
        }

        #ifndef DOXYGEN_SHOULD_SKIP_THIS
         template<typename Data, typename FileName>
         void writeToFile_cppStream( Data                const &dat,
                                                     FileName    const &fileName,
                                                     std::size_t const  size)
         {
                 std::ofstream os;
                 os.exceptions(std::ofstream::badbit | std::ofstream::failbit);
                 try
                 {
                         os.open(fileName, std::ios::binary | std::ios::trunc);

                         std::size_t count = 0;
                         if constexpr (is_matrix<Data>)
                         {
                                 using DataType  = typename MatrixTraits<Data>::DataType;
                                 Matrix dat_T;
                                 dat_T           = dat.transpose();
                                 count           = size*sizeof(DataType);
                                 os.write(reinterpret_cast<char*>(dat_T.data()), count);

                         }
                         else
                         {
                                 using DataType  = typename TensorTraits<Data>::DataType;
                                 count           = size*sizeof(DataType);
                                 os.write((char*)dat.data(), count);
                         }
                         os.close();
                 }
                 catch (std::ofstream::failure const &ex)
```

```
        {
            std::cerr « "Exception opening/writing/closing file: " « fileName « std::endl;
        }
    }
    #endif // end of DOXYGEN_SHOULD_SKIP_THIS

    #ifndef DOXYGEN_SHOULD_SKIP_THIS
    template<typename Data, typename FileName>
    void writeToFile_cStream( Data        const &dat,
                 FileName        const &fileName,
                             std::size_t const  size     )
    {
        try
        {
            FILE *of;
            if constexpr (! is_c_str<FileName>)
            {
                of = fopen(fileName.c_str(), "wb");
            }
            else
            {
                of = fopen(fileName, "wb");
            }

            std::size_t fw = 0;
            if constexpr (is_matrix<Data>)
            {
                using DataType = typename MatrixTraits<Data>::DataType;
                Matrix dat_T;
                dat_T = dat.transpose();
                fw    = fwrite(dat_T.data(), sizeof(DataType), size, of);
            }
            else
            {
                using DataType = typename TensorTraits<Data>::DataType;
                fw = fwrite(dat.data(), sizeof(DataType), size, of);
            }
            if(fw != size)
                    throw 0;

            fclose(of);
        }
        catch(...)
        {
            std::cerr « "Exception opening/writing/closing file: " « fileName « std::endl;
    }
    }
    #endif // end of DOXYGEN_SHOULD_SKIP_THIS

    #ifndef DOXYGEN_SHOULD_SKIP_THIS
    template<typename Data, typename FileName>
    void writeToFile_sysCalls( Data            const &dat,
                             FileName    const &fileName,
                             std::size_t const  size     )
    {
        // equal to open() with flags = O_CREAT|O_WRONLY|O_TRUNC
        // S_IRWXU - user (file owner) has write only permission
        try
        {
            int fileDescriptor;
            if constexpr (! is_c_str<FileName>)
            {
                fileDescriptor = creat(fileName.c_str(), S_IWUSR);
            }
            else
            {
                fileDescriptor = creat(fileName, S_IWUSR);
            }
            if(fileDescriptor > 0)
            {
                std::size_t count = 0;
                ssize_t     writeToFile;
                if constexpr (is_matrix<Data>)
                {
                    using DataType = typename MatrixTraits<Data>::DataType;
                    Matrix dat_T;
                    dat_T         = dat.transpose();
                    count            = size*sizeof(DataType);
                writeToFile    = write(fileDescriptor, dat_T.data(), count);
                }
                else
                {
                    using DataType = typename TensorTraits<Data>::DataType;
                    count         = size*sizeof(DataType);
                    writeToFile    = write(fileDescriptor, dat.data(), count);
                }
                if (writeToFile <= 0)
```

```
                                {
                                        throw -1;
                                }
                        }
                        if(close(fileDescriptor) < 0)
                        {
                                throw 0;
                        }
                }
                catch(...)
                {
                        std::cerr « "Exception opening/writing/closing file: " « fileName « std::endl;
                }
        }
        #endif // end of DOXYGEN_SHOULD_SKIP_THIS

    template<typename Data, typename FileName>
    void write( Data                const &dat,
                        FileName     const &fileName,
                        std::size_t const  size      )
        {
                writeToFile_cppStream(dat, fileName, size);
        }

        // =====================================================
        // =================== READ FUNCTIONS ==================
        // =====================================================

        #ifndef DOXYGEN_SHOULD_SKIP_THIS
        template<typename Data, typename FileName>
        void readFromFile_cppStream( FileName    const &fileName,
                                                                 std::size_t const  size,
                                                                 std::size_t const  skip,
                                                                 Data                        &dat       )
        {
                std::ifstream is;
                is.exceptions(std::ifstream::badbit | std::ifstream::failbit);
                try
                {
                        is.open(fileName, std::ios::binary);
                        std::size_t count = 0;
                        if constexpr (is_matrix<Data>)
                        {
                                using DataType = typename MatrixTraits<Data>::DataType;
                                Matrix dat_T(dat.cols(), dat.rows());
                                count              = size*sizeof(DataType);
                                is.ignore(skip*sizeof(DataType));  // where to start
                                is.read(reinterpret_cast<char *>(dat_T.data()), count);
                                dat = dat_T.transpose();
                        }
                        else
                        {
                                using DataType = typename TensorTraits<Data>::DataType;
                                count              = size*sizeof(DataType);
                                is.ignore(skip*sizeof(DataType));  // where to start
                                is.read(reinterpret_cast<char*>(dat.data()), count);
                        }
                        is.close();
        }
                catch (std::ifstream::failure const &ex)
                {
                        std::cerr « "Exception opening/reading/closing file: " « fileName « std::endl;
                }
        }
        #endif // end of DOXYGEN_SHOULD_SKIP_THIS

        #ifndef DOXYGEN_SHOULD_SKIP_THIS
        template<typename Data, typename FileName>
        void readFromFile_cStream( FileName    const &fileName,
                                                           std::size_t const  size,
                                                           std::size_t const  skip,
                                                           Data              &dat      )
        {
                try
                {
                        FILE *inputStream;
                        if constexpr (! is_c_str<FileName>)
                        {
                                inputStream = fopen(fileName.c_str(), "rb");
                        }
                        else
                        {
                                inputStream = fopen(fileName, "rb");
                        }
                        std::size_t fr = 0;
                        if constexpr (is_matrix<Data>)
                        {
```

```
                        using DataType = typename MatrixTraits<Data>::DataType;
                        fseek(inputStream, skip*sizeof(DataType), SEEK_SET); // where to start

                        Matrix dat_T(dat.cols(), dat.rows());
                fr  = fread(dat_T.data(), sizeof(DataType), size, inputStream);
                        dat = dat_T.transpose();
                }
                else
                {
                        using DataType = typename TensorTraits<Data>::DataType;
                        fseek(inputStream, skip*sizeof(DataType), SEEK_SET); // where to start
                fr = fread(dat.data(), sizeof(DataType), size, inputStream);
                }
                if(fr!=size)
                                {
                        throw 0;
                }
                fclose(inputStream);
        }
        catch(...)
        {
                std::cerr « "Exception opening/writing/closing file: " « fileName « std::endl;
        }
}
#endif // end of DOXYGEN_SHOULD_SKIP_THIS

#ifndef DOXYGEN_SHOULD_SKIP_THIS
template<typename Data, typename FileName>
void readFromFile_sysCalls( FileName    const &fileName,
                                                        std::size_t const  size,
                                                        off_t       const  skip,
                                                        Data               &dat      )
{
        // S_IRWXU - user (file owner) has read only permission
        try
        {
                int fileDescriptor;
                if constexpr (! is_c_str<FileName>)
                {
                        fileDescriptor = open(fileName.c_str(), S_IRUSR);
                }
                else
                {
                        fileDescriptor = open(fileName, S_IRUSR);
                }
                if(fileDescriptor > 0)
                {
                        std::size_t  count = 0;
                        off_t        seek  = 0;
                        ssize_t      readFromFile;
                        if constexpr (is_matrix<Data>)
                        {
                                using DataType = typename MatrixTraits<Data>::DataType;
                                count              = size*sizeof(DataType);
                                seek               = lseek(fileDescriptor,
    skip*sizeof(DataType), SEEK_SET);

                                Matrix dat_T(dat.cols(), dat.rows());
                                readFromFile = read(fileDescriptor, dat_T.data(), count);
                                dat          = dat_T.transpose();
                        }
                        else
                        {
                                using DataType = typename TensorTraits<Data>::DataType;
                                count              = size*sizeof(DataType);
                                seek               = lseek(fileDescriptor,
    skip*sizeof(DataType), SEEK_SET);
                                readFromFile   = read(fileDescriptor, dat.data(), count);
                        }

                        if(seek == -1 || readFromFile <= 0)
                        {
                                throw -1;
                        }
                }
                if(close(fileDescriptor) < 0)
                {
                        throw 0;
                }
        }
        catch(...)
        {
                std::cerr « "Exception opening/reading/closing file: " « fileName « std::endl;
        }
}
#endif // end of DOXYGEN_SHOULD_SKIP_THIS

```

```
#ifndef DOXYGEN_SHOULD_SKIP_THIS
template<typename Data, typename FileName>
void readFromFile_memMap( FileName const &fileName,
                          Data           &dat      )
{
        try
        {
                boost::interprocess::mode_t        mode = boost::interprocess::read_only;
        boost::interprocess::file_mapping  fm;
                if constexpr (is_c_str<FileName>)
                {
                        fm = boost::interprocess::file_mapping{fileName, mode};
                }
                else
                {
                        fm = boost::interprocess::file_mapping{fileName.c_str(), mode};
                }
        boost::interprocess::mapped_region region(fm, mode, 0, 0);
                if constexpr (is_matrix<Data>)
                {
                        using DataType = typename MatrixTraits<Data>::DataType;
                        DataType* addr = static_cast<DataType*>(region.get_address());

                        Matrix dat_T(dat.cols(), dat.rows());
                dat_T               = Eigen::Map<Data>(addr, dat.cols(), dat.rows());
                        dat             = dat_T.transpose();
                }
                else
                {
                        using DataType = typename TensorTraits<Data>::DataType;
                        using Dimensions = typename TensorTraits<Data>::Dimensions;
                        const Dimensions tnsDims = dat.dimensions();
                        DataType* addr = static_cast<DataType*>(region.get_address());
                dat                                             = Eigen::TensorMap<Data>(addr, tnsDims);
                }
        }
        catch(boost::interprocess::interprocess_exception &ex)
        {
                std::cerr « ex.what() « std::endl;
        }
        catch(...)
        {
                std::cerr « "Unknown Exception" « std::endl;
        }
}
#endif // end of DOXYGEN_SHOULD_SKIP_THIS

template<typename Data, typename FileName>
void read( FileName    const &fileName,
           std::size_t const  size,
           std::size_t const  skip,
           Data              &dat      )
{
        readFromFile_cppStream(fileName, size, skip, dat);
}

template<typename FileName, typename Dimensions>
void readFMRI_matrix( FileName    const &fileName,
                      Dimensions  const  tnsDims,
                      std::size_t const  skip,
                      Matrix            &dat      )
{
        std::string prefix              = fileName.substr(0,fileName.size()-4);
        std::string extension           = fileName.substr(fileName.size()-4,50);
        std::string newFileName;
        Matrix      _temp(tnsDims[0],1);

        for(auto fn=0; fn<tnsDims[1] * tnsDims[2]; fn++)
        {
                newFileName = prefix + "_";
                newFileName += std::to_string(fn/10000) + std::to_string(fn/1000) + std::to_string(fn/100);
                newFileName += std::to_string(fn/10) + std::to_string(fn%10) + extension;

                readFromFile_cppStream(newFileName, tnsDims[0], skip, _temp);
                dat.col(fn) = _temp;
        }
}

template<typename Data, typename FileName>
void readFMRI_tensor( FileName    const &fileName,
                      std::size_t const  skip,
                      Data              &dat      )
```

```
    {
        using Dimensions = typename TensorTraits<Data>::Dimensions;

        const Dimensions&      tnsDims         = dat.dimensions();
        std::string                 prefix              = fileName.substr(0,fileName.size()-4);
        std::string                 extension           = fileName.substr(fileName.size()-4,50);
        std::string                 newFileName;
        int                                 fn                          = 0;
        Tensor<1> _temp(tnsDims[0]);

        for(int i=0; i<tnsDims[2]; i++)
        {
            for(int j=0; j<tnsDims[1]; j++, fn++)
            {
                newFileName = prefix + "_";
                newFileName += std::to_string(fn/10000) + std::to_string(fn/1000) + std::to_string(fn/100);
                newFileName += std::to_string(fn/10) + std::to_string(fn%10) + extension;

                readFromFile_cppStream(newFileName, tnsDims[0], skip, _temp);
                dat.chip(i,2).chip(j,1) = _temp;
            }
        }
    }

    // ==============================================================
    // =================== READ FUNCTIONS FOR MPI ==================
    // ==============================================================

    #ifndef DOXYGEN_SHOULD_SKIP_THIS
    template<std::size_t TnsSize>
    void offset_calculation( std::array<int, TnsSize> const &dimensions,
                                                       std::array<int, TnsSize> const &skip,
                                                       std::array<int, TnsSize>               &offset
  )
    {
        for (int i=TnsSize-1; i>1; i--)
        {
            // How many cubes to skip
            offset[i] = std::accumulate(dimensions.begin(), dimensions.end()-(TnsSize-i), 1, std::multiplies<int>()) * skip[i];
        }
        offset[1] =  dimensions[0] * skip[1]; // how many matrices to skip
        offset[0] =  skip[0];                        // how many rows to skip
    }
    #endif // end of DOXYGEN_SHOULD_SKIP_THIS

    #ifndef DOXYGEN_SHOULD_SKIP_THIS
    template<std::size_t idx, std::size_t TnsSize>
    void readTensor_util( std::array<int, TnsSize> const    &tnsDims,
                                              std::array<int, TnsSize> const    &local_tnsDims,
                                              std::array<int, TnsSize> const    &offset,
                                              std::ifstream                        &is,
                                              Tensor<static_cast<int>(TnsSize)> &dat           )
    {
        using DT = typename TensorTraits<Tensor<static_cast<int>(TnsSize)»::DataType;
        if constexpr (idx>0)
        {
            // e.g. I*J*K in 4D tensor
            int dim_product = std::accumulate(tnsDims.begin(), tnsDims.begin()+idx, 1, std::multiplies<int>());
            // ignore
            is.ignore(offset[idx]*sizeof(DT));

            for (int i=0; i<local_tnsDims[idx]; i++)
            {
                readTensor_util<idx-1, TnsSize>(tnsDims, local_tnsDims, offset, is, dat);
            }
            is.ignore( dim_product*(tnsDims[idx] - local_tnsDims[idx] - offset[idx])*sizeof(DT) );
        }
        else
        {
            Tensor<1> tns_col(local_tnsDims[0]);
            // ignore offset[0] rows
            is.ignore(offset[0]*sizeof(DT));
            is.read(reinterpret_cast<char*>(tns_col.data()), local_tnsDims[0]*sizeof(DT));
            // use tensor indexing for reading from file
            for (int i=0; i<local_tnsDims[0]; i++)
            {
                static int idx_read = 0;
                dat(idx_read) = tns_col(i);

```

```
                                idx_read++;
                        }
                        // ignore remaining column elements
                        is.ignore( (tnsDims[0] - local_tnsDims[0] - offset[0])*sizeof(DT) );
                }
        }
        #endif // end of DOXYGEN_SHOULD_SKIP_THIS

        template<std::size_t TnsSize, typename FileName>
        void readTensor( FileName                                   const &fileName,
                         std::array<int, TnsSize>          const &tnsDims,
                                        std::array<int, TnsSize>          const &local_tnsDims,
                                        std::array<int, TnsSize>          const &local_skip,
                                        Tensor<static_cast<int>(TnsSize)>            &dat           )
        {
                std::ifstream  is;
                try
                {
                        is.open(fileName, std::ios::binary);
                        std::array<int, TnsSize> offset;
                        offset_calculation( tnsDims, local_skip, offset );
                        readTensor_util<TnsSize-1, TnsSize>( tnsDims, local_tnsDims, offset, is, dat );

                        is.close();
                }
                catch (std::ofstream::failure const &ex)
                {
                        std::cerr « "Exception opening/reading/closing file: " « fileName « std::endl;
                }
        }

        template<std::size_t TnsSize, typename FileName>
        void readFMRI_mpi(  FileName                                          const &fileName,
                                            std::array<int, TnsSize>          const &tnsDims,
                                            std::array<int, TnsSize>          const &local_tnsDims,
                                            std::array<int, TnsSize>          const &local_skip,
                                            Tensor<static_cast<int>(TnsSize)>       &dat           )
        {
                std::string                       prefix       = fileName.substr(0,fileName.size()-4);
                std::string                       extension    = fileName.substr(fileName.size()-4,50);
                std::string               newFileName;
                int                                       fn                = local_skip[2] * tnsDims[1];
                Tensor<1> _temp(local_tnsDims[0]);

                for(int i=0; i<local_tnsDims[2]; i++)
                {
                        fn += local_skip[1];
                        for(int j=0; j<local_tnsDims[1]; j++)
                        {
                                newFileName = prefix + "_";
                                newFileName += std::to_string(fn/10000) + std::to_string(fn/1000) + std::to_string(fn/100);
                                newFileName += std::to_string(fn/10) + std::to_string(fn%10) + extension;
                                readFromFile_cppStream(newFileName.c_str(), local_tnsDims[0], local_skip[0], _temp);

                                dat.chip(i,2).chip(j,1) = _temp;
                                fn++;
                        }
                        fn += tnsDims[1] - local_tnsDims[1] - local_skip[1];
                }
    }

} // end namespace partensor

#endif // end of PARTENSOR_READ_WRITE_HPP
```

8.59 temp.hpp

```
template<std::size_t _TnsSize>
double ComputeCostFun(SparseTensor                   const &sparse_tns,
                      std::array<Matrix, _TnsSize> const &factors,
                      std::array<int, _TnsSize>    const &tns_dimensions)
{
    int const rank  = factors[0].cols();
    Matrix temp_1_R(1, rank);
    double temp_1_1 = 0;
    double f_value  = 0;
     std::array<int, _TnsSize> offsets[tns_size - 1];
     offsets[0] = 1;

     for (int j = 1; j < static_cast<int>(_TnsSize) - 1; j++)
```

```
    {
        offsets[j] = offsets[j - 1] * factors[j - 1].rows();
    }

    for (long int i = 0; i < tns_sparse.outerSize(); ++i)
    {
        int row;
        for (SparseTensor::InnerIterator it(tns_sparse, i); it; ++it)
        {
            temp_1_R = factors[tns_size - 1].row(it.col());
            // Select rows of each factor an compute the Hadamard product of the respective row of the Khatri-Rao product, and the row of factor A_N.
            for (int mode_i = static_cast<int>(_TnsSize) - 2; mode_i >= 0; mode_i--)
            {
                row                  = ((it.row()) / offsets[mode_i]) % (tns_dimensions[mode_i]);
                temp_1_R.noalias() = temp_1_R.cwiseProduct(factors[mode_i].row(row));
            }
            temp_1_1 = it.value() - temp_1_R.sum();
            f_value += temp_1_1 * temp_1_1;
        }
    }

    return f_value;
}


template<std::size_t _TnsSize>
void UnconUpdate(std::array<int, _TnsSize>    const &tns_dimensions,
                 SparseMatrix                 const &sparse_tns,
                 double                       const  lambda,
                 int                          const  cur_mode,
                 std::array<Matrix, _TnsSize>        &factors)
{
    // int m = factors[cur_mode].rows();
    int r = factors[cur_mode].cols();

    Matrix eye = lambda * Matrix::Identity(r, r);

    std::array<int, _TnsSize> offsets[_TnsSize - 1];
    offsets[0] = 1;
    for (int j = 1, mode = 0; j < static_cast<int>(_TnsSize) - 1; j++, mode++)
    {
        if (cur_mode == mode)
        {
            mode++;
        }
        offsets[j] = offsets[j - 1] * ((factors[mode]).rows());
    }

    int last_mode = (cur_mode == static_cast<int>(_TnsSize) - 1) ? static_cast<int>(_TnsSize) - 2 : static_cast<int>(_TnsSize) - 1;

    Matrix MTTKRP_row(1, r);
    Matrix temp_RxR(r, r);
    Matrix temp_1_R(1, r);

    // Compute MTTKRP
    for (long int i = 0; i < sparse_tns.outerSize(); ++i)
    {
        MTTKRP_row.setZero();
        temp_RxR.setZero(); // is the Hadamard product of Grammians of the Factors, that correspond to the nnz elements of the Tensor.
        for (SparseTensor::InnerIterator it(sparse_tns, i); it; ++it)
        {
            temp_1_R = Matrix::Ones(1, r);
            int row;
            // Select rows of each factor an compute the respective row of the Khatri-Rao product.
            for (int mode_i = last_mode, kr_counter = static_cast<int>(_TnsSize) - 2; mode_i >= 0 && kr_counter >= 0; mode_i--)
            {
                if (mode_i == cur_mode)
                {
                    continue;
                }
                row      = ((it.row()) / offsets[kr_counter]) % (tns_dimensions[mode_i]);
                temp_1_R = temp_1_R.cwiseProduct(factors[mode_i].row(row));
                kr_counter--;
            }
            // Subtract from the previous row the respective row of W, according to relation (9).
            MTTKRP_row.noalias() += it.value() * temp_1_R;
            temp_RxR.noalias()   += temp_1_R.transpose() * temp_1_R;
            // std::cerr « "ERROR!!! proc_ID = " « my_rank « "\t" « i « "\t" « it.col() « "\n";
        }
        factors[cur_mode].row(i) = MTTKRP_row * (temp_RxR + eye).inverse();
    }
}

```

```
namespace v2 // transposed_v
{
    template<std::size_t _TnsSize>
    void UnconUpdate(std::array<int, _TnsSize>   const &tns_dimensions,
                     SparseMatrix                const &sparse_tns,
                     double                      const  lambda,
                     int                         const  cur_mode,
                     std::array<Matrix, _TnsSize>       &factors)
    {
        // int m = factors[cur_mode].rows();
        int r = factors[cur_mode].rows();

        Matrix eye = lambda * Matrix::Identity(r, r);

        std::array<int, _TnsSize> offsets[_TnsSize - 1];
        offsets[0] = 1;
        for (int j = 1, mode = 0; j < static_cast<int>(_TnsSize) - 1; j++, mode++)
        {
            if (cur_mode == mode)
            {
                mode++;
            }
            offsets[j] = offsets[j - 1] * ((factors[mode]).cols());
        }

        int last_mode = (cur_mode == static_cast<int>(_TnsSize) - 1) ? static_cast<int>(_TnsSize) - 2 : static_cast<int>(_TnsSize) - 1;

        Matrix MTTKRP_col(r, 1);
        Matrix temp_RxR(r, r);
        Matrix temp_R_1(r, 1);

        // Compute MTTKRP
        for (long int i = 0; i < sparse_tns.outerSize(); ++i)
        {
            MTTKRP_col.setZero();
            temp_RxR.setZero(); // is the Hadamard product of Grammians of the Factors, that correspond to the nnz elements of the Tensor.
            for (SparseTensor::InnerIterator it(sparse_tns, i); it; ++it)
            {
                temp_R_1 = Matrix::Ones(r, 1);
                int row;
                // Select rows of each factor an compute the respective row of the Khatri-Rao product.
                for (int mode_i = last_mode, kr_counter = static_cast<int>(_TnsSize) - 2; mode_i >= 0 && kr_counter >= 0; mode_i--)
                {
                    if (mode_i == cur_mode)
                    {
                        continue;
                    }
                    row      = ((it.row()) / offsets[kr_counter]) % (tns_dimensions[mode_i]);
                    temp_R_1 = temp_R_1.cwiseProduct(factors[mode_i].col(row));
                    kr_counter--;
                }
                // Subtract from the previous row the respective row of W, according to relation (9).
                MTTKRP_col.noalias() += it.value() * temp_R_1;
                temp_RxR.noalias()   += temp_R_1.transpose() * temp_R_1;
                // std::cerr « "ERROR!!! proc_ID = " « my_rank « "\t" « i « "\t" « it.col() « "\n";
            }
            factors[cur_mode].col(i) = ((temp_RxR + eye).inverse()) * MTTKRP_col;
        }
    }

template<std::size_t _TnsSize>
double ComputeCostFun(SparseTensor                 const &sparse_tns,
                      std::array<Matrix, _TnsSize> const &factors,
                      std::array<int, _TnsSize>    const &tns_dimensions)
{
    int const rank = factors[0].rows();
    Matrix temp_R_1(rank, 1);
    double temp_1_1 = 0;
    double f_value = 0;
    std::array<int, _TnsSize> offsets[tns_size - 1];

    offsets[0] = 1;
    for (int j = 1; j < static_cast<int>(_TnsSize) - 1; j++)
    {
        offsets[j] = offsets[j - 1] * factors[j - 1].cols();
    }

    for (long int i = 0; i < tns_sparse.outerSize(); ++i)
    {
        int row;
        for (SparseTensor::InnerIterator it(tns_sparse, i); it; ++it)
        {
            temp_R_1 = factors[tns_size - 1].col(it.col());
            // Select rows of each factor an compute the Hadamard product of the respective row of the
```

```
    Khatri-Rao product, and the row of factor A_N.
            for (int mode_i = static_cast<int>(_TnsSize) - 2; mode_i >= 0; mode_i--)
            {
                row                  = ((it.row()) / offsets[mode_i]) % (tns_dimensions[mode_i]);
                temp_R_1.noalias() = temp_R_1.cwiseProduct(factors[mode_i].col(row));
            }
            temp_1_1 = it.value() - temp_R_1.sum();
            f_value += temp_1_1 * temp_1_1;
        }
    }

    return f_value;
}
} // end of namespace v2
```

8.60 Tensor.hpp File Reference

Classes

- struct Factor< FactorType >
- struct MatrixArrayTraits< MA >
- struct MatrixArrayTraits< std::array< T, _Size > >
- struct MatrixTraits< Matrix >
- struct SparseTensorTraits< SparseTensor< _TnsSize > >
- struct TensorTraits< Tensor< _TnsSize > >

Typedefs

- using DefaultDataType = double
- using LongMatrix = Eigen::Matrix< long int, Eigen::Dynamic, Eigen::Dynamic, Eigen::ColMajor >
- using Matrix = Eigen::Matrix< DefaultDataType, Eigen::Dynamic, Eigen::Dynamic, Eigen::ColMajor >
- using SparseMatrix = Eigen::SparseMatrix< DefaultDataType, Eigen::ColMajor, long int >
- template<int _TnsSize>
  using Tensor = Eigen::Tensor< DefaultDataType, _TnsSize >

Variables

- template<typename T >
  constexpr bool is_matrix

8.60.1 Detailed Description

Containts a variety of Types, used in `partensor` library. Implements different wrappers and traits for both `Eigen` Matrix
and `Eigen` Tensor modules.

8.60.2 Typedef Documentation

8.60.2.1 DefaultDataType

```
using DefaultDataType = double
```

Default Data Type for `Eigen` Data through project.

8.60.2.2 LongMatrix

```
using LongMatrix = Eigen::Matrix<long int, Eigen::Dynamic, Eigen::Dynamic, Eigen::ColMajor>
```

Alias for `Eigen` Matrix with rows and columns computed dynamically and the data are long int type. Also the matrix is stored with column major.

8.60.2.3 Matrix

```
using Matrix = Eigen::Matrix<DefaultDataType, Eigen::Dynamic, Eigen::Dynamic, Eigen::ColMajor>
```

Alias for `Eigen` Matrix with rows and columns computed dynamically. Also the matrix is stored with column major.

8.60.2.4 SparseMatrix

```
using SparseMatrix = Eigen::SparseMatrix<DefaultDataType, Eigen::ColMajor, long int>
```

Alias for `Eigen` Sparse Matrix with rows and columns computed dynamically. Also the matrix is stored with column major.

8.60.2.5 Tensor

```
using Tensor = Eigen::Tensor<DefaultDataType, _TnsSize>
```

Alias for `Eigen` Tensor with data type equal to `DefaultDataType` and tensor order equal to template parameter `_TnsSize`.

Template Parameters

_TnsSize	Order of the Tensor

8.60.3 Variable Documentation

8.60.3.1 is_matrix

```
constexpr bool is_matrix  [inline], [constexpr]
```

Checks if input type `T` is equal to `Matrix` type.

Template Parameters

T	Type of data to process.

Returns

`true` if the data type is `Matrix`, otherwise returns `false`.

8.61 Tensor.hpp

Go to the documentation of this file.

```
#ifndef DOXYGEN_SHOULD_SKIP_THIS
#endif  // DOXYGEN_SHOULD_SKIP_THIS
/*********************************************************************/
#ifndef PARTENSOR_TYPES_HPP
#define PARTENSOR_TYPES_HPP

// #include "PARTENSOR_basic.hpp"
// #include "Constants.hpp"

namespace partensor {

  using DefaultDataType = double;

  using Matrix = Eigen::Matrix<DefaultDataType, Eigen::Dynamic, Eigen::Dynamic, Eigen::ColMajor>;

  using LongMatrix = Eigen::Matrix<long int, Eigen::Dynamic, Eigen::Dynamic, Eigen::ColMajor>;

        using SparseMatrix = Eigen::SparseMatrix<DefaultDataType, Eigen::ColMajor, long int>;

  template<int _TnsSize>
  using Tensor = Eigen::Tensor<DefaultDataType, _TnsSize>;

  template <typename Tensor>
  struct TensorTraits;

  template <int _TnsSize>
  struct TensorTraits<Tensor<_TnsSize»
  {
    using DataType     = DefaultDataType;
    using MatrixType   = partensor::Matrix;
    using Dimensions   = typename Tensor<_TnsSize>::Dimensions;
    using Constraints  = std::array<Constraint,_TnsSize>;
    using MatrixArray  = std::array<MatrixType,_TnsSize>;
    using DoubleArray  = std::array<double,_TnsSize>;
    using IntArray     = std::array<int,_TnsSize>;
    static constexpr std::size_t TnsSize = _TnsSize;
  };

  template <typename Tensor>
  using MatrixArray = typename TensorTraits<Tensor>::MatrixArray;

  template <typename Tensor>
  using Constraints = typename TensorTraits<Tensor>::Constraints;

  template <typename Tensor>
  using DoubleArray = typename TensorTraits<Tensor>::DoubleArray;

  template <typename SparseTensor>
  struct SparseTensorTraits;

  template <std::size_t _TnsSize>
  using SparseTensor = std::array<SparseMatrix, _TnsSize>;

  template <std::size_t _TnsSize>
  struct SparseTensorTraits<SparseTensor<_TnsSize» // std::array<SparseMatrix, _TnsSize>
  {
    using DataType           = DefaultDataType;
    using MatrixType         = partensor::Matrix;
    using SparseMatrixType   = partensor::SparseMatrix;
    using LongMatrixType     = partensor::LongMatrix;
    using Dimensions         = std::array<int,_TnsSize>;
    using Constraints        = std::array<Constraint,_TnsSize>;
    using SparseTensor       = std::array<SparseMatrix, _TnsSize>;
    using MatrixArray        = std::array<MatrixType,_TnsSize>;
    using DoubleArray        = std::array<double,_TnsSize>;
```

```
  using IntArray          = std::array<int,_TnsSize>;
  static constexpr std::size_t TnsSize = _TnsSize;
};

template <typename Matrix>
struct MatrixTraits;

template <>
// struct MatrixTraits<Eigen::Matrix<DefaultDataType, Eigen::Dynamic, Eigen::Dynamic, Eigen::ColMajor»
struct MatrixTraits<Matrix>
{
  using DataType   = DefaultDataType;
  using MatrixType = partensor::Matrix;
};

template <typename MA>
struct MatrixArrayTraits
{
  using value_type = typename MA::value_type;
  using array_type = MA;

  static constexpr unsigned Size = std::tuple_size<MA>::value;
};

template <typename T, std::size_t _Size>
struct MatrixArrayTraits<std::array<T,_Size»
{
  using value_type = T;
  using array_type = std::array<T,_Size>;

  static constexpr std::size_t Size = _Size;
};

// template<typename MatrixArray>
// using MatrixArrayType = typename MatrixArrayTraits<MatrixArray>::value_type;

      template<typename T>
      inline constexpr bool is_matrix = std::is_same<Matrix, typename std::decay<T>::type>::value;

template<typename FactorType>
struct Factor
{
   FactorType    factor;
   FactorType    gramian;
   Constraint    constraint;
};

} // end namespace partensor

#endif // end of PARTENSOR_TYPES_HPP
```

8.62 TensorOperations.hpp File Reference

```
#include <iostream>
#include "PARTENSOR_basic.hpp"
```

Functions

- template<std::size_t TnsSize>
  void BalanceDataset (long int const nnz, std::array< int, TnsSize > const tns_dimensions, Matrix const &Ratings_Base_T, std::array< std::vector< long int >, TnsSize > &perm_tns_indices, Matrix &Balanced_↩ Ratings_Base_T)
- template<std::size_t TnsSize>
  void DepermuteFactors (std::array< Matrix, TnsSize > const &permuted_factors, std::array< std::vector< long int >, TnsSize > const &perm_tns_indices, std::array< Matrix, TnsSize > &depermuted_factors)
- template<typename Array_ , typename Tensor_ >
  void IdentityTensorGen (Array_ const tnsDims, Tensor_ &tnsX)

  Creates an identity `Tensor`*.*

- template<typename Dimensions >
  auto matrixToTensor (Matrix const &mtx, Dimensions const tnsDims)

  Change from `Matrix` to an `Tensor` type.
- auto matrixToTensor (Matrix const &mtx, int const dim0, int const dim1)

  Change from `Matrix` to 2D `Tensor` type.
- template<typename Tensor_ >
  double norm (Tensor_ const &tnsX)

  Frobenius norm of a `Tensor`.
- template<std::size_t TnsSize>
  void PermuteFactors (std::array< Matrix, TnsSize > const &depermuted_factors, std::array< std::vector< long int >, TnsSize > const &perm_tns_indices, std::array< Matrix, TnsSize > &permuted_factors_T)
- void PermuteModeN (long int const nnz, int const cur_mode, int const tns_dim, Matrix const &Ratings_←
  Base_T, std::vector< long int > &perm_tns_indices, Matrix &Balanced_Ratings_Base_T)
- template<typename Array_ , typename Tensor_ >
  void RandomTensorGen (Array_ const tnsDims, Tensor_ &tnsX)

  Creates random `Tensor`.
- template<typename Tensor_ >
  double square_norm (Tensor_ const &tnsX)

  Squared Frobenius norm of a `Tensor`.
- template<typename Tensor_ >
  TensorTraits< Tensor_ >::MatrixType tensorToMatrix (Tensor_ const &tnsX, int const rows, int const cols)

  Change from `Tensor` to `Matrix` type.
- template<typename Array_ , typename Tensor_ >
  void ZeroTensorGen (Array_ const tnsDims, Tensor_ &tnsX)

  Creates a zero `Tensor`.

8.62.1 Detailed Description

Contains functions required in `DimTrees.hpp`, but also implementations to be used in case of `Eigen Tensor` module.

8.62.2 Function Documentation

8.62.2.1 BalanceDataset()

```
void partensor::BalanceDataset (
            long int const nnz,
            std::array< int, TnsSize > const tns_dimensions,
            Matrix const & Ratings_Base_T,
            std::array< std::vector< long int >, TnsSize > & perm_tns_indices,
            Matrix & Balanced_Ratings_Base_T )
```

Shuffles (or permutes) the indices of nonzeros in order to distribute them uniformly.

Template Parameters

TnsSize	Tensor Order.

Parameters

nnz	[in] The nonzeros number.
tns_dimensions	[in] Stl array containing the Tensor dimensions, whose length must be same as the Tensor order.
Ratings_Base_T	[in] The input matrix which containts all nonzeros.
perm_tns_indices	[in,out] Stl array containing the Tensor indices (vector), which will pe permuted.
Balanced_Ratings_Base↩_T	[in/out] The ouput matrix which containts all permuted nonzeros.

8.62.2.2 DepermuteFactors()

```
void partensor::DepermuteFactors (
            std::array< Matrix, TnsSize > const & permuted_factors,
            std::array< std::vector< long int >, TnsSize > const & perm_tns_indices,
            std::array< Matrix, TnsSize > & depermuted_factors )
```

Depermute rows of factors according to shuffled indices perm_tns_indices.

Template Parameters

TnsSize	Tensor Order.

Parameters

depermuted_factors	[in] The input factors, whose rows are permuted.
perm_tns_indices	[in] Stl array containing the Tensor indices (vector), which will pe permuted.
permuted_factors	[in/out] The output factors, whose rows are depermuted.

8.62.2.3 IdentityTensorGen()

```
void partensor::IdentityTensorGen (
            Array_ const tnsDims,
            Tensor_ & tnsX )
```

Creates an identity `Tensor`.

In case a `Tensor` is declared, but with no dimensions or there is a need to change Tensor's dimensions, `IdentityTensorGen` can be used. In both cases the Tensor's order cannot be changed. Fills `tnsX` as an identity `Tensor`. Meaning that it will have 1-elements in the hyperdiagonal and 0-elements in the rest.

Note

Implementation supports ONLY Tensors with order in range of [2,8].

Template Parameters

Array←→ _	An array container type.
Tensor←→ _	Type(data type and order) of input Tensor.

Parameters

tnsDims	[in] Contains the lengths of each of `tnsX` dimensions.
tnsX	[in,out] `Eigen` Tensor filled with the data.

Note

`tnsX` must be initialized before function call.

8.62.2.4 matrixToTensor() [1/2]

```
auto partensor::matrixToTensor (
            Matrix const & mtx,
            Dimensions const tnsDims )
```

Change from `Matrix` to an `Tensor` type.

Changes the type of an `Matrix` and converts it to an N-dimension `Tensor` type.

Template Parameters

Dimensions	An array container type.

Parameters

mtx	[in] The `Matrix` to be converted.
tnsDims	[in] A `Dimensions` array with the lengths of each of Tensor dimension.

Returns

An N-dimension `Tensor` (`tnsDims`[0],tnsDims[1],...), which type same as of `mtx`.

8.62.2.5 matrixToTensor() [2/2]

```
auto partensor::matrixToTensor (
            Matrix const & mtx,
```

```
            int const dim0,
            int const dim1 )
```

Change from `Matrix` to 2D `Tensor` type.

Changes the type of a `Matrix` and converts it to a 2-dimension `Tensor` type.

Parameters

mtx	[in] The `Matrix` to be converted.
dim0	[in] Number of rows of the resulting `Tensor`.
dim1	[in] Number of columns of the resulting `Tensor`.

Returns

A 2-dimension `Tensor` (`rows`, `cols`), which type same as of `mtx`.

8.62.2.6 norm()

```
double partensor::norm (
            Tensor_ const & tnsX )
```

Frobenius norm of a `Tensor`.

Computes the Frobenius Norm of a `Tensor`.

Template Parameters

Tensor	Type(data type and order) of input Tensor.

Parameters

tnsX	[in] The `Tensor` used for this operation.

Returns

A `double` quantity, with the Frobenius Norm of `tnsX`.

8.62.2.7 PermuteFactors()

```
void partensor::PermuteFactors (
            std::array< Matrix, TnsSize > const & depermuted_factors,
            std::array< std::vector< long int >, TnsSize > const & perm_tns_indices,
            std::array< Matrix, TnsSize > & permuted_factors_T )
```

Permute rows of factors according to shuffled indices perm_tns_indices.

Template Parameters

TnsSize	Tensor Order.

Parameters

depermuted_factors	[in] The input factors.
perm_tns_indices	[in] Stl array containing the Tensor indices (vector), which will pe permuted.
permuted_factors←_T	[in/out] The output factors, whose rows are permuted.

8.62.2.8 PermuteModeN()

```
void partensor::PermuteModeN (
            long int const nnz,
            int const cur_mode,
            int const tns_dim,
            Matrix const & Ratings_Base_T,
            std::vector< long int > & perm_tns_indices,
            Matrix & Balanced_Ratings_Base_T )
```

Shuffles (or permutes) the indices of nonzeros in order to distribute them uniformly.

Parameters

nnz	[in] The nonzeros number.
cur_mode	[in] The current mode.
tns_dim	[in] The current mode tensor dimension.
Ratings_Base_T	[in] The input matrix which containts all nonzeros.
perm_tns_indices	[in,out] Stl array containing the Tensor indices (vector), which will pe permuted.
Balanced_Ratings_Base←_T	[in/out] The ouput matrix which containts all permuted nonzeros.

8.62.2.9 RandomTensorGen()

```
void partensor::RandomTensorGen (
            Array_ const tnsDims,
            Tensor_ & tnsX )
```

Creates random `Tensor`.

In case an `Tensor` is declared, but with no dimensions or there is a need to change Tensor dimensions, `Random←TensorGen` can be used. In both cases the Tensor order cannot be changed. Generates pseudo-random data for `tnsX`, in a uniform distribution.

Note

The data will be in range of [-1,1].

Template Parameters

Array←↩_	An array container type.
Tensor←↩_	Type(data type and order) of input Tensor.

Parameters

tnsDims	[in] Contains the lengths of each of `tnsX` dimensions.
tnsX	[in,out] `Tensor` filled with the data.

Note

`tnsX` must be initialized before function call.

8.62.2.10 square_norm()

```
double partensor::square_norm (
            Tensor_ const & tnsX )
```

Squared Frobenius norm of a `Tensor`.

Computes the Squared Frobenius Norm of a `Tensor`.

Template Parameters

Tensor	Type(data type and order) of input Tensor.

Parameters

tnsX	[in] The `Tensor` used for this operation.

Returns

A `double` quantity, with the Squared Frobenius Norm of `tnsX`.

8.62.2.11 tensorToMatrix()

```
TensorTraits< Tensor_ >::MatrixType partensor::tensorToMatrix (
            Tensor_ const & tnsX,
            int const rows,
            int const cols )
```

Change from `Tensor` to `Matrix` type.

Changes the type of an `Tensor` and converts it to an `Matrix` type.

Template Parameters

Tensor←_	Type(data type and order) of input Tensor.

Parameters

tnsX	[in] The Tensor to be converted.
rows	[in] Number of rows of the resulting Matrix.
cols	[in] Number of columns of the resulting Matrix.

Returns

A Matrix (rows, cols), which type same as of tnsX.

8.62.2.12 ZeroTensorGen()

```
void partensor::ZeroTensorGen (
            Array_ const tnsDims,
            Tensor_ & tnsX )
```

Creates a zero Tensor.

In case a Tensor is declared, but with no dimensions or there is a need to change Tensor dimensions, Zero←TensorGen can be used. In both cases the Tensor order cannot be changed. Fills tnsX with zero elements.

Template Parameters

Array←_	An array container type.
Tensor←_	Type(data type and order) of input Tensor.

Parameters

tnsDims	[in] Contains the lengths of each of tnsX dimensions.
tnsX	[in,out] Tensor filled with the data.

Note

tnsX must be initialized before function call.

8.63 TensorOperations.hpp

Go to the documentation of this file.

```
#ifndef DOXYGEN_SHOULD_SKIP_THIS
#endif  // DOXYGEN_SHOULD_SKIP_THIS
/*************************************************************************/
#ifndef PARTENSOR_TENSOR_OPERATIONS_HPP
#define PARTENSOR_TENSOR_OPERATIONS_HPP

#include <iostream>
#include "PARTENSOR_basic.hpp"

namespace partensor
{

        template<typename Tensor_>
        typename TensorTraits<Tensor_>::MatrixType tensorToMatrix( Tensor_ const &tnsX,
                                                                   int     const  rows,
                     int     const  cols  )
        {
                using  Matrix = typename TensorTraits<Tensor_>::MatrixType;
                return Eigen::Map<const Matrix> (tnsX.data(), rows, cols);
        }

        auto matrixToTensor( Matrix const &mtx,
                             int        const  dim0,
                                                 int     const  dim1 )
        {
          return Eigen::TensorMap<Eigen::Tensor<const DefaultDataType,2»(mtx.data(), {dim0,dim1});
        }

        template<typename Dimensions>
        auto matrixToTensor( Matrix     const &mtx,
                                                 Dimensions const tnsDims )
        {
                static constexpr std::size_t TnsSize = tnsDims.size();
                using tensormap = typename Eigen::TensorMap<Eigen::Tensor<const DefaultDataType,TnsSize»;

        if  constexpr (TnsSize == 3) {
                        return tensormap(mtx.data(), {tnsDims[0],tnsDims[1],tnsDims[2]});
                }
                 else if constexpr (TnsSize == 4) {
                        return tensormap(mtx.data(), {tnsDims[0],tnsDims[1],tnsDims[2],tnsDims[3]});
                 }
                 else if constexpr (TnsSize == 5) {
                        return tensormap(mtx.data(),
    {tnsDims[0],tnsDims[1],tnsDims[2],tnsDims[3],tnsDims[4]});
                 }
                 else if constexpr (TnsSize == 6) {
                        return tensormap(mtx.data(),
    {tnsDims[0],tnsDims[1],tnsDims[2],tnsDims[3],tnsDims[4],tnsDims[5]});
                 }
                 else if constexpr (TnsSize == 7) {
                        return tensormap(mtx.data(),
    {tnsDims[0],tnsDims[1],tnsDims[2],tnsDims[3],tnsDims[4],tnsDims[5],tnsDims[6]});
                 }
                 else {
                        return tensormap(mtx.data(),
    {tnsDims[0],tnsDims[1],tnsDims[2],tnsDims[3],tnsDims[4],tnsDims[5],tnsDims[6],tnsDims[7]});
                 }
        }

        template<typename Tensor_>
        double norm( Tensor_ const &tnsX )
        {
                 Tensor<0> frob_norm_tens = tnsX.square().sum().sqrt();
                 return frob_norm_tens.coeff();
        }

        template<typename Tensor_>
        double square_norm( Tensor_ const &tnsX )
        {
                 Tensor<0> frob_norm_tens = tnsX.square().sum();
                 return frob_norm_tens.coeff();
        }

        #ifndef DOXYGEN_SHOULD_SKIP_THIS
        template<std::size_t _TnsSize>
        void TensorContraction( Tensor<static_cast<int>(_TnsSize)> const &parentTensor,
                                                         Tensor<2>                          const
    &factor,
                                                         int                                const
    contractDim1,
                                                         int                                const
    contractDim2,
                                                         Tensor<static_cast<int>(_TnsSize)>
    &contractionRes )
        {
                std::array<Eigen::IndexPair<int>, 1> product_dims = {
```

```
Eigen::IndexPair<int>(contractDim1, contractDim2) };
                std::array<int, _TnsSize>           shuffles;

                Tensor<static_cast<int>(_TnsSize)>   _temp;

                for(std::size_t i=0; i<_TnsSize; i++) { shuffles[i] = i+1; }
                shuffles[_TnsSize-1] = 0;

                _temp          = factor.contract(parentTensor, product_dims);
                contractionRes = _temp.shuffle(shuffles);
        }
        #endif  // DOXYGEN_SHOULD_SKIP_THIS

        #ifndef DOXYGEN_SHOULD_SKIP_THIS
        template<std::size_t _ParTnsSize, std::size_t _TnsSize>
        void TensorPartialProduct_R(Tensor<static_cast<int>(_ParTnsSize)> const &parentTensor,
                                                                          Tensor<2>
const &factor,
                                                                          int
const  chipDim,
                                                                          int
const  contractDim,
                                                          Tensor<static_cast<int>(_TnsSize)>
*contractionRes )
        {
                static_assert(_ParTnsSize == _TnsSize+1,"Wrong call!");

                const int R = factor.dimension(1);

                std::array<Eigen::IndexPair<int>, 1> product_dims = { Eigen::IndexPair<int>(contractDim,
0) };

                for (int i=0; i<R; i++)
                {
                contractionRes->chip(i,chipDim) =
(parentTensor.chip(i,chipDim)).contract(factor.chip(i,1),product_dims);
                }
        }
        #endif  // DOXYGEN_SHOULD_SKIP_THIS

        template<typename Array_, typename Tensor_>
        void RandomTensorGen(Array_ const tnsDims, Tensor_ &tnsX)
        {
                std::srand((unsigned int) time(NULL)+std::rand());
                tnsX.resize(tnsDims);
                tnsX.template setRandom<Eigen::internal::UniformRandomGenerator<double»();
        }

        template<typename Array_, typename Tensor_>
        void ZeroTensorGen(Array_ const tnsDims, Tensor_ &tnsX)
        {
                tnsX.resize(tnsDims);
                tnsX.setZero();
        }

        template<typename Array_, typename Tensor_>
        void IdentityTensorGen(Array_ const tnsDims, Tensor_ &tnsX)
        {
                static constexpr std::size_t TnsSize = TensorTraits<Tensor_>::TnsSize;

                const int dim0 = tnsDims[0];
                ZeroTensorGen(tnsDims, tnsX);

                if constexpr(TnsSize == 2) {
                        for (int i=0; i<dim0; i++) {
                                tnsX(i,i) = 1;
                        }
                }
                else if constexpr(TnsSize == 3) {
                        for (int i=0; i<dim0; i++) {
                                tnsX(i,i,i) = 1;
                        }
                }
                else if constexpr(TnsSize == 4) {
                        for (int i=0; i<dim0; i++) {
                                tnsX(i,i,i,i) = 1;
                        }
                }
                else if constexpr(TnsSize == 5) {
                        for (int i=0; i<dim0; i++) {
                                tnsX(i,i,i,i,i) = 1;
                        }
                }
                else if constexpr(TnsSize == 6) {
                        for (int i=0; i<dim0; i++) {
                                tnsX(i,i,i,i,i,i) = 1;
                        }
```

```
                }
                else if constexpr(TnsSize == 7) {
                        for (int i=0; i<dim0; i++) {
                                tnsX(i,i,i,i,i,i,i) = 1;
                        }
                }
                else {
                        for (int i=0; i<dim0; i++) {
                                tnsX(i,i,i,i,i,i,i,i) = 1;
                        }
                }
        }

        /* Parallel Version of ReserveSparseTensor */
        template <std::size_t _TnsSize>
        void ReserveSparseTensor(std::array<SparseMatrix, _TnsSize>       &layer_tns_sparse,
                                                         std::vector<std::vector<int>>      const
        &local_tns_dimensions,
                                                         std::array<int,_TnsSize>           const
        &fiber_rank,
                                                         int                                const
        grid_size,
                                                         long int                           const   nnz)
        {
                for (std::size_t i = 0; i < _TnsSize; i++)
                {
                        long int col = 1;
                        for (std::size_t j = 0; j < _TnsSize; j++)
                        {
                                if (j == i)
                                        continue;

                                col = col * local_tns_dimensions[j][fiber_rank[j]];
                        }
                        layer_tns_sparse[i].resize(col, local_tns_dimensions[i][fiber_rank[i]]);

                        layer_tns_sparse[i].reserve((long int)(nnz / grid_size) + 1);
                }
        }

        /* Serial Version of ReserveSparseTensor */
        template<std::size_t _TnsSize>
        void ReserveSparseTensor(std::array<SparseMatrix, _TnsSize>       &tns_sparse,
                                                         std::array<int, _TnsSize>          const
        &tns_dimensions,
                                                         const long int                             nnz)
        {
                for (std::size_t i = 0; i < _TnsSize; i++)
                {
                        long int col = 1;
                        for (std::size_t j = 0; j < _TnsSize; j++)
                        {
                                if (j == i)
                                        continue;

                                col = col * tns_dimensions[j];
                        }

                        tns_sparse[i].resize(col, tns_dimensions[i]);
                        tns_sparse[i].reserve(nnz);
                }
        }

        // Assign nonzeros to the respective layer_tns_sparse subtensor.
        template <std::size_t _TnsSize>
        void FillSparseTensor(std::array<SparseMatrix, _TnsSize>       &tns_sparse,
                                                  long int                             const  nnz,
                                                  Matrix                               const
        &Ratings_Base_T,
                                                  std::array<int, _TnsSize>            const
        &tns_dimensions)
        {
                LongMatrix matr_mapping(static_cast<int>(_TnsSize), static_cast<int>(_TnsSize));
                for (int i = 0; i < static_cast<int>(_TnsSize); i++)
                {
                        for (int j = 0, first = 1, prev = 0; j < static_cast<int>(_TnsSize); j++)
                        {
                                if (j == i)
                                {
                                        matr_mapping(i, j) = 0;
                                        continue;
                                }
                                if (first == 1)
                                {
                                        matr_mapping(i, j) = 1;
                                        first = 0;
                                        prev = j;
```

```
                }
                else
                {
                    matr_mapping(i, j) = matr_mapping(i, prev) * tns_dimensions[prev];
                    prev = j;
                }
            }
        }

        LongMatrix tuple(1, _TnsSize);

        for (long int nnz_k = 0; nnz_k < nnz; nnz_k++)
        {
            for (int column_idx = 0; column_idx < static_cast<int>(_TnsSize); column_idx++)
            {
                tuple(0, column_idx) = static_cast<long int>(Ratings_Base_T(column_idx, nnz_k));
            }

            for (int mode_i = 0; mode_i < static_cast<int>(_TnsSize); mode_i++)
            {
                long int linear_col = ((matr_mapping.row(mode_i)).cwiseProduct(tuple)).sum();
                long int row        = tuple(0, mode_i);

                if (tns_sparse[mode_i].outerSize() < row || tns_sparse[mode_i].innerSize() < linear_col)
                {
                    std::cerr « "error!" « tns_sparse[mode_i].outerSize() « " " « row « " " « tns_sparse[mode_i].innerSize() « " " « linear_col « std::endl;
                }

                if (tns_sparse[mode_i].coeff(linear_col, row) == 0)
                {
                    // Using insert to fill sparse matrix
                    tns_sparse[mode_i].insert(linear_col, row) = Ratings_Base_T(_TnsSize, nnz_k);
                }
            }

        }
    }

    template<std::size_t _TnsSize>
    void FillSparseMatricization(std::array<SparseMatrix, _TnsSize> &tns_sparse,
                                 const long int nnz,
                                 Matrix &Ratings_Base_T,
                                 std::array<int, _TnsSize>    const &tns_dimensions,
                                 const int cur_mode)
    {
        LongMatrix matr_mapping(static_cast<int>(_TnsSize), static_cast<int>(_TnsSize));
        for (int i = 0; i < static_cast<int>(_TnsSize); i++)
        {
            for (int j = 0, first = 1, prev = 0; j < static_cast<int>(_TnsSize); j++)
            {
                if (j == i)
                {
                    matr_mapping(i, j) = 0;
                    continue;
                }
                if (first == 1)
                {
                    matr_mapping(i, j) = 1;
                    first = 0;
                    prev = j;
                }
                else
                {
                    matr_mapping(i, j) = matr_mapping(i, prev) * tns_dimensions[prev];
                    prev = j;
                }
            }
        }

        LongMatrix tuple(1, static_cast<int>(_TnsSize));

        for (int nnz_k = 0; nnz_k < nnz; nnz_k++)
        {
            for (int column_idx = 0; column_idx < static_cast<int>(_TnsSize); column_idx++)
            {
                tuple(0, column_idx) = static_cast<long int>(Ratings_Base_T(column_idx,
```

```
nnz_k));
                        }

                        long linear_col = ((matr_mapping.row(cur_mode)).cwiseProduct(tuple)).sum();

                        long row = tuple(0, cur_mode);

                        if (tns_sparse[cur_mode].outerSize() < row || tns_sparse[cur_mode].innerSize() < linear_col)
                        {
                                std::cerr « "error!" « tns_sparse[cur_mode].outerSize() « " " « row « " " « tns_sparse[cur_mode].innerSize() « " " « linear_col « std::endl;
                        }

                        if (tns_sparse[cur_mode].coeff(linear_col, row) == 0)
                        {
                                // Using insert to fill sparse matrix
                                tns_sparse[cur_mode].insert(linear_col, row) = Ratings_Base_T(static_cast<int>(_TnsSize), nnz_k);
                        }
                }
        }

        template <std::size_t _TnsSize>
        void Dist_NNZ(std::array<SparseMatrix, _TnsSize>        &layer_tns_sparse,
                                long int                        const  nnz,
                                std::vector<std::vector<int»       const &skip_rows,
                                std::array<int,_TnsSize>         const &fiber_rank,
                                Matrix                          const &Ratings_Base_T,
                                std::vector<std::vector<int»       const &local_tns_dimensions)
        {
                LongMatrix matr_mapping(static_cast<int>(_TnsSize), static_cast<int>(_TnsSize));
                for (int i = 0; i < static_cast<int>(_TnsSize); i++)
                {
                        for (int j = 0, first = 1, prev = 0; j < static_cast<int>(_TnsSize); j++)
                        {
                                if (j == i)
                                {
                                        matr_mapping(i, j) = 0;
                                        continue;
                                }
                                if (first == 1)
                                {
                                        matr_mapping(i, j) = 1;
                                        first = 0;
                                        prev = j;
                                }
                                else
                                {
                                        matr_mapping(i, j) = matr_mapping(i, prev) * local_tns_dimensions[prev][fiber_rank[prev]];
                                        prev = j;
                                }
                        }
                }

                long int local_nnz_counter = 0;

                LongMatrix tuple(1, static_cast<int>(_TnsSize));

                for (int nnz_k = 0; nnz_k < nnz; nnz_k++)
                {
                        for (int column_idx = 0, insert_tuple_flag = 0; column_idx < static_cast<int>(_TnsSize) && insert_tuple_flag == column_idx; column_idx++)
                        {
                                if((Ratings_Base_T(column_idx, nnz_k) >= skip_rows[column_idx][fiber_rank[column_idx]]) && (Ratings_Base_T(column_idx, nnz_k) < local_tns_dimensions[column_idx][fiber_rank[column_idx]] + skip_rows[column_idx][fiber_rank[column_idx]]))
                                {
                                        tuple(0, column_idx) = (long int)(Ratings_Base_T(column_idx, nnz_k)) - skip_rows[column_idx][fiber_rank[column_idx]];
                                        insert_tuple_flag++;
                                }
                                else
                                {
                                        break;
                                }
                                if (column_idx == static_cast<int>(_TnsSize) - 1)
                                {
                                        local_nnz_counter++;
                                        for (int mode_i = 0; mode_i < static_cast<int>(_TnsSize); mode_i++)
                                        {
                                                long int linear_col = ((matr_mapping.row(mode_i)).cwiseProduct(tuple)).sum();
                                                long int row = tuple(0, mode_i);
```

```

                        if (layer_tns_sparse[mode_i].coeff(linear_col, row) == 0)
                        {
                            layer_tns_sparse[mode_i].insert(linear_col, row) = Ratings_Base_T(static_cast<int>(_TnsSize), nnz_k);
                        }
                    }
                }
            }
        }
    }

    template <std::size_t _TnsSize>
    void Dist_NNZ_sorted(std::array<SparseMatrix, _TnsSize>         &layer_tns_sparse,
                         long int                                   const  nnz,
                         std::vector<std::vector<int»              const &skip_rows,
                         std::array<int,_TnsSize>                   const &fiber_rank,
                         Matrix                                     const &Ratings_Base_T,
                         std::vector<std::vector<int»              const &local_tns_dimensions,
                         int                                        const  cur_mode)
    {
        LongMatrix matr_mapping(static_cast<int>(_TnsSize), static_cast<int>(_TnsSize));
        for (int i = 0; i < static_cast<int>(_TnsSize); i++)
        {
            for (int j = 0, first = 1, prev = 0; j < static_cast<int>(_TnsSize); j++)
            {
                if (j == i)
                {
                    matr_mapping(i, j) = 0;
                    continue;
                }
                if (first == 1)
                {
                    matr_mapping(i, j) = 1;
                    first = 0;
                    prev = j;
                }
                else
                {
                    matr_mapping(i, j) = matr_mapping(i, prev) * local_tns_dimensions[prev][fiber_rank[prev]];
                    prev = j;
                }
            }
        }

        long int local_nnz_counter = 0;

        LongMatrix tuple(1, static_cast<int>(_TnsSize));

        for (int nnz_k = 0; nnz_k < nnz; nnz_k++)
        {
            for (int column_idx = 0, insert_tuple_flag = 0; column_idx < static_cast<int>(_TnsSize) && insert_tuple_flag == column_idx; column_idx++)
            {
                if((Ratings_Base_T(column_idx, nnz_k) >= skip_rows[column_idx][fiber_rank[column_idx]]) && (Ratings_Base_T(column_idx, nnz_k) < local_tns_dimensions[column_idx][fiber_rank[column_idx]] + skip_rows[column_idx][fiber_rank[column_idx]]))
                {
                    tuple(0, column_idx) = (long int)(Ratings_Base_T(column_idx, nnz_k)) - skip_rows[column_idx][fiber_rank[column_idx]];
                    insert_tuple_flag++;
                }
                else
                {
                    break;
                }
                if (column_idx == static_cast<int>(_TnsSize) - 1)
                {
                    local_nnz_counter++;

                    long int linear_col = ((matr_mapping.row(cur_mode)).cwiseProduct(tuple)).sum();
                    long int row = tuple(0, cur_mode);

                    if (layer_tns_sparse[cur_mode].coeff(linear_col, row) == 0)
                    {
                        layer_tns_sparse[cur_mode].insert(linear_col, row) = Ratings_Base_T(static_cast<int>(_TnsSize), nnz_k);
                    }
                }
            }
        }
```

```
    }

    template <int TnsSize, int mode, typename Type>
    bool SortRows(const std::vector<Type> &v1, const std::vector<Type> &v2)
    {
        std::array<int, TnsSize> sort_direction;
        bool final_expr{false}; // final criterion for sorting
        bool prev_equal{false}; // keep history of equal comparisons between v1,v2

        sort_direction[0] = mode;
        sort_direction[1] = (mode < TnsSize - 1) ? TnsSize - 1 : TnsSize - 2;
        for (int i = 2; i < TnsSize; i++)
        {
            sort_direction[i] = sort_direction[i - 1] - 1;
            if (sort_direction[i] == mode)
            {
                sort_direction[i]--;
            }
        }
        std::array<bool, TnsSize> expr;
        for (int i = 0; i < TnsSize; i++)
        {
            if (i > 0)
            {
                expr[i] = prev_equal && (v1[sort_direction[i]] < v2[sort_direction[i]]);
                final_expr = final_expr || expr[i];
                prev_equal = prev_equal && (v1[sort_direction[i]] == v2[sort_direction[i]]);

                if (final_expr)
                {
                    return final_expr;
                }
            }
            else
            {
                expr[i] = v1[sort_direction[i]] < v2[sort_direction[i]];
                final_expr = final_expr || expr[i];
                prev_equal = (v1[sort_direction[i]] == v2[sort_direction[i]]);
                if (final_expr)
                {
                    return final_expr;
                }
            }
        }

        return final_expr;
    }

    void PermuteModeN(long int              const  nz,
                      int                   const  cur_mode,
                      int                   const  tns_dim,
                      Matrix                const &Ratings_Base_T,
                      std::vector<long int>        &perm_tns_indices,
                      Matrix                       &Balanced_Ratings_Base_T)

    {
        // Copy values
        Balanced_Ratings_Base_T = Ratings_Base_T;

        // Allocate & Initialize permuted dims
        perm_tns_indices.reserve(tns_dim);

        for (int i_i = 0; i_i < tns_dim; i_i++)
        {
            perm_tns_indices.push_back(i_i);
        }

        // Permute dims
        std::random_device rd;
        std::mt19937 g(rd());
        std::shuffle(perm_tns_indices.begin(), perm_tns_indices.end(), g);
        double prev = -1;
        long int idx;
        std::vector<long int>::iterator it;
        long int perm_idx = 0;
        for (long int nnz_i = 0; nnz_i < nnz; nnz_i++)
        {
            if(Ratings_Base_T(cur_mode, nnz_i) != prev)
            {
                idx = static_cast<long int>(Ratings_Base_T(cur_mode, nnz_i));
                it = std::find(perm_tns_indices.begin(), perm_tns_indices.end(), idx);
                perm_idx = it - perm_tns_indices.begin();
                Balanced_Ratings_Base_T(cur_mode, nnz_i) = static_cast<double>(perm_idx);
                prev = Ratings_Base_T(cur_mode, nnz_i);
            }
```

```
                        else
                        {
                                Balanced_Ratings_Base_T(cur_mode, nnz_i) = static_cast<double>(perm_idx);
                                prev = Ratings_Base_T(cur_mode, nnz_i);
                        }
                }
        }

        template <std::size_t TnsSize>
        void BalanceDataset(long int const  nnz,
                                                std::array<int, TnsSize> const  tns_dimensions,
                                                Matrix const &Ratings_Base_T,
                                                std::array<std::vector<long int>, TnsSize> &perm_tns_indices,
                                                Matrix &Balanced_Ratings_Base_T)
        {
                for (int i = 0; i < static_cast<int>(TnsSize); i++)
                {
                        PermuteModeN(nnz, i, tns_dimensions[i], Ratings_Base_T, perm_tns_indices[i], Balanced_Ratings_Base_T);
                }
        }

        template <std::size_t TnsSize>
        void PermuteFactors(std::array<Matrix, TnsSize>                 const &depermuted_factors,
                                                std::array<std::vector<long int>, TnsSize> const &perm_tns_indices,
                                                std::array<Matrix, TnsSize> &permuted_factors_T)
        {
                Matrix temp_permuted_factor;
                for (int i = 0; i < static_cast<int>(TnsSize); i++)
                {
                        temp_permuted_factor = Matrix::Zero(depermuted_factors[i].rows(), depermuted_factors[i].cols());
                        for (int row = 0; row < temp_permuted_factor.rows(); row++)
                        {
                                temp_permuted_factor.row(row) = depermuted_factors[i].row(perm_tns_indices[i][row]);
                        }
                        permuted_factors_T[i] = temp_permuted_factor.transpose();
                }
        }

        template <std::size_t TnsSize>
        void DepermuteFactors(std::array<Matrix, TnsSize>             const &permuted_factors,
                                                std::array<std::vector<long int>, TnsSize> const &perm_tns_indices,
                                                std::array<Matrix, TnsSize> &depermuted_factors)
        {
                for (int i = 0; i < static_cast<int>(TnsSize); i++)
                {
                        depermuted_factors[i] = Matrix::Zero(permuted_factors[i].rows(), permuted_factors[i].cols());
                        for (int row = 0; row < permuted_factors[i].rows(); row++)
                        {
                                depermuted_factors[i].row(perm_tns_indices[i][row]) = permuted_factors[i].row(row);
                        }
                }
        }

} // end namespace partensor

#endif // PARTENSOR_TENSOR_OPERATIONS_HPP
```

8.64 TerminationConditions.hpp File Reference

```
#include "DataGeneration.hpp"
#include "math.h"
```

Classes

- struct Conditions

Functions

- template< typename MatrixArray >
  double error (typename MatrixArrayTraits< MatrixArray >::value_type const &origMatrTns, MatrixArray const &factors, std::size_t const mode)

  Computes the "distance" between the current Tensor and the real one.
- int maxIterations (int const countIter)
- int objectiveValueError (double const newValue, double const pastValue)
- int relativeCostFunction (double const newValue, double const trueValue)
- int relativeError (double const newValue, double const trueValue)

8.64.1 Detailed Description

Containts implementations, that check if specific terminations conditions are satisfied.

8.64.2 Function Documentation

8.64.2.1 error()

```
double partensor::error (
            typename MatrixArrayTraits< MatrixArray >::value_type const & origMatrTns,
            MatrixArray const & factors,
            std::size_t const mode )
```

Computes the "distance" between the current Tensor and the real one.

Calculate the `squaredNorm()` between the real matricized Tensor `origMatrTns` and the Tensor generated from a factorization algorithm.

Template Parameters

MatrixArray	An array container type.

Parameters

origMatrTns	[in] The matricization of the original/true Tensor. It must be in `Eigen` Matrix format.
factors	[in] Contains the factors generated from a factorization algorithm, like `cpd`, `cpdDimTree`, etc.
mode	[in] Specify in which mode, is `origMatrTns` matricized.

Returns

The "distance" between `origMatrTns` and the generated Tensor. If returned value is 0, then the two Tensors are identical.

8.64.2.2 maxIterations()

```
int partensor::maxIterations (
            int const countIter ) [inline]
```

< A class `Conditions` object. Checks if `countIter` is greater that the `max_iter` of `Conditions` class.

Parameters

countIter	[in] Current iteration.

Returns

If `counterIter` does NOT surpass `max_iter` then returns 1, otherwise returns 0.

8.64.2.3 objectiveValueError()

```
int partensor::objectiveValueError (
            double const newValue,
            double const pastValue ) [inline]
```

Checks if the current objective value error has smaller value than the default tolerance `ao_tol`.

Parameters

newValue	[in] Current value.
pastValue	[in] True value.

Returns

If objective value error does NOT surpass `ao_tol` then returns 1, otherwise returns 0.

8.64.2.4 relativeCostFunction()

```
int partensor::relativeCostFunction (
            double const newValue,
            double const trueValue ) [inline]
```

Checks if the current relative cost function has smaller value than the default tolerance `ao_tol`.

Parameters

newValue	[in] Current value.
trueValue	[in] True value.

Returns

If relative cost function does NOT surpass `ao_tol` then returns 1, otherwise returns 0.

8.64.2.5 relativeError()

```
int partensor::relativeError (
            double const newValue,
            double const trueValue )  [inline]
```

Checks if the current relative error has smaller value than the default tolerance `ao_tol`.

Parameters

newValue	[in] Current value.
trueValue	[in] True value.

Returns

If relative error does NOT surpass `ao_tol` then returns 1, otherwise returns 0.

8.65 TerminationConditions.hpp

Go to the documentation of this file.

```
#ifndef DOXYGEN_SHOULD_SKIP_THIS
#endif  // DOXYGEN_SHOULD_SKIP_THIS
/*********************************************************************/
#ifndef PARTENSOR_TERMINATION_CONDITIONS_HPP
#define PARTENSOR_TERMINATION_CONDITIONS_HPP

#include "DataGeneration.hpp"
#include "math.h"

namespace partensor {

    struct Conditions {
      int     const max_iter = 500;          // Maximum Number of iterations
      double const ao_tol    = 1e-2;                              // Tolerance for AO Algorithm
      double const delta_1   = 1e-2;                                    // | Tolerance for Nesterov Algorithm
      double const delta_2   = 1e-2;                                    // |
    };

    static inline Conditions con;
    inline int maxIterations(int const countIter)
    {
      return (countIter >= con.max_iter) ? 1 : 0;
    }

    inline int relativeCostFunction(double const newValue, double const trueValue)
    {
      return (newValue/sqrt(trueValue) <= con.ao_tol) ? 1 : 0;
```

```
    }

    inline int relativeError(double const newValue, double const trueValue)
    {
      return (abs(trueValue-newValue)/trueValue <= con.ao_tol) ? 1 : 0;
    }

    inline int objectiveValueError(double const newValue, double const pastValue)
    {
       return (abs(newValue-pastValue) <= con.ao_tol) ? 1 : 0;
     }

     template<typename MatrixArray>
     double error( typename MatrixArrayTraits<MatrixArray>::value_type const &origMatrTns,
                   MatrixArray                                         const &factors,
                   std::size_t                                         const  mode       )
     {
       using Matrix = typename MatrixArrayTraits<MatrixArray>::value_type;

       Matrix localMatrTns = generateTensor(mode, factors);
       return (origMatrTns - localMatrTns).squaredNorm();
     }

} // end namespace partensor

#endif // PARTENSOR_TERMINATION_CONDITIONS_HPP
```

8.66 Timers.hpp File Reference

```
#include <chrono>
#include "mpi.h"
```

Classes

- struct Timers

Variables

- static Timers timer

8.66.1 Detailed Description

Implements the following time functions,

- `clock`,
- `std::chrono` with both steady and high resolution clock,
- `MPI_Wtime`.

8.66.2 Variable Documentation

8.66.2.1 timer

```
Timers timer  [inline], [static]
```

A class `Timers` object

8.67 Timers.hpp

Go to the documentation of this file.

```
#ifndef DOXYGEN_SHOULD_SKIP_THIS
#endif  // DOXYGEN_SHOULD_SKIP_THIS
/**********************************************************************/
#ifndef PARTENSOR_TIMERS_HPP
#define PARTENSOR_TIMERS_HPP

#include <chrono>
#include "mpi.h"

namespace partensor {

  struct Timers
  {
    using ClockSteady = std::chrono::time_point<std::chrono::steady_clock>;
    using ClockHigh   = std::chrono::time_point<std::chrono::high_resolution_clock>;
    clock_t     cpu_current_time;
    double          cpu_elapsed_time;

    ClockHigh   chrono_high_current_time;
    double      chrono_high_elapsed_time;

    ClockSteady chrono_steady_current_time;
    double      chrono_steady_elapsed_time;

    double      mpi_current_time;
    double          mpi_elapsed_time;

    void startCpuTimer ()
    {
      cpu_current_time = clock();
    }

    double endCpuTimer ()
    {
      auto finish       = clock();
      cpu_elapsed_time = (float)(finish - cpu_current_time)/(float)CLOCKS_PER_SEC ;
      return cpu_elapsed_time;
    }

    void startChronoHighTimer ()
    {
      chrono_high_current_time = std::chrono::high_resolution_clock::now();
    }

    double endChronoHighTimer ()
    {
      std::chrono::duration<double> finish = std::chrono::high_resolution_clock::now() - chrono_high_current_time;
      chrono_high_elapsed_time = finish.count();
      return chrono_high_elapsed_time;
    }

     void startChronoSteadyTimer ()
     {
       chrono_steady_current_time = std::chrono::steady_clock::now();
     }

     double endChronoSteadyTimer ()
     {
       std::chrono::duration<double> finish = std::chrono::steady_clock::now() - chrono_steady_current_time;
       chrono_steady_elapsed_time = finish.count();
       return chrono_steady_elapsed_time;
     }

     void startMpiTimer ()
     {
       mpi_current_time = MPI_Wtime();
     }
```

```
    double endMpiTimer ()
    {
      auto finish = MPI_Wtime();
      mpi_elapsed_time = (finish - mpi_current_time);
      return mpi_elapsed_time;
    }

  };

  static inline Timers timer;
} // end namespace partensor


#endif // end of PARTENSOR_TIMERS_HPP
```

Index