

Partensor

Generated by Doxygen 1.9.3

1 Chapters	1
1.1 Factorization	1
1.1.1 Canonical Polyadic Decomposition	2
1.1.1.1 Sequential Policy	2
1.1.1.2 Parallel Policy with MPI	3
1.1.1.3 Parallel Policy with OpenMP	5
1.1.2 Canonical Polyadic Decomposition with Dimension Tree	7
1.1.2.1 Sequential Policy	7
1.1.2.2 Parallel Policy with MPI	8
1.1.3 Options for ALS Method	10
1.2 Completion	11
1.2.1 General Tensor Completion	11
1.2.1.1 Sequential Policy	11
1.2.1.2 Parallel Policy with MPI	13
1.2.1.3 Parallel Policy with OpenMP	15
1.2.2 General Tensor Completion using Stochastic Methods	17
1.2.2.1 Sequential Policy	17
1.2.2.2 Parallel Policy with MPI	18
1.2.2.3 Parallel Policy with OpenMP	19
1.2.3 Options for GTC	20
1.3 Matrix and Tensor Operations	22
1.3.1 Element-wise Product	22
1.3.1.1 How to use this function	22
1.3.1.2 Comments on the Example	23
1.3.2 Khatri-Rao Product	23
1.3.2.1 How to use this function	23
1.3.2.2 Comments on the Example	24
1.3.3 Kronecker Product	24
1.3.3.1 How to use this function	24
1.3.3.2 Comments on the Example	25
1.3.4 Tensor Matricization	25
1.3.4.1 How to use this function	25
1.3.4.2 Comments on the Example	25
1.3.5 Data Generation	25
1.3.5.1 Randomly Generated Data for Initialized Matrix	26
1.3.5.2 Randomly Generated Data for Initialized Tensor	26
1.3.5.3 Matricization of a Tensor from Factors	26
1.3.5.4 Generation of an Array with Factors	27
1.3.5.5 Generation of a Tensor with Constraints Applied	28
1.4 Read/Write Data from/in Files	28
1.4.1 Write To a File	29
1.4.2 Read From a File	29

1.5 Timers	29
2 Getting Started	31
2.1 How to "install" Partensor?	31
2.2 A Simple Example	32
2.3 Important Notes	33
2.3.1 Log File	33
2.3.2 Python Utility	33
3 Overview	35
3.1 Abstract	35
3.2 Funding	35
3.3 Contributors	35
3.4 References	36
3.5 Links	36
4 Tensor Operations	37
4.1 Data Generation	37
4.1.1 Randomly Generated Data for Initialized Matrix	37
4.1.2 Randomly Generated Data for Initialized Tensor	37
4.1.3 Matricization of a Tensor from Factors	38
4.1.3.1 Matricized Tensor	38
4.1.3.2 Tensor	38
4.1.4 Generation of an Array with Factors	39
4.1.5 Generation of a Tensor with Constraints Applied	40
4.2 Element-wise Product	40
4.2.1 How to use this function	40
4.2.2 Comments on the Example	41
4.3 Khatri-Rao Product	41
4.3.1 How to use this function	41
4.3.2 Comments on the Example	42
4.4 Kronecker Product	42
4.4.1 How to use this function	42
4.4.2 Comments on the Example	43
4.5 Tensor Matricization	43
4.5.1 How to use this function	43
4.5.2 Comments on the Example	44
5 Class Index	45
5.1 Class List	45
6 File Index	47
6.1 File List	47

7 Class Documentation	49
7.1 cartesian_communicator Struct Reference	49
7.1.1 Detailed Description	49
7.1.2 Constructor & Destructor Documentation	49
7.1.2.1 cartesian_communicator() [1/2]	49
7.1.2.2 cartesian_communicator() [2/2]	50
7.2 cartesian_dimension Struct Reference	50
7.2.1 Detailed Description	50
7.2.2 Constructor & Destructor Documentation	50
7.2.2.1 cartesian_dimension()	50
7.3 cartesian_topology Struct Reference	51
7.3.1 Detailed Description	51
7.3.2 Constructor & Destructor Documentation	51
7.3.2.1 cartesian_topology()	51
7.4 communicator Struct Reference	52
7.4.1 Detailed Description	52
7.4.2 Constructor & Destructor Documentation	52
7.4.2.1 communicator()	52
7.5 Conditions Struct Reference	52
7.5.1 Detailed Description	53
7.6 CPD< Tensor_, execution::openmp_policy > Struct Template Reference	53
7.6.1 Detailed Description	53
7.6.2 Member Function Documentation	53
7.6.2.1 operator() [1/9]	54
7.6.2.2 operator() [2/9]	54
7.6.2.3 operator() [3/9]	55
7.6.2.4 operator() [4/9]	55
7.6.2.5 operator() [5/9]	56
7.6.2.6 operator() [6/9]	56
7.6.2.7 operator() [7/9]	57
7.6.2.8 operator() [8/9]	57
7.6.2.9 operator() [9/9]	58
7.7 CPD< Tensor_, execution::openmpi_policy > Struct Template Reference	58
7.7.1 Detailed Description	58
7.7.2 Member Typedef Documentation	59
7.7.2.1 IntArray	59
7.7.3 Member Function Documentation	59
7.7.3.1 operator() [1/9]	60
7.7.3.2 operator() [2/9]	61
7.7.3.3 operator() [3/9]	62
7.7.3.4 operator() [4/9]	62
7.7.3.5 operator() [5/9]	63

7.7.3.6 operator() [6/9]	63
7.7.3.7 operator() [7/9]	64
7.7.3.8 operator() [8/9]	64
7.7.3.9 operator() [9/9]	65
7.8 CPD_DIMTREE < Tensor_, execution::openmpi_policy > Struct Template Reference	65
7.8.1 Detailed Description	65
7.8.2 Member Function Documentation	66
7.8.2.1 operator() [1/8]	66
7.8.2.2 operator() [2/8]	67
7.8.2.3 operator() [3/8]	67
7.8.2.4 operator() [4/8]	68
7.8.2.5 operator() [5/8]	68
7.8.2.6 operator() [6/8]	69
7.8.2.7 operator() [7/8]	69
7.8.2.8 operator() [8/8]	70
7.9 DefaultValues < Tensor_ > Struct Template Reference	70
7.9.1 Detailed Description	70
7.9.2 Member Data Documentation	71
7.9.2.1 DefaultAcceleration	71
7.9.2.2 DefaultAccelerationCoefficient	71
7.9.2.3 DefaultAccelerationFail	71
7.9.2.4 DefaultConstraint	71
7.9.2.5 DefaultLambda	71
7.9.2.6 DefaultMaxDuration	72
7.9.2.7 DefaultMaxIter	72
7.9.2.8 DefaultMethod	72
7.9.2.9 DefaultNesterovTolerance	72
7.9.2.10 DefaultNormalization	72
7.9.2.11 DefaultProcessorPerMode	72
7.9.2.12 DefaultThresholdError	72
7.9.2.13 DefaultWriteToFile	73
7.10 environment Struct Reference	73
7.10.1 Detailed Description	73
7.10.2 Constructor & Destructor Documentation	73
7.10.2.1 environment()	73
7.10.2.2 environment()	74
7.11 ExprNode < _LabelSetSize, _ParLabelSetSize, _RootSize > Struct Template Reference	74
7.11.1 Detailed Description	74
7.11.2 Constructor & Destructor Documentation	76
7.11.2.1 ExprNode() [1/2]	76
7.11.2.2 ExprNode() [2/2]	76
7.11.3 Member Function Documentation	76

7.11.3.1 DeltaSet()	76
7.11.3.2 LabelSet()	76
7.11.3.3 Left()	77
7.11.3.4 Parent()	77
7.11.3.5 Right()	77
7.11.3.6 SearchKey()	77
7.11.3.7 TnsDims()	78
7.11.3.8 TreeMode_N_Product()	78
7.11.3.9 TTVs()	79
7.11.3.10 TTVs_util()	79
7.11.3.11 UpdateTree()	80
7.11.4 Member Data Documentation	80
7.11.4.1 BrotherLabelSetSize	81
7.11.4.2 DIM_HALF_SIZE	81
7.11.4.3 DIM_LEFT_SIZE	81
7.11.4.4 DIM_RIGHT_SIZE	81
7.11.4.5 IsFirstChild	81
7.11.4.6 IsLeaf	81
7.11.4.7 IsRoot	81
7.11.4.8 LabelSetSize	81
7.11.4.9 left	82
7.11.4.10 mDeltaSet	82
7.11.4.11 mGramian	82
7.11.4.12 mKey	82
7.11.4.13 mLabelSet	82
7.11.4.14 mTnsDims	82
7.11.4.15 mTnsX	82
7.11.4.16 mUpdated	82
7.11.4.17 parent	83
7.11.4.18 ParLabelSetSize	83
7.11.4.19 ParTnsSize	83
7.11.4.20 right	83
7.11.4.21 RootSize	83
7.11.4.22 TnsSize	83
7.12 ExprTree <_TnsSize > Struct Template Reference	83
7.12.1 Detailed Description	83
7.12.2 Member Function Documentation	84
7.12.2.1 Create() [1/2]	84
7.12.2.2 Create() [2/2]	84
7.12.3 Member Data Documentation	85
7.12.3.1 IsNull	85
7.12.3.2 root	85

7.13 Factor< FactorType > Struct Template Reference	85
7.13.1 Detailed Description	85
7.14 FactorDimTree Struct Reference	86
7.14.1 Detailed Description	86
7.15 GTC< TnsSize_, execution::openmpi_policy > Struct Template Reference	86
7.15.1 Detailed Description	86
7.15.2 Member Typedef Documentation	87
7.15.2.1 IntArray	87
7.15.3 Member Function Documentation	87
7.15.3.1 initialize_factors()	87
7.15.3.2 operator>() [1/2]	87
7.15.3.3 operator>() [2/2]	88
7.16 GTC_STOCHASTIC< TnsSize_, execution::openmpi_policy > Struct Template Reference	88
7.16.1 Detailed Description	88
7.16.2 Member Typedef Documentation	89
7.16.2.1 IntArray	89
7.16.3 Member Function Documentation	89
7.16.3.1 initialize_factors()	89
7.16.3.2 operator>() [1/2]	89
7.16.3.3 operator>() [2/2]	90
7.17 I_TnsNode Struct Reference	90
7.17.1 Detailed Description	90
7.17.2 Constructor & Destructor Documentation	91
7.17.2.1 I_TnsNode()	91
7.17.3 Member Function Documentation	91
7.17.3.1 DeltaSet()	91
7.17.3.2 Gramian()	92
7.17.3.3 Key()	92
7.17.3.4 LabelSet()	92
7.17.3.5 Left()	92
7.17.3.6 Parent()	92
7.17.3.7 Right()	93
7.17.3.8 SearchKey()	93
7.17.3.9 SetOutdated()	93
7.17.3.10 TensorX()	93
7.17.3.11 TnsDims()	94
7.17.3.12 Updated()	94
7.17.3.13 UpdateTree()	94
7.17.4 Member Data Documentation	94
7.17.4.1 TnsSize	94
7.18 MatrixArrayTraits< MA > Struct Template Reference	95
7.18.1 Detailed Description	95

7.19 MatrixArrayTraits < std::array < T, _Size > > Struct Template Reference	95
7.19.1 Detailed Description	95
7.20 MatrixTraits < Matrix > Struct Template Reference	96
7.20.1 Detailed Description	96
7.21 MatrixTraits < Matrix > Struct Reference	96
7.21.1 Detailed Description	96
7.22 Options < Tensor_, ExecutionPolicy_, DefaultValues_ > Struct Template Reference	96
7.22.1 Detailed Description	96
7.22.2 Member Typedef Documentation	97
7.22.2.1 DataType	97
7.22.2.2 MatrixType	97
7.22.3 Constructor & Destructor Documentation	97
7.22.3.1 Options()	98
7.22.4 Member Data Documentation	98
7.22.4.1 TnsSize	98
7.23 SparseStatus < _TnsSize, ExecutionPolicy_, DefaultValues_ > Struct Template Reference	98
7.23.1 Detailed Description	98
7.23.2 Member Typedef Documentation	99
7.23.2.1 MatrixArray	99
7.23.3 Member Data Documentation	99
7.23.3.1 ao_iter	99
7.23.3.2 f_value	99
7.23.3.3 factors	99
7.23.3.4 frob_tns	99
7.23.3.5 rel_costFunction	100
7.24 SparseTensorTraits < SparseTensor > Struct Template Reference	100
7.24.1 Detailed Description	100
7.25 SparseTensorTraits < SparseTensor < _TnsSize > > Struct Template Reference	100
7.25.1 Detailed Description	100
7.25.2 Member Typedef Documentation	101
7.25.2.1 Constraints	101
7.25.2.2 DoubleArray	101
7.25.2.3 IntArray	101
7.25.2.4 MatrixArray	101
7.26 Status < Tensor_, ExecutionPolicy_, DefaultValues_ > Struct Template Reference	101
7.26.1 Detailed Description	101
7.26.2 Member Typedef Documentation	102
7.26.2.1 MatrixArray	102
7.26.3 Member Data Documentation	102
7.26.3.1 ao_iter	102
7.26.3.2 f_value	102
7.26.3.3 factors	103

7.26.3.4	frob_tns	103
7.26.3.5	rel_costFunction	103
7.27	TensorTraits< Tensor > Struct Template Reference	103
7.27.1	Detailed Description	103
7.28	TensorTraits< Tensor< _TnsSize > > Struct Template Reference	103
7.28.1	Detailed Description	104
7.28.2	Member Typedef Documentation	104
7.28.2.1	Constraints	104
7.28.2.2	DoubleArray	104
7.28.2.3	IntArray	104
7.28.2.4	MatrixArray	105
7.29	Timers Struct Reference	105
7.29.1	Detailed Description	105
7.29.2	Member Typedef Documentation	105
7.29.2.1	ClockHigh	105
7.29.2.2	ClockSteady	106
7.29.3	Member Function Documentation	106
7.29.3.1	endChronoHighTimer()	106
7.29.3.2	endChronoSteadyTimer()	106
7.29.3.3	endCpuTimer()	106
7.29.3.4	endMpiTimer()	107
7.29.3.5	startChronoHighTimer()	107
7.29.3.6	startChronoSteadyTimer()	107
7.29.3.7	startCpuTimer()	107
7.29.3.8	startMpiTimer()	107
7.30	TnsNode< _TnsSize > Struct Template Reference	107
7.30.1	Detailed Description	107
7.30.2	Constructor & Destructor Documentation	108
7.30.2.1	TnsNode()	108
7.30.3	Member Function Documentation	108
7.30.3.1	Gramian()	109
7.30.3.2	Key()	109
7.30.3.3	SetOutdated()	109
7.30.3.4	TensorX()	109
7.30.3.5	Updated()	110
7.30.4	Member Data Documentation	110
7.30.4.1	IsNull	110
7.30.4.2	mGramian	110
7.30.4.3	mKey	110
7.30.4.4	mTnsX	110
7.30.4.5	mUpdated	110
7.30.4.6	TnsSize	111

7.31 TnsNode < 0 > Struct Reference	111
7.31.1 Detailed Description	111
7.31.2 Member Typedef Documentation	112
7.31.2.1 Tensor_Type	112
7.31.3 Constructor & Destructor Documentation	112
7.31.3.1 TnsNode()	112
7.31.4 Member Function Documentation	112
7.31.4.1 DeltaSet()	112
7.31.4.2 Gramian()	113
7.31.4.3 Key()	113
7.31.4.4 LabelSet()	113
7.31.4.5 Left()	113
7.31.4.6 Parent()	113
7.31.4.7 Right()	114
7.31.4.8 SearchKey()	114
7.31.4.9 SetOutdated()	114
7.31.4.10 TensorX()	114
7.31.4.11 TnsDims()	115
7.31.4.12 Updated()	115
7.31.4.13 UpdateTree()	115
7.31.5 Member Data Documentation	115
7.31.5.1 IsNull	115
7.31.5.2 TnsSize	116
8 File Documentation	117
8.1 Config.hpp File Reference	117
8.1.1 Detailed Description	117
8.2 Config.hpp	117
8.3 Constants.hpp File Reference	117
8.3.1 Detailed Description	118
8.3.2 Enumeration Type Documentation	118
8.3.2.1 Constraint	118
8.3.2.2 Distribution	118
8.3.2.3 Method	118
8.3.2.4 ProblemType	118
8.4 Constants.hpp	119
8.5 Cpd.hpp File Reference	119
8.5.1 Detailed Description	120
8.5.2 Function Documentation	120
8.5.2.1 cpd() [1/9]	120
8.5.2.2 cpd() [2/9]	121
8.5.2.3 cpd() [3/9]	122

8.5.2.4 cpd() [4/9]	123
8.5.2.5 cpd() [5/9]	123
8.5.2.6 cpd() [6/9]	124
8.5.2.7 cpd() [7/9]	125
8.5.2.8 cpd() [8/9]	125
8.5.2.9 cpd() [9/9]	126
8.6 Cpd.hpp	127
8.7 CpdDimTree.hpp File Reference	141
8.7.1 Detailed Description	142
8.7.2 Function Documentation	142
8.7.2.1 cpdDimTree() [1/8]	143
8.7.2.2 cpdDimTree() [2/8]	143
8.7.2.3 cpdDimTree() [3/8]	144
8.7.2.4 cpdDimTree() [4/8]	145
8.7.2.5 cpdDimTree() [5/8]	145
8.7.2.6 cpdDimTree() [6/8]	146
8.7.2.7 cpdDimTree() [7/8]	147
8.7.2.8 cpdDimTree() [8/8]	147
8.8 CpdDimTree.hpp	148
8.9 CpdDimTreeMpi.hpp File Reference	161
8.9.1 Detailed Description	161
8.10 CpdDimTreeMpi.hpp	161
8.11 CpdMpi.hpp File Reference	173
8.11.1 Detailed Description	173
8.12 CpdMpi.hpp	174
8.13 CpdOpenMP.hpp File Reference	187
8.13.1 Detailed Description	187
8.14 CpdOpenMP.hpp	187
8.15 CwiseProd.hpp File Reference	194
8.15.1 Detailed Description	194
8.15.2 Function Documentation	194
8.15.2.1 CwiseProd()	194
8.16 CwiseProd.hpp	195
8.17 DataGeneration.hpp File Reference	195
8.17.1 Detailed Description	196
8.17.2 Function Documentation	196
8.17.2.1 generateRandomMatrix()	196
8.17.2.2 generateRandomTensor()	197
8.17.2.3 generateTensor() [1/2]	197
8.17.2.4 generateTensor() [2/2]	198
8.17.2.5 makeFactors()	198
8.17.2.6 makeTensor()	199

8.18 DataGeneration.hpp	199
8.19 DimTrees.hpp File Reference	206
8.19.1 Detailed Description	206
8.19.2 Function Documentation	207
8.19.2.1 search_leaf()	207
8.20 DimTrees.hpp	207
8.21 execution.hpp File Reference	215
8.21.1 Detailed Description	216
8.22 execution.hpp	216
8.23 Gtc.hpp File Reference	219
8.23.1 Detailed Description	220
8.23.2 Function Documentation	220
8.23.2.1 gtc() [1/2]	220
8.23.2.2 gtc() [2/2]	221
8.24 Gtc.hpp	221
8.25 GtcMpi.hpp File Reference	229
8.25.1 Detailed Description	230
8.26 GtcMpi.hpp	230
8.27 GtcOpenMP.hpp File Reference	242
8.27.1 Detailed Description	242
8.28 GtcOpenMP.hpp	242
8.29 GtcStochastic.hpp File Reference	250
8.29.1 Detailed Description	250
8.29.2 Function Documentation	250
8.29.2.1 gtc_stochastic() [1/2]	251
8.29.2.2 gtc_stochastic() [2/2]	251
8.30 GtcStochastic.hpp	252
8.31 GtcStochasticMpi.hpp File Reference	260
8.31.1 Detailed Description	260
8.32 GtcStochasticMpi.hpp	260
8.33 GtcStochasticOpenMP.hpp File Reference	270
8.33.1 Detailed Description	270
8.34 GtcStochasticOpenMP.hpp	270
8.35 KhatriRao.hpp File Reference	277
8.35.1 Detailed Description	277
8.35.2 Function Documentation	277
8.35.2.1 KhatriRao()	278
8.36 KhatriRao.hpp	278
8.37 Kronecker.hpp File Reference	285
8.37.1 Detailed Description	285
8.37.2 Function Documentation	285
8.37.2.1 Kronecker()	286

8.38 Kronecker.hpp	287
8.39 Matricization.hpp File Reference	287
8.39.1 Detailed Description	288
8.39.2 Function Documentation	288
8.39.2.1 Matricization()	288
8.40 Matricization.hpp	289
8.41 MTTKRP.hpp File Reference	295
8.41.1 Detailed Description	296
8.41.2 Function Documentation	296
8.41.2.1 mttkrp() [1/2]	296
8.41.2.2 mttkrp() [2/2]	296
8.42 MTTKRP.hpp	297
8.43 NesterovMNLS.hpp File Reference	302
8.43.1 Detailed Description	302
8.43.2 Function Documentation	302
8.43.2.1 ComputeSVD()	302
8.43.2.2 GLambda()	303
8.43.2.3 NesterovMNLS()	303
8.43.2.4 UpdateAlpha()	304
8.44 NesterovMNLS.hpp	304
8.45 Normalize.hpp File Reference	316
8.45.1 Detailed Description	316
8.45.2 Function Documentation	316
8.45.2.1 choose_normilization_factor()	316
8.45.2.2 Normalize() [1/2]	317
8.45.2.3 Normalize() [2/2]	318
8.46 Normalize.hpp	318
8.47 ParallelWrapper.hpp File Reference	321
8.47.1 Detailed Description	322
8.47.2 Typedef Documentation	322
8.47.2.1 Boost_CartCommunicator	322
8.47.2.2 Boost_CartDimension	322
8.47.2.3 Boost_CartTopology	322
8.47.2.4 Boost_Communicator	322
8.47.2.5 Boost_Environment	323
8.47.3 Function Documentation	323
8.47.3.1 all_reduce()	323
8.47.3.2 create_fiber_grid()	324
8.47.3.3 create_layer_grid()	324
8.47.3.4 DisCount()	325
8.47.3.5 inplace()	325
8.48 ParallelWrapper.hpp	325

8.49 PARTENSOR.hpp File Reference	331
8.49.1 Detailed Description	332
8.50 PARTENSOR.hpp	332
8.51 PARTENSOR_basic.hpp File Reference	332
8.51.1 Detailed Description	333
8.51.2 Typedef Documentation	333
8.51.2.1 Clock	333
8.51.2.2 Duration	334
8.51.3 Function Documentation	334
8.51.3.1 MakeOptions() [1/2]	334
8.51.3.2 MakeOptions() [2/2]	334
8.51.3.3 MakeSparseOptions() [1/2]	334
8.51.3.4 MakeSparseOptions() [2/2]	334
8.52 PARTENSOR_basic.hpp	335
8.53 PartialCwiseProd.hpp File Reference	344
8.53.1 Detailed Description	344
8.53.2 Function Documentation	344
8.53.2.1 PartialCwiseProd()	344
8.54 PartialCwiseProd.hpp	345
8.55 PartialKhatriRao.hpp File Reference	347
8.55.1 Detailed Description	348
8.55.2 Function Documentation	348
8.55.2.1 PartialKhatriRao()	348
8.56 PartialKhatriRao.hpp	349
8.57 ReadWrite.hpp File Reference	351
8.57.1 Detailed Description	352
8.57.2 Function Documentation	352
8.57.2.1 read()	352
8.57.2.2 readFMRI_matrix()	353
8.57.2.3 readFMRI_mpi()	354
8.57.2.4 readFMRI_tensor()	354
8.57.2.5 readTensor()	355
8.57.2.6 write()	356
8.57.2.7 writeToFile_append()	356
8.58 ReadWrite.hpp	357
8.59 temp.hpp	363
8.60 Tensor.hpp File Reference	366
8.60.1 Detailed Description	366
8.60.2 Typedef Documentation	366
8.60.2.1 DefaultDataType	367
8.60.2.2 LongMatrix	367
8.60.2.3 Matrix	367

8.60.2.4 SparseMatrix	367
8.60.2.5 Tensor	367
8.60.3 Variable Documentation	367
8.60.3.1 is_matrix	367
8.61 Tensor.hpp	368
8.62 TensorOperations.hpp File Reference	369
8.62.1 Detailed Description	370
8.62.2 Function Documentation	370
8.62.2.1 BalanceDataset()	370
8.62.2.2 DepermuteFactors()	371
8.62.2.3 IdentityTensorGen()	371
8.62.2.4 matrixToTensor() [1/2]	372
8.62.2.5 matrixToTensor() [2/2]	372
8.62.2.6 norm()	373
8.62.2.7 PermuteFactors()	373
8.62.2.8 PermuteModeN()	374
8.62.2.9 RandomTensorGen()	374
8.62.2.10 square_norm()	376
8.62.2.11 tensorToMatrix()	376
8.62.2.12 ZeroTensorGen()	377
8.63 TensorOperations.hpp	377
8.64 TerminationConditions.hpp File Reference	385
8.64.1 Detailed Description	386
8.64.2 Function Documentation	386
8.64.2.1 error()	386
8.64.2.2 maxIterations()	387
8.64.2.3 objectiveValueError()	387
8.64.2.4 relativeCostFunction()	387
8.64.2.5 relativeError()	388
8.65 TerminationConditions.hpp	388
8.66 Timers.hpp File Reference	389
8.66.1 Detailed Description	389
8.66.2 Variable Documentation	389
8.66.2.1 timer	390
8.67 Timers.hpp	390
Index	393

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 [MPI CH](#) 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_factors_init.cpp, cpd.cpp).

- By enabling writeToFile and specifying final_factors_paths in the options object, the result 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 << "m factor " << i << "m" << 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 `std::array<int, tensor_order>`. 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 `std::array<string, 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_factors_init_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 << "mn factor " << i << "mn" << 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 `std::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_factors_init.cpp`, `cpd_mpi.cpp`).
- By enabling `wri teToFi l e` and specifying `fi nal _factors_paths` in the `opti ons` object, the resulting factors can be saved in files. Their location will be indicated in the `fi nal _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()->MPI_Communicator();
    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::MPI_Status<Tensor>;
    using Options = partensor::MPI_Options<Tensor>;
    using Constraint = partensor::Constraint;
    std::array<int, tensor_order> tnsDims = {10, 11, 12};
    std::array<Matrix, tensor_order> ini_t_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, ini_t_factors);
    Status status = partensor::cpd(partensor::execution::mpi, tnsX, rank, options, ini_t_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_factors_init_file.cpp`, `cpd_mpi_file.cpp`).
- By enabling `wri teToFi l e` and specifying `fi nal _factors_paths` in the `opti ons` object, the resulting factors can be saved in files. Their location will be indicated in the `fi nal _factors_paths` variable.

```

#define USE_MPI 1
#include "PARTENSOR.hpp"
int main(int argc, char** argv)
{
    partensor::Int(argc, argv);
    partensor::MPI_Communicator _comm = partensor::Partensor()->MPI_Communicator();
    constexpr std::size_t tensor_order = 3;
    constexpr std::size_t rank = 2;
    using Tensor = partensor::Tensor<tensor_order>;
    using Status = partensor::MPI_Status<Tensor>;
    using Options = partensor::MPI_Options<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 << "m factor " << i << "m" << 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 `std::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_factors_init.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 << "nn factor " << i << "nn" << 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 `std::array`, `tnsDims`. Again, the factorization rank is required. A full list of the options can be found in [Options for ALS Method](#).

Finally, an `std::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_factors_init_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 << "nn factor " << i << "nn" << 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 **MPI CH** 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 `std::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_factors_init.cpp`, `cpd_dimtrees.cpp`).
- By enabling `wri teToFi le` and specifying `fi nal _factors_paths` in the `opti ons` object, the resulting factors can be saved in files. Their location will be indicated in the `fi nal _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> ini_t_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, ini_t_factors);
    Status status = partensor::cpdDimTree(tnsX, rank, options, ini_t_factors);
    for(std::size_t i=0; i<tensor_order; ++i)
        std::cout << "m factor " << i << "m" << 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 `std::array<int, tensor_order>` `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 `std::array<std::string, tensor_order+1>` 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_dtmrees_options_file.cpp`, `cpd_dtmrees_factors_init_file.cpp`, `cpd_dtmrees_file.cpp`).
- By enabling `wri teToFi le` and specifying `fi nal _factors_paths` in the `opti ons` object, the resulting factors can be saved in files. Their location will be indicated in the `fi nal _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::cpdDtmTree(tnsDims, rank, paths, options);
    for(std::size_t i=0; i<tensor_order; ++i)
        std::cout << "m factor " << i << "m" << 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 `std::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_distributed_mpi_options.cpp`, `cpd_distributed_mpi_factors_init.cpp`, `cpd_distributed_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()->MPI_Communicator();
    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::MPI_Status<Tensor>;
    using Options = partensor::MPI_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(partensor::execution::mpi, tnsX, rank, options, init_factors);
    if(!_comm.rank() == 0)
    {
        for(std::size_t i=0; i<tensor_order; ++i)
            std::cout << "m factor " << i << "m" << 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 `std` 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 `std` 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_factors_init_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()->MPI_Communicator();
    constexpr std::size_t tensor_order = 3;
    constexpr std::size_t rank = 2;
    using Tensor = partensor::Tensor<tensor_order>;
    using Status = partensor::MPI_Status<Tensor>;
    std::array<int, tensor_order> tnsDims = {10, 11, 12};
    std::string path = "path to where your Tensor file is located";
    Status status = partensor::cpdDi mTree(partensor::execution::mpi, tnsDims, rank, path);
    if(_comm.rank() == 0)
    {
        for(std::size_t i=0; i<tensor_order; ++i)
            std::cout << "m factor " << i << "m" << 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**, `std` 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 `std` 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 `MPI CH` 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 `std` 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 `wri teToFile` and specifying `fi nal _factors_paths` in the `opti ons` object, the resulting factors can be saved in files. Their location will be indicated in the `fi nal _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 `std::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 result 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 `std` 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 `std::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()->MPI_Communicator();
    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 `std::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(plt::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 `std::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 `MPI CH` 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 `std::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 `wri teToFi le` and specifying `fi nal _factors_paths` in the `opti ons` object, the resulting factors can be saved in files. Their location will be indicated in the `fi nal _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.wri teToFi le = false;
    std::s1 = partensor::gtc_stochastic(pty::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 `std::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**, `std` 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 `std` 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 `std` 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 3x3 matrices **A** and **B**, the Hadamard or Element-wise product is defined as

$$\begin{matrix} 2 & & 3 & 2 & & 3 & 2 & & 3 \\ a_{11} & a_{12} & a_{13} & b_{11} & b_{12} & b_{13} & a_{11} b_{11} & a_{12} b_{12} & a_{13} b_{13} \\ 4 & a_{21} & a_{22} & a_{23} & b_{21} & b_{22} & b_{23} & a_{21} b_{21} & a_{22} b_{22} & a_{23} b_{23} \\ & a_{31} & a_{32} & a_{33} & b_{31} & b_{32} & b_{33} & a_{31} b_{31} & a_{32} b_{32} & a_{33} b_{33} \end{matrix} : \quad \begin{matrix} 3 \\ 5 \\ 5 \end{matrix} = \begin{matrix} 3 \\ 4 \\ 5 \end{matrix}$$

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

$$H = A_1 \cdot A_2 \cdot \dots \cdot A_n$$

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 $M \times N$ matrix A and a $P \times N$ matrix B , the Khatri-Rao product is defined as

$$A \cdot B = [a_1 \quad b_1 \quad a_2 \quad b_2 \quad \dots \quad a_n \quad b_n]$$

which is an $MP \times N$ 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 [Khatri Rao. hpp](#), the implementation is described. The mathematical formula to compute the Khatri-Rao product follows,

$$\text{Krao} = A_1 \cdot A_2 \cdot \dots \cdot A_n$$

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 Khatri-Rao symbol denotes the Khatri-Rao operation.
- Because Khatri Rao 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 A and a PQ matrix B , the Kronecker product is defined as

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

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

$$Kr = A_1 \otimes A_2 \otimes \dots \otimes A_n$$

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 $X \in \mathbb{R}^{I_1 \times I_2 \times \dots \times I_n}$ can be matricized with respect to the n -th mode, namely, the n -mode matricization is the matrix $X_{(n)} \in \mathbb{R}^{I_n \times \prod_{k \neq n} I_k}$.

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 << "Matrixm" << 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 $\text{mean}() = 0$ and $\text{standard deviation}() = 1$.

```
#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 distributionm" << tnsX << std::endl;
    generateRandomTensor(tnsX, 1);
    std::cout << "mNormal distributionm" << 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 `std::array`, then a `Tensor` can be created using the `generateTensor` 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 matrixzed_tensor = generateTensor(0, factors);
    std::cout << "Matricization of first mode" << matrixzed_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" << 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 `std::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 `std::array` with factors, of either `Matrix`, `Tensor<2>` or even `FactoDi mTree` 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 `std::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]m" << 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]m" << 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]m" << factors_DimTrees[2].factor << std::endl;
    return 0;
}

```

Note

FactorDimTree can be found in in [Di mTrees. 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 m" << 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 << "matrixm" << mtx << std::endl;
    std::cout << "mtensorm" << 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 << "matrixm" << mtx << std::endl;
    std::cout << "mtensorm" << 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*, *std 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,

- `std::clock`,
- `std::chrono::high_resolution`,
- `std::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	$\geq 8.3.0$
	Eigen	$\geq 1.4.1$
	spdlog	$\geq 1.4.1$
Parallel with MPI	Boost	$\geq 1.71.0$
	OpenMPI , or	$\geq 4.0.1$
	MPI CH	$\geq 3.2.0$

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 << "m factor " << i << "m" << 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 Advances 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 $\text{mean}() = 0$ and $\text{standard deviation}() = 1$.

```
#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" << tnsX << std::endl;
    generateRandomTensor(tnsX, 1);
    std::cout << "Normal distribution" << tnsX << std::endl;
    return 0;
}
```

4.1.3 Matricization of a Tensor from Factors

In case the factors are available and stored in an `std::array`, then a Tensor can be created using the `generateTensor` 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 modern" << 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 $m$ " << 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 `std` 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 `std` array with factors, of either `Matrix`, `Tensor<2>` or even `FactorDimTree` 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 `std` 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] $m$ " << 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] $m$ " << 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] $m$ " << 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 m" << 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 **A** and **B**, the Hadamard or Element-wise product is defined as

$$\begin{matrix} 2 & & 3 & 2 & & 3 & 2 & & 3 \\ a_{11} & a_{12} & a_{13} & b_{11} & b_{12} & b_{13} & a_{11} b_{11} & a_{12} b_{12} & a_{13} b_{13} \\ 4 & a_{21} & a_{22} & a_{23} & b_{21} & b_{22} & b_{23} & a_{21} b_{21} & a_{22} b_{22} & a_{23} b_{23} \\ & a_{31} & a_{32} & a_{33} & b_{31} & b_{32} & b_{33} & a_{31} b_{31} & a_{32} b_{32} & a_{33} b_{33} \end{matrix} =$$

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

$$H = A_1 \cdot A_2 \cdot \dots \cdot A_n$$

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 $M \times N$ matrix A and a $P \times N$ matrix B , the Khatri-Rao product is defined as

$$A \text{ B} = [a_1 \quad b_1 \ a_2 \quad b_2 \quad \dots \quad a_n \quad b_n]$$

which is an $MP \times N$ 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 [Khatri Rao. hpp](#), the implementation is described. The mathematical formula to compute the Khatri-Rao product follows,

$$\text{Krao} = A_1 \quad A_2 \quad \dots \quad A_n$$

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 KhatriRao symbol denotes the Khatri-Rao operation.
- Because Khatri Rao 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 A and a PQ matrix B , the Kronecker product is defined as

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

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

$$\text{Kr} = \mathbf{A}_1 \otimes \mathbf{A}_2 \otimes \dots \otimes \mathbf{A}_n$$

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 `⊗` 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 $X \in \mathbb{R}^{I_1 \times I_2 \times \dots \times I_n}$ can be matricized with respect to the n -th mode, namely, the n -mode matricization is the matrix $X_{(n)} \in \mathbb{R}^{I_n \times \prod_{k \neq n} I_k}$.

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 `generateRandom - Tensor`. 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:

cartesian_communicator	An MPI communicator with a cartesian topology	49
cartesian_communicator	An MPI communicator with a cartesian topology	49
cartesian_dimension	Specify the size and periodicity of the grid in a single dimension	50
cartesian_dimension	Specify the size and periodicity of the grid in a single dimension	50
cartesian_topology	Describe the topology of a cartesian grid	51
cartesian_topology	Describe the topology of a cartesian grid	51
communicator	A communicator that permits communication and synchronization among a set of processes	52
communicator	A communicator that permits communication and synchronization among a set of processes	52
Conditions	52
CPD < Tensor_, execution::openmp_policy >	53
CPD < Tensor_, execution::openmpi_policy >	58
CPD_DIMTREE < Tensor_, execution::openmpi_policy >	65
DefaultValues < Tensor_ >	Default Values for CPD algorithm	70
environment	Initialize, finalize, and query the MPI environment	73
environment	Initialize, finalize, and query the MPI environment	73
ExprNode < _LabelSetSize, _ParLabelSetSize, _RootSize >	74
ExprTree < _TnsSize >	83
Factor < FactorType >	85
FactorDimTree	86
GTC < TnsSize_, execution::openmpi_policy >	86
GTC_STOCHASTIC < TnsSize_, execution::openmpi_policy >	88
I_TnsNode	90
MatrixArrayTraits < MA >	95
MatrixArrayTraits < std::array < T, _Size > >	95

MatrixTraits< Matrix >	96
MatrixTraits< Matrix >	96
Options< Tensor_, ExecutionPolicy_, DefaultValues_ >	
Manage defaults parameters for CPD algorithm	96
SparseStatus< _TnsSize, ExecutionPolicy_, DefaultValues_ >	98
SparseTensorTraits< SparseTensor >	100
SparseTensorTraits< SparseTensor< _TnsSize > >	100
Status< Tensor_, ExecutionPolicy_, DefaultValues_ >	
Returned Type of CPD algorithm	101
TensorTraits< Tensor >	103
TensorTraits< Tensor< _TnsSize > >	103
Timers	105
TnsNode< _TnsSize >	107
TnsNode< 0 >	111

Chapter 6

File Index

6.1 File List

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

Config.hpp	117
Constants.hpp	117
Cpd.hpp	119
CpdDimTree.hpp	141
CpdDimTreeMpi.hpp	161
CpdMpi.hpp	173
CpdOpenMP.hpp	187
CwiseProd.hpp	194
DataGeneration.hpp	195
DimTrees.hpp	206
execution.hpp	215
Gtc.hpp	219
GtcMpi.hpp	229
GtcOpenMP.hpp	242
GtcStochastic.hpp	250
GtcStochasticMpi.hpp	260
GtcStochasticOpenMP.hpp	270
KhatriRao.hpp	277
Kronecker.hpp	285
Matricization.hpp	287
MTTKRP.hpp	295
NesterovMNLS.hpp	302
Normalize.hpp	316
ParallelWrapper.hpp	321
PARTENSOR.hpp	331
PARTENSOR_basic.hpp	332
PartialCwiseProd.hpp	344
PartialKhatriRao.hpp	347
ReadWrite.hpp	351
temp.hpp	363
Tensor.hpp	366
TensorOperations.hpp	369
TerminationConditions.hpp	385
Timers.hpp	389

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

<i>comm</i>	The communicator that the new, cartesian communicator will be based on.
<i>dims</i>	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.
<i>reorder</i>	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

<i>comm</i>	the original communicator.
<i>keep</i>	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

<i>sz</i>	The size of the grid n this dimension.
<i>p</i>	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

<i>dims</i>	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

<i>Tensor_</i>	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

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

<i>tnsDims</i>	[in] Stl array containing the Tensor dimensions, whose length must be same as the Tensor order.
<i>R</i>	[in] The rank of decomposition.
<i>paths</i>	[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

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

<i>tnsDims</i>	[in] <code>Stl</code> array containing the Tensor dimensions, whose length must be same as the Tensor order.
<i>R</i>	[in] The rank of decomposition.
<i>path</i>	[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

<i>tnsDims</i>	[in] Std array containing the Tensor dimensions, whose length must be same as the Tensor order.
<i>R</i>	[in] The rank of decomposition.
<i>path</i>	[in] The path where the tensor is located.
<i>options</i>	[in] User's options, other than the default. It must be of <code>partensor::Options<partensor::Tensor<order>></code> 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

<i>tnsX</i>	[in] The given Tensor to be factorized of <code>Tensor_</code> type, with double data.
<i>R</i>	[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

<i>tnsX</i>	[in] The given Tensor to be factorized of Tensor_ type, with double data.
<i>R</i>	[in] The rank of decomposition.
<i>factorsInit</i>	[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

<i>tnsX</i>	[in] The given Tensor to be factorized of Tensor_ type, with double data.
<i>R</i>	[in] The rank of decomposition.
<i>options</i>	[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

<i>tnsX</i>	[in] The given Tensor to be factorized of Tensor_ type, with double data.
<i>R</i>	[in] The rank of decomposition.
<i>options</i>	[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].
<i>factorsInit</i>	[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 CPD MPI factorization. Based on the given parameters one of the four overloaded operators will be called.

Template Parameters

<i>Tensor</i>	-	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 CPDMP factorization with default values in [Options](#) and initialized factors.

Template Parameters

<i>TnsSize</i>	Order of input Tensor.
----------------	------------------------

Parameters

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

<i>TnsSize</i>	Order of input Tensor.
----------------	------------------------

Parameters

<i>tnsDims</i>	[in] <code>Stl</code> array containing the Tensor dimensions, whose length must be same as the Tensor order.
<i>R</i>	[in] The rank of decomposition.
<i>paths</i>	[in] An <code>stl</code> 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 CPDMPPI factorization with default values in [Options](#) and initialized factors.

Template Parameters

<i>TnsSize</i>	Order of input Tensor.
----------------	------------------------

Parameters

<i>tnsDims</i>	[in] Stl array containing the Tensor dimensions, whose length must be same as the Tensor order.
<i>R</i>	[in] The rank of decomposition.
<i>paths</i>	[in] An stl array containing paths for the Tensor to be factorized and after that the paths for the initialized factors.
<i>options</i>	[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 CPDMPPI factorization with default values in [Options](#) and no initialized factors.

Template Parameters

<i>TnsSize</i>	Order of input Tensor.
----------------	------------------------

Parameters

<i>tnsDims</i>	[in] Stl array containing the Tensor dimensions, whose length must be same as the Tensor order.
<i>R</i>	[in] The rank of decomposition.
<i>path</i>	[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 CPDMPPI factorization with default values in [Options](#) and no initialized factors.

Template Parameters

<i>TnsSize</i>	Order of input Tensor.
----------------	------------------------

Parameters

<i>tnsDims</i>	[in] Std array containing the Tensor dimensions, whose length must be same as the Tensor order.
<i>R</i>	[in] The rank of decomposition.
<i>path</i>	[in] The path where the tensor is located.
<i>options</i>	[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 CPDMPPI factorization with default values in [Options](#) and no initialized factors.

Parameters

<i>tnsX</i>	[in] The given Tensor to be factorized of Tensor_ type, with double data.
<i>R</i>	[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 CPDMPPI factorization with default values in [Options](#) and no initialized factors.

Parameters

<i>tnsX</i>	[in] The given Tensor to be factorized of Tensor_ type, with double data.
<i>R</i>	[in] The rank of decomposition.
<i>factorsInit</i>	[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 CPDMPPI factorization with default values in [Options](#) and no initialized factors.

Parameters

<i>tnsX</i>	[in] The given Tensor to be factorized of Tensor_ type, with double data.
<i>R</i>	[in] The rank of decomposition.
<i>options</i>	[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

<i>tnsX</i>	[in] The given Tensor to be factorized of Tensor_ type, with double data.
<i>R</i>	[in] The rank of decomposition.
<i>options</i>	[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].
<i>factorsInit</i>	[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

<i>Tensor</i>	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 CPDMP factorization with default values in [Options](#) and no initialized factors.

Template Parameters

<i>TnsSize</i>	Order of the input Tensor.
----------------	----------------------------

Parameters

<i>tnsDims</i>	[in] <code>Stl</code> array containing the Tensor dimensions, whose length must be same as the Tensor order.
<i>R</i>	[in] The rank of decomposition.
<i>paths</i>	[in] An <code>stl</code> 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 CPDMPPI factorization with default values in [Options](#) and no initialized factors.

Template Parameters

<i>TnsSize</i>	Order of the input Tensor.
----------------	----------------------------

Parameters

<i>tnsDims</i>	[in] Stl array containing the Tensor dimensions, whose length must be same as the Tensor order.
<i>R</i>	[in] The rank of decomposition.
<i>paths</i>	[in] An stl array containing paths for the Tensor to be factorized and after that the paths for the initialized factors.
<i>options</i>	[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 CPDMPPI factorization with default values in [Options](#) and no initialized factors.

Template Parameters

<i>TnsSize</i>	Order of the input Tensor.
----------------	----------------------------

Parameters

<i>tnsDims</i>	[in] Stl array containing the Tensor dimensions, whose length must be same as the Tensor order.
<i>R</i>	[in] The rank of decomposition.
<i>path</i>	[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 CPDMPPI factorization with default values in [Options](#) and no initialized factors.

Template Parameters

<i>TnsSize</i>	Order of the input Tensor.
----------------	----------------------------

Parameters

<i>tnsDims</i>	[in] Std array containing the Tensor dimensions, whose length must be same as the Tensor order.
<i>R</i>	[in] The rank of decomposition.
<i>path</i>	[in] The path where the tensor is located.
<i>options</i>	[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 CPDMPPI factorization with default values in [Options](#) and no initialized factors.

Parameters

<i>tnsX</i>	[in] The given Eigen Tensor to be factorized.
<i>R</i>	[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 CPDMPPI factorization with default values in [Options](#) and no initialized factors.

Parameters

<i>tnsX</i>	[in] The given Eigen Tensor to be factorized.
<i>R</i>	[in] The rank of decomposition.
<i>factorsInit</i>	[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 CPDMPPI factorization with default values in [Options](#) and no initialized factors.

Parameters

<i>tnsX</i>	[in] The given Eigen Tensor to be factorized.
<i>R</i>	[in] The rank of decomposition.
<i>options</i>	[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 CPD MPI factorization with default values in [Options](#) and no initialized factors.

Parameters

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

Returns

If `spdl og` 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_basics.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 thresholders or termination conditions parameters. These values can be changed using [Options](#) struct and the appropriate `cpd` or `cpdDimTree` call.

Template Parameters

<i>Tensor</i>	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

<i>argc</i>	Number of arguments provided in <i>argv</i> , as passed into the program's <i>main</i> function.
<i>argv</i>	Array of argument strings passed to the program via <i>main</i> function.
<i>abort_on_exception</i>	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

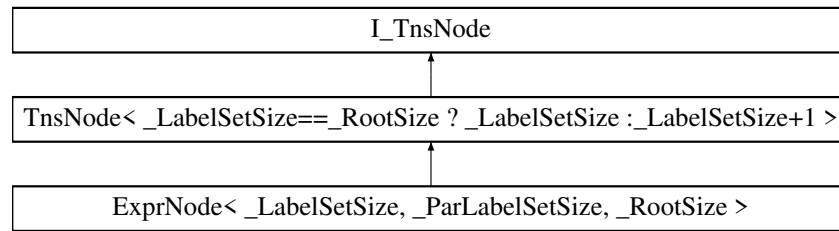
```
template <std::size_t _LabelSetSize, std::size_t _ParLabelSetSize, std::size_t _RootSize >
struct partensor::ExprNode < _LabelSetSize, _ParLabelSetSize, _RootSize >
```

Configuration for [TnsNode](#). Representation of the tree structure.

Template Parameters

<code>_LabelSetSize</code>	Size of the LabelSet.
<code>_ParLabelSetSize</code>	Size of the LabelSet of the parent node.
<code>_RootSize</code>	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 neighbor- 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 ExprNode 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 ExprNode 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 ExprNode 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

<code>aKey</code>	[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 `std::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

<code>_LabelSetSize</code>	Size of the LabelSet of this node.
<code>_ParLabelSetSize</code>	Size of the LabelSet of the parent node.
<code>_RootSize</code>	Size of the LabelSet of the root node.

Parameters

<code>aFactor</code>	[in] Factor (of type FactorDimTree) to use for tree mode N product.
<code>aNumFactors</code>	[in] Total number of factors.
<code>id</code>	[in] Indexing for the updating factor, <code>aFactor</code> .
<code>aGramian</code>	[in,out] Gramian matrix of the node.
<code>aDeltaSet</code>	[in,out] Label set of the brother node after the N-mode product.
<code>aTnsDims</code>	[in,out] <code>std::array</code> 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

<code>_LabelSetSize</code>	Size of the LabelSet of this node.
<code>_ParLabelSetSize</code>	Size of the LabelSet of the parent node.
<code>_RootSize</code>	Size of the LabelSet of the root node.

Parameters

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

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

Parameters

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

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

<i>_LabelSetSize</i>	Size of the LabelSet.
<i>_ParLabelSetSize</i>	Size of the LabelSet of the parent node.
<i>_RootSize</i>	Size of the LabelSet of the root node.

Parameters

<i>aNumFactors</i>	[in] Total number of factors.
<i>id</i>	[in] Identification of the updating factor.
<i>aFactor</i>	[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

<i>TnsSize</i>	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](#), except the root and its children [ExprNodes](#).

Parameters

<i>aTnsDims</i>	[in] Dimensions of the initial Tensor.
<i>R</i>	[in] Rank of the factorization.
<i>expr</i>	[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

<i>aLabelSet</i>	[in] LabelSet of the root ExprNode .
<i>aTnsDims</i>	[in] Dimensions of the initial Tensor.
<i>R</i>	[in] Rank of the factorization.
<i>aTnsX</i>	[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

<i>FactorType</i>	Either FactorDimTree or 2-dimension Tensor.
-------------------	---

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

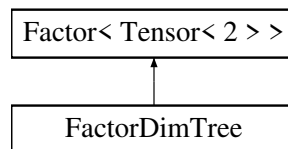
- [Tensor.hpp](#)

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

<i>Tensor</i>	-	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

<i>tnsX</i>	[in] The given Tensor to be factorized of <code>Tensor_ type</code> , with double data.
<i>R</i>	[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

<i>tnsX</i>	[in] The given Tensor to be factorized of <code>Tensor_ type</code> , with double data.
<i>R</i>	[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

<i>Tensor</i>	-	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

<i>tnsX</i>	[in] The given Tensor to be factorized of <code>Tensor_ type</code> , with double data.
<i>R</i>	[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

<i>tnsX</i>	[in] The given Tensor to be factorized of <code>Tensor_ type</code> , with double data.
<i>R</i>	[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:

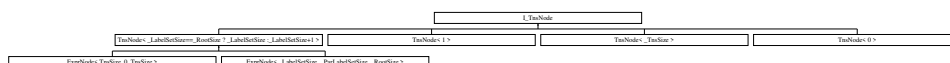
- [GtcStochasticMpi.hpp](#)

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

<i>num_factors</i>	[in] Number of Factors to update.
<i>id</i>	[in] Indexing for Factor to update.
<i>factors</i>	[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

<i>MA</i>	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

<i>T</i>	Type of Eigen Data.
<i>_Size</i>	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

<i>Matrix</i>	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_basics.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 default parameters for CPD algorithm.

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

Template Parameters

<i>Tensor_</i>	Type(data type and order) of input Tensor.
<i>Execution Policy_</i>	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

<code>_TnsSize</code>	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_basi.c.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

<i>Tensor_</i>	Type(data type and order) of input Tensor.
<i>ExecutionPolicy</i>	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

<i>Tensor</i>	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

<code>_TnsSize</code>	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>
```

Typedef 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

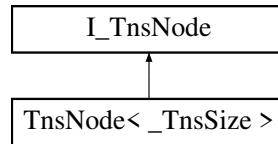
```
template<std::size_t_TnsSize>
struct partensor::TnsNode<_TnsSize>
```

Implementation of the [I_TnsNode](#) interface.

Template Parameters

<code>_TnsSize</code>	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:

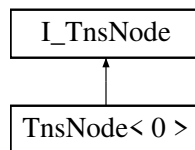
- [DimTrees.hpp](#)

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 Label Set () [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 null ptr.

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 null ptr.

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 null ptr.

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

<i>num_factors</i>	[in] Number of Factors to update.
<i>id</i>	[in] Indexing for Factor to update.
<i>factors</i>	[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.](#)

```
1
2
3
4
5
6
7 #if !defined (USE_MPI)
8 #define USE_MPI 0
9 #endif /* !defined (USE_MPI) */
10
11 #if !defined (USE_OPENMP)
12 #define USE_OPENMP 0
13 #endif /* !defined (USE_OPENMP) */
14
15 #if !defined (USE_CUDA)
16 #define USE_CUDA 0
17 #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.](#)

```

1 #ifndef DOXYGEN_SHOULD_SKIP_THIS
15 #endif // DOXYGEN_SHOULD_SKIP_THIS
16 /*****
22 #ifndef PARTENSOR_CONSTANTS_HPP
23 #define PARTENSOR_CONSTANTS_HPP
24
25 namespace partensor {
26
31     enum class ProblemType : uint8_t {
32         dense = 0,
33         sparse = 1,
34         incomplete = 2
35     };
36
40     enum class Method : uint8_t {
41         als = 0, // alternating least squares
42         rnd = 1, // randomized
43         bc = 2 // block coordinate descent
44     };
45
49     enum class Constraint : uint8_t {
50         unconstrained = 0,
51         nonnegativity = 1,
52         orthogonality = 2,
53         sparsity = 3,
54         constant = 4,
55         symmetric_nonnegativity = 5,
56         symmetric = 6
57     };
58
64     enum class Distribution : uint8_t {
65         uniform = 0,
66         gaussian = 1
67     };
68
69 } // end namespace partensor
70
71 #endif // PARTENSOR_CONSTANTS_HPP

```

8.5 Cpd.hpp File Reference

```

#include "PARTENSOR_basics.hpp"
#include "Matrixization.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 ExecutionPolicy, 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

<i>ExecutionPolicy</i>	Type of stl ExecutionPolicy (sequential, parallel-mpi).
<i>TnsSize</i>	Order of input Tensor.

Parameters

<i>tnsDims</i>	[in] Stl array containing the Tensor dimensions, whose length must be same as the Tensor order.
<i>R</i>	[in] The rank of decomposition.
<i>true_paths</i>	[in] An stl array containing paths for the true factors.
<i>init_paths</i>	[in] An stl array containing paths for initialized factors.
<i>options</i>	[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 ExecutionPolicy, 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

<i>ExecutionPolicy</i>	Type of <code>stl ExecutionPolicy</code> (sequential, parallel-mpi).
<i>TnsSize</i>	Order of input Tensor.

Parameters

<i>tnsDims</i>	[in] <code>Stl</code> array containing the Tensor dimensions, whose length must be same as the Tensor order.
<i>R</i>	[in] The rank of decomposition.
<i>paths</i>	[in] An <code>stl</code> 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 `ExecutionPolicy`, 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

<i>ExecutionPolicy</i>	Type of <code>stl ExecutionPolicy</code> (sequential, parallel-mpi).
<i>TnsSize</i>	Order of input Tensor.

Parameters

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

Parameters

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

Returns

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

8.5.2.4 `cpd()` [4/9]

```

executi on: : internal: : enable_i f_executi on_poli cy< Executi onPoli cy, _Status< Tensor< stati c_ -
cast< i nt >(TnsSi ze) >, _executi on: : executi on_poli cy_t< Executi onPoli cy >, Defaul tVal ues > >
partensor: : cpd (
    Executi onPoli cy && ,
    std: : array< i nt, TnsSi ze > const & tnsDi ms,
    std: : si ze_t const R,
    std: : string const & path )

```

Interface of Canonical Polyadic Decomposition(`cpd`), with the use of an `Executi onPoli cy`, 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 `executi on: : seq`, `executi on: : mpi` or `executi on: : omp`. Default value is sequential, in case no `Executi onPoli cy` is passed.

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

Template Parameters

<i>ExecutionPolicy</i>	Type of <code>stl Executi onPoli cy</code> (sequential, parallel-mpi).
<i>TnsSize</i>	Order of input Tensor.

Parameters

<i>tnsDims</i>	[in] <code>Stl</code> array containing the Tensor dimensions, whose length must be same as the Tensor order.
<i>R</i>	[in] The rank of decomposition.
<i>path</i>	[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]

```

executi on: : internal: : enable_i f_executi on_poli cy< Executi onPoli cy, _Status< Tensor< stati c_ -
cast< i nt >(TnsSi ze) >, _executi on: : executi on_poli cy_t< Executi onPoli cy >, Defaul tVal ues > >

```

```
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 `ExecutionPolicy`, 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

<i>ExecutionPolicy</i>	Type of <code>std::ExecutionPolicy</code> (sequential, parallel-mpi).
<i>TnsSize</i>	Order of input Tensor.

Parameters

<i>tnsDims</i>	[in] <code>std::array</code> containing the Tensor dimensions, whose length must be same as the Tensor order.
<i>R</i>	[in] The rank of decomposition.
<i>path</i>	[in] The path where the tensor is located.
<i>options</i>	[in] User's options, other than the default. It must be of <code>partensor::Options<partensor::Tensor<order>></code> 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 `ExecutionPolicy`, 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

<i>ExecutionPolicy</i>	Type of <code>std::ExecutionPolicy</code> (sequential, parallel-mpi).
<i>Tensor_</i>	Type(data type and order) of input Tensor. <code>Tensor_</code> must be <code>partensor::Tensor<order></code> , where order must be in range of [3-8].

Parameters

<i>tnsX</i>	[in] The given Tensor to be factorized of Tensor_ type, with double data.
<i>R</i>	[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 ExecutionPolicy, 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

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

Parameters

<i>tnsX</i>	[in] The given Tensor to be factorized of Tensor_ type, with double data.
<i>R</i>	[in] The rank of decomposition.
<i>factorsInit</i>	[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 ExecutionPolicy, 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

<i>ExecutionPolicy</i>	Type of std::ExecutionPolicy (sequential, parallel-mpi).
<i>Tensor_</i>	Type(data type and order) of input Tensor. Tensor_ must be partensor::Tensor<order>, where order must be in range of [3-8].

Parameters

<i>tnsX</i>	[in] The given Tensor to be factorized of Tensor_ type, with double data.
<i>R</i>	[in] The rank of decomposition.
<i>options</i>	[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 ExecutionPolicy, 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

<i>ExecutionPolicy</i>	Type of std::ExecutionPolicy (sequential, parallel-mpi).
<i>Tensor_</i>	Type(data type and order) of input Tensor. Tensor_ must be partensor::Tensor<order>, where order must be in range of [3-8].

Parameters

<i>tnsX</i>	[in] The given Tensor to be factorized of Tensor_ type, with double data.
<i>R</i>	[in] The rank of decomposition.
<i>options</i>	[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].
<i>factorsInit</i>	[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.](#)

```

1 #ifndef DOXYGEN_SHOULD_SKIP_THIS
15 #endif // DOXYGEN_SHOULD_SKIP_THIS
16 /*****
26 #ifndef PARTENSOR_CPD_HPP
27 #define PARTENSOR_CPD_HPP
28
29 #include "PARTENSOR_basics.hpp"
30 #include "Matrixization.hpp"
31 #include "PartialCwiseProd.hpp"
32 #include "MTTKRP.hpp"
33 #include "NesterovMNLs.hpp"
34 #include "Normalize.hpp"
35 #include "Timers.hpp"
36 #include "ReadWrite.hpp"
37
38 namespace partensor
39 {
40     inline namespace v1
41     {
42         namespace internal
43         {
44             //template <typename ExecutionPolicy, typename Tensor>
45             //execution::internal::enable_if_execution_policy<ExecutionPolicy, Tensor>
46             //Status cpd_f(ExecutionPolicy &&, Tensor const &tnsX, std::size_t rank);
47
48             /*
49             * Includes the implementation of CP Decomposition. Based on the given
50             * parameters one of the overloaded operators will be called.
51             */
52             template <typename Tensor_>
53             struct CPD_Base {
54                 static constexpr std::size_t TnsSize = TensorTraits<Tensor_>::TnsSize;
55                 static constexpr std::size_t lastFactor = TnsSize - 1;
56                 using DataType = typename TensorTraits<Tensor_>::DataType;
57                 using MatrixType = typename TensorTraits<Tensor_>::MatrixType;
58                 using Dimensions = typename TensorTraits<Tensor_>::Dimensions;
59                 using Constraints = std::array<Constraint, TnsSize>;
60                 using MatrixArray = std::array<MatrixType, TnsSize>;
61                 using DoubleArray = std::array<double, TnsSize>;
62             };
63
64             template <typename Tensor_, typename ExecutionPolicy = execution::sequenced_policy>
65             struct CPD : public CPD_Base<Tensor_>
66             {
67                 using CPD_Base<Tensor_>::TnsSize;
68                 using CPD_Base<Tensor_>::lastFactor;
69                 using typename CPD_Base<Tensor_>::Dimensions;
70                 using typename CPD_Base<Tensor_>::MatrixArray;
71                 using typename CPD_Base<Tensor_>::DataType;
72
73                 using Options = partensor::Options<Tensor_, execution::sequenced_policy, DefaultValues>;
74                 using Status = partensor::Status<Tensor_, execution::sequenced_policy, DefaultValues>;
75
76                 // Variables that will be used in cpd implementations.
77                 struct Member_Variables {

```

```

80     MatrixArray krao;
81     MatrixArray factor_T_factor;
82     MatrixArray mttkrp;
83     MatrixArray tns_mat;
84     MatrixArray norm_factors;
85     MatrixArray old_factors;
86     MatrixArray true_factors;
87
88     Matrix     cwise_factor_product;
89     Matrix     tnsX_mat_LastFactor_T;
90     Matrix     temp_matrix;
91
92     Tensor_    tnsX;
93
94     bool       all_orthogonal = true;
95     int        weight_factor;
96
97     Member_Variab les() = default;
98     Member_Variab les(Member_Variab les const &) = default;
99     Member_Variab les(Member_Variab les      &&) = default;
100
101     Member_Variab les &operator=(Member_Variab les const &) = default;
102     Member_Variab les &operator=(Member_Variab les      &&) = default;
103 };
104
105 /*
106  * In case option variable @c writeToFile is enabled, then, before the end
107  * of the algorithm, it writes the resulted factors in files, whose
108  * paths are specified before compiling in @c options.final_factors_path.
109  *
110  * @param st [in] Struct where the returned values of @c Cpd are stored.
111  */
112 void writeFactorsToFile(Status const &st)
113 {
114     std::size_t size;
115     for(std::size_t i=0; i<TnsSize; ++i)
116     {
117         size = st.factors[i].rows() * st.factors[i].cols();
118         partensor::write(st.factors[i],
119                        st.options.final_factors_paths[i],
120                        size);
121     }
122 }
123
124 /*
125  * Compute the cost function value based on the initial factors.
126  *
127  * @param mv [in] Struct where ALS variables are stored.
128  * @param st [in,out] Struct where the returned values of @c Cpd are stored.
129  * In this case the cost function value is updated.
130  */
131 void cost_function_init(Member_Variab les const &mv,
132                       Status      &st)
133 {
134     st.f_value = sqrt( ( mv.tns_mat[LastFactor] - st.factors[LastFactor] *
135     PartialKhatriRao(st.factors, LastFactor).transpose()).squaredNorm() );
136 }
137
138 /*
139  * Compute the cost function value at the end of each outer iteration
140  * based on the last factor.
141  *
142  * @param mv [in] Struct where ALS variables are stored.
143  * @param st [in,out] Struct where the returned values of @c Cpd are stored.
144  * In this case the cost function value is updated.
145  */
146 void cost_function(Member_Variab les const &mv,
147                  Status      &st)
148 {
149     st.f_value = sqrt( ( mv.tns_mat[LastFactor] - st.factors[LastFactor] *
150     mv.krao[LastFactor].transpose()).squaredNorm() );
151 }
152
153 /*
154  * Compute the cost function value at the end of each outer iteration
155  * based on the last accelerated factor.
156  *
157  * @param mv [in] Struct where ALS variables are stored.
158  * @param st [in] Struct where the returned values of @c Cpd are stored.
159  * @param factors [in] Accelerated factors.
160  * @param factors_T_factors [in] Gramian matrices of factors.
161  *
162  * @returns The cost function calculated with the accelerated factors.
163  */
164 double accel_cost_function(Member_Variab les const &mv,
165                           Status      const &st,
166                           MatrixArray const &factors,

```



```

165             MatrixArray      const &factors_T_factors)
166     {
167         return sqrt( st.frob_tns + (PartialCwiseProd(factors_T_factors, lastFactor) *
factors_T_factors[lastFactor]).trace()
168             - 2 * ((PartialKhatriRao(factors, lastFactor).transpose() * mv.tnsX_mat_lastFactor_T) *
factors[lastFactor]).trace() );
169     }
170
171     void cost_function2(Member_Variables const &mv,
172                       Status             &st)
173     {
174         st.f_value = sqrt( st.frob_tns - 2 * (mv.mttkrp.cwiseProduct(st.factors[lastFactor])).sum() +
175             (mv.cwise_factor_product.cwiseProduct(mv.factor_T_factor[lastFactor])).sum() );
176     }
177
178     /*
179     * Based on each factor's constraint, a different
180     * update function is used at every outer iteration.
181     *
182     * Computes also factor^T * factor at the end.
183     *
184     * @param idx [in]      Factor to be updated.
185     * @param mv   [in]      Struct where ALS variables are stored.
186     * @param st   [in,out] Struct where the returned values of @c Cpd are stored.
187     *                  Updates the @c stl array with the factors.
188     */
189     void update_factor(int             const idx,
190                      Member_Variables &mv,
191                      Status           &st )
192     {
193         // Update factor
194         switch ( st.options.constraints[idx] )
195         {
196             case Constraint::unconstrained:
197             {
198                 st.factors[idx].noalias() = mv.mttkrp[idx] * mv.cwise_factor_product.inverse();
199                 break;
200             }
201             case Constraint::nonnegativity:
202             {
203                 mv.temp_matrix = st.factors[idx];
204                 NesterovMNLs(mv.cwise_factor_product, mv.mttkrp[idx], st.options.nesterov_delta_1,
205                     st.options.nesterov_delta_2, st.factors[idx]);
206                 if(st.factors[idx].cwiseAbs().colwise().sum().minCoeff() == 0)
207                     st.factors[idx] = 0.9 * st.factors[idx] + 0.1 * mv.temp_matrix;
208                 break;
209             }
210             case Constraint::orthogonality:
211             {
212                 mv.temp_matrix = mv.mttkrp[idx].transpose() * mv.mttkrp[idx];
213                 Eigen::SelfAdjointEigenSolver<Matrix> ei_gensolver(mv.temp_matrix);
214                 mv.temp_matrix.noalias() = (ei_gensolver.eigenvalues()
215                     *
216                     (ei_gensolver.eigenvalues().cwiseInverse().cwiseSqrt()).asDiagonal())
217                     * (ei_gensolver.eigenvalues().transpose());
218                 st.factors[idx].noalias() = mv.mttkrp[idx] * mv.temp_matrix;
219                 break;
220             }
221             case Constraint::sparsity:
222                 break;
223             default: // in case of Constraint::constant
224                 break;
225         }
226
227         // Compute A^T * A + B^T * B + ...
228         mv.factor_T_factor[idx].noalias() = st.factors[idx].transpose() * st.factors[idx];
229     }
230
231     /*
232     * @brief Line Search Acceleration
233     *
234     * Performs an acceleration step on the updated factors, and keeps the accelerated factors
235     * when the step succeeds. Otherwise, the acceleration step is ignored.
236     * Line Search Acceleration reduces the number of outer iterations in the ALS algorithm.
237     *
238     * @note This implementation ONLY, if factors are of @c Matrix type.
239     *
240     * @param mv [in,out] Struct where ALS variables are stored.
241     *                  In case the acceleration step is successful the Gramian
242     *                  matrices of factors are updated.
243     * @param st [in,out] Struct where the returned values of @c Cpd are stored.
244     *                  If the acceleration succeeds updates @c factors
245     *                  and cost function value.
246     */
247     void line_search_accel(Member_Variables &mv,
248                          Status           &st)

```

```

249     {
250         double f_accel = 0.0; // Objective Value after the acceleration step
251         double const accel_step = pow(st.ao_iter+1, (1.0/(st.options.accel_coeff)));
252
253         MatrixArray accel_factors;
254         MatrixArray accel_gramians;
255
256         for(std::size_t i=0; i<TnsSize; ++i)
257         {
258             accel_factors[i] = mv.old_factors[i] + accel_step * (st.factors[i] - mv.old_factors[i]);
259             accel_gramians[i] = accel_factors[i].transpose() * accel_factors[i];
260         }
261
262         f_accel = accel_cost_function(mv, st, accel_factors, accel_gramians);
263         if (st.f_value > f_accel)
264         {
265             st.factors = accel_factors;
266             mv.factor_T_factor = accel_gramians;
267             st.f_value = f_accel;
268             Partensor()->Logger()->info("Acceleration Step SUCCEEDED! at iter: {}", st.ao_iter);
269         }
270         else
271             st.options.accel_fail++;
272
273         if (st.options.accel_fail==5)
274         {
275             st.options.accel_fail=0;
276             st.options.accel_coeff++;
277         }
278     }
279
280     /*
281     * Sequential implementation of Alternating Least Squares (ALS) method.
282     *
283     * @param R [in] The rank of decomposition.
284     * @param mv [in] Struct where ALS variables are stored and being updated
285     * until a termination condition is true.
286     * @param st [in,out] Struct where the returned values of @c Cpd are stored.
287     */
288     void als(std::size_t const R,
289             MemberVariables &mv,
290             Status &status)
291     {
292         for (std::size_t i=0; i<TnsSize; i++)
293         {
294             mv.factor_T_factor[i].noalias() = status.factors[i].transpose() * status.factors[i];
295             mv.tns_mat[i] = Matricization(mv.tnsX, i);
296         }
297
298         if(status.options.acceleration)
299         {
300             mv.tnsX_mat_lastFactor_T = mv.tns_mat[lastFactor].transpose();
301         }
302
303         if(status.options.normalization)
304         {
305             choose_normlization_factor(status, mv.all_orthogonal, mv.weight_factor);
306         }
307
308         // Normalize(status.cast<int>(R), mv.factor_T_factor, status.factors);
309         status.frob_tns = square_norm(mv.tnsX);
310         cost_function_init(mv, status);
311         status.rel_costFunction = status.f_value/sqrt(status.frob_tns);
312
313         // ---- Loop until ALS converges ----
314         while(1)
315         {
316             status.ao_iter++;
317             Partensor()->Logger()->info("Iter: {} -- fvalue: {} -- relative_costFunction: {}",
318 status.ao_iter,
319                                     status.f_value, status.rel_costFunction);
320
321             for (std::size_t i=0; i<TnsSize; i++)
322             {
323                 mttkrp(status.factors, mv.tns_mat[i], i, mv.krao[i], mv.mttkrp[i]);
324                 mv.cwise_factor_product = PartialCwiseProd(mv.factor_T_factor, i);
325
326                 // Update factor
327                 update_factor(i, mv, status);
328
329                 // Cost function Computation
330                 if(i == lastFactor)
331                     cost_function(mv, status);
332             }
333
334             status.rel_costFunction = status.f_value/sqrt(status.frob_tns);
335             if(status.options.normalization && !mv.all_orthogonal)

```

```

335         Normalize(mv.weight_factor, static_cast<int>(R), mv.factor_T_factor, status.factors);
336
337
338         // ---- Terminating condition ----
339         if (status.rel_costFunction < status.options.threshold_error || status.ao_iter >=
status.options.max_iter)
340         {
341             if(status.options.writeToFile)
342                 writeFactorsToFile(status);
343             break;
344         }
345
346         if (status.options.acceleration)
347         {
348             mv.norm_factors = status.factors;
349             // ---- Acceleration Step ----
350             if (status.ao_iter > 1)
351                 line_search_accel(mv, status);
352
353             mv.old_factors = mv.norm_factors;
354         }
355     } // end of while
356 }
357
358 /*
359 * Sequential implementation of Alternating Least Squares (ALS) method.
360 * Make use of true factors read from files instead of the Tensor.
361 *
362 * @param R [in] The rank of decomposition.
363 * @param mv [in] Struct where ALS variables are stored and being updated
364 * until a termination condition is true.
365 * @param st [in,out] Struct where the returned values of @c Cpd are stored.
366 */
367 void als_true_factors( std::size_t const R,
368                     MemberVariables &mv,
369                     Status &status )
370 {
371     for (std::size_t i=0; i<TnsSize; i++)
372     {
373         mv.factor_T_factor[i].noalias() = status.factors[i].transpose() * status.factors[i];
374         mv.tns_mat[i] = generateTensor(i, mv.true_factors);
375     }
376
377     if(status.options.acceleration)
378     {
379         mv.tnsX_mat_LastFactor_T = mv.tns_mat[lastFactor].transpose();
380     }
381
382     if(status.options.normalization)
383     {
384         choose_normlization_factor(status, mv.all_orthogonal, mv.weight_factor);
385     }
386
387     // Normalize(static_cast<int>(R), mv.factor_T_factor, status.factors);
388     status.frob_tns = (mv.tns_mat[lastFactor]).squaredNorm();
389     cost_function_init(mv, status);
390     status.rel_costFunction = status.f_value/sqrt(status.frob_tns);
391
392     // ---- Loop until ALS converges ----
393     while(1)
394     {
395         status.ao_iter++;
396         Partensor()->Logger()->info("Iter: {} -- fvalue: {} -- relative_costFunction: {}",
status.ao_iter,
397                                     status.f_value, status.rel_costFunction);
398
399         for (std::size_t i=0; i<TnsSize; i++)
400         {
401             mv.krao[i] = PartialKhatRao(status.factors, i);
402             mv.cwise_factor_product = PartialCwiseProd(mv.factor_T_factor, i);
403             mv.mttkrp[i].noalias() = mv.tns_mat[i] * mv.krao[i];
404
405             // Update factor
406             update_factor(i, mv, status);
407
408             // Cost function Computation
409             if(i == lastFactor)
410                 cost_function(mv, status);
411         }
412
413         status.rel_costFunction = status.f_value/sqrt(status.frob_tns);
414         if(status.options.normalization && !mv.all_orthogonal)
415             Normalize(mv.weight_factor, static_cast<int>(R), mv.factor_T_factor, status.factors);
416
417         // ---- Terminating condition ----
418         if (status.rel_costFunction < status.options.threshold_error || status.ao_iter >=
status.options.max_iter)

```

```

419     {
420         if(status.options.writeToFile)
421             writeFactorsToFile(status);
422         break;
423     }
424
425     if (status.options.acceleration)
426     {
427         mv.norm_factors = status.factors;
428         // ---- Acceleration Step ----
429         if (status.ao_iter > 1)
430             line_search_accel(mv, status);
431
432         mv.old_factors = mv.norm_factors;
433     }
434 } // end of while
435 }
436
437 Status operator()(Tensor_ const &tnsX,
438                 std::size_t const R)
439 {
440     Status status = MakeStatus<Tensor_>();
441     Member_Variab les mv;
442
443     // extract dimensions from tensor
444     Dimensions const &tnsDims = tnsX.dimensions();
445     // produce estimate factors using uniform distribution with entries in [0, 1].
446     makeFactors(tnsDims, status.options.constraints, R, status.factors);
447     mv.tnsX = tnsX;
448     als(R, mv, status);
449
450     return status;
451 }
452
453 Status operator()(Tensor_ const &tnsX,
454                 std::size_t const R,
455                 Options const &options)
456 {
457     Status status(options);
458     Member_Variab les mv;
459
460     // extract dimensions from tensor
461     Dimensions const &tnsDims = tnsX.dimensions();
462     // produce estimate factors using uniform distribution with entries in [0, 1].
463     makeFactors(tnsDims, status.options.constraints, R, status.factors);
464     mv.tnsX = tnsX;
465
466     switch ( status.options.method )
467     {
468     case Method::als:
469     {
470         als(R, mv, status);
471         break;
472     }
473     case Method::rnd:
474         break;
475     case Method::bc:
476         break;
477     default:
478         break;
479     }
480
481     return status;
482 };
483
484 Status operator()(Tensor_ const &tnsX,
485                 std::size_t const R,
486                 MatrixArray const &factorsInit)
487 {
488     Status status = MakeStatus<Tensor_>();
489     Member_Variab les mv;
490
491     status.factors = factorsInit;
492     mv.tnsX = tnsX;
493     als(R, mv, status);
494
495     return status;
496 }
497
498 Status operator()(Tensor_ const &tnsX,
499                 std::size_t const R,
500                 Options const &options,
501                 MatrixArray const &factorsInit)
502 {
503     Status status(options);
504     Member_Variab les mv;
505
506     status.factors = factorsInit;
507     mv.tnsX = tnsX;
508     als(R, mv, status);
509
510     return status;
511 }
512
513 Status operator()(Tensor_ const &tnsX,
514                 std::size_t const R,
515                 Options const &options,
516                 MatrixArray const &factorsInit)
517 {
518     Status status(options);
519     Member_Variab les mv;
520
521     status.factors = factorsInit;
522     mv.tnsX = tnsX;
523     als(R, mv, status);
524
525     return status;
526 }
527
528 Status operator()(Tensor_ const &tnsX,
529                 std::size_t const R,
530                 Options const &options,
531                 MatrixArray const &factorsInit)
532 {
533     Status status(options);
534     Member_Variab les mv;
535
536     status.factors = factorsInit;
537     mv.tnsX = tnsX;
538     als(R, mv, status);
539
540     return status;
541 }
542
543 Status operator()(Tensor_ const &tnsX,
544                 std::size_t const R,
545                 Options const &options,
546                 MatrixArray const &factorsInit)
547 {
548     Status status(options);
549     Member_Variab les mv;
550
551     status.factors = factorsInit;
552     mv.tnsX = tnsX;
553     als(R, mv, status);
554
555     return status;
556 }
557

```

```

558     status.factors = factorsInit;
559     mv.tnsX       = tnsX;
560
561     switch ( status.options.method )
562     {
563     case Method::als:
564     {
565         als(R, mv, status);
566         break;
567     }
568     case Method::rnd:
569         break;
570     case Method::bc:
571         break;
572     default:
573         break;
574     }
575
576     return status;
577 }
578
579 Status operator()(std::array<int, TnsSize> const &tnsDims,
580                 std::size_t const R,
581                 std::string const &path)
582 {
583     using Tensor = Tensor<static_cast<int>(TnsSize)>;
584
585     Status status = MakeStatus<Tensor>();
586     Member_VariabLes mv;
587
588     long long int fileSize = 1;
589     for(auto &dim : tnsDims)
590         fileSize *= static_cast<long long int>(dim);
591
592     mv.tnsX.resize(tnsDims);
593     // Read the whole Tensor from a file
594     read( path,
595          fileSize,
596          0,
597          mv.tnsX );
598     // produce estimate factors using uniform distribution with entries in [0, 1].
599     makeFactors(tnsDims, status.options.constraints, R, status.factors);
600     als(R, mv, status);
601
602     return status;
603 }
604
605 Status operator()(std::array<int, TnsSize> const &tnsDims,
606                 std::size_t const R,
607                 std::string const &path,
608                 Options const &options)
609 {
610     Status status(options);
611     Member_VariabLes mv;
612
613     long long int fileSize = 1;
614     for(auto &dim : tnsDims)
615         fileSize *= static_cast<long long int>(dim);
616
617     mv.tnsX.resize(tnsDims);
618     // Read the whole Tensor from a file
619     read( path,
620          fileSize,
621          0,
622          mv.tnsX );
623     // produce estimate factors using uniform distribution with entries in [0, 1].
624     makeFactors(tnsDims, status.options.constraints, R, status.factors);
625
626     switch ( status.options.method )
627     {
628     case Method::als:
629     {
630         als(R, mv, status);
631         break;
632     }
633     case Method::rnd:
634         break;
635     case Method::bc:
636         break;
637     default:
638         break;
639     }
640
641     return status;
642 }
643
644 Status operator()(std::array<int, TnsSize> const &tnsDims,
645                 const &tnsDims,

```

```

686         std::size_t                               const R,
687         std::array<std::string, TnsSize+1> const &paths)
688     {
689         using Tensor = Tensor<static_cast<int>(TnsSize)>;
690         Status status = MakeStatus<Tensor>();
691         Member_Variables mv;
692
693         long long int fileSize = 1;
694         for(auto &dim : tnsDims)
695             fileSize *= static_cast<long long int>(dim);
696
697         mv.tnsX.resize(tnsDims);
698         // Read the whole Tensor from a file
699         read( paths.front(),
700             fileSize,
701             0,
702             mv.tnsX );
703
704         // Read initialized factors from files
705         for(std::size_t i=0; i<TnsSize; ++i)
706         {
707             status.factors[i] = Matrix(tnsDims[i], static_cast<int>(R));
708             read( paths[i+1],
709                 tnsDims[i]*R,
710                 0,
711                 status.factors[i] );
712         }
713
714         als(R, mv, status);
715
716         return status;
717     }
718
719     Status operator()(std::array<int, TnsSize> const &tnsDims,
720                     std::size_t const R,
721                     std::array<std::string, TnsSize+1> const &paths,
722                     Options const &options)
723     {
724         Status status(options);
725         Member_Variables mv;
726
727         long long int fileSize = 1;
728         for(auto &dim : tnsDims)
729             fileSize *= static_cast<long long int>(dim);
730
731         mv.tnsX.resize(tnsDims);
732         // Read the whole Tensor from a file
733         read( paths.front(),
734             fileSize,
735             0,
736             mv.tnsX );
737
738         // Read initialized factors from files
739         for(std::size_t i=0; i<TnsSize; ++i)
740         {
741             status.factors[i] = Matrix(tnsDims[i], static_cast<int>(R));
742             read( paths[i+1],
743                 tnsDims[i]*R,
744                 0,
745                 status.factors[i] );
746         }
747
748         switch ( status.options.method )
749         {
750             case Method::als:
751             {
752                 als(R, mv, status);
753                 break;
754             }
755             case Method::rnd:
756                 break;
757             case Method::bc:
758                 break;
759             default:
760                 break;
761         }
762
763         return status;
764     }
765
766     Status operator()(std::array<int, TnsSize> const &tnsDims,
767                     std::size_t const R,
768                     std::array<std::string, TnsSize> const &>true_paths,
769                     std::array<std::string, TnsSize> const &ini_paths,
770                     Options const &options)
771     {
772         Status status(options);

```

```

815     Member_Variab les mv;
816
817     for(std::size_t i=0; i<TnsSize; ++i)
818     {
819         mv.true_factors[i] = Matrix(tnsDims[i], stati c_cast<int>(R));
820         status.factors[i] = Matrix(tnsDims[i], stati c_cast<int>(R));
821
822         // Read ini tialized factors from files
823         read( true_paths[i],
824             tnsDi ms[i]*R,
825             0,
826             mv.true_factors[i] );
827         // Read ini tialized factors from files
828         read( ini t_paths[i],
829             tnsDi ms[i]*R,
830             0,
831             status.factors[i] );
832     }
833
834     swi tch ( status.opti ons.method )
835     {
836     case Method::als:
837     {
838         als_true_factors(R, mv, status);
839         break;
840     }
841     case Method::rnd:
842     {
843         break;
844     }
845     case Method::bc:
846     {
847         break;
848     }
849     default:
850     {
851         break;
852     }
853     }
854     return status;
855 }
856
857 #if USE_MPI
858 #include "CpdMpi .hpp"
859 #endif /* USE_MPI */
860
861 #if USE_OPENMP
862 #include "CpdOpenMP .hpp"
863 #endif /* USE_OPENMP */
864
865 #if USE_CUDA
866 #include "CUDA/CpdCUDA .hpp"
867 #endif /* USE_CUDA */
868
869 namespace partensor
870 {
871     template <typename Tensor_, typename ExecutionPol icy>
872     cpd( ExecutionPol icy &&,
873         Tensor_ const &tnsX,
874         std::size_t const R )
875     {
876         using ExPol icy = execution::execution_pol icy<ExecutionPol icy>;
877
878         if constexpr (std::is_same_v<ExPol icy, execution::sequenced_pol icy>)
879         {
880             return internal::CPD<Tensor_>(tnsX, R);
881         }
882         else if constexpr (std::is_same_v<ExPol icy, execution::openmpi_pol icy>)
883         {
884             return internal::CPD<Tensor_, execution::openmpi_pol icy>(tnsX, R);
885         }
886         else if constexpr (std::is_same_v<ExPol icy, execution::openmp_pol icy>)
887         {
888             return internal::CPD<Tensor_, execution::openmp_pol icy>(tnsX, R);
889         }
890         else if constexpr (std::is_same_v<ExPol icy, execution::cuda_pol icy>)
891         {
892             return internal::CPD<Tensor_, execution::cuda_pol icy>(tnsX, R);
893         }
894         else
895         {
896             return internal::CPD<Tensor_>(tnsX, R);
897         }
898     }
899 }

```

```

920
921 /*
922  * Interface of Canonical Polyadic Decomposition(cpd). Sequential Policy.
923  *
924  * @tparam Tensor_      Type(data type and order) of input Tensor.
925  *                      @c Tensor_ must be @c partensor::Tensor<order>, where
926  *                      @c order must be in range of @c [3-8].
927  * @param tnsX      [in] The given Tensor to be factorized of @c Tensor_ type,
928  *                      with @c double data.
929  * @param R      [in] The rank of decomposition.
930  *
931  * @returns An object of type @c Status, containing the results of the algorithm.
932  */
933 template<typename Tensor_>
934 auto cpd(Tensor_ const &tnsX,
935          std::size_t const R)
936 {
937     return internal::CPD<Tensor_, execution::sequenced_policy>()(tnsX, R);
938 }
939
940 template<typename Tensor_, typename ExecutionPolicy>
941 execution::internal::enable_if_execution_policy<ExecutionPolicy, Status<Tensor_, execution::execution_policy_t<ExecutionPolicy>>>
942 cpd(ExecutionPolicy &&,
943     Tensor_ const &tnsX,
944     std::size_t const R,
945     Options<Tensor_, execution::execution_policy_t<ExecutionPolicy>, DefaultValues> const &options )
946 {
947     using ExPolicy = execution::execution_policy_t<ExecutionPolicy>;
948
949     if constexpr (std::is_same_v<ExPolicy, execution::sequenced_policy>)
950     {
951         return internal::CPD<Tensor_>()(tnsX, R, options);
952     }
953     else if constexpr (std::is_same_v<ExPolicy, execution::openmpi_policy>)
954     {
955         return internal::CPD<Tensor_, execution::openmpi_policy>()(tnsX, R, options);
956     }
957     else if constexpr (std::is_same_v<ExPolicy, execution::openmp_policy>)
958     {
959         return internal::CPD<Tensor_, execution::openmp_policy>()(tnsX, R, options);
960     }
961     else if constexpr (std::is_same_v<ExPolicy, execution::cuda_policy>)
962     {
963         return internal::CPD<Tensor_, execution::cuda_policy>()(tnsX, R, options);
964     }
965     else
966         return internal::CPD<Tensor_>()(tnsX, R, options);
967 }
968
969 /*
970  * Interface of Canonical Polyadic Decomposition(cpd). Sequential Policy.
971  *
972  * @tparam Tensor_      Type(data type and order) of input Tensor.
973  *                      @c Tensor_ must be @c partensor::Tensor<order>, where
974  *                      @c order must be in range of @c [3-8].
975  * @param tnsX      [in] The given Tensor to be factorized of @c Tensor_ type,
976  *                      with @c double data.
977  * @param R      [in] The rank of decomposition.
978  * @param options [in] User's @c options, other than the default. It must be of
979  *                      @c partensor::Options<partensor::Tensor<order>> type,
980  *                      where @c order must be in range of @c [3-8].
981  *
982  * @returns An object of type @c Status with the results of the algorithm.
983  */
984 template<typename Tensor_>
985 auto cpd(Tensor_ const &tnsX,
986          std::size_t const R,
987          Options<Tensor_> const &options)
988 {
989     return internal::CPD<Tensor_, execution::sequenced_policy>()(tnsX, R, options);
990 }
991
992 template<typename ExecutionPolicy, typename Tensor_>
993 execution::internal::enable_if_execution_policy<ExecutionPolicy, Status<Tensor_, execution::execution_policy_t<ExecutionPolicy>>>
994 cpd(ExecutionPolicy &&,
995     Tensor_ const &tnsX,
996     std::size_t const R,
997     MatrixArray<Tensor_> const &factorsInit )
998 {
999     using ExPolicy = execution::execution_policy_t<ExecutionPolicy>;
1000
1001     if constexpr (std::is_same_v<ExPolicy, execution::sequenced_policy>)
1002     {
1003         return internal::CPD<Tensor_>()(tnsX, R, factorsInit);
1004     }
1005 }

```



```

1051     else if constexpr (std::is_same_v<ExPolicy, execution::openmpi_policy>)
1052     {
1053         return internal::CPD<Tensor_, execution::openmpi_policy>()(tnsX, R, factorslnit);
1054     }
1055     else if constexpr (std::is_same_v<ExPolicy, execution::openmp_policy>)
1056     {
1057         return internal::CPD<Tensor_, execution::openmp_policy>()(tnsX, R, factorslnit);
1058     }
1059     else if constexpr (std::is_same_v<ExPolicy, execution::cuda_policy>)
1060     {
1061         return internal::CPD<Tensor_, execution::cuda_policy>()(tnsX, R, factorslnit);
1062     }
1063     else
1064         return internal::CPD<Tensor_>()(tnsX, R, factorslnit);
1065 }
1066
1067 /*
1068 * Interface of Canonical Polyadic Decomposition(cpd). Sequential Policy.
1069 *
1070 * @tparam Tensor_      Type(data type and order) of input Tensor.
1071 *                      @c Tensor_ must be @c partensor::Tensor<order>, where
1072 *                      @c order must be in range of @c [3-8].
1073 * @param tnsX          [in] The given Tensor to be factorized of @c Tensor_ type,
1074 *                      with @c double data.
1075 * @param R             [in] The rank of decomposition.
1076 * @param factorslnit  [in] Uses initialized factors instead of randomly generated. The
1077 *                      data must be of @c partensor::Matrix type and stored in an
1078 *                      @c stl array with size same as the @c order of @c tnsX.
1079 *
1080 * @returns An object of type @c Status with the results of the algorithm.
1081 */
1082 template<typename Tensor_>
1083 auto cpd(Tensor_ const &tnsX,
1084         std::size_t const R,
1085         MatrixArray<Tensor_> const &factorslnit)
1086 {
1087     return internal::CPD<Tensor_, execution::sequenced_policy>()(tnsX, R, factorslnit);
1088 }
1089
1090 template <typename ExecutionPolicy, typename Tensor_>
1091 execution::internal::enable_if_execution_policy<ExecutionPolicy, Status<Tensor_, execution::execution_policy_t<ExecutionPolicy>> &&
1092 cpd( ExecutionPolicy const &&,
1093     Tensor_ const &tnsX,
1094     std::size_t const R,
1095     Options<Tensor_, execution::execution_policy_t<ExecutionPolicy>, DefaultValues> const &options,
1096     MatrixArray<Tensor_> const &factorslnit )
1097 {
1098     using ExPolicy = execution::execution_policy_t<ExecutionPolicy>;
1099
1100     if constexpr (std::is_same_v<ExPolicy, execution::sequenced_policy>)
1101     {
1102         return internal::CPD<Tensor_>()(tnsX, R, options, factorslnit);
1103     }
1104     else if constexpr (std::is_same_v<ExPolicy, execution::openmpi_policy>)
1105     {
1106         return internal::CPD<Tensor_, execution::openmpi_policy>()(tnsX, R, options, factorslnit);
1107     }
1108     else if constexpr (std::is_same_v<ExPolicy, execution::openmp_policy>)
1109     {
1110         return internal::CPD<Tensor_, execution::openmp_policy>()(tnsX, R, options, factorslnit);
1111     }
1112     else if constexpr (std::is_same_v<ExPolicy, execution::cuda_policy>)
1113     {
1114         return internal::CPD<Tensor_, execution::cuda_policy>()(tnsX, R, options, factorslnit);
1115     }
1116     else
1117         return internal::CPD<Tensor_>()(tnsX, R, options, factorslnit);
1118 }
1119
1120 /*
1121 * Interface of Canonical Polyadic Decomposition(cpd). Sequential Policy.
1122 *
1123 * @tparam Tensor_      Type(data type and order) of input Tensor.
1124 *                      @c Tensor_ must be @c partensor::Tensor<order>, where
1125 *                      @c order must be in range of @c [3-8].
1126 * @param tnsX          [in] The given Tensor to be factorized of @c Tensor_ type,
1127 *                      with @c double data.
1128 * @param R             [in] The rank of decomposition.
1129 * @param options       [in] User's @c options, other than the default. It must be of
1130 *                      @c partensor::Options<partensor::Tensor<order>> type,
1131 *                      where @c order must be in range of @c [3-8].
1132 * @param factorslnit  [in] Uses initialized factors instead of randomly generated. The
1133 *                      data must be of @c partensor::Matrix type and stored in an
1134 *                      @c stl array with size same as the @c order of @c tnsX.
1135 *
1136 * @returns An object of type @c Status with the results of the algorithm.
1137 */

```

```

1163  */
1164  template<typename Tensor_>
1165  auto cpd(Tensor_ &tnsX,
1166          std::size_t R,
1167          Options<Tensor_> &opti ons,
1168          MatrixArray<Tensor_> &factorsI ni t)
1169  {
1170      return i nternal::CPD<Tensor_>(tnsX, R, opti ons, factorsI ni t);
1171  }
1172
1173  template <typename Executi onPol i cy, std::size_t TnsSi ze>
1174
1175  executi on::i nternal::enabl e_i f_executi on_pol i cy<Executi onPol i cy, Status<Tensor<stati c_cast<i nt>(TnsSi ze)>, executi on::ex
1176  cpd( Executi onPol i cy &&,
1177      std::array<i nt, TnsSi ze> const &tnsDi ms,
1178      std::size_t R,
1179      std::stri ng const &path )
1180  {
1181      usi ng ExPol i cy = executi on::executi on_pol i cy_t<Executi onPol i cy>;
1182      usi ng Tensor_ = Tensor<stati c_cast<i nt>(TnsSi ze)>;
1183
1184      i f constexpr (std::i s_same_v<ExPol i cy, executi on::sequenced_pol i cy>)
1185      {
1186          return i nternal::CPD<Tensor_>(tnsDi ms, R, path);
1187      }
1188      el se i f constexpr (std::i s_same_v<ExPol i cy, executi on::openmpi _pol i cy>)
1189      {
1190          return i nternal::CPD<Tensor_>(tnsDi ms, R, path);
1191      }
1192      el se i f constexpr (std::i s_same_v<ExPol i cy, executi on::openmp_pol i cy>)
1193      {
1194          return i nternal::CPD<Tensor_>(tnsDi ms, R, path);
1195      }
1196      el se i f constexpr (std::i s_same_v<ExPol i cy, executi on::cuda_pol i cy>)
1197      {
1198          return i nternal::CPD<Tensor_>(tnsDi ms, R, path);
1199      }
1200      el se
1201          return i nternal::CPD<Tensor_>(tnsDi ms, R, path);
1202  }
1203
1204  /*
1205  * I nterface of Canoni cal Pol yadi c Decomposi ti on(cpd). Sequenti al Pol i cy.
1206  * Wi th thi s versi on of @c cpd, the Tensor can be read from a fi le, speci fi ed i n
1207  * @c path variabl e.
1208  *
1209  * @tparam TnsSi ze      Order of i nput Tensor.
1210  *
1211  * @param tnsDi ms      [i n] @c Stl array containi ng the Tensor dimensi ons, whose
1212  *                      length must be same as the Tensor order.
1213  * @param R              [i n] The rank of decomposi ti on.
1214  * @param path           [i n] The path where the tensor is l ocat ed.
1215  *
1216  * @returns An obj ect of type @c Status wi th the resul ts of the al gori thm.
1217  */
1218  template<std::size_t TnsSi ze>
1219  auto cpd(std::array<i nt, TnsSi ze> const &tnsDi ms,
1220          std::size_t R,
1221          std::stri ng const &path )
1222  {
1223      usi ng Tensor_ = Tensor<stati c_cast<i nt>(TnsSi ze)>;
1224      return i nternal::CPD<Tensor_>(tnsDi ms, R, path);
1225  }
1226
1227  template <typename Executi onPol i cy, std::size_t TnsSi ze>
1228
1229  executi on::i nternal::enabl e_i f_executi on_pol i cy<Executi onPol i cy, Status<Tensor<stati c_cast<i nt>(TnsSi ze)>, executi on::ex
1230  cpd( Executi onPol i cy &&,
1231      std::array<i nt, TnsSi ze> const &tnsDi ms,
1232      std::size_t R,
1233      std::stri ng const &path,
1234      Opti ons<Tensor<stati c_cast<i nt>(TnsSi ze)>, executi on::executi on_pol i cy_t<Executi onPol i cy>, Defaul tVal ues>
1235      const &opti ons )
1236  {
1237      usi ng ExPol i cy = executi on::executi on_pol i cy_t<Executi onPol i cy>;
1238      usi ng Tensor_ = Tensor<stati c_cast<i nt>(TnsSi ze)>;
1239
1240      i f constexpr (std::i s_same_v<ExPol i cy, executi on::sequenced_pol i cy>)
1241      {
1242          return i nternal::CPD<Tensor_>(tnsDi ms, R, path, opti ons);
1243      }
1244      el se i f constexpr (std::i s_same_v<ExPol i cy, executi on::openmpi _pol i cy>)
1245      {
1246          return i nternal::CPD<Tensor_>(tnsDi ms, R, path, opti ons);
1247      }
1248      el se i f constexpr (std::i s_same_v<ExPol i cy, executi on::openmp_pol i cy>)
1249      {
1250          return i nternal::CPD<Tensor_>(tnsDi ms, R, path, opti ons);
1251      }
1252      el se i f constexpr (std::i s_same_v<ExPol i cy, executi on::openmp_pol i cy>)
1253      {
1254          return i nternal::CPD<Tensor_>(tnsDi ms, R, path, opti ons);
1255      }
1256  }

```

```

1293     {
1294         return internal::CPD<Tensor_, execution::openmp_policy>() (tnsDims, R, path, options);
1295     }
1296     else if constexpr (std::is_same_v<ExPolicy, execution::cuda_policy>)
1297     {
1298         return internal::CPD<Tensor_, execution::cuda_policy>() (tnsDims, R, path, options);
1299     }
1300     else
1301         return internal::CPD<Tensor_>() (tnsDims, R, path, options);
1302 }
1303
1304 /*
1305 * Interface of Canonical Polyadic Decomposition(cpd). Sequential Policy.
1306 * With this version of @c cpd, the Tensor can be read from a file, specified in
1307 * @c path variable.
1308 *
1309 * @tparam TnsSize      Order of input Tensor.
1310 *
1311 * @param tnsDims      [in] @c STL array containing the Tensor dimensions, whose
1312 *                      length must be same as the Tensor order.
1313 * @param R            [in] The rank of decomposition.
1314 * @param path         [in] The path where the tensor is located.
1315 * @param options      [in] User's @c options, other than the default. It must be of
1316 *                      @c partensor::Options<partensor::Tensor<order> type,
1317 *                      where @c order must be in range of @c [3-8].
1318 *
1319 * @returns An object of type @c Status with the results of the algorithm.
1320 */
1321 template<std::size_t TnsSize>
1322 auto cpd(std::array<int, TnsSize>          const &tnsDims,
1323         std::size_t                      const R,
1324         std::string                      const &path,
1325         Options<Tensor<static_cast<int>(TnsSize)>> const &options )
1326 {
1327     using Tensor_ = Tensor<static_cast<int>(TnsSize)>;
1328     return internal::CPD<Tensor_, execution::sequenced_policy>() (tnsDims, R, path, options);
1329 }
1330
1331
1332
1333
1334
1335 template <typename ExecutionPolicy, std::size_t TnsSize>
1336
1337 execution::internal::enable_if_execution_policy<ExecutionPolicy, Status<Tensor<static_cast<int>(TnsSize)>, execution::internal::
1338 cpd( ExecutionPolicy          &&,
1339      std::array<int, TnsSize>          const &tnsDims,
1340      std::size_t                      const R,
1341      std::array<std::string, TnsSize+1> const &paths )
1342 {
1343     using ExPolicy = execution::execution_policy_t<ExecutionPolicy>;
1344     using Tensor_ = Tensor<static_cast<int>(TnsSize)>;
1345
1346     if constexpr (std::is_same_v<ExPolicy, execution::sequenced_policy>)
1347     {
1348         return internal::CPD<Tensor_>() (tnsDims, R, paths);
1349     }
1350     else if constexpr (std::is_same_v<ExPolicy, execution::openmpi_policy>)
1351     {
1352         return internal::CPD<Tensor_, execution::openmpi_policy>() (tnsDims, R, paths);
1353     }
1354     else if constexpr (std::is_same_v<ExPolicy, execution::openmp_policy>)
1355     {
1356         return internal::CPD<Tensor_, execution::openmp_policy>() (tnsDims, R, paths);
1357     }
1358     else if constexpr (std::is_same_v<ExPolicy, execution::cuda_policy>)
1359     {
1360         return internal::CPD<Tensor_, execution::cuda_policy>() (tnsDims, R, paths);
1361     }
1362     else
1363         return internal::CPD<Tensor_>() (tnsDims, R, paths);
1364 }
1365
1366 /*
1367 * Interface of Canonical Polyadic Decomposition(cpd). Sequential Policy.
1368 * With this version of @c cpd, the Tensor can be read from a file, specified in
1369 * @c path variable.
1370 *
1371 * @tparam TnsSize      Order of input Tensor.
1372 *
1373 * @param tnsDims      [in] @c STL array containing the Tensor dimensions, whose
1374 *                      length must be same as the Tensor order.
1375 * @param R            [in] The rank of decomposition.
1376 * @param paths        [in] An @c stl array containing paths for the Tensor to be
1377 *                      factorized and after that the paths for the initialized
1378 *                      factors.
1379 *
1380 * @returns An object of type @c Status with the results of the algorithm.
1381 */
1382 template<std::size_t TnsSize>

```

```

1403 auto cpd(std::array<int, TnsSi ze>          const &tnsDi ms,
1404          std::size_t                       const R,
1405          std::array<std::string, TnsSi ze+1> const &paths )
1406 {
1407     using Tensor_ = Tensor<static_cast<int>(TnsSi ze)>;
1408     return internal::CPD<Tensor_, execution::sequenced_pol icy>()(tnsDi ms, R, paths);
1409 }
1410
1438 template <std::size_t TnsSi ze, typename ExecutionPol icy>
1439
1440 execution::internal::enable_if_execution_pol icy<ExecutionPol icy, Status<Tensor<static_cast<int>(TnsSi ze)>, execution::exe
1441 cpd( ExecutionPol icy &&,
1442      std::array<int, TnsSi ze>          const &tnsDi ms,
1443      std::size_t                       const R,
1444      std::array<std::string, TnsSi ze+1> const &paths,
1445
1446      Options<Tensor<static_cast<int>(TnsSi ze)>, execution::execution_pol icy_t<ExecutionPol icy>, DefaultVal ues>
1447      const &options )
1448 {
1449     using ExPol icy = execution::execution_pol icy_t<ExecutionPol icy>;
1450     using Tensor_ = Tensor<static_cast<int>(TnsSi ze)>;
1451
1452     if constexpr (std::is_same_v<ExPol icy, execution::sequenced_pol icy>)
1453     {
1454         return internal::CPD<Tensor_>()(tnsDi ms, R, paths, options);
1455     }
1456     else if constexpr (std::is_same_v<ExPol icy, execution::openmpi_pol icy>)
1457     {
1458         return internal::CPD<Tensor_, execution::openmpi_pol icy>()(tnsDi ms, R, paths, options);
1459     }
1460     else if constexpr (std::is_same_v<ExPol icy, execution::openmp_pol icy>)
1461     {
1462         return internal::CPD<Tensor_, execution::openmp_pol icy>()(tnsDi ms, R, paths, options);
1463     }
1464     else if constexpr (std::is_same_v<ExPol icy, execution::cuda_pol icy>)
1465     {
1466         return internal::CPD<Tensor_, execution::cuda_pol icy>()(tnsDi ms, R, paths, options);
1467     }
1468     else
1469         return internal::CPD<Tensor_>()(tnsDi ms, R, paths, options);
1470 }
1471
1472 /*
1473 * Interface of Canonical Polyadic Decomposition(cpd). Sequential Pol icy.
1474 * With this version of @c cpd, the Tensor can be read from a file, specified in
1475 * @c path variable.
1476 *
1477 * @tparam TnsSi ze          Order of input Tensor.
1478 *
1479 * @param tnsDi ms          [in] @c Stl array containing the Tensor dimensions, whose
1480 *                          length must be same as the Tensor order.
1481 * @param R                [in] The rank of decomposition.
1482 * @param paths            [in] An @c stl array containing paths for the Tensor to be
1483 *                          factorized and after that the paths for the initialized
1484 *                          factors.
1485 * @param options          [in] User's @c options, other than the default. It must be of
1486 *                          @c partensor::Options<partensor::Tensor<order> type,
1487 *                          where @c order must be in range of @c [3-8].
1488 *
1489 * @returns An object of type @c Status with the results of the algorithm.
1490 */
1491
1492 template<std::size_t TnsSi ze>
1493 auto cpd(std::array<int, TnsSi ze>          const &tnsDi ms,
1494          std::size_t                       const R,
1495          std::array<std::string, TnsSi ze+1> const &paths,
1496          Options<Tensor<static_cast<int>(TnsSi ze)> const &options )
1497 {
1498     using Tensor_ = Tensor<static_cast<int>(TnsSi ze)>;
1499     return internal::CPD<Tensor_, execution::sequenced_pol icy>()(tnsDi ms, R, paths, options);
1500 }
1501
1502 template <std::size_t TnsSi ze, typename ExecutionPol icy>
1503
1504 execution::internal::enable_if_execution_pol icy<ExecutionPol icy, Status<Tensor<static_cast<int>(TnsSi ze)>, execution::exe
1505 cpd( ExecutionPol icy &&,
1506      std::array<int, TnsSi ze>          const &tnsDi ms,
1507      std::size_t                       const R,
1508      std::array<std::string, TnsSi ze> const &true_paths,
1509      std::array<std::string, TnsSi ze> const &ini t_paths,
1510
1511      Options<Tensor<static_cast<int>(TnsSi ze)>, execution::execution_pol icy_t<ExecutionPol icy>, DefaultVal ues>
1512      const &options )
1513 {
1514     using ExPol icy = execution::execution_pol icy_t<ExecutionPol icy>;
1515     using Tensor_ = Tensor<static_cast<int>(TnsSi ze)>;
1516
1517     if constexpr (std::is_same_v<ExPol icy, execution::sequenced_pol icy>)

```

```

1538     {
1539         return internal::CPD<Tensor_>()(tnsDims, R, true_paths, ini_t_paths, options);
1540     }
1541     else if constexpr (std::is_same_v<ExPolicy, execution::openmpi_policy>)
1542     {
1543         return
1544         internal::CPD<Tensor_, execution::openmpi_policy>()(tnsDims, R, true_paths, ini_t_paths, options);
1545     }
1546     else if constexpr (std::is_same_v<ExPolicy, execution::openmp_policy>)
1547     {
1548         return
1549         internal::CPD<Tensor_, execution::openmp_policy>()(tnsDims, R, true_paths, ini_t_paths, options);
1550     }
1551     else if constexpr (std::is_same_v<ExPolicy, execution::cuda_policy>)
1552     {
1553         return
1554         internal::CPD<Tensor_, execution::cuda_policy>()(tnsDims, R, true_paths, ini_t_paths, options);
1555     }
1556     }
1557     /*
1558     * Interface of Canonical Polyadic Decomposition(cpd). Sequential Policy.
1559     * With this version of @c cpd, the Tensor can be read from a file, specified in
1560     * @c path variable.
1561     *
1562     * @tparam TnsSize      Order of input Tensor.
1563     *
1564     * @param tnsDims      [in] @c Stl array containing the Tensor dimensions, whose
1565     *                      length must be same as the Tensor order.
1566     * @param R            [in] The rank of decomposition.
1567     * @param true_paths  [in] An @c stl array containing paths for the true factors.
1568     * @param ini_t_paths [in] An @c stl array containing paths for initialized
1569     *                      factors.
1570     * @param options     [in] User's @c options, other than the default. It must be of
1571     *                      @c partensor::Options<partensor::Tensor<order> type,
1572     *                      where @c order must be in range of @c [3-8].
1573     *
1574     * @returns An object of type @c Status with the results of the algorithm.
1575     */
1576     template<std::size_t TnsSize>
1577     auto cpd(std::array<int, TnsSize>          const &tnsDims,
1578            std::size_t                        const R,
1579            std::array<std::string, TnsSize>   const &>true_paths,
1580            std::array<std::string, TnsSize>   const &ini_t_paths,
1581            Options<Tensor<static_cast<int>(TnsSize)>> const &options )
1582     {
1583         using Tensor_ = Tensor<static_cast<int>(TnsSize)>;
1584         return
1585         internal::CPD<Tensor_, execution::sequenced_policy>()(tnsDims, R, true_paths, ini_t_paths, options);
1586     }
1587 }
1588
1589 #endif // PARTENSOR_CPD_HPP

```

8.7 CpdDimTree.hpp File Reference

```

#include "PARTENSOR_basics.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 "Matrixization.hpp"
#include "PartialCwiseProd.hpp"
#include "PartialKhatRao.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]

```

executi on: : internal: : enable_i f_executi on_poli cy< Executi onPoli cy, Status< Tensor< stati c_ -
cast< int >(TnsSize) >, executi on: : executi on_poli cy_t< Executi onPoli cy >, Defaul tVal ues > >
partensor: : cpdDi mTree (
    Executi onPoli cy && ,
    std: : array< int, TnsSize > const & tnsDi ms,
    std: : si ze_t const R,
    std: : array< std: : stri ng, TnsSi ze+1 > const & paths )

```

Interface of Canonical Polyadic Decomposition(cpd) with the use of Dimension Trees. Also, an Executi onPoli cy can be used, which can be either sequenti al or paral lel with the use of MPI. In order to choose a policy, type executi on: : seq or executi on: : mpi. Default value is sequenti al, in case no Executi onPoli cy is passed.

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

Template Parameters

<i>ExecutionPolicy</i>	Type of stl Executi onPoli cy (sequential, parallel-mpi).
<i>TnsSize</i>	Order of the input Tensor.

Parameters

<i>tnsDims</i>	[in] Stl array containing the Tensor dimensions, whose length must be same as the Tensor order.
<i>R</i>	[in] The rank of decomposition.
<i>paths</i>	[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]

```

executi on: : internal: : enable_i f_executi on_poli cy< Executi onPoli cy, Status< Tensor< stati c_ -
cast< int >(TnsSize) >, executi on: : executi on_poli cy_t< Executi onPoli cy >, Defaul tVal ues > >
partensor: : cpdDi mTree (
    Executi onPoli cy && ,
    std: : array< int, TnsSize > const & tnsDi ms,
    std: : si ze_t const R,
    std: : array< std: : stri ng, TnsSi ze+1 > const & paths,
    Opti ons< Tensor< stati c_ cast< int >(TnsSize) >, executi on: : executi on_poli cy_t<
Executi onPoli cy >, Defaul tVal ues > const & opti ons )

```

Interface of Canonical Polyadic Decomposition(cpd) with the use of Dimension Trees. Also, an Executi onPoli cy can be used, which can be either sequenti al or paral lel with the use of MPI. In order to choose a policy, type executi on: : seq or executi on: : mpi. Default value is sequenti al, in case no Executi onPoli cy is passed.

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

Template Parameters

<i>ExecutionPolicy</i>	Type of <code>std::ExecutionPolicy</code> (sequential, parallel-mpi).
<i>TnsSize</i>	Order of the input Tensor.

Parameters

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

<i>ExecutionPolicy</i>	Type of <code>std::ExecutionPolicy</code> (sequential, parallel-mpi).
<i>TnsSize</i>	Order of the input Tensor.

Parameters

<i>tnsDims</i>	[in] <code>std::array</code> containing the Tensor dimensions, whose length must be same as the Tensor order.
<i>R</i>	[in] The rank of decomposition.
<i>path</i>	[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]

```

executi on: : internal: : enable_i f_executi on_poli cy< Executi onPoli cy, Status< Tensor< stati c_
cast< int >(TnsSi ze) >, executi on: : executi on_poli cy_t< Executi onPoli cy >, Defaul tVal ues > >
partensor: : cpdDi mTree (
    Executi onPoli cy && ,
    std: : array< int, TnsSi ze > const & tnsDi ms,
    std: : si ze_t const R,
    std: : string const & path,
    Opti ons< Tensor< stati c_cast< int >(TnsSi ze) >, executi on: : executi on_poli cy_t<
Executi onPoli cy >, Defaul tVal ues > const & opti ons )

```

Interface of Canonical Polyadic Decomposition(cpd) with the use of Dimension Trees. Also, an `Executi onPoli cy` can be used, which can be either sequential or parallel with the use of MPI. In order to choose a policy, type `executi on: : seq` or `executi on: : mpi`. Default value is sequential, in case no `Executi onPoli cy` is passed.

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

Template Parameters

<i>ExecutionPolicy</i>	Type of stl <code>Executi onPoli cy</code> (sequential, parallel-mpi).
<i>TnsSize</i>	Order of the input Tensor.

Parameters

<i>tnsDims</i>	[in] Stl array containing the Tensor dimensions, whose length must be same as the Tensor order.
<i>R</i>	[in] The rank of decomposition.
<i>path</i>	[in] The path where the tensor is located.
<i>options</i>	[in] User's opti ons, other than the default. It must be of <code>partensor: : Opti ons<partensor: : Tensor<order >></code> 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]

```

executi on: : internal: : enable_i f_executi on_poli cy< Executi onPoli cy, Status< Tensor_, executi on_
: : executi on_poli cy_t< Executi onPoli cy >, Defaul tVal ues > > partensor: : cpdDi mTree (

```

```

    ExecutionPolicy && ,
    Tensor_ const & tnsX,
    std::size_t const R )

```

Interface of Canonical Polyadic Decomposition(cpd) with the use of Dimension Trees. Also, an ExecutionPolicy 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

<i>ExecutionPolicy</i>	Type of std::ExecutionPolicy (sequential, parallel-mpi).
<i>Tensor_</i>	Type(data type and order) of input Tensor. Tensor_ must be partensor: : Tensor<order>, where order must be in range of [3-8].

Parameters

<i>tnsX</i>	[in] The given Tensor to be factorized of Tensor_ type, with double data.
<i>R</i>	[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 ExecutionPolicy 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

<i>ExecutionPolicy</i>	Type of std::ExecutionPolicy (sequential, parallel-mpi).
<i>Tensor_</i>	Type(data type and order) of input Tensor.
<i>MatrixArray_</i>	An std::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

<i>tnsX</i>	[in] The given Tensor to be factorized of Tensor_ type, with double data.
-------------	---

Parameters

<i>R</i>	[in] The rank of decomposition.
<i>factorsInit</i>	[in] Uses initialized factors instead of randomly generated. The data can be either <code>partensor::Matrix</code> , or <code>partensor::Tensor<2></code> type and stored in an <code>std::array</code> with size same as the order of <code>tnsX</code> .

Returns

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

8.7.2.7 cpdDimTree() [7/8]

```

executi on:: internal:: enable_if_executi on_policy< Executi onPolicy, Status< Tensor_, executi on_
:: executi on_policy_t< Executi onPolicy >, DefaultValues >> partensor:: cpdDi mTree (
    Executi onPolicy &&,
    Tensor_ const & tnsX,
    std:: size_t const R,
    Opti ons< Tensor_, executi on:: executi on_policy_t< Executi onPolicy >, DefaultValues
> const & opti ons )

```

Interface of Canonical Polyadic Decomposition(cpd) with the use of Dimension Trees. Also, an `Executi on Policy` can be used, which can be either sequential or parallel with the use of MPI. In order to choose a policy, type `executi on:: seq` or `executi on:: mpi`. Default value is sequential, in case no `Executi onPolicy` is passed.

Template Parameters

<i>ExecutionPolicy</i>	Type of <code>std:: Executi on Policy</code> (sequential, parallel-mpi).
<i>Tensor_</i>	Type(data type and order) of input Tensor.

Parameters

<i>tnsX</i>	[in] The given Tensor to be factorized of <code>Tensor_</code> type, with double data.
<i>R</i>	[in] The rank of decomposition.
<i>options</i>	[in] User's opti ons, other than the default. It must be of <code>partensor:: Opti ons<partensor:: Tensor<order >></code> 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]

```

executi on:: internal:: enable_if_executi on_policy< Executi onPolicy, Status< Tensor_, executi on_
:: executi on_policy_t< Executi onPolicy >, DefaultValues >> partensor:: cpdDi mTree (

```

```

    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 ExecutionPolicy can be used, which can be either sequential or parallel with the use of MPI. In order to choose a policy, type execution_policy_t::seq or execution_policy_t::mpi. Default value is sequential, in case no ExecutionPolicy is passed.

Template Parameters

<i>ExecutionPolicy</i>	Type of stl ExecutionPolicy (sequential, parallel-mpi).
<i>Tensor_</i>	Type(data type and order) of input Tensor.
<i>MatrixArray_</i>	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

<i>tnsX</i>	[in] The given Tensor to be factorized of Tensor_ type, with double data.
<i>R</i>	[in] The rank of decomposition.
<i>options</i>	[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].
<i>factorsInit</i>	[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.](#)

```

1 #ifndef DOXYGEN_SHOULD_SKIP_THIS
15 #endif // DOXYGEN_SHOULD_SKIP_THIS
16 /*****
27 #ifndef PARTENSOR_CPD_DIMTREE_DIMTREE_HPP
28 #define PARTENSOR_CPD_DIMTREE_DIMTREE_HPP
29
30 #include "PARTENSOR_basics.hpp"
31 #include <unsupported/Eigen/MatrixFunctions>
32 #include "execution.hpp"
33 #include "DataGeneration.hpp"
34 #include "DimTrees.hpp"
35 #include "Normalize.hpp"
36 #include "NesterovMNLs.hpp"
37 #include "Constants.hpp"
38 #include "Timers.hpp"
39 #include "ReadWrite.hpp"
40 #include "Matrixization.hpp"
41 #include "PartialCwiseProd.hpp"
42 #include "PartialKhatriRao.hpp"
43
44 namespace partensor
45 {
46     inline namespace v1

```

```

47 {
48     namespace internal
49     {
50
51         //template <typename ExecutionPolicy, typename Tensor>
52         //execution::internal::enable_if_execution_policy<ExecutionPolicy, Tensor>
53         //Status cpd_f(ExecutionPolicy &&, Tensor const &tnsX, std::size_t R);
54
55         /*
56         * Includes the implementation of CP Decomposition with Dimension Trees.
57         * Based on the given parameters one of the overloaded operators will
58         * be called.
59         * @tparam Tensor_ Type(data type and order) of input Tensor.
60         */
61         template <typename Tensor_>
62         struct CPD_DIMTREE_Base {
63             static constexpr std::size_t TnsSize = TensorTraits<Tensor_>::TnsSize;
64             static constexpr std::size_t LastFactor = TnsSize - 1;
65
66             using DataType = typename TensorTraits<Tensor_>::DataType;
67             using MatrixType = typename TensorTraits<Tensor_>::MatrixType;
68             using Dimensions = typename TensorTraits<Tensor_>::Dimensions;
69             using IntArray = typename TensorTraits<Tensor_>::IntArray;
70             using TensorMatrixType = Tensor<2>;
71             using Constraints = std::array<Constraint, TnsSize>;
72             using MatrixArray = std::array<MatrixType, TnsSize>;
73             using DoubleArray = std::array<double, TnsSize>;
74             using FactorArray = std::array<FactorDimTree, TnsSize>;
75             using IndexPair = typename std::array<Eigen::IndexPair<int>, 1>;
76         };
77
78         template <typename Tensor_, typename ExecutionPolicy = execution::sequenced_policy>
79         struct CPD_DIMTREE : public CPD_DIMTREE_Base<Tensor_>
80         {
81             using CPD_DIMTREE_Base<Tensor_>::TnsSize;
82             using CPD_DIMTREE_Base<Tensor_>::LastFactor;
83             using CPD_DIMTREE_Base<Tensor_>::TensorMatrixType;
84             using CPD_DIMTREE_Base<Tensor_>::Dimensions;
85             using CPD_DIMTREE_Base<Tensor_>::MatrixArray;
86             using CPD_DIMTREE_Base<Tensor_>::DataType;
87             using CPD_DIMTREE_Base<Tensor_>::IntArray;
88             using CPD_DIMTREE_Base<Tensor_>::FactorArray;
89             using CPD_DIMTREE_Base<Tensor_>::IndexPair;
90
91             using Options = partensor::Options<Tensor_, execution::sequenced_policy, DefaultValues>;
92             using Status = partensor::Status<Tensor_, execution::sequenced_policy, DefaultValues>;
93
94             // Variables that will be used in cpd with
95             // the Dimension Trees implementations.
96             struct Member_Variables {
97                 Matrix last_gramian;
98                 Matrix cwise_factor_product;
99                 Matrix mtkrp;
100                Matrix currentFactor;
101                Matrix temp_matrix;
102                Matrix tnsX_mat_lastFactor_T;
103
104                FactorArray factors;
105                FactorArray norm_factors;
106                FactorArray old_factors;
107
108                typename FactorArray::iterator it_factor;
109                typename FactorArray::iterator it_old_factor;
110
111                Tensor_ tnsX;
112                Tensor_ tnsX_approx;
113                IntArray labelSet; // starting label set for root
114                const IndexPair product_dims = { Eigen::IndexPair<int>(0, 0) }; // used for tensor
115                contractions
116
117                bool all_orthogonal = true;
118                int weight_factor;
119
120                Member_Variables() = default;
121                Member_Variables(Member_Variables const &) = default;
122                Member_Variables(Member_Variables &&) = default;
123
124                Member_Variables &operator=(Member_Variables const &) = default;
125                Member_Variables &operator=(Member_Variables &&) = default;
126            };
127
128             /*
129             * In case option variable @c writeToFile is enabled, then, before the end
130             * of the algorithm, it writes the resulted factors in files, whose
131             * paths are specified before compiling in @ options.final_factors_path.
132             *
133             * @param st [in] Struct where the returned values of @c CpdDimTree are stored.
134             */
135         };

```

```

136 void writeFactorsToFile(Status const &st)
137 {
138     std::size_t size;
139     for(std::size_t i=0; i<TnsSize; ++i)
140     {
141         size = st.factors[i].rows() * st.factors[i].cols();
142         partensor::write(st.factors[i],
143             st.options.final_factors_paths[i],
144             size);
145     }
146 }
147
148 /*
149  * Compute the cost function value at the end of each outer iteration
150  * based on the last factor.
151  *
152  * @param mv [in] Struct where ALS variables are stored.
153  * @param st [in,out] Struct where the returned values of @c CpdDimTree are stored.
154  * In this case the cost function value is updated.
155  */
156 void cost_function ( Member_Variables const &mv,
157                     Status &st)
158 {
159     st.f_value = sqrt( st.frob_tns -2 * ((mv.mttkrp * mv.currentFactor.transpose()).trace()) +
160 (mv.cwise_factor_product * mv.last_gramian).trace() );
161 }
162
163 /*
164  * Compute the cost function value at the end of each outer iteration
165  * based on the last accelerated factor.
166  *
167  * @param mv [in] Struct where ALS variables are stored.
168  * @param st [in] Struct where the returned values of @c CpdDimTree
169  * are stored.
170  * @param factors [in] Accelerated factors.
171  * @param factors_T_factors [in] Gramian matrices of factors.
172  *
173  * @returns The cost function calculated with the accelerated factors.
174  */
175 double accel_cost_function(Member_Variables const &mv,
176                             Status const &st,
177                             MatrixArray const &factors,
178                             MatrixArray const &factors_T_factors)
179 {
180     return sqrt( st.frob_tns + (PartialCwiseProd(factors_T_factors, lastFactor) *
181 factors_T_factors[lastFactor]).trace()
182 - 2 * ((PartialKhatriRao(factors, lastFactor).transpose() * mv.tnsX_mat_lastFactor_T) *
183 factors[lastFactor]).trace() );
184 }
185
186 /*
187  * Based on each factor's constraint, a different
188  * update function is used at every outer iteration.
189  *
190  * Computes also factor^AT * factor at the end.
191  *
192  * @tparam Dimensions Array type containing the Tensor dimensions.
193  *
194  * @param idx [in] Factor to be updated.
195  * @param R [in] The rank of decomposition.
196  * @param tnsDims [in] Tensor Dimensions. Each index contains the corresponding factor's
197  * rows length.
198  * @param st [in] Struct where the returned values of @c CpdDimTree are stored.
199  * @param mv [in,out] Struct where ALS variables are stored.
200  * Updates the current factor (@c Matrix type) and then updates
201  * the same factor of @c FactorDimTree type.
202  */
203 template<typename Dimensions>
204 void update_factor(int const idx,
205                   std::size_t const R,
206                   Dimensions const &tnsDims,
207                   Status const &st,
208                   Member_Variables &mv )
209 {
210     switch ( st.options.constraints[idx] )
211     {
212     case Constraint::unconstrained:
213         {
214             mv.currentFactor = mv.mttkrp * mv.cwise_factor_product.inverse();
215             break;
216         }
217     case Constraint::nonnegativity:
218         {
219             mv.temp_matrix = mv.currentFactor;
220             NesterovMNLs(mv.cwise_factor_product, mv.mttkrp, st.options.nesterov_delta_1,
221 st.options.nesterov_delta_2, mv.currentFactor);
222             if(mv.currentFactor.cwiseAbs().colwise().sum().minCoeff() == 0)

```

```

219         mv.currentFactor = 0.9 * mv.currentFactor + 0.1 * mv.temp_matrix;
220     }
221     }
222     case Constraint::orthogonality:
223     {
224         mv.temp_matrix = mv.mttkrp.transpose() * mv.mttkrp;
225         Eigen::SelfAdjointEigenSolver<Matrix> ei_gensolver(mv.temp_matrix);
226         mv.temp_matrix.noalias() = (ei_gensolver.eigenvalues().cwiseInverse().cwiseSqrt().asDiagonal())
227             * (ei_gensolver.eigenvalues().cwiseInverse().cwiseSqrt().asDiagonal())
228             * (ei_gensolver.eigenvalues().cwiseInverse().cwiseSqrt().asDiagonal())
229             * (ei_gensolver.eigenvalues().cwiseInverse().cwiseSqrt().asDiagonal());
230         mv.currentFactor.noalias() = mv.mttkrp * mv.temp_matrix;
231         break;
232     }
233     case Constraint::sparsity:
234         break;
235     default: // in case of Constraint::constant
236         break;
237 }
238 mv.it_factor->factor = matrixToTensor(mv.currentFactor, tnsDims[idx],
static_cast<int>(R)); // Map factor from Eigen Matrix to Eigen Tensor
239 mv.it_factor->gramian = (mv.it_factor->factor).contract(mv.it_factor->factor,
mv.product_dims); // Compute Covariance Tensor
240 }
241
242 /*
243 * @brief Line Search Acceleration
244 *
245 * Performs an acceleration step in the updated factors, and keeps the accelerated factors when
246 * the step succeeds. Otherwise, the acceleration step is ignored.
247 * Line Search Acceleration reduces the number outer iterations in the ALS algorithm.
248 *
249 * @note This implementation ONLY, if factors are of @c Matrix type.
250 *
251 * @tparam Dimensions Array type containing the Tensor dimensions.
252 *
253 * @param mv [in,out] Struct where ALS variables are stored.
254 * In case the acceleration is successful factor and
255 * factor^T * factor of @c FactorDimTree type are updated.
256 * @param st [in,out] Struct where the returned values of @c CpdDimTree are stored.
257 * If the acceleration succeeds updates the cost function value.
258 */
259 template<typename Dimensions>
260 void line_search_accel(Dimensions const &tnsDims,
261                     std::size_t const R,
262                     MemberVariables &mv,
263                     Status &st)
264 {
265     double f_accel = 0.0; // Objective Value after the acceleration step
266     double const accel_step = pow(st.ao_iter+1, (1.0/(st.options.accel_coeff)));
267
268     Matrix factor;
269     Matrix old_factor;
270     MatrixArray accel_factors;
271     MatrixArray accel_gramians;
272
273     for(std::size_t i=0; i<TnsSize; ++i)
274     {
275         factor = tensorToMatrix(mv.factors[i].factor, tnsDims[i], static_cast<int>(R));
276         old_factor = tensorToMatrix(mv.old_factors[i].factor,
tnsDims[i], static_cast<int>(R));
277         accel_factors[i] = old_factor + accel_step * (factor - old_factor);
278         accel_gramians[i] = accel_factors[i].transpose() * accel_factors[i];
279
280         mv.it_factor++;
281         mv.it_old_factor++;
282     }
283
284     f_accel = accel_cost_function(mv, st, accel_factors, accel_gramians);
285     if (st.f_value > f_accel)
286     {
287         for(std::size_t i=0; i<TnsSize; ++i)
288         {
289             mv.factors[i].factor = matrixToTensor(accel_factors[i], tnsDims[i],
static_cast<int>(R));
290             mv.factors[i].gramian = matrixToTensor(accel_gramians[i], static_cast<int>(R),
static_cast<int>(R));
291         }
292         st.f_value = f_accel;
293         Partensor()->Logger()->info("Acceleration Step SUCCEEDED! at iter: {}", st.ao_iter);
294     }
295     else
296         st.options.accel_fail++;
297
298     if (st.options.accel_fail==5)
299     {

```

```

300         st.options.accel_fail=0;
301         st.options.accel_coeff++;
302     }
303 }
304
305 /*
306  * Sequential implementation of Alternating Least Squares (ALS) method
307  * with Dimension Trees.
308  *
309  * @tparam Dimensions      Array type containing the Tensor dimensions.
310  *
311  * @param tnsDims [in]     Tensor Dimensions. Each index contains the corresponding factor's
rows length.
312  * @param R          [in]   The rank of decomposition.
313  * @param mv         [in]   Struct where ALS variables are stored and being updated
314  *                          until a termination condition is true.
315  * @param st         [in,out] Struct where the returned values of @c CpdDimTree are stored.
316  */
317 template<typename Dimensions>
318 void dals(Dimensions const &tnsDims,
319          std::size_t const R,
320          Member_Variab les &mv,
321          Status &status)
322 {
323     if (status.options.accelerati on)
324     {
325         mv.tnsX_mat_l astFactor_T = (Matrici zati on(mv.tnsX, l astFactor)).transpose();
326     }
327
328     if(status.options.normal izati on)
329     {
330         choose_normi l izati on_factor(status, mv.all_orthogonal, mv.wei ght_factor);
331     }
332
333     mv.tnsX_approx.resize(tnsDi ms);
334     CpdGen(mv.factors, R, mv.tnsX_approx);
335
336     status.frob_tns = square_norm(mv.tnsX);
337     status.f_value = norm(mv.tnsX - mv.tnsX_approx); // Error_tnsX
338     status.rel_costFuncti on = status.f_value/sqrt(status.frob_tns);
339     // increments from 1 to l abel Set. si ze()
340     std::iota(mv.l abel Set. begi n(), mv.l abel Set. end(), 1);
341
342     ExprTree<TnsSi ze> tree;
343     tree.Create(mv.l abel Set, tnsDi ms, R, mv.tnsX);
344
345     mv.it_factor = mv.factors.begi n();
346     for(std::si ze_t k = 0; k<TnsSi ze; k++)
347     {
348         mv.it_factor->l eaf = stati c_cast<TnsNode<1>*>(search_l eaf(k+1, tree));
349         mv.it_factor++;
350     }
351
352     // ---- Loop unti l ALS converges ----
353     whi le(1)
354     {
355         status.ao_i ter++;
356         mv.it_factor = mv.factors.begi n();
357         Partensor()->Logger()->i nfo("i ter: {} -- fval ue: {} -- rel ati ve_costFuncti on: {}",
status.ao_i ter,
358                                     status.f_value, status.rel_costFuncti on);
359
360         for(std::si ze_t i=0; i<TnsSi ze; i++)
361         {
362             mv.it_factor->l eaf->UpdateTree(TnsSi ze, i, mv.it_factor);
363
364             // Maps from Eigen Tensor to Eigen Matrix
365             mv.temp_matri x = tensorToMatri x(*rei nterpret_cast<TensorMatri xType
*>(mv.it_factor->l eaf->TensorX()), stati c_cast<i nt>(R), tnsDi ms[i]);
366             mv.mttkrp = mv.temp_matri x.transpose();
367             mv.cwise_factor_product = tensorToMatri x(mv.it_factor->l eaf->Grami an(),
stati c_cast<i nt>(R), stati c_cast<i nt>(R));
368             mv.currentFactor = tensorToMatri x(mv.it_factor->factor, tnsDi ms[i],
stati c_cast<i nt>(R));
369
370             update_factor(i, R, tnsDi ms, status, mv);
371             mv.it_factor++;
372         }
373
374         mv.it_factor = mv.factors.end()-1;
375         mv.l ast_grami an = tensorToMatri x(mv.it_factor->grami an, stati c_cast<i nt>(R),
stati c_cast<i nt>(R));
376         cost_functi on(mv, status);
377         status.rel_costFuncti on = status.f_value / sqrt(status.frob_tns);
378         if(status.options.normal izati on && !mv.all_orthogonal)
379             Normali ze(mv.wei ght_factor, stati c_cast<i nt>(R), tnsDi ms, mv.factors);
380

```



```

381         // ---- Terminating condition ----
382         if (status.ao_iter >= status.options.max_iter || status.rel_costFunction <
status.options.threshold_error)
383         {
384             for(std::size_t i=0; i<TnsSize; ++i)
385                 status.factors[i] = tensorToMatrix(mv.factors[i].factor, tnsDims[i],
static_cast<int>(R));
386
387             if(status.options.writeToFile)
388                 writeFactorsToFile(status);
389             break;
390         }
391         if (status.options.acceleration)
392         {
393             mv.norm_factors = mv.factors;
394             if (status.ao_iter > 1)
395                 line_search_accel(tnsDims, R, mv, status);
396             mv.old_factors = mv.norm_factors;
397         }
398     } // end of while
399 }
400
401 Status operator()(Tensor_ const &tnsX,
402                 std::size_t const R)
403 {
404     Status status = MakeStatus<Tensor_>();
405     Member_Variables mv;
406
407     // extract dimensions from tensor
408     Dimensions const &tnsDims = tnsX.dimensions();
409     // produce estimate factors using uniform distribution with entries in [0, 1].
410     makeFactors(tnsDims, status.options.constraints, R, mv.factors);
411     // Normalize(static_cast<int>(R), tnsDims, factors);
412
413     mv.tnsX = tnsX;
414     als(tnsDims, R, mv, status);
415
416     return status;
417 }
418
419 Status operator()(Tensor_ const &tnsX,
420                 std::size_t const R,
421                 Options const &options)
422 {
423     Status status(options);
424     Member_Variables mv;
425
426     // extract dimensions from tensor
427     Dimensions const &tnsDims = tnsX.dimensions();
428     // produce estimate factors using uniform distribution with entries in [0, 1].
429     makeFactors(tnsDims, status.options.constraints, R, mv.factors);
430     // Normalize(static_cast<int>(R), tnsDims, factors);
431     mv.tnsX = tnsX;
432
433     switch (status.options.method)
434     {
435     case Method::als:
436     {
437         als(tnsDims, R, mv, status);
438         break;
439     }
440     case Method::rnd:
441         break;
442     case Method::bc:
443         break;
444     default:
445         break;
446     }
447     return status;
448 }
449
450 template<typename MatrixArray_>
451 Status operator()(Tensor_ const &tnsX,
452                 std::size_t const R,
453                 MatrixArray_ const &factorsInit)
454 {
455     Status status = MakeStatus<Tensor_>();
456     Member_Variables mv;
457
458     // extract dimensions from tensor
459     Dimensions const &tnsDims = tnsX.dimensions();
460     // Copy factorsInit data to factors - FactorDimTree data struct
461     fillDimTreeFactors(factorsInit, status.options.constraints, mv.factors);
462     // Normalize(static_cast<int>(R), tnsDims, factors);
463 }

```

```

508     mv.tnsX = tnsX;
509     als(tnsDims, R, mv, status);
510     return status;
511 }
512
513 template<typename MatrixArray_>
514 Status operator()(Tensor_ const &tnsX,
515                 std::size_t const R,
516                 Options const &options,
517                 MatrixArray_ const &factorsInit)
518 {
519     Status status(options);
520     Member_Variab les mv;
521
522     // extract dimensions from tensor
523     Dimensions const &tnsDims = tnsX.dimensions();
524     // Copy factorsInit data to factors - FactorDimTree data struct
525     fillDimTreeFactors(factorsInit, status.options.constraints, mv.factors);
526     // Normalize(static_cast<int>(R), tnsDims, factors);
527     mv.tnsX = tnsX;
528
529     switch ( status.options.method )
530     {
531     case Method::als:
532     {
533         als(tnsDims, R, mv, status);
534         break;
535     }
536     case Method::rnd:
537         break;
538     case Method::bc:
539         break;
540     default:
541         break;
542     }
543     return status;
544 }
545
546 Status operator()(std::array<int, TnsSize> const &tnsDims,
547                 std::size_t const R,
548                 std::string const &path)
549 {
550     using Tensor = Tensor<static_cast<int>(TnsSize)>;
551
552     Status status = MakeStatus<Tensor>();
553     Member_Variab les mv;
554
555     long long int fileSize = 1;
556     for(auto &dim : tnsDims)
557         fileSize *= static_cast<long long int>(dim);
558
559     mv.tnsX.resize(tnsDims);
560     // Read the whole Tensor from a file
561     read( path,
562          fileSize,
563          0,
564          mv.tnsX );
565     // produce estimate factors using uniform distribution with entries in [0, 1].
566     makeFactors(tnsDims, status.options.constraints, R, mv.factors);
567     // Normalize(static_cast<int>(R), tnsDims, factors);
568
569     als(tnsDims, R, mv, status);
570     return status;
571 }
572
573 Status operator()(std::array<int, TnsSize> const &tnsDims,
574                 std::size_t const R,
575                 std::string const &path,
576                 Options const &options)
577 {
578     Status status(options);
579     Member_Variab les mv;
580
581     long long int fileSize = 1;
582     for(auto &dim : tnsDims)
583         fileSize *= static_cast<long long int>(dim);
584
585     mv.tnsX.resize(tnsDims);
586     // Read the whole Tensor from a file
587     read( path,
588          fileSize,
589          0,
590          mv.tnsX );
591     // produce estimate factors using uniform distribution with entries in [0, 1].
592     makeFactors(tnsDims, status.options.constraints, R, mv.factors);
593     // Normalize(static_cast<int>(R), tnsDims, mv.factors);
594 }

```

```

643     switch ( status.options.method )
644     {
645     case Method::als:
646     {
647         als(tnsDims, R, mv, status);
648         break;
649     }
650     case Method::rnd:
651         break;
652     case Method::bc:
653         break;
654     default:
655         break;
656     }
657     return status;
658 }
659
674 Status operator()(std::array<int, TnsSize>          const &tnsDims,
675                  std::size_t                        const R,
676                  std::array<std::string, TnsSize+1> const &paths)
677 {
678     using Tensor = Tensor<static_cast<int>(TnsSize)>;
679
680     Status status = MakeStatus<Tensor>();
681     Member_Variables mv;
682
683     long long int fileSize = 1;
684     for(auto &dim : tnsDims)
685         fileSize *= static_cast<long long int>(dim);
686
687     mv.tnsX.resize(tnsDims);
688     // Read the whole Tensor from a file
689     read( paths.front(),
690          fileSize,
691          0,
692          mv.tnsX );
693     // Read initialized factors from files
694     for(std::size_t i=0; i<TnsSize; ++i)
695     {
696         status.factors[i] = Matrix(tnsDims[i], static_cast<int>(R));
697         read( paths[i+1],
698              tnsDims[i]*R,
699              0,
700              status.factors[i] );
701     }
702     // Copy factorsInit data to factors - FactorDimTree data struct
703     fillDimTreeFactors(status.factors, status.options.constraints, mv.factors);
704     // Normalize(static_cast<int>(R), tnsDims, mv.factors);
705
706     als(tnsDims, R, mv, status);
707     return status;
708 }
709
728 Status operator()(std::array<int, TnsSize>          const &tnsDims,
729                  std::size_t                        const R,
730                  std::array<std::string, TnsSize+1> const &paths,
731                  const options                      const &options)
732 {
733     Status status(options);
734     Member_Variables mv;
735
736     long long int fileSize = 1;
737     for(auto &dim : tnsDims)
738         fileSize *= static_cast<long long int>(dim);
739
740     mv.tnsX.resize(tnsDims);
741     // Read the whole Tensor from a file
742     read( paths.front(),
743          fileSize,
744          0,
745          mv.tnsX );
746     // Read initialized factors from files
747     for(std::size_t i=0; i<TnsSize; ++i)
748     {
749         status.factors[i] = Matrix(tnsDims[i], static_cast<int>(R));
750         read( paths[i+1],
751              tnsDims[i]*R,
752              0,
753              status.factors[i] );
754     }
755     // Copy factorsInit data to factors - FactorDimTree data struct
756     fillDimTreeFactors(status.factors, status.options.constraints, mv.factors);
757     // Normalize(static_cast<int>(R), tnsDims, factors);
758     switch ( status.options.method )
759     {
760     case Method::als:
761     {

```

```

762         als(tnsDims, R, mv, status);
763         break;
764     }
765     case Method: : rnd:
766         break;
767     case Method: : bc:
768         break;
769     default:
770         break;
771     }
772     return status;
773 }
774 };
775 };
776 } // namespace internal
777 } // namespace v1
778 } // end namespace partensor
779 } // end namespace partensor
780
781 #if USE_MPI
782
783 #include "CpdDimTreeMpi.hpp"
784 #endif /* USE_MPI */
785
786 namespace partensor
787 {
788
789     template <typename ExecutionPolicy, typename Tensor_>
790     execution: : internal: : enable_if_execution_policy<ExecutionPolicy, Status<Tensor_, execution: : execution_policy_t<ExecutionPolicy>>>
791     cpdDimTree( ExecutionPolicy      &&,
792                Tensor_              const &tnsX,
793                std: : size_t         const R )
794     {
795         using ExPolicy = execution: : execution_policy_t<ExecutionPolicy>;
796
797         if constexpr (std: : is_same_v<ExPolicy, execution: : sequenced_policy>)
798         {
799             return internal: : CPD_DIMTREE<Tensor_>()(tnsX, R);
800         }
801         else if constexpr (std: : is_same_v<ExPolicy, execution: : openmpi_policy>)
802         {
803             return internal: : CPD_DIMTREE<Tensor_, execution: : openmpi_policy>()(tnsX, R);
804         }
805         else
806             return internal: : CPD_DIMTREE<Tensor_>()(tnsX, R);
807     }
808
809     /*
810     * Interface of Canonical Polyadic Decomposition(cpd) with the use of Dimensional
811     * Trees. Sequential Policy.
812     *
813     * @tparam Tensor_      Type(data type and order) of input Tensor.
814     *                      @c Tensor_ must be @c partensor: : Tensor<order>, where
815     *                      @c order must be in range of @c [3-8].
816     * @param tnsX          [in] The given Tensor to be factorized of @c Tensor_ type,
817     *                      with @c double data.
818     * @param R             [in] The rank of decomposition.
819     *
820     * @returns An object of type @c Status with the results of the algorithm.
821     */
822     template<typename Tensor_>
823     auto cpdDimTree(Tensor_ const &tnsX,
824                    std: : size_t const R)
825     {
826         return internal: : CPD_DIMTREE<Tensor_>()(tnsX, R);
827     }
828
829     template <typename Tensor_, typename ExecutionPolicy>
830     execution: : internal: : enable_if_execution_policy<ExecutionPolicy, Status<Tensor_, execution: : execution_policy_t<ExecutionPolicy>>>
831     cpdDimTree( ExecutionPolicy      &&,
832                Tensor_              const &tnsX,
833                std: : size_t         const R,
834                Options<Tensor_, execution: : execution_policy_t<ExecutionPolicy>, DefaultValues> const
835                &options )
836     {
837         using ExPolicy = execution: : execution_policy_t<ExecutionPolicy>;
838
839         if constexpr (std: : is_same_v<ExPolicy, execution: : sequenced_policy>)
840         {
841             return internal: : CPD_DIMTREE<Tensor_>()(tnsX, R, options);
842         }
843         else if constexpr (std: : is_same_v<ExPolicy, execution: : openmpi_policy>)
844         {
845             return internal: : CPD_DIMTREE<Tensor_, execution: : openmpi_policy>()(tnsX, R, options);
846         }
847     }
848 }

```

```

883     else
884         return internal::CPD_DIMTREE<Tensor_>()(tnsX, R, options);
885 }
886
887 /*
888 * Interface of Canonical Polyadic Decomposition(cpd) with the use of Dimensions
889 * Trees. Sequential Policy.
890 *
891 * @tparam Tensor_      Type(data type and order) of input Tensor.
892 * @param tnsX          [in] The given Tensor to be factorized of @c Tensor_ type,
893 *                        with @c double data.
894 * @param R              [in] The rank of decomposition.
895 * @param options       [in] User's @c options, other than the default. It must be of
896 *                        @c partensor::Options<partensor::Tensor<order> type,
897 *                        where @c order must be in range of @c [3-8].
898 *
899 * @returns An object of type @c Status with the results of the algorithm.
900 */
901 template<typename Tensor_>
902 auto cpdDimTree(Tensor_ const &tnsX,
903                std::size_t const R,
904                Options<Tensor_> const &options)
905 {
906     return internal::CPD_DIMTREE<Tensor_, execution::sequenced_policy>()(tnsX, R, options);
907 }
908
909 template <typename Tensor_, typename MatrixArray_, typename ExecutionPolicy>
910 cpdDimTree(ExecutionPolicy &&,
911            Tensor_ const &tnsX,
912            std::size_t const R,
913            MatrixArray_ const &factorsInit)
914 {
915     using ExPolicy = execution::execution_policy_t<ExecutionPolicy>;
916
917     if constexpr (std::is_same_v<ExPolicy, execution::sequenced_policy>)
918     {
919         return internal::CPD_DIMTREE<Tensor_>()(tnsX, R, factorsInit);
920     }
921     else if constexpr (std::is_same_v<ExPolicy, execution::openmpi_policy>)
922     {
923         return internal::CPD_DIMTREE<Tensor_, execution::openmpi_policy>()(tnsX, R, factorsInit);
924     }
925     else
926         return internal::CPD_DIMTREE<Tensor_>()(tnsX, R, factorsInit);
927 }
928
929 /*
930 * Interface of Canonical Polyadic Decomposition(cpd) with the use of Dimensions
931 * Trees. Sequential Policy.
932 *
933 * @tparam Tensor_      Type(data type and order) of input Tensor.
934 * @tparam MatrixArray_ An @c stl array, where the initialized factors will
935 *                        be stored. Its size must be equal to the Tensor's @c tnsX
936 *                        @c order. The type can be either @c partensor::Matrix,
937 *                        or @c partensor::Tensor<2>.
938 *
939 * @param tnsX          [in] The given Tensor to be factorized of @c Tensor_ type,
940 *                        with @c double data.
941 * @param R              [in] The rank of decomposition.
942 * @param factorsInit   [in] Uses initialized factors instead of randomly generated. The
943 *                        data can be either @c partensor::Matrix, or
944 *                        @c partensor::Tensor<2> type and stored in an @c stl array
945 *                        with size same as the @c order of @c tnsX.
946 *
947 * @returns An object of type @c Status with the results of the algorithm.
948 */
949 template<typename Tensor_, typename MatrixArray_>
950 auto cpdDimTree(Tensor_ const &tnsX,
951                std::size_t const R,
952                MatrixArray_ const &factorsInit)
953 {
954     return internal::CPD_DIMTREE<Tensor_, execution::sequenced_policy>()(tnsX, R, factorsInit);
955 }
956
957 template <typename Tensor_, typename MatrixArray_, typename ExecutionPolicy>
958 cpdDimTree(ExecutionPolicy &&,
959            Tensor_ const &tnsX,
960            std::size_t const R,
961            Options<Tensor_, execution::execution_policy_t<ExecutionPolicy>, DefaultValues> const
962            &options,
963            MatrixArray_ const &factorsInit)
964 {
965     using ExPolicy = execution::execution_policy_t<ExecutionPolicy>;

```

```

1020
1021     if constexpr (std::is_same_v<ExPolicy, execution::sequenced_policy>)
1022     {
1023         return internal::CPD_DIMTREE<Tensor_>()(tnsX, R, options, factorsInit);
1024     }
1025     else if constexpr (std::is_same_v<ExPolicy, execution::openmpi_policy>)
1026     {
1027         return internal::CPD_DIMTREE<Tensor_, execution::openmpi_policy>()(tnsX, R, options, factorsInit);
1028     }
1029     else
1030         return internal::CPD_DIMTREE<Tensor_>()(tnsX, R, options, factorsInit);
1031 }
1032
1033 /*
1034  * Interface of Canonical Polyadic Decomposition(cpd) with the use of Dimension
1035  * Trees. Sequential Policy.
1036  *
1037  * @tparam Tensor_      Type(data type and order) of input Tensor.
1038  * @tparam MatrixArray_ An @c stl array, where the initialized factors will
1039  *                       be stored. Its size must be equal to the Tensor's @c tnsX
1040  *                       @c order. The type can be either @c partensor::Matrix,
1041  *                       or @c partensor::Tensor<2>.
1042  *
1043  * @param tnsX          [in] The given Tensor to be factorized of @c Tensor_ type,
1044  *                       with @c double data.
1045  * @param R             [in] The rank of decomposition.
1046  * @param options       [in] User's @c options, other than the default. It must be of
1047  *                       @c partensor::Options<partensor::Tensor<order> type,
1048  *                       where @c order must be in range of @c [3-8].
1049  * @param factorsInit [in] Uses initialized factors instead of randomly generated. The
1050  *                       data must be of @c partensor::Matrix type and stored in an
1051  *                       @c stl array with size same as the @c order of @c tnsX.
1052  *
1053  * @returns An object of type @c Status with the results of the algorithm.
1054  */
1055 template<typename Tensor_, typename MatrixArray_>
1056 auto cpdDimTree(Tensor_ const &tnsX,
1057                std::size_t const R,
1058                Options<Tensor_> const &options,
1059                MatrixArray_ const &factorsInit)
1060 {
1061     return internal::CPD_DIMTREE<Tensor_, execution::sequenced_policy>()(tnsX, R, options, factorsInit);
1062 }
1063
1064 template <typename ExecutionPolicy, std::size_t TnsSize>
1065 execution::internal::enable_if_execution_policy<ExecutionPolicy, Status<Tensor<static_cast<int>(TnsSize)>, execution::execution::cpdDimTree( ExecutionPolicy &&,
1066                          std::array<int, TnsSize> const &tnsDims,
1067                          std::size_t const R,
1068                          std::string const &path )
1069 {
1070     using ExPolicy = execution::execution_policy_t<ExecutionPolicy>;
1071     using Tensor_ = Tensor<static_cast<int>(TnsSize)>;
1072
1073     if constexpr (std::is_same_v<ExPolicy, execution::sequenced_policy>)
1074     {
1075         return internal::CPD_DIMTREE<Tensor_>()(tnsDims, R, path);
1076     }
1077     else if constexpr (std::is_same_v<ExPolicy, execution::openmpi_policy>)
1078     {
1079         return internal::CPD_DIMTREE<Tensor_, execution::openmpi_policy>()(tnsDims, R, path);
1080     }
1081     else
1082         return internal::CPD_DIMTREE<Tensor_>()(tnsDims, R, path);
1083 }
1084
1085 /*
1086  * Interface of Canonical Polyadic Decomposition(cpd) with the use of Dimension
1087  * Trees. Sequential Policy.
1088  *
1089  * With this version of @c cpdDimTree, the Tensor can be read from a file, specified
1090  * in @c path variable.
1091  *
1092  * @tparam TnsSize      Order of the input Tensor.
1093  *
1094  * @param tnsDims       [in] @c Stl array containing the Tensor dimensions, whose
1095  *                       length must be same as the Tensor order.
1096  * @param R             [in] The rank of decomposition.
1097  * @param path          [in] The path where the tensor is located.
1098  *
1099  * @returns An object of type @c Status with the results of the algorithm.
1100  */
1101 template<std::size_t TnsSize>
1102 auto cpdDimTree(std::array<int, TnsSize> const &tnsDims,
1103                std::size_t const R,
1104                std::string const &path)

```

```

1127 {
1128     using Tensor_ = Tensor<stati_c_cast<int>(TnsSi ze)>;
1129     return internal::CPD_DIMTREE<Tensor_, executon::sequenced_policy>() (tnsDms, R, path);
1130 }
1131
1132 template <typename Executi onPol icy, std::size_t TnsSi ze>
1133
1134 executon::internal::enable_if_executi on_pol icy<Executi onPol icy, Status<Tensor<stati_c_cast<int>(TnsSi ze)>, executon::
1135 cpdDimTree( Executi onPol icy &&,
1136             std::array<int, TnsSi ze> const &tnsDms,
1137             std::size_t const R,
1138             std::string const &path,
1139             Options<Tensor<stati_c_cast<int>(TnsSi ze)>, executon::executi on_pol icy_t<Executi onPol icy>, DefaultVal ues>
1140             const &opti ons )
1141 {
1142     using ExPol icy = executon::executi on_pol icy_t<Executi onPol icy>;
1143     using Tensor_ = Tensor<stati_c_cast<int>(TnsSi ze)>;
1144
1145     if constexpr (std::is_same_v<ExPol icy, executon::sequenced_policy>)
1146     {
1147         return internal::CPD_DIMTREE<Tensor_>() (tnsDms, R, path, opti ons);
1148     }
1149     else if constexpr (std::is_same_v<ExPol icy, executon::openmpi_policy>)
1150     {
1151         return internal::CPD_DIMTREE<Tensor_, executon::openmpi_policy>() (tnsDms, R, path, opti ons);
1152     }
1153     else
1154         return internal::CPD_DIMTREE<Tensor_>() (tnsDms, R, path, opti ons);
1155 }
1156
1157 /*
1158 * Interface of Canonical Polyadic Decomposition(cpd) with the use of Dimension
1159 * Trees. Sequential Policy.
1160 *
1161 * With this version of @c cpdDimTree, the Tensor can be read from a file, specified
1162 * in @c path variable.
1163 *
1164 * @tparam TnsSi ze          Order of the input Tensor.
1165 *
1166 * @param tnsDms            [in] @c STL array containing the Tensor dimensions, whose
1167 *                          length must be same as the Tensor order.
1168 * @param R                 [in] The rank of decomposition.
1169 * @param path              [in] The path where the tensor is located.
1170 * @param opti ons          [in] User's @c options, other than the default. It must be of
1171 *                          @c partensor::Options<partensor::Tensor<order> type,
1172 *                          where @c order must be in range of @c [3-8].
1173 *
1174 * @returns An object of type @c Status with the results of the algorithm.
1175 */
1176 template<std::size_t TnsSi ze>
1177 auto cpdDimTree(std::array<int, TnsSi ze> const &tnsDms,
1178                std::size_t const R,
1179                std::string const &path,
1180                Options<Tensor<stati_c_cast<int>(TnsSi ze)> const &opti ons )
1181 {
1182     using Tensor_ = Tensor<stati_c_cast<int>(TnsSi ze)>;
1183     return internal::CPD_DIMTREE<Tensor_, executon::sequenced_policy>() (tnsDms, R, path, opti ons);
1184 }
1185
1186 template <typename Executi onPol icy, std::size_t TnsSi ze>
1187
1188 executon::internal::enable_if_executi on_pol icy<Executi onPol icy, Status<Tensor<stati_c_cast<int>(TnsSi ze)>, executon::
1189 cpdDimTree( Executi onPol icy &&,
1190             std::array<int, TnsSi ze> const &tnsDms,
1191             std::size_t const R,
1192             std::array<std::string, TnsSi ze+1> const &paths )
1193 {
1194     using ExPol icy = executon::executi on_pol icy_t<Executi onPol icy>;
1195     using Tensor_ = Tensor<stati_c_cast<int>(TnsSi ze)>;
1196
1197     if constexpr (std::is_same_v<ExPol icy, executon::sequenced_policy>)
1198     {
1199         return internal::CPD_DIMTREE<Tensor_>() (tnsDms, R, paths);
1200     }
1201     else if constexpr (std::is_same_v<ExPol icy, executon::openmpi_policy>)
1202     {
1203         return internal::CPD_DIMTREE<Tensor_, executon::openmpi_policy>() (tnsDms, R, paths);
1204     }
1205     else
1206         return internal::CPD_DIMTREE<Tensor_>() (tnsDms, R, paths);
1207 }
1208
1209 /*
1210 * Interface of Canonical Polyadic Decomposition(cpd) with the use of Dimension
1211 * Trees. Sequential Policy.
1212 */

```

```

1257 * With this version of @c cpdDimTree, the Tensor can be read from a file, specified
1258 * in @c path variable.
1259 *
1260 * @tparam TnsSize      Order of the input Tensor.
1261 *
1262 * @param tnsDims      [in] @c Stl array containing the Tensor dimensions, whose
1263 *                      length must be same as the Tensor order.
1264 * @param R            [in] The rank of decomposition.
1265 * @param paths        [in] An @c stl array containing paths for the Tensor to be
1266 *                      factorized and after that the paths for the initialized
1267 *                      factors.
1268 *
1269 * @returns An object of type @c Status with the results of the algorithm.
1270 */
1271 template<std::size_t TnsSize>
1272 auto cpdDimTree(std::array<int, TnsSize>          const &tnsDims,
1273                std::size_t                      const R,
1274                std::array<std::string, TnsSize+1> const &paths )
1275 {
1276     using Tensor_ = Tensor<static_cast<int>(TnsSize)>;
1277     return internal::CPD_DIMTREE<Tensor_, execution::sequenced_policy>()(tnsDims, R, paths);
1278 }
1279
1306 template <typename ExecutionPolicy, std::size_t TnsSize>
1307
1308 execution::internal::enable_if_execution_policy<ExecutionPolicy, Status<Tensor<static_cast<int>(TnsSize)>, execution::internal::
cpdDimTree( ExecutionPolicy          &&,
1309            std::array<int, TnsSize>  const &tnsDims,
1310            std::size_t              const R,
1311            std::array<std::string, TnsSize+1> const &paths,
1312
1313 Options<Tensor<static_cast<int>(TnsSize)>, execution::execution_policy_t<ExecutionPolicy>, DefaultValues>
const &options )
1314 {
1315     using ExPolicy = execution::execution_policy_t<ExecutionPolicy>;
1316     using Tensor_ = Tensor<static_cast<int>(TnsSize)>;
1317
1318     if constexpr (std::is_same_v<ExPolicy, execution::sequenced_policy>)
1319     {
1320         return internal::CPD_DIMTREE<Tensor_>()(tnsDims, R, paths, options);
1321     }
1322     else if constexpr (std::is_same_v<ExPolicy, execution::openmpi_policy>)
1323     {
1324         return internal::CPD_DIMTREE<Tensor_, execution::openmpi_policy>()(tnsDims, R, paths, options);
1325     }
1326     else
1327     {
1328         return internal::CPD_DIMTREE<Tensor_>()(tnsDims, R, paths, options);
1329     }
1330 }
1331
1332 /*
1333 * Interface of Canonical Polyadic Decomposition(cpd) with the use of Dimension
1334 * Trees. Sequential Policy.
1335 *
1336 * With this version of @c cpdDimTree, the Tensor can be read from a file, specified
1337 * in @c path variable.
1338 *
1339 * @tparam TnsSize      Order of the input Tensor.
1340 *
1341 * @param tnsDims      [in] @c Stl array containing the Tensor dimensions, whose
1342 *                      length must be same as the Tensor order.
1343 * @param R            [in] The rank of decomposition.
1344 * @param paths        [in] An @c stl array containing paths for the Tensor to be
1345 *                      factorized and after that the paths for the initialized
1346 *                      factors.
1347 * @param options      [in] User's @c options, other than the default. It must be of
1348 *                      @c partensor::Options<partensor::Tensor<order> type,
1349 *                      where @c order must be in range of @c [3-8].
1350 *
1351 * @returns An object of type @c Status with the results of the algorithm.
1352 */
1353 template<std::size_t TnsSize>
1354 auto cpdDimTree(std::array<int, TnsSize>          const &tnsDims,
1355                std::size_t                      const R,
1356                std::array<std::string, TnsSize+1> const &paths,
1357                Options<Tensor<static_cast<int>(TnsSize)> const &options )
1358 {
1359     using Tensor_ = Tensor<static_cast<int>(TnsSize)>;
1360     return internal::CPD_DIMTREE<Tensor_, execution::sequenced_policy>()(tnsDims, R, paths, options);
1361 }
1362 } // end namespace partensor
1363
1364 #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.](#)

```
1 #ifndef DOXYGEN_SHOULD_SKIP_THIS
15 #endif // DOXYGEN_SHOULD_SKIP_THIS
16 /*****
26 #ifndef(PARTENSOR_CPD_DIMTREE_DIM_TREE_HPP)
27 #error "CpdDimTreeMpi can only be included inside CpdDimTree"
28 #endif /* PARTENSOR_CPD_DIMTREE_DIM_TREE_HPP */
29
30 #include <math.h>
31 #include "PartialCwiseProd.hpp"
32 #include "TensorOperations.hpp"
33 #include "unsupported/Eigen/MatrixFunctions"
34
35 namespace partensor
36 {
37
38     inline namespace v1 {
39         namespace internal {
40
41             template<typename Tensor_>
42             struct CPD_DIMTREE<Tensor_, execution::openmpi_policy> : public CPD_DIMTREE_Base<Tensor_>
43             {
44                 using CPD_DIMTREE_Base<Tensor_>::TnsSize;
45                 using CPD_DIMTREE_Base<Tensor_>::LastFactor;
46                 using CPD_DIMTREE_Base<Tensor_>::TensorMatrixType;
47                 using CPD_DIMTREE_Base<Tensor_>::Dimensions;
48                 using CPD_DIMTREE_Base<Tensor_>::MatrixArray;
49                 using CPD_DIMTREE_Base<Tensor_>::DataType;
50                 using CPD_DIMTREE_Base<Tensor_>::IntArray;
51                 using CPD_DIMTREE_Base<Tensor_>::FactorArray;
52                 using CPD_DIMTREE_Base<Tensor_>::IndexPair;
53
54                 // For MPI usage
55                 using CartCommunicator = partensor::cartesian_communicator; // From ParallelWrapper.hpp
56                 using CartCommVector = std::vector<CartCommunicator>;
57                 using IntVector = std::vector<int>;
58                 using Int2DVector = std::vector<std::vector<int>>;
59
60                 using Options = partensor::Options<Tensor_, execution::openmpi_policy, DefaultValues>;
61                 using Status = partensor::Status<Tensor_, execution::openmpi_policy, DefaultValues>;
62
63                 // Variables that will be used in cpd with
64                 // Dimension Trees implementations.
65                 struct Member_Variables
66                 {
67                     MPI_Communicator &world = Partensor()->MPI_Communicator(); // MPI_COMM_WORLD
68
69                     double local_f_value;
```

```

75         int          RxR;
76         int          world_size;
77         const IndexPair product_dims = { Eigen::IndexPair<int>(0, 0) }; // used for tensor
    contractions
78
79         Int2DVector    displs_subTns;        // skipping dimension "rows" for each subtensor
80         Int2DVector    displs_subTns_R;      // skipping dimension "rows" for each subtensor
    times R ( for MPI communication purposes )
81         Int2DVector    subTnsDims;          // dimensions of subtensor
82         Int2DVector    subTnsDims_R;        // dimensions of subtensor times R ( for MPI
    communication purposes )
83         Int2DVector    displs_local_update;  // displacement in the local factor for update
    rows
84         Int2DVector    send_recv_counts;     // rows to be communicated after update times R
85
86         CartCommVector layer_comm;
87         CartCommVector fiber_comm;
88
89         IntArray       layer_rank;
90         IntArray       fiber_rank;
91         IntArray       rows_for_update;
92         IntArray       subTns_offsets;
93         IntArray       subTns_extents;
94         IntArray       labelSet;            // starting label set for root
95
96         MatrixArray    local_factors;
97         MatrixArray    local_factors_T;
98         MatrixArray    layer_factors;
99         MatrixArray    layer_factors_T;
100        MatrixArray    local_mttkrp;
101        MatrixArray    layer_mttkrp;
102        MatrixArray    local_mttkrp_T;
103        MatrixArray    layer_mttkrp_T;
104
105        Matrix          cwise_factor_product;
106        Matrix          factor_T_factor;
107        Matrix          last_cov;
108        Matrix          temp_matrix;
109        Matrix          tnsX_mat_lastFactor_T;
110        Matrix          nesterov_ol_d_layer_factor;
111
112        Tensor_         subTns;
113        Tensor_         subTnsX_approx;     // tensor in order to compute starting f_value from
    random generated factors
114
115        FactorArray     factors;
116        FactorArray     layer_factors_dmtree;
117        FactorArray     norm_factors;
118        FactorArray     old_factors;
119
120        std::array<int, 2> subfactor_offsets;
121        std::array<int, 2> subfactor_extents;
122
123        typename FactorArray::iterator status_factor_it;
124        typename FactorArray::iterator layer_factor_it;
125
126        bool            all_orthogonal = true;
127        int              weight_factor;
128
129        /*
130        * Calculates if the number of processors given from terminal
131        * are equal to the processors in the implementation.
132        *
133        * @param procs [in] @c stl array with the number of processors per
134        *                  dimension of the tensor.
135        */
136        void check_processor_availability(std::array<int, TnsSi ze> const &procs)
137        {
138            // MPI_Environment &env = Partensor()->MpiEnvironment();
139            world_size = world_size();
140            // numprocs must be product of options.proc_per_mode
141            if (std::accumulate(procs.begin(), procs.end(), 1,
142                std::multiplies<int>()) != world_size && world.rank() == 0)
143            {
144                Partensor()->Logger()->error("The product of the processors per mode must be
    equal to {}m", world_size);
145                // env.abort(-1);
146            }
147        }
148
149        Member_Variab les() = default;
150        Member_Variab les(int R, std::array<int, TnsSi ze> &procs) : local_f_value(0.0),
    RxR(R*R),
151        displs_subTns(TnsSi ze),
152        displs_subTns_R(TnsSi ze),
153        subTnsDims(TnsSi ze),

```

```

154                                     subTnsDims_R(TnsSize),
155
156     displocal_update(TnsSize),
157
158     send_rcv_counts(TnsSize)
159     {
160         check_processor_availability(procs);
161         layer_comm.reserve(TnsSize);
162         fiber_comm.reserve(TnsSize);
163     }
164
165     Member_Variables(Member_Variables const &) = default;
166     Member_Variables(Member_Variables &&) = default;
167
168     Member_Variables &operator=(Member_Variables const &) = default;
169     Member_Variables &operator=(Member_Variables &&) = default;
170 };
171
172 /*
173  * In case option variable @c writeToFile is enabled then, before the end
174  * of the algorithm writes the resulted factors in files, where their
175  * paths are specified before compiling in @ options.final_factors_path.
176  *
177  * @param st [in] Struct where the returned values of @c CpdDimTree are stored.
178  */
179 void writeFactorsToFile(Status const &st)
180 {
181     std::size_t size;
182     for(std::size_t i=0; i<TnsSize; ++i)
183     {
184         size = st.factors[i].rows() * st.factors[i].cols();
185         partensor::write(st.factors[i],
186                        st.options.final_factors_paths[i],
187                        size);
188     }
189 }
190
191 /*
192  * Compute the cost function value based on the initial factors.
193  *
194  * @param grid_comm [in] The communication grid, where the processors
195  * communicate their cost function.
196  * @param R [in] The rank of decomposition.
197  * @param mv [in] Struct where ALS variables are stored.
198  * @param st [in,out] Struct where the returned values of @c CpdDimTree are
199  * stored.
200  *
201  * In this case the cost function value and the Frobenius
202  * squared norm of the tensor are updated.
203  */
204 void cost_function_init(CartCommunicator const &grid_comm,
205                       std::size_t const R,
206                       Member_Variables &mv,
207                       Status &st)
208 {
209     mv.subTnsX_approx.resize(mv.subTns_extents);
210     CpdGen(mv.layer_factors_dimTree, R, mv.subTnsX_approx);
211
212     // communicate the squared norm of sub tensor, in order to compute frob_tns
213     all_reduce(grid_comm,
214               square_norm(mv.subTns),
215               st.frob_tns,
216               std::plus<double>());
217     // communication among all processors for f_value
218     all_reduce(grid_comm,
219               square_norm(mv.subTns - mv.subTnsX_approx),
220               st.f_value,
221               std::plus<double>());
222     st.f_value = sqrt(st.f_value);
223 }
224
225 /*
226  * Compute the cost function value at the end of each outer iteration
227  * based on the last factor.
228  *
229  * @param grid_comm [in] The communication grid, where the processors
230  * communicate their cost function.
231  * @param mv [in] Struct where ALS variables are stored.
232  * @param st [in,out] Struct where the returned values of @c CpdDimTree are
233  * stored.
234  *
235  * In this case the cost function value is updated.
236  */
237 void cost_function(CartCommunicator const &grid_comm,
238                  Member_Variables &mv,
239                  Status &st)
240 {
241     mv.local_f_value = ((mv.layer_mttkrp_T[lastFactor] *
242                        mv.layer_factors[lastFactor]).trace());

```

```

236         all_reduce( grid_comm,
237                     inplace(&mv.local_f_value),
238                     1,
239                     std::plus<double>() );
240         st.f_value = sqrt(st.frob_tns - 2 * mv.local_f_value +
(mv.cwise_factor_product.cwiseProduct(mv.factor_T_factor).sum()));
241     }
242
243     /*
244     * Compute the cost function value at the end of each outer iteration
245     * based on the last accelerated factor.
246     *
247     * @param grid_comm      [in] The communication grid, where the processors
248     *                        communicate their cost function.
249     * @param mv             [in] Struct where ALS variables are stored.
250     * @param st             [in] Struct where the returned values of @c CpdDimTree
251     *                        are stored.
252     * @param factors        [in] Accelerated factors.
253     * @param factors_T_factors [in] Gramian matrices of factors.
254     *
255     * @returns The cost function calculated with the accelerated factors.
256     */
257     double accel_cost_function(CartCommunicator const &grid_comm,
258                               MemberVariables const &mv,
259                               Status const &st,
260                               MatrixArray const &factors,
261                               MatrixArray const &factors_T_factors)
262     {
263         double local_f_value =
264             ((PartialKhatrao(factors, lastFactor).transpose() * mv.tnsX_mat_lastFactor_T
* factors[lastFactor]).trace());
265         all_reduce( grid_comm,
266                     inplace(&local_f_value),
267                     1,
268                     std::plus<double>() );
269         Matrix cwiseFactor_prod = PartialCwiseProd(factors_T_factors, lastFactor) *
factors_T_factors[lastFactor];
270         return sqrt(st.frob_tns - 2 * local_f_value + cwiseFactor_prod.trace());
271     }
272
273     /*
274     * Make use of the dimensions and the number of processors per dimension
275     * and then calculates the dimensions of the subtensor and subfactor for
276     * each processor.
277     *
278     * Also initialize the FactorDimTree struct for each processor with the
279     * data from factors.
280     *
281     * @tparam Dimensions      Array type containing the Tensor dimensions.
282     *
283     * @param tnsDims          [in] Tensor Dimensions. Each index contains the corresponding
284     *                        factor's rows length.
285     * @param st               [in] Struct where the returned values of @c CpdDimTree are
286     *                        stored.
287     * @param R                [in] The rank of decomposition.
288     * @param mv               [in,out] Struct where ALS variables are stored.
289     *                        Updates @c stl arrays with dimensions for subtensors and
290     *                        subfactors.
291     */
292     template<typename Dimensions>
293     void compute_sub_dimensions(Dimensions const &tnsDims,
294                               Status const &st,
295                               std::size_t const R,
296                               MemberVariables &mv)
297     {
298         mv.status_factor_it = mv.factors.begin();
299         mv.layer_factor_it = mv.layer_factors_dimTree.begin();
300         for (std::size_t i = 0; i < TnsSize; ++i)
301         {
302             DiSCount(mv.dspl_subTns[i], mv.subTnsDims[i], st.options.proc_per_mode[i],
tnsDims[i], 1);
303             // for fiber communication and GatherV
304             DiSCount(mv.dspl_subTns_R[i], mv.subTnsDims_R[i], st.options.proc_per_mode[i],
tnsDims[i], static_cast<int>(R));
305             // information per layer
306             DiSCount(mv.dspl_local_update[i], mv.send_rcv_counts[i], mv.world_size /
st.options.proc_per_mode[i],
mv.subTnsDims[i][mv.fiber_rank[i]],
static_cast<int>(R));
307             // sizes and skips for sub factor
308             mv.subfactor_offsets = { mv.dspl_subTns[i][mv.fiber_rank[i]], 0 };
309             mv.subfactor_extents = { mv.subTnsDims[i][mv.fiber_rank[i]], static_cast<int>(R)
};
310             // get sub factor and compute its covariance matrix
311             mv.layer_factor_it->factor =
mv.status_factor_it->factor.slice(mv.subfactor_offsets, mv.subfactor_extents);
312             mv.layer_factor_it->gramian =

```

```

mv.layer_factor_it->factor.contract(mv.layer_factor_it->factor, mv.product_dims);
313     all_reduce( mv.fiber_comm[i],
314               inplace(mv.layer_factor_it->gramian.data()),
315               mv.RxR,
316               std::plus<double>() );
317
318     mv.rows_for_update[i] = mv.send_recv_counts[i][mv.layer_rank[i]] /
static_cast<int>(R);
319     // sizes and skips for sub tensor
320     mv.subTns_offsets[i] = mv.dims_subTns[i][mv.fiber_rank[i]];
321     mv.subTns_extents[i] = mv.subTnsDims[i][mv.fiber_rank[i]];
322     mv.local_mttkrp_T[i].resize(R, mv.rows_for_update[i]);
323     mv.layer_factors_T[i].resize(R, mv.subTnsDims[i][mv.fiber_rank[i]]);
324
325     mv.status_factor_it++;
326     mv.layer_factor_it++;
327   }
328 }
329
330 /*
331  * Make use of the dimensions and the number of processors per dimension
332  * and then calculates the dimensions of the subtensor and subfactor for
333  * each processor.
334  *
335  * After reading from files the given factors, then initializes the
336  * FactorDimTree struct for each processor with the data read from files.
337  *
338  * @tparam Dimensions      Array type containing the Tensor dimensions.
339  *
340  * @param tnsDims [in]     Tensor Dimensions. Each index contains the correspondng
341  *                          factor's rows length.
342  * @param st [in]         Struct where the returned values of @c CpdDimTree are
stored.
343  * @param R [in]         The rank of decomposition.
344  *
345  * @param paths [in]     Paths where the starting point-factors are located.
346  * @param mv [in,out]   Struct where ALS variables are stored.
347  *                       Updates @c st[] arrays with dimensions for subtensors and
348  *                       subfactors.
349  */
350 template<typename Dimensions>
351 void compute_sub_dimensions(Dimensions const &tnsDims,
352                             Status const &st,
353                             std::size_t const R,
354                             std::array<std::string, TnsSize+1> const &paths,
355                             MemberVariables &mv)
356 {
357     mv.layer_factor_it = mv.layer_factors_dimTree.begin();
358     for (std::size_t i = 0; i < TnsSize; ++i)
359     {
360         tnsDims[i], 1);
361         // for fiber communication and Gatherv
362         Di sCount(mv.dims_subTns_R[i], mv.subTnsDims_R[i], st.options.proc_per_mode[i],
tnsDims[i], static_cast<int>(R));
363         // information per layer
364         Di sCount(mv.dims_local_update[i], mv.send_recv_counts[i], mv.world_size /
st.options.proc_per_mode[i],
365                 mv.subTnsDims[i][mv.fiber_rank[i]],
static_cast<int>(R));
366         // sizes and skips for sub factor
367         mv.subfactor_offsets = { mv.dims_subTns[i][mv.fiber_rank[i]], 0 };
368         mv.subfactor_extents = { mv.subTnsDims[i][mv.fiber_rank[i]], static_cast<int>(R)
};
369
370         mv.temp_matrix = Matrix(mv.subTnsDims[i][mv.fiber_rank[i]], static_cast<int>(R));
371         read( paths[i+1],
372             mv.subTnsDims[i][mv.fiber_rank[i]]*static_cast<int>(R),
373             mv.dims_subTns_R[i][mv.fiber_rank[i]],
374             mv.temp_matrix );
375
376         // get sub factor and compute its covariance matrix
377         mv.layer_factor_it->factor = matrixToTensor(mv.temp_matrix,
mv.subTnsDims[i][mv.fiber_rank[i]], static_cast<int>(R));
378         mv.layer_factor_it->gramian =
mv.layer_factor_it->factor.contract(mv.layer_factor_it->factor, mv.product_dims);
379         all_reduce( mv.fiber_comm[i],
380                   inplace(mv.layer_factor_it->gramian.data()),
381                   mv.RxR,
382                   std::plus<double>() );
383
384         mv.rows_for_update[i] = mv.send_recv_counts[i][mv.layer_rank[i]] /
static_cast<int>(R);
385         // sizes and skips for sub tensor
386         mv.subTns_offsets[i] = mv.dims_subTns[i][mv.fiber_rank[i]];
387         mv.subTns_extents[i] = mv.subTnsDims[i][mv.fiber_rank[i]];
388         mv.local_mttkrp_T[i].resize(R, mv.rows_for_update[i]);

```

```

389         mv.layer_factors_T[i].resize(R, mv.subTnsDims[i][mv.fiber_rank[i]]);
390
391     mv.layer_factor_i_t++;
392 }
393 }
394
395 /*
396  * Based on each factor's constraint, a different
397  * update function is used at every outer iteration.
398  *
399  * Computes also factor^T * factor at the end.
400  *
401  * @param idx [in] Factor to be updated.
402  * @param R [in] The rank of decomposition.
403  * @param st [in] Struct where the returned values of @c CpdDimTree
404  * are stored.
405  * @param mv [in,out] Struct where ALS variables are stored.
406  * Updates the current layer factor (@c Matrix type) and
407  * then updates the same factor of @c FactorDimTree type.
408  */
409 void update_factor( int idx, const int,
410                   std::size_t R, const int,
411                   Status const &st,
412                   Member_Variables &mv )
413 {
414     switch ( st.options.constraints[idx] )
415     {
416     case Constraint::unconstrained:
417     {
418         // communicate the local mttkrp
419         v2::reduce_scatter( mv.layer_comm[idx],
420                           mv.layer_mttkrp_T[idx],
421                           mv.send_recv_counts[idx][0],
422                           mv.local_mttkrp_T[idx] );
423
424         mv.local_mttkrp[idx] = mv.local_mttkrp_T[idx].transpose();
425
426         if(mv.rows_for_update[idx] != 0)
427             mv.local_factors[idx] = mv.local_mttkrp[idx] *
428             mv.cwise_factor_product.inverse(); // Compute new factor
429
430         break;
431     }
432     case Constraint::nonnegativity:
433     {
434         // communicate the local mttkrp
435         v2::reduce_scatter( mv.layer_comm[idx],
436                           mv.layer_mttkrp_T[idx],
437                           mv.send_recv_counts[idx][0],
438                           mv.local_mttkrp_T[idx] );
439
440         mv.local_mttkrp[idx] = mv.local_mttkrp_T[idx].transpose();
441         mv.nesterov_old_layer_factor = mv.layer_factors[idx];
442         if (mv.rows_for_update[idx] != 0) {
443             NesterovMMLS(mv.cwise_factor_product, mv.local_mttkrp[idx],
444             st.options.nesterov_delta_1,
445             st.options.nesterov_delta_2, mv.local_factors[idx]);
446         }
447         break;
448     }
449     case Constraint::orthogonality:
450     {
451         all_reduce( mv.layer_comm[idx],
452                   inplace(mv.layer_mttkrp[idx].data()),
453                   mv.subTnsDims_R[idx][mv.fiber_rank[idx]],
454                   std::plus<double>() );
455
456         if (mv.rows_for_update[idx] != 0) {
457             mv.local_mttkrp[idx] =
458             mv.layer_mttkrp[idx].block(mv.dims_local_update[idx][mv.layer_rank[idx]] / static_cast<int>(R), 0,
459             mv.rows_for_update[idx],
460             static_cast<int>(R));
461             mv.layer_mttkrp[idx].noalias() = mv.layer_mttkrp[idx].transpose() *
462             mv.layer_mttkrp[idx];
463         }
464
465         all_reduce( mv.fiber_comm[idx],
466                   inplace(mv.temp_matrix.data()),
467                   mv.RxR,
468                   std::plus<double>() );
469
470         Eigen::SelfAdjointEigenSolver<Matrix> ei_gensol_ver(mv.temp_matrix);
471         mv.temp_matrix.noalias() = (ei_gensol_ver.eigenvalues()
472         * (ei_gensol_ver.eigenvalues().cwiseInverse().cwiseSqrt()).asDiagonal())
473         * (ei_gensol_ver.eigenvalues().transpose());
474     }
475     }
476 }

```

```

469
470         if(mv.rows_for_update[idx] != 0)
471             mv.local_factors[idx].noalias() = mv.local_mttkrp[idx] * mv.temp_matrix;
472         // mv.local_factors[idx].noalias() = mv.local_mttkrp[idx] *
(mv.temp_matrix.pow(-0.5));
473         break;
474     }
475     case Constraint::sparsity:
476         break;
477     default: // in case of Constraint::constant
478         break;
479 } // end of constraints switch
480
481 if(st.options.constraints[idx] != Constraint::constant)
482 {
483     mv.local_factors_T[idx] = mv.local_factors[idx].transpose();
484     v2::all_gatherv(mv.layer_comm[idx],
485                   mv.local_factors_T[idx],
486                   mv.send_recv_counts[idx][mv.layer_rank[idx]],
487                   mv.send_recv_counts[idx][0],
488                   mv.dispatch_local_update[idx][0],
489                   mv.layer_factors_T[idx]);
490
491     mv.layer_factors[idx] = mv.layer_factors_T[idx].transpose();
492     mv.factor_T_factor = mv.layer_factors_T[idx] * mv.layer_factors[idx];
493     all_reduce(mv.fiber_comm[idx],
494               inplace(mv.factor_T_factor.data()),
495               mv.RxR,
496               std::plus<double>());
497
498     if(st.options.constraints[idx] == Constraint::nonnegativity)
499     {
500         if((mv.factor_T_factor.diagonal()).minCoeff()==0)
501         {
502             mv.layer_factors[idx] = 0.9 * mv.layer_factors[idx] + 0.1 *
mv.nesterov_ol_d_layer_factor;
503             all_reduce(mv.fiber_comm[idx],
504                       inplace(mv.factor_T_factor.data()),
505                       mv.RxR,
506                       std::plus<double>());
507         }
508     }
509
510     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
511     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
512     }
513 }
514
515 /*
516 * At the end of the algorithm processor 0
517 * collects each part of the factor that each
518 * processor holds and return them in status.factors.
519 *
520 * @tparam Dimensions      Array type containing the Tensor dimensions.
521 *
522 * @tparam tnsDims [in]    Tensor Dimensions. Each index contains the corresponding
523 *                          factor's rows length.
524 * @tparam R          [in] The rank of decomposition.
525 * @tparam mv         [in] Struct where ALS variables are stored.
526 *                    Use variables to compute result factors by gathering each
527 *                    part of the factor from processors.
528 * @tparam st         [in, out] Struct where the returned values of @c CpdDimTree are
stored.
529 *
530 * Stores the resulted factors.
531 */
532 template<typename Dimensions>
533 void gather_final_factors(Dimensions const &tnsDims,
534                          std::size_t const R,
535                          MemberVariables &mv,
536                          Status &st)
537 {
538     for(std::size_t i=0; i<TnsSize; ++i)
539     {
540         mv.temp_matrix = tensorToMatrix(mv.layer_factors_dimTree[i].factor,
mv.subTnsDims[i][mv.fiber_rank[i]], static_cast<int>(R));
541         mv.layer_factors_T[i] = mv.temp_matrix.transpose();
542     }
543     for(std::size_t i=0; i<TnsSize; ++i)
544     {
545         mv.temp_matrix.resize(static_cast<int>(R), tnsDims[i]);
546         // Gather from all processors to processor with rank 0 the final factors
547         v2::gather(mv.fiber_comm[i],

```

```

548         mv.layer_factors_T[i],
549         mv.subTnsDims_R[i][mv.fiber_rank[i]],
550         mv.subTnsDims_R[i][0],
551         mv.dims_subTns_R[i][0],
552         0,
553         mv.temp_matrix);
554     }
555     st.factors[i] = mv.temp_matrix.transpose();
556 }
557 }
558
559 /*
560 * @brief Line Search Acceleration
561 *
562 * Performs an acceleration step in the updated factors, and keeps the accelerated
563 factors when
564 * the step succeeds. Otherwise, the acceleration step is ignored.
565 * Line Search Acceleration reduces the number outer iterations in the ALS algorithm.
566 *
567 * @note This implementation ONLY, if factors are of @c Matrix type.
568 *
569 * @param grid_comm [in] MPI communicator where the new cost function value
570 * will be communicated and computed.
571 * @param R [in] Rank of the factorization.
572 * @param mv [in,out] Struct where ALS variables are stored.
573 * In case the acceleration is successful layer factor^T *
574 factor
575 * and layer factor variables are updated.
576 * @param st [in,out] Struct of the returned values of @c CpdDimTree are stored.
577 * If the acceleration succeeds updates cost function value.
578
579 */
580 void Line_search_accel (CartCommunicator const &grid_comm,
581                       std::size_t const R,
582                       MemberVariables &mv,
583                       Status &st)
584 {
585     double f_accel = 0.0; // Objective Value after the acceleration step
586     double const accel_step = pow(st.ao_iter+1, (1.0/(st.options.accel_coeff)));
587
588     Matrix factor;
589     Matrix old_factor;
590     MatrixArray accel_factors;
591     MatrixArray accel_gramians;
592
593     for(std::size_t i=0; i<TnsSize; ++i)
594     {
595         factor = tensorToMatrix(mv.layer_factors_dimTree[i].factor,
596 mv.subTnsDims[i][mv.fiber_rank[i]], static_cast<int>(R));
597         old_factor = tensorToMatrix(mv.old_factors[i].factor,
598 mv.subTnsDims[i][mv.fiber_rank[i]], static_cast<int>(R));
599         accel_factors[i] = old_factor + accel_step * (factor - old_factor);
600         accel_gramians[i] = accel_factors[i].transpose() * accel_factors[i];
601         all_reduce(mv.fiber_comm[i],
602 iplace(accel_gramians[i].data()),
603 mv.RxR,
604 mv.std::plus<double>());
605     }
606
607     f_accel = accel_cost_function(grid_comm, mv, st, accel_factors, accel_gramians);
608     if (st.f_value > f_accel)
609     {
610         for(std::size_t i=0; i<TnsSize; ++i)
611         {
612             mv.layer_factors_dimTree[i].factor = matrixToTensor(accel_factors[i],
613 mv.subTnsDims[i][mv.fiber_rank[i]], static_cast<int>(R));
614             mv.layer_factors_dimTree[i].gramian = matrixToTensor(accel_gramians[i],
615 static_cast<int>(R), static_cast<int>(R));
616         }
617         st.f_value = f_accel;
618         if(grid_comm.rank() == 0)
619             Partensor()->Logger()->info("Acceleration Step SUCCEEDED! at iter: {}",
620 st.ao_iter);
621     }
622     else
623         st.options.accel_fail++;
624
625     if (st.options.accel_fail==5)
626     {
627         st.options.accel_fail=0;
628         st.options.accel_coeff++;
629     }
630 }
631
632 /*
633 * Parallel implementation of als method with MPI.
634 */

```



```

627         * @tparam Dimensions      Array type containing the Tensor dimensions.
628         *
629         * @param grid_comm [in]    MPI communicator where the new cost function value
630         *                        will be communicated and computed.
631         * @param tnsDims      [in]    Tensor Dimensions. Each index contains the corresponding
632         *                        factor's rows length.
633         * @param R            [in]    The rank of decomposition.
634         * @param mv           [in]    Struct where ALS variables are stored and being updated
635         *                        until a termination condition is true.
636         * @param st           [in,out] Struct where the returned values of @c CpdDimTree are
        stored.
637     */
638     template<typename Dimensions>
639     void als(CartCommunicator const &grid_comm,
640            Dimensions const &tnsDims,
641            std::size_t const R,
642            MemberVariables &mv,
643            Status &status)
644     {
645         mv.tnsX_mat_lastFactor_T = (Matrixzation(mv.subTns, lastFactor)).transpose();
646         if(status.options.normalization)
647         {
648             choose_normlization_factor(status, mv.all_orthogonal, mv.weight_factor);
649         }
650
651         std::iota(mv.LabelSet.begin(), mv.LabelSet.end(), 1); // increments from 1 to
        LabelSet.size())
652
653         ExprTree<TnsSize> tree;
654         tree.Create(mv.LabelSet, mv.subTns_extents, R, mv.subTns);
655
656         mv.layer_factor_it = mv.layer_factors_dimTree.begin();
657         for(std::size_t k = 0; k<TnsSize; k++)
658         {
659             mv.layer_factor_it->l_eaf = static_cast<TnsNode<1>*>(search_leaf(k+1, tree));
660             mv.layer_factor_it++;
661         }
662
663         // Wait for all processors to reach here
664         grid_comm.barrier();
665
666         // ---- Loop until ALS converges ----
667         while(1)
668         {
669             status.ao_iter++;
670             mv.layer_factor_it = mv.layer_factors_dimTree.begin();
671             if(!grid_comm.rank())
672                 Partensor()->info("iter: {} -- fvalue: {} --
        relative_costFunction: {}", status.ao_iter,
        status.f_value, status.rel_costFunction);
673
674             for(std::size_t i=0; i<TnsSize; i++)
675             {
676                 mv.layer_factor_it->l_eaf->updateTree(TnsSize, i, mv.layer_factor_it);
677
678                 // Maps from Eigen Tensor to Eigen Matrix
679                 mv.layer_mttkrp_T[i] = tensorToMatrix(*reinterpret_cast<TensorMatrixType
        *>(mv.layer_factor_it->l_eaf->TensorX()),
        static_cast<int>(R),
        mv.subTnsDims[i][mv.fiber_rank[i]]);
680                 mv.layer_mttkrp_T[i] = mv.layer_mttkrp_T[i].transpose();
681                 mv.cwise_factor_product =
        tensorToMatrix(mv.layer_factor_it->l_eaf->Gramian(), static_cast<int>(R), static_cast<int>(R));
682                 mv.layer_factors[i] = tensorToMatrix(mv.layer_factor_it->factor,
        mv.subTnsDims[i][mv.fiber_rank[i]], static_cast<int>(R));
683                 mv.local_factors[i] =
        mv.layer_factors[i].block(mv.dpls_local_update[i][mv.layer_rank[i]] / static_cast<int>(R), 0,
        mv.rows_for_update[i],
        static_cast<int>(R));
684
685                 update_factor(i, R, status, mv);
686                 mv.layer_factor_it++;
687             }
688             cost_function(grid_comm, mv, status);
689             status.rel_costFunction = status.f_value / sqrt(status.frob_tns);
690             if(status.options.normalization && !mv.all_orthogonal)
691                 Normalize(mv.weight_factor, static_cast<int>(R), tnsDims, mv.factors);
692
693             // ---- Termination condition ----
694             if (status.ao_iter >= status.options.max_iter || status.rel_costFunction <
        status.options.threshold_error)
695             {
696                 gather_final_factors(tnsDims, R, mv, status);
697                 if(grid_comm.rank() == 0)
698                     {
699
700
701
702

```

```

703         Partensor()->Logger()->info("Processor 0 collected all {} factors.mn",
TnsSize);
704         if(status.options.writeToFile)
705             writeFactorsToFile(status);
706     }
707     break;
708 }
709
710     if (status.options.acceleration)
711     {
712         mv.norm_factors = mv.layer_factors_dimTree;
713         // ---- Acceleration Step ----
714         if (status.ao_iter > 1)
715             line_search_accel(grid_comm, R, mv, status);
716
717         mv.old_factors = mv.norm_factors;
718     }
719 }
720 }
721
722 Status operator()(Tensor_ const &tnsX,
723                  std::size_t const R)
724 {
725     Options options = MakeOptions<Tensor_>(execution::openmpi_policy());
726     Status status(options);
727     Member_Variabls mv(R, status.options.proc_per_mode);
728
729     // Communicator with cartesian topology
730     CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);
731
732     // Functions that create layer and fiber grids.
733     create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
734     create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);
735
736     // extract dimensions from tensor
737     Dimensions const &tnsDims = tnsX.dimensions();
738     // produce estimate factors using uniform distribution with entries in [0, 1].
739     makeFactors(tnsDims, status.options.constraints, R, mv.factors);
740
741     compute_sub_dimensions(tnsDims, status, R, mv);
742     // Normalize each layer_factor, compute status.frob_tns and status.f_value
743     // Normalize(R, subTns_extents, layer_factors_dimTree);
744     // Each processor takes a subtensor from tnsX
745     mv.subTns = tnsX.slice(mv.subTns_offsets, mv.subTns_extents);
746     cost_function_init(grid_comm, R, mv, status);
747     status.rel_costFunction = status.f_value / sqrt(status.frob_tns);
748
749     als(grid_comm, tnsDims, R, mv, status);
750
751     return status;
752 }
753
754 Status operator()(Tensor_ const &tnsX,
755                  std::size_t const R,
756                  Options const &options)
757 {
758     Status status(options);
759     Member_Variabls mv(R, status.options.proc_per_mode);
760
761     // Communicator with cartesian topology
762     CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);
763
764     // Functions that create layer and fiber grids.
765     create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
766     create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);
767
768     // extract dimensions from tensor
769     Dimensions const &tnsDims = tnsX.dimensions();
770     // produce estimate factors using uniform distribution with entries in [0, 1].
771     makeFactors(tnsDims, status.options.constraints, R, mv.factors);
772
773     compute_sub_dimensions(tnsDims, status, R, mv);
774     // Normalize each layer_factor, compute status.frob_tns and status.f_value
775     // Normalize(R, subTns_extents, layer_factors_dimTree);
776     // Each processor takes a subtensor from tnsX
777     mv.subTns = tnsX.slice(mv.subTns_offsets, mv.subTns_extents);
778     cost_function_init(grid_comm, R, mv, status);
779     status.rel_costFunction = status.f_value / sqrt(status.frob_tns);
780     switch (status.options.method)
781     {
782     case Method::als:
783     {
784         als(grid_comm, tnsDims, R, mv, status);
785         break;
786     }
787     case Method::rnd:
788     break;
789 }

```

```

808         case Method::bc:
809             break;
810         default:
811             break;
812     }
813     return status;
814 }
815
816 template <typename MatrixArray_>
817 Status operator()(Tensor_ const &tnsX,
818                 std::size_t const R,
819                 MatrixArray_ const &factorsInit)
820 {
821     Options options = MakeOptions<Tensor_>(execution::openmpi_policy());
822     Status status(options);
823     Member_VariabLes mv(R, status.options.proc_per_mode);
824
825     // Communicator with cartesian topology
826     CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);
827
828     // Functions that create layer and fiber grids.
829     create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
830     create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);
831
832     // extract dimensions from tensor
833     Dimensions const &tnsDims = tnsX.dimensions();
834     // Copy factorsInit data to status.factors - FactorDimTree data struct
835     fillDimTreeFactors(factorsInit, status.options.constraints, mv.factors);
836
837     compute_sub_dimensions(tnsDims, status, R, mv);
838     // Normalize each layer_factor, compute status.frob_tns and status.f_value
839     // Normalize(R, subTns_extents, layer_factors_dimTree);
840     // Each processor takes a subtensor from tnsX
841     mv.subTns = tnsX.slice(mv.subTns_offsets, mv.subTns_extents);
842     cost_function_init(grid_comm, R, mv, status);
843     status.rel_costFunction = status.f_value / sqrt(status.frob_tns);
844
845     als(grid_comm, tnsDims, R, mv, status);
846     return status;
847 }
848
849 template <typename MatrixArray_>
850 Status operator()(Tensor_ const &tnsX,
851                 std::size_t const R,
852                 Options const &options,
853                 MatrixArray_ const &factorsInit)
854 {
855     Status status(options);
856     Member_VariabLes mv(R, status.options.proc_per_mode);
857
858     // Communicator with cartesian topology
859     CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);
860
861     // Functions that create layer and fiber grids.
862     create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
863     create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);
864
865     // extract dimensions from tensor
866     Dimensions const &tnsDims = tnsX.dimensions();
867     // Copy factorsInit data to status.factors - FactorDimTree data struct
868     fillDimTreeFactors(factorsInit, status.options.constraints, mv.factors);
869
870     compute_sub_dimensions(tnsDims, status, R, mv);
871     // Normalize each layer_factor, compute status.frob_tns and status.f_value
872     // Normalize(R, subTns_extents, layer_factors_dimTree);
873     // Each processor takes a subtensor from tnsX
874     mv.subTns = tnsX.slice(mv.subTns_offsets, mv.subTns_extents);
875     cost_function_init(grid_comm, R, mv, status);
876     status.rel_costFunction = status.f_value / sqrt(status.frob_tns);
877     switch (status.options.method)
878     {
879     case Method::als:
880     {
881         als(grid_comm, tnsDims, R, mv, status);
882         break;
883     }
884     case Method::rnd:
885         break;
886     case Method::bc:
887         break;
888     default:
889         break;
890     }
891     return status;
892 }
893
894 template <std::size_t TnsSize>

```

```

930     Status operator()(std::array<int, TnsSize> const &tnsDims,
931                     std::size_t      const R,
932                     std::string       const &path)
933     {
934         using TensorType = Tensor<stati_c_cast<int>(TnsSize)>;
935
936         Options      options = MakeOptions<TensorType>(executi on::openmpi_policy());
937         Status       status(options);
938         Member_Vari ables mv(R, status.options.proc_per_mode);
939
940         // Communicator with cartesian topology
941         CartCommuni cator grid_comm(mv.world, status.options.proc_per_mode, true);
942
943         // Functions that create layer and fiber grids.
944         create_l ayer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
945         create_fi ber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);
946
947         // produce estimate factors using uniform distribution with entries in [0, 1].
948         makeFactors(tnsDims, status.options.constraints, R, mv.factors);
949
950         compute_sub_dims(tnsDims, status, R, mv);
951         // Normalize each layer_factor, compute status.frob_tns and status.f_value
952         // Normalize(R, subTns_extents, layer_factors_dimTree);
953         // Each processor takes a subtensor from tnsX
954         mv.subTns.resize(mv.subTns_extents);
955         readTensor( path, tnsDims, mv.subTns_extents, mv.subTns_offsets, mv.subTns );
956         cost_function_init(grid_comm, R, mv, status);
957         status.rel_costFunction = status.f_value / sqrt(status.frob_tns);
958
959         als(grid_comm, tnsDims, R, mv, status);
960         return status;
961     }
962
963     template <std::size_t TnsSize>
964     Status operator()(std::array<int, TnsSize> const &tnsDims,
965                     std::size_t      const R,
966                     std::string       const &path,
967                     Options           const &options)
968     {
969         Status       status(options);
970         Member_Vari ables mv(R, status.options.proc_per_mode);
971
972         // Communicator with cartesian topology
973         CartCommuni cator grid_comm(mv.world, status.options.proc_per_mode, true);
974
975         // Functions that create layer and fiber grids.
976         create_l ayer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
977         create_fi ber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);
978
979         // produce estimate factors using uniform distribution with entries in [0, 1].
980         makeFactors(tnsDims, status.options.constraints, R, mv.factors);
981
982         compute_sub_dims(tnsDims, status, R, mv);
983         // Normalize each layer_factor, compute status.frob_tns and status.f_value
984         // Normalize(R, subTns_extents, layer_factors_dimTree);
985         // Each processor takes a subtensor from tnsX
986         mv.subTns.resize(mv.subTns_extents);
987         readTensor( path, tnsDims, mv.subTns_extents, mv.subTns_offsets, mv.subTns );
988         cost_function_init(grid_comm, R, mv, status);
989         status.rel_costFunction = status.f_value / sqrt(status.frob_tns);
990         swit ch ( status.options.method )
991         {
992             case Method:: als:
993             {
994                 als(grid_comm, tnsDims, R, mv, status);
995                 break;
996             }
997             case Method:: rnd:
998                 break;
999             case Method:: bc:
1000                 break;
1001             default:
1002                 break;
1003         }
1004         return status;
1005     }
1006
1007     template <std::size_t TnsSize>
1008     Status operator()(std::array<int, TnsSize> const &tnsDims,
1009                     std::size_t      const R,
1010                     std::array<std::string, TnsSize+1> const &paths)
1011     {
1012         using TensorType = Tensor<stati_c_cast<int>(TnsSize)>;
1013
1014         Options      options = MakeOptions<TensorType>(executi on::openmpi_policy());
1015         Status       status(options);
1016         Member_Vari ables mv(R, status.options.proc_per_mode);

```

```

1046
1047 // Communicator with cartesian topology
1048 CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);
1049
1050 // Functions that create layer and fiber grids.
1051 create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
1052 create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);
1053
1054 compute_sub_dimensions(tnsDims, status, R, paths, mv);
1055 // Normalize each layer_factor, compute status.frob_tns and status.f_value
1056 // Normalize(R, subTns_extents, layer_factors_dimTree);
1057 // Each processor takes a subtensor from tnsX
1058 mv.subTns.resize(mv.subTns_extents);
1059 readTensor(paths[0], tnsDims, mv.subTns_extents, mv.subTns_offsets, mv.subTns);
1060 cost_function_init(grid_comm, R, mv, status);
1061 status.rel_costFunction = status.f_value / sqrt(status.frob_tns);
1062 als(grid_comm, tnsDims, R, mv, status);
1063
1064 return status;
1065 }
1066
1067 template <std::size_t TnsSize>
1068 Status operator()(std::array<int, TnsSize> const &tnsDims,
1069                 std::size_t const R,
1070                 std::array<std::string, TnsSize+1> const &paths,
1071                 Options const &options)
1072 {
1073     Status status(options);
1074     MemberVariables mv(R, status.options.proc_per_mode);
1075
1076     // Communicator with cartesian topology
1077     CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);
1078
1079     // Functions that create layer and fiber grids.
1080     create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
1081     create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);
1082
1083     compute_sub_dimensions(tnsDims, status, R, paths, mv);
1084     // Normalize each layer_factor, compute status.frob_tns and status.f_value
1085     // Normalize(R, subTns_extents, layer_factors_dimTree);
1086     // Each processor takes a subtensor from tnsX
1087     mv.subTns.resize(mv.subTns_extents);
1088     readTensor(paths[0], tnsDims, mv.subTns_extents, mv.subTns_offsets, mv.subTns);
1089     cost_function_init(grid_comm, R, mv, status);
1090     status.rel_costFunction = status.f_value / sqrt(status.frob_tns);
1091     switch ( status.options.method )
1092     {
1093     case Method::als:
1094     {
1095         als(grid_comm, tnsDims, R, mv, status);
1096         break;
1097     }
1098     case Method::rnd:
1099     break;
1100     case Method::bc:
1101     break;
1102     default:
1103     break;
1104     }
1105     return status;
1106 }
1107
1108 };
1109 } // end namespace internal
1110 } // end namespace v1
1111
1112 } //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.](#)

```

1 #ifndef DOXYGEN_SHOULD_SKIP_THIS
15 #endif // DOXYGEN_SHOULD_SKIP_THIS
16 /*****
25 #ifndef PARTENSOR_CPD_HPP
26 #error "CPD_MPI can only be included inside CPD"
27 #endif /* PARTENSOR_CPD_HPP */
28
29 namespace partensor
30 {
31
32     inline namespace v1 {
33
34         namespace internal {
35             template<typename Tensor_>
36             struct CPD<Tensor_, execution::openmpi_policy> : public CPD_Base<Tensor_>
37             {
38                 using CPD_Base<Tensor_>::TnsSize;
39                 using CPD_Base<Tensor_>::lastFactor;
40                 using typename CPD_Base<Tensor_>::Dimensions;
41                 using typename CPD_Base<Tensor_>::MatrixArray;
42                 using typename CPD_Base<Tensor_>::DataType;
43
44                 using IntArray = typename TensorTraits<Tensor_>::IntArray;
45                 using CartCommunicator = partensor::cartesian_communicator; // From ParallelWrapper.hpp
46                 using CartCommVector = std::vector<CartCommunicator>;
47                 using IntVector = std::vector<int>;
48                 using Int2DVector = std::vector<std::vector<int>>;
49
50                 using Options = partensor::Options<Tensor_, execution::openmpi_policy, DefaultValues>;
51                 using Status = partensor::Status<Tensor_, execution::openmpi_policy, DefaultValues>;
52
53                 // Variables that will be used in cpd implementations.
54                 struct Member_Variables
55                 {
56                     MPI_Communicator &world = Partensor()->MpiCommunicator(); // MPI_COMM_WORLD
57
58                     double local_f_value;
59                     int RxR;
60                     int world_size;
61
62                     Int2DVector displ_subTns; // skipping dimension "rows" for each subtensor
63                     Int2DVector displ_subTnsR; // skipping dimension "rows" for each subtensor times R (
64                     for MPI communication purposes )
65                     Int2DVector subTnsDims; // dimensions of subtensor
66                     Int2DVector subTnsDimsR; // dimensions of subtensor times R ( for MPI communication
67                     purposes )
68                     Int2DVector displ_local_update; // displacement in the local factor for update rows
69
70                     Int2DVector send_recv_counts; // rows to be communicated after update times R
71
72                     CartCommVector layer_comm;
73                     CartCommVector fiber_comm;
74
75                     IntArray layer_rank;
76                     IntArray fiber_rank;
77                     IntArray rows_for_update;
78                     IntArray subTns_offsets;
79                     IntArray subTns_extents;
80
81                     MatrixArray proc_krao;
82                     MatrixArray layer_factors;
83                     MatrixArray layer_factors_T;
84                     MatrixArray factors_T;
85                     MatrixArray factor_T_factor;
86                     MatrixArray local_mttkrp;
87                     MatrixArray layer_mttkrp;
88                     MatrixArray local_mttkrp_T;
89                     MatrixArray layer_mttkrp_T;
90                     MatrixArray subTns_mat;
91                     MatrixArray local_factors;
92                     MatrixArray local_factors_T;
93                     MatrixArray norm_factors;
94                     MatrixArray old_factors;
95                     MatrixArray true_factors;
96
97                     Matrix xwise_factor_product;
98                     Matrix tnsX_mat_lastFactor_T;
99                     Matrix temp_matrix;
100                    Matrix nesterov_old_layer_factor;
101
102                    Tensor_ subTns;
103
104                };
105            };
106        };
107    };

```

```

107         bool                all_orthogonal = true;
108         int                 weight_factor;
109
110         /*
111         * Calculates if the number of processors given from terminal
112         * are equal to the processors in the implementation.
113         *
114         * @param procs [in] @c stl array with the number of processors per
115         *                 dimension of the tensor.
116         */
117         void check_processor_availability(std::array<int, TnsSize> const &procs)
118         {
119             // MPI_Environment &env = Partensor()->MPI_Environment();
120             world_size = world_size();
121             // numprocs must be product of options.proc_per_mode
122             if (std::accumulate(procs.begin(), procs.end(), 1,
123                               std::multiplies<int>()) != world_size && world.rank() == 0) {
124                 Partensor()->Logger()->error("The product of the processors per mode must be equal to
125                 {}m", world_size);
126                 // env.abort(-1);
127             }
128
129             Member_Variabes() = default;
130             Member_Variabes(int R, std::array<int, TnsSize> &procs) : local_f_value(0.0),
131                 RxR(R*R),
132                 disp_s_subTns(TnsSize),
133                 disp_s_subTns_R(TnsSize),
134                 subTnsDiMs(TnsSize),
135                 subTnsDiMs_R(TnsSize),
136                 disp_s_local_update(TnsSize),
137                 send_recv_counts(TnsSize)
138         {
139             check_processor_availability(procs);
140             layer_comm.reserve(TnsSize);
141             fiber_comm.reserve(TnsSize);
142         }
143
144             Member_Variabes(Member_Variabes const &) = default;
145             Member_Variabes(Member_Variabes &&) = default;
146
147             Member_Variabes &operator=(Member_Variabes const &) = default;
148             Member_Variabes &operator=(Member_Variabes &&) = default;
149         };
150
151         /*
152         * In case option variable @c writeToFile is enabled then, before the end
153         * of the algorithm writes the resulted factors in files, where their
154         * paths are specified before compiling in @ options.final_factors_path.
155         *
156         * @param st [in] Struct where the returned values of @c Cpd are stored.
157         */
158         void writeFactorsToFile(Status const &st)
159         {
160             std::size_t size;
161             for(std::size_t i=0; i<TnsSize; ++i)
162             {
163                 size = st.factors[i].rows() * st.factors[i].cols();
164                 partensor::write(st.factors[i],
165                                st.options.final_factors_paths[i],
166                                size);
167             }
168         }
169
170         /*
171         * Compute the cost function value at the end of each outer iteration
172         * based on the last factor.
173         *
174         * @param grid_comm [in] MPI communicator where the new cost function value
175         *                   will be communicated and computed.
176         * @param mv [in] Struct where ALS variables are stored.
177         * @param st [in,out] Struct where the returned values of @c Cpd are stored.
178         *                   In this case the cost function value is updated.
179         */
180         void cost_function_init( CartCommunicator const &grid_comm,
181                               Member_Variabes &mv,
182                               Status &st )
183         {
184             mv.local_f_value =
185                 ((mv.proc_krao[lastFactor].transpose() * mv.tnsX_mat_lastFactor_T) *
186                 mv.layer_factors[lastFactor]).trace();
187             all_reduce( grid_comm,
188                       inplace(&mv.local_f_value),
189                       1,
190                       std::plus<double>() );
191
192             Matrix cwiseFactor_prod = PartialCwiseProd(mv.factor_T_factor, lastFactor) *

```

```

192     mv.factor_T_factor[LastFactor];
193     st.f_value =
194         sqrt(st.frob_tns - 2 * mv.local_f_value + cwiseFactor_prod.trace());
195     }
196     /*
197     * Compute the cost function value at the end of each outer iteration
198     * based on the last factor.
199     *
200     * @param grid_comm [in] MPI communicator where the new cost function value
201     * will be communicated and computed.
202     * @param mv [in] Struct where ALS variables are stored.
203     * @param st [in,out] Struct where the returned values of @c Cpd are stored.
204     * In this case the cost function value is updated.
205     */
206     void cost_function( CartCommunicator const &grid_comm,
207                       Member_Variab les &mv,
208                       Status &st )
209     {
210         mv.local_f_value = ((mv.layer_mttkrp_T[LastFactor] * mv.layer_factors[LastFactor]).trace());
211         all_reduce( grid_comm,
212                   inplace(&mv.local_f_value),
213                   1,
214                   std::plus<double>() );
215
216         Matrix cwiseFactor_prod = PartialCwiseProd(mv.factor_T_factor, LastFactor) *
192 mv.factor_T_factor[LastFactor];
217     st.f_value =
218         sqrt(st.frob_tns - 2 * mv.local_f_value + cwiseFactor_prod.trace());
219     }
220
221     /*
222     * Compute the cost function value at the end of each outer iteration
223     * based on the last accelerated factor.
224     *
225     * @param grid_comm [in] MPI communicator where the new cost function value
226     * will be communicated and computed.
227     * @param mv [in] Struct where ALS variables are stored.
228     * @param st [in] Struct where the returned values of @c Cpd are stored.
229     * In this case the cost function value is updated.
230     * @param factors [in] Accelerated factors.
231     * @param factors_T_factors [in] Gramian matrices of factors.
232     *
233     * @returns The cost function calculated with the accelerated factors.
234     */
235     double accel_cost_function(CartCommunicator const &grid_comm,
236                               Member_Variab les const &mv,
237                               Status const &st,
238                               MatrixArray const &factors,
239                               MatrixArray const &factors_T_factors)
240     {
241         double local_f_value =
242             ((PartialKhatRao(factors, LastFactor).transpose() * mv.tnsX_mat_LastFactor_T) *
243              factors[LastFactor]).trace();
244         all_reduce( grid_comm,
245                   inplace(&local_f_value),
246                   1,
247                   std::plus<double>() );
248         Matrix cwiseFactor_prod = PartialCwiseProd(factors_T_factors, LastFactor) *
249         factors_T_factors[LastFactor];
250         return sqrt(st.frob_tns - 2 * local_f_value + cwiseFactor_prod.trace());
251     }
252     /*
253     * Make use of the dimensions and the number of processors per dimension
254     * and then calculates the dimensions of the subtensor and subfactor for
255     * each processor.
256     *
257     * @tparam Dimensions Array type containing the length of Tensor's dimensions.
258     *
259     * @param tnsDims [in] Tensor Dimensions. Each index contains the corresponding
260     * factor's rows length.
261     * @param st [in] Struct where the returned values of @c Cpd are stored.
262     * @param R [in] The rank of decomposition.
263     * @param mv [in,out] Struct where ALS variables are stored.
264     * Updates @c stl arrays with dimensions for subtensors and
265     * subfactors.
266     */
267     template<typename Dimensions>
268     void compute_sub_dimensions(Dimensions const &tnsDims,
269                               Status const &st,
270                               std::size_t const R,
271                               Member_Variab les &mv)
272     {
273         for (std::size_t i = 0; i < TnsSize; ++i)
274         {
275             mv.factor_T_factor[i].noalias() = st.factors[i].transpose() * st.factors[i];

```



```

275         DisCount(mv.dspl_s_subTns[i], mv.subTnsDims[i], st.options.proc_per_mode[i], tnsDims[i], 1);
276         // for fiber communication and Gatherv
277         DisCount(mv.dspl_s_subTns_R[i], mv.subTnsDims_R[i], st.options.proc_per_mode[i], tnsDims[i],
278 static_cast<int>(R));
279         // information per layer
280         DisCount(mv.dspl_s_local_update[i], mv.send_recv_counts[i], mv.world_size /
st.options.proc_per_mode[i],
281                 mv.subTnsDims[i][mv.fiber_rank[i]],
282 static_cast<int>(R));
283         mv.rows_for_update[i] = mv.send_recv_counts[i][mv.layer_rank[i]] / static_cast<int>(R);
284         mv.subTns_offsets[i] = mv.dspl_s_subTns[i][mv.fiber_rank[i]];
285         mv.subTns_extents[i] = mv.subTnsDims[i][mv.fiber_rank[i]];
286     }
287 }
288
289 /*
290 * Based on each factor's constraint, a different
291 * update function is used at every outer iteration.
292 *
293 * Computes also factor^T * factor at the end.
294 *
295 * @param idx [in] Factor to be updated.
296 * @param R [in] The rank of decomposition.
297 * @param st [in] Struct where the returned values of @c Cpd are stored.
298 * Here constraints and options variables are needed.
299 * @param mv [in,out] Struct where ALS variables are stored.
300 * Updates the factors of each layer.
301 */
302 void update_factor(int const idx,
303                   std::size_t const R,
304                   Status const &st,
305                   Member_Variab les &mv )
306 {
307     switch ( st.options.constraints[idx] )
308     {
309     case Constraint::unconstrained:
310     {
311         v2::reduce_scatter( mv.layer_comm[idx],
312                             mv.layer_mttkrp_T[idx],
313                             mv.send_recv_counts[idx][0],
314                             mv.local_mttkrp_T[idx] );
315
316         mv.local_mttkrp[idx] = mv.local_mttkrp_T[idx].transpose();
317         if (mv.rows_for_update[idx] != 0)
318             mv.local_factors[idx].noalias() = mv.local_mttkrp[idx] *
mv.cwise_factor_product.inverse();
319         break;
320     }
321     case Constraint::nonnegativity:
322     {
323         v2::reduce_scatter( mv.layer_comm[idx],
324                             mv.layer_mttkrp_T[idx],
325                             mv.send_recv_counts[idx][0],
326                             mv.local_mttkrp_T[idx] );
327
328         mv.local_mttkrp[idx] = mv.local_mttkrp_T[idx].transpose();
329         mv.nesterov_ol_d_layer_factor = mv.layer_factors[idx];
330         if (mv.rows_for_update[idx] != 0)
331         {
332             NesterovMnls(mv.cwise_factor_product, mv.local_mttkrp[idx], st.options.nesterov_delta_1,
333                          st.options.nesterov_delta_2, mv.local_factors[idx]);
334         }
335         break;
336     }
337     case Constraint::orthogonality:
338     {
339         all_reduce( mv.layer_comm[idx],
340                    inplace(mv.layer_mttkrp[idx].data()),
341                    mv.subTnsDims_R[idx][mv.fiber_rank[idx]],
342                    std::plus<double>() );
343
344         if (mv.rows_for_update[idx] != 0)
345         {
346             mv.local_mttkrp[idx] =
mv.layer_mttkrp[idx].block(mv.dspl_s_local_update[idx][mv.layer_rank[i]] / static_cast<int>(R), 0,
347                             mv.rows_for_update[idx],
348                             static_cast<int>(R));
349             mv.temp_matrix.noalias() = mv.layer_mttkrp[idx].transpose() * mv.layer_mttkrp[idx];
350         }
351         all_reduce( mv.fiber_comm[idx],
352                    inplace(mv.temp_matrix.data()),
353                    mv.RxR,
354                    std::plus<double>() );
355     }
356 }

```

```

355         Eigen::SelfAdjointEigenSolver<Matrix> ei_gensolver(mv.temp_matrix);
356         mv.temp_matrix.noalias() = (ei_gensolver.eigenvalues())
357         *
(eigenvalues().cwiseInverse().cwiseSqrt().asDiagonal())
        * (ei_gensolver.eigenvalues().transpose());
358
359         if(mv.rows_for_update[idx] != 0)
360             mv.local_factors[idx].noalias() = mv.local_mttkrp[idx] * mv.temp_matrix;
361             break;
362         }
363     case Constraint::sparsity:
364         break;
365     default: // in case of Constraint::constant
366         break;
367 } // end of constraints switch
368
369 if(st.options.constraints[idx] != Constraint::constant)
370 {
371     mv.local_factors_T[idx] = mv.local_factors[idx].transpose();
372     v2::gatherv( mv.layer_comm[idx],
373                 mv.local_factors_T[idx],
374                 mv.send_recv_counts[idx][mv.layer_rank[idx]],
375                 mv.send_recv_counts[idx][0],
376                 mv.dispatch_local_update[idx][0],
377                 mv.layer_factors_T[idx] );
378
379     mv.layer_factors[idx] = mv.layer_factors_T[idx].transpose();
380     mv.factor_T_factor[idx].noalias() = mv.layer_factors_T[idx] * mv.layer_factors[idx];
381 }
382
383 all_reduce( mv.fiber_comm[idx],
384             inplace(mv.factor_T_factor[idx].data()),
385             mv.RxR,
386             std::plus<double>() );
387
388 if(st.options.constraints[idx] == Constraint::nonnegativity)
389 {
390     if((mv.factor_T_factor[idx].diagonal()).minCoeff()==0)
391     {
392         mv.layer_factors[idx] = 0.9 * mv.layer_factors[idx] + 0.1 * mv.nesterov_ol_d_layer_factor;
393         all_reduce( mv.fiber_comm[idx],
394                     inplace(mv.factor_T_factor[idx].data()),
395                     mv.RxR,
396                     std::plus<double>() );
397     }
398 }
399 }
400 }
401
402 /*
403 * At the end of the algorithm processor 0
404 * collects each part of the factor that each
405 * processor holds and return them in status.factors.
406 *
407 * @tparam Dimensions      Array type containing the Tensor dimensions.
408 *
409 * @param tnsDims [in]    Tensor Dimensions. Each index contains the corresponding
410 *                        factor's rows length.
411 * @param R [in]         The rank of decomposition.
412 * @param mv [in]        Struct where ALS variables are stored.
413 *                        Use variables to compute result factors by gathering each
414 *                        part of the factor from processors.
415 * @param st [in,out]    Struct where the returned values of @c Cpd are stored.
416 *                        Stores the result factors.
417 */
418 template<typename Dimensions>
419 void gather_final_factors(Dimensions const &tnsDims,
420                          std::size_t const R,
421                          MemberVariables &mv,
422                          Status &st)
423 {
424     for(std::size_t i=0; i<TnsSize; ++i)
425         mv.layer_factors_T[i] = mv.layer_factors[i].transpose();
426
427     for(std::size_t i=0; i<TnsSize; ++i)
428     {
429         mv.temp_matrix.resize(static_cast<int>(R), tnsDims[i]);
430         // Gather from all processors to processor with rank 0 the final factors
431         v2::gatherv( mv.fiber_comm[i],
432                     mv.layer_factors_T[i],
433                     mv.subTnsDims_R[i][mv.fiber_rank[i]],
434                     mv.subTnsDims_R[i][0],
435                     mv.dispatch_subTns_R[i][0],
436                     0,
437                     mv.temp_matrix );
438
439         st.factors[i] = mv.temp_matrix.transpose();
440     }

```

```

441     }
442
443     /*
444     * @brief Line Search Acceleration
445     *
446     * Performs an acceleration step in the updated factors, and keeps the accelerated factors when
447     * the step succeeds. Otherwise, the acceleration step is ignored.
448     * Line Search Acceleration reduces the number outer iterations in the ALS algorithm.
449     *
450     * @note This implementation ONLY, if factors are of @c Matrix type.
451     *
452     * @param grid_comm [in] MPI communicator where the new cost function value
453     *                       will be communicated and computed.
454     * @param mv         [in, out] Struct where ALS variables are stored.
455     *                       In case the acceleration is successful layer factor^T * factor
456     *                       and layer factor variables are updated.
457     * @param st         [in, out] Struct where the returned values of @c Cpd are stored.
458     *                       If the acceleration succeeds updates cost function value.
459     *
460     */
461     void line_search_accel (CartCommunicator const &grid_comm,
462                           Member_Variables &mv,
463                           Status &st)
464     {
465         double f_accel = 0.0; // Objective Value after the acceleration step
466         double const accel_step = pow(st.ao_iter+1, (1.0/(st.options.accel_coeff)));
467
468         MatrixArray accel_factors;
469         MatrixArray accel_gramians;
470
471         for(std::size_t i=0; i<TnsSize; ++i)
472         {
473             accel_factors[i] = mv.old_factors[i] + accel_step * (mv.layer_factors[i] -
474             mv.old_factors[i]);
475             accel_gramians[i] = accel_factors[i].transpose() * accel_factors[i];
476             all_reduce( mv.fiber_comm[i],
477                       i nplace(accel_gramians[i].data()),
478                       mv.RxR,
479                       std::plus<double>() );
480         }
481
482         f_accel = accel_cost_function(grid_comm, mv, st, accel_factors, accel_gramians);
483         if (st.f_value > f_accel)
484         {
485             mv.layer_factors = accel_factors;
486             mv.factor_T_factor = accel_gramians;
487             st.f_value = f_accel;
488             if(grid_comm.rank() == 0)
489                 Partensor()->Logger()->info("Acceleration Step SUCCEEDED! at iter: {}", st.ao_iter);
490         }
491         else
492             st.options.accel_fail++;
493
494         if (st.options.accel_fail==5)
495         {
496             st.options.accel_fail=0;
497             st.options.accel_coeff++;
498         }
499     }
500     /*
501     * Parallel implementation of als method with MPI.
502     *
503     * @tparam Dimensions Array type containing the Tensor dimensions.
504     *
505     * @param grid_comm [in] The communication grid, where the processors
506     *                       communicate their cost function.
507     * @param tnsDims [in] Tensor Dimensions. Each index contains the corresponding
508     *                       factor's rows length.
509     * @param R [in] The rank of decomposition.
510     * @param mv [in] Struct where ALS variables are stored and being updated
511     *               until a termination condition is true.
512     * @param status [in, out] Struct where the returned values of @c Cpd are stored.
513     */
514     template<typename Dimensions>
515     void als(CartCommunicator const &grid_comm,
516             Dimensions const &tnsDims,
517             std::size_t const R,
518             Member_Variables &mv,
519             Status &status)
520     {
521         for (std::size_t i = 0; i < TnsSize; ++i)
522         {
523             mv.proc_krao[i] = PartialKhatriRao(mv.layer_factors, i);
524             mv.subTns_mat[i] = Matricization(mv.subTns, i);
525
526             mv.factor_T_factor[i].noalias() = mv.layer_factors[i].transpose() * mv.layer_factors[i];

```

```

527     all_reduce( mv.fiber_comm[i],
528               inplace(mv.factor_T_factor[i].data()),
529               mv.RxR,
530               std::plus<double>() );
531
532     mv.local_mttkrp_T[i].resize(R, mv.rows_for_update[i]);
533     mv.layer_factors_T[i].resize(R, mv.subTnsDims[i][mv.fiber_rank[i]]);
534 }
535
536 mv.tnsX_mat_LastFactor_T = mv.subTns_mat[LastFactor].transpose();
537 if(status.options.normalization)
538 {
539     choose_normalization_factor(status, mv.all_orthogonal, mv.weight_factor);
540 }
541
542 all_reduce( grid_comm,
543           square_norm(mv.subTns),
544           status.frob_tns,
545           std::plus<double>());
546 cost_function_init( grid_comm, mv, status );
547 status.rel_costFunction = status.f_value / sqrt(status.frob_tns);
548
549 // Wait for all processors to reach here
550 grid_comm.barrier();
551
552 // ---- Loop until ALS converges ----
553 while(1)
554 {
555     status.ao_iter++;
556     if (!grid_comm.rank())
557         Partensor()->Logger()->info("iter: {} -- fvalue: {} -- relative_costFunction: {}",
558 status.ao_iter,
559                               status.f_value, status.rel_costFunction);
560
561     for (std::size_t i = 0; i < TnsSize; i++)
562     {
563         mttkrp(mv.layer_factors, mv.subTns_mat[i], i, mv.proc_krao[i], mv.layer_mttkrp[i]);
564         mv.cwise_factor_product = PartialCwiseProd(mv.factor_T_factor, i);
565         mv.layer_mttkrp_T[i] = mv.layer_mttkrp[i].transpose();
566         mv.local_factors[i] =
567 mv.layer_factors[i].block(mv.dsplit_s_local_update[i][mv.layer_rank[i]] / static_cast<int>(R), 0,
568                               mv.rows_for_update[i],
569                               static_cast<int>(R));
570
571         update_factor(i, R, status, mv);
572     }
573
574     // ---- Cost function Computation ----
575     cost_function(grid_comm, mv, status);
576     status.rel_costFunction = status.f_value / sqrt(status.frob_tns);
577     if(status.options.normalization && !mv.all_orthogonal)
578         Normalize(mv.weight_factor, R, mv.factor_T_factor, mv.layer_factors);
579
580     // ---- Termination condition ----
581     if (status.rel_costFunction < status.options.threshold_error || status.ao_iter >=
582 status.options.max_iter)
583     {
584         gather_final_factors(tnsDims, R, mv, status);
585         if(grid_comm.rank() == 0)
586         {
587             Partensor()->Logger()->info("Processor 0 collected all {} factors.m", TnsSize);
588             if(status.options.writeToFile)
589                 writeFactorsToFile(status);
590         }
591         break;
592     }
593
594     if (status.options.acceleration)
595     {
596         mv.norm_factors = mv.layer_factors;
597         // ---- Acceleration Step ----
598         if (status.ao_iter > 1)
599             line_search_accel(grid_comm, mv, status);
600
601         mv.old_factors = mv.norm_factors;
602     }
603 } // end of outer while loop
604 }
605
606 /*
607 * Parallel implementation of als method with MPI.
608 *
609 * @tparam Dimensions      Array type containing the Tensor dimensions.
610 *
611 * @param grid_comm[in]    The communication grid, where the processors
612 *                          communicate their cost function.
613 */

```

```

610     * @param tnsDims [in] Tensor Dimensions. Each index contains the corresponding
611     *                 factor's rows length.
612     * @param R [in] The rank of decomposition.
613     * @param mv [in] Struct where ALS variables are stored and being updated
614     *                 until a termination condition is true.
615     * @param status [in,out] Struct where the returned values of @c Cpd are stored.
616     */
617     template<typename Dimensions>
618     void al_s_true_factors( CartCommunicator const &grid_comm,
619                           Dimensions const &tnsDims,
620                           std::size_t const R,
621                           Member_Variables &mv,
622                           Status &status )
623     {
624         for (std::size_t i = 0; i < TnsSize; ++i)
625         {
626             mv.proc_krao[i] = PartialKhatriRao(mv.layer_factors, i);
627             mv.subTns_mat[i] = generateTensor(i, mv.true_factors);
628
629             mv.factor_T_factor[i].noalias() = mv.layer_factors[i].transpose() * mv.layer_factors[i];
630             all_reduce( mv.fiber_comm[i],
631                       inplace(mv.factor_T_factor[i].data()),
632                       mv.RxR,
633                       std::plus<double>() );
634
635             mv.local_mttkrp_T[i].resize(R, mv.rows_for_update[i]);
636             mv.layer_factors_T[i].resize(R, mv.subTnsDims[i][mv.fiber_rank[i]]);
637         }
638
639         mv.tnsX_mat_lastFactor_T = mv.subTns_mat[lastFactor].transpose();
640         if(status.options.normalization)
641         {
642             choose_normlization_factor(status, mv.all_orthogonal, mv.weight_factor);
643         }
644
645         all_reduce( grid_comm,
646                   (mv.subTns_mat[lastFactor]).squaredNorm(),
647                   status.frob_tns,
648                   std::plus<double>());
649         cost_function_init( grid_comm, mv, status );
650         status.rel_costFunction = status.f_value / sqrt(status.frob_tns);
651
652         // Wait for all processors to reach here
653         grid_comm.barrier();
654
655         // ---- Loop until ALS converges ----
656         while(1)
657         {
658             status.ao_iter++;
659             if (!grid_comm.rank())
660                 Partensor()->Logger()->info("iter: {} -- fvalue: {} -- relative_costFunction: {}",
661 status.ao_iter,
662                                     status.f_value, status.rel_costFunction);
663
664             for (std::size_t i = 0; i < TnsSize; i++)
665             {
666                 mv.proc_krao[i] = PartialKhatriRao(mv.layer_factors, i);
667                 mv.cwise_factor_product = PartialCwiseProd(mv.factor_T_factor, i);
668                 mv.layer_mttkrp_T[i].noalias() = mv.subTns_mat[i] * mv.proc_krao[i];
669                 mv.layer_mttkrp_T[i] = mv.layer_mttkrp_T[i].transpose();
670                 mv.local_factors[i] =
671                 mv.layer_factors[i].block(mv.dsplit_local_update[i][mv.layer_rank[i]] / static_cast<int>(R), 0,
672                                     mv.rows_for_update[i],
673                                     static_cast<int>(R));
674
675                 update_factor(i, R, status, mv);
676                 all_reduce( mv.fiber_comm[i],
677                           inplace(mv.factor_T_factor[i].data()),
678                           mv.RxR,
679                           std::plus<double>() );
680             }
681
682             // ---- Cost function Computation ----
683             cost_function(grid_comm, mv, status);
684             status.rel_costFunction = status.f_value / sqrt(status.frob_tns);
685             if(status.options.normalization && !mv.all_orthogonal)
686                 Normalize(mv.weight_factor, R, mv.factor_T_factor, mv.layer_factors);
687
688             // ---- Termination condition ----
689             if (status.rel_costFunction < status.options.threshold_error || status.ao_iter >=
690 status.options.max_iter)
691             {
692                 gather_final_factors(tnsDims, R, mv, status);
693                 if(grid_comm.rank() == 0)
694                 {
695                     Partensor()->Logger()->info("Processor 0 collected all {} factors.m", TnsSize);
696                     if(status.options.writeToFile)

```

```

693         writeFactorsToFile(status);
694     }
695     break;
696 }
697
698     if (status.options.acceleration)
699     {
700         mv.norm_factors = mv.layer_factors;
701         // ---- Acceleration Step ----
702         if (status.ao_iter > 1)
703             line_search_accel(grid_comm, mv, status);
704
705         mv.old_factors = mv.norm_factors;
706     }
707
708 } // end of outer while loop
709 }
710
711 Status operator()(Tensor_ const &tnsX,
712                  std::size_t const R)
713 {
714     Options options = MakeOptions<Tensor_>(execution::openmpi_policy());
715     Status status(options);
716     Member_Variabls mv(R, status.options.proc_per_mode);
717
718     // Communicator with cartesian topology
719     CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);
720
721     // Functions that create layer and fiber grids.
722     create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
723     create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);
724
725     // extract dimensions from tensor
726     Dimensions const &tnsDims = tnsX.dimensions();
727     // produce estimate factors using uniform distribution with entries in [0, 1].
728     makeFactors(tnsDims, status.options.constraints, R, status.factors);
729
730     compute_sub_dimensions(tnsDims, status, R, mv);
731     // Normalize each layer_factor, compute status.frob_tns and status.f_value
732     // Normalize(R, factor_T_factor, status.factors);
733     // After factor normalization scatter to each processor a part of each factor.
734     for (std::size_t i = 0; i < TnsSize; ++i)
735     {
736         mv.layer_factors[i] = status.factors[i].block(mv.dims_subTns[i][mv.fiber_rank[i]], 0,
737                                                       mv.subTnsDims[i][mv.fiber_rank[i]],
738 static_cast<int>(R));
739     }
740
741     // Each processor takes a subtensor from tnsX
742     mv.subTns = tnsX.slice(mv.subTns_offsets, mv.subTns_extents);
743     als(grid_comm, tnsDims, R, mv, status);
744
745     return status;
746 }
747
748 Status operator()(Tensor_ const &tnsX,
749                  std::size_t const R,
750                  Options const &options)
751 {
752     Status status(options);
753     Member_Variabls mv(R, status.options.proc_per_mode);
754
755     // Communicator with cartesian topology
756     CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);
757
758     // Functions that create layer and fiber grids.
759     create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
760     create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);
761
762     // extract dimensions from tensor
763     Dimensions const &tnsDims = tnsX.dimensions();
764     // produce estimate factors using uniform distribution with entries in [0, 1].
765     makeFactors(tnsDims, status.options.constraints, R, status.factors);
766
767     compute_sub_dimensions(tnsDims, status, R, mv);
768     // Normalize each layer_factor, compute status.frob_tns and status.f_value
769     // Normalize(R, factor_T_factor, status.factors);
770     // After factor normalization scatter to each processor a part of each factor.
771     for (std::size_t i = 0; i < TnsSize; ++i)
772     {
773         mv.layer_factors[i] = status.factors[i].block(mv.dims_subTns[i][mv.fiber_rank[i]], 0,
774                                                       mv.subTnsDims[i][mv.fiber_rank[i]],
775 static_cast<int>(R));
776     }
777
778     // Each processor takes a subtensor from tnsX
779     mv.subTns = tnsX.slice(mv.subTns_offsets, mv.subTns_extents);
780

```

```

801
802     switch ( status.options.method )
803     {
804         case Method::als:
805             {
806                 als(grid_comm, tnsDims, R, mv, status);
807                 break;
808             }
809         case Method::rnd:
810             break;
811         case Method::bc:
812             break;
813         default:
814             break;
815     }
816
817     return status;
818 }
819
820 Status operator()(Tensor_ const &tnsX,
821                  std::size_t const R,
822                  MatrixArray const &factorsInit)
823 {
824     Options options = MakeOptions<Tensor_>(execution::openmpi_policy());
825     Status status(options);
826     Member_VariabLes mv(R, status.options.proc_per_mode);
827
828     // Communicator with cartesian topology
829     CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);
830
831     // Functions that create layer and fiber grids.
832     create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
833     create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);
834
835     // extract dimensions from tensor
836     Dimensions const &tnsDims = tnsX.dimensions();
837     status.factors = factorsInit;
838
839     compute_sub_dimensions(tnsDims, status, R, mv);
840     // Normalize each layer_factor, compute status.frob_tns and status.f_value
841     // Normalize(R, factor_T_factor, status.factors);
842     // After factor normalization scatter to each processor a part of each factor.
843     for (std::size_t i = 0; i < TnsSize; ++i)
844     {
845         mv.layer_factors[i] = status.factors[i].block(mv.dims_subTns[i][mv.fiber_rank[i]], 0,
846                                                       mv.subTnsDims[i][mv.fiber_rank[i]],
847                                                       static_cast<int>(R));
848     }
849
850     // Each processor takes a subtensor from tnsX
851     mv.subTns = tnsX.slice(mv.subTns_offsets, mv.subTns_extents);
852
853     als(grid_comm, tnsDims, R, mv, status);
854
855     return status;
856 }
857
858 Status operator()(Tensor_ const &tnsX,
859                  std::size_t const R,
860                  Options const &options,
861                  MatrixArray const &factorsInit)
862 {
863     Status status(options);
864     Member_VariabLes mv(R, status.options.proc_per_mode);
865
866     // Communicator with cartesian topology
867     CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);
868
869     // Functions that create layer and fiber grids.
870     create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
871     create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);
872
873     // extract dimensions from tensor
874     Dimensions const &tnsDims = tnsX.dimensions();
875     status.factors = factorsInit;
876
877     compute_sub_dimensions(tnsDims, status, R, mv);
878     // Normalize each layer_factor, compute status.frob_tns and status.f_value
879     // Normalize(R, factor_T_factor, status.factors);
880     // After factor normalization scatter to each processor a part of each factor.
881     for (std::size_t i = 0; i < TnsSize; ++i)
882     {
883         mv.layer_factors[i] = status.factors[i].block(mv.dims_subTns[i][mv.fiber_rank[i]], 0,
884                                                       mv.subTnsDims[i][mv.fiber_rank[i]],
885                                                       static_cast<int>(R));
886     }
887
888     return status;
889 }

```

```

915 // Each processor takes a subtensor from tnsX
916 mv.subTns.resize(mv.subTns_extents);
917 mv.subTns = tnsX.slice(mv.subTns_offsets, mv.subTns_extents);
918
919 switch ( status.options.method )
920 {
921     case Method::als:
922     {
923         als(grid_comm, tnsDims, R, mv, status);
924         break;
925     }
926     case Method::rnd:
927         break;
928     case Method::bc:
929         break;
930     default:
931         break;
932 }
933
934 return status;
935 }
936
937 template <std::size_t TnsSize>
938 Status operator()(std::array<int, TnsSize> const &tnsDims,
939                 std::size_t const R,
940                 std::string const &path)
941 {
942     Options options = MakeOptions<Tensor_>(execution::openmpi_policy());
943     Status status(options);
944     Member_VariabLes mv(R, status.options.proc_per_mode);
945
946     // Communicator with cartesian topology
947     CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);
948
949     // Functions that create layer and fiber grids.
950     create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
951     create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);
952
953     // produce estimate factors using uniform distribution with entries in [0, 1].
954     makeFactors(tnsDims, status.options.constraints, R, status.factors);
955
956     compute_sub_dimensions(tnsDims, status, R, mv);
957     // Normalize each layer_factor, compute status.frob_tns and status.f_value
958     // Normalize(R, factor_T_factor, status.factors);
959     // After factor normalization scatter to each processor a part of each factor.
960     for (std::size_t i = 0; i < TnsSize; ++i)
961     {
962         mv.layer_factors[i] = status.factors[i].block(mv.dims_subTns[i][mv.fiber_rank[i]], 0,
963             mv.subTnsDims[i][mv.fiber_rank[i]],
964             static_cast<int>(R));
965     }
966
967     // Each processor takes a subtensor from tnsX
968     mv.subTns.resize(mv.subTns_extents);
969     readTensor( path, tnsDims, mv.subTns_extents, mv.subTns_offsets, mv.subTns );
970     als(grid_comm, tnsDims, R, mv, status);
971
972     return status;
973 }
974
975 template <std::size_t TnsSize>
976 Status operator()(std::array<int, TnsSize> const &tnsDims,
977                 std::size_t const R,
978                 std::string const &path,
979                 Options const &options)
980 {
981     Status status(options);
982     Member_VariabLes mv(R, status.options.proc_per_mode);
983
984     // Communicator with cartesian topology
985     CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);
986
987     // Functions that create layer and fiber grids.
988     create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
989     create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);
990
991     // produce estimate factors using uniform distribution with entries in [0, 1].
992     makeFactors(tnsDims, status.options.constraints, R, status.factors);
993
994     compute_sub_dimensions(tnsDims, status, R, mv);
995     // Normalize each layer_factor, compute status.frob_tns and status.f_value
996     // Normalize(R, factor_T_factor, status.factors);
997     // After factor normalization scatter to each processor a part of each factor.
998     for (std::size_t i = 0; i < TnsSize; ++i)
999     {
1000         mv.layer_factors[i] = status.factors[i].block(mv.dims_subTns[i][mv.fiber_rank[i]], 0,
1001             mv.subTnsDims[i][mv.fiber_rank[i]],

```



```

1028     static_cast<int>(R));
1029     }
1030     mv.subTns.resize(mv.subTns_extents);
1031     readTensor( path, tnsDims, mv.subTns_extents, mv.subTns_offsets, mv.subTns );
1032
1033     switch ( status.options.method )
1034     {
1035     case Method::als:
1036     {
1037         als(grid_comm, tnsDims, R, mv, status);
1038         break;
1039     }
1040     case Method::rnd:
1041         break;
1042     case Method::bc:
1043         break;
1044     default:
1045         break;
1046     }
1047
1048     return status;
1049 }
1050
1051 template <std::size_t TnsSize>
1052 Status operator()(std::array<int, TnsSize> const &tnsDims,
1053                 std::size_t const R,
1054                 std::array<std::string, TnsSize+1> const &paths)
1055 {
1056     Options options = MakeOptions<Tensor>(execution::openmpi_policy());
1057     Status status(options);
1058     Member_Variabls mv(R, status.options.proc_per_mode);
1059
1060     // Communicator with cartesian topology
1061     CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);
1062
1063     // Functions that create layer and fiber grids.
1064     create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
1065     create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);
1066
1067     compute_sub_dimensions(tnsDims, status, R, mv);
1068     // Read initialized factors from files
1069     for (std::size_t i = 0; i < TnsSize; ++i)
1070     {
1071         mv.layer_factors[i] = Matrix(mv.subTnsDims[i][mv.fiber_rank[i]], static_cast<int>(R));
1072
1073         read( paths[i+1],
1074             mv.subTnsDims[i][mv.fiber_rank[i]]*static_cast<int>(R),
1075             mv.dims_subTns_R[i][mv.fiber_rank[i]],
1076             mv.layer_factors[i] );
1077     }
1078
1079     // Normalize each layer_factor, compute status.frob_tns and status.f_value
1080     // Normalize(R, factor_T_factor, status.factors);
1081     // Each processor takes a subtensor from tnsX
1082     mv.subTns.resize(mv.subTns_extents);
1083     readTensor( paths[0], tnsDims, mv.subTns_extents, mv.subTns_offsets, mv.subTns );
1084     als(grid_comm, tnsDims, R, mv, status);
1085
1086     return status;
1087 }
1088
1089 template <std::size_t TnsSize>
1090 Status operator()(std::array<int, TnsSize> const &tnsDims,
1091                 std::size_t const R,
1092                 std::array<std::string, TnsSize+1> const &paths,
1093                 Options const &options)
1094 {
1095     Status status(options);
1096     Member_Variabls mv(R, status.options.proc_per_mode);
1097
1098     // Communicator with cartesian topology
1099     CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);
1100
1101     // Functions that create layer and fiber grids.
1102     create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
1103     create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);
1104
1105     compute_sub_dimensions(tnsDims, status, R, mv);
1106     // Read initialized factors from files
1107     for (std::size_t i = 0; i < TnsSize; ++i)
1108     {
1109         mv.layer_factors[i] = Matrix(mv.subTnsDims[i][mv.fiber_rank[i]], static_cast<int>(R));
1110
1111         read( paths[i+1],
1112             mv.subTnsDims[i][mv.fiber_rank[i]]*static_cast<int>(R),
1113             mv.dims_subTns_R[i][mv.fiber_rank[i]],
1114

```

```

1145         mv.layer_factors[i ] );
1146     }
1147
1148     // Normalize each layer_factor, compute status.frob_tns and status.f_value
1149     // Normalize(R, factor_T_factor, status.factors);
1150     // Each processor takes a subtensor from tnsX
1151     mv.subTns.resize(mv.subTns_extents);
1152     readTensor( paths[0], tnsDims, mv.subTns_extents, mv.subTns_offsets, mv.subTns );
1153
1154     switch ( status.options.method )
1155     {
1156     case Method::als:
1157     {
1158         als(grid_comm, tnsDims, R, mv, status);
1159         break;
1160     }
1161     case Method::rnd:
1162     {
1163         break;
1164     }
1165     case Method::bc:
1166     {
1167         break;
1168     }
1169     default:
1170     {
1171         break;
1172     }
1173     }
1174
1175     return status;
1176 }
1177
1178 template <std::size_t TnsSize>
1179 Status operator()(std::array<int, TnsSize> const &tnsDims,
1180                 std::size_t const R,
1181                 std::array<std::string, TnsSize> const &>true_paths,
1182                 std::array<std::string, TnsSize> const &init_paths,
1183                 const Options
1184 {
1185     Status status(options);
1186     MemberVariables mv(R, status.options.proc_per_mode);
1187
1188     // Communicator with cartesian topology
1189     CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);
1190
1191     // Functions that create layer and fiber grids.
1192     create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
1193     create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);
1194
1195     compute_sub_dimensions(tnsDims, status, R, mv);
1196     // Read initialized factors from files
1197     for (std::size_t i = 0; i < TnsSize; ++i)
1198     {
1199         mv.layer_factors[i] = Matrix(mv.subTnsDims[i][mv.fiber_rank[i]], static_cast<int>(R));
1200         mv.true_factors[i] = Matrix(mv.subTnsDims[i][mv.fiber_rank[i]], static_cast<int>(R));
1201
1202         read( init_paths[i],
1203             mv.subTnsDims[i][mv.fiber_rank[i]]*static_cast<int>(R),
1204             mv.dims_subTns_R[i][mv.fiber_rank[i]],
1205             mv.layer_factors[i] );
1206
1207         read( true_paths[i],
1208             mv.subTnsDims[i][mv.fiber_rank[i]]*static_cast<int>(R),
1209             mv.dims_subTns_R[i][mv.fiber_rank[i]],
1210             mv.true_factors[i] );
1211     }
1212
1213     // Normalize each layer_factor, compute status.frob_tns and status.f_value
1214     // Normalize(R, factor_T_factor, status.factors);
1215     switch ( status.options.method )
1216     {
1217     case Method::als:
1218     {
1219         als_true_factors(grid_comm, tnsDims, R, mv, status);
1220         break;
1221     }
1222     case Method::rnd:
1223     {
1224         break;
1225     }
1226     case Method::bc:
1227     {
1228         break;
1229     }
1230     default:
1231     {
1232         break;
1233     }
1234     }
1235
1236     return status;
1237 }
1238
1239 };
1240 } // namespace internal
1241 } // namespace v1
1242
1243 } // 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.](#)

```

1  #ifndef DOXYGEN_SHOULD_SKIP_THIS
15 #endif // DOXYGEN_SHOULD_SKIP_THIS
16 /*****
27 #ifndef PARTENSOR_CPD_HPP
28 #error "CPD_OPENMP can only be included inside CPD"
29 #endif /* PARTENSOR_CPD_HPP */
30
31 namespace partensor {
32
33     inline namespace v1 {
34
35         namespace internal {
41             template <typename Tensor_>
42             struct CPD<Tensor_, execution::openmp_policy> : public CPD_Base<Tensor_>
43             {
44                 using CPD_Base<Tensor_>::TnsSize;
45                 using CPD_Base<Tensor_>::LastFactor;
46                 using typename CPD_Base<Tensor_>::Dimensions;
47                 using typename CPD_Base<Tensor_>::MatrixArray;
48                 using typename CPD_Base<Tensor_>::DataType;
49
50                 using Options = partensor::Options<Tensor_, execution::openmp_policy, DefaultValues>;
51                 using Status = partensor::Status<Tensor_, execution::openmp_policy, DefaultValues>;
52
53                 // Variables that will be used in cpd implementations.
54                 struct Member_Variables {
55                     MatrixArray krao;
56                     MatrixArray factor_T_factor;
57                     MatrixArray mttkrp;
58                     MatrixArray tns_mat;
59                     MatrixArray norm_factors;
60                     MatrixArray old_factors;
61                     MatrixArray true_factors;
62
63                     Matrix cwise_factor_product;
64                     Matrix temp_matrix;
65
66                     Tensor_ tnsX;
67
68                     Dimensions tnsDims;
69
70                     bool all_orthogonal = true;
71                     int weight_factor;
72
73                     Member_Variables() = default;
74                     Member_Variables(Member_Variables const &) = default;
75                     Member_Variables(Member_Variables &&) = default;
76
77                     Member_Variables &operator=(Member_Variables const &) = default;
78                     Member_Variables &operator=(Member_Variables &&) = default;
79                 };
80
81                 /*
82                 * In case option variable @c writeToFile is enabled, then, before the end
83                 * of the algorithm, it writes the resulted factors in files, whose
84                 * paths are specified before compiling in @ options.final_factors_path.
85                 */

```

```

86     * @param st [in] Struct where the returned values of @c Cpd are stored.
87     */
88     void writeFactorsToFile(Status const &st)
89     {
90         std::size_t size;
91         for(std::size_t i=0; i<TnsSize; ++i)
92         {
93             size = st.factors[i].rows() * st.factors[i].cols();
94             partensor::write(st.factors[i],
95                             st.options.final_factors_paths[i],
96                             size);
97         }
98     }
99
100     /*
101     * Compute the cost function value based on the initial factors.
102     *
103     * @param mv [in] Struct where ALS variables are stored.
104     * @param st [in,out] Struct where the returned values of @c Cpd are stored.
105     * In this case the cost function value is updated.
106     */
107     void cost_function_init(Member_Variables const &mv,
108                             Status &st)
109     {
110         st.f_value = sqrt( ( mv.tns_mat[lastFactor] - st.factors[lastFactor] *
Partial Khatri Rao(st.factors, lastFactor).transpose() ).squaredNorm() );
111     }
112
113     /*
114     * Compute the cost function value at the end of each outer iteration
115     * based on the last factor.
116     *
117     * @param mv [in] Struct where ALS variables are stored.
118     * @param st [in,out] Struct where the returned values of @c Cpd are stored.
119     * In this case the cost function value is updated.
120     */
121     void cost_function(Member_Variables const &mv,
122                         Status &st)
123     {
124         st.f_value = sqrt( ( mv.tns_mat[lastFactor] - st.factors[lastFactor] *
mv.krao[lastFactor].transpose() ).squaredNorm() );
125     }
126
127     /*
128     * Compute the cost function value at the end of each outer iteration
129     * based on the last accelerated factor.
130     *
131     * @param mv [in] Struct where ALS variables are stored.
132     * @param st [in] Struct where the returned values of @c Cpd are stored.
133     * @param factors [in] Accelerated factors.
134     * @param factors_T_factors [in] Gramian matrices of factors.
135     *
136     * @returns The cost function calculated with the accelerated factors.
137     */
138     double accel_cost_function(Member_Variables const &mv,
139                               Status const &st,
140                               MatrixArray const &factors,
141                               MatrixArray const &factors_T_factors)
142     {
143         return sqrt( st.frob_tns + (Partial CwiseProd(factors_T_factors,
lastFactor).cwiseProduct(factors_T_factors[lastFactor])).sum()
- 2 * (mv.mttkrp[lastFactor].cwiseProduct(factors[lastFactor])).sum() );
144     }
145
146     void cost_function2(Member_Variables const &mv,
147                       Status &st)
148     {
149         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() );
150     }
151
152
153     /*
154     * Based on each factor's constraint, a different
155     * update function is used at every outer iteration.
156     *
157     *
158     * Computes also factor^T * factor at the end.
159     *
160     * @param idx [in] Factor to be updated.
161     * @param mv [in] Struct where ALS variables are stored.
162     * @param st [in,out] Struct where the returned values of @c Cpd are stored.
163     * Updates the @c stl array with the factors.
164     */
165     void update_factor(int const idx,
166                       Member_Variables &mv,
167                       Status &st )
168     {

```

```

169         // Update factor
170         switch ( st.options.constraints[idx] )
171         {
172             case Constraint::unconstrained:
173             {
174                 st.factors[idx].noalias() = mv.mttkrp[idx] * mv.cwise_factor_product.inverse();
175                 break;
176             }
177             case Constraint::nonnegativity:
178             {
179                 mv.temp_matrix = st.factors[idx];
180                 NesterovMNLs(mv.cwise_factor_product, mv.mttkrp[idx],
181 st.options.nesterov_delta_1,
182 st.options.nesterov_delta_2, st.factors[idx]);
183                 if(st.factors[idx].cwiseAbs().colwise().sum().minCoeff() == 0)
184                     st.factors[idx] = 0.9 * st.factors[idx] + 0.1 * mv.temp_matrix;
185                 break;
186             }
187             case Constraint::orthogonality:
188             {
189                 mv.temp_matrix = mv.mttkrp[idx].transpose() * mv.mttkrp[idx];
190                 Eigen::SelfAdjointEigenSolver<Matrix> eigensolver(mv.temp_matrix);
191                 mv.temp_matrix.noalias() = (eigensolver.eigenvalues()
192 (eigensolver.eigenvalues().cwiseInverse().cwiseSqrt()).asDiagonal())
193                 * (eigensolver.eigenvalues().transpose());
194                 st.factors[idx].noalias() = mv.mttkrp[idx] * mv.temp_matrix;
195                 break;
196             }
197             case Constraint::sparsity:
198                 break;
199             default: // in case of Constraint::constant
200                 break;
201         }
202         // Compute A^T * A + B^T * B + ...
203         mv.factor_T_factor[idx].noalias() = st.factors[idx].transpose() * st.factors[idx];
204     }
205
206     /*
207     * @brief Line Search Acceleration
208     *
209     * Performs an acceleration step on the updated factors, and keeps the accelerated factors
210     * when the step succeeds. Otherwise, the acceleration step is ignored.
211     * Line Search Acceleration reduces the number of outer iterations in the ALS algorithm.
212     *
213     * @note This implementation ONLY, if factors are of @c Matrix type.
214     *
215     * @param mv [in,out] Struct where ALS variables are stored.
216     *             In case the acceleration step is successful the Gramian
217     *             matrices of factors are updated.
218     * @param st [in,out] Struct where the returned values of @c Cpd are stored.
219     *             If the acceleration succeeds updates @c factors
220     *             and cost function value.
221     */
222     void line_search_accel (MemberVariables &mv,
223                          Status &st)
224     {
225         double f_accel = 0.0; // Objective Value after the acceleration step
226         double const accel_step = pow(st.ao_iter+1, (1.0/(st.options.accel_coeff)));
227
228         MatrixArray accel_factors;
229         MatrixArray accel_gramians;
230
231         for(std::size_t i=0; i<TnsSize; ++i)
232         {
233             accel_factors[i] = mv.old_factors[i] + accel_step * (st.factors[i] -
234 mv.old_factors[i]);
235             accel_gramians[i] = accel_factors[i].transpose() * accel_factors[i];
236         }
237
238         mttkrp(mv.tnsDims, accel_factors, mv.tns_mat[lastFactor], lastFactor, get_num_threads(),
239 mv.mttkrp[lastFactor]);
240         f_accel = accel_cost_function(mv, st, accel_factors, accel_gramians);
241         if (st.f_value > f_accel)
242         {
243             st.factors = accel_factors;
244             mv.factor_T_factor = accel_gramians;
245             st.f_value = f_accel;
246             Partensor()->Logger()->info("Acceleration Step SUCCEEDED! at iter: {}", st.ao_iter);
247         }
248         else
249             st.options.accel_fail++;
250
251         if (st.options.accel_fail==5)
252     
```

```

252         st.opti ons.accel _fai l =0;
253         st.opti ons.accel _coeff++;
254     }
255 }
256
257 /*
258 * Sequential implementation of Alternating Least Squares (ALS) method,
259 * using Shared Memory and OpenMP.
260 *
261 * @param R [in] The rank of decomposition.
262 * @param mv [in] Struct where ALS variables are stored and being updated
263 * until a termination condition is true.
264 * @param st [in,out] Struct where the returned values of @c Cpd are stored.
265 */
266 void als(std::size_t const R,
267         Member_Variables &mv,
268         Status &status)
269 {
270     for (std::size_t i=0; i<TnsSize; i++)
271     {
272         mv.factor_T_factor[i].noalias() = status.factors[i].transpose() * status.factors[i];
273         mv.tns_mat[i] = Matricizati on(mv.tnsX, i);
274     }
275
276     if(status.opti ons.normal izati on)
277     {
278         choose_normi lizati on_factor(status, mv.all_orthogonal, mv.wei ght_factor);
279     }
280
281     // Normal izati on(stati c_cast<int>(R), mv.factor_T_factor, status.factors);
282     status.frob_tns = square_norm(mv.tnsX);
283     cost_functi on_i ni t(mv, status);
284     status.rel_costFuncti on = status.f_val ue/sqrt(status.frob_tns);
285
286     // ---- Loop until ALS converges ----
287     while (1)
288     {
289         status.ao_i ter++;
290         Partensor()->Logger()->i nfo("i ter: {} -- fval ue: {} -- rel ati ve_costFuncti on: {}");
291
292         status.ao_i ter,
293         status.rel_costFuncti on);
294
295         for (std::size_t i=0; i<TnsSize; i++)
296         {
297             mv.krao[i] = Parti al Khatr i Rao(status.factors, i);
298             mttkrp(mv.tnsDi ms, status.factors, mv.tns_mat[i], i, get_num_threads(),
299             mv.mttkrp[i]);
300             mv.cwi se_factor_product = Parti al Cwi seProd(mv.factor_T_factor, i);
301
302             // Update factor
303             update_factor(i, mv, status);
304
305             // Cost functi on Computati on
306             if(i == lastFactor)
307                 cost_functi on(mv, status);
308         }
309
310         status.rel_costFuncti on = status.f_val ue/sqrt(status.frob_tns);
311         if(status.opti ons.normal izati on && !mv.all_orthogonal)
312             Normal izati on(mv.wei ght_factor, stati c_cast<int>(R), mv.factor_T_factor,
313             status.factors);
314
315         // ---- Termini ng condi ti on ----
316         if (status.rel_costFuncti on < status.opti ons.threshold_error || status.ao_i ter >=
317             status.opti ons.max_i ter)
318         {
319             if(status.opti ons.wri teToFi le)
320                 wri teFactorsToFi le(status);
321             break;
322         }
323
324         if (status.opti ons.accel erati on)
325         {
326             mv.norm_factors = status.factors;
327             // ---- Accelerati on Step ----
328             if (status.ao_i ter > 1)
329                 li ne_search_accel (mv, status);
330
331             mv.ol d_factors = mv.norm_factors;
332         }
333     } // end of while
334 }
335
336 Status operator()(Tensor_ const &tnsX,
337                 std::size_t const R)
338 {

```

```

344     Options      options = MakeOptions<Tensor_, execution::openmp_policy, DefaultValues>();
345     Status       status(options);
346     Member_Variab les mv;
347
348     // extract dimensions from tensor
349     mv.tnsDims = tnsX.dimensions();
350     // produce estimate factors using uniform distribution with entries in [0, 1].
351     makeFactors(mv.tnsDims, status.options.constraints, R, status.factors);
352     mv.tnsX = tnsX;
353     als(R, mv, status);
354
355     return status;
356 }
357
371 Status operator()(Tensor_      const &tnsX,
372                  std::size_t  const R,
373                  Options      const &options)
374 {
375     Status       status(options);
376     Member_Variab les mv;
377
378     // extract dimensions from tensor
379     mv.tnsDims = tnsX.dimensions();
380     // produce estimate factors using uniform distribution with entries in [0, 1].
381     makeFactors(mv.tnsDims, status.options.constraints, R, status.factors);
382     mv.tnsX = tnsX;
383
384     switch ( status.options.method )
385     {
386     case Method::als:
387     {
388         als(R, mv, status);
389         break;
390     }
391     case Method::rnd:
392     {
393         break;
394     }
395     case Method::bc:
396     {
397         break;
398     }
399     default:
400     {
401         break;
402     }
403     }
404     return status;
405 };
406
415 Status operator()(Tensor_      const &tnsX,
416                  std::size_t  const R,
417                  MatrixArray  const &factorsInIt)
418 {
419     Options      options = MakeOptions<Tensor_, execution::openmp_policy, DefaultValues>();
420     Status       status(options);
421     Member_Variab les mv;
422
423     status.factors = factorsInIt;
424     mv.tnsX       = tnsX;
425     // extract dimensions from tensor
426     mv.tnsDims   = tnsX.dimensions();
427     als(R, mv, status);
428
429     return status;
430 }
431
448 Status operator()(Tensor_      const &tnsX,
449                  std::size_t  const R,
450                  Options      const &options,
451                  MatrixArray  const &factorsInIt)
452 {
453     Status status(options);
454     Member_Variab les mv;
455
456     status.factors = factorsInIt;
457     mv.tnsX       = tnsX;
458     // extract dimensions from tensor
459     mv.tnsDims   = tnsX.dimensions();
460
461     switch ( status.options.method )
462     {
463     case Method::als:
464     {
465         als(R, mv, status);
466         break;
467     }
468     case Method::rnd:
469     {
470         break;
471     }
472     case Method::bc:
473     {
474         break;
475     }
476     default:
477     {
478         break;
479     }
480     }

```

```

473         break;
474     }
475
476     return status;
477 }
478
491 Status operator()(std::array<int, TnsSize> const &tnsDims,
492                 std::size_t const R,
493                 std::string const &path)
494 {
495     Options options = MakeOptions<Tensor_, execution::openmp_policy, DefaultValues>();
496     Status status(options);
497     MemberVariables mv;
498
499     long long int fileSize = 1;
500     for(auto &dim : tnsDims)
501         fileSize *= static_cast<long long int>(dim);
502
503     mv.tnsX.resize(tnsDims);
504     // Read the whole Tensor from a file
505     read(path,
506         fileSize,
507         0,
508         mv.tnsX);
509     // produce estimate factors using uniform distribution with entries in [0, 1].
510     makeFactors(tnsDims, status.options.constraints, R, status.factors);
511     // mv.tnsDims = tnsDims;
512     std::copy(tnsDims.begin(), tnsDims.end(), mv.tnsDims.begin());
513     als(R, mv, status);
514
515     return status;
516 }
517
533 Status operator()(std::array<int, TnsSize> const &tnsDims,
534                 std::size_t const R,
535                 std::string const &path,
536                 Options const &options)
537 {
538     Status status(options);
539     MemberVariables mv;
540
541     long long int fileSize = 1;
542     for(auto &dim : tnsDims)
543         fileSize *= static_cast<long long int>(dim);
544
545     mv.tnsX.resize(tnsDims);
546     // Read the whole Tensor from a file
547     read(path,
548         fileSize,
549         0,
550         mv.tnsX);
551     // produce estimate factors using uniform distribution with entries in [0, 1].
552     makeFactors(tnsDims, status.options.constraints, R, status.factors);
553     std::copy(tnsDims.begin(), tnsDims.end(), mv.tnsDims.begin());
554
555     switch (status.options.method)
556     {
557     case Method::als:
558     {
559         als(R, mv, status);
560         break;
561     }
562     case Method::rnd:
563         break;
564     case Method::bc:
565         break;
566     default:
567         break;
568     }
569
570     return status;
571 }
572
587 Status operator()(std::array<int, TnsSize> const &tnsDims,
588                 std::size_t const R,
589                 std::array<std::string, TnsSize+1> const &paths)
590 {
591     Options options = MakeOptions<Tensor_, execution::openmp_policy, DefaultValues>();
592     Status status(options);
593     MemberVariables mv;
594
595     long long int fileSize = 1;
596     for(auto &dim : tnsDims)
597         fileSize *= static_cast<long long int>(dim);
598
599     mv.tnsX.resize(tnsDims);
600     // Read the whole Tensor from a file

```



```

601         read( paths.front(),
602              fileSi ze,
603              0,
604              mv. tnsX );
605
606         // Read initialized factors from files
607         for(std::size_t i=0; i<TnsSi ze; ++i)
608         {
609             status. factors[i] = Matrix(tnsDi ms[i], stati c_cast<i nt>(R));
610             read( paths[i+1],
611                  tnsDi ms[i]*R,
612                  0,
613                  status. factors[i] );
614         }
615         std:: copy(tnsDi ms. begi n(), tnsDi ms. end(), mv. tnsDi ms. begi n());
616
617         als(R, mv, status);
618
619         return status;
620     }
621
622     Status operator()(std::array<i nt, TnsSi ze> const &tnsDi ms,
623                     std::size_t const R,
624                     std::array<std::stri ng, TnsSi ze+1> const &paths,
625                     Options const &opti ons)
626     {
627         Status status(opti ons);
628         Member_Vari ables mv;
629
630         long long i nt fileSi ze = 1;
631         for(auto &di m : tnsDi ms)
632             fileSi ze *= stati c_cast<long long i nt>(di m);
633
634         mv. tnsX. resi ze(tnsDi ms);
635         // Read the whole Tensor from a file
636         read( paths.front(),
637              fileSi ze,
638              0,
639              mv. tnsX );
640
641         // Read initialized factors from files
642         for(std::size_t i=0; i<TnsSi ze; ++i)
643         {
644             status. factors[i] = Matrix(tnsDi ms[i], stati c_cast<i nt>(R));
645             read( paths[i+1],
646                  tnsDi ms[i]*R,
647                  0,
648                  status. factors[i] );
649         }
650         std:: copy(tnsDi ms. begi n(), tnsDi ms. end(), mv. tnsDi ms. begi n());
651
652         swi tch ( status. opti ons. method )
653         {
654             case Method:: als:
655             {
656                 als(R, mv, status);
657                 break;
658             }
659             case Method:: rnd:
660             {
661                 break;
662             }
663             case Method:: bc:
664             {
665                 break;
666             }
667             default:
668             {
669                 break;
670             }
671         }
672
673         return status;
674     }
675
676     Status operator()(std::array<i nt, TnsSi ze> const &tnsDi ms,
677                     std::size_t const R,
678                     std::array<std::stri ng, TnsSi ze> const &true_paths,
679                     std::array<std::stri ng, TnsSi ze> const &i ni t_paths,
680                     Options const &opti ons)
681     {
682         Status status(opti ons);
683         Member_Vari ables mv;
684
685         for(std::size_t i=0; i<TnsSi ze; ++i)
686         {
687             mv. true_factors[i] = Matrix(tnsDi ms[i], stati c_cast<i nt>(R));
688             status. factors[i] = Matrix(tnsDi ms[i], stati c_cast<i nt>(R));
689
690             // Read initialized factors from files
691             read( true_paths[i],
692                  tnsDi ms[i]*R,
693                  0,

```

```

731         mv.true_factors[i] );
732     // Read initialized factors from files
733     read( init_paths[i],
734         tnsDims[i]*R,
735         0,
736         status.factors[i] );
737     }
738     std::copy(tnsDims.begin(), tnsDims.end(), mv.tnsDims.begin());
739
740     switch ( status.options.method )
741     {
742     case Method::als:
743     {
744         als_true_factors(R, mv, status);
745         break;
746     }
747     case Method::rnd:
748     break;
749     case Method::bc:
750     break;
751     default:
752     break;
753     }
754
755     return status;
756 }
757 };
758 } // end namespace internal
759 } // end namespace v1
760 } // end namespace partensor
761

```

8.15 CwiseProd.hpp File Reference

```
#include "PARTENSOR_basics.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

<i>Matrices</i>	A template parameter pack (<code>std::variadic</code>) type, with possible multiple <i>Matrices</i> .
-----------------	---

Parameters

<i>mat1</i>	[in] A partensor: : <i>Matrix</i> .
<i>mat2</i>	[in] A partensor: : <i>Matrix</i> .
<i>mats</i>	[in] Possible 0 or more A partensor <i>Matrices</i> .

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.](#)

```

1 #ifndef DOXYGEN_SHOULD_SKIP_THIS
15 #endif // DOXYGEN_SHOULD_SKIP_THIS
16 /*****
24 #ifndef PARTENSOR_CWISE_PROD_HPP
25 #define PARTENSOR_CWISE_PROD_HPP
26
27 #include "PARTENSOR_basics.hpp"
28
29 namespace partensor {
30
31     inline namespace v1 {
48         template<typename... Matrices>
49         Matrix CwiseProd( Matrix const &mat1,
50                          Matrix const &mat2,
51                          Matrices const &... mats )
52         {
53             if constexpr( sizeof... (mats) == 0 )
54             {
55                 return mat1.cwiseProduct(mat2);
56             }
57             else
58             {
59                 auto _temp = CwiseProd(mat2, mats...);
60                 return CwiseProd(mat1, _temp);
61             }
62         }
63     }
64 } // end namespace v1
65 } // end namespace partensor
68
69 #endif // PARTENSOR_CWISE_PROD_HPP

```

8.17 DataGeneration.hpp File Reference

```

#include "PARTENSOR_basics.hpp"
#include "unsupported/Eigen/CXX11/src/Tensor/TensorRandom.h"
#include "PartialKhatRao.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 matrixed 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

<i>mtx</i>	[in,out] The Matrix, where the data will be stored.
------------	---

Note

mtx must initialized 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

<i>Tensor_</i>	Type(data type and order) of input Tensor tnsX. Tensor_ must be Tensor<order>, where order must be in range of [3-8].
----------------	---

Parameters

<i>tnsX</i>	[in,out] The given Tensor to be filled with data, based on <i>distribution</i> .
<i>distribution</i>	[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

<i>_TnsSize</i>	Tensor Order of the Tensor.
-----------------	-----------------------------

Parameters

<i>factorArray</i>	[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 `std::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

<i>MatrixArray</i>	An array container with Matrices of <code>Matrix</code> type.
--------------------	---

Parameters

<i>mode</i>	[in] The dimension, in which the matricized Tensor will be returned.
<i>factorArray</i>	[in] The array with the factors of <code>Matrix</code> 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 `std::array`, where Matrices-Factors are stored.

Creates a pseudo-random `std::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

<i>_TnsSize</i>	Essentially tensor order, but also the size of constraints and factorArray arrays.
<i>Dimensions</i>	Array container for <code>tnsDims</code> .
<i>FactorType</i>	The type for factorArray and the generated Factors, either <code>Eigen Matrix</code> or <code>FactorDimTree</code> .

Parameters

<i>tnsDims</i>	[in] The row dimension for each factor.
<i>constraints</i>	[in] The Constraint to apply to each Factor, check Constants.hpp .
<i>R</i>	[in] Rank of factorization (Number of columns in each Matrix or FactorDimTree).
<i>factorArray</i>	[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

<i>_TnsSize</i>	Size of the tnsDims, constraints arrays and the number of tnsX dimensions.
-----------------	--

Parameters

<i>tnsDims</i>	[in] stl array with each dimension for tnsX.
<i>constraints</i>	[in] The constraints to be applied in on each factor that will be used to generate tnsX.
<i>R</i>	[in] Essentially, the number of columns for each factor.
<i>tnsX</i>	[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.](#)

```
1 #ifndef DOXYGEN_SHOULD_SKIP_THIS
15 #endif // DOXYGEN_SHOULD_SKIP_THIS
16 /*****
24 #ifndef PARTENSOR_DATA_GENERATION_HPP
25 #define PARTENSOR_DATA_GENERATION_HPP
26
27 #include "PARTENSOR_basics.hpp"
28 #include "unsupported/Eigen/CXX11/src/Tensor/TensorRandom.h"
29 #include "PartialKhatririRao.hpp"
30 #include "TensorOperations.hpp"
31 #include "DimTrees.hpp"
32 #include "Constants.hpp"
33 // #include "Normalize.hpp"
34 #include <time.h>
35 #include <cassert>
```

```

36
37 // Use (void) to silent unused warnings.
38 #define assertm(exp, msg) assert(((void)msg, exp))
39
40 namespace partensor
41 {
42
43     void generateRandomMatrix(Matrix &mtx)
44     {
45         int m = mtx.rows();
46         int n = mtx.cols();
47         assertm(m>0 && n>0, "Rows and columns must be greater than 1 in generateRandomMatrixm");
48         std::srand((unsigned int) time(NULL)+std::rand());
49
50         mtx = (Matrix::Random(m, n) + Matrix::Ones(m, n))/2;
51     }
52
53     template<typename Tensor_>
54     void generateRandomTensor(Tensor_ &tnsX,
55                             unsigned const distribution = 0)
56     {
57         using DataType = typename TensorTraits<Tensor_>::DataType;
58         std::srand((unsigned int) time(NULL)+std::rand());
59
60         if(distribution == 0) // uniform distribution with numbers in [0, 1]
61         {
62             tnsX.template setRandom<Eigen::Internal::UniformRandomGenerator<DataType>>();
63         }
64         else if (distribution == 1) // normal distribution with mean = 0 and deviation() = 1
65         {
66             tnsX.template setRandom<Eigen::Internal::NormalRandomGenerator<DataType>>();
67         }
68         else
69         {
70             throw std::runtime_error("Choose correct distribution in
71 generateRandomTensor().m");
72         }
73     }
74
75     template<std::size_t Size>
76     Matrix generateTensor(std::size_t mode,
77                         const array<Matrix, Size> &factorArray)
78     {
79         if (mode > Size-1) { throw std::runtime_error("Mode must be in [0, factorArray.Size) in
80 generateTensor().m"); }
81
82         Matrix partial_krao = PartialKhatRao(factorArray, mode);
83         return (factorArray[mode] * partial_krao.transpose());
84     }
85
86     template<std::size_t _TnsSize>
87     Tensor<static_cast<int>(_TnsSize)> generateTensor(std::array<Tensor<2>, _TnsSize> &factorArray)
88     {
89         static_assert(_TnsSize>0, "Tensor cannot be scalar in generateTensor()!.m");
90
91         using MatrixArray = std::array<Tensor<2>, _TnsSize>;
92         // same for all factors
93         const std::size_t R = factorArray[0].dimension(1);
94         // Initialize Core Tensor for PARAFAC
95         std::array<int, _TnsSize> dim;
96         int i = 0;
97         constexpr int w = 1;
98
99         Tensor<static_cast<int>(_TnsSize)> tnsX;
100        Tensor<static_cast<int>(_TnsSize)> Temp_X;
101
102        std::fill(dim.begin(), dim.end(), R);
103        IdentityTensorGen(dim, tnsX);
104
105        std::array<Eigen::IndexPair<int>, 1> product_dims;
106
107        for(typename MatrixArray::reverse_iterator it=factorArray.rbegin(); it !=
108 factorArray.rend(); ++it)
109        {
110            product_dims = { Eigen::IndexPair<int>(w, i) };
111
112            Temp_X = (*it).contract(tnsX, product_dims);
113            tnsX = Temp_X;
114            i++;
115        }
116        return tnsX;
117    }
118
119    #ifndef DOXYGEN_SHOULD_SKIP_THIS
120    template <std::size_t idx, std::size_t _TnsSize>
121    void fillDimTreeFactors_Matrix(std::array<Matrix, _TnsSize> const &factorArray,

```



```

194                                                                                               std::array<Constraint, _TnsSize>
195     const &constraints,                                                                                               std::array<FactorDi mTree,
196     _TnsSize>      &factorDi mTreeArray )
197     {
198         const Matrix      &_factor      = factorArray[i dx];
199         FactorDi mTree    &_factorDi mTree = factorDi mTreeArray[i dx];
200         std::array<Eigen::IndexPair<int>, 1> product_dims = { Eigen::IndexPair<int>(0, 0) };
201
202         const int rows = _factor.rows();
203         const int cols = _factor.cols();
204
205         _factorDi mTree.factor.resize(rows, cols);
206         _factorDi mTree.graman.resize(cols, cols);
207         _factorDi mTree.constraint = constraints[i dx];
208         _factorDi mTree.factor     = matrixToTensor(_factor, rows, cols);
209         _factorDi mTree.graman     = _factorDi mTree.factor.contract(_factorDi mTree.factor,
product_dims);
210
211         if constexpr (i dx+1 < _TnsSize)
212         {
213             fillDi mTreeFactors_Matrix<i dx + 1, _TnsSize>(factorArray, constraints,
factorDi mTreeArray);
214         }
215     #endif // DOXYGEN_SHOULD_SKIP_THIS
216
217     #ifndef DOXYGEN_SHOULD_SKIP_THIS
218     template <std::size_t i dx, std::size_t _TnsSize>
219     void fillDi mTreeFactors_Tensor( std::array<Tensor<2>, _TnsSize> const &factorArray,
std::array<Constraint, _TnsSize>
220     const &constraints,
221     _TnsSize>      &factorDi mTreeArray )
222     {
223         const Tensor<2>      &_factor      = factorArray[i dx];
224         FactorDi mTree    &_factorDi mTree = factorDi mTreeArray[i dx];
225         std::array<Eigen::IndexPair<int>, 1> product_dims = { Eigen::IndexPair<int>(0, 0) };
226
227         const int rows = _factor.dimension(0); // rows
228         const int cols = _factor.dimension(1); // cols
229
230         _factorDi mTree.factor.resize(rows, cols);
231         _factorDi mTree.graman.resize(cols, cols);
232         _factorDi mTree.constraint = constraints[i dx];
233         _factorDi mTree.factor     = _factor;
234         _factorDi mTree.graman     = _factorDi mTree.factor.contract(_factorDi mTree.factor,
product_dims);
235
236         if constexpr (i dx+1 < _TnsSize)
237         {
238             fillDi mTreeFactors_Tensor<i dx + 1, _TnsSize>(factorArray, constraints,
factorDi mTreeArray);
239         }
240     #endif // DOXYGEN_SHOULD_SKIP_THIS
241
242     #ifndef DOXYGEN_SHOULD_SKIP_THIS
243     template <std::size_t _TnsSize, typename FactorType>
244     void fillDi mTreeFactors( std::array<FactorType, _TnsSize> const &factorArray,
std::array<Constraint, _TnsSize> const &constraints,
std::array<FactorDi mTree, _TnsSize>
245     &factorDi mTreeArray )
246     {
247         constexpr bool check = (std::is_same_v<FactorType, Matrix> ||
std::is_same_v<FactorType, Tensor<2>>);
248         static_assert(check, "Factors must be of type Matrix or Tensor<2>,"
fillDi mTreeFactors()!m");
249
250         if constexpr (std::is_same_v<FactorType, Matrix>)
251             fillDi mTreeFactors_Matrix<0, _TnsSize>(factorArray, constraints,
factorDi mTreeArray);
252         else
253             fillDi mTreeFactors_Tensor<0, _TnsSize>(factorArray, constraints, factorDi mTreeArray);
254     #endif // DOXYGEN_SHOULD_SKIP_THIS
255
256     #ifndef DOXYGEN_SHOULD_SKIP_THIS
257     template<std::size_t _TnsSize>
258     void CpdGen( std::array<FactorDi mTree, _TnsSize>      &FactorArray,
int R, const R,
259                 Tensor<static_cast<int>(_TnsSize)>      &tnsX)
260     {
261         static_assert(_TnsSize>0, "Tensor cannot be scalar in CpdGen()!m");
262         assertm(R>0, "Variable R - factor column must be greater than one in CpdGen()!m");
263     #endif
264

```

```

322     using MatrixArray = std::array<FactorDimTree, _TnsSi ze>;
323     // Initialize Core Tensor for PARAFAC
324     std::array<int, _TnsSi ze> dim;
325     int i = 0;
326     constexpr int w = 1;
327
328     Tensor<static_cast<int>(_TnsSi ze)> Temp_X;
329     std::array<Eigen::IndexPair<int>, 1> product_dims;
330
331     std::fill(dim.begin(), dim.end(), R);
332     IdentityTensorGen(dim, tnsX);
333
334     for(typename MatrixArray::reverse_iterator it=FactorArray.rbegin(); it!=
FactorArray.rend(); ++it)
335     {
336         product_dims = { Eigen::IndexPair<int>(w, i) };
337
338         Temp_X = (it->factor).contract(tnsX, product_dims);
339         tnsX = Temp_X;
340         i++;
341     }
342 }
343 #endif // DOXYGEN_SHOULD_SKIP_THIS
344
345 #ifndef DOXYGEN_SHOULD_SKIP_THIS
360 template<std::size_t idx, std::size_t _TnsSi ze, typename Dimensi ons>
361 void makeFactors_Matrix( Dimensi ons const &tnsDims,
362                         std::array<Constraint, _TnsSi ze> const
&constraints,
363                         int
const R,
364                         std::array<Matrix, _TnsSi ze>
&factorArray )
365 {
366     Matrix &_factor = factorArray[idx];
367     _factor.resize(tnsDims[idx], R);
368     generateRandomMatrix(_factor);
369
370     switch(constraints[idx])
371     {
372     case Constraint::unconstrained:
373     {
374         break;
375     }
376     case Constraint::nonnegativity:
377     {
378         Matrix Zeros = Matrix::Zero(_factor.rows(), _factor.cols());
379         _factor = _factor.cwiseMax(Zeros);
380         break;
381     }
382     case Constraint::orthogonality:
383     {
384         Eigen::JacobiSVD<Matrix> svd(_factor, Eigen::ComputeThinU |
Eigen::ComputeThinV);
385         _factor = svd.matrixU();
386         break;
387     }
388     case Constraint::sparsity:
389     {
390         break;
391     }
392     default: // in case of Constraint::constant
393     {
394         break;
395     }
396     }
397
398     if constexpr (idx+1 < _TnsSi ze)
399     {
400         makeFactors_Matrix<idx + 1, _TnsSi ze>(tnsDims, constraints, R, factorArray);
401     }
402 }
403 #endif // DOXYGEN_SHOULD_SKIP_THIS
404
405 #ifndef DOXYGEN_SHOULD_SKIP_THIS
423 template<std::size_t idx, std::size_t _TnsSi ze, typename Dimensi ons>
424 void makeFactors_Tensor( Dimensi ons const &tnsDims,
425                         std::array<Constraint, _TnsSi ze> const
&constraints,
426                         int
const R,
427                         std::array<Tensor<2>, _TnsSi ze>
&factorArray )
428 {
429     Tensor<2> &_factor = factorArray[idx];
430
431     _factor.resize(tnsDims[idx], R);
432     generateRandomTensor(_factor);

```

```

433
434
435         switch(constraints[idx])
436         {
437             case Constraint::unconstrained:
438                 break;
439         }
440         case Constraint::nonnegativity:
441         {
442             _factor = (_factor.abs()).eval();
443             break;
444         }
445         case Constraint::orthogonality:
446         {
447             Matrix _mtx = tensorToMatrix(_factor, tnsDims[idx], R);
448             Eigen::JacobiSVD<Matrix> svd(_mtx, Eigen::ComputeThinU |
449 Eigen::ComputeThinV);
450             _mtx = svd.matrixU();
451             _factor = matrixToTensor(_mtx, tnsDims[idx], R);
452             break;
453         }
454         case Constraint::sparsity:
455         {
456             break;
457         }
458         default: // in case of Constraint::constant
459         {
460             break;
461         }
462     }
463 }
464
465 if constexpr (idx+1 < _TnsSize)
466 {
467     makeFactors_Tensor<idx + 1, _TnsSize>(tnsDims, constraints, R, factorArray);
468 }
469 }
470 #endif // DOXYGEN_SHOULD_SKIP_THIS
471
472 #ifndef DOXYGEN_SHOULD_SKIP_THIS
473 template<std::size_t idx, std::size_t _TnsSize, typename Dimensions>
474 void makeFactors_DimTree( Dimensions const &tnsDims,
475                          std::array<Constraint, _TnsSize> const
476 &constraints,
477                          int const R,
478                          std::array<FactorDimTree, _TnsSize>
479 &factorArray )
480 {
481     FactorDimTree &_factor = factorArray[idx];
482     std::array<Eigen::IndexPair<int>, 1> product_dims = { Eigen::IndexPair<int>(0, 0) };
483
484     _factor.factor.resize(tnsDims[idx], R);
485     _factor.gramian.resize(R, R);
486     _factor.constraint = constraints[idx];
487
488     generateRandomTensor(_factor.factor);
489
490     switch(constraints[idx])
491     {
492         case Constraint::unconstrained:
493             break;
494         case Constraint::nonnegativity:
495             {
496                 _factor.factor = (_factor.factor.abs()).eval();
497                 break;
498             }
499         case Constraint::orthogonality:
500         {
501             Matrix _mtx = tensorToMatrix(_factor.factor, tnsDims[idx], R);
502             Eigen::JacobiSVD<Matrix> svd(_mtx, Eigen::ComputeThinU |
503 Eigen::ComputeThinV);
504             _mtx = svd.matrixU();
505             _factor.factor = matrixToTensor(_mtx, tnsDims[idx], R);
506             break;
507         }
508         case Constraint::sparsity:
509         {
510             break;
511         }
512         default: // in case of Constraint::constant
513         {
514             break;
515         }
516     }
517 }
518
519 #endif
520
521 #endif
522
523 #endif
524
525 #endif
526
527 #endif
528
529 #endif

```

```

530     }
531   }
532   _factor.grami an = _factor.factor.contract(_factor.factor, product_dims);
533
534   if constexpr (i dx+1 < _TnsSi ze)
535   {
536     makeFactors_Di mTree<i dx + 1, _TnsSi ze>(tnsDi ms, constrai nts, R, factorArray);
537   }
538 }
539 #endif // DOXYGEN_SHOULD_SKIP_THIS
540
541 template<std::si ze_t _TnsSi ze, typename Di mensi ons, typename FactorType>
542 void makeFactors( Di mensi ons const &tnsDi ms,
543                 std::array<Constrai nt, _TnsSi ze> const &constrai nts,
544                 i nt const R,
545                 std::array<FactorType, _TnsSi ze> &factorArray )
546 {
547   constexpr bool check = (std::i s_same_v<FactorType, Matri x> ||
548                          std::i s_same_v<FactorType, Tensor<2>> ||
549                          std::i s_same_v<FactorType, FactorDi mTree>);
550   static_assert(check, "Factors must be of type Matri x, Tensor<2> or FactorDi mTree,
551   makeFactors()!m");
552
553   if constexpr (std::i s_same_v<FactorType, Matri x>)
554     makeFactors_Matri x<0, _TnsSi ze>(tnsDi ms, constrai nts, R, factorArray);
555   else if constexpr (std::i s_same_v<FactorType, Tensor<2>>)
556     makeFactors_Tensor<0, _TnsSi ze>(tnsDi ms, constrai nts, R, factorArray);
557   else
558     makeFactors_Di mTree<0, _TnsSi ze>(tnsDi ms, constrai nts, R, factorArray);
559 }
560
561 template <std::si ze_t _TnsSi ze>
562 void makeTensor( std::array<i nt, _TnsSi ze> const &tnsDi ms,
563                std::array<Constrai nt, _TnsSi ze> const &constrai nts,
564                std::si ze_t const R,
565                Tensor<stati c_cast<i nt>(_TnsSi ze)> &tnsX )
566 {
567   static_assert(_TnsSi ze>0, "Tensor cannot be scalar in makeTensor()!m");
568   assertm(R>0, "Variable R - factor column must be greater than one in makeTensor()!m");
569
570   usi ng Matri xArray = typename TensorTrai ts<Tensor<_TnsSi ze>>::Matri xArray;
571
572   Matri xArray true_factors;
573   Matri xArray grami ans;
574   Matri x matri ci zed_tensor;
575
576   tnsX.resi ze(tnsDi ms);
577   makeFactors(tnsDi ms, constrai nts, R, true_factors);
578   for (std::si ze_t i=0; i<_TnsSi ze; ++i)
579   {
580     grami ans[i].noali as() = true_factors[i].transpose() * true_factors[i];
581   }
582
583   // Normali ze(R, grami ans, true_factors);
584   matri ci zed_tensor = generateTensor(0, true_factors);
585   tnsX = matri xToTensor(matri ci zed_tensor, tnsDi ms);
586 }
587
588 #ifndef DOXYGEN_SHOULD_SKIP_THIS
589 template <std::si ze_t _TnsSi ze>
590 void makeTensor( std::array<i nt, _TnsSi ze> const &tnsDi ms,
591                std::array<Constrai nt, _TnsSi ze> const &constrai nts,
592                std::si ze_t const R,
593                std::array<Matri x, _TnsSi ze> &true_factors,
594                Tensor<stati c_cast<i nt>(_TnsSi ze)> &tnsX)
595 {
596   static_assert(_TnsSi ze>0, "Tensor cannot be scalar in makeTensor()!m");
597   assertm(R>0, "Variable R - factor column must be greater than one in makeTensor()!m");
598
599   usi ng Matri xArray = typename TensorTrai ts<Tensor<_TnsSi ze>>::Matri xArray;
600
601   Matri xArray grami ans;
602   Matri x matri ci zed_tensor;
603
604   tnsX.resi ze(tnsDi ms);
605   makeFactors(tnsDi ms, constrai nts, R, true_factors);
606
607   for (std::si ze_t i=0; i<_TnsSi ze; ++i)
608   {
609     grami ans[i].noali as() = true_factors[i].transpose() * true_factors[i];
610   }
611
612   matri ci zed_tensor = generateTensor(0, true_factors);
613   tnsX = matri xToTensor(matri ci zed_tensor, tnsDi ms);
614 }

```

```

681     }
682     #endif // DOXYGEN_SHOULD_SKIP_THIS
683
684     #ifndef DOXYGEN_SHOULD_SKIP_THIS
685     template<typename Dimensions, typename MatrixArray>
686     void generateFactors( std::size_t const rank,
687                         Dimensions const &tnsDims,
688                         unsigned const distribution,
689                         MatrixArray &factorArray)
690     {
691         using Matrix = typename MatrixArrayTraits<MatrixArray>::value_type;
692
693         constexpr std::size_t TnsSize = MatrixArrayTraits<MatrixArray>::Size;
694
695         std::srand((unsigned int) time(NULL)+std::rand());
696         switch(distribution)
697         {
698             case 0: // uniform distribution with numbers in [-1,1]
699             {
700                 for(std::size_t i=0; i<TnsSize; i++) { factorArray[i] =
MatrixX::Random(tnsDims[i], rank); }
701                 break;
702             }
703             case 1: // uniform distribution with numbers in [0,1]
704             {
705                 for(std::size_t i=0; i<TnsSize; i++) { factorArray[i] =
(MatrixX::Random(tnsDims[i], rank) + MatrixX::Ones(tnsDims[i], rank))/2; }
706                 break;
707             }
708             case 2: // normal distribution with mean = 0 and deviation() = 1
709             {
710                 std::random_device rd;
711                 std::mt19937 e2(rd());
712                 std::normal_distribution<> dist(0.0, 1.0);
713                 for(std::size_t i=0; i<TnsSize; i++)
714                 {
715                     factorArray[i] = Matrix(tnsDims[i], rank);
716                     for(std::size_t j=0; j<static_cast<std::size_t>(tnsDims[i]); j++)
717                     {
718                         for(std::size_t k=0; k<rank; k++)
719                         {
720                             factorArray[i](j,k) = dist(e2);
721                         }
722                     }
723                 }
724                 break;
725             }
726             default:
727             {
728                 break;
729             }
730         }
731     }
732     #endif // DOXYGEN_SHOULD_SKIP_THIS
733
734     #ifndef DOXYGEN_SHOULD_SKIP_THIS
735     template<typename Dimensions, typename Tensor_>
736     void customTensor( Dimensions const &tns_dim,
737                      Tensor_ &tnsX )
738     {
739         static constexpr std::size_t TnsSize = TensorTraits<Tensor_>::TnsSize;
740
741         tnsX.resize(tns_dim);
742         int count = 0;
743
744         if constexpr (TnsSize == 3)
745         {
746             for(int k=0; k<tns_dim[2]; k++)
747             {
748                 for(int j=0; j<tns_dim[1]; j++)
749                 {
750                     for(int i=0; i<tns_dim[0]; i++)
751                     {
752                         tnsX(i,j,k) = count+1;
753                         count++;
754                     }
755                 }
756             }
757         }
758         else if constexpr (TnsSize == 4)
759         {
760             for(int l=0; l<tns_dim[3]; l++)
761             {
762                 for(int k=0; k<tns_dim[2]; k++)
763                 {
764                     for(int j=0; j<tns_dim[1]; j++)
765                     {
766                         for(int i=0; i<tns_dim[0]; i++)
767                         {
768                             tnsX(i,j,k,l) = count+1;
769                             count++;
770                         }
771                     }
772                 }
773             }
774         }
775     }
776     #endif // DOXYGEN_SHOULD_SKIP_THIS

```

```

792         for(int i=0; i<tns_dim[0]; i++)
793         {
794             tnsX(i,j,k,l) = count+1;
795             count++;
796         }
797     }
798 }
799 }
800     }
801     else if constexpr (TnsSize == 5)
802     {
803         for(int m=0; m<tns_dim[4]; m++)
804     {
805         for(int l=0; l<tns_dim[3]; l++)
806         {
807             for(int k=0; k<tns_dim[2]; k++)
808             {
809                 for(int j=0; j<tns_dim[1]; j++)
810                 {
811                     for(int i=0; i<tns_dim[0]; i++)
812                     {
813                         tnsX(i,j,k,l,m) = count+1;
814                         count++;
815                     }
816                 }
817             }
818         }
819     }
820 }
821 #endif // DOXYGEN_SHOULD_SKIP_THIS
822 }
823 }
824 } // end namespace partensor
825
826 #endif // end of PARTENSOR_DATA_GENERATION_HPP

```

8.19 DimTrees.hpp File Reference

```

#include "PARTENSOR_basics.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

<i>key</i>	[in] Searching key value
<i>tree</i>	[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.](#)

```
1 #ifndef DOXYGEN_SHOULD_SKIP_THIS
15 #endif // DOXYGEN_SHOULD_SKIP_THIS
16 /*****
22 #ifndef DIM_TREES_HPP
23 #define DIM_TREES_HPP
24
25 #include "PARTENSOR_basics.hpp"
26 #include "TensorOperations.hpp"
27 #include "Constants.hpp"
28
29 namespace partensor
30 {
31     struct FactorDimTree;
32
33     struct I_TnsNode
34     {
35         using DataType = DefaultDataType;
36         std::size_t TnsSize;
37         I_TnsNode(std::size_t _TnsSize) : TnsSize(_TnsSize)
38         { }
39         virtual I_TnsNode *Parent() = 0;
40         virtual I_TnsNode *Left() = 0;
41         virtual I_TnsNode *Right() = 0;
42
43         virtual bool Updated() = 0;
44         virtual void SetOutdated() = 0;
45
46         virtual int Key() = 0;
47
48         virtual Tensor<2> &Gramian() = 0;
49
50         virtual void *TnsDims() = 0;
51
52         virtual void *LabelSet() = 0;
53
54         virtual void *DeltaSet() = 0;
55
56         virtual void *TensorX() = 0;
57
58         virtual I_TnsNode *SearchKey (int const key ) = 0;
59     };
60 }
```

```

114     virtual void      UpdateTree(int const num_factors, int const id, FactorDimTree      * factors) = 0;
115
116 };
117
118 template <std::size_t _TnsSize>
119 struct TnsNode : I_TnsNode
120 {
121     static constexpr std::size_t TnsSize = _TnsSize;
122     static constexpr bool        IsNull  = false;
123     using DataType               = I_TnsNode::DataType;
124     using Tensor_Type            = Tensor<static_cast<int>(TnsSize)>;
125     using Hessian_Type          = Tensor<2>;
126
127     Tensor_Type      mTnsX;
128     Hessian_Type     mGramian;
129     int              mKey;
130     bool             mUpdated;
131     TnsNode() : I_TnsNode(TnsSize), mKey(0), mUpdated(false)
132     { }
133
134     bool Updated() override
135     {
136         return mUpdated;
137     }
138
139     void SetOutdated() override
140     {
141         if (mUpdated)
142         {
143             if ( !Left() && !Right() )
144             {
145                 mUpdated = false;
146             }
147             else
148             {
149                 Left()->SetOutdated();
150                 Right()->SetOutdated();
151                 mUpdated = false;
152             }
153         }
154     }
155
156     int Key() override
157     {
158         return mKey;
159     }
160
161     Hessian_Type &Gramian() override
162     {
163         return mGramian;
164     }
165
166     void *TensorX() override
167     {
168         return &mTnsX;
169     }
170 };
171
172 template <>
173 struct TnsNode<0> : public I_TnsNode
174 {
175     static constexpr std::size_t TnsSize = 0;
176     static constexpr bool        IsNull  = true;
177     using DataType               = I_TnsNode::DataType;
178     using Tensor_Type            = Tensor<0>;
179     template <typename N>
180     TnsNode(N par = nullptr) : I_TnsNode(TnsSize)
181     { (void) par; }
182
183     I_TnsNode *Parent() override { return nullptr; }
184     I_TnsNode *Left ()  override { return nullptr; }
185     I_TnsNode *Right () override { return nullptr; }
186
187     void SetOutdated() override { throw std::runtime_error("SetOutdated()"); }
188     bool Updated()           override { throw std::runtime_error("Updated()"); }
189     int Key()                 override { throw std::runtime_error("Key()"); }
190     void *TnsDims()           override { throw std::runtime_error("TnsDims()"); }
191     void *LabelSet()           override { throw std::runtime_error("LabelSet()"); }
192     void *DeltaSet()           override { throw std::runtime_error("DeltaSet()"); }
193     Tensor<2> &Gramian()       override { throw std::runtime_error("Gramian()"); }
194     void *TensorX()           override { throw std::runtime_error("TensorX()"); }
195     void UpdateTree(int const, int const, FactorDimTree *) override { throw
196     std::runtime_error("UpdateTree()"); }
197     I_TnsNode *SearchKey (int const ) override { throw
198     std::runtime_error("SearchKey()"); }
199 };

```



```

248
249 using NullTensorType = TnsNode<0>::Tensor_Type;
250
251 template <std::size_t _LabelSetSize, std::size_t _ParLabelSetSize, std::size_t _RootSize>
252 struct ExprNode : public TnsNode<_LabelSetSize == _RootSize ? _LabelSetSize : _LabelSetSize + 1>
253 {
254     using DataType = typename I_TnsNode::DataType;
255
256     //static_assert(LabelSetSize != 0, "Error in expansion!");
257     static constexpr std::size_t LabelSetSize = _LabelSetSize;
258     static constexpr std::size_t ParLabelSetSize = _ParLabelSetSize;
259     static constexpr std::size_t RootSize = _RootSize;
260     static constexpr bool IsRoot = LabelSetSize == RootSize;
261     static constexpr bool IsFirstChild = ParLabelSetSize == RootSize;
262     static constexpr bool IsLeaf = LabelSetSize == 1;
263     static constexpr std::size_t BrotherLabelSetSize = IsRoot ? 0 : ParLabelSetSize - LabelSetSize;
264
265     static constexpr std::size_t TnsSize = LabelSetSize == RootSize ? LabelSetSize :
LabelSetSize + 1;
266     static constexpr std::size_t ParTnsSize = IsRoot ? 0 : (IsFirstChild ? ParLabelSetSize :
ParLabelSetSize+1);
267     static constexpr std::size_t DIM_HALF_SIZE = (1+LabelSetSize)/2;
268
269     static constexpr std::size_t DIM_LEFT_SIZE = IsLeaf ? 0 : DIM_HALF_SIZE;
270
271     static constexpr std::size_t DIM_RIGHT_SIZE = IsLeaf ? 0 : LabelSetSize - DIM_LEFT_SIZE;
272
273     std::array<int, TnsSize> mTnsDims;
274     std::array<int, LabelSetSize> mLabelSet;
275     std::array<int, BrotherLabelSetSize> mDeltaSet;
276
277     using Tns_Node_Type = TnsNode<TnsSize>;
278     using Parent_Tns_Node_Type = TnsNode<ParTnsSize>;
279     using Node_Type = ExprNode<LabelSetSize, ParLabelSetSize, RootSize>;
280     using Left_Node_Type = std::conditional_t<IsLeaf, TnsNode<0>,
ExprNode<DIM_LEFT_SIZE, LabelSetSize, RootSize>>;
281     using Right_Node_Type = std::conditional_t<IsLeaf, TnsNode<0>,
ExprNode<DIM_RIGHT_SIZE, LabelSetSize, RootSize>>;
282
283     // const int mParLabelSetSize;
284     // const int mParParLabelSetSize;
285     // const std::ptrdiff_t mParOffset;
286
287     using Tensor_Type = typename Tns_Node_Type::Tensor_Type;
288     using Hessian_Type = typename Tns_Node_Type::Hessian_Type;
289     using Parent_Tensor_Type = typename Parent_Tns_Node_Type::Tensor_Type;
290
291     using Tns_Node_Type::mTnsX;
292     using Tns_Node_Type::mGramian;
293     using Tns_Node_Type::mKey;
294     using Tns_Node_Type::mUpdated;
295
296     I_TnsNode *parent;
297     Left_Node_Type left;
298     Right_Node_Type right;
299     ExprNode() : TnsNode<TnsSize>(),
300 // mParLabelSetSize(0),
301 // mParParLabelSetSize(0),
302 // mParOffset(0),
303 parent(nullptr),
304 left(this),
305 right(this)
306 {
307     static_assert(IsRoot, "Wrong expansion!");
308 }
309
310 template <std::size_t _ParLabelSetSize2, std::size_t _ParParLabelSetSize, std::size_t _RootSize2>
311 friend struct ExprNode;
312
313 protected:
314 template <std::size_t _ParLabelSetSize2, std::size_t _ParParLabelSetSize, std::size_t _RootSize2>
315 ExprNode(ExprNode<_ParLabelSetSize2, _ParParLabelSetSize, _RootSize2> *parent_) :
316 TnsNode<TnsSize>(),
317 // mParLabelSetSize(_ParLabelSetSize2),
318 // mParParLabelSetSize(_ParParLabelSetSize),
319 // mParOffset(reinterpret_cast<char *>(parent_) -
reinterpret_cast<char *>(this)),
320 parent(parent_),
321 left(this),
322 right(this)
323 {
324     static_assert(_ParLabelSetSize2 == ParLabelSetSize, "Wrong expansion!");
325     static_assert(_RootSize2 == RootSize, "Wrong expansion!");
326     static_assert(_ParLabelSetSize2 != 0, "Wrong expansion!");
327 }
328
329 public:
330
331

```

```

350 void *TnsDims()
351 {
352     return &mTnsDims;
353 }
354
360 void *LabelSet()
361 {
362     return &mLabelSet;
363 }
364
370 void *DeltaSet()
371 {
372     return &mDeltaSet;
373 }
374
383 L_TnsNode *SearchKey (int const aKey)
384 {
385     auto length = LabelSetSize;
386     if (length > 1)
387     {
388         if (aKey <= mKey)
389             return Left()->SearchKey(aKey);
390         else
391             return Right()->SearchKey(aKey);
392     }
393     else
394     {
395         if (aKey == mKey)
396             return this;
397         else
398             return nullptr;
399     }
400 }
401
406 L_TnsNode *Parent() { return IsRoot ? nullptr : parent; }
412 L_TnsNode *Left () { return IsLeaf ? nullptr : &left; }
418 L_TnsNode *Right () { return IsLeaf ? nullptr : &right; }
419
420 /*
421  * Computes the N mode product of a tensor with a matrix.
422  *
423  * @tparam _LabelSetSize      Size of the LabelSet of @c this node.
424  * @tparam _ParLabelSetSize   Size of the LabelSet of the parent node.
425  * @tparam _RootSize         Size of the LabelSet of the root node.
426  * @param  aFactor           [in] Factor, of type @c FactorDimTree, to use
427  *                            for tree mode N product.
428  * @param  aNumFactors [in] Total number of factors.
429  * @param  id             [in] Identification of the updating factor @c aFactor.
430  * @param  aGramian       [in,out] Gramian matrix of the node.
431  * @param  aDeltaSet      [in,out] Label set of the brother node after the N-mode product.
432  * @param  aTnsDims       [in,out] @c stl with the dimensions of the final Tensor.
433  *
434  * @returns A @c TnsNode of @c ParTnsSize is returned.
435  */
436 Parent_Tensor_Type TreeMode_N_Product( FactorDimTree                *const aFactor,
437                                       int                          const aNumFactors,
438                                       int                          const id,
439                                       std::array<int,ParTnsSize>   const &aTnsDims,
440                                       Hessian_Type                &aGramian,
441                                       std::array<int,BrotherLabelSetSize> &aDeltaSet
442                                       );
443
444 /*
445  * Interface of @c TTV product computation, between a tensor and a matrix.
446  * @tparam _LabelSetSize      Size of the LabelSet of @c this node.
447  * @tparam _ParLabelSetSize   Size of the LabelSet of the parent node.
448  * @tparam _RootSize         Size of the LabelSet of the root node.
449  * @param  aFactor           [in] Factor (of type @c FactorDimTree) to use for @c TTV product.
450  * @param  aNumFactors [in] Total number of factors.
451  * @param  id             [in] Identification of the updating factor.
452  * @param  aDeltaSet      [in] @c stl with the Label set of the brother @c ExprNode
453  *                            after the @c TTV product of size @c BrotherLabelSetSize.
454  * @param  aX_partial      [in] @c Tensor used for @c TTV product.
455  * @param  aTnsDims       [in,out] @c stl with the dimensions of the computed Tensor
456  *                            of size @c ParTnsSize.
457  * @param  aGramian       [in,out] Gramian matrix of the @c ExprNode.
458  *
459  * @returns A @c TnsNode with size equal to @c TnsSize.
460  */
461 Tensor_Type TTVs ( FactorDimTree                *const aFactor,
462                   int                          const aNumFactors,
463                   int                          const id,
464                   std::array<int,BrotherLabelSetSize> const &aDeltaSet,
465                   Parent_Tensor_Type            const &aX_partial,
466                   std::array<int,ParTnsSize>   const &aTnsDims,
467                   Hessian_Type                &aGramian
468                   );

```

```

469     * Computes the TTV product of a tensor with a matrix, using recursion.
470     *
471     * @tparam _LabelSetSize      Size of the LabelSet of @c this node.
472     * @tparam _ParLabelSetSize  Size of the LabelSet of the parent node.
473     * @tparam _RootSize        Size of the LabelSet of the root node.
474     * @tparam DeltaSetSize     Size of the DeltaSet of this node.
475     * @tparam ResTnsSize       Order of the resulting @c Tensor.
476     * @tparam ResParTnsSize    Order of the parent's @c Tensor.
477     * @param it                 [in] Factor (of @c FactorDimTree type) to use for @c TTV product.
478     * @param aX_partial        [in] @c Tensor for @c TTV product.
479     * @param aContractDim     [in] Dimension for @c TTV product, based on being a Left
480     *                          or Right child.
481     * @param aTnsDims         [in,out] @c stl @c array with the dimensions of the computed Tensor
482     *                          of size @c ResParTnsSize.
483     * @param aGramian         [in,out] Gramian matrix of the @c ExprNode.
484     * @param aX_result        [in,out] The result of TTV ( @c Tensor ) of size @c ResTnsSize.
485     */
486     template <std::size_t DeltaSetSize, std::size_t ResTnsSize, std::size_t ResParTnsSize>
487     void TTVs_util ( FactorDimTree * const it,
488                   Tensor<static_cast<int>(ResParTnsSize)> const &aX_partial,
489                   int const
490                   aContractDim,
491                   std::array<int,ResParTnsSize> const &aTnsDims,
492                   Hessain_Type &aGramian,
493                   Tensor<static_cast<int>(ResTnsSize)> &aX_result
494                 );
495     / *
496     * Updates the factors in each node until computing the leaf nodes and their
497     * @c Eigen @c Tensors. Based on the position of the node chooses to execute
498     * @c TreeMode_N_Product or @c TTV. Works in recursive way.
499     *
500     * @tparam _LabelSetSize      Size of the LabelSet.
501     * @tparam _ParLabelSetSize  Size of the LabelSet of the parent node.
502     * @tparam _RootSize        Size of the LabelSet of the root node.
503     * @param aNumFactors [in] Total number of factors.
504     * @param id          [in] Identification of the updating factor.
505     * @param aFactor     [in,out] The factor to be updated.
506     */
507     void UpdateTree(int const aNumFactors, int const id, FactorDimTree * aFactor) override;
508 };
509 / *
510 * typedef for zero order tensor.
511 * @tparam TreeDim Tensor Order.
512 */
513     template <std::size_t _TreeDim>
514     using NullExprNode = ExprNode<0,1,_TreeDim>;
515
516     template <std::size_t _TnsSize>
517     struct ExprTree
518     {
519     public:
520         static_assert(_TnsSize >= 1, "Expansion problem in ExprTree!");
521         static constexpr std::size_t TnsSize = _TnsSize;
522         static constexpr bool IsNull = false;
523         using RootExprNode = ExprNode<TnsSize,0,TnsSize>;
524         using DataType = typename RootExprNode::DataType;
525
526     private:
527         RootExprNode root;
528     public:
529         template<typename Array>
530         void Create( std::array<int,RootExprNode::LabelSetSize> &aLabelSet,
531                   Array const &aTnsDims,
532                   int const R,
533                   Tensor<static_cast<int>(TnsSize)> const &aTnsX )
534         {
535             constexpr bool IsLeaf = RootExprNode::IsLeaf;
536             constexpr bool IsRoot = RootExprNode::IsRoot;
537
538             static_assert(! (IsLeaf && IsRoot), "Tree expression with 1 dimension!" );
539
540             constexpr std::size_t vHalf = RootExprNode::DIM_HALF_SIZE;
541             constexpr std::size_t vLabelSetSize = RootExprNode::LabelSetSize;
542
543             root.mUpdated = true;
544             root.mTnsX = aTnsX;
545             std::copy(aTnsDims.begin(), aTnsDims.begin()+TnsSize, root.mTnsDims.begin());
546             root.mLabelSet = aLabelSet;
547             root.mGramian.setConstant(1);
548
549             // Create left child
550             std::copy(aLabelSet.begin(), aLabelSet.begin()+vHalf, root.left.mLabelSet.begin());
551             std::copy(aLabelSet.end()-TnsSize+vHalf, aLabelSet.end(), root.left.mDeltaSet.begin());
552             std::copy(aTnsDims.begin(), aTnsDims.begin()+vHalf, root.left.mTnsDims.begin()+1);
553             root.left.mTnsDims.front() = R;
554             RandomTensorGen(root.left.mTnsDims, root.left.mTnsX);
555
556             // Create right child

```

```

569     std::copy(aLabelSet.end()-(TnsSize-vHalf), aLabelSet.end(),
root.right.mLabelSet.begin());
570     std::copy(aLabelSet.begin(), aLabelSet.begin()+vHalf,
root.right.mDeltaSet.begin());
571     std::copy(aTnsDims.begin()+vHalf, aTnsDims.begin()+vLabelSetSize,
root.right.mTnsDims.begin()+1);
572     root.right.mTnsDims.front() = R;
573     RandomTensorGen(root.right.mTnsDims, root.right.mTnsX);
574
575     // Set key in root node
576     root.mKey = root.left.mLabelSet[vHalf-1];
577
578     Create(root.left.mTnsDims, R, root.left);
579     Create(root.right.mTnsDims, R, root.right);
580 }
581
582 template<typename Array, typename ExprNode>
583 void Create( Array const &aTnsDims,
584             [[maybe_unused]] int const R, // not used in leaf nodes.
585             ExprNode &expr )
586 {
587     static_assert(!std::is_same_v<ExprNode, NullExprNode<TnsSize>, "Expansion problem!");
588
589     using Expr_Node_Type = ExprNode;
590
591     constexpr bool IsLeaf = Expr_Node_Type::IsLeaf;
592     constexpr std::size_t vExprTnsSize = Expr_Node_Type::TnsSize;
593
594     if constexpr (!IsLeaf)
595     {
596         constexpr std::size_t vHalf = Expr_Node_Type::DIM_HALF_SIZE;
597         constexpr std::size_t vLabelSetSize = Expr_Node_Type::LabelSetSize;
598
599         // Create left child
600         std::copy(expr.mLabelSet.begin(), expr.mLabelSet.begin()+vHalf,
601 expr.left.mLabelSet.begin());
602         std::copy(expr.mLabelSet.end()-(vExprTnsSize-vHalf-1), expr.mLabelSet.end(),
603 expr.left.mDeltaSet.begin());
604         std::copy(aTnsDims.begin(), aTnsDims.begin()+vHalf+1,
605 expr.left.mTnsDims.begin());
606         RandomTensorGen(expr.left.mTnsDims, expr.left.mTnsX);
607
608         // Create right child
609         std::copy(expr.mLabelSet.end()-(vExprTnsSize-vHalf-1), expr.mLabelSet.end(),
610 expr.right.mLabelSet.begin());
611         std::copy(expr.mLabelSet.begin(), expr.mLabelSet.begin()+vHalf,
612 expr.right.mDeltaSet.begin());
613         std::copy(aTnsDims.begin()+vHalf+1, aTnsDims.begin()+vLabelSetSize+1,
614 expr.right.mTnsDims.begin()+1);
615         expr.right.mTnsDims.front() = R;
616         RandomTensorGen(expr.right.mTnsDims, expr.right.mTnsX);
617
618         expr.mKey = expr.left.mLabelSet[vHalf-1];
619
620         Create(expr.left.mTnsDims, R, expr.left);
621         Create(expr.right.mTnsDims, R, expr.right);
622     }
623     else
624     {
625         expr.mKey = expr.mLabelSet[0];
626     }
627 }
628 };
629
630 /*
631 * ExprTree with no nodes.
632 */
633 template <>
634 struct ExprTree<0>
635 {
636     static constexpr bool IsNull = true;
637 };
638
639 struct FactorDimTree : public Factor<Tensor<2>>
640 {
641     TnsNode<1> * leaf;
642     FactorDimTree() : leaf(nullptr)
643     { }
644 };
645
646 template <std::size_t _LabelSetSize, std::size_t _ParLabelSetSize, std::size_t _RootSize>
647 typename ExprNode<_LabelSetSize,_ParLabelSetSize,_RootSize>::Parent_Tensor_Type
648 ExprNode<_LabelSetSize,_ParLabelSetSize,_RootSize>::TreeMode_N_Product( FactorDimTree
649 * const aFactor,
650
651 int

```

```

672     const aNumFactors,
673     const id,
674     const &aTnsDims,
675     &aGramian,
676     std::array<int, BrotherLabelSetSize> &aDeltaSet )
677 {
678     int vContractDim;
679     Parent_Tensor_Type vX_partial;
680     FactorDimTree * it;
681     constexpr std::size_t aDeltaSetSize = BrotherLabelSetSize-1;
682     static_assert(!IsRoot, "TreeMode_N_Product() must not be called on root node!");
683     const std::size_t R = aFactor->gramian.dimension(0);
684     std::array<int, ParTnsSize> vTnsDims;
685     std::array<Eigen::IndexPair<int>, 1> product_dims;
686     if (this == Parent()->Left())
687     {
688         it = aFactor+aNumFactors-id-1;
689         vContractDim = RootSize-1;
690         std::copy(mDeltaSet.begin(), mDeltaSet.begin()+aDeltaSetSize, aDeltaSet.begin());
691         std::copy(aTnsDims.begin(), aTnsDims.end()-1, vTnsDims.begin()+1);
692         vTnsDims.front() = R;
693     }
694     else
695     {
696         it = aFactor-id;
697         vContractDim = 0;
698         std::copy(mDeltaSet.end()-aDeltaSetSize, mDeltaSet.end(), aDeltaSet.begin());
699         std::copy(aTnsDims.begin()+1, aTnsDims.end(), vTnsDims.begin()+1);
700         vTnsDims.front() = R;
701     }
702     aGramian = (* it).gramian;
703     product_dims = { Eigen::IndexPair<int>(0, vContractDim) };
704     vX_partial.resize(vTnsDims);
705     vX_partial = it->factor.contract( *reinterpret_cast<Parent_Tensor_Type *>(Parent()->TensorX()),
706     product_dims);
707     return vX_partial;
708 }
709
710 template <std::size_t _LabelSetSize, std::size_t _ParLabelSetSize, std::size_t _RootSize>
711 template <std::size_t DeltaSetSize, std::size_t ResTnsSize, std::size_t ResParTnsSize>
712 void
713 ExprNode<_LabelSetSize,_ParLabelSetSize,_RootSize>::TTVs_util( FactorDimTree
714 * const it,
715 Tensor<static_cast<int>(ResParTnsSize)> const &aX_partial,
716 const aContractDim,
717 const &aTnsDims,
718 &aGramian,
719 &aX_result )
720 {
721     constexpr int _ResParTnsSize = ResParTnsSize - 1;
722     using Result_Tensor_Type = Tensor<_ResParTnsSize>;
723     std::array<int, static_cast<std::size_t>(_ResParTnsSize)> vTnsDims;
724     const std::size_t R = aGramian.dimension(0);
725     aGramian *= (* it).gramian; // Hadamard Product.
726     // Allocate the reduction result
727     if (this == Parent()->Right()) // Right Child
728     {
729         std::copy(aTnsDims.end()-_ResParTnsSize+1, aTnsDims.end(), vTnsDims.begin()+1);
730         vTnsDims.front() = R;
731     }
732     else // Left Child
733     {
734         // 1o TTV
735         if constexpr (IsFirstChild)
736         {
737             // First Child and First TTV
738             if constexpr (DeltaSetSize == BrotherLabelSetSize - 1)

```

```

764     {
765         std::copy(aTnsDims.begin(), aTnsDims.begin()+_LabelSetSize, vTnsDims.begin()+1);
766         if constexpr (_ResParTnsSize - _LabelSetSize > 0)
767             std::copy(aTnsDims.end() - (_ResParTnsSize - _LabelSetSize), aTnsDims.end(),
vTnsDims.begin()+_LabelSetSize+1);
768         vTnsDims.front() = R;
769     }
770     else
771     {
772         std::copy(aTnsDims.begin(), aTnsDims.begin()+_LabelSetSize+1, vTnsDims.begin());
773         if constexpr (_ResParTnsSize - _LabelSetSize > 0)
774             std::copy(aTnsDims.end() - (_ResParTnsSize - _LabelSetSize) + 1, aTnsDims.end(),
vTnsDims.begin()+_LabelSetSize+1);
775     }
776 }
777 else
778 {
779     std::copy(aTnsDims.begin(), aTnsDims.begin()+_LabelSetSize+1, vTnsDims.begin());
780     if constexpr (_ResParTnsSize - _LabelSetSize > 0)
781         std::copy(aTnsDims.end() - (_ResParTnsSize - _LabelSetSize) + 1, aTnsDims.end(),
vTnsDims.begin()+_LabelSetSize+1);
782 }
783 }
784
785 // Apply reduction
786 Result_Tensor_Type vTnsX;
787 vTnsX.resize(vTnsDims);
788
789 TensorPartialProduct_R<ResParTnsSize,_ResParTnsSize>(aX_partial, it->factor, 0, aContractDim,
&vTnsX);
790
791 // Update tensor orders
792 if constexpr (_ResParTnsSize > ResTnsSize)
793 {
794     TTVs_util<DeltaSetSize-1,ResTnsSize>(it+1, vTnsX, aContractDim, vTnsDims, aGramian, aX_result);
795 }
796 else
797 {
798     aX_result = vTnsX;
799 }
800 }
801
802 template <std::size_t _LabelSetSize, std::size_t _ParLabelSetSize, std::size_t _RootSize>
803 typename ExprNode<_LabelSetSize,_ParLabelSetSize,_RootSize>::Tensor_Type
804 ExprNode<_LabelSetSize,_ParLabelSetSize,_RootSize>::TTVs( FactorDimTree * const
805 aFactor,
806
807 int const
808 aNumFactors,
809
810 int const
811 id,
812
813 std::array<int,BrotherLabelSetSize> const
814 &aDeltaSet,
815
816 Parent_Tensor_Type const
817 &aX_partial,
818
819 std::array<int,ParTnsSize> const
820 &aTnsDims,
821
822 Hessian_Type
823 &aGramian )
824 {
825     using Result_Tensor_Type = ExprNode<_LabelSetSize,_ParLabelSetSize,_RootSize>::Tensor_Type;
826
827     constexpr std::size_t vDeltaSetSize = IsFirstChild ? BrotherLabelSetSize - 1 : BrotherLabelSetSize;
828
829     static_assert( (vDeltaSetSize+TnsSize) == ParTnsSize, "Wrong call!" );
830
831     int vContractDim = (this == Parent()->Right()) ? 0 : LabelSetSize;
832
833     FactorDimTree * it;
834
835     it = aFactor; // TODO There is no range check for it !
836     assert(aDeltaSet[0] - 1 < aNumFactors);
837     it = aFactor + (aDeltaSet[0] - id - 1);
838
839     Result_Tensor_Type vResTensor;
840
841     TTVs_util<vDeltaSetSize,TnsSize>(it, aX_partial, vContractDim, aTnsDims, aGramian, vResTensor);
842
843     return vResTensor;
844 }
845
846 template <std::size_t _LabelSetSize, std::size_t _ParLabelSetSize, std::size_t _RootSize>
847 void ExprNode<_LabelSetSize,_ParLabelSetSize,_RootSize>::UpdateTree(int const aNumFactors, int const
848 id, FactorDimTree * aFactor)
849 {
850     if (mUpdated) // Root is always mUpdated == true
851     {
852         if constexpr (!IsRoot)

```

```

868     left.SetOutdated();
869     else
870     {
871         if constexpr (!IsLeaf)
872         {
873             if (left.mUpdated)
874                 left.SetOutdated();
875             else if (right.mUpdated)
876                 right.SetOutdated();
877         }
878     }
879 }
880 else
881 {
882     Parent()->UpdateTree(aNumFactors, id, aFactor);
883     int R = mTnsDims.back();
884
885     Hessian_Type          vGramian(R,R);
886     Tensor_Type          vX_temp;
887     Parent_Tensor_Type   vX_partial;
888
889     std::array<int, ParTnsSize> vTnsDims;
890     std::array<int, BrotherLabelSetSize> vDeltaSet;
891
892     std::size_t vDeltaSetSize = (IsFirstChild) ? BrotherLabelSetSize-1 : BrotherLabelSetSize;
893
894     vTnsDims = *reinterpret_cast<std::array<int, ParTnsSize>> *>(Parent()->TnsDims());
895
896     if constexpr (IsFirstChild)
897     { // Case: leaf is a child of the root: mode n product is required
898         vX_partial = TreeMode_N_Product(aFactor, aNumFactors, id, vTnsDims, vGramian, vDeltaSet);
899     }
900     else
901     {
902         std::copy(mDeltaSet.begin(), mDeltaSet.begin()+vDeltaSetSize, vDeltaSet.begin());
903         vX_partial = *reinterpret_cast<Parent_Tensor_Type *>(Parent()->TensorX());
904         vGramian = Parent()->Gramian();
905     }
906
907     if constexpr (TnsSize == ParTnsSize)
908     {
909         vX_temp = vX_partial;
910     }
911     else if constexpr (!IsRoot)
912     {
913         if (vDeltaSetSize > 0)
914         {
915             vX_temp = TTVs(aFactor, aNumFactors, id, vDeltaSet, vX_partial, vTnsDims, vGramian); // TODO
916             check for bad allocation
917         } // TODO check all paths
918     }
919     else
920     {
921         // TODO Now???
922     }
923
924     // Update the current tree node
925     mTnsX = vX_temp;
926     mGramian = vGramian;
927     mUpdated = true;
928 }
929
930 template<std::size_t _TreeDim>
931 L_TnsNode * search_leaf(int const key, ExprTree<_TreeDim> &tree)
932 {
933     if constexpr (_TreeDim >= 0)
934         return tree.root.SearchKey(key);
935     else
936         return nullptr;
937 }
938 } // end namespace partensor
939
940 #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.](#)

```

1 #ifndef DOXYGEN_SHOULD_SKIP_THIS
15 #endif // DOXYGEN_SHOULD_SKIP_THIS
16 / ***** /
24 #ifndef PARTENSOR_EXECUTION_HPP
25 #define PARTENSOR_EXECUTION_HPP
26
27 #include <type_traits>
28
29 namespace partensor::execution {
30 inline namespace v1 {
31
32 class sequenced_policy
33 {
34 public:
35     // For internal use only
36     static constexpr std::false_type __allow_unsequenced()
37     {
38         return std::false_type{};
39     }
40
41     static constexpr std::false_type __allow_vector()
42     {
43         return std::false_type{};
44     }
45
46     static constexpr std::false_type __allow_parallel()
47     {
48         return std::false_type{};
49     }
50
51     static constexpr std::false_type __allow_cuda()
52     {
53         return std::false_type{};
54     }
55
56     static constexpr std::false_type __allow_openmpi()
57     {
58         return std::false_type{};
59     }
60
61     static constexpr std::false_type __allow_openmp()
62     {
63         return std::false_type{};
64     }
65 };
66
67 class parallel_policy
68 {
69 public:
70     // For internal use only
71     static constexpr std::false_type __allow_unsequenced()
72     {
73         return std::false_type{};
74     }
75
76     static constexpr std::false_type __allow_vector()
77     {
78         return std::false_type{};
79     }
80
81     static constexpr std::true_type __allow_parallel()
82     {
83         return std::true_type{};
84     }
85
86     static constexpr std::false_type __allow_cuda()
87     {
88         return std::false_type{};
89     }
90
91     static constexpr std::false_type __allow_openmpi()
92     {

```



```
93     return std::false_type{};
94 }
95
96 static constexpr std::false_type __allow_openmp()
97 {
98     return std::false_type{};
99 }
100 };
101
102 class parallel_unsequenced_policy
103 {
104 public:
105     // For internal use only
106     static constexpr std::true_type __allow_unsequenced()
107     {
108         return std::true_type{};
109     }
110
111     static constexpr std::true_type __allow_vector()
112     {
113         return std::true_type{};
114     }
115
116     static constexpr std::true_type __allow_parallel()
117     {
118         return std::true_type{};
119     }
120
121     static constexpr std::false_type __allow_cuda()
122     {
123         return std::false_type{};
124     }
125
126     static constexpr std::false_type __allow_openmpi()
127     {
128         return std::false_type{};
129     }
130
131     static constexpr std::false_type __allow_openmp()
132     {
133         return std::false_type{};
134     }
135 };
136
137 class unsequenced_policy
138 {
139 public:
140     // For internal use only
141     static constexpr std::true_type __allow_unsequenced()
142     {
143         return std::true_type{};
144     }
145
146     static constexpr std::true_type __allow_vector()
147     {
148         return std::true_type{};
149     }
150
151     static constexpr std::false_type __allow_parallel()
152     {
153         return std::false_type{};
154     }
155
156     static constexpr std::false_type __allow_cuda()
157     {
158         return std::false_type{};
159     }
160
161     static constexpr std::false_type __allow_openmpi()
162     {
163         return std::false_type{};
164     }
165
166     static constexpr std::false_type __allow_openmp()
167     {
168         return std::false_type{};
169     }
170 };
171
172 class cuda_policy
173 {
174 public:
175     // For internal use only
176     static constexpr std::false_type __allow_unsequenced()
177     {
178         return std::false_type{};
179     }
180
181     static constexpr std::false_type __allow_vector()
```

```
180 {
181     return std::false_type();
182 }
183
184 static constexpr std::false_type __allow_parallel()
185 {
186     return std::false_type();
187 }
188
189 static constexpr std::true_type __allow_cuda()
190 {
191     return std::true_type();
192 }
193
194 static constexpr std::false_type __allow_openmpi()
195 {
196     return std::false_type();
197 }
198
199 static constexpr std::false_type __allow_openmp()
200 {
201     return std::false_type();
202 }
203 };
204
205 class openmpi_policy
206 {
207 public:
208     // For internal use only
209     static constexpr std::false_type __allow_unsequenced()
210     {
211         return std::false_type();
212     }
213
214     static constexpr std::false_type __allow_vector()
215     {
216         return std::false_type();
217     }
218
219     static constexpr std::false_type __allow_parallel()
220     {
221         return std::false_type();
222     }
223
224     static constexpr std::false_type __allow_cuda()
225     {
226         return std::false_type();
227     }
228
229     static constexpr std::true_type __allow_openmpi()
230     {
231         return std::true_type();
232     }
233
234     static constexpr std::false_type __allow_openmp()
235     {
236         return std::false_type();
237     }
238 };
239
240 class openmp_policy
241 {
242 public:
243     // For internal use only
244     static constexpr std::false_type __allow_unsequenced()
245     {
246         return std::false_type();
247     }
248
249     static constexpr std::false_type __allow_vector()
250     {
251         return std::false_type();
252     }
253
254     static constexpr std::false_type __allow_parallel()
255     {
256         return std::false_type();
257     }
258
259     static constexpr std::false_type __allow_cuda()
260     {
261         return std::false_type();
262     }
263
264     static constexpr std::false_type __allow_openmpi()
265     {
266         return std::false_type();
```

```

267 }
268
269 static constexpr std::true_type __allow_openmp()
270 {
271     return std::true_type{};
272 }
273 };
274
275 constexpr sequenced_policy      seq{};
276 constexpr parallel_policy       par{};
277 constexpr parallel_unsequenced_policy par_unseq{};
278 constexpr unsequenced_policy   unseq{};
279 constexpr cuda_policy           cuda{};
280 constexpr openmpi_policy        mpi{};
281 constexpr openmp_policy         omp{};
282
283 template <class T>
284 struct is_execution_policy : std::false_type
285 { };
286
287 template <>
288 struct is_execution_policy<sequenced_policy> : std::true_type
289 { };
290
291 template <>
292 struct is_execution_policy<parallel_policy> : std::true_type
293 { };
294
295 template <>
296 struct is_execution_policy<parallel_unsequenced_policy> : std::true_type
297 { };
298
299 template <>
300 struct is_execution_policy<unsequenced_policy> : std::true_type
301 { };
302
303 template <>
304 struct is_execution_policy<cuda_policy> : std::true_type
305 { };
306
307 template <>
308 struct is_execution_policy<openmpi_policy> : std::true_type
309 { };
310
311 template <>
312 struct is_execution_policy<openmp_policy> : std::true_type
313 { };
314
315 template <class T>
316 constexpr bool is_execution_policy_v = is_execution_policy<T>::value;
317
318 template <typename P>
319 using execution_policy_t = std::remove_cv_t<std::remove_reference_t<P>>;
320
321 } // v1
322
323 namespace internal
324 {
325     template <class ExecPolicy, class T>
326     using enable_if_execution_policy = typename
327         std::enable_if<partensor::execution::is_execution_policy<typename
328             std::decay<ExecPolicy>::type>::value, T>::type;
329 } // namespace internal
330
331 } // partensor::execution
332
333 #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 >, Sparse -`
`DefaultValues > 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 &&, Sparse -`
`Options< _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          >, 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 <code>partensor::Tensor < order ></code> , 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 <code>partensor::Tensor < order ></code> , 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.](#)

```

1 #ifndef DOXYGEN_SHOULD_SKIP_THIS
15 #endif // DOXYGEN_SHOULD_SKIP_THIS
16 / ***** /
26 #ifndef PARTENSOR_GTC_HPP
27 #define PARTENSOR_GTC_HPP
28
29 #include "PARTENSOR_basic.hpp"
30 #include "PartialCwiseProd.hpp"
31 #include "MTTKRP.hpp"
32 #include "NesterovMNLS.hpp"
33 #include "Normalize.hpp"
34 #include "Timers.hpp"
35 #include "ReadWrite.hpp"
36
37 namespace partensor
38 {
39     inline namespace v1
40     {
41         namespace internal
42         {

```

```

43 //template <typename ExecutionPolicy, typename Tensor>
44 //execution::internal::enable_if_execution_policy<ExecutionPolicy,Tensor>
45 //Status gtc_f(ExecutionPolicy &&, Tensor const &tnsX, std::size_t rank);
46
47 / *
48  * Includes the implementation of General Tensor Completion. Based on the given
49  * parameters one of the overloaded operators will be called.
50  */
51 template <std::size_t TnsSize_>
52 struct GTC_Base
53 {
54     static constexpr std::size_t TnsSize = TnsSize_;
55     static constexpr std::size_t lastFactor = TnsSize - 1;
56     using SparseTensor = typename partensor::SparseTensor<TnsSize_>;
57     using DataType = typename SparseTensorTraits<SparseTensor>::DataType;
58     using MatrixType = typename SparseTensorTraits<SparseTensor>::MatrixType;
59     using Dimensions = typename SparseTensorTraits<SparseTensor>::Dimensions;
60     using SparseMatrix = typename SparseTensorTraits<SparseTensor>::SparseMatrixType;
61     using LongMatrix = typename SparseTensorTraits<SparseTensor>::LongMatrixType;
62     using Constraints = typename SparseTensorTraits<SparseTensor>::Constraints;
63     using MatrixArray = typename SparseTensorTraits<SparseTensor>::MatrixArray;
64     using DoubleArray = typename SparseTensorTraits<SparseTensor>::DoubleArray;
65     using IntArray = typename SparseTensorTraits<SparseTensor>::IntArray;
66     template<int mode>
67     void sort_ratings_base_util(Matrix const &Ratings_Base_T,
68                               SparseTensor &tnsX,
69                               IntArray const &tnsDims,
70                               long int const nnz)
71     {
72         Matrix ratings_base_temp = Ratings_Base_T;
73         std::vector<std::vector<double>> vectorized_ratings_base;
74         vectorized_ratings_base.resize(nnz,std::vector<double>(TnsSize + 1));
75
76         for (int rows = 0; rows < static_cast<int>(TnsSize) + 1; rows++)
77         {
78             for (int cols = 0; cols < nnz; cols++)
79             {
80                 vectorized_ratings_base[cols][rows] = Ratings_Base_T(rows, cols);
81             }
82         }
83         // Sort
84         std::sort(vectorized_ratings_base.begin(), vectorized_ratings_base.end(), SortRows<TnsSize_,
85 mode, double>);
86
87         for (int rows = 0; rows < static_cast<int>(TnsSize) + 1; rows++)
88         {
89             for (int cols = 0; cols < nnz; cols++)
90             {
91                 ratings_base_temp(rows, cols) = vectorized_ratings_base[cols][rows];
92             }
93         }
94
95         FillSparseMatricization<TnsSize>(tnsX, nnz, ratings_base_temp, tnsDims, mode);
96
97         if constexpr (mode+1 < TnsSize)
98             sort_ratings_base_util<mode+1>(Ratings_Base_T, tnsX, tnsDims, nnz);
99     }
100
101 void sort_ratings_base(Matrix const &Ratings_Base_T,
102                      SparseTensor &tnsX,
103                      IntArray const &tnsDims,
104                      long int const nnz)
105 {
106     ReserveSparseTensor<TnsSize>(tnsX, tnsDims, nnz);
107     sort_ratings_base_util<0>(Ratings_Base_T, tnsX, tnsDims, nnz);
108 }
109
110 template <std::size_t TnsSize_, typename ExecutionPolicy = execution::sequenced_policy>
111 struct GTC : public GTC_Base<TnsSize_>
112 {
113     using GTC_Base<TnsSize_>::TnsSize;
114     using GTC_Base<TnsSize_>::lastFactor;
115     using GTC_Base<TnsSize_>::Dimensions;
116     using GTC_Base<TnsSize_>::MatrixArray;
117     using GTC_Base<TnsSize_>::DataType;
118     using GTC_Base<TnsSize_>::SparseTensor;
119     using GTC_Base<TnsSize_>::IntArray;
120     using GTC_Base<TnsSize_>::LongMatrix;
121
122     using Options =
123     partensor::SparseOptions<TnsSize_,execution::sequenced_policy,SparseDefaultValues>;
124     using Status =
125     partensor::SparseStatus<TnsSize_,execution::sequenced_policy,SparseDefaultValues>;
126
127     // Variables that will be used in gtc implementations.
128     struct Member_Variables
129 
```

```

130     {
131         MatrixArray factors_T;
132         MatrixArray factor_T_factor;
133         MatrixArray mttkrp_T;
134         IntArray tnsDims;
135         std::array<std::array<int, TnsSize_ -1>, TnsSize_> offsets;
136
137         MatrixArray norm_factors_T;
138         MatrixArray old_factors_T;
139
140         Matrix cwise_factor_product;
141         SparseTensor tnsX;
142         int rank;
143
144         Member_Variables() = default;
145
146         Member_Variables(int R, IntArray dims) : tnsDims(dims),
147                                                 rank(R)
148     {}
149
150     Member_Variables(Member_Variables const &) = default;
151     Member_Variables(Member_Variables &&) = default;
152
153     Member_Variables &operator=(Member_Variables const &) = default;
154     Member_Variables &operator=(Member_Variables &&) = default;
155 };
156
157 /*
158  * In case option variable @c writeToFile is enabled, then, before the end
159  * of the algorithm, it writes the resulted factors in files, whose
160  * paths are specified before compiling in @ options.final_factors_path.
161  *
162  * @param st [in] Struct where the returned values of @c Gtc are stored.
163  */
164 void writeFactorsToFile(Status const &st)
165 {
166     std::size_t size;
167     for(std::size_t i=0; i<TnsSize; ++i)
168     {
169         size = st.factors[i].rows() * st.factors[i].cols();
170         partensor::write(st.factors[i],
171                        st.options.final_factors_paths[i],
172                        size);
173     }
174 }
175
176 /*
177  * Compute the cost function value at the end of each outer iteration
178  * based on the last factor.
179  *
180  * @param mv [in] Struct where ALS variables are stored.
181  * @param st [in,out] Struct where the returned values of @c Gtc are stored.
182  * In this case the cost function value is updated.
183  */
184 void cost_function(Member_Variables const &mv,
185                  Status &st)
186 {
187     Matrix temp_R_1(mv.rank, 1);
188     double temp_1_1 = 0;
189     st.f_value = 0;
190
191     std::array<int, TnsSize-1> offsets;
192     offsets[0] = 1;
193     for (int j = 1; j < static_cast<int>(TnsSize) - 1; j++)
194     {
195         offsets[j] = offsets[j - 1] * mv.tnsDims[j-1];
196     }
197
198     for (long int i = 0; i < mv.tnsX[lastFactor].outerSize(); ++i)
199     {
200         int row = 0;
201         for (SparseMatrix::InnerIterator it(mv.tnsX[lastFactor], i); it; ++it)
202         {
203             temp_R_1 = mv.factors_T[lastFactor].col(it.col());
204             // Select rows of each factor and compute the Hadamard product of the respective row of
the Khatri-Rao product, and the row of factor A_N.
205             for (int mode_i = static_cast<int>(TnsSize) - 2; mode_i >= 0; mode_i--)
206             {
207                 row = ((it.row()) / offsets[mode_i]) % (mv.tnsDims[mode_i]);
208                 temp_R_1 = temp_R_1.cwiseProduct(mv.factors_T[mode_i].col(row));
209             }
210             temp_1_1 = it.value() - temp_R_1.sum();
211             st.f_value += temp_1_1 * temp_1_1;
212         }
213     }
214 }
215

```

```

216  /*
217  * Compute the cost function value at the end of each outer iteration
218  * based on the last accelerated factor.
219  *
220  * @param mv [in] Struct where ALS variables are stored.
221  * @param accel_factors [in] Accelerated factors.
222  *
223  * @returns The cost function calculated with the accelerated factors.
224  */
225 double accel_cost_function(Member_Variables const &mv,
226 MatrixArray const &accel_factors)
227 {
228 Matrix temp_R_1(mv.rank, 1);
229 double temp_1_1 = 0;
230 double f_value = 0;
231
232 std::array<int,TnsSize-1> offsets;
233 offsets[0] = 1;
234 for (int j = 1; j < static_cast<int>(TnsSize) - 1; j++)
235 {
236 offsets[j] = offsets[j - 1] * mv.tnsDims[j-1];
237 }
238
239 for (long int i = 0; i < mv.tnsX[lastFactor].outerSize(); ++i)
240 {
241 int row = 0;
242 for (SparseMatrix::InnerIterator it(mv.tnsX[lastFactor], i); it; ++it)
243 {
244 temp_R_1 = accel_factors[lastFactor].col(it.col());
245 // 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.
246 // temp_R_1 = A_N(i_N,:) . * ... . * A_2(i_2,:) . * A_1(i_1,:)
247 for (int mode_i = static_cast<int>(TnsSize) - 2; mode_i >= 0; mode_i--)
248 {
249 row = ((it.row()) / offsets[mode_i]) % (mv.tnsDims[mode_i]);
250 temp_R_1 = temp_R_1.cwiseProduct(accel_factors[mode_i].col(row));
251 }
252 temp_1_1 = it.value() - temp_R_1.sum();
253 f_value += temp_1_1 * temp_1_1;
254 }
255 }
256 return f_value;
257 }
258
259 void calculate_offsets(Member_Variables &mv)
260 {
261 for (int idx = 0; idx < static_cast<int>(TnsSize); idx++)
262 {
263 mv.offsets[idx][0] = 1;
264 for (int j = 1, mode = 0; j < static_cast<int>(TnsSize) - 1; j++, mode++)
265 {
266 if (idx == mode)
267 {
268 mode++;
269 }
270 mv.offsets[idx][j] = mv.offsets[idx][j - 1] * mv.tnsDims[mode];
271 }
272 }
273 }
274
275 void unconstraint_update(int const idx,
276 Member_Variables &mv,
277 Status &st)
278 {
279 int r = mv.rank;
280
281 Matrix eye = st.options.lambdas[idx] * Matrix::Identity(r, r);
282
283 int last_mode = (idx == static_cast<int>(TnsSize) - 1) ? static_cast<int>(TnsSize) - 2 :
static_cast<int>(TnsSize) - 1;
284
285 Matrix MTTKRP_col(r, 1);
286 Matrix temp_RxR(r, r);
287 Matrix temp_R_1(r, 1);
288
289 // Compute MTTKRP
290 for (long int i = 0; i < mv.tnsX[idx].outerSize(); ++i)
291 {
292 MTTKRP_col.setZero();
293 temp_RxR.setZero(); // is the Hadamard product of Grammians of the Factors, that
correspond to the nnz elements of the Tensor.
294 for (SparseMatrix::InnerIterator it(mv.tnsX[idx], i); it; ++it)
295 {
296 temp_R_1 = Matrix::Ones(r, 1);
297 int row;
298 // Select rows of each factor an compute the respective row of the Khatri-Rao
product.

```



```

299         for (int mode_i = last_mode, kr_counter = static_cast<int>(TnsSize) - 2; mode_i >= 0
    && kr_counter >= 0; mode_i--)
300         {
301             if (mode_i == idx)
302             {
303                 continue;
304             }
305             row = ((it.row()) / mv.offsets[idx][kr_counter]) % (mv.tnsDims[mode_i]);
306             temp_R_1 = temp_R_1.cwiseProduct(mv.factors_T[mode_i].col(row));
307             kr_counter--;
308         }
309         // Subtract from the previous row the respective row of W, according to relation
(9).
310         MTTKRP_col.noalias() += it.value() * temp_R_1;
311         temp_RxR.noalias() += temp_R_1 * temp_R_1.transpose();
312     }
313     mv.factors_T[idx].col(i) = (temp_RxR + eye).inverse() * MTTKRP_col;
314 }
315 }
316
317 /*
318 * Based on each factor's constraint, a different
319 * update function is used at every outer iteration.
320 *
321 * Computes also factor^T * factor at the end.
322 *
323 * @param idx [in] Factor to be updated.
324 * @param mv [in] Struct where ALS variables are stored.
325 * @param st [in,out] Struct where the returned values of @c Gtc are stored.
326 * Updates the @c stl array with the factors.
327 */
328 void update_factor(int Member_Variables const idx,
329                   Status &mv,
330                   &st )
331 {
332     // Update factor
333     switch ( st.options.constraints[idx] )
334     {
335     case Constraint::unconstrained:
336     case Constraint::symmetric:
337     {
338         unconstraint_update(idx, mv, st);
339         break;
340     }
341     case Constraint::nonnegativity:
342     case Constraint::symmetric_nonnegativity:
343     {
344         int last_mode = (idx == static_cast<int>(TnsSize) - 1) ? static_cast<int>(TnsSize) - 2 :
static_cast<int>(TnsSize) - 1;
345         SparseMTTKRP(mv.tnsDims, mv.tnsX[idx], mv.factors_T, mv.rank, mv.offsets[idx], last_mode,
idx, mv.mttkrp_T[idx]);
346
347         // NesterovMNLS(mv.cwise_factor_product, mv.factors_T, mv.tnsDims, mv.tnsX[idx],
mv.offsets[idx], st.options.max_nesterov_iter,
348         // st.options.lambdas[idx], idx, st.options.constraints[idx],
mv.mttkrp_T[idx]);
349         local_L::NesterovMNLS(mv.factors_T, mv.tnsDims, mv.tnsX[idx], mv.offsets[idx],
st.options.max_nesterov_iter,
350         st.options.lambdas[idx], idx, mv.mttkrp_T[idx]);
351         break;
352     }
353     default: // in case of Constraint::constant
354         break;
355     }
356
357     // Compute A^T * A + B^T * B + ...
358     st.factors[idx] = mv.factors_T[idx].transpose();
359     if (st.options.constraints[idx] == Constraint::symmetric_nonnegativity ||
st.options.constraints[idx] == Constraint::symmetric)
360     {
361         for (std::size_t i=0; i<TnsSize; i++)
362         {
363             if (i != static_cast<std::size_t>(idx))
364             {
365                 mv.factors_T[i] = mv.factors_T[idx];
366             }
367         }
368     }
369     mv.factor_T_factor[idx].noalias() = mv.factors_T[idx] * st.factors[idx];
370 }
371
372 /*
373 * @brief Line Search Acceleration
374 *
375 * Performs an acceleration step on the updated factors, and keeps the accelerated factors
376 * when the step succeeds. Otherwise, the acceleration step is ignored.
377 * Line Search Acceleration reduces the number of outer iterations in the ALS algorithm.

```

```

378      *
379      * @note This implementation ONLY, if factors are of @c Matrix type.
380      *
381      * @param mv [in,out] Struct where ALS variables are stored.
382      *           In case the acceleration step is successful the Gramian
383      *           matrices of factors are updated.
384      * @param st [in,out] Struct where the returned values of @c Gtc are stored.
385      *           If the acceleration succeeds updates @c factors
386      *           and cost function value.
387      *
388      */
389 void line_search_accel(Member_Variables &mv,
390                      Status           &st)
391 {
392     double f_accel = 0.0; // Objective Value after the acceleration step
393     double const accel_step = pow(st.ao_iter+1,(1.0/(st.options.accel_coeff)));
394
395     MatrixArray accel_factors_T;
396     MatrixArray accel_gramians;
397
398     for(std::size_t i=0; i<TnsSize; ++i)
399     {
400         mv.old_factors_T[i] = mv.old_factors_T[i] + accel_step
401                               * (mv.factors_T[i] -
402                                   mv.old_factors_T[i]);
403         accel_gramians[i] = accel_factors_T[i]
404                             * accel_factors_T[i].transpose();
405     }
406
407     f_accel = accel_cost_function(mv, accel_factors_T);
408     if (st.f_value > f_accel)
409     {
410         mv.factors_T = accel_factors_T;
411         mv.factor_T_factor = accel_gramians;
412         st.f_value = f_accel;
413         Partensor()->Logger()->info("Acceleration Step SUCCEEDED! at iter: {}", st.ao_iter);
414     }
415     else
416     {
417         st.options.accel_fail++;
418     }
419     if (st.options.accel_fail==5)
420     {
421         st.options.accel_fail=0;
422         st.options.accel_coeff++;
423     }
424 }
425
426 /*
427 * Sequential implementation of Alternating Least Squares (ALS) method.
428 *
429 * @param R [in] The rank of decomposition.
430 * @param mv [in] Struct where ALS variables are stored and being updated
431 *               until a termination condition is true.
432 * @param st [in,out] Struct where the returned values of @c Gtc are stored.
433 */
434 void aogtc(Member_Variables &mv,
435           Status           &status)
436 {
437     for (std::size_t i=0; i<TnsSize; i++)
438     {
439         // mv.factors_T[i] = status.factors[i].transpose();
440         // mv.factor_T_factor[i].noalias() = mv.factors_T[i]
441         //                               * status.factors[i];
442         mv.factor_T_factor[i].noalias() = mv.factors_T[i]
443                                           * mv.factors_T[i].transpose();
444         mv.mttkrp_T[i] = Matrix(mv.rank, mv.tnsDims[i]);
445     }
446
447     // if(status.options.normalization)
448     // {
449     //     choose_normilization_factor(status, mv.all_orthogonal, mv.weight_factor);
450     // }
451     // Normalize(static_cast<int>(R), mv.factor_T_factor, status.factors);
452
453     status.frob_tns = (mv.tnsX[0]).squaredNorm();
454     cost_function(mv, status);
455     status.rel_costFunction = status.f_value/status.frob_tns;
456
457     // ---- Loop until ALS converges ----
458     while(1)
459     {
460         status.ao_iter++;
461         Partensor()->Logger()->info("iter: {} -- fvalue: {} -- relative_costFunction: {}",
462             status.ao_iter,
463             status.f_value, status.rel_costFunction);
464
465         for (std::size_t i=0; i<TnsSize; i++)
466         {
467             mv.cwise_factor_product = PartialCwiseProd(mv.factor_T_factor, i);
468
469             // Update factor

```

```

463         update_factor(i, mv, status);
464     }
465
466     cost_function(mv, status);
467     status.rel_costFunction = status.f_value/status.frob_tns;
468
469     // if(status.options.normalization && !mv.all_orthogonal)
470     //     Normalize(mv.weight_factor, static_cast<int>(R), mv.factor_T_factor, status.factors);
471
472     // ---- Terminating condition ----
473     if (status.rel_costFunction < status.options.threshold_error || status.ao_iter >=
status.options.max_iter)
474     {
475         if(status.options.writeToFile)
476             writeFactorsToFile(status);
477         break;
478     }
479
480     if (status.options.acceleration)
481     {
482         mv.norm_factors_T = mv.factors_T;
483         // ---- Acceleration Step ----
484         if (status.ao_iter > 1)
485             line_search_accel(mv, status);
486
487         mv.old_factors_T = mv.norm_factors_T;
488     }
489     } // end of while
490 }
491
492 void initialize_factors(Member_Variables &mv,
493                       Status          &status)
494 {
495     if(status.options.initialized_factors)
496     {
497         if(status.options.read_factors_from_file)
498         {
499             for(std::size_t i=0; i<TnsSize; ++i)
500             {
501                 status.factors[i] = Matrix(mv.tnsDims[i], static_cast<int>(mv.rank));
502                 read( status.options.initial_factors_paths[i],
503                     mv.tnsDims[i] * mv.rank,
504                     0,
505                     status.factors[i] );
506             }
507         }
508         else
509             status.factors = status.options.factorsInit;
510     }
511     else // produce estimate factors using uniform distribution with entries in [0,1].
512         makeFactors(mv.tnsDims, status.options.constraints, mv.rank, status.factors);
513 }
514
515 Status operator()(Options const &options)
516 {
517     Status          status(options);
518     Member_Variables mv(options.rank, options.tnsDims);
519
520     long long int fileSize = (TnsSize + 1) * options.nonZeros;
521
522     // Begin Load Balancing
523     Matrix          Balanced_Ratings_Base_T(TnsSize + 1,
options.nonZeros);
524     std::array<std::vector<long int>, TnsSize> perm_tns_indices;
525
526     Matrix Ratings_Base_T = Matrix(static_cast<int>(TnsSize+1), options.nonZeros);
527     // Read the whole Tensor from a file
528     read( options.ratings_path,
529         fileSize,
530         0,
531         Ratings_Base_T );
532
533     BalanceDataset<TnsSize>(options.nonZeros, options.tnsDims, Ratings_Base_T,
perm_tns_indices, Balanced_Ratings_Base_T);
534
535     // GTC_Base<TnsSize>::sort_ratings_base(Ratings_Base_T, mv.tnsX, options.tnsDims,
options.nonZeros);
536     GTC_Base<TnsSize>::sort_ratings_base(Balanced_Ratings_Base_T, mv.tnsX, options.tnsDims,
options.nonZeros);
537     Ratings_Base_T.resize(0,0);
538     Balanced_Ratings_Base_T.resize(0,0);
539
540     for (std::size_t i=0; i<TnsSize; i++)
541     {
542         mv.tnsX[i].makeCompressed();
543     }
544 }
545
546
547

```

```

558     calculate_offsets(mv);
559
560     initialize_factors(mv, status);
561
562     PermuteFactors<TnsSize>(status.factors, perm_tns_indices, mv.factors_T);
563
564     aogtc(mv, status);
565
566     // IF Depermute ....
567
568     return status;
569 }
570
571 Status operator()(Matrix          const &Ratings_Base_T,
572                  Options         const &options)
573 {
574     Status          status(options);
575     Member_Variables mv(options.rank, options.tnsDims);
576
577     // Begin Load Balancing
578     Matrix          Balanced_Ratings_Base_T(TnsSize + 1,
options.nonZeros);
579     std::array<std::vector<long int>, TnsSize> perm_tns_indices;
580
581     BalanceDataset<TnsSize>(options.nonZeros, options.tnsDims, Ratings_Base_T,
perm_tns_indices, Balanced_Ratings_Base_T);
582
583     ReserveSparseTensor<TnsSize>(mv.tnsX, options.tnsDims, options.nonZeros);
584     // FillSparseTensor<TnsSize>(mv.tnsX, options.nonZeros, Ratings_Base_T, options.tnsDims);
585     FillSparseTensor<TnsSize>(mv.tnsX, options.nonZeros, Balanced_Ratings_Base_T,
options.tnsDims);
586     // Ratings_Base_T.resize(0,0);
587     Balanced_Ratings_Base_T.resize(0,0);
588
589     for (std::size_t i=0; i<TnsSize; i++)
590     {
591         mv.tnsX[i].makeCompressed();
592     }
593
594     calculate_offsets(mv);
595
596     // produce estimate factors using uniform distribution with entries in [0,1].
597     initialize_factors(mv, status);
598
599     PermuteFactors<TnsSize>(status.factors, perm_tns_indices, mv.factors_T);
600
601     aogtc(mv, status);
602
603     // IF Depermute ....
604
605     return status;
606 }
607 } // namespace internal
608 } // namespace v1
609 } // end namespace partensor
610
611 #if USE_MPI
612 #include "GtcMpi.hpp"
613 #endif /* USE_MPI */
614
615 #if USE_OPENMP
616 #include "GtcOpenMP.hpp"
617 #endif /* USE_OPENMP */
618
619 namespace partensor
620 {
621     template <std::size_t _TnsSize, typename ExecutionPolicy>
622     execution::internal::enable_if_execution_policy<ExecutionPolicy,SparseStatus<_TnsSize,execution::execution_policy_t<ExecutionPolicy>,SparseDefaultValues>
623     gtc( ExecutionPolicy
624         &&,
625         SparseOptions<_TnsSize,execution::execution_policy_t<ExecutionPolicy>,SparseDefaultValues>
626         const &options )
627     {
628         using ExPolicy = execution::execution_policy_t<ExecutionPolicy>;
629
630         if constexpr (std::is_same_v<ExPolicy,execution::sequenced_policy>)
631         {
632             return internal::GTC<_TnsSize>(options);
633         }
634         else if constexpr (std::is_same_v<ExPolicy,execution::openmpi_policy>)
635         {
636             return internal::GTC<_TnsSize,execution::openmpi_policy>(options);
637         }
638     }
639 }

```

```

668     else if constexpr (std::is_same_v<ExPolicy,execution::openmp_policy>)
669     {
670         return internal::GTC<_TnsSize,execution::openmp_policy>()(options);
671     }
672     else
673         return internal::GTC<_TnsSize>()(options);
674 }
675
676 /*
677  * Interface of General Tensor Completion(gtc). Sequential Policy.
678  *
679  * @tparam Tensor_      Type(data type and order) of input Tensor.
680  *                      @c Tensor_ must be @c partensor::Tensor<order>, where
681  *                      @c order must be in range of @c [3-8].
682  * @param tnsX         [in] The given Tensor to be factorized of @c Tensor_ type,
683  *                      with @c double data.
684  * @param R            [in] The rank of decomposition.
685  *
686  * @returns An object of type @c Status, containing the results of the algorithm.
687  */
688 template<std::size_t _TnsSize>
689 auto gtc(SparseOptions<_TnsSize> const &options )
690 {
691     return internal::GTC<_TnsSize,execution::sequenced_policy>()(options);
692 }
693
694 template <std::size_t _TnsSize, typename ExecutionPolicy>
695
696 execution::internal::enable_if_execution_policy<ExecutionPolicy,SparseStatus<_TnsSize,execution::execution_policy_t<ExecutionPolicy>,SparseDefaultValues>
697 gtc( ExecutionPolicy
698     &&,
699     Matrix
700     const &Ratings_Base_T,
701     SparseOptions<_TnsSize,execution::execution_policy_t<ExecutionPolicy>,SparseDefaultValues>
702     const &options )
703 {
704     using ExPolicy = execution::execution_policy_t<ExecutionPolicy>;
705
706     if constexpr (std::is_same_v<ExPolicy,execution::sequenced_policy>)
707     {
708         return internal::GTC<_TnsSize>()(Ratings_Base_T,options);
709     }
710     else if constexpr (std::is_same_v<ExPolicy,execution::openmpi_policy>)
711     {
712         return internal::GTC<_TnsSize,execution::openmpi_policy>()(Ratings_Base_T,options);
713     }
714     else if constexpr (std::is_same_v<ExPolicy,execution::openmp_policy>)
715     {
716         return internal::GTC<_TnsSize,execution::openmp_policy>()(Ratings_Base_T,options);
717     }
718     else
719         return internal::GTC<_TnsSize>()(Ratings_Base_T,options);
720 }
721
722 /*
723  * Interface of General Tensor Completion(gtc). Sequential Policy.
724  *
725  * @tparam Tensor_      Type(data type and order) of input Tensor.
726  *                      @c Tensor_ must be @c partensor::Tensor<order>, where
727  *                      @c order must be in range of @c [3-8].
728  * @param tnsX         [in] The given Tensor to be factorized of @c Tensor_ type,
729  *                      with @c double data.
730  * @param R            [in] The rank of decomposition.
731  *
732  * @returns An object of type @c Status, containing the results of the algorithm.
733  */
734 template<std::size_t _TnsSize>
735 auto gtc(Matrix
736     SparseOptions<_TnsSize>
737     const &Ratings_Base_T,
738     const &options )
739 {
740     return internal::GTC<_TnsSize,execution::sequenced_policy>()(Ratings_Base_T,options);
741 }
742 }
743
744 #endif // PARTENSOR_GTC_HPP

```

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.](#)

```

1 #ifndef DOXYGEN_SHOULD_SKIP_THIS
2 #endif // DOXYGEN_SHOULD_SKIP_THIS
3 / *****/
4
5 #if !defined(PARTENSOR_GTC_HPP)
6 #error "GTC_MPI can only included inside GTC"
7 #endif // * PARTENSOR_GTC_HPP/
8
9 namespace partensor
10 {
11
12     inline namespace v1 {
13
14         namespace internal {
15             template<std::size_t TnsSize_>
16             struct GTC<TnsSize_,execution::openmpi_policy> : public GTC_Base<TnsSize_>
17             {
18                 using GTC_Base<TnsSize_>::TnsSize;
19                 using GTC_Base<TnsSize_>::lastFactor;
20                 using typename GTC_Base<TnsSize_>::Dimensions;
21                 using typename GTC_Base<TnsSize_>::MatrixArray;
22                 using typename GTC_Base<TnsSize_>::DataType;
23                 using typename GTC_Base<TnsSize_>::SparseTensor;
24                 using typename GTC_Base<TnsSize_>::LongMatrix;
25
26                 using IntArray = typename SparseTensorTraits<SparseTensor>::IntArray;
27                 using CartCommunicator = partensor::cartesian_communicator; // From ParallelWrapper.hpp
28                 using CartCommVector = std::vector<CartCommunicator>;
29                 using IntVector = std::vector<int>;
30                 using Int2DVector = std::vector<std::vector<int>>;
31
32                 using Options = partensor::SparseOptions<TnsSize_,execution::openmpi_policy,SparseDefaultValues>;
33                 using Status = partensor::SparseStatus<TnsSize_,execution::openmpi_policy,SparseDefaultValues>;
34
35                 // Variables that will be used in gtc implementations.
36                 struct Member_Variables
37                 {
38                     MPI_Communicator &world = Partensor()->MpiCommunicator(); // MPI_COMM_WORLD
39
40                     double local_f_value;
41                     int RxR;
42                     int world_size;
43
44                     Int2DVector displs_subTns; // skipping dimension "rows" for each subtensor
45                     Int2DVector displs_subTns_R; // skipping dimension "rows" for each subtensor times R (
46                     for MPI communication purposes )
47                     Int2DVector subTnsDims; // dimensions of subtensor
48                     Int2DVector subTnsDims_R; // dimensions of subtensor times R ( for MPI communication
49                     purposes )
50                     Int2DVector displs_local_update; // displacement in the local factor for update rows
51
52                     Int2DVector send_recv_counts; // rows to be communicated after update times R
53
54                     CartCommVector layer_comm;
55                     CartCommVector fiber_comm;
56
57                     IntArray layer_rank;
58                     IntArray fiber_rank;
59                     IntArray rows_for_update;
60                     IntArray tnsDims;
61
62                     MatrixArray layer_factors;
63                     MatrixArray layer_factors_T;
64                     MatrixArray factors;
65                     MatrixArray factors_T;
66                     MatrixArray factor_T_factor;
67                     MatrixArray local_factors_T;
68                     MatrixArray norm_factors_T;
69                     MatrixArray old_factors_T;
70
71                     Matrix cwise_factor_product;
72                     Matrix temp_matrix;
73
74                 };
75
76             };
77
78         };
79
80     };
81
82 };

```

```

96     Matrix      Ratings_Base_T;
97     SparseTensor subTns;
98
99     int          rank;
100    std::array<std::array<int, TnsSize-1>, TnsSize> offsets;
101
102    /*
103     * Calculates if the number of processors given from terminal
104     * are equal to the processors in the implementation.
105     *
106     * @param procs [in] @c stl array with the number of processors per
107     *                  dimension of the tensor.
108     */
109    void check_processor_availability(std::array<int, TnsSize> const &procs)
110    {
111        // MPI_Environment &env = Partensor()->MpiEnvironment();
112        world_size = world.size();
113        // numprocs must be product of options.proc_per_mode
114        if (std::accumulate(procs.begin(), procs.end(), 1,
115                          std::multiplies<int>()) != world_size && world.rank() == 0) {
116            Partensor()->Logger()->error("The product of the processors per mode must be equal to
117 {} nn", world_size);
118            // env.abort(-1);
119        }
120    }
121
122    Member_Variables() = default;
123    Member_Variables(int R, IntArray dims, std::array<int, TnsSize> &procs) : local_f_value(0.0),
124                                                                                               RxR(R* R),
125
126    displs_subTns(TnsSize),
127    displs_subTns_R(TnsSize),
128    subTnsDims(TnsSize),
129    subTnsDims_R(TnsSize),
130    displs_local_update(TnsSize),
131    send_rcv_counts(TnsSize),
132
133    {
134        check_processor_availability(procs);
135        layer_comm.reserve(TnsSize);
136        fiber_comm.reserve(TnsSize);
137    }
138
139    Member_Variables(Member_Variables const &) = default;
140    Member_Variables(Member_Variables &&) = default;
141
142    Member_Variables &operator=(Member_Variables const &) = default;
143    Member_Variables &operator=(Member_Variables &&) = default;
144 };
145
146 template<int mode>
147 void sort_ratings_base_util(Matrix const &Ratings_Base_T,
148                             long int const nnz,
149                             Member_Variables &mv)
150 {
151     Matrix ratings_base_temp = Ratings_Base_T;
152     std::vector<std::vector<double>> vectorized_ratings_base;
153     vectorized_ratings_base.resize(nnz, std::vector<double>(TnsSize + 1));
154
155     for (int rows = 0; rows < static_cast<int>(TnsSize) + 1; rows++)
156     {
157         for (int cols = 0; cols < nnz; cols++)
158         {
159             vectorized_ratings_base[cols][rows] = Ratings_Base_T(rows, cols);
160         }
161     }
162
163     // Sort
164     std::sort(vectorized_ratings_base.begin(), vectorized_ratings_base.end(), SortRows<TnsSize_,
165 mode, double>);
166
167     for (int rows = 0; rows < static_cast<int>(TnsSize) + 1; rows++)
168     {
169         for (int cols = 0; cols < nnz; cols++)
170         {
171             ratings_base_temp(rows, cols) = vectorized_ratings_base[cols][rows];
172         }
173     }
174
175     Dist_NNZ_sorted<TnsSize>(mv.subTns, nnz, mv.displs_subTns, mv.fiber_rank, ratings_base_temp,
176 mv.subTnsDims, mode);

```

```

174     mv.subTns[mode].makeCompressed();
175
176     if constexpr (mode+1 < TnsSize)
177         sort_ratings_base_util<mode+1>(Ratings_Base_T, nnz, mv);
178 }
179
180 void sort_ratings_base(Matrix          const &Ratings_Base_T,
181                       long int       const nnz,
182                       Member_Variables &mv)
183 {
184     ReserveSparseTensor<TnsSize>(mv.subTns, mv.subTnsDims, mv.fiber_rank, mv.world_size, nnz);
185     sort_ratings_base_util<0>(Ratings_Base_T, nnz, mv);
186 }
187
188 void NesterovMMLS(Member_Variables          &mv,
189                  Status                    const &st,
190                  int                      const idx,
191                  Matrix                    &MTTKRP_T)
192 {
193     double L, mu, q, alpha, new_alpha, beta, lambda;
194     int iter = 0;
195
196     Matrix grad_Y_T(mv.rank, mv.subTnsDims[idx][mv.fiber_rank[idx]]);
197     Matrix grad_Y_local_T(mv.rank, mv.rows_for_update[idx]);
198
199     Matrix Y_T(mv.rank, mv.subTnsDims[idx][mv.fiber_rank[idx]]);
200     Matrix Y_local_T(mv.rank, mv.rows_for_update[idx]);
201
202     Matrix new_A(mv.rank, mv.rows_for_update[idx]);
203     Matrix A(mv.rank, mv.rows_for_update[idx]);
204     Matrix Zero_Matrix = Matrix::Zero(mv.rank, mv.rows_for_update[idx]);
205
206     ComputeEIG(mv.cwise_factor_product, L, mu);
207
208     lambda = st.options.lambdas[idx];
209     L       = L + lambda;
210     q       = lambda / L;
211     alpha   = 1;
212
213     A       = mv.local_factors_T[idx];
214     Y_T     = mv.layer_factors_T[idx]; // layer_factor
215     Y_local_T = A;
216
217     int last_mode = (idx == static_cast<int>(TnsSize) - 1) ? static_cast<int>(TnsSize) - 2 :
static_cast<int>(TnsSize) - 1;
218
219     Matrix temp_R_1(mv.rank, 1);
220     while (1)
221     {
222         grad_Y_T.setZero();
223
224         if (iter >= st.options.max_nesterov_iter)
225         {
226             break;
227         }
228
229         // Compute grad_Y
230         Matrix temp_col = Matrix::Zero(mv.rank, 1);
231         for (long int i = 0; i < mv.subTns[idx].outerSize(); ++i)
232         {
233             temp_col.setZero();
234             for (SparseMatrix::InnerIterator it(mv.subTns[idx], i); it; ++it)
235             {
236                 temp_R_1 = Matrix::Ones(mv.rank, 1);
237                 // Select rows of each factor an compute the respective row of the Khatri-Rao product.
238                 for (int mode_i = last_mode, kr_counter = static_cast<int>(TnsSize) - 2; mode_i >= 0 &&
kr_counter >= 0; mode_i--)
239                 {
240                     if (mode_i == idx)
241                     {
242                         continue;
243                     }
244                     int row;
245                     row = ((it.row()) / mv.offsets[idx][kr_counter]) %
mv.subTnsDims[mode_i][mv.fiber_rank[mode_i]];
246                     temp_R_1 = temp_R_1.cwiseProduct(mv.layer_factors_T[mode_i].col(row));
247                     kr_counter--;
248                 }
249                 // Computation of row of Z according the relation (10) of the paper.
250                 temp_col += (temp_R_1.transpose() * Y_T.col(i))(0) * temp_R_1;
251             }
252             grad_Y_T.col(i) = temp_col;
253         }
254
255         // Add each process' results and scatter the block rows among the processes in the layer.
256         MPI_Reduce_scatter(grad_Y_T.data(), grad_Y_local_T.data(), send_rcv_counts, MPI_DOUBLE,
MPI_SUM, mode_layer_comm);

```



```

257         v2::reduce_scatter( mv.layer_comm[idx],
258                           grad_Y_T,
259                           mv.send_recv_counts[idx][0],
260                           grad_Y_local_T );
261
262         // Add proximal term.
263         grad_Y_local_T += MTTKRP_T + lambda * Y_local_T;
264         new_A          = (Y_local_T - grad_Y_local_T / L).cwiseMax(Zero_Matrix);
265
266         new_alpha = UpdateAlpha(alpha, q);
267         beta      = alpha * (1 - alpha) / (alpha * alpha + new_alpha);
268
269         Y_local_T = (1 + beta) * new_A - beta * A;
270
271         // The updated block rows of Y are all gathered, and we have the whole updated Y of the
layer.
272         // MPI_Allgather(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
273         v2::all_gather( mv.layer_comm[idx],
274                       Y_local_T,
275                       mv.send_recv_counts[idx][mv.layer_rank[idx]],
276                       mv.send_recv_counts[idx][0],
277                       mv.displs_local_update[idx][0],
278                       Y_T );
279
280         A      = new_A;
281         alpha  = new_alpha;
282         iter++;
283     }
284     mv.local_factors_T[idx] = A;
285
286 }
287
288 void NesterovMNLS_localL(Member_Variables &mv,
289                          Status          &st,
290                          int            const &idx,
291                          Matrix         const &MTTKRP_T)
292 {
293     int iter = 0;
294
295     double L2;
296     double sqrt_q = 0, beta = 0;
297     double lambda = st.options.lambdas[idx];
298
299     Matrix inv_L2(mv.subTns[idx].outerSize(), 1); // rows_layer
300
301     Matrix grad_Y_T(mv.rank, mv.subTnsDims[idx][mv.fiber_rank[idx]]);
302     Matrix grad_Y_local_T(mv.rank, mv.rows_for_update[idx]);
303
304     Matrix Y_T(mv.rank, mv.subTnsDims[idx][mv.fiber_rank[idx]]);
305     Matrix Y_local_T(mv.rank, mv.rows_for_update[idx]);
306
307     Matrix new_A(mv.rank, mv.rows_for_update[idx]);
308     Matrix A(mv.rank, mv.rows_for_update[idx]);
309
310     const Matrix zero_vec = Matrix::Zero(mv.rank, 1);
311
312     A      = mv.local_factors_T[idx];
313     Y_T    = mv.layer_factors_T[idx]; // layer_factor
314     Y_local_T = A;
315
316     int last_mode = (idx == static_cast<int>(TnsSize) - 1) ? static_cast<int>(TnsSize) - 2 :
static_cast<int>(TnsSize) - 1;
317
318     Matrix temp_R_1(mv.rank, 1);
319     Matrix temp_RxR(mv.rank, mv.rank);
320
321     while (1)
322     {
323         grad_Y_T.setZero();
324
325         if (iter >= st.options.max_nesterov_iter)
326         {
327             break;
328         }
329
330         // Compute grad_Y
331         Matrix temp_col = Matrix::Zero(mv.rank, 1);
332         for (long int i = 0; i < mv.subTns[idx].outerSize(); ++i)
333         {
334             temp_col.setZero();
335             if (iter < 1)
336             {
337                 temp_RxR.setZero();
338             }
339             for (SparseMatrix::InnerIterator it(mv.subTns[idx], i); it; ++it)
340             {
341                 temp_R_1 = Matrix::Ones(mv.rank, 1);

```

```

341         // Select rows of each factor and compute the respective row of the Khatri-Rao product.
342         for (int mode_i = last_mode, kr_counter = static_cast<int>(TnsSize) - 2; mode_i >= 0 &&
kr_counter >= 0; mode_i--)
343         {
344             if (mode_i == idx)
345             {
346                 continue;
347             }
348             long long int row;
349             row = ((it.row()) / mv.offsets[idx][kr_counter]) %
mv.subTnsDims[mode_i][mv.fiber_rank[mode_i]];
350             temp_R_1 = temp_R_1.cwiseProduct(mv.layer_factors_T[mode_i].col(row));
351             kr_counter--;
352         }
353         // Computation of row of Z according the relation (10) of the paper.
354         temp_col += (temp_R_1.transpose() * Y_T.col(i))(0) * temp_R_1;
355
356         // Compute only once!
357         if (iter < 1)
358         {
359             temp_RxR.noalias() += (temp_R_1 * temp_R_1.transpose());
360         }
361     }
362     grad_Y_T.col(i) = temp_col;
363
364     if (iter < 1)
365     {
366         // Communicate only once!
367         all_reduce( mv.layer_comm[idx],
368                   inplace(temp_RxR.data()),
369                   mv.RxR,
370                   std::plus<double>() );
371
372         L2 = PowerMethod(temp_RxR, 1e-3);
373         L2 += lambda;
374         inv_L2(i) = 1 / L2;
375     }
376 }
377
378 // Add each process' results and scatter the block rows among the processes in the layer.
379 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 );
380
381 // Add proximal term.
382 grad_Y_local_T += MTTKRP_T + lambda * Y_local_T;
383
384 for (long int i=0; i<mv.rows_for_update[idx]; i++)
385 {
386     long int translate_i = i + mv.displs_local_update[idx][mv.layer_rank[idx]]/mv.rank;
387
388     new_A.col(i) = (Y_local_T.col(i) - grad_Y_local_T.col(i) *
389                   inv_L2(translate_i)).cwiseMax(zero_vec);
390
391     sqrt_q = sqrt( lambda * inv_L2(translate_i) );
392     beta = (1 - sqrt_q) / (1 + sqrt_q);
393
394     // Update Y
395     Y_local_T.col(i) = (1 + beta) * new_A.col(i) - beta * A.col(i);
396 }
397
398 // The updated block rows of Y are all gathered, and we have the whole updated Y of the
399 layer.
400 v2::all_gatherv( mv.layer_comm[idx],
401                 Y_local_T,
402                 mv.send_recv_counts[idx][mv.layer_rank[idx]],
403                 mv.send_recv_counts[idx][0],
404                 mv.displs_local_update[idx][0],
405                 Y_T );
406
407 A = new_A;
408
409 iter++;
410 }
411 mv.local_factors_T[idx] = A;
412 }
413
414 / *
415 * In case option variable @c writeToFile is enabled then, before the end
416 * of the algorithm writes the resulted factors in files, where their
417 * paths are specified before compiling in @ options.final_factors_path.
418 *
419 * @param st [in] Struct where the returned values of @c Gtc are stored.
420 * /
421
422

```

```

423 void writeFactorsToFile(Status const &st)
424 {
425     std::size_t size;
426     for(std::size_t i=0; i<TnsSize; ++i)
427     {
428         size = st.factors[i].rows()          * st.factors[i].cols();
429         partensor::write(st.factors[i],
430                         st.options.final_factors_paths[i],
431                         size);
432     }
433 }
434
435 /*
436  * Compute the cost function value at the end of each outer iteration
437  * based on the last factor.
438  *
439  * @param grid_comm [in] MPI communicator where the new cost function value
440  * will be communicated and computed.
441  * @param mv [in] Struct where ALS variables are stored.
442  * @param st [in,out] Struct where the returned values of @c Gtc are stored.
443  * In this case the cost function value is updated.
444  */
445 void cost_function( CartCommunicator const &grid_comm,
446                   Member_Variables &mv,
447                   Status &st )
448 {
449     Matrix temp_R_1(mv.rank, 1);
450     double temp_1_1 = 0;
451     mv.local_f_value = 0;
452     std::array<int, TnsSize-1> offsets;
453     offsets[0] = 1;
454     for (int j = 1; j < static_cast<int>(TnsSize) - 1; j++)
455     {
456         offsets[j] = offsets[j - 1] * mv.subTnsDims[j-1][mv.fiber_rank[j-1]]; //
457         mv.layer_factors_T[j - 1].cols()
458     }
459     for (long int i = 0; i < mv.subTns[lastFactor].outerSize(); ++i)
460     {
461         int row;
462         for (SparseMatrix::InnerIterator it(mv.subTns[lastFactor], i); it; ++it)
463         {
464             temp_R_1 = mv.layer_factors_T[lastFactor].col(it.col());
465             // Select rows of each factor an compute the Hadamard product of the respective row
466             // of the Khatri-Rao product, and the row of factor A_N.
467             // temp_R_1 = A_N(i,N,:) . * ... . * A_2(i,2,:) . * A_1(i,1,:)
468             for (int mode_i = static_cast<int>(TnsSize) - 2; mode_i >= 0; mode_i--)
469             {
470                 row = ((it.row()) / offsets[mode_i]) %
471                     (mv.subTnsDims[mode_i][mv.fiber_rank[mode_i]]);
472                 temp_R_1.noalias() = temp_R_1.cwiseProduct(mv.layer_factors_T[mode_i].col(row));
473                 temp_1_1 = it.value() - temp_R_1.sum();
474                 mv.local_f_value += temp_1_1 * temp_1_1;
475             }
476         }
477         all_reduce( grid_comm,
478                   mv.local_f_value,
479                   st.f_value,
480                   std::plus<double>() );
481     }
482 }
483 /*
484  * Compute the cost function value at the end of each outer iteration
485  * based on the last accelerated factor.
486  *
487  * @param grid_comm [in] MPI communicator where the new cost function value
488  * will be communicated and computed.
489  * @param mv [in] Struct where ALS variables are stored.
490  * @param st [in] Struct where the returned values of @c Gtc are stored.
491  * In this case the cost function value is updated.
492  * @param factors [in] Accelerated factors.
493  * @param factors_T_factors [in] Gramian matrices of factors.
494  *
495  * @returns The cost function calculated with the accelerated factors.
496  */
497 double accel_cost_function(CartCommunicator const &grid_comm,
498                           Member_Variables const &mv,
499                           MatrixArray const &layer_factors_T)
500 {
501     Matrix temp_R_1(mv.rank, 1);
502     double temp_1_1 = 0;
503     double f_value = 0;
504
505     std::array<int, TnsSize-1> offsets;
506     offsets[0] = 1;

```

```

507     for (int j = 1; j < static_cast<int>(TnsSize) - 1; j++)
508     {
509         offsets[j] = offsets[j - 1]          * mv.subTnsDims[j-1][mv.fiber_rank[j-1]];
510     }
511
512     for (long int i = 0; i < mv.subTns[lastFactor].outerSize(); ++i)
513     {
514         int row;
515         for (SparseMatrix::InnerIterator it(mv.subTns[lastFactor], i); it; ++it)
516         {
517             temp_R_1 = layer_factors_T[lastFactor].col(it.col());
518             // Select rows of each factor and compute the Hadamard product of the respective row of
the Khatri-Rao product, and the row of factor A_N.
519             // temp_R_1 = A_N(i,N,:) . * ... . * A_2(i,2,:) . * A_1(i,1,:);
520             for (int mode_i = static_cast<int>(TnsSize) - 2; mode_i >= 0; mode_i--)
521             {
522                 row = ((it.row()) / offsets[mode_i]) %
(mv.subTnsDims[mode_i][mv.fiber_rank[mode_i]]);
523                 temp_R_1.noalias() = temp_R_1.cwiseProduct(layer_factors_T[mode_i].col(row));
524             }
525             temp_1_1 = it.value() - temp_R_1.sum();
526             f_value += temp_1_1 * temp_1_1;
527         }
528     }
529
530     all_reduce( grid_comm,
531               inplace(&f_value),
532               1,
533               std::plus<double>() );
534
535     return f_value;
536 }
537
538 / *
539 * Make use of the dimensions and the number of processors per dimension
540 * and then calculates the dimensions of the subtensor and subfactor for
541 * each processor.
542 *
543 * @tparam Dimensions          Array type containing the length of Tensor's dimensions.
544 *
545 * @param tnsDims [in]        Tensor Dimensions. Each index contains the corresponding
546 *                             factor's rows length.
547 * @param st [in]            Struct where the returned values of @c Gtc are stored.
548 * @param R [in]            The rank of decomposition.
549 * @param mv [in,out]       Struct where ALS variables are stored.
550 *
551 * Updates @c stl arrays with dimensions for subtensors and
552 * subfactors.
553 */
554 void compute_sub_dimensions(Status const &st,
555                            Member_Variables &mv)
556 {
557     for (std::size_t i = 0; i < TnsSize; ++i)
558     {
559         mv.factor_T_factor[i].noalias() = st.factors[i].transpose() * st.factors[i];
560
561         DisCount(mv.displs_subTns[i], mv.subTnsDims[i], st.options.proc_per_mode[i], mv.tnsDims[i],
1);
562         // for fiber communication and Gather
563         DisCount(mv.displs_subTns_R[i], mv.subTnsDims_R[i], st.options.proc_per_mode[i],
mv.tnsDims[i], static_cast<int>(mv.rank));
564         // information per layer
565         DisCount(mv.displs_local_update[i], mv.send_recv_counts[i], mv.world_size /
st.options.proc_per_mode[i],
566                 mv.subTnsDims[i][mv.fiber_rank[i]],
567                 static_cast<int>(mv.rank));
568
569         mv.rows_for_update[i] = mv.send_recv_counts[i][mv.layer_rank[i]] /
570                 static_cast<int>(mv.rank);
571     }
572
573     calculate_offsets(mv);
574 }
575
576 void calculate_offsets(Member_Variables &mv)
577 {
578     for (int idx = 0; idx < static_cast<int>(TnsSize); idx++)
579     {
580         mv.offsets[idx][0] = 1;
581         for (int j = 1, mode = 0; j < static_cast<int>(TnsSize) - 1; j++, mode++)
582         {
583             if (idx == mode)
584             {
585                 mode++;
586             }
587             mv.offsets[idx][j] = mv.offsets[idx][j - 1] * mv.subTnsDims[mode][mv.fiber_rank[mode]];
588         }
589     }
590 }

```

```

587     }
588
589     /*
590     * Based on each factor's constraint, a different
591     * update function is used at every outer iteration.
592     *
593     * Computes also factor^T * factor at the end.
594     *
595     * @param idx [in] Factor to be updated.
596     * @param R [in] The rank of decomposition.
597     * @param st [in] Struct where the returned values of @c Gtc are stored.
598     * Here constraints and options variables are needed.
599     * @param mv [in,out] Struct where ALS variables are stored.
600     * Updates the factors of each layer.
601     */
602     void update_factor(int          const idx,
603                       Status      const &st,
604                       Member_Variables &mv )
605     {
606         switch ( st.options.constraints[idx] )
607         {
608             case Constraint::unconstrained:
609             case Constraint::symmetric:
610             {
611                 // std::cout << " Inside symmetric update factor ... " << std::endl;
612                 Matrix A(mv.rank, mv.subTnsDims[idx][mv.fiber_rank[idx]]);
613                 Matrix A_local(mv.rank, mv.rows_for_update[idx]);
614                 Matrix eye = st.options.lambdas[idx] * Matrix::Identity(mv.rank, mv.rank);
615
616                 // int first_mode = (idx == 0) ? 1 : 0;
617                 int last_mode = (idx == static_cast<int>(TnsSize) - 1) ? static_cast<int>(TnsSize) - 2 :
static_cast<int>(TnsSize) - 1;
618
619                 Matrix MTTKRP_col(mv.rank, 1);
620                 Matrix temp_RxR(mv.rank, mv.rank);
621                 Matrix temp_R_1(mv.rank, 1);
622
623                 // Compute MTTKRP
624                 for (long long int i = 0; i < mv.subTns[idx].outerSize(); ++i)
625                 {
626                     MTTKRP_col.setZero();
627                     temp_RxR.setZero(); // temp_RxR : is the Hadamard product of Grammians of the Factors,
that correspond to the nnz elements of the Tensor.
628                     for (SparseMatrix::InnerIterator it(mv.subTns[idx], i); it; ++it)
629                     {
630                         temp_R_1 = Matrix::Ones(mv.rank, 1);
631                         long long int row;
632                         // Select rows of each factor an compute the respective row of the Khatri-Rao product.
633                         for (int mode_i = last_mode, kr_counter = static_cast<int>(TnsSize) - 2; mode_i >= 0
&& kr_counter >= 0; mode_i--)
634                         {
635                             if (mode_i == idx)
636                             {
637                                 continue;
638                             }
639                             row = ((it.row()) / mv.offsets[idx][kr_counter]) %
(mv.subTnsDims[mode_i][mv.fiber_rank[mode_i]]);
640
641                             temp_R_1 = temp_R_1.cwiseProduct(mv.layer_factors_T[mode_i].col(row));
642                             kr_counter--;
643                         }
644                         // Subtract from the previous row the respective row of W, according to relation (9).
645                         MTTKRP_col.noalias() += it.value() * temp_R_1;
646
647                         temp_RxR.noalias() += temp_R_1 * temp_R_1.transpose();
648                     }
649
650                     all_reduce( mv.layer_comm[idx],
651                               inplace(MTTKRP_col.data()),
652                               1 * mv.rank,
653                               std::plus<double>() );
654
655                     all_reduce( mv.layer_comm[idx],
656                               inplace(temp_RxR.data()),
657                               mv.RxR,
658                               std::plus<double>() );
659
660                     A.col(i) = ((temp_RxR + eye).inverse()) * MTTKRP_col;
661                 }
662                 // std::cout << " Inside symmetric update factor After loop ... " << std::endl;
663                 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]);
664
665                 break;
666             }
667             case Constraint::nonnegativity:

```

```

669         case Constraint::symmetric_nonnegativity:
670         {
671             int last_mode = (idx == static_cast<int>(TnsSize) - 1) ? static_cast<int>(TnsSize) - 2 :
static_cast<int>(TnsSize) - 1;
672
673             Matrix local_MTTKRP_T = Matrix::Zero(mv.rank, mv.rows_for_update[idx]);
674             Matrix MTTKRP_T      = Matrix::Zero(mv.rank, mv.subTnsDims[idx][mv.fiber_rank[idx]]);
675
676             // Compute MTTKRP
677             SparseMTTKRP<TnsSize>(mv.subTnsDims, mv.fiber_rank, mv.subTns[idx], mv.layer_factors_T,
mv.rank, mv.offsets[idx], last_mode, idx, MTTKRP_T);
678
679             // Add each process' results and scatter the block rows among the processes in the layer.
680             // MPI_Reduce_scatter(MTTKRP_T.data(), local_MTTKRP_T.data(), send_recv_counts,
MPI_DOUBLE, MPI_SUM, mode_layer_comm);
681             v2::reduce_scatter( mv.layer_comm[idx],
682                               MTTKRP_T,
683                               mv.send_recv_counts[idx][0],
684                               local_MTTKRP_T );
685
686             // NesterovMNLS(mv, st, idx, local_MTTKRP_T);
687             NesterovMNLS_localL(mv, st, idx, local_MTTKRP_T);
688
689             break;
690         }
691         case Constraint::sparsity:
692             break;
693         default: // in case of Constraint::constant
694             break;
695     } // end of constraints switch
696
697     // std::cout << " Inside update factor After switch ... " << std::endl;
698     if (st.options.constraints[idx] != Constraint::constant)
699     {
700         v2::all_gatherv( mv.layer_comm[idx],
701                       mv.local_factors_T[idx],
702                       mv.send_recv_counts[idx][mv.layer_rank[idx]],
703                       mv.send_recv_counts[idx][0],
704                       mv.displs_local_update[idx][0],
705                       mv.layer_factors_T[idx] );
706
707         mv.layer_factors[idx]          = mv.layer_factors_T[idx].transpose();
708
709         if (st.options.constraints[idx] == Constraint::symmetric_nonnegativity ||
st.options.constraints[idx] == Constraint::symmetric)
710         {
711             for (std::size_t i=0; i<TnsSize; i++)
712             {
713                 if (i != static_cast<std::size_t>(idx))
714                 {
715                     mv.layer_factors_T[i] = mv.layer_factors_T[idx];
716                 }
717             }
718         }
719
720         mv.factor_T_factor[idx].noalias() = mv.layer_factors_T[idx]          * mv.layer_factors[idx];
721     }
722
723     // std::cout << " Inside update factor .... switch ... " << std::endl;
724
725     all_reduce( mv.fiber_comm[idx],
726               inplace(mv.factor_T_factor[idx].data()),
727               mv.RxR,
728               std::plus<double>() );
729 }
730
731 /*
732  * At the end of the algorithm processor 0
733  * collects each part of the factor that each
734  * processor holds and return them in status.factors.
735  *
736  * @param mv      [in]      Struct where ALS variables are stored.
737  *                  Use variables to compute result factors by gathering each
738  *                  part of the factor from processors.
739  * @param st      [in,out]  Struct where the returned values of @c Gtc are stored.
740  *                  Stores the resulted factors.
741  */
742 void gather_final_factors(Member_Variables      &mv,
743                          Status                &st)
744 {
745     for(std::size_t i=0; i<TnsSize; ++i)
746     {
747         mv.temp_matrix.resize(static_cast<int>(mv.rank), mv.tnsDims[i]);
748         // Gather from all processors to processor with rank 0 the final factors
749         v2::gather( mv.fiber_comm[i],
750                   mv.layer_factors_T[i],

```

```

752             mv.subTnsDims_R[i][mv.fiber_rank[i]],
753             mv.subTnsDims_R[i][0],
754             mv.displs_subTns_R[i][0],
755             0,
756             mv.temp_matrix );
757
758         st.factors[i] = mv.temp_matrix.transpose();
759     }
760 }
761
762 /*
763  * @brief Line Search Acceleration
764  *
765  * Performs an acceleration step in the updated factors, and keeps the accelerated factors when
766  * the step succeeds. Otherwise, the acceleration step is ignored.
767  * Line Search Acceleration reduces the number outer iterations in the ALS algorithm.
768  *
769  * @note This implementation ONLY, if factors are of @c Matrix type.
770  *
771  * @param grid_comm [in] MPI communicator where the new cost function value
772  * will be communicated and computed.
773  * @param mv [in,out] Struct where ALS variables are stored.
774  * In case the acceration is successful layer factor^T * factor
775  * and layer factor variables are updated.
776  * @param st [in,out] Struct where the returned values of @c Gtc are stored.
777  * If the acceleration succeeds updates cost function value.
778  *
779  */
780 void line_search_accel(CartCommunicator const &grid_comm,
781                     Member_Variables &mv,
782                     Status &st)
783 {
784     double f_accel = 0.0; // Objective Value after the acceleration step
785     double const accel_step = pow(st.ao_iter+1,(1.0/(st.options.accel_coeff)));
786
787     MatrixArray accel_factors_T;
788     MatrixArray accel_gramians;
789
790     for(std::size_t i=0; i<TnsSize; ++i)
791     {
792         accel_factors_T[i] = mv.old_factors_T[i] + accel_step * (mv.layer_factors_T[i] -
793 mv.old_factors_T[i]);
794         accel_gramians[i] = accel_factors_T[i] * accel_factors_T[i].transpose();
795         all_reduce( mv.fiber_comm[i],
796                   inplace(accel_gramians[i].data()),
797                   mv.RxR,
798                   std::plus<double>() );
799     }
800
801     f_accel = accel_cost_function(grid_comm, mv, accel_factors_T);
802     if (st.f_value > f_accel)
803     {
804         mv.layer_factors_T = accel_factors_T;
805         mv.factor_T_factor = accel_gramians;
806         st.f_value = f_accel;
807         if(grid_comm.rank() == 0)
808             Partensor()->Logger()->info("Acceleration Step SUCCEEDED! at iter: {}", st.ao_iter);
809     }
810     else
811         st.options.accel_fail++;
812
813     if (st.options.accel_fail==5)
814     {
815         st.options.accel_fail=0;
816         st.options.accel_coeff++;
817     }
818 }
819
820 /*
821  * Parallel implementation of als method with MPI.
822  *
823  * @tparam Dimensions Array type containing the Tensor dimensions.
824  *
825  * @param grid_comm [in] The communication grid, where the processors
826  * communicate their cost function.
827  * @param tnsDims [in] Tensor Dimensions. Each index contains the corresponding
828  * factor's rows length.
829  * @param R [in] The rank of decomposition.
830  * @param mv [in] Struct where ALS variables are stored and being updated
831  * until a termination condition is true.
832  * @param status [in,out] Struct where the returned values of @c Gtc are stored.
833  */
834 void aogtc(CartCommunicator const &grid_comm,
835           Member_Variables &mv,
836           Status &status)
837 {
838     status.frob_tns = (mv.subTns[0]).squaredNorm();

```

```

838         all_reduce( grid_comm,
839                     inplace(&status.frob_tns),
840                     1,
841                     std::plus<double>() );
842
843     cost_function(grid_comm, mv, status);
844     status.rel_costFunction = status.f_value / status.frob_tns;
845
846     for (int i=0; i< static_cast<int>(TnsSize); i++)
847     {
848         mv.factor_T_factor[i].noalias() = mv.layer_factors_T[i] * mv.layer_factors[i];
849         all_reduce( mv.fiber_comm[i],
850                   inplace(mv.factor_T_factor[i].data()),
851                   mv.RxR,
852                   std::plus<double>() );
853     }
854
855     // Wait for all processors to reach here
856     grid_comm.barrier();
857
858     // ---- Loop until ALS converges ----
859     while(1)
860     {
861         status.ao_iter++;
862         if (!grid_comm.rank())
863         {
864             Partensor()->Logger()->info("iter: {} -- fvalue: {} -- relative_costFunction: {}",
865             status.ao_iter,
866                                     status.f_value, status.rel_costFunction);
867
868             std::cout << "iter: " << status.ao_iter << " - status.f_value: " << status.f_value << " -
869             status.rel_costFunction: " << status.rel_costFunction << std::endl;
870
871             // -----> <----- loop for every mode
872             // ----->
873             for (std::size_t i = 0; i < TnsSize; i++)
874             {
875                 // Compute hadamard of grammians to compute L.
876                 mv.cwise_factor_product = partensor::PartialCwiseProd(mv.factor_T_factor, i);
877
878                 // Partition rows of subfactor to the processes in the respective layer.
879                 mv.local_factors_T[i] = mv.layer_factors_T[i].block(0,
880                 mv.displs_local_update[i][mv.layer_rank[i]] / mv.rank, mv.rank, mv.rows_for_update[i]);
881
882                 update_factor(i, status, mv);
883             }
884
885             // ---- Cost function Computation ----
886             cost_function(grid_comm, mv, status);
887
888             status.rel_costFunction = status.f_value / status.frob_tns;
889
890             // if(status.options.normalization && !mv.all_orthogonal)
891             //     Normalize(mv.weight_factor, mv.rank, mv.factor_T_factor, mv.layer_factors);
892
893             // ---- Terminating condition ----
894             if (status.ao_iter >= status.options.max_iter)
895             {
896                 gather_final_factors(mv, status);
897                 if(grid_comm.rank() == 0)
898                 {
899                     std::cout << "status.rel_costFunction : " << status.rel_costFunction << std::endl;
900                     Partensor()->Logger()->info("Processor 0 collected all {} factors.          nn", TnsSize);
901                     if(status.options.writeToFile)
902                         writeFactorsToFile(status);
903                 }
904                 break;
905             }
906
907             if (status.options.acceleration)
908             {
909                 mv.norm_factors_T = mv.layer_factors_T;
910                 // ---- Acceleration Step ----
911                 if (status.ao_iter > 1)
912                     line_search_accel(grid_comm, mv, status);
913
914                 mv.old_factors_T = mv.norm_factors_T;
915             }
916         } // end of outer while loop
917     }
918
919     void initialize_factors(Member_Variables &mv,
920

```



```

924         Status          &status)
925     {
926         if(status.options.initialized_factors)
927         {
928             if(status.options.read_factors_from_file)
929             {
930                 for(std::size_t i=0; i<TnsSize; ++i)
931                 {
932                     status.factors[i] = Matrix(mv.tnsDims[i], static_cast<int>(mv.rank));
933                     read( status.options.initial_factors_paths[i],
934                         mv.tnsDims[i] * mv.rank,
935                         0,
936                         status.factors[i] );
937                 }
938             }
939             else
940                 status.factors = status.options.factorsInIt;
941         }
942         else
943             makeFactors(mv.tnsDims, status.options.constraints, mv.rank, status.factors);
944     }
945
946     Status operator()(Options const &options)
947     {
948         Status          status(options);
949         Member_Variables mv(options.rank, options.tnsDims, status.options.proc_per_mode);
950
951         // Communicator with cartesian topology
952         CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);
953
954         // Functions that create layer and fiber grids.
955         create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
956         create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);
957
958         // produce estimate factors using uniform distribution with entries in [0,1].
959         initialize_factors(mv, status);
960
961         compute_sub_dimensions(status, mv);
962
963         for (std::size_t i = 0; i < TnsSize; ++i)
964         {
965             mv.layer_factors[i] =
966             status.factors[i].block(mv.displs_subTns[i][mv.fiber_rank[i]], 0,
967                                   mv.subTnsDims[i][mv.fiber_rank[i]], mv.rank);
968             mv.layer_factors_T[i] = mv.layer_factors[i].transpose();
969             mv.factor_T_factor[i].noalias() = mv.layer_factors_T[i] * mv.layer_factors[i];
970         }
971
972         long long int fileSize = (TnsSize + 1) * options.nonZeros;
973
974         // Matrix Ratings_Base = Matrix(options.nonZeros, static_cast<int>(TnsSize+1));
975         Matrix Ratings_Base_T = Matrix(static_cast<int>(TnsSize+1), options.nonZeros);
976
977         // Read the whole Tensor from a file
978         read( options.ratings_path,
979             fileSize,
980             0,
981             Ratings_Base_T );
982
983         // Matrix Ratings_Base_T = Ratings_Base.transpose();
984
985         sort_ratings_base(Ratings_Base_T, options.nonZeros, mv);
986         Ratings_Base_T.resize(0,0);
987         // Ratings_Base.resize(0,0);
988
989         aogtc(grid_comm, mv, status);
990
991         return status;
992     }
993
994     Status operator()(Matrix          const &Ratings_Base_T,
995                     Options          const &options)
996     {
997         Status          status(options);
998         Member_Variables mv(options.rank, options.tnsDims, status.options.proc_per_mode);
999
1000        // Communicator with cartesian topology
1001        CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);
1002
1003        // Functions that create layer and fiber grids.
1004        create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
1005        create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);
1006
1007        // produce estimate factors using uniform distribution with entries in [0,1].
1008        initialize_factors(mv, status);
1009
1010        compute_sub_dimensions(status, mv);

```

```

1030
1031     // Begin Load Balancing
1032     // Matrix                                     Balanced_Ratings_Base_T(TnsSize + 1,
options.nonZeros);
1033     // std::array<std::vector<long int>, TnsSize> perm_tns_indices;
1034
1035     // BalanceDataset<TnsSize>(options.nonZeros, options.tnsDims, Ratings_Base_T,
perm_tns_indices, Balanced_Ratings_Base_T);
1036
1037     // PermuteFactors<TnsSize>(status.factors, perm_tns_indices, mv.factors_T);
1038
1039     std::cout << "After Balance....." << std::endl;
1040     for (std::size_t i = 0; i < TnsSize; ++i)
1041     {
1042         mv.layer_factors[i] =
status.factors[i].block(mv.displs_subTns[i][mv.fiber_rank[i]], 0,
1043         mv.subTnsDims[i][mv.fiber_rank[i]],
mv.rank);
1044         mv.layer_factors_T[i] = mv.layer_factors[i].transpose();
1045         mv.factor_T_factor[i].noalias() = mv.layer_factors_T[i] * mv.layer_factors[i];
1046
1047         // mv.layer_factors_T[i] = mv.factors_T[i].block(0, mv.displs_subTns[i][mv.fiber_rank[i]],
1048         // mv.subTnsDims[i][mv.fiber_rank[i]], mv.rank,
mv.subTnsDims[i][mv.fiber_rank[i]]);
1049         // mv.layer_factors[i] = mv.layer_factors_T[i].transpose();
1050         // mv.factor_T_factor[i].noalias() = mv.layer_factors_T[i] * mv.layer_factors[i];
1051     }
1052
1053     // Each processor takes a subtensor from tnsX
1054     ReserveSparseTensor<TnsSize>(mv.subTns, mv.subTnsDims, mv.fiber_rank, mv.world_size,
options.nonZeros);
1055
1056     Dist_NNZ<TnsSize>(mv.subTns, options.nonZeros, mv.displs_subTns, mv.fiber_rank,
Ratings_Base_T, mv.subTnsDims);
1057     // Dist_NNZ<TnsSize>(mv.subTns, options.nonZeros, mv.displs_subTns, mv.fiber_rank,
Bridged_Ratings_Base_T, mv.subTnsDims);
1058     // Ratings_Base_T.resize(0,0);
1059     // Bridged_Ratings_Base_T.resize(0,0);
1060
1061     for(int mode_i = 0; mode_i < static_cast<int>(TnsSize); mode_i++)
1062     {
1063         mv.subTns[mode_i].makeCompressed();
1064     }
1065
1066     aogtc(grid_comm, mv, status);
1067
1068     return status;
1069 }
1070
1071 };
1072 } // namespace internal
1073 } // namespace v1
1074
1075 } // 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.](#)

```

1 #ifndef DOXYGEN_SKIP_THIS
15 #endif // DOXYGEN_SKIP_THIS
16 /*****
25 #if !defined(PARTENSOR_GTC_HPP)
26 #error "GTC_OMP can only be included inside GTC"
27 #endif // * PARTENSOR_GTC_HPP
28
29 namespace partensor

```

```

30 {
31     inline namespace v1
32     {
33         namespace internal
34         {
35             template <std::size_t TnsSize_>
36             struct GTC<TnsSize_, execution::openmp_policy> : public GTC_Base<TnsSize_>
37             {
38                 using GTC_Base<TnsSize_>::TnsSize;
39                 using GTC_Base<TnsSize_>::lastFactor;
40                 using typename GTC_Base<TnsSize_>::Dimensions;
41                 using typename GTC_Base<TnsSize_>::MatrixArray;
42                 using typename GTC_Base<TnsSize_>::DataType;
43                 using typename GTC_Base<TnsSize_>::SparseTensor;
44                 using typename GTC_Base<TnsSize_>::IntArray;
45                 using typename GTC_Base<TnsSize_>::LongMatrix;
46
47                 using Options = partensor::SparseOptions<TnsSize_, execution::openmp_policy, SparseDefaultValues>;
48                 using Status = partensor::SparseStatus<TnsSize_, execution::openmp_policy, SparseDefaultValues>;
49
50                 // Variables that will be used in gtc implementations.
51                 struct Member_Variables
52                 {
53                     MatrixArray factors_T;
54                     MatrixArray factor_T_factor;
55                     MatrixArray mttkrp_T;
56                     IntArray tnsDims;
57                     std::array<std::array<int, TnsSize_ - 1>, TnsSize_> offsets;
58
59                     MatrixArray norm_factors_T;
60                     MatrixArray old_factors_T;
61
62                     Matrix cwise_factor_product;
63                     Matrix Ratings_Base_T;
64                     SparseTensor tnsX;
65
66                     // bool all_orthogonal = true;
67                     // int weight_factor;
68                     int rank;
69
70                     MatrixArray grad;
71                     MatrixArray Y;
72                     MatrixArray invL;
73
74                     Member_Variables() = default;
75
76                     Member_Variables(int R, IntArray dims) : tnsDims(dims),
77                                                             rank(R)
78                     {}
79
80                     Member_Variables(Member_Variables const &) = default;
81                     Member_Variables(Member_Variables &&) = default;
82
83                     Member_Variables &operator=(Member_Variables const &) = default;
84                     Member_Variables &operator=(Member_Variables &&) = default;
85                 };
86
87                 /*
88                 * In case option variable @c writeToFile is enabled, then, before the end
89                 * of the algorithm, it writes the resulted factors in files, whose
90                 * paths are specified before compiling in @ options.final_factors_path.
91                 *
92                 * @param st [in] Struct where the returned values of @c Gtc are stored.
93                 */
94                 void writeFactorsToFile(Status const &st)
95                 {
96                     std::size_t size;
97                     for(std::size_t i=0; i<TnsSize; ++i)
98                     {
99                         size = st.factors[i].rows() * st.factors[i].cols();
100                         partensor::write(st.factors[i],
101                                       st.options.final_factors_paths[i],
102                                       size);
103                     }
104                 }
105
106                 /*
107                 * Compute the cost function value at the end of each outer iteration
108                 * based on the last factor.
109                 *
110                 * @param mv [in] Struct where ALS variables are stored.
111                 * @param st [in,out] Struct where the returned values of @c Gtc are stored.
112                 * In this case the cost function value is updated.
113                 */
114                 void cost_function(Member_Variables const &mv,
115                                   Status &st)
116                 {

```

```

117     Matrix temp_R_1(mv.rank, 1);
118     double temp_1_1 = 0;
119     double f_value_loc = 0;
120
121     #pragma omp master
122     st.f_value = 0;
123
124     #pragma omp barrier
125
126     std::array<int,TnsSize-1> offsets;
127     offsets[0] = 1;
128     for (int j = 1; j < static_cast<int>(TnsSize) - 1; j++)
129     {
130         offsets[j] = offsets[j - 1] * mv.tnsDims[j-1];
131     }
132
133     #pragma omp for schedule(static)
134     for (long int i = 0; i < mv.tnsX[lastFactor].outerSize(); ++i)
135     {
136         int row = 0;
137         for (SparseMatrix::InnerIterator it(mv.tnsX[lastFactor], i); it; ++it)
138         {
139             temp_R_1 = mv.factors_T[lastFactor].col(it.col());
140             // 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.
141             for (int mode_i = static_cast<int>(TnsSize) - 2; mode_i >= 0; mode_i--)
142             {
143                 row = ((it.row()) / offsets[mode_i]) % (mv.tnsDims[mode_i]);
144                 temp_R_1.noalias() = temp_R_1.cwiseProduct(mv.factors_T[mode_i].col(row));
145             }
146             temp_1_1 = it.value() - temp_R_1.sum();
147             f_value_loc += temp_1_1 * temp_1_1;
148         }
149     }
150     #pragma omp atomic
151     st.f_value += f_value_loc;
152
153     #pragma omp barrier
154
155 }
156
157 /*
158 * Compute the cost function value at the end of each outer iteration
159 * based on the last accelerated factor.
160 *
161 * @param mv [in] Struct where ALS variables are stored.
162 * @param accel_factors [in] Accelerated factors.
163 *
164 * @returns The cost function calculated with the accelerated factors.
165 */
166 double accel_cost_function(Member_Variables const &mv,
167 MatrixArray const &accel_factors)
168 {
169     Matrix temp_R_1(mv.rank, 1);
170     double temp_1_1 = 0;
171     double f_value = 0;
172
173     std::array<int,TnsSize-1> offsets;
174     offsets[0] = 1;
175     for (int j = 1; j < static_cast<int>(TnsSize) - 1; j++)
176     {
177         offsets[j] = offsets[j - 1] * mv.tnsDims[j-1];
178     }
179
180     #pragma omp for schedule(static)
181     for (long int i = 0; i < mv.tnsX[lastFactor].outerSize(); ++i)
182     {
183         int row = 0;
184         for (SparseMatrix::InnerIterator it(mv.tnsX[lastFactor], i); it; ++it)
185         {
186             temp_R_1 = accel_factors[lastFactor].col(it.col());
187             // 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.
188             // temp_R_1 = A_N(i_N,:) . * ... . * A_2(i_2,:) . * A_1(i_1,:)
189             for (int mode_i = static_cast<int>(TnsSize) - 2; mode_i >= 0; mode_i--)
190             {
191                 row = ((it.row()) / offsets[mode_i]) % (mv.tnsDims[mode_i]);
192                 temp_R_1.noalias() = temp_R_1.cwiseProduct(accel_factors[mode_i].col(row));
193             }
194             temp_1_1 = it.value() - temp_R_1.sum();
195             f_value += temp_1_1 * temp_1_1;
196         }
197     }
198     return f_value;
199 }
200
201 void calculate_offsets(Member_Variables &mv)

```

```

202     {
203         for (int idx = 0; idx < static_cast<int>(TnsSize); idx++)
204         {
205             mv.offsets[idx][0] = 1;
206             for (int j = 1, mode = 0; j < static_cast<int>(TnsSize) - 1; j++, mode++)
207             {
208                 if (idx == mode)
209                 {
210                     mode++;
211                 }
212                 mv.offsets[idx][j] = mv.offsets[idx][j - 1] * mv.tnsDims[mode];
213             }
214         }
215     }
216
217     void unconstraint_update(int          const  idx,
218                             Member_Variables  &mv,
219                             Status            &st)
220     {
221         int r = mv.rank;
222
223         Matrix eye = st.options.lambdas[idx] * Matrix::Identity(r, r);
224
225         int last_mode = (idx == static_cast<int>(TnsSize) - 1) ? static_cast<int>(TnsSize) - 2 :
static_cast<int>(TnsSize) - 1;
226
227         Matrix MTTKRP_col(r, 1);
228         Matrix temp_RxR(r, r);
229         Matrix temp_R_1(r, 1);
230
231         // Compute MTTKRP
232         #pragma omp for schedule(dynamic) //nowait
233         for (long int i = 0; i < mv.tnsX[idx].outerSize(); ++i)
234         {
235             MTTKRP_col.setZero();
236             temp_RxR.setZero(); // is the Hadamard product of Grammians of the Factors, that
correspond to the nnz elements of the Tensor.
237             for (SparseMatrix::InnerIterator it(mv.tnsX[idx], i); it; ++it)
238             {
239                 temp_R_1 = Matrix::Ones(r, 1);
240                 int row;
241                 // Select rows of each factor an compute the respective row of the Khatri-Rao
product.
242                 for (int mode_i = last_mode, kr_counter = static_cast<int>(TnsSize) - 2; mode_i >= 0
&& kr_counter >= 0; mode_i--)
243                 {
244                     if (mode_i == idx)
245                     {
246                         continue;
247                     }
248                     row = ((it.row()) / mv.offsets[idx][kr_counter]) % (mv.tnsDims[mode_i]);
249                     temp_R_1 = temp_R_1.cwiseProduct(mv.factors_T[mode_i].col(row));
250                     kr_counter--;
251                 }
252                 // Subtract from the previous row the respective row of W, according to relation
(9).
253                 MTTKRP_col.noalias() += it.value() * temp_R_1;
254                 temp_RxR.noalias() += temp_R_1 * temp_R_1.transpose();
255             }
256             mv.factors_T[idx].col(i) = (temp_RxR + eye).inverse() * MTTKRP_col;
257         }
258     }
259
260     / *
261     * Based on each factor's constraint, a different
262     * update function is used at every outer iteration.
263     *
264     * Computes also factor^T * factor at the end.
265     *
266     * @param idx [in] Factor to be updated.
267     * @param mv [in] Struct where ALS variables are stored.
268     * @param st [in,out] Struct where the returned values of @c Gtc are stored.
269     *
270     * Updates the @c stl array with the factors.
271     */
272     void update_factor(int          const  idx,
273                       Member_Variables  &mv,
274                       Status            &st )
275     {
276         // Update factor
277         switch ( st.options.constraints[idx] )
278         {
279             case Constraint::unconstrained:
280             {
281                 unconstraint_update(idx, mv, st);
282                 break;
283             }
284             case Constraint::nonnegativity:

```

```

284     {
285         int last_mode = (idx == static_cast<int>(TnsSize) - 1) ? static_cast<int>(TnsSize) - 2 :
static_cast<int>(TnsSize) - 1;
286         SparseMTTKRP_omp(mv.tnsDims, mv.tnsX[idx], mv.factors_T, mv.rank, mv.offsets[idx],
last_mode, idx, mv.mttkrp_T[idx]);
287         #pragma omp barrier
288
289         // NesterovMNLs(mv.cwise_factor_product, mv.factors_T, mv.tnsDims, mv.tnsX[idx],
mv.offsets[idx], mv.Y[idx],
290         // st.options.max_nesterov_iter, st.options.lambdas[idx], idx,
st.options.constraints[idx], mv.mttkrp_T[idx]);
291
292         local_L::NesterovMNLs(mv.invL[idx], mv.factors_T, mv.tnsDims, mv.tnsX[idx],
mv.offsets[idx], mv.Y[idx],
293         st.options.max_nesterov_iter, st.options.lambdas[idx], idx,
mv.mttkrp_T[idx]);
294
295         break;
296     }
297     default: // in case of Constraint::constant
298         break;
299 }
300
301 // Compute A^T * A + B^T * B + ...
302 #pragma omp master
303 {
304     st.factors[idx] = mv.factors_T[idx].transpose();
305     mv.factor_T_factor[idx].noalias() = mv.factors_T[idx] * mv.factors_T[idx].transpose();
306 }
307 }
308
309 /*
310  * @brief Line Search Acceleration
311  *
312  * Performs an acceleration step on the updated factors, and keeps the accelerated factors
313  * when the step succeeds. Otherwise, the acceleration step is ignored.
314  * Line Search Acceleration reduces the number of outer iterations in the ALS algorithm.
315  *
316  * @note This implementation ONLY, if factors are of @c Matrix type.
317  *
318  * @param mv [in,out] Struct where ALS variables are stored.
319  *           In case the acceleration step is successful the Gramian
320  *           matrices of factors are updated.
321  * @param st [in,out] Struct where the returned values of @c Gtc are stored.
322  *           If the acceleration succeeds updates @c factors
323  *           and cost function value.
324  *
325  */
326 void line_search_accel(Member_Variables &mv,
327                       Status           &st,
328                       double          &f_accel,
329                       double          &accel_step,
330                       MatrixArray    &accel_factors_T,
331                       MatrixArray    &accel_gramians)
332 {
333     #pragma omp master
334     {
335         for(std::size_t i=0; i<TnsSize; ++i)
336         {
337             accel_factors_T[i] = mv.old_factors_T[i] + accel_step * (mv.factors_T[i] -
mv.old_factors_T[i]);
338             accel_gramians[i] = accel_factors_T[i] * accel_factors_T[i].transpose();
339         }
340
341         f_accel = 0;
342     }
343
344     #pragma omp barrier
345
346     double f_accel_loc = accel_cost_function(mv, accel_factors_T);
347
348     #pragma omp atomic
349     f_accel += f_accel_loc;
350
351     #pragma omp barrier
352
353     #pragma omp master
354     {
355         if (st.f_value > f_accel)
356         {
357             mv.factors_T = accel_factors_T;
358             mv.factor_T_factor = accel_gramians;
359             st.f_value = f_accel;
360             Partensor()->Logger()->info("Acceleration Step SUCCEEDED! at iter: {}", st.ao_iter);
361         }
362         else
363             st.options.accel_fail++;

```

```

364
365     if (st.options.accel_fail==5)
366     {
367         st.options.accel_fail=0;
368         st.options.accel_coeff++;
369     }
370 }
371 }
372
373 /*
374 * Sequential implementation of Alternating Least Squares (ALS) method.
375 *
376 * @param R [in] The rank of decomposition.
377 * @param mv [in] Struct where ALS variables are stored and being updated
378 *               until a termination condition is true.
379 * @param st [in,out] Struct where the returned values of @c Gtc are stored.
380 */
381 void aogtc(Member_Variables &mv,
382           Status &status)
383 {
384     double f_accel = 0.0; // Objective Value after the acceleration step
385     double accel_step = 0.0;
386
387     MatrixArray accel_factors_T;
388     MatrixArray accel_gramians;
389
390     for (std::size_t i=0; i<TnsSize; i++)
391     {
392         mv.Y[i] = Matrix::Zero(mv.rank, mv.tnsDims[i]);
393         mv.factors_T[i] = status.factors[i].transpose();
394         mv.mttkrp_T[i] = Matrix(mv.rank, mv.tnsDims[i]);
395         mv.factor_T_factor[i].noalias() = mv.factors_T[i] * status.factors[i];
396         accel_factors_T[i] = mv.factors_T[i];
397         accel_gramians[i] = Matrix::Zero(mv.rank, mv.rank);
398         mv.invL[i] = Matrix::Zero(mv.tnsDims[i], 1);
399     }
400
401     // if(status.options.normalization)
402     // {
403     //     choose_normilization_factor(status, mv.all_orthogonal, mv.weight_factor);
404     // }
405     // Normalize(static_cast<int>(R), mv.factor_T_factor, status.factors);
406
407     const int total_num_threads = get_num_threads();
408     omp_set_nested(0);
409
410     status.frob_tns = (mv.tnsX[0]).squaredNorm();
411
412     #pragma omp parallel n
413     num_threads(total_num_threads) n
414     proc_bind(spread) n
415     default(shared) n
416     shared(status, mv)
417     {
418         cost_function(mv, status);
419         #pragma omp barrier
420
421         #pragma omp master
422         {
423             status.rel_costFunction = status.f_value / status.frob_tns;
424         }
425         #pragma omp barrier
426
427         // ---- Loop until ALS converges ----
428         while(1)
429         {
430             #pragma omp master
431             {
432                 status.ao_iter++;
433                 Partensor()->Logger()->info("iter: {} -- fvalue: {} -- relative_costFunction: {}",
434 status.ao_iter,
435                                     status.f_value, status.rel_costFunction);
436
437                 std::cout << "iter : " << status.ao_iter << " - fvalue : " << status.f_value << " -
438 relative_costFunction : " << status.rel_costFunction << std::endl;
439                 // status.f_value = 0;
440             }
441             #pragma omp barrier
442             for (std::size_t i=0; i<TnsSize; i++)
443             {
444                 #pragma omp master
445                 {
446                     mv.cwise_factor_product = PartialCwiseProd(mv.factor_T_factor, i);
447                 }
448                 #pragma omp barrier
449                 // Update factor

```

```

449         update_factor(i, mv, status);
450         #pragma omp barrier
451     }
452
453     #pragma omp barrier
454     // Avg Factors if symmetric tensor
455     if (status.options.averaging && status.ao_iter >= 1)
456     {
457         #pragma omp master
458         {
459             for (int i = 1; i < static_cast<int>(TnsSize); i++)
460             {
461                 mv.factors_T[0].noalias() += mv.factors_T[i];
462             }
463
464             mv.factors_T[0].noalias() = mv.factors_T[0] / TnsSize;
465
466             for (int i = 0; i < static_cast<int>(TnsSize); i++)
467             {
468                 mv.factors_T[i] = mv.factors_T[0];
469
470                 status.factors[i] = mv.factors_T[i].transpose();
471                 mv.factor_T_factor[i].noalias() = mv.factors_T[i]           * mv.factors_T[i].transpose();
472             }
473         }
474     }
475     #pragma omp barrier
476
477     cost_function(mv, status);
478     #pragma omp master
479     {
480         status.rel_costFunction = status.f_value / status.frob_tns;
481     }
482     #pragma omp barrier
483
484     // -----
485     // if(status.options.normalization && !mv.all_orthogonal)
486     // Normalize(mv.weight_factor, static_cast<int>(R), mv.factor_T_factor,
status.factors);
487
488     // ---- Terminating condition ----
489     if (status.rel_costFunction < status.options.threshold_error || status.ao_iter >=
status.options.max_iter)
490     {
491         #pragma omp master
492         {
493             if (status.options.writeToFile)
494                 writeFactorsToFile(status);
495         }
496         break;
497     }
498     #pragma omp barrier // DON'T REMOVE!
499
500     if (status.options.acceleration)
501     {
502         // ---- Acceleration Step ----
503         if (status.ao_iter > 1)
504         {
505             #pragma omp master
506             accel_step = pow(status.ao_iter+1,(1.0/(status.options.accel_coef)));
507
508             line_search_accel(mv, status, f_accel, accel_step, accel_factors_T,
accel_gramians);
509
510             #pragma omp barrier
511         }
512         // Averaging
513         #pragma omp master
514         {
515             for (int i = 1; i < static_cast<int>(TnsSize); i++)
516             {
517                 mv.factors_T[0].noalias() += mv.factors_T[i];
518             }
519
520             mv.factors_T[0].noalias() = mv.factors_T[0] / TnsSize;
521
522             for (int i = 0; i < static_cast<int>(TnsSize); i++)
523             {
524                 mv.factors_T[i] = mv.factors_T[0];
525
526                 status.factors[i] = mv.factors_T[i].transpose();
527                 mv.factor_T_factor[i].noalias() = mv.factors_T[i]           * mv.factors_T[i].transpose();
528
529                 if (status.options.acceleration)
530                 {
531                     mv.old_factors_T[i] = mv.factors_T[i];
532                 }
533             }
534         }
535     }

```



```

532         }
533     }
534     #pragma omp master
535     {
536         for (int i = 0; i < static_cast<int>(TnsSize); i++)
537             mv.old_factors_T[i] = mv.factors_T[i];
538     }
539     #pragma omp barrier
540 }
541 } // end of while
542 } // end of pragma
543 }
544
545 void initialize_factors(Member_Variables &mv,
546                       Status          &status)
547 {
548     if(status.options.initialized_factors)
549     {
550         if(status.options.read_factors_from_file)
551         {
552             for(std::size_t i=0; i<TnsSize; ++i)
553             {
554                 status.factors[i] = Matrix(mv.tnsDims[i], static_cast<int>(mv.rank));
555                 read( status.options.initial_factors_paths[i],
556                     mv.tnsDims[i] * mv.rank,
557                     0,
558                     status.factors[i] );
559             }
560         }
561         else
562             status.factors = status.options.factorsInit;
563     }
564     else // produce estimate factors using uniform distribution with entries in [0,1].
565         makeFactors(mv.tnsDims, status.options.constraints, mv.rank, status.factors);
566 }
567
568 Status operator()(Options const &options)
569 {
570     Status          status(options);
571     Member_Variables mv(options.rank, options.tnsDims);
572
573     long long int fileSize = (TnsSize + 1) * options.nonZeros;
574
575     Matrix Ratings_Base_T = Matrix(static_cast<int>(TnsSize+1), options.nonZeros);
576     // Read the whole Tensor from a file
577     read( options.ratings_path,
578         fileSize,
579         0,
580         Ratings_Base_T );
581
582     // GTC_Base<TnsSize>::sort_ratings_base(mv.Ratings_Base_T, options.nonZeros);
583     // ReserveSparseTensor<TnsSize>(mv.tnsX, options.tnsDims, options.nonZeros);
584     // FillSparseTensor<TnsSize>(mv.tnsX, options.nonZeros, mv.Ratings_Base_T, options.tnsDims);
585     // mv.Ratings_Base_T.resize(0,0);
586     GTC_Base<TnsSize>::sort_ratings_base(Ratings_Base_T, mv.tnsX, options.tnsDims,
587     options.nonZeros);
588     Ratings_Base_T.resize(0,0);
589
590     for (std::size_t i=0; i<TnsSize; i++)
591     {
592         mv.tnsX[i].makeCompressed();
593     }
594
595     calculate_offsets(mv);
596
597     initialize_factors(mv, status);
598
599     aogtc(mv, status);
600
601     return status;
602 }
603
604 Status operator()(Matrix          const &Ratings_Base_T,
605                  Options          const &options)
606 {
607     Status          status(options);
608     Member_Variables mv(options.rank, options.tnsDims);
609
610     // ReserveSparseTensor<TnsSize>(mv.tnsX, options.tnsDims, options.nonZeros);
611     // FillSparseTensor<TnsSize>(mv.tnsX, options.nonZeros, Ratings_Base_T, options.tnsDims);
612     // Ratings_Base_T.resize(0,0);
613     GTC_Base<TnsSize>::sort_ratings_base(Ratings_Base_T, mv.tnsX, options.tnsDims,
614     options.nonZeros);
615     // Ratings_Base_T.resize(0,0);
616
617     for (std::size_t i=0; i<TnsSize; i++)
618     {

```

```

640         mv.tnsX[i].makeCompressed();
641     }
642
643     calculate_offsets(mv);
644
645     // produce estimate factors using uniform distribution with entries in [0,1].
646     initialize_factors(mv, status);
647
648     partensor::timer.startChronoHighTimer();
649     aogtc(mv, status);
650     double end_gtc_time_omp = partensor::timer.endChronoHighTimer();
651     std::cout << "GtcOpenMP took " << end_gtc_time_omp << " sec." << std::endl;
652     return status;
653 }
654 };
655 } // namespace internal
656 } // namespace v1
657 } // 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 <code>partensor::Tensor < order ></code> , 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 <code>partensor::Tensor < order ></code> , 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.](#)

```

1 #ifndef DOXYGEN_SHOULD_SKIP_THIS
15 #endif // DOXYGEN_SHOULD_SKIP_THIS
16 / ..... /
26 #ifndef PARTENSOR_GTC_STOCHASTIC_HPP
27 #define PARTENSOR_GTC_STOCHASTIC_HPP
28
29 #include "PARTENSOR_basic.hpp"
30 #include "PartialCwiseProd.hpp"
31 #include "NesterovMMLS.hpp"
32 #include "Normalize.hpp"
33 #include "Timers.hpp"
34 #include "ReadWrite.hpp"
35
36 namespace partensor
37 {
38     inline namespace v1
39     {
40         namespace internal
41         {
42             /*
43              * Includes the implementation of Stochastic General Tensor Completion. Based on the given
44              * parameters one of the overloaded operators will be called.
45              */
46             template <std::size_t TnsSize_>
47             struct GTC_STOCHASTIC_Base
48             {
49                 static constexpr std::size_t TnsSize = TnsSize_;
50                 static constexpr std::size_t lastFactor = TnsSize - 1;
51                 using SparseTensor = typename partensor::SparseTensor<TnsSize_>;
52                 using DataType = typename SparseTensorTraits<SparseTensor>::DataType;
53                 using MatrixType = typename SparseTensorTraits<SparseTensor>::MatrixType;
54                 using Dimensions = typename SparseTensorTraits<SparseTensor>::Dimensions;
55                 using SparseMatrix = typename SparseTensorTraits<SparseTensor>::SparseMatrixType;
56                 using LongMatrix = typename SparseTensorTraits<SparseTensor>::LongMatrixType;
57                 using Constraints = typename SparseTensorTraits<SparseTensor>::Constraints;
58                 using MatrixArray = typename SparseTensorTraits<SparseTensor>::MatrixArray;
59                 using DoubleArray = typename SparseTensorTraits<SparseTensor>::DoubleArray;
60                 using IntArray = typename SparseTensorTraits<SparseTensor>::IntArray;
61                 template<int mode>
62                 void sort_ratings_base_util(Matrix const &Ratings_Base_T,
63                                             SparseTensor &tnsX,
64                                             IntArray const &tnsDims,
65                                             long int const nnz)
66                 {
67                     Matrix ratings_base_temp = Ratings_Base_T;
68                     std::vector<std::vector<double>> vectorized_ratings_base;
69                     vectorized_ratings_base.resize(nnz, std::vector<double>(TnsSize + 1));
70
71                     for (int rows = 0; rows < static_cast<int>(TnsSize) + 1; rows++)
72                     {
73                         for (int cols = 0; cols < nnz; cols++)
74                         {
75                             vectorized_ratings_base[cols][rows] = Ratings_Base_T(rows, cols);
76                         }
77                     }
78                     // Sort
79                     std::sort(vectorized_ratings_base.begin(), vectorized_ratings_base.end(), SortRows<TnsSize_,
80 mode, double>);
81
82                     for (int rows = 0; rows < static_cast<int>(TnsSize) + 1; rows++)
83                     {
84                         for (int cols = 0; cols < nnz; cols++)
85                         {
86                             ratings_base_temp(rows, cols) = vectorized_ratings_base[cols][rows];
87                         }
88                     }

```

```

89     }
90   }
91
92   FillSparseMatricization<TnsSize>(tnsX, nnz, ratings_base_temp, tnsDims, mode);
93
94   if constexpr (mode+1 < TnsSize)
95     sort_ratings_base_util<mode+1>(Ratings_Base_T, tnsX, tnsDims, nnz);
96 }
97
98 void sort_ratings_base(Matrix          const &Ratings_Base_T,
99                      SparseTensor    &tnsX,
100                      IntArray        const &tnsDims,
101                      long int        const nnz)
102 {
103   ReserveSparseTensor<TnsSize>(tnsX, tnsDims, nnz);
104   sort_ratings_base_util<0>(Ratings_Base_T, tnsX, tnsDims, nnz);
105 }
106 };
107
108 template <std::size_t TnsSize_, typename ExecutionPolicy = execution::sequenced_policy>
109 struct GTC_STOCHASTIC : public GTC_STOCHASTIC_Base<TnsSize_>
110 {
111   using GTC_STOCHASTIC_Base<TnsSize_>::TnsSize;
112   using GTC_STOCHASTIC_Base<TnsSize_>::lastFactor;
113   using typename GTC_STOCHASTIC_Base<TnsSize_>::Dimensions;
114   using typename GTC_STOCHASTIC_Base<TnsSize_>::MatrixArray;
115   using typename GTC_STOCHASTIC_Base<TnsSize_>::DataType;
116   using typename GTC_STOCHASTIC_Base<TnsSize_>::SparseTensor;
117   using typename GTC_STOCHASTIC_Base<TnsSize_>::IntArray;
118   using typename GTC_STOCHASTIC_Base<TnsSize_>::LongMatrix;
119
120   using Options =
121   partensor::SparseOptions<TnsSize_, execution::sequenced_policy, SparseDefaultValues>;
122   using Status =
123   partensor::SparseStatus<TnsSize_, execution::sequenced_policy, SparseDefaultValues>;
124
125   // Variables that will be used in gtc stochastic implementations.
126   struct Member_Variables
127   {
128     MatrixArray  factors_T;
129     MatrixArray  factor_T_factor;
130     MatrixArray  mttkrp_T;
131     IntArray     tnsDims;
132     IntArray     blocksize;
133     std::array<std::array<int, TnsSize_ -1>, TnsSize_> offsets;
134
135     MatrixArray  norm_factors_T;
136     MatrixArray  old_factors_T;
137
138     Matrix      cwise_factor_product;
139     SparseTensor tnsX;
140     int         rank;
141     double      c_stochastic_perc;
142
143     Member_Variables() = default;
144     Member_Variables(int R, IntArray dims) : tnsDims(dims),
145                                             rank(R)
146     {}
147     Member_Variables(Member_Variables const &) = default;
148     Member_Variables(Member_Variables &&) = default;
149     Member_Variables &operator=(Member_Variables const &) = default;
150     Member_Variables &operator=(Member_Variables &&) = default;
151   };
152
153   /*
154   * In case option variable @c writeToFile is enabled, then, before the end
155   * of the algorithm, it writes the resulted factors in files, whose
156   * paths are specified before compiling in @ options.final_factors_path.
157   *
158   * @param st [in] Struct where the returned values of @c GtcStochastic are stored.
159   */
160   void writeFactorsToFile(Status const &st)
161   {
162     std::size_t size;
163     for(std::size_t i=0; i<TnsSize; ++i)
164     {
165       size = st.factors[i].rows() * st.factors[i].cols();
166       partensor::write(st.factors[i],
167                      st.options.final_factors_paths[i],
168                      size);
169     }
170   }
171 }
172
173 /*

```

```

174     * Compute the cost function value at the end of each outer iteration
175     * based on the last factor.
176     *
177     * @param mv [in] Struct where ALS variables are stored.
178     * @param st [in,out] Struct where the returned values of @c GtcStochastic are stored.
179     * In this case the cost function value is updated.
180     */
181 void cost_function(Member_Variables const &mv,
182                  Status &st)
183 {
184     Matrix temp_R_1(mv.rank, 1);
185     double temp_1_1 = 0;
186     st.f_value = 0;
187
188     std::array<int,TnsSize-1> offsets;
189     offsets[0] = 1;
190     for (int j = 1; j < static_cast<int>(TnsSize) - 1; j++)
191     {
192         offsets[j] = offsets[j - 1] * mv.tnsDims[j-1];
193     }
194
195     for (long int i = 0; i < mv.tnsX[lastFactor].outerSize(); ++i)
196     {
197         int row = 0;
198         for (SparseMatrix::InnerIterator it(mv.tnsX[lastFactor], i); it; ++it)
199         {
200             temp_R_1 = mv.factors_T[lastFactor].col(it.col());
201             // Select rows of each factor and compute the Hadamard product of the respective row of
the Khatri-Rao product, and the row of factor A_N.
202             for (int mode_i = static_cast<int>(TnsSize) - 2; mode_i >= 0; mode_i--)
203             {
204                 row = ((it.row()) / offsets[mode_i]) % (mv.tnsDims[mode_i]);
205                 temp_R_1 = temp_R_1.cwiseProduct(mv.factors_T[mode_i].col(row));
206             }
207             temp_1_1 = it.value() - temp_R_1.sum();
208             st.f_value += temp_1_1 * temp_1_1;
209         }
210     }
211 }
212
213 /*
214 * Compute the cost function value at the end of each outer iteration
215 * based on the last accelerated factor.
216 *
217 * @param mv [in] Struct where ALS variables are stored.
218 * @param accel_factors [in] Accelerated factors.
219 *
220 * @returns The cost function calculated with the accelerated factors.
221 */
222 double accel_cost_function(Member_Variables const &mv,
223                           MatrixArray const &accel_factors)
224 {
225     Matrix temp_R_1(mv.rank, 1);
226     double temp_1_1 = 0;
227     double f_value = 0;
228
229     std::array<int,TnsSize-1> offsets;
230     offsets[0] = 1;
231     for (int j = 1; j < static_cast<int>(TnsSize) - 1; j++)
232     {
233         offsets[j] = offsets[j - 1] * mv.tnsDims[j-1];
234     }
235
236     for (long int i = 0; i < mv.tnsX[lastFactor].outerSize(); ++i)
237     {
238         int row = 0;
239         for (SparseMatrix::InnerIterator it(mv.tnsX[lastFactor], i); it; ++it)
240         {
241             temp_R_1 = accel_factors[lastFactor].col(it.col());
242             // Select rows of each factor and compute the Hadamard product of the respective row of
the Khatri-Rao product, and the row of factor A_N.
243             // temp_R_1 = A_N(i_N,:) . * ... . * A_2(i_2,:) . * A_1(i_1,:)
244             for (int mode_i = static_cast<int>(TnsSize) - 2; mode_i >= 0; mode_i--)
245             {
246                 row = ((it.row()) / offsets[mode_i]) % (mv.tnsDims[mode_i]);
247                 temp_R_1 = temp_R_1.cwiseProduct(accel_factors[mode_i].col(row));
248             }
249             temp_1_1 = it.value() - temp_R_1.sum();
250             f_value += temp_1_1 * temp_1_1;
251         }
252     }
253     return f_value;
254 }
255
256 void calculate_offsets(Member_Variables &mv)
257 {
258     for (int idx = 0; idx < static_cast<int>(TnsSize); idx++)

```

```

259     {
260         mv.offsets[idx][0] = 1;
261         for (int j = 1, mode = 0; j < static_cast<int>(TnsSize) - 1; j++, mode++)
262             {
263                 if (idx == mode)
264                     {
265                         mode++;
266                     }
267                 mv.offsets[idx][j] = mv.offsets[idx][j] - 1] * mv.tnsDims[mode];
268             }
269     }
270 }
271
272 void unconstraint_update(int          const idx,
273                         Member_Variables &mv,
274                         Status           &st)
275 {
276     int r = mv.rank;
277
278     Matrix eye = st.options.lambdas[idx] * Matrix::Identity(r, r);
279
280     int last_mode = (idx == static_cast<int>(TnsSize) - 1) ? static_cast<int>(TnsSize) - 2 :
static_cast<int>(TnsSize) - 1;
281
282     Matrix MTTKRP_col(r, 1);
283     Matrix temp_RxR(r, r);
284     Matrix temp_R_1(r, 1);
285
286     // Compute MTTKRP
287     for (long int i = 0; i < mv.tnsX[idx].outerSize(); ++i)
288     {
289         MTTKRP_col.setZero();
290         temp_RxR.setZero(); // is the Hadamard product of Grammians of the Factors, that
correspond to the nnz elements of the Tensor.
291         for (SparseMatrix::InnerIterator it(mv.tnsX[idx], i); it; ++it)
292             {
293                 temp_R_1 = Matrix::Ones(r, 1);
294                 int row;
295                 // Select rows of each factor an compute the respective row of the Khatri-Rao
product.
296                 for (int mode_i = last_mode, kr_counter = static_cast<int>(TnsSize) - 2; mode_i >= 0
&& kr_counter >= 0; mode_i--)
297                     {
298                         if (mode_i == idx)
299                             {
300                                 continue;
301                             }
302                         row = ((it.row()) / mv.offsets[idx][kr_counter]) % (mv.tnsDims[mode_i]);
303                         temp_R_1 = temp_R_1.cwiseProduct(mv.factors_T[mode_i].col(row));
304                         kr_counter--;
305                     }
306                 // Subtract from the previous row the respective row of W, according to relation
(9).
307                 MTTKRP_col.noalias() += it.value() * temp_R_1;
308                 temp_RxR.noalias() += temp_R_1 * temp_R_1.transpose();
309             }
310             mv.factors_T[idx].col(i) = (temp_RxR + eye).inverse() * MTTKRP_col;
311         }
312     }
313
314     /*
315     * Based on each factor's constraint, a different
316     * update function is used at every outer iteration.
317     *
318     * Computes also factor^T * factor at the end.
319     *
320     * @param idx [in] Factor to be updated.
321     * @param mv [in] Struct where ALS variables are stored.
322     * @param st [in,out] Struct where the returned values of @c GtcStochastic are stored.
323     * Updates the @c stl array with the factors.
324     */
325 void update_factor(int          const idx,
326                   Member_Variables &mv,
327                   Status           &st )
328 {
329     // Update factor
330     switch ( st.options.constraints[idx] )
331     {
332     case Constraint::unconstrained:
333         {
334             break;
335         }
336     case Constraint::nonnegativity:
337         {
338             dynamic_blocksize::StochasticNesterovMNLS(mv.factors_T, mv.tnsDims, mv.tnsX[idx],
mv.offsets[idx],
339                                                         mv.c_stochastic_perc,

```

```

340     st.options.max_nesterov_iter, st.options.lambdas[idx], idx);
341         break;
342     }
343     default: // in case of Constraint::constant
344         break;
345     }
346     // Compute  $A^T * A + B^T * B + \dots$ 
347     st.factors[idx] = mv.factors_T[idx].transpose();
348     mv.factor_T_factor[idx].noalias() = mv.factors_T[idx] * st.factors[idx];
349 }
350
351 / *
352  * @brief Line Search Acceleration
353  *
354  * Performs an acceleration step on the updated factors, and keeps the accelerated factors
355  * when the step succeeds. Otherwise, the acceleration step is ignored.
356  * Line Search Acceleration reduces the number of outer iterations in the ALS algorithm.
357  *
358  * @note This implementation ONLY, if factors are of @c Matrix type.
359  *
360  * @param mv [in,out] Struct where ALS variables are stored.
361  *           In case the acceleration step is successful the Gramian
362  *           matrices of factors are updated.
363  * @param st [in,out] Struct where the returned values of @c GtcStochastic are stored.
364  *           If the acceleration succeeds updates @c factors
365  *           and cost function value.
366  *
367  */
368 void line_search_accel(Member_Variables &mv,
369                       Status          &st)
370 {
371     double f_accel = 0.0; // Objective Value after the acceleration step
372     double const accel_step = pow(st.ao_iter+1,(1.0/(st.options.accel_coeff)));
373
374     MatrixArray accel_factors_T;
375     MatrixArray accel_gramians;
376
377     for(std::size_t i=0; i<TnsSize; ++i)
378     {
379         mv.old_factors_T[i] = mv.old_factors_T[i] + accel_step * (mv.factors_T[i] -
380         mv.old_factors_T[i]);
381         accel_gramians[i] = accel_factors_T[i] * accel_factors_T[i].transpose();
382     }
383
384     f_accel = accel_cost_function(mv, accel_factors_T);
385     if (st.f_value > f_accel)
386     {
387         mv.factors_T = accel_factors_T;
388         mv.factor_T_factor = accel_gramians;
389         st.f_value = f_accel;
390         Partensor()->Logger()->info("Acceleration Step SUCCEEDED! at iter: {}", st.ao_iter);
391     }
392     else
393     {
394         st.options.accel_fail++;
395     }
396     if (st.options.accel_fail==5)
397     {
398         st.options.accel_fail=0;
399         st.options.accel_coeff++;
400     }
401 }
402 / *
403  * Sequential implementation of Alternating Least Squares (ALS) method.
404  *
405  * @param R [in] The rank of decomposition.
406  * @param mv [in] Struct where ALS variables are stored and being updated
407  *               until a termination condition is true.
408  * @param st [in,out] Struct where the returned values of @c GtcStochastic are stored.
409  */
410 void aogtc_stochastic(Member_Variables &mv,
411                      Status          &status)
412 {
413     for (std::size_t i=0; i<TnsSize; i++)
414     {
415         mv.factors_T[i] = status.factors[i].transpose();
416         mv.factor_T_factor[i].noalias() = mv.factors_T[i] * status.factors[i];
417         mv.mttkrp_T[i] = Matrix(mv.rank, mv.tnsDims[i]);
418     }
419
420     // if(status.options.normalization)
421     // {
422     //     choose_normilization_factor(status, mv.all_orthogonal, mv.weight_factor);
423     // }
424     // Normalize(static_cast<int>(R), mv.factor_T_factor, status.factors);

```



```

425
426     status.frob_tns          = (mv.tnsX[0]).squaredNorm();
427     cost_function(mv, status);
428     status.rel_costFunction = status.f_value/status.frob_tns;
429
430     // ---- Loop until ALS converges ----
431     int epoch = int(mv.tnsX[0].nonZeros() / (mv.blocksize[0] * mv.tnsDims[0])) + 1;
432     std::cout << "epoch = " << epoch << std::endl;
433     std::size_t epoch_counter = 0;
434
435     while(1)
436     {
437         status.ao_iter++;
438         std::cout << "iter: " << status.ao_iter << " -- fvalue: " << status.f_value << " --
relative_costFunction: " << status.rel_costFunction << std::endl;
439         Partensor()->Logger()->info("iter: {} -- fvalue: {} -- relative_costFunction: {}",
status.ao_iter,
status.f_value, status.rel_costFunction);
440
441
442         for (std::size_t i=0; i<TnsSize; i++)
443         {
444             mv.cwise_factor_product = PartialCwiseProd(mv.factor_T_factor, i);
445
446             // Update factor
447             update_factor(i, mv, status);
448         }
449
450         // <----- End loop for every mode
451         // ----->
452         // Cost function computation.
453         if (status.ao_iter % epoch == 0)
454         {
455             epoch_counter++;
456             cost_function(mv, status);
457             status.rel_costFunction = status.f_value/status.frob_tns;
458             // std::cout << epoch_counter << " - " << f_value << " - " << f_value / frob_tns << " - " <<
frob_tns << "
nn";
459         }
460         // <----- Terminating condition ----->
461         // if (epoch_counter > max_iter || f_value < ao_tol)
462
463         // ---- Terminating condition ----
464         if (status.rel_costFunction < status.options.threshold_error || epoch_counter >=
status.options.max_iter )
465         {
466             if(status.options.writeToFile)
467                 writeFactorsToFile(status);
468             break;
469         }
470
471         // if (status.options.acceleration)
472         // {
473         //     mv.norm_factors_T = mv.factors_T;
474         //     // ---- Acceleration Step ----
475         //     if (status.ao_iter > 1)
476         //         line_search_accel(mv, status);
477
478         //     mv.old_factors_T = mv.norm_factors_T;
479         // }
480     } // end of while
481 }
482
483 void initialize_factors(Member_Variables &mv,
484                       Status &status)
485 {
486     if(status.options.initialized_factors)
487     {
488         if(status.options.read_factors_from_file)
489         {
490             for(std::size_t i=0; i<TnsSize; ++i)
491             {
492                 status.factors[i] = Matrix(mv.tnsDims[i], static_cast<int>(mv.rank));
493                 read( status.options.initial_factors_paths[i],
494                     mv.tnsDims[i] * mv.rank,
495                     0,
496                     status.factors[i] );
497             }
498         }
499         else
500             status.factors = status.options.factorsInIt;
501     }
502     else // produce estimate factors using uniform distribution with entries in [0,1].
503         makeFactors(mv.tnsDims, status.options.constraints, mv.rank, status.factors);
504 }
505
506
507
508

```

```

519     Status operator()(Options const &options)
520     {
521         Status          status(options);
522         Member_Variables mv(options.rank, options.tnsDims);
523
524         long long int fileSize = (TnsSize + 1)          * options.nonZeros;
525
526         Matrix Ratings_Base_T = Matrix(static_cast<int>(TnsSize+1), options.nonZeros);
527         // Read the whole Tensor from a file
528         read( options.ratings_path,
529             fileSize,
530             0,
531             Ratings_Base_T );
532
533         GTC_STOCHASTIC_Base<TnsSize>::sort_ratings_base(Ratings_Base_T, mv.tnsX, options.tnsDims,
options.nonZeros);
534         Ratings_Base_T.resize(0,0);
535
536         for (std::size_t i=0; i<TnsSize; i++)
537         {
538             mv.tnsX[i].makeCompressed();
539         }
540
541         mv.c_stochastic_perc = options.c_stochastic_perc;
542
543         // c := percentage (%) of dataset to be sampled.
544         // blocksize(mode) := c          * ( nnz / tns_dimensions(mode) )
545         for (std::size_t i = 0; i < TnsSize; i++)
546         {
547             mv.blocksize[i] = int(mv.c_stochastic_perc          * options.nnz / mv.tnsDims[i]) + 1;
548         }
549
550         calculate_offsets(mv);
551
552         initialize_factors(mv, status);
553
554         aogtc_stochastic(mv, status);
555
556         return status;
557     }
558
559     Status operator()(Matrix          const &Ratings_Base_T,
560                      Options          const &options)
561     {
562         Status          status(options);
563         Member_Variables mv(options.rank, options.tnsDims);
564
565         ReserveSparseTensor<TnsSize>(mv.tnsX, options.tnsDims, options.nonZeros);
566         FillSparseTensor<TnsSize>(mv.tnsX, options.nonZeros, Ratings_Base_T, options.tnsDims);
567         // Ratings_Base_T.resize(0,0);
568
569         for (std::size_t i=0; i<TnsSize; i++)
570         {
571             mv.tnsX[i].makeCompressed();
572         }
573
574         mv.c_stochastic_perc = options.c_stochastic_perc;
575
576         // c := percentage (%) of dataset to be sampled.
577         // blocksize(mode) := c          * ( nnz / tns_dimensions(mode) )
578         for (std::size_t i = 0; i < TnsSize; i++)
579         {
580             mv.blocksize[i] = int(mv.c_stochastic_perc          * options.nonZeros / mv.tnsDims[i]) + 1;
581         }
582
583         calculate_offsets(mv);
584
585         // produce estimate factors using uniform distribution with entries in [0,1].
586         initialize_factors(mv, status);
587
588         aogtc_stochastic(mv, status);
589
590         return status;
591     }
592 };
593 } // namespace internal
594 } // namespace v1
595 } // end namespace partensor
596
597 #if USE_MPI
598
599 #include "GtcStochasticMpi.hpp"
600 #endif /* USE_MPI */
601
602 #if USE_OPENMP
603
604 #include "GtcStochasticOpenMP.hpp"
605

```

```

615 #endif / * USE_OPENMP /
616
617 namespace partensor
618 {
638     template <std::size_t _TnsSize, typename ExecutionPolicy>
639         execution::internal::enable_if_execution_policy<ExecutionPolicy,SparseStatus<_TnsSize,execution::execution_policy_t<ExecutionPolicy>,SparseDefaultValues>
640         gtc_stochastic( ExecutionPolicy
641             &&,
642             SparseOptions<_TnsSize,execution::execution_policy_t<ExecutionPolicy>,SparseDefaultValues>
643             const &options )
644     {
645         using ExPolicy = execution::execution_policy_t<ExecutionPolicy>;
646         if constexpr (std::is_same_v<ExPolicy,execution::sequenced_policy>)
647         {
648             return internal::GTC_STOCHASTIC<_TnsSize>()(options);
649         }
650         else if constexpr (std::is_same_v<ExPolicy,execution::openmpi_policy>)
651         {
652             return internal::GTC_STOCHASTIC<_TnsSize,execution::openmpi_policy>()(options);
653         }
654         else if constexpr (std::is_same_v<ExPolicy,execution::openmp_policy>)
655         {
656             return internal::GTC_STOCHASTIC<_TnsSize,execution::openmp_policy>()(options);
657         }
658         else
659             return internal::GTC_STOCHASTIC<_TnsSize>()(options);
660     }
661
662     / *
663     * Interface of Stochastic General Tensor Completion(gtc_stochastic). Sequential Policy.
664     * @tparam Tensor_      Type(data type and order) of input Tensor.
665     *                      @c Tensor_ must be @c partensor::Tensor<order>, where
666     *                      @c order must be in range of @c [3-8].
667     * @param tnsX      [in] The given Tensor to be factorized of @c Tensor_ type,
668     *                  with @c double data.
669     * @param R          [in] The rank of decomposition.
670     *
671     * @returns An object of type @c Status, containing the results of the algorithm.
672     */
673     template<std::size_t _TnsSize>
674     auto gtc_stochastic(SparseOptions<_TnsSize> const &options )
675     {
676         return internal::GTC_STOCHASTIC<_TnsSize,execution::sequenced_policy>()(options);
677     }
678
679     template <std::size_t _TnsSize, typename ExecutionPolicy>
680         execution::internal::enable_if_execution_policy<ExecutionPolicy,SparseStatus<_TnsSize,execution::execution_policy_t<ExecutionPolicy>,SparseDefaultValues>
681         gtc_stochastic( ExecutionPolicy
682             &&,
683             Matrix
684             const &Ratings_Base_T,
685             SparseOptions<_TnsSize,execution::execution_policy_t<ExecutionPolicy>,SparseDefaultValues>
686             const &options )
687     {
688         using ExPolicy = execution::execution_policy_t<ExecutionPolicy>;
689         if constexpr (std::is_same_v<ExPolicy,execution::sequenced_policy>)
690         {
691             return internal::GTC_STOCHASTIC<_TnsSize>()(Ratings_Base_T,options);
692         }
693         else if constexpr (std::is_same_v<ExPolicy,execution::openmpi_policy>)
694         {
695             return internal::GTC_STOCHASTIC<_TnsSize,execution::openmpi_policy>()(Ratings_Base_T,options);
696         }
697         else if constexpr (std::is_same_v<ExPolicy,execution::openmp_policy>)
698         {
699             return internal::GTC_STOCHASTIC<_TnsSize,execution::openmp_policy>()(Ratings_Base_T,options);
700         }
701         else
702             return internal::GTC_STOCHASTIC<_TnsSize>()(Ratings_Base_T,options);
703     }
704
705     / *
706     * Interface of Stochastic General Tensor Completion(gtc_stochastic). Sequential Policy.
707     * @tparam Tensor_      Type(data type and order) of input Tensor.
708     *                      @c Tensor_ must be @c partensor::Tensor<order>, where
709     *                      @c order must be in range of @c [3-8].
710     * @param tnsX      [in] The given Tensor to be factorized of @c Tensor_ type,
711     *                  with @c double data.
712     * @param R          [in] The rank of decomposition.
713     *
714     * @returns An object of type @c Status, containing the results of the algorithm.
715     */
716
717     template<std::size_t _TnsSize>
718     auto gtc_stochastic(SparseOptions<_TnsSize> const &options )
719     {
720         return internal::GTC_STOCHASTIC<_TnsSize,execution::sequenced_policy>()(options);
721     }
722
723     / *
724     * Interface of Stochastic General Tensor Completion(gtc_stochastic). Sequential Policy.
725     * @tparam Tensor_      Type(data type and order) of input Tensor.
726     *                      @c Tensor_ must be @c partensor::Tensor<order>, where
727     *                      @c order must be in range of @c [3-8].
728     * @param tnsX      [in] The given Tensor to be factorized of @c Tensor_ type,
729     *                  with @c double data.
730     * @param R          [in] The rank of decomposition.
731     *
732     * @returns An object of type @c Status, containing the results of the algorithm.
733     */

```

```

733  */
734  template<std::size_t _TnsSize>
735  auto gtc_stochastic(Matrix          const &Ratings_Base_T,
736                    SparseOptions<_TnsSize> const &options )
737  {
738      return internal::GTC_STOCHASTIC<_TnsSize,execution::sequenced_policy>()(Ratings_Base_T,options);
739  }
740
741 }
742
743 #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.](#)

```

1  #ifndef DOXYGEN_SHOULD_SKIP_THIS
15 #endif // DOXYGEN_SHOULD_SKIP_THIS
16 / ***** /
25 #if !defined(PARTENSOR_GTC_STOCHASTIC_HPPP)
26 #error "GTC_STOCHASTIC_MPI can only included inside GTC"
27 #endif / * PARTENSOR_GTC_STOCHASTIC_HPPP
28
29 namespace partensor
30 {
31
32     inline namespace v1 {
33
34         namespace internal {
35             template<std::size_t TnsSize_>
36             struct GTC_STOCHASTIC<TnsSize_,execution::openmpi_policy> : public GTC_STOCHASTIC_Base<TnsSize_>
37             {
38                 using GTC_STOCHASTIC_Base<TnsSize_>::TnsSize;
39                 using GTC_STOCHASTIC_Base<TnsSize_>::lastFactor;
40                 using typename GTC_STOCHASTIC_Base<TnsSize_>::Dimensions;
41                 using typename GTC_STOCHASTIC_Base<TnsSize_>::MatrixArray;
42                 using typename GTC_STOCHASTIC_Base<TnsSize_>::DataType;
43                 using typename GTC_STOCHASTIC_Base<TnsSize_>::SparseTensor;
44                 using typename GTC_STOCHASTIC_Base<TnsSize_>::LongMatrix;
45
46                 using IntArray          = typename SparseTensorTraits<SparseTensor>::IntArray;
47                 using CartCommunicator = partensor::cartesian_communicator; // From ParallelWrapper.hpp
48                 using CartCommVector   = std::vector<CartCommunicator>;
49                 using IntVector        = std::vector<int>;
50                 using Int2DVector      = std::vector<std::vector<int>>;
51
52                 using Options = partensor::SparseOptions<TnsSize_,execution::openmpi_policy,SparseDefaultValues>;
53                 using Status  = partensor::SparseStatus<TnsSize_,execution::openmpi_policy,SparseDefaultValues>;
54
55                 // Variables that will be used in gtc implementations.
56                 struct Member_Variables
57                 {
58                     MPI_Communicator &world = Partensor()->MpiCommunicator(); // MPI_COMM_WORLD
59
60                     double          local_f_value;
61                     int              RxR;
62                     int              world_size;
63                     double           c_stochastic_perc;
64                 };
65
66             };
67
68         };
69
70     };

```

```

71     Int2DVector    displs_subTns;        // skipping dimension "rows" for each subtensor
72     Int2DVector    displs_subTns_R;     // skipping dimension "rows" for each subtensor times R (
for MPI communication purposes )
73     Int2DVector    subTnsDims;         // dimensions of subtensor
74     Int2DVector    subTnsDims_R;      // dimensions of subtensor times R ( for MPI communication
purposes )
75     Int2DVector    displs_local_update; // displacement in the local factor for update rows

76     Int2DVector    send_rcv_counts;    // rows to be communicated after update times R
77
78     CartCommVector layer_comm;
79     CartCommVector fiber_comm;
80
81     IntArray       layer_rank;
82     IntArray       fiber_rank;
83     IntArray       rows_for_update;
84     IntArray       tnsDims;
85
86     MatrixArray    layer_factors;
87     MatrixArray    layer_factors_T;
88     MatrixArray    factors_T;
89     MatrixArray    factor_T_factor;
90     MatrixArray    local_factors_T;
91     MatrixArray    norm_factors_T;
92     MatrixArray    old_factors_T;
93
94     Matrix         temp_matrix;
95     Matrix         Ratings_Base_T;
96     SparseTensor  subTns;
97
98     int            rank;
99     std::array<std::array<int, TnsSize-1>, TnsSize> offsets;
100
101     /*
102     * Calculates if the number of processors given from terminal
103     * are equal to the processors in the implementation.
104     *
105     * @param procs [in] @c stl array with the number of processors per
106     *                  dimension of the tensor.
107     */
108     void check_processor_availability(std::array<int, TnsSize> const &procs)
109     {
110         // MPI_Environment &env = Partensor()->MpiEnvironment();
111         world_size = world.size();
112         // numprocs must be product of options.proc_per_mode
113         if (std::accumulate(procs.begin(), procs.end(), 1,
114                             std::multiplies<int>()) != world_size && world.rank() == 0) {
115             Partensor()->Logger()->error("The product of the processors per mode must be equal to
{} nn", world_size);
116             // env.abort(-1);
117         }
118     }
119
120     Member_Variables() = default;
121     Member_Variables(int R, IntArray dims, std::array<int, TnsSize> &procs) : local_f_value(0.0),
122                                                                                                     RxR(R*R),
123
124     displs_subTns(TnsSize),
125     displs_subTns_R(TnsSize),
126     subTnsDims(TnsSize),
127     subTnsDims_R(TnsSize),
128     displs_local_update(TnsSize),
129     send_rcv_counts(TnsSize),
130                                                                                                     tnsDims(dims),
131                                                                                                     rank(R)
132     {
133         check_processor_availability(procs);
134         layer_comm.reserve(TnsSize);
135         fiber_comm.reserve(TnsSize);
136     }
137
138     Member_Variables(Member_Variables const &) = default;
139     Member_Variables(Member_Variables &&) = default;
140
141     Member_Variables &operator=(Member_Variables const &) = default;
142     Member_Variables &operator=(Member_Variables &&) = default;
143 };
144
145 template<int mode>
146 void sort_ratings_base_util(Matrix const &Ratings_Base_T,
147                             long int const nnz,
148                             Member_Variables &mv)

```

```

148     {
149         Matrix ratings_base_temp = Ratings_Base_T;
150         std::vector<std::vector<double>> vectorized_ratings_base;
151         vectorized_ratings_base.resize(nnz, std::vector<double>(TnsSize + 1));
152
153         for (int rows = 0; rows < static_cast<int>(TnsSize) + 1; rows++)
154         {
155             for (int cols = 0; cols < nnz; cols++)
156             {
157                 vectorized_ratings_base[cols][rows] = Ratings_Base_T(rows, cols);
158             }
159         }
160
161         // Sort
162         std::sort(vectorized_ratings_base.begin(), vectorized_ratings_base.end(), SortRows<TnsSize_,
mode, double>);
163
164         for (int rows = 0; rows < static_cast<int>(TnsSize) + 1; rows++)
165         {
166             for (int cols = 0; cols < nnz; cols++)
167             {
168                 ratings_base_temp(rows, cols) = vectorized_ratings_base[cols][rows];
169             }
170         }
171
172         Dist_NNZ_sorted<TnsSize>(mv.subTns, nnz, mv.displs_subTns, mv.fiber_rank, ratings_base_temp,
mv.subTnsDims, mode);
173         mv.subTns[mode].makeCompressed();
174
175         if constexpr (mode+1 < TnsSize)
176             sort_ratings_base_util<mode+1>(Ratings_Base_T, nnz, mv);
177     }
178
179     void sort_ratings_base(Matrix const &Ratings_Base_T,
180                          long int const nnz,
181                          Member_Variables &mv)
182     {
183         ReserveSparseTensor<TnsSize>(mv.subTns, mv.subTnsDims, mv.fiber_rank, mv.world_size, nnz);
184         sort_ratings_base_util<0>(Ratings_Base_T, nnz, mv);
185     }
186
187
188     void NesterovMNLS_stochastic( Member_Variables &mv,
189                                  Status const &st,
190                                  int const idx)
191     {
192         int iter = 0;
193
194         double L2;
195         double sqrt_q = 0, beta = 0;
196         double lambda = st.options.lambdas[idx];
197         int rows_layer = mv.subTnsDims[idx][mv.fiber_rank[idx]];
198
199         Matrix inv_L2(rows_layer, 1); // rows_layer
200
201         Matrix grad_Y_T(mv.rank, rows_layer);
202         Matrix grad_Y_local_T(mv.rank, mv.rows_for_update[idx]);
203
204         Matrix Y_T(mv.rank, rows_layer);
205         Matrix Y_local_T(mv.rank, mv.rows_for_update[idx]);
206
207         Matrix new_A(mv.rank, mv.rows_for_update[idx]);
208         Matrix A(mv.rank, mv.rows_for_update[idx]);
209
210         const Matrix zero_mat = Matrix::Zero(mv.rank, mv.rank);
211         const Matrix zero_vec = Matrix::Zero(mv.rank, 1);
212
213         A = mv.local_factors_T[idx];
214         Y_T = mv.layer_factors_T[idx]; // layer_factor
215         Y_local_T = A;
216
217         int last_mode = (idx == static_cast<int>(TnsSize) - 1) ? static_cast<int>(TnsSize) - 2 :
static_cast<int>(TnsSize) - 1;
218
219         Matrix temp_R_1(mv.rank, 1);
220         Matrix temp_RxR(mv.rank, mv.rank);
221         Matrix temp_col(mv.rank, 1);
222
223         std::srand(std::time(nullptr));
224
225         while (1)
226         {
227             grad_Y_T.setZero();
228
229             if (iter >= st.options.max_nesterov_iter)
230             {
231                 break;
232             }
233         }

```

```

232
233 // Compute grad_Y
234 for (long int i = 0; i < mv.subTns[idx].outerSize(); ++i)
235 {
236     temp_col.setZero();
237
238     SparseMatrix::InnerIterator it(mv.subTns[idx], i);
239
240     // Get the number of nnz per row of matricization.
241     long int nnzs_per_col = mv.subTns[idx].innerVector(i).nonZeros();
242
243     long int var_blocksize_i = static_cast<long
int>(mv.c_stochastic_perc * nnzs_per_col);
244
245     if (var_blocksize_i > 0)
246     {
247         // Choose a pivot from [0, nnzs_per_col - blocksize].
248         long int pivot = (var_blocksize_i > nnzs_per_col ? 0 : (std::rand() % (nnzs_per_col -
var_blocksize_i + 1)));
249
250         SparseMatrix::InnerIterator it(mv.subTns[idx], i);
251
252         // Iterate over [pivot, pivot + blocksize] nnz elements per row.
253         // it += pivot;
254         for (long int acuum = 0; acuum < pivot; acuum++)
255             ++it;
256
257         temp_RxR.setZero();
258
259         for (long int sample = 0; sample < var_blocksize_i; sample++, ++it)
260         {
261             temp_R_1 = Matrix::Ones(mv.rank, 1);
262             // Select rows of each factor an compute the respective row of the Khatri-Rao product.
263             for (int mode_i = last_mode, kr_counter = static_cast<int>(TnsSize) - 2; mode_i >= 0
&& kr_counter >= 0; mode_i--)
264             {
265                 if (mode_i == idx)
266                 {
267                     continue;
268                 }
269                 long int row;
270                 row = ((it.row()) / mv.offsets[idx][kr_counter]) %
mv.subTnsDims[mode_i][mv.fiber_rank[mode_i]];
271                 temp_R_1 = temp_R_1.cwiseProduct(mv.layer_factors_T[mode_i].col(row));
272                 kr_counter--;
273             }
274             // Computation of row of Z according the relation (10) of the paper.
275             temp_col += ((temp_R_1.transpose() * Y_T.col(i))(0) - it.value()) * temp_R_1;
276
277             temp_RxR.noalias() += (temp_R_1 * temp_R_1.transpose());
278         }
279
280         grad_Y_T.col(i) = temp_col;
281
282         all_reduce( mv.layer_comm[idx],
283                   inplace(temp_RxR.data()),
284                   mv.RxR,
285                   std::plus<double>() );
286     }
287     else
288     {
289         all_reduce( mv.layer_comm[idx],
290                   zero_mat.data(),
291                   mv.RxR,
292                   temp_RxR.data(),
293                   std::plus<double>() );
294     }
295
296     L2 = PowerMethod(temp_RxR, 1e-3);
297     L2 += lambda;
298     inv_L2(i) = 1 / L2;
299 } // end of for loop on layer_rows
300
301 // Add each process' results and scatter the block rows among the processes in the layer.
302 // MPI_Reduce_scatter(grad_Y_T.data(), grad_Y_local_T.data(), send_recv_counts, MPI_DOUBLE,
MPI_SUM, mode_layer_comm);
303 v2::reduce_scatter( mv.layer_comm[idx],
304                   grad_Y_T,
305                   mv.send_recv_counts[idx][0],
306                   grad_Y_local_T );
307
308 // Add proximal term.
309 grad_Y_local_T += lambda * (Y_local_T + A);
310
311 for (long int i=0; i<mv.rows_for_update[idx]; i++)
312 {
313

```

```

314         long int translate_i = i + mv.displs_local_update[idx][mv.layer_rank[idx]]/mv.rank;
315
316         new_A.col(i) = (Y_local_T.col(i) - grad_Y_local_T.col(i)
317         inv_L2(translate_i)).cwiseMax(zero_vec);
318
319         sqrt_q = sqrt( lambda * inv_L2(translate_i) );
320         beta = (1 - sqrt_q) / (1 + sqrt_q);
321
322         // Update Y
323         Y_local_T.col(i) = (1 + beta) * new_A.col(i) - beta * A.col(i);
324     }
325
326     // The updated block rows of Y are all gathered, and we have the whole updated Y of the
327     layer.
328     v2::all_gatherv( mv.layer_comm[idx],
329                     Y_local_T,
330                     mv.send_recv_counts[idx][mv.layer_rank[idx]],
331                     mv.send_recv_counts[idx][0],
332                     mv.displs_local_update[idx][0],
333                     Y_T );
334
335     A = new_A;
336
337     iter++;
338 }
339
340 /*
341  * In case option variable @c writeToFile is enabled then, before the end
342  * of the algorithm writes the resulted factors in files, where their
343  * paths are specified before compiling in @ options.final_factors_path.
344  *
345  * @param st [in] Struct where the returned values of @c Gtc are stored.
346  */
347 void writeFactorsToFile(Status const &st)
348 {
349     std::size_t size;
350     for(std::size_t i=0; i<TnsSize; ++i)
351     {
352         size = st.factors[i].rows() * st.factors[i].cols();
353         partensor::write(st.factors[i],
354                         st.options.final_factors_paths[i],
355                         size);
356     }
357 }
358
359 /*
360  * Compute the cost function value at the end of each outer iteration
361  * based on the last factor.
362  *
363  * @param grid_comm [in] MPI communicator where the new cost function value
364  * will be communicated and computed.
365  * @param mv [in] Struct where ALS variables are stored.
366  * @param st [in,out] Struct where the returned values of @c Gtc are stored.
367  * In this case the cost function value is updated.
368  */
369 void cost_function( CartCommunicator const &grid_comm,
370                   Member_Variables &mv,
371                   Status &st )
372 {
373     Matrix temp_R_1(mv.rank, 1);
374     double temp_1_1 = 0;
375     mv.local_f_value = 0;
376     std::array<int, TnsSize-1> offsets;
377     offsets[0] = 1;
378     for (int j = 1; j < static_cast<int>(TnsSize) - 1; j++)
379     {
380         offsets[j] = offsets[j - 1] * mv.subTnsDims[j-1][mv.fiber_rank[j-1]]; //
381         mv.layer_factors_T[j - 1].cols()
382     }
383
384     for (long int i = 0; i < mv.subTns[lastFactor].outerSize(); ++i)
385     {
386         int row;
387         for (SparseMatrix::InnerIterator it(mv.subTns[lastFactor], i); it; ++it)
388         {
389             temp_R_1 = mv.layer_factors_T[lastFactor].col(it.col());
390             // Select rows of each factor and compute the Hadamard product of the respective row
391             // of the Khatri-Rao product, and the row of factor A_N.
392             // temp_R_1 = A_N(i_N,:) . * ... . * A_2(i_2,:) . * A_1(i_1,:);
393             for (int mode_i = static_cast<int>(TnsSize) - 2; mode_i >= 0; mode_i--)
394             {
395                 row = ((it.row()) / offsets[mode_i]) %
396                 (mv.subTnsDims[mode_i][mv.fiber_rank[mode_i]]);
397                 temp_R_1.noalias() = temp_R_1.cwiseProduct(mv.layer_factors_T[mode_i].col(row));
398             }
399         }
400     }

```



```

396         temp_1_1 = it.value() - temp_R_1.sum();
397         mv.local_f_value += temp_1_1 * temp_1_1;
398     }
399 }
400
401 all_reduce( grid_comm,
402            mv.local_f_value,
403            st.f_value,
404            std::plus<double>() );
405 }
406
407 /*
408 * Compute the cost function value at the end of each outer iteration
409 * based on the last accelerated factor.
410 *
411 * @param grid_comm      [in] MPI communicator where the new cost function value
412 *                        will be communicated and computed.
413 * @param mv             [in] Struct where ALS variables are stored.
414 * @param st             [in] Struct where the returned values of @c Gtc are stored.
415 *                        In this case the cost function value is updated.
416 * @param factors        [in] Accelerated factors.
417 * @param factors_T_factors [in] Gramian matrices of factors.
418 *
419 * @returns The cost function calculated with the accelerated factors.
420 */
421 double accel_cost_function(CartCommunicator const &grid_comm,
422                          Member_Variables const &mv,
423                          MatrixArray const &layer_factors_T)
424 {
425     Matrix temp_R_1(mv.rank, 1);
426     double temp_1_1 = 0;
427     double f_value = 0;
428
429     std::array<int, TnsSize-1> offsets;
430     offsets[0] = 1;
431     for (int j = 1; j < static_cast<int>(TnsSize) - 1; j++)
432     {
433         offsets[j] = offsets[j - 1] * mv.subTnsDims[j-1][mv.fiber_rank[j-1]];
434     }
435
436     for (long int i = 0; i < mv.subTns[lastFactor].outerSize(); ++i)
437     {
438         int row;
439         for (SparseMatrix::InnerIterator it(mv.subTns[lastFactor], i); it; ++it)
440         {
441             temp_R_1 = layer_factors_T[lastFactor].col(it.col());
442             // Select rows of each factor and compute the Hadamard product of the respective row of
443             // the Khatri-Rao product, and the row of factor A_N.
444             // temp_R_1 = A_N(i,N,:) * ... * A_2(i,2,:) * A_1(i,1,:);
445             for (int mode_i = static_cast<int>(TnsSize) - 2; mode_i >= 0; mode_i--)
446             {
447                 row = ((it.row()) / offsets[mode_i]) %
448                     (mv.subTnsDims[mode_i][mv.fiber_rank[mode_i]]);
449                 temp_R_1.noalias() = temp_R_1.cwiseProduct(layer_factors_T[mode_i].col(row));
450             }
451             temp_1_1 = it.value() - temp_R_1.sum();
452             f_value += temp_1_1 * temp_1_1;
453         }
454     }
455
456     all_reduce( grid_comm,
457                inplace(&f_value),
458                1,
459                std::plus<double>() );
460
461     return f_value;
462 }
463
464 /*
465 * Make use of the dimensions and the number of processors per dimension
466 * and then calculates the dimensions of the subtensor and subfactor for
467 * each processor.
468 *
469 * @tparam Dimensions      Array type containing the length of Tensor's dimensions.
470 *
471 * @param tnsDims          [in] Tensor Dimensions. Each index contains the corresponding
472 *                        factor's rows length.
473 * @param st               [in] Struct where the returned values of @c Gtc are stored.
474 * @param R                [in] The rank of decomposition.
475 * @param mv               [in,out] Struct where ALS variables are stored.
476 *                        Updates @c stl arrays with dimensions for subtensors and
477 *                        subfactors.
478 */
479 void compute_sub_dimensions(Status const &st,
480                          Member_Variables &mv)
481 {
482     for (std::size_t i = 0; i < TnsSize; ++i)

```

```

481     {
482         mv.factor_T_factor[i].noalias() = st.factors[i].transpose()           * st.factors[i];
483     }
484     DisCount(mv.displs_subTns[i], mv.subTnsDims[i], st.options.proc_per_mode[i], mv.tnsDims[i],
1);
485     // for fiber communication and Gatherv
486     DisCount(mv.displs_subTns_R[i], mv.subTnsDims_R[i], st.options.proc_per_mode[i],
mv.tnsDims[i], static_cast<int>(mv.rank));
487     // information per layer
488     DisCount(mv.displs_local_update[i], mv.send_rcv_counts[i], mv.world_size /
st.options.proc_per_mode[i],
489             mv.subTnsDims[i][mv.fiber_rank[i]],
static_cast<int>(mv.rank));
490
491     mv.rows_for_update[i] = mv.send_rcv_counts[i][mv.layer_rank[i]] /
static_cast<int>(mv.rank);
492 }
493
494     calculate_offsets(mv);
495 }
496
497 void calculate_offsets(Member_Variables &mv)
498 {
499     for (int idx = 0; idx < static_cast<int>(TnsSize); idx++)
500     {
501         mv.offsets[idx][0] = 1;
502         for (int j = 1, mode = 0; j < static_cast<int>(TnsSize) - 1; j++, mode++)
503         {
504             if (idx == mode)
505             {
506                 mode++;
507             }
508             mv.offsets[idx][j] = mv.offsets[idx][j - 1] * mv.subTnsDims[mode][mv.fiber_rank[mode]];
509         }
510     }
511 }
512
513 /*
514  * Based on each factor's constraint, a different
515  * update function is used at every outer iteration.
516  *
517  * Computes also factor^T * factor at the end.
518  *
519  * @param idx [in] Factor to be updated.
520  * @param R [in] The rank of decomposition.
521  * @param st [in] Struct where the returned values of @c Gtc are stored.
522  * Here constraints and options variables are needed.
523  * @param mv [in,out] Struct where ALS variables are stored.
524  * Updates the factors of each layer.
525  */
526 void update_factor(int Status, const idx,
527                   Member_Variables const &st,
528                   Member_Variables &mv )
529 {
530     switch ( st.options.constraints[idx] )
531     {
532     case Constraint::unconstrained:
533     {
534         break;
535     }
536     case Constraint::nonnegativity:
537     {
538         NesterovMNLS_stochastic(mv, st, idx);
539         break;
540     }
541     case Constraint::sparsity:
542     {
543         break;
544     }
545     default: // in case of Constraint::constant
546     {
547         break;
548     }
549     } // end of constraints switch
550
551     if (st.options.constraints[idx] != Constraint::constant)
552     {
553         v2::all_gatherv( mv.layer_comm[idx],
554                         mv.local_factors_T[idx],
555                         mv.send_rcv_counts[idx][mv.layer_rank[idx]],
556                         mv.send_rcv_counts[idx][0],
557                         mv.displs_local_update[idx][0],
558                         mv.layer_factors_T[idx] );
559
560         mv.layer_factors[idx] = mv.layer_factors_T[idx].transpose();
561         mv.factor_T_factor[idx].noalias() = mv.layer_factors_T[idx] * mv.layer_factors[idx];
562     }
563 }
564
565 /*
566  * At the end of the algorithm processor 0

```

```

563     * collects each part of the factor that each
564     * processor holds and return them in status.factors.
565     *
566     * @param mv      [in]   Struct where ALS variables are stored.
567     *               Use variables to compute result factors by gathering each
568     *               part of the factor from processors.
569     * @param st      [in,out] Struct where the returned values of @c Gtc are stored.
570     *               Stores the resulted factors.
571     */
572 void gather_final_factors(Member_Variables      &mv,
573                          Status               &st)
574 {
575     for(std::size_t i=0; i<TnsSize; ++i)
576     {
577         mv.temp_matrix.resize(static_cast<int>(mv.rank), mv.tnsDims[i]);
578         // Gatherv from all processors to processor with rank 0 the final factors
579         v2::gatherv( mv.fiber_comm[i],
580                   mv.layer_factors_T[i],
581                   mv.subTnsDims_R[i][mv.fiber_rank[i]],
582                   mv.subTnsDims_R[i][0],
583                   mv.displs_subTns_R[i][0],
584                   0,
585                   mv.temp_matrix );
586
587         st.factors[i] = mv.temp_matrix.transpose();
588     }
589 }
590
591 /*
592  * @brief Line Search Acceleration
593  *
594  * Performs an acceleration step in the updated factors, and keeps the accelerated factors when
595  * the step succeeds. Otherwise, the acceleration step is ignored.
596  * Line Search Acceleration reduces the number outer iterations in the ALS algorithm.
597  *
598  * @note This implementation ONLY, if factors are of @c Matrix type.
599  *
600  * @param grid_comm [in]   MPI communicator where the new cost function value
601  *               will be communicated and computed.
602  * @param mv        [in,out] Struct where ALS variables are stored.
603  *               In case the acceleration is successful layer factor^T * factor
604  *               and layer factor variables are updated.
605  * @param st        [in,out] Struct where the returned values of @c Gtc are stored.
606  *               If the acceleration succeeds updates cost function value.
607  *
608  */
609 void line_search_accel(CartCommunicator const &grid_comm,
610                      Member_Variables      &mv,
611                      Status               &st)
612 {
613     double f_accel = 0.0; // Objective Value after the acceleration step
614     double const accel_step = pow(st.ao_iter+1,(1.0/(st.options.accel_coeff)));
615
616     MatrixArray accel_factors_T;
617     MatrixArray accel_gramians;
618
619     for(std::size_t i=0; i<TnsSize; ++i)
620     {
621         accel_factors_T[i] = mv.old_factors_T[i] + accel_step * (mv.layer_factors_T[i] -
622 mv.old_factors_T[i]);
623         accel_gramians[i] = accel_factors_T[i] * accel_factors_T[i].transpose();
624         all_reduce( mv.fiber_comm[i],
625                   inplace(accel_gramians[i].data()),
626                   mv.RxR,
627                   std::plus<double>() );
628     }
629
630     f_accel = accel_cost_function(grid_comm, mv, accel_factors_T);
631     if (st.f_value > f_accel)
632     {
633         mv.layer_factors_T = accel_factors_T;
634         mv.factor_T_factor = accel_gramians;
635         st.f_value = f_accel;
636         if(grid_comm.rank() == 0)
637             Partensor()->Logger()->info("Acceleration Step SUCCEEDED! at iter: {}", st.ao_iter);
638     }
639     else
640         st.options.accel_fail++;
641
642     if (st.options.accel_fail==5)
643     {
644         st.options.accel_fail=0;
645         st.options.accel_coeff++;
646     }
647 }
648 /*

```

```

649     * Parallel implementation of als method with MPI.
650     *
651     * @tparam Dimensions          Array type containing the Tensor dimensions.
652     *
653     * @param grid_comm [in]      The communication grid, where the processors
654     *                          communicate their cost function.
655     * @param tnsDims   [in]      Tensor Dimensions. Each index contains the corresponding
656     *                          factor's rows length.
657     * @param R         [in]      The rank of decomposition.
658     * @param mv        [in]      Struct where ALS variables are stored and being updated
659     *                          until a termination condition is true.
660     * @param status   [in,out] Struct where the returned values of @c Gtc are stored.
661     */
662 void aogtc_stochastic(CartCommunicator const &grid_comm,
663                     Member_Variables &mv,
664                     Status &status)
665 {
666     status.frob_tns = (mv.subTns[0]).squaredNorm();
667     all_reduce( grid_comm,
668               inplace(&status.frob_tns),
669               1,
670               std::plus<double>() );
671
672     cost_function(grid_comm, mv, status);
673     status.rel_costFunction = status.f_value / status.frob_tns;
674
675     std::size_t epoch = static_cast<std::size_t>(1/mv.c_stochastic_perc);
676     std::size_t epoch_counter = 0;
677
678     for (int i=0; i< static_cast<int>(TnsSize); i++)
679     {
680         mv.factor_T_factor[i].noalias() = mv.layer_factors_T[i] * mv.layer_factors[i];
681         all_reduce( mv.fiber_comm[i],
682                   inplace(mv.factor_T_factor[i].data()),
683                   mv.RxR,
684                   std::plus<double>() );
685     }
686
687     // Wait for all processors to reach here
688     grid_comm.barrier();
689
690     // ---- Loop until ALS converges ----
691     while(1)
692     {
693         status.ao_iter++;
694         if (!grid_comm.rank())
695         {
696             Partensor()->Logger()->info("iter: {} -- fvalue: {} -- relative_costFunction: {}",
697 status.ao_iter,
698                                     status.f_value, status.rel_costFunction);
699         }
700
701         // -----> // loop for every mode
702         // -----> //
703         for (std::size_t i = 0; i < TnsSize; i++)
704         {
705             // Partition rows of subfactor to the processes in the respective layer.
706             mv.local_factors_T[i] = mv.layer_factors_T[i].block(0,
707 mv.displs_local_update[i][mv.layer_rank[i]] / mv.rank, mv.rank, mv.rows_for_update[i]);
708
709             update_factor(i, status, mv);
710         }
711
712         // ---- Cost function Computation ----
713         if (status.ao_iter % epoch == 0)
714         {
715             epoch_counter++;
716             cost_function(grid_comm, mv, status);
717             status.rel_costFunction = status.f_value / status.frob_tns;
718         }
719
720         // if(status.options.normalization && !mv.all_orthogonal)
721         //     Normalize(mv.weight_factor, mv.rank, mv.factor_T_factor, mv.layer_factors);
722
723         // ---- Terminating condition ----
724         if (status.rel_costFunction < status.options.threshold_error || epoch_counter >=
725 status.options.max_iter)
726         {
727             gather_final_factors(mv, status);
728             if(grid_comm.rank() == 0)
729             {
730                 Partensor()->Logger()->info("Processor 0 collected all {} factors.          nn", TnsSize);
731                 if(status.options.writeToFile)
732                     writeFactorsToFile(status);
733             }
734             break;
735         }
736     }
737 }

```

```

732
733     if (status.options.acceleration)
734     {
735         mv.norm_factors_T = mv.layer_factors_T;
736         // ---- Acceleration Step ----
737         if (status.ao_iter > 1)
738             line_search_accel(grid_comm, mv, status);
739
740         mv.old_factors_T = mv.norm_factors_T;
741     }
742
743 } // end of outer while loop
744 }
745
746 void initialize_factors(Member_Variables &mv,
747                        Status &status)
748 {
749     if(status.options.initialized_factors)
750     {
751         if(status.options.read_factors_from_file)
752         {
753             for(std::size_t i=0; i<TnsSize; ++i)
754             {
755                 status.factors[i] = Matrix(mv.tnsDims[i], static_cast<int>(mv.rank));
756                 read( status.options.initial_factors_paths[i],
757                     mv.tnsDims[i] * mv.rank,
758                     0,
759                     status.factors[i] );
760             }
761         }
762         else
763             status.factors = status.options.factorsInit;
764     }
765     else
766         makeFactors(mv.tnsDims, status.options.constraints, mv.rank, status.factors);
767 }
768
769 Status operator()(Options const &options)
770 {
771     Status status(options);
772     Member_Variables mv(options.rank, options.tnsDims, status.options.proc_per_mode);
773
774     // Communicator with cartesian topology
775     CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);
776
777     // Functions that create layer and fiber grids.
778     create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
779     create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);
780
781     mv.c_stochastic_perc = options.c_stochastic_perc;
782
783     // produce estimate factors using uniform distribution with entries in [0,1].
784     initialize_factors(mv, status);
785
786     compute_sub_dimensions(status, mv);
787
788     for (std::size_t i = 0; i < TnsSize; ++i)
789     {
790         mv.layer_factors[i] =
791         status.factors[i].block(mv.displs_subTns[i][mv.fiber_rank[i]], 0,
792                                mv.subTnsDims[i][mv.fiber_rank[i]], mv.rank);
793         mv.layer_factors_T[i] = mv.layer_factors[i].transpose();
794         mv.factor_T_factor[i].noalias() = mv.layer_factors_T[i] * mv.layer_factors[i];
795     }
796
797     long long int fileSize = (TnsSize + 1) * options.nonZeros;
798
799     // Matrix Ratings_Base = Matrix(options.nonZeros, static_cast<int>(TnsSize+1));
800     Matrix Ratings_Base_T = Matrix(static_cast<int>(TnsSize+1), options.nonZeros);
801
802     // Read the whole Tensor from a file
803     read( options.ratings_path,
804           fileSize,
805           0,
806           Ratings_Base_T );
807
808     // Matrix Ratings_Base_T = Ratings_Base.transpose();
809
810     sort_ratings_base(Ratings_Base_T, options.nonZeros, mv);
811     Ratings_Base_T.resize(0,0);
812     // Ratings_Base.resize(0,0);
813
814     aogtc_stochastic(grid_comm, mv, status);
815
816     return status;
817 }
818
819
820

```

```

841     Status operator()(Matrix const &Ratings_Base_T,
842                       Options const &options)
843     {
844         Status status(options);
845         Member_Variables mv(options.rank, options.tnsDims, status.options.proc_per_mode);
846
847         // Communicator with cartesian topology
848         CartCommunicator grid_comm(mv.world, status.options.proc_per_mode, true);
849
850         // Functions that create layer and fiber grids.
851         create_layer_grid(grid_comm, mv.layer_comm, mv.layer_rank);
852         create_fiber_grid(grid_comm, mv.fiber_comm, mv.fiber_rank);
853
854         mv.c_stochastic_perc = options.c_stochastic_perc;
855
856         // produce estimate factors using uniform distribution with entries in [0,1].
857         initialize_factors(mv, status);
858
859         compute_sub_dimensions(status, mv);
860
861         for (std::size_t i = 0; i < TnsSize; ++i)
862         {
863             mv.layer_factors[i] =
864             status.factors[i].block(mv.displs_subTns[i][mv.fiber_rank[i]], 0,
865                                   mv.subTnsDims[i][mv.fiber_rank[i]], mv.rank);
866             mv.layer_factors_T[i] = mv.layer_factors[i].transpose();
867             mv.factor_T_factor[i].noalias() = mv.layer_factors_T[i] * mv.layer_factors[i];
868         }
869
870         // Each processor takes a subtensor from tnsX
871         ReserveSparseTensor<TnsSize>(mv.subTns, mv.subTnsDims, mv.fiber_rank, mv.world_size,
872                                     options.nonZeros);
873
874         Dist_NNZ<TnsSize>(mv.subTns, options.nonZeros, mv.displs_subTns, mv.fiber_rank,
875                          Ratings_Base_T, mv.subTnsDims);
876         // Ratings_Base_T.resize(0,0);
877
878         for(int mode_i = 0; mode_i < static_cast<int>(TnsSize); mode_i++)
879         {
880             mv.subTns[mode_i].makeCompressed();
881         }
882
883         aogtc_stochastic(grid_comm, mv, status);
884
885         return status;
886     }
887 } // namespace partensor
888
889 } // 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.](#)

```

1  #ifndef DOXYGEN_SHOULD_SKIP_THIS
15 #endif // DOXYGEN_SHOULD_SKIP_THIS
16 /*****
25 #if !defined(PARTENSOR_GTC_STOCHASTIC_HPP)
26 #error "GTC_STOCHASTIC_OMP can only included inside GTC_STOCHASTIC"
27 #endif / * PARTENSOR_GTC_STOCHASTIC_HPP
28
29 namespace partensor
30 {
31     inline namespace v1

```

```

32 {
33     namespace internal
34     {
35         template <std::size_t TnsSize_>
36         struct GTC_STOCHASTIC<TnsSize_, execution::openmp_policy> : public GTC_STOCHASTIC_Base<TnsSize_>
37         {
38             using GTC_STOCHASTIC_Base<TnsSize_>::TnsSize;
39             using GTC_STOCHASTIC_Base<TnsSize_>::lastFactor;
40             using typename GTC_STOCHASTIC_Base<TnsSize_>::Dimensions;
41             using typename GTC_STOCHASTIC_Base<TnsSize_>::MatrixArray;
42             using typename GTC_STOCHASTIC_Base<TnsSize_>::DataType;
43             using typename GTC_STOCHASTIC_Base<TnsSize_>::SparseTensor;
44             using typename GTC_STOCHASTIC_Base<TnsSize_>::IntArray;
45             using typename GTC_STOCHASTIC_Base<TnsSize_>::LongMatrix;
46
47             using Options = partensor::SparseOptions<TnsSize_, execution::openmp_policy, SparseDefaultValues>;
48             using Status = partensor::SparseStatus<TnsSize_, execution::openmp_policy, SparseDefaultValues>;
49
50             // Variables that will be used in gtc implementations.
51             struct Member_Variables
52             {
53                 MatrixArray factors_T;
54                 MatrixArray factor_T_factor;
55                 MatrixArray mttkrp_T;
56                 IntArray tnsDims;
57                 std::array<std::array<int, TnsSize_ - 1>, TnsSize_> offsets;
58
59                 MatrixArray norm_factors_T;
60                 MatrixArray old_factors_T;
61
62                 Matrix cwise_factor_product;
63                 Matrix Ratings_Base_T;
64                 SparseTensor tnsX;
65
66                 // bool all_orthogonal = true;
67                 // int weight_factor;
68                 int rank;
69                 double c_stochastic_perc;
70
71                 MatrixArray grad;
72                 MatrixArray Y;
73                 MatrixArray invL;
74
75                 Member_Variables() = default;
76
77                 Member_Variables(int R, IntArray dims) : tnsDims(dims),
78                                                         rank(R)
79                 {}
80
81                 Member_Variables(Member_Variables const &) = default;
82                 Member_Variables(Member_Variables &&) = default;
83
84                 Member_Variables &operator=(Member_Variables const &) = default;
85                 Member_Variables &operator=(Member_Variables &&) = default;
86             };
87
88             /*
89             * In case option variable @c writeToFile is enabled, then, before the end
90             * of the algorithm, it writes the resulted factors in files, whose
91             * paths are specified before compiling in @ options.final_factors_path.
92             *
93             * @param st [in] Struct where the returned values of @c Gtc are stored.
94             */
95             void writeFactorsToFile(Status const &st)
96             {
97                 std::size_t size;
98                 for(std::size_t i=0; i<TnsSize; ++i)
99                 {
100                     size = st.factors[i].rows() * st.factors[i].cols();
101                     partensor::write(st.factors[i],
102                                    st.options.final_factors_paths[i],
103                                    size);
104                 }
105             }
106
107             /*
108             * Compute the cost function value at the end of each outer iteration
109             * based on the last factor.
110             *
111             * @param mv [in] Struct where ALS variables are stored.
112             * @param st [in,out] Struct where the returned values of @c Gtc are stored.
113             * In this case the cost function value is updated.
114             */
115             void cost_function(Member_Variables const &mv,
116                               Status &st)
117             {
118                 Matrix temp_R_1(mv.rank, 1);

```

```

119     double temp_1_1 = 0;
120     double f_value_loc = 0;
121
122     #pragma omp master
123     st.f_value = 0;
124
125     #pragma omp barrier
126
127     std::array<int,TnsSize-1> offsets;
128     offsets[0] = 1;
129     for (int j = 1; j < static_cast<int>(TnsSize) - 1; j++)
130     {
131         offsets[j] = offsets[j - 1]          * mv.tnsDims[j-1];
132     }
133
134     #pragma omp for schedule(static)
135     for (long int i = 0; i < mv.tnsX[lastFactor].outerSize(); ++i)
136     {
137         int row = 0;
138         for (SparseMatrix::InnerIterator it(mv.tnsX[lastFactor], i); it; ++it)
139         {
140             temp_R_1 = mv.factors_T[lastFactor].col(it.col());
141             // 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.
142             for (int mode_i = static_cast<int>(TnsSize) - 2; mode_i >= 0; mode_i--)
143             {
144                 row = ((it.row()) / offsets[mode_i]) % (mv.tnsDims[mode_i]);
145                 temp_R_1.noalias() = temp_R_1.cwiseProduct(mv.factors_T[mode_i].col(row));
146             }
147             temp_1_1 = it.value() - temp_R_1.sum();
148             f_value_loc += temp_1_1          * temp_1_1;
149         }
150     }
151     #pragma omp atomic
152     st.f_value += f_value_loc;
153
154 }
155
156 /*
157 * Compute the cost function value at the end of each outer iteration
158 * based on the last accelerated factor.
159 *
160 * @param mv          [in] Struct where ALS variables are stored.
161 * @param accel_factors [in] Accelerated factors.
162 *
163 * @returns The cost function calculated with the accelerated factors.
164 */
165 double accel_cost_function(Member_Variables      const &mv,
166                           MatrixArray          const &accel_factors)
167 {
168     Matrix temp_R_1(mv.rank, 1);
169     double temp_1_1 = 0;
170     double f_value = 0;
171
172     std::array<int,TnsSize-1> offsets;
173     offsets[0] = 1;
174     for (int j = 1; j < static_cast<int>(TnsSize) - 1; j++)
175     {
176         offsets[j] = offsets[j - 1]          * mv.tnsDims[j-1];
177     }
178
179     #pragma omp for schedule(static)
180     for (long int i = 0; i < mv.tnsX[lastFactor].outerSize(); ++i)
181     {
182         int row = 0;
183         for (SparseMatrix::InnerIterator it(mv.tnsX[lastFactor], i); it; ++it)
184         {
185             temp_R_1 = accel_factors[lastFactor].col(it.col());
186             // 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.
187             // temp_R_1 = A_N(i,N,:) . * ... . * A_2(i,2,:) . * A_1(i,1,:)
188             for (int mode_i = static_cast<int>(TnsSize) - 2; mode_i >= 0; mode_i--)
189             {
190                 row = ((it.row()) / offsets[mode_i]) % (mv.tnsDims[mode_i]);
191                 temp_R_1.noalias() = temp_R_1.cwiseProduct(accel_factors[mode_i].col(row));
192             }
193             temp_1_1 = it.value() - temp_R_1.sum();
194             f_value += temp_1_1          * temp_1_1;
195         }
196     }
197     return f_value;
198 }
199
200 void calculate_offsets(Member_Variables &mv)
201 {
202     for (int idx = 0; idx < static_cast<int>(TnsSize); idx++)
203     {

```



```

204     mv.offsets[idx][0] = 1;
205     for (int j = 1, mode = 0; j < static_cast<int>(TnsSize) - 1; j++, mode++)
206     {
207         if (idx == mode)
208         {
209             mode++;
210         }
211         mv.offsets[idx][j] = mv.offsets[idx][j - 1] * mv.tnsDims[mode];
212     }
213 }
214 }
215
216 void unconstraint_update(int          const  idx,
217                          Member_Variables  &mv,
218                          Status            &st)
219 {
220     int r = mv.rank;
221     Matrix eye = st.options.lambdas[idx] * Matrix::Identity(r, r);
222
223     int last_mode = (idx == static_cast<int>(TnsSize) - 1) ? static_cast<int>(TnsSize) - 2 :
224     static_cast<int>(TnsSize) - 1;
225
226     Matrix MTTKRP_col(r, 1);
227     Matrix temp_RxR(r, r);
228     Matrix temp_R_1(r, 1);
229
230     // Compute MTTKRP
231     #pragma omp for schedule(guided) nowait
232     for (long int i = 0; i < mv.tnsX[idx].outerSize(); ++i)
233     {
234         MTTKRP_col.setZero();
235         temp_RxR.setZero(); // is the Hadamard product of Grammians of the Factors, that
correspond to the nnz elements of the Tensor.
236         for (SparseMatrix::InnerIterator it(mv.tnsX[idx], i); it; ++it)
237         {
238             temp_R_1 = Matrix::Ones(r, 1);
239             int row;
240             // Select rows of each factor an compute the respective row of the Khatri-Rao
product.
241             for (int mode_i = last_mode, kr_counter = static_cast<int>(TnsSize) - 2; mode_i >= 0
&& kr_counter >= 0; mode_i--)
242             {
243                 if (mode_i == idx)
244                 {
245                     continue;
246                 }
247                 row = ((it.row()) / mv.offsets[idx][kr_counter]) % (mv.tnsDims[mode_i]);
248                 temp_R_1 = temp_R_1.cwiseProduct(mv.factors_T[mode_i].col(row));
249                 kr_counter--;
250             }
251             // Subtract from the previous row the respective row of W, according to relation
(9).
252             MTTKRP_col.noalias() += it.value() * temp_R_1;
253             temp_RxR.noalias() += temp_R_1 * temp_R_1.transpose();
254         }
255         mv.factors_T[idx].col(i) = (temp_RxR + eye).inverse() * MTTKRP_col;
256     }
257 }
258
259 /*
260 * Based on each factor's constraint, a different
261 * update function is used at every outer iteration.
262 *
263 * Computes also factor^T * factor at the end.
264 *
265 * @param idx [in] Factor to be updated.
266 * @param mv [in] Struct where ALS variables are stored.
267 * @param st [in,out] Struct where the returned values of @c Gtc are stored.
268 * Updates the @c stl array with the factors.
269 */
270 void update_factor(int          const  idx,
271                   Member_Variables  &mv,
272                   Status            &st )
273 {
274     // Update factor
275     switch ( st.options.constraints[idx] )
276     {
277     case Constraint::unconstrained:
278     {
279         // unconstraint_update(idx, mv, st);
280         break;
281     }
282     case Constraint::nonnegativity:
283     {
284         dynamic_blocksize::local_L::StochasticNesterovMNLs(mv.factors_T, mv.tnsDims, mv.tnsX[idx],
mv.offsets[idx], mv.c_stochastic_perc, mv.Y[idx],

```

```

285             st.options.max_nesterov_iter, st.options.lambdas[idx], idx);
286         }
287     }
288     default: // in case of Constraint::constant
289         break;
290 }
291
292 // Compute  $A^T * A + B^T * B + \dots$ 
293 #pragma omp master
294 {
295     st.factors[idx] = mv.factors_T[idx].transpose();
296     mv.factor_T_factor[idx].noalias() = mv.factors_T[idx]
297         * st.factors[idx];
298 }
299
300 /*
301  * @brief Line Search Acceleration
302  *
303  * Performs an acceleration step on the updated factors, and keeps the accelerated factors
304  * when the step succeeds. Otherwise, the acceleration step is ignored.
305  * Line Search Acceleration reduces the number of outer iterations in the ALS algorithm.
306  *
307  * @note This implementation ONLY, if factors are of @c Matrix type.
308  *
309  * @param mv [in,out] Struct where ALS variables are stored.
310  *             In case the acceleration step is successful the Gramian
311  *             matrices of factors are updated.
312  * @param st [in,out] Struct where the returned values of @c Gtc are stored.
313  *             If the acceleration succeeds updates @c factors
314  *             and cost function value.
315  */
316
317 void line_search_accel(Member_Variables &mv,
318                      Status           &st,
319                      double           &f_accel,
320                      double           &accel_step,
321                      MatrixArray      &accel_factors_T,
322                      MatrixArray      &accel_gramians)
323 {
324     #pragma omp master
325     {
326         for(std::size_t i=0; i<TnsSize; ++i)
327         {
328             accel_factors_T[i] = mv.old_factors_T[i] + accel_step
329                 * (mv.factors_T[i] -
330                 mv.old_factors_T[i]);
331             accel_gramians[i] = accel_factors_T[i]
332                 * accel_factors_T[i].transpose();
333         }
334         f_accel = 0;
335     }
336     #pragma omp barrier
337     double f_accel_loc = accel_cost_function(mv, accel_factors_T);
338
339     #pragma omp atomic
340     f_accel += f_accel_loc;
341
342     #pragma omp barrier
343
344     #pragma omp master
345     {
346         if (st.f_value > f_accel)
347         {
348             mv.factors_T = accel_factors_T;
349             mv.factor_T_factor = accel_gramians;
350             st.f_value = f_accel;
351             Partensor()->Logger()->info("Acceleration Step SUCCEEDED! at iter: {}", st.ao_iter);
352         }
353         else
354             st.options.accel_fail++;
355
356         if (st.options.accel_fail==5)
357         {
358             st.options.accel_fail=0;
359             st.options.accel_coeff++;
360         }
361     }
362 }
363
364 /*
365  * Sequential implementation of Alternating Least Squares (ALS) method.
366  *
367  * @param R [in] The rank of decomposition.
368  * @param mv [in] Struct where ALS variables are stored and being updated
369  *                until a termination condition is true.
370  * @param st [in,out] Struct where the returned values of @c Gtc are stored.

```

```

371     */
372     void aogtc_stochastic(Member_Variables &mv,
373                          Status &status)
374     {
375         double f_accel = 0.0; // Objective Value after the acceleration step
376         double accel_step = 0.0;
377
378         MatrixArray accel_factors_T;
379         MatrixArray accel_gramians;
380
381         for (std::size_t i=0; i<TnsSize; i++)
382         {
383             mv.Y[i] = Matrix::Zero(mv.rank, mv.tnsDims[i]);
384             mv.factors_T[i] = status.factors[i].transpose();
385             mv.mtkrp_T[i] = Matrix(mv.rank, mv.tnsDims[i]);
386             mv.factor_T_factor[i].noalias() = mv.factors_T[i] * status.factors[i];
387             accel_factors_T[i] = mv.factors_T[i];
388             accel_gramians[i] = Matrix::Zero(mv.rank, mv.rank);
389         }
390
391         // if(status.options.normalization)
392         // {
393         //     choose_normilization_factor(status, mv.all_orthogonal, mv.weight_factor);
394         // }
395         // Normalize(static_cast<int>(R), mv.factor_T_factor, status.factors);
396
397         std::size_t epoch = static_cast<std::size_t>(1/mv.c_stochastic_perc);
398         std::size_t epoch_counter = 0;
399
400         const int total_num_threads = get_num_threads();
401         omp_set_nested(0);
402
403         status.frob_tns = (mv.tnsX[0]).squaredNorm();
404
405         #pragma omp parallel n
406             num_threads(total_num_threads) n
407             proc_bind(spread) n
408             default(shared)
409         {
410             cost_function(mv, status);
411             #pragma omp barrier
412
413             #pragma omp master
414             {
415                 status.rel_costFunction = status.f_value / status.frob_tns;
416             }
417             #pragma omp barrier
418
419             // ---- Loop until ALS converges ----
420             while(1)
421             {
422                 #pragma omp master
423                 {
424                     status.ao_iter++;
425                     Partensor()->Logger()->info("iter: {} -- fvalue: {} -- relative_costFunction: {}",
426 status.ao_iter,
427                                     status.f_value, status.rel_costFunction);
428                 }
429
430                 for (std::size_t i=0; i<TnsSize; i++)
431                 {
432                     // Update factor
433                     update_factor(i, mv, status);
434                 }
435
436                 // <----- End loop for every mode ----->
437                 // Cost function computation.
438                 if (status.ao_iter % epoch == 0)
439                 {
440                     #pragma omp master
441                     {
442                         epoch_counter++;
443                     }
444
445                     #pragma omp barrier
446
447                     cost_function(mv, status);
448
449                     #pragma omp barrier
450
451                     #pragma omp master
452                     {
453                         status.rel_costFunction = status.f_value/status.frob_tns;
454                     }
455                 }

```

```

456
457     #pragma omp barrier
458
459     // ---- Terminating condition ----
460     if (status.rel_costFunction < status.options.threshold_error || epoch_counter >=
status.options.max_iter)
461     {
462         #pragma omp master
463         {
464             if (status.options.writeToFile)
465                 writeFactorsToFile(status);
466         }
467         break;
468     }
469
470     // if (status.options.acceleration)
471     // {
472     //     // ---- Acceleration Step ----
473     //     if (status.ao_iter > 1)
474     //     {
475     //         #pragma omp master
476     //         accel_step = pow(status.ao_iter+1,(1.0/(status.options.accel_coeff)));
477     //     }
478     //     line_search_accel(mv, status, f_accel, accel_step, accel_factors_T,
accel_gramians);
479     //     #pragma omp barrier
480     // }
481     // #pragma omp master
482     // {
483     //     for (int i = 0; i < static_cast<int>(TnsSize); i++)
484     //         mv.old_factors_T[i] = mv.factors_T[i];
485     // }
486     // }
487 } // end of while
488 } // end of pragma
489 }
490
491 void initialize_factors(Member_Variables &mv,
492                        Status &status)
493 {
494     if(status.options.initialized_factors)
495     {
496         if(status.options.read_factors_from_file)
497         {
498             for(std::size_t i=0; i<TnsSize; ++i)
499             {
500                 status.factors[i] = Matrix(mv.tnsDims[i], static_cast<int>(mv.rank));
501                 read( status.options.initial_factors_paths[i],
502                     mv.tnsDims[i] * mv.rank,
503                     0,
504                     status.factors[i] );
505             }
506         }
507         else
508             status.factors = status.options.factorsInIt;
509     }
510     else // produce estimate factors using uniform distribution with entries in [0,1].
511         makeFactors(mv.tnsDims, status.options.constraints, mv.rank, status.factors);
512 }
513
514 Status operator()(Options const &options)
515 {
516     Status status(options);
517     Member_Variables mv(options.rank, options.tnsDims);
518
519     long long int fileSize = (TnsSize + 1) * options.nonZeros;
520
521     Matrix Ratings_Base_T = Matrix(static_cast<int>(TnsSize+1), options.nonZeros);
522     // Read the whole Tensor from a file
523     read( options.ratings_path,
524         fileSize,
525         0,
526         Ratings_Base_T );
527
528     // GTC_STOCHASTIC_Base<TnsSize>::sort_ratings_base(mv.Ratings_Base_T, options.nonZeros);
529     // ReserveSparseTensor<TnsSize>(mv.tnsX, options.tnsDims, options.nonZeros);
530     // FillSparseTensor<TnsSize>(mv.tnsX, options.nonZeros, mv.Ratings_Base_T, options.tnsDims);
531     // mv.Ratings_Base_T.resize(0,0);
532     GTC_STOCHASTIC_Base<TnsSize>::sort_ratings_base(Ratings_Base_T, mv.tnsX, options.tnsDims,
options.nonZeros);
533     Ratings_Base_T.resize(0,0);
534
535     for (std::size_t i=0; i<TnsSize; i++)
536     {
537         mv.tnsX[i].makeCompressed();
538     }
539 }
540

```

```

553         mv.c_stochastic_perc = options.c_stochastic_perc;
554
555         calculate_offsets(mv);
556
557         initialize_factors(mv, status);
558
559         aogtc_stochastic(mv, status);
560
561         return status;
562     }
563
564     Status operator()(Matrix const &Ratings_Base_T,
565                      Options const &options)
566     {
567         Status status(options);
568         Member_Variables mv(options.rank, options.tnsDims);
569
570         // ReserveSparseTensor<TnsSize>(mv.tnsX, options.tnsDims, options.nonZeros);
571         // FillSparseTensor<TnsSize>(mv.tnsX, options.nonZeros, Ratings_Base_T, options.tnsDims);
572         // Ratings_Base_T.resize(0,0);
573         GTC_STOCHASTIC_Base<TnsSize>::sort_ratings_base(Ratings_Base_T, mv.tnsX, options.tnsDims,
options.nonZeros);
574         // Ratings_Base_T.resize(0,0);
575
576         for (std::size_t i=0; i<TnsSize; i++)
577         {
578             mv.tnsX[i].makeCompressed();
579         }
580
581         mv.c_stochastic_perc = options.c_stochastic_perc;
582
583         calculate_offsets(mv);
584
585         // produce estimate factors using uniform distribution with entries in [0,1].
586         initialize_factors(mv, status);
587
588         partensor::timer.startChronoHighTimer();
589         aogtc_stochastic(mv, status);
590         double end_gtc_time_omp = partensor::timer.endChronoHighTimer();
591         std::cout << "GtcStochasticOpenMP took " << end_gtc_time_omp << " sec." << std::endl;
592         return status;
593     }
594 };
595 } // namespace internal
596 } // namespace v1
597 } // 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.](#)

```
1 #ifndef DOXYGEN_SHOULD_SKIP_THIS
15 #endif // DOXYGEN_SHOULD_SKIP_THIS
16 / ***** /
25 #ifndef PARTENSOR_KHATRI_RAO_HPP
26 #define PARTENSOR_KHATRI_RAO_HPP
27
28 #include "PARTENSOR_basic.hpp"
29 #include "unsupported/Eigen/KroneckerProduct"
30
31 #if __has_include("tbb/parallel_for.h")
32     #include "boost/range/irange.hpp"
33     #include "tbb/parallel_for.h"
34 #endif / * __has_include("tbb/parallel_for.h") */
35
36 // #if USE_TBB
37 //     #include "boost/range/irange.hpp"
38 //     #include "tbb/parallel_for.h"
39 // #endif / * USE_TBB */
40
41 namespace partensor
42 {
43     inline namespace v1 {
44
45         #ifndef DOXYGEN_SHOULD_SKIP_THIS
46
47         namespace internal {
48
49             template <typename ...Matrices>
50             Matrix KhatriRao_seq( Matrix const &mat1,
51                                 Matrix const &mat2,
52                                 Matrices const &...mats )
53
54             {
55                 if constexpr (sizeof... (mats) == 0)
56                 {
57                     int R = mat1.cols();
58                     int I1 = mat1.rows();
59                     int I2 = mat2.rows();
60
61                 }
62             }
63
64         }
65
66     }
67
68 }
69
70
```

```

71         Matrix res(l1 * l2,R);
72
73         for (int i=0; i < R; i++)
74         {
75             res.block(0,i,l1 * l2,1) =
kroneckerProduct(mat1.block(0,i,l1,1),mat2.block(0,i,l2,1));
76         }
77
78         return res;
79     }
80     else
81     {
82         auto _temp = KhatriRao_seq(mat2,mats...);
83
84         return KhatriRao_seq(mat1,_temp);
85     }
86 }
87 // end namespace internal
88
89 #endif // DOXYGEN_SHOULD_SKIP_THIS
90
91 #ifndef DOXYGEN_SHOULD_SKIP_THIS
111 template <typename ExecutionPolicy, typename ...Matrices>
112 execution::internal::enable_if_execution_policy<ExecutionPolicy,Matrix>
113 KhatriRao( ExecutionPolicy      &&,
114           Matrix               const &mat1,
115           Matrix               const &mat2,
116           Matrices             const &...mats )
117 {
118     return internal::KhatriRao_seq(mat1,mat2,mats...);
119 }
120
121 #endif // DOXYGEN_SHOULD_SKIP_THIS
122
123 template <typename ...Matrices>
124 Matrix KhatriRao( Matrix const &mat1,
125                 Matrix const &mat2,
126                 Matrices const &...mats )
127 {
128     return KhatriRao(execution::seq,mat1,mat2,mats...);
129 }
130
131 } // namespace v1
132
133 #if __has_include("tbb/parallel_for.h")
134 #ifndef DOXYGEN_SHOULD_SKIP_THIS
135 namespace experimental {
136     inline namespace v1{
137         namespace internal {
138             template <typename ...Matrices>
139             Matrix KhatriRao_par( Matrix const &mat1,
140                                 Matrix const &mat2,
141                                 Matrices const &...mats )
142             {
143                 if constexpr (sizeof... (mats) == 0)
144                 {
145                     int R = mat1.cols();
146                     int l1 = mat1.rows();
147                     int l2 = mat2.rows();
148
149                     Matrix res(l1 * l2,R);
150                     // auto r = boost::irange(0,R);
151
152                     tbb::parallel_for(tbb::blocked_range<std::size_t>(0, R),
153                                     [&](const
154                                     tbb::blocked_range<size_t>& r) {
155                         for (std::size_t
156                             i = r.begin(); i != r.end(); ++i)
157                             res.block(0,i,l1 * l2,1) = kroneckerProduct(mat1.block(0,i,l1,1),mat2.block(0,i,l2,1));
158                     });
159                 }
160                 return res;
161             }
162             else
163             {
164                 //int R = mat1.cols();
165                 //int l1 = (mats.rows() * ... * mat2.rows());
166
167                 auto _temp = KhatriRao_par(mat2,mats...);
168
169                 return KhatriRao_par(mat1,_temp);
170             }
171         }
172     }
173 }
174
175 #endif // DOXYGEN_SHOULD_SKIP_THIS
176
177 #endif // DOXYGEN_SHOULD_SKIP_THIS

```

```

195     } // namespace internal
196
197     template <typename ExecutionPolicy, typename ...Matrices>
198     execution::internal::enable_if_execution_policy<ExecutionPolicy,Matrix>
199     KhatriRao( ExecutionPolicy      &&,
200               Matrix              const &mat1,
201               Matrix              const &mat2,
202               Matrices            const &...mats )
203     {
204         using ExPolicy =
205         std::remove_cv_t<std::remove_reference_t<ExecutionPolicy>>;
206
207         if constexpr (std::is_same_v<ExPolicy,execution::sequenced_policy>)
208         {
209             return partensor::v1::internal::KhatriRao_seq(mat1,mat2,mats...);
210         }
211         else if constexpr (std::is_same_v<ExPolicy,execution::parallel_policy>)
212         {
213             return internal::KhatriRao_par(mat1,mat2,mats...);
214         }
215         else
216         {
217             return partensor::v1::internal::KhatriRao_seq(mat1,mat2,mats...);
218         }
219     }
220 } // end namespace
221
222 namespace v2 {
223     namespace internal {
224         template <typename ...Matrices>
225         int KhatriRao_seq( Matrix      &res,
226                           int         I1,
227                           Matrix      const &mat2,
228                           Matrices   const &...mats )
229         {
230             const int R = mat2.cols();
231             int I2 = mat2.rows();
232
233             [[maybe_unused]] int I = I1 * I2 * (mats.rows() * ... * 1);
234
235             for (int i=0; i < R; i++)
236             {
237                 res.block(0,i,I1 * I2,1) =
238                 kroneckerProduct(res.block(0,i,I1,1),mat2.block(0,i,I2,1)).eval();
239             }
240
241             I1 *= I2;
242
243             if constexpr (sizeof... (mats) == 0)
244             {
245                 return I1;
246             }
247             else
248             {
249                 return KhatriRao_seq(res, I1, mats...);
250             }
251         }
252     }
253
254     template <typename ...Matrices>
255     Matrix KhatriRao_seq( Matrix      const &mat1,
256                           Matrix      const &mat2,
257                           Matrices   const &...mats )
258     {
259         int R = mat1.cols();
260         int I = mat1.rows() * mat2.rows() * (mats.rows() * ... * 1);
261
262         Matrix res(I,R);
263
264         res.block(0,0,mat1.rows(),mat1.cols()) = mat1;
265
266         int I_KR = KhatriRao_seq(res,mat1.rows(),mat2,mats...);
267         assert(I == I_KR);
268
269         return res;
270     }
271
272     template <typename DT = DefaultDataType, typename ...Matrices>
273     int KhatriRao_par( Matrix      &res,
274                       int         I1,
275                       Matrix      const &mat2,
276                       Matrices   const &...mats )
277     {
278         const int R = mat2.cols();

```



```

339         int      l2 = mat2.rows();
340
341         // Matrix res(l1 * l2,R);
342         // auto      r = boost::irange(0,R);
343
344         tbb::parallel_for(tbb::blocked_range<std::size_t>(0, R),
345             [&](const tbb::blocked_range<size_t>& r) {
346             Eigen::VectorXd::Zero(l2,1);
347
348                 for (std::size_t j = r.begin(); j != r.end();
349                     j++)
350                 {
351                     temp = mat2.col(j);
352                     for (auto i = 0; i < l1;
353                         i++)//std::size_t
354                     {
355                         res(i,j) * temp;
356                     }
357                 }
358
359                 l1 *= l2;
360
361                 if constexpr (sizeof... (mats) == 0)
362                 {
363                     return l1;
364                 }
365                 else
366                 {
367                     return KhatriRao_par(res, l1, mats...);
368                 }
369             }
370
371     template <typename DT = DefaultDataType, typename ...Matrices>
372     Matrix KhatriRao_par( Matrix const &mat1,
373                         Matrix const &mat2,
374                         Matrices const &...mats )
375     {
376         int R = mat1.cols();
377         int l = mat1.rows() * mat2.rows() * (mats.rows() * ... * 1);
378
379         Matrix res(l,R);
380
381         res.block(l-mat1.rows(),0,mat1.rows(),mat1.cols()) = mat1;
382
383         int l_KR = KhatriRao_par(res,mat1.rows(),mat2,mats...);
384
385         assert(l == l_KR);
386
387         return res;
388     }
389
390     // namespace internal
391
392     template <typename ExecutionPolicy, typename ...Matrices>
393     execution::internal::enable_if_execution_policy<ExecutionPolicy,Matrix>
394     KhatriRao( ExecutionPolicy &&,
395               Matrix const &mat1,
396               Matrix const &mat2,
397               Matrices const &...mats )
398     {
399         using ExPolicy = std::remove_cv_t<std::remove_reference_t<ExecutionPolicy>;
400
401         if constexpr (std::is_same_v<ExPolicy,execution::sequenced_policy>)
402         {
403             return internal::KhatriRao_seq(mat1,mat2,mats...);
404         }
405         else if constexpr (std::is_same_v<ExPolicy,execution::parallel_policy>)
406         {
407             return internal::KhatriRao_par(mat1,mat2,mats...);
408         }
409         else
410         {
411             return internal::KhatriRao_seq(mat1,mat2,mats...);
412         }
413     }
414
415     template <typename ...Matrices>
416     Matrix KhatriRao( Matrix const &mat1,
417                     Matrix const &mat2,
418                     Matrices const &...mats
419                 )
420     {
421         return KhatriRao(execution::seq,mat1,mat2,mats...);
422     }
423
424     // end namespace v2
425
426     namespace v3 {

```

```

459         namespace internal {
460
475             template <typename DT = DefaultDataType, typename ...Matrices>
476             int KhatriRao_seq( Matrix      &res,
477                               int          I1,
478                               Matrix const &mat2,
479                               Matrices const &...mats )
480             {
481                 const int R = mat2.cols();
482                 int      I2 = mat2.rows();
483                 int      I  = I1 * I2 * (mats.rows() * ... * 1);
484
485                 for (int i=0; i < R; i++)
486                 {
487                     Eigen::Map<Matrix, 0,
Eigen::InnerStride<>(res.data()+i
=
488                     Eigen::InnerStride<>(res.data()+i
mat2.block(0,i,I2,1));
489
490
491                     I1 *= I2;
492
493                     if constexpr (sizeof... (mats) == 0)
494                     {
495                         return I1;
496                     }
497                     else
498                     {
499                         return KhatriRao_seq(res, I1, mats...);
500                     }
501                 }
502
503             template <typename ...Matrices>
504             Matrix KhatriRao_seq( Matrix const &mat1,
505                                  Matrix const &mat2,
506                                  Matrices const &...mats)
507             {
508                 int R = mat1.cols();
509                 int I = mat1.rows() * mat2.rows() * (mats.rows() * ... * 1);
510
511                 Matrix res(I,R);
512
513                 Eigen::Map<Matrix, 0, Eigen::InnerStride<>(res.data(),
mat1.rows(), R, Eigen::InnerStride<>(res.innerStride()
* (I/mat1.rows())))) = mat1;
514
515                 int I_KR = KhatriRao_seq(res,mat1.rows(),mat2,mats...);
516
517                 assert(I == I_KR);
518
519                 return res;
520             }
521
522             template <typename ...Matrices>
523             int KhatriRao_par( Matrix      &res,
524                               int          I1,
525                               Matrix const &mat2,
526                               Matrices const &...mats )
527             {
528                 const int R = mat2.cols();
529                 int      I2 = mat2.rows();
530
531                 // Matrix res(I1 * I2,R);
532                 // auto      r = boost::irange(0,R);
533
534                 tbb::parallel_for(tbb::blocked_range<std::size_t>(0, R),
535                                 [&](const tbb::blocked_range<size_t>& r) {
536                     Eigen::VectorXd temp =
537
538                     Eigen::VectorXd::Zero(I2,1);
539
540                     for (std::size_t j = r.begin(); j !=
541                          r.end(); j++)
542                     {
543                         temp = mat2.col(j);
544                         for (auto i = 0; i < I1; i++)
545                         {
546                             res.block(i * I2, j, I2,
547
548                             1).noalias() = res(i,j) * temp;
549
550                         }
551                     }
552                 });
553
554                 I1 *= I2;
555
556                 if constexpr (sizeof... (mats) == 0)
557                 {
558                     return I1;
559                 }
560             }
561         }
562     }
563 }
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579

```

```

580         }
581         else
582         {
583             return KhatriRao_par(res, l1, mats...);
584         }
585     }
586
587     template <typename ...Matrices>
588     Matrix KhatriRao_par( Matrix const &mat1,
589                          Matrix const &mat2,
590                          Matrices const &...mats )
591     {
592         int R = mat1.cols();
593         int l1 = mat1.rows() * mat2.rows() * (mats.rows() * ... * 1);
594
595         Matrix res(l,R);
596
597         res.block(l-mat1.rows(),0,mat1.rows(),mat1.cols()) = mat1;
598
599         int l_KR = KhatriRao_par(res,mat1.rows(),mat2,mats...);
600
601         assert(l == l_KR);
602
603         return res;
604     }
605 } // namespace internal
606
607     template <typename ExecutionPolicy, typename ...Matrices>
608     execution::internal::enable_if_execution_policy<ExecutionPolicy,Matrix>
609     KhatriRao( ExecutionPolicy &&,
610              Matrix const &mat1,
611              Matrix const &mat2,
612              Matrices const &...mats)
613     {
614         using ExPolicy =
615             std::remove_cv_t<std::remove_reference_t<ExecutionPolicy>>;
616
617         if constexpr (std::is_same_v<ExPolicy,execution::sequenced_policy>)
618         {
619             return internal::KhatriRao_seq(mat1,mat2,mats...);
620         }
621         else if constexpr (std::is_same_v<ExPolicy,execution::parallel_policy>)
622         {
623             return internal::KhatriRao_par(mat1,mat2,mats...);
624         }
625         else
626             return internal::KhatriRao_seq(mat1,mat2,mats...);
627     }
628
629     template <typename ...Matrices>
630     Matrix KhatriRao( Matrix const &mat1,
631                     Matrix const &mat2,
632                     Matrices const &...mats )
633     {
634         return KhatriRao(execution::seq,mat1,mat2,mats...);
635     }
636 } // end namespace v3
637
638 // namespace v4 {
639 //     namespace internal {
640 //         template <typename DT = DefaultDataType, typename ...Matrices>
641 //         int KhatriRao_seq(Matrix &res, int l1, Matrix const &mat1, Matrix const &mat2, Matrices
642 //         const &...mats)
643 //         {
644 //             // const int R = mat2.cols();
645 //             // int l2 = mat2.rows();
646 //             // int l = l1 * l2 * (mats.rows() * ... * 1);
647 //             // for (int i=0; i < R; i++)
648 //             // {
649 //                 res.block(0,i,l1 * l2,1) =
650 //                 kroneckerProduct(res.block(0,i,l1,1),mat2.block(0,i,l2,1)).eval();
651 //             // }
652 //             // l1 *= l2;
653 //             // if constexpr (sizeof... (mats) == 0)
654 //             // {
655 //                 // for (int i=0; i < R; i++)
656 //                 // {
657 //                     res.block(0,i,l1 * l2,1) =
658 //                     kroneckerProduct(mat1.block(0,i,l1,1),mat2.block(0,i,l2,1)).eval();
659 //                 // }
660 //             // }
661 //         }
662 //     }
663 // }

```

```

700 // return l1;
701 // }
702 // else
703 // {
704 //     return KhatriRao_seq(res, l1, mats...);
705 // }
706 // for (int i=0; i < R; i++)
707 // {
708 //     Eigen::Map<MatrixType<dataType>, 0, Eigen::InnerStride<> > (res.data()+i
709 // 1, Eigen::InnerStride<>( res.innerStride() * (l1 * l2))) =
710 //     kroneckerProduct( Eigen::Map<MatrixType<dataType>, 0, Eigen::InnerStride<> >
711 // (res.data()+i * l, l1, 1, Eigen::InnerStride<>( res.innerStride() * (l1))), mat2.block(0,i,l2,1));
712 // }
713 // }
714 // }
715 //
716 // template <typename DT = DefaultDataType, typename ...Matrices>
717 // Matrix KhatriRao_seq(Matrix const &mat1, Matrix const &mat2, Matrices
718 // const &...mats)
719 // {
720 //     int R = mat1.cols();
721 //     int l = mat1.rows() * mat2.rows() * (mats.rows() *... * 1);
722 //     Matrix res(l,R);
723 //     res.block(0,0,mat1.rows(),mat1.cols()) = mat1;
724 //
725 //     int l_KR = KhatriRao_seq(res,mat1,mat2,mats...);
726 //     assert(l == l_KR);
727 //     return res;
728 // }
729 //
730 //
731 //
732 // template <typename DT = DefaultDataType, typename ...Matrices>
733 // int KhatriRao_par(Matrix &res, int l1, Matrix const &mat2, Matrices const &...mats)
734 // {
735 //     const int R = mat2.cols();
736 //     int l2 = mat2.rows();
737 //     Matrix res(l1 * l2,R);
738 //     auto r = boost::irange(0,R);
739 //     tbb::parallel_for(tbb::blocked_range<std::size_t>(0, R),
740 // [&](const tbb::blocked_range<size_t>& r) {
741 //         Eigen::VectorXd temp = Eigen::VectorXd::Zero(l2,1);
742 //         for (std::size_t j = r.begin(); j != r.end(); j++)
743 //         {
744 //             temp = mat2.col(j);
745 //             for (std::size_t i = 0; i < l1; i++)
746 //             {
747 //                 res.block(i * l2, j, l2, 1).noalias() = res(i,j) * temp;
748 //             }
749 //         }
750 //     });
751 // }
752 //
753 //
754 //
755 // l1 *= l2;
756 //
757 // if constexpr (sizeof... (mats) == 0)
758 // {
759 //     return l1;
760 // }
761 // else
762 // {
763 //     return KhatriRao_par(res, l1, mats...);
764 // }
765 // }
766 //
767 // template <typename DT = DefaultDataType, typename ...Matrices>
768 // Matrix KhatriRao_par(Matrix const &mat1, Matrix const &mat2, Matrices const &...mats)
769 // {
770 //     int R = mat1.cols();
771 //     int l = mat1.rows() * mat2.rows() * (mats.rows() *... * 1);
772 //     Matrix res(l,R);
773 //     res.block(l-mat1.rows(),0,mat1.rows(),mat1.cols()) = mat1;
774 //     int l_KR = KhatriRao_par(res,mat1.rows(),mat2,mats...);
775 //     assert(l == l_KR);
776 //     return res;
777 // }
778 // } // namespace internal
779 //
780 //
781 //
782 //
783 //

```

```

784         //      template <typename ExecutionPolicy, typename DT = DefaultDataType, typename
...Matrices>
785         //      execution::internal::enable_if_execution_policy<ExecutionPolicy,Matrix>
786         //      KhatriRao(ExecutionPolicy &&, Matrix const &mat1, Matrix const &mat2, Matrices
const &...mats)
787         //      {
788         //      using ExPolicy = std::remove_cv_t<std::remove_reference_t<ExecutionPolicy>;
789         //      if constexpr (std::is_same_v<ExPolicy,execution::sequenced_policy>)
790         //      {
791         //      {
792         //      return internal::KhatriRao_seq(mat1,mat2,mats...);
793         //      }
794         //      else if constexpr (std::is_same_v<ExPolicy,execution::parallel_policy>)
795         //      {
796         //      return internal::KhatriRao_par(mat1,mat2,mats...);
797         //      }
798         //      else
799         //      return internal::KhatriRao_seq(mat1,mat2,mats...);
800         //      }
801         //      template <typename DT = DefaultDataType, typename ...Matrices>
802         //      Matrix KhatriRao(Matrix const &mat1, Matrix const &mat2, Matrices const
&...mats)
803         //      {
804         //      return KhatriRao(execution::seq,mat1,mat2,mats...);
805         //      }
806         //      } // end namespace v4
807     } // end namespace experimental
808 } // end namespace experimental
809 #endif // end of DOXYGEN_SHOULD_SKIP_THIS
810 #endif // end of #if __has_include("tbb/parallel_for.h")
811 } // end namespace partensor
812 #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 `kroneckerProduct` 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.](#)

```

1 #ifndef DOXYGEN_SHOULD_SKIP_THIS
15 #endif // DOXYGEN_SHOULD_SKIP_THIS
16 / *****
26 #ifndef PARTENSOR_KRONECKER_HPP
27 #define PARTENSOR_KRONECKER_HPP
28
29 #include "PARTENSOR_basic.hpp"
30 #include "unsupported/Eigen/KroneckerProduct"
31
32 namespace partensor {
33
34     inline namespace v1 {
35
36         template <typename ...Matrices>
37         Matrix Kronecker( Matrix const &mat1,
38                         Matrix const &mat2,
39                         Matrices const &...mats )
40         {
41             if constexpr (sizeof... (mats) == 0)
42             {
43                 int I1 = mat1.rows();
44                 int I2 = mat1.cols();
45                 int I3 = mat2.rows();
46                 int I4 = mat2.cols();
47
48                 Matrix res(I1 * I3, I2 * I4);
49                 res = Eigen::kroneckerProduct(mat1, mat2);
50
51                 return res;
52             }
53             else
54             {
55                 auto _temp = Kronecker(mat2, mats...);
56                 return Kronecker(mat1, _temp);
57             }
58         }
59     } // namespace v1
60 } // end namespace partensor
61
62 #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 1st 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 1st 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

<code>_TnsSize</code>	Tensor Order of <code>tnsX</code> .
-----------------------	-------------------------------------

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.](#)

```

1  #ifndef DOXYGEN_SHOULD_SKIP_THIS
15 #endif // DOXYGEN_SHOULD_SKIP_THIS
16 /***** /
43 #ifndef PARTENSOR_MATRICIZATION_HPP
44 #define PARTENSOR_MATRICIZATION_HPP
45
46 #include "PARTENSOR_basic.hpp"
47 #include "TensorOperations.hpp"
48
49 namespace partensor {
50
51     #ifndef DOXYGEN_SHOULD_SKIP_THIS
65     template<std::size_t _TnsSize>
66     Eigen::array<int, _TnsSize> permute(std::size_t mode)
67     {
68         Eigen::array<int, _TnsSize> permutation;
69         std::iota(permutation.begin(), permutation.end(), 0); // {0, 1, 2, ..., N}
70         permutation[0] = mode; // {n, 0, 1,
71         ..., n-1, n+1, ..., N}
72         for(std::size_t i=1; i <= mode; i++) {
73             permutation[i] = i - 1;
74         }
75         return permutation;
76     }
77     #endif // DOXYGEN_SHOULD_SKIP_THIS
78     inline namespace v1 {
79
101         template<int _TnsSize>
102         Matrix Matricization( Tensor<_TnsSize> const &tnsX,
103                               std::size_t mode )
104         {
105             using Dimensions = typename Tensor<_TnsSize>::Dimensions; //
106             Type of @c Eigen Tensor Dimensions.
107             const Dimensions& tnsDims = tnsX.dimensions(); // Eigen
108             Array with the lengths of each of Tensor Dimension.
109             Tensor<2> matricedTns; // Temporary @c Eigen Tensor in order to keep the
110             matricized Tensor.
111             auto permutation = partensor::permute<_TnsSize>(mode);
112             // Compute the column dimension for the matricized tensor.
113             long int newColDim = 1;
114             for(int i=0; i<_TnsSize; ++i)
115             {
116                 if(i!=static_cast<int>(mode))
117                     newColDim *= tnsDims[i];
118             }
119             // reshape: View of the input tensor that has been reshaped to the specified new
120             dimensions.
121             // shuffle: A copy of the input tensor whose dimensions have been reordered
122             according to the specified permutation.

```

```

120 Eigen::array<long int, 2> reshape({static_cast<long int>(tnsDims[mode]),
newColDim));
121 matricedTns = tnsX.shuffle(permutation).reshape(reshape);
122 // Map the @c Eigen Tensor to @c Eigen Matrix type.
123 return tensorToMatrix(matricedTns, tnsDims[mode], newColDim);
124 }
125 } // end namespace v1
126 } // end namespace v1
127
128 #ifndef DOXYGEN_SHOULD_SKIP_THIS
129 namespace experimental {
130
131     inline namespace v1 {
132
133         template<int _TnsSize>
134         Matrix Matricization( Tensor<_TnsSize> const &tnsX,
135                               std::size_t const mode )
136         {
137             using Dimensions = typename Tensor<_TnsSize>::Dimensions; //
138             Type of @c Eigen Tensor Dimensions.
139             const Dimensions& tnsDims = tnsX.dimensions(); //
140             Eigen Array with the lengths of each of Tensor Dimension.
141
142             Tensor<2> matricedTns; // Temporary @c Eigen Tensor in order to keep the
143             matricized Tensor.
144
145             auto permutation = partensor::permute<_TnsSize>(mode);
146
147             // Compute the column dimension for the matricized tensor.
148             long int newColDim = 1;
149             for(int i=0; i<_TnsSize; ++i)
150             {
151                 if(!=static_cast<int>(mode))
152                     newColDim *= tnsDims[i];
153             }
154
155             if constexpr (_TnsSize<4)
156             {
157                 // reshape: View of the input tensor that has been reshaped to
158                 the specified new dimensions.
159                 // shuffle: A copy of the input tensor whose dimensions have
160                 been reordered according to the specified permutation.
161                 Eigen::array<long int, 2> newshape({tnsDims[mode], newColDim});
162                 matricedTns = tnsX.shuffle(permutation).reshape(newshape);
163
164                 // Map the @c Eigen Tensor to @c Eigen Matrix type.
165                 return tensorToMatrix(matricedTns, tnsDims[mode], newColDim);
166             }
167             else
168             {
169                 long int matricized_dim = 1;
170                 for(std::size_t i=1; i < permutation.size()-1; ++i)
171                 {
172                     matricized_dim *= tnsDims[permutation[i]];
173                 }
174                 Tensor<_TnsSize-1> cubeTns;
175                 Matrix matricedTns(tnsDims[mode], newColDim);
176                 // Recursive Call for each cube, based on the mode if it is on
177                 the last dimension or not.
178                 if(mode<_TnsSize-1)
179                 {
180                     for(int cubeld=0; cubeld<tnsDims[_TnsSize-1]; cubeld++)
181                     {
182                         // Get a _TnsSize-1 order hypercube from whole
183                         tensor.
184                         cubeTns = tnsX.chip(cubeld,_TnsSize-1);
185                         // Save the matriced 3D Tensor to matricedTns,
186                         after returning from the recursive calls.
187                         matricedTns.block(0, cubeld * matricized_dim,
188 tnsDims[mode], matricized_dim) = Matricization(cubeTns, mode);
189                     }
190                 }
191                 else
192                 {
193                     for(int cubeld=0; cubeld<tnsDims[_TnsSize-2]; cubeld++)
194                     {
195                         // Get a _TnsSize-1 order hypercube from whole
196                         tensor.
197                         cubeTns = tnsX.chip(cubeld,_TnsSize-2);
198                         // Save the matriced 3D Tensor to matricedTns,
199                         after returning from the recursive calls.
200                         matricedTns.block(0, cubeld * matricized_dim,
201 tnsDims[mode], matricized_dim) = Matricization(cubeTns, _TnsSize-2);
202                     }
203                 }
204                 return matricedTns;
205             }
206         }
207     }
208 }
209
210

```

```

211     }
212
213     template<int _TnsSize>
214     Matrix Matricization_3( Tensor<_TnsSize> const &tnsX,
215                           std::size_t mode )
216     {
217         using Dimensions = typename Tensor<_TnsSize>::Dimensions;
218         const Dimensions& tnsDims = tnsX.dimensions();
219         Tensor<2> matricizedTns; // Temporary @c Eigen Tensor in order to keep the
matricized Tensor.
220
221         auto permutation = partensor::permute<_TnsSize>(mode);
222
223         long int newColDim = 1;
224         for(int i=0; i<_TnsSize; ++i)
225         {
226             if(!=static_cast<int>(mode))
227                 newColDim *= tnsDims[i];
228         }
229         Eigen::array<long int, 2> newshape((tnsDims[mode], newColDim));
230         matricizedTns = tnsX.shuffle(permutation).reshape(newshape);
231
232         // Map the @c Eigen Tensor to @c Eigen Matrix type.
233         return tensorToMatrix(matricizedTns, tnsDims[mode], newColDim);
234     }
235
236     template<int _TnsSize>
237     Matrix Matricization_4( Tensor<_TnsSize> const &tnsX,
238                           std::size_t mode )
239     {
240         using Dimensions = typename Tensor<_TnsSize>::Dimensions;
241         const Dimensions& tnsDims = tnsX.dimensions();
242         Tensor<_TnsSize-1> cubeTns; // Temporary @c Eigen Tensor in order to
keep the matricized Tensor.
243
244         auto permutation = partensor::permute<_TnsSize>(mode);
245
246         long int newColDim = 1;
247         for(int i=0; i<_TnsSize; ++i)
248         {
249             if(!=static_cast<int>(mode))
250                 newColDim *= tnsDims[i];
251         }
252         long int matricized_dim = 1;
253         for(std::size_t i=1; i < permutation.size()-1; ++i)
254         {
255             matricized_dim *= tnsDims[permutation[i]];
256         }
257         Matrix matricizedTns(tnsDims[mode], newColDim);
258         // Recursive Call for each cube, based on the mode if it is on the last
dimension or not.
259
260         if(mode<_TnsSize-1)
261         {
262             for(int cubeld=0; cubeld<tnsDims[_TnsSize-1]; cubeld++)
263             {
264                 // Get a _TnsSize-1 order hypercube from whole tensor.
265                 cubeTns = tnsX.chip(cubeld,_TnsSize-1);
266                 // Save the matricized 3D Tensor to matricizedTns, after
returning from the recursive calls.
267                 matricizedTns.block(0, cubeld,
tnsDims[mode], matricized_dim) = Matricization_3(cubeTns, mode);
268             }
269         }
270         else
271         {
272             for(int cubeld=0; cubeld<tnsDims[_TnsSize-2]; cubeld++)
273             {
274                 // Get a _TnsSize-1 order hypercube from whole tensor.
275                 cubeTns = tnsX.chip(cubeld,_TnsSize-2);
276                 // Save the matricized 3D Tensor to matricizedTns, after
returning from the recursive calls.
277                 matricizedTns.block(0, cubeld,
tnsDims[mode], matricized_dim) = Matricization_3(cubeTns, _TnsSize-2);
278             }
279         }
280         return matricizedTns;
281     }
282
283     template<int _TnsSize>
284     Matrix Matricization_5( Tensor<_TnsSize> const &tnsX,
285                           std::size_t mode )
286     {
287         using Dimensions = typename Tensor<_TnsSize>::Dimensions;
288         const Dimensions& tnsDims = tnsX.dimensions();
289         Tensor<_TnsSize-1> cubeTns;
290         auto permutation = partensor::permute<_TnsSize>(mode);

```

```

290
291
std::multiplies<long int>());
292
293
294
295
296
297
298
299
300
dimension or not.
301
302
303
304
305
306
307
returning from the recursive calls.
308
tnsDims[mode], matricized_dim) = Matricization_4(cubeTns, mode);
309
310
311
312
313
314
315
316
317
returning from the recursive calls.
318
tnsDims[mode], matricized_dim) = Matricization_4(cubeTns, _TnsSize-2);
319
320
321
322
}
323
324
template<int _TnsSize>
325
Matrix Matricization_6( Tensor<_TnsSize> const &tnsX,
326
                        std::size_t const mode )
327
{
328
    using Dimensions = typename Tensor<_TnsSize>::Dimensions;
329
    const Dimensions& tnsDims = tnsX.dimensions();
330
331
    Tensor<_TnsSize-1> cubeTns;
332
    auto permutation = partensor::permute<_TnsSize>(mode);
333
334
    long int newColDim = std::accumulate(tnsDims.begin(), tnsDims.end(), 1,
std::multiplies<long int>());
335
336
    newColDim /= tnsDims[mode];
337
338
    long int matricized_dim = 1;
339
    for(std::size_t i=1; i < permutation.size()-1; ++i)
340
    {
341
        matricized_dim *= tnsDims[permutation[i]];
342
    }
343
    Matrix matricedTns(tnsDims[mode], newColDim);
    // Recursive Call for each cube, based on the mode if it is on the last
dimension or not.
344
    if(mode<_TnsSize-1)
345
    {
346
        for(int cubeld=0; cubeld<tnsDims[_TnsSize-1]; cubeld++)
347
        {
348
            // Get a _TnsSize-1 order hypercube from whole tensor.
349
            cubeTns = tnsX.chip(cubeld,_TnsSize-1);
350
            // Save the matriced 3D Tensor to matricedTns, after
returning from the recursive calls.
351
            matricedTns.block(0, cubeld * matricized_dim,
tnsDims[mode], matricized_dim) = Matricization_5(cubeTns, mode);
352
        }
353
    }
354
    else
355
    {
356
        for(int cubeld=0; cubeld<tnsDims[_TnsSize-2]; cubeld++)
357
        {
358
            // Get a _TnsSize-1 order hypercube from whole tensor.
359
            cubeTns = tnsX.chip(cubeld,_TnsSize-2);
360
            // Save the matriced 3D Tensor to matricedTns, after
returning from the recursive calls.
361
            matricedTns.block(0, cubeld * matricized_dim,
tnsDims[mode], matricized_dim) = Matricization_5(cubeTns, _TnsSize-2);
362
        }
363
    }
364
    return matricedTns;

```

```

365     }
366
367     template<int _TnsSize>
368     Matrix Matricization_7( Tensor<_TnsSize> const &tnsX,
369                            std::size_t const mode )
370     {
371         using Dimensions = typename Tensor<_TnsSize>::Dimensions;
372         const Dimensions& tnsDims = tnsX.dimensions();
373
374         Tensor<_TnsSize-1> cubeTns;
375         auto permutation = partensor::permute<_TnsSize>(mode);
376
377         long int newColDim = std::accumulate(tnsDims.begin(), tnsDims.end(), 1,
378 std::multiplies<long int>());
379         newColDim /= tnsDims[mode];
380
381         long int matricized_dim = 1;
382         for(std::size_t i=1; i < permutation.size()-1; ++i)
383         {
384             matricized_dim *= tnsDims[permutation[i]];
385         }
386         Matrix matricedTns(tnsDims[mode], newColDim);
387         // Recursive Call for each cube, based on the mode if it is on the last
388         dimension or not.
389         if(mode<_TnsSize-1)
390         {
391             for(int cubeld=0; cubeld<tnsDims[_TnsSize-1]; cubeld++)
392             {
393                 // Get a _TnsSize-1 order hypercube from whole tensor.
394                 cubeTns = tnsX.chip(cubeld,_TnsSize-1);
395                 // Save the matriced 3D Tensor to matricedTns, after
396                 returning from the recursive calls.
397                 matricedTns.block(0, cubeld * matricized_dim,
398 tnsDims[mode], matricized_dim) = Matricization_6(cubeTns, mode);
399             }
400         }
401         else
402         {
403             for(int cubeld=0; cubeld<tnsDims[_TnsSize-2]; cubeld++)
404             {
405                 // Get a _TnsSize-1 order hypercube from whole tensor.
406                 cubeTns = tnsX.chip(cubeld,_TnsSize-2);
407                 // Save the matriced 3D Tensor to matricedTns, after
408                 returning from the recursive calls.
409                 matricedTns.block(0, cubeld * matricized_dim,
410 tnsDims[mode], matricized_dim) = Matricization_6(cubeTns, _TnsSize-2);
411             }
412         }
413         return matricedTns;
414     }
415
416     template<int _TnsSize>
417     Matrix Matricization_8( Tensor<_TnsSize> const &tnsX,
418                            std::size_t const mode )
419     {
420         using Dimensions = typename Tensor<_TnsSize>::Dimensions;
421         const Dimensions& tnsDims = tnsX.dimensions();
422
423         Tensor<_TnsSize-1> cubeTns;
424         auto permutation = partensor::permute<_TnsSize>(mode);
425
426         long int newColDim = std::accumulate(tnsDims.begin(), tnsDims.end(), 1,
427 std::multiplies<long int>());
428         newColDim /= tnsDims[mode];
429
430         long int matricized_dim = 1;
431         for(std::size_t i=1; i < permutation.size()-1; ++i)
432         {
433             matricized_dim *= tnsDims[permutation[i]];
434         }
435         Matrix matricedTns(tnsDims[mode], newColDim);
436         // Recursive Call for each cube, based on the mode if it is on the last
437         dimension or not.
438         if(mode<_TnsSize-1)
439         {
440             for(int cubeld=0; cubeld<tnsDims[_TnsSize-1]; cubeld++)
441             {
442                 // Get a _TnsSize-1 order hypercube from whole tensor.
443                 cubeTns = tnsX.chip(cubeld,_TnsSize-1);
444                 // Save the matriced 3D Tensor to matricedTns, after
445                 returning from the recursive calls.
446                 matricedTns.block(0, cubeld * matricized_dim,
447 tnsDims[mode], matricized_dim) = Matricization_7(cubeTns, mode);
448             }
449         }
450         else
451         {

```

```

442                                     for(int cubeld=0; cubeld<tnsDims[_TnsSize-2]; cubeld++)
443                                     {
444                                         // Get a _TnsSize-1 order hypercube from whole tensor.
445                                         cubeTns = tnsX.chip(cubeld,_TnsSize-2);
446                                         // Save the matriced 3D Tensor to matricedTns, after
returning from the recursive calls.
447                                     matricedTns.block(0, cubeld * matricized_dim,
tnsDims[mode], matricized_dim) = Matricization_7(cubeTns, _TnsSize-2);
448                                     }
449                                 }
450                                 return matricedTns;
451                             }
452                         } // end namespace v1
453                     } // end namespace v2 {
454                 namespace v2 {
455                     template<typename Tensor_>
456                     typename TensorTraits<Tensor_>::MatrixType Matricization( Tensor_ const
&tnsX,
457                                     std::size_t const mode )
458                 {
459                     // using DataType = typename TensorTraits<Tensor_>::DataType; //
Type of @c Eigen Tensor Data (e.g double, float, etc.).
470                     constexpr std::size_t TnsSize = TensorTraits<Tensor_>::TnsSize;
471                     using Dimensions = typename
TensorTraits<Tensor_>::Dimensions; // Type of @c Eigen Tensor Dimensions.
472                     const Dimensions& tnsDims = tnsX.dimensions(); // Eigen Array with the
lengths of each of Tensor Dimension.
473                     Tensor<2> matricedTns; // Temporary @c Eigen
Tensor in order to keep the matricized Tensor.
474                     auto permutation = partensor::permute<TnsSize>(mode);
475                     // Compute the column dimension for the matricized tensor.
476                     long int newColDim = 1;
477                     for(std::size_t i=0; i<TnsSize; ++i)
478                     {
479                         if(i!=mode)
480                             newColDim *= tnsDims[i];
481                     }
482                     // reshape: View of the input tensor that has been reshaped to the
483                     // specified new dimensions.
484                     // shuffle: A copy of the input tensor whose dimensions have been
485                     // reordered according to the specified permutation.
486                     Eigen::array<long int, 2> reshape((tnsDims[mode], newColDim));
487                     matricedTns = tnsX.shuffle(permutation).reshape(reshape);
488                     // Map the @c Eigen Tensor to @c Eigen Matrix type.
489                     return tensorToMatrix(matricedTns, tnsDims[mode], newColDim);
490                 }
491             }
492         }
493     }
494     template<typename Tensor_>
495     typename TensorTraits<Tensor_>::MatrixType Matricization_rec( Tensor_ const
&tnsX,
496                                     std::size_t const mode )
497     {
498         // using DataType = typename TensorTraits<Tensor_>::DataType;
// Type of @c Eigen Tensor Data (e.g double, float, etc.).
499         using MatrixType = typename TensorTraits<Tensor_>::MatrixType;
500         // Type of @c Eigen Matrix based on Tensor Type.
501         using Dimensions = typename TensorTraits<Tensor_>::Dimensions;
502         // Type of @c Eigen Tensor Dimensions.
503         static constexpr std::size_t TnsSize = TensorTraits<Tensor_>::TnsSize;
504         // Tensor Order.
505         const Dimensions& tnsDims = tnsX.dimensions(); // Eigen Array with
the lengths of each of Tensor Dimension.
506         Tensor<2> matricedTns; // Temporary @c
Eigen Tensor in order to keep the matricized Tensor.
507         auto permutation = partensor::permute<TnsSize>(mode);
508         // Compute the column dimension for the matricized tensor.
509         long int newColDim = std::accumulate(tnsDims.begin(), tnsDims.end(), 1,
std::multiplies<long int>());
510         newColDim /= tnsDims[mode];
511         if constexpr (TnsSize<4)
512         {
513             // reshape: View of the input tensor that has been reshaped to
the specified new dimensions.
514             // shuffle: A copy of the input tensor whose dimensions have
been reordered according to the specified permutation.
515             Eigen::array<long int, 2> newshape((tnsDims[mode], newColDim));

```

```

535         matricedTns = tnsX.shuffle(permutation).reshape(newshape);
536
537         // Map the @c Eigen Tensor to @c Eigen Matrix type.
538         return tensorToMatrix(matricedTns, tnsDims[mode], newColDim);
539     }
540     else
541     {
542         long int matricized_dim = 1;
543         for(std::size_t i=1; i < permutation.size()-1; ++i)
544         {
545             matricized_dim *= tnsDims[permutation[i]];
546         }
547         Tensor<TnsSize-1> cubeTns;
548         MatrixType matricedTns(tnsDims[mode], newColDim);
549         // Recursive Call for each cube, based on the mode if it is on
550         the last dimension or not.
551         if(mode<TnsSize-1)
552         {
553             for(int cubeld=0; cubeld<tnsDims[TnsSize-1]; cubeld++)
554             {
555                 // Get a TnsSize-1 order hypercube from whole
556                 tensor. cubeTns = tnsX.chip(cubeld,TnsSize-1);
557                 // Save the matriced 3D Tensor to matricedTns,
558                 after returning from the recursive calls.
559                 matricedTns.block(0, cubeld * matricized_dim,
560 tnsDims[mode], matricized_dim) = Matricization_rec(cubeTns, mode);
561             }
562             else
563             {
564                 for(int cubeld=0; cubeld<tnsDims[TnsSize-2]; cubeld++)
565                 {
566                     // Get a TnsSize-1 order hypercube from whole
567                     tensor. cubeTns = tnsX.chip(cubeld,TnsSize-2);
568                     // Save the matriced 3D Tensor to matricedTns,
569                     after returning from the recursive calls.
570                     matricedTns.block(0, cubeld * matricized_dim,
571 tnsDims[mode], matricized_dim) = Matricization_rec(cubeTns, TnsSize-2);
572                 }
573             }
574             return matricedTns;
575         }
576     }
577 } // end namespace v2
578 } // end namespace experimental
579 #endif // DOXYGEN_SHOULD_SKIP_THIS
580 } // end namespace partensor
581
582 #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

<code>_TnsSize</code>	Tensor Order.
<code>Dimensions</code>	Array type containing the Tensor dimensions.

Parameters

<code>tnsDims</code>	[in] Stl array containing the Tensor dimensions, whose length must be same as the Tensor order.
<code>factors</code>	[in] An stl array with the factors.
<code>tns_mat</code>	[in] Matricization of the Tensor based on mode.
<code>mode</code>	[in] The dimension where the tensor was matricized and the MTTKRP will be computed.
<code>num_threads</code>	[in] The number of available threads, defined by the environmental variable <code>OMP_NUM_THREADS</code>
<code>result</code>	[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 nally calculate the MTTKRP.

Template Parameters

<code>_TnsSize</code>	Size of the factors	array.
-----------------------	---------------------	--------

Parameters

<code>factors</code>	[in] An <code>std::array</code> with the factors.
<code>tns_mat</code>	[in] Matricization of the Tensor based on mode.
<code>mode</code>	[in] The dimension where the tensor was matricized and the MTTKRP will be computed.
<code>krao</code>	[in,out] The result Khatri-Rao product for the factors excluding the mode factor.
<code>result</code>	[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.](#)

```

1 #ifndef DOXYGEN_SHOULD_SKIP_THIS
15 #endif // DOXYGEN_SHOULD_SKIP_THIS
16 / ****
23 #ifndef PARTENSOR_TENSOR_KHATRIRAO_PRODUCT_HPP
24 #define PARTENSOR_TENSOR_KHATRIRAO_PRODUCT_HPP
25
26 #include "PARTENSOR_basic.hpp"
27 #include "PartialKhatriRao.hpp"
28 #include <omp.h>
29
30 // #define EIGEN_DONT_PARALLELIZE
31 // #define NUM_SOCKETS 1
32 #define STATIC_SCHEDULE_CHUNK_SIZE 1
33 /* -- Declare custom reduction of variable_type : Matrix -- */
34 #pragma omp declare reduction(sum : Eigen::MatrixXd n
35                               : Eigen::MatrixXd n
36                               : omp_out = omp_out + omp_in) n
37     initializer(omp_priv = Eigen::MatrixXd::Zero(omp_orig.rows(), omp_orig.cols()))
38
39 namespace partensor {
40
41     inline namespace v1 {
42
43     template<std::size_t _TnsSize>
44     void mtkrp(std::array<Matrix,_TnsSize> const &factors,
45              Matrix const &tns_mat,
46              int const &mode,
47              Matrix &krao,
48              Matrix &result)
49     {
50         krao = PartialKhatriRao(factors, mode);
51         result.noalias() = tns_mat * krao;
52     }
53
54     /*
55     * Get number of threads, defined by the environmental variable $(OMP_NUM_THREADS).
56     */
57     inline int get_num_threads()
58     {
59         int threads;
60         #pragma omp parallel
61         {
62             threads = omp_get_num_threads();
63         }
64         return threads;
65     }
66
67     template <std::size_t _TnsSize, typename Dimensions>
68     void mtkrp(Dimensions const &tnsDims,
69              std::array<Matrix,_TnsSize> const &factors,
70              Matrix const &tns_mat,
71              std::size_t const mode,
72              int const num_threads,
73              Matrix &result)
74     {
75         #ifndef EIGEN_DONT_PARALLELIZE

```

```

111     Eigen::setNbThreads(1);
112 #endif
113
114 constexpr std::size_t NUM_SOCKETS = 1;
115 std::size_t inner_num_threads = num_threads / NUM_SOCKETS;
116 if (inner_num_threads < 1)
117     inner_num_threads = 1;
118
119 const int rank = factors[0].cols();
120 std::size_t last_mode = (mode == _TnsSize-1) ? (_TnsSize - 2) : (_TnsSize - 1);
121 std::size_t first_mode = (mode == 0) ? 1 : 0;
122
123 result = Matrix::Zero(tnsDims[mode], rank);
124
125 // dim = I_(1) * ... * I_(mode-1) * I_(mode+1) * ... * I_(N)
126 long int dim = 1;
127 for(std::size_t i=0; i<_TnsSize; ++i)
128 {
129     if(i!=mode)
130         dim *= tnsDims[i];
131 }
132 // I_(first_mode+1) x I_(first_mode+2) x ... x I_(last_mode), where <I_(first_mode)> #rows of the
starting factor.
133 int num_of_blocks = static_cast<int>(dim / static_cast<long
int>(tnsDims[first_mode]));
134 int numOfBlocks_div_NumSockets = num_of_blocks / NUM_SOCKETS;
135 std::array<int, _TnsSize-2> rows_offset;
136
137 for (int i = static_cast<int>(_TnsSize - 3), j = last_mode; i >= 0; i--, j--)
138 {
139     if (j == static_cast<int>(mode))
140     {
141         j--;
142     }
143     if (i == static_cast<int>(_TnsSize - 3))
144     {
145         rows_offset[j] = num_of_blocks / tnsDims[j];
146     }
147     else
148     {
149         rows_offset[j] = rows_offset[j + 1] / tnsDims[j];
150     }
151 }
152
153 // --- If Factors are Transposed ---
154 // omp_set_nested(1);
155 // #pragma omp parallel for num_threads(NUM_SOCKETS) proc_bind(spread)
156 // for (std::size_t sock_id=0; sock_id<NUM_SOCKETS; sock_id++)
157 // {
158 //     #pragma omp parallel for reduction(sum: result) schedule(static, STATIC_SCHEDULE_CHUNK_SIZE)
num_threads(inner_num_threads) proc_bind(close)
159 //     for (std::size_t block_idx = sock_id * num_of_blocks_div_NumSockets; block_idx < (sock_id + 1)
* num_of_blocks_div_NumSockets + (sock_id + 1 == NUM_SOCKETS) * (num_of_blocks % NUM_SOCKETS);
block_idx++)
160 //     {
161 //         // Compute Kr = KhatriRao(A_(last_mode)(l,:), A_(last_mode-1)(k,:), ..., A_(2)(j,:))
162 //         // Initialize vector Kr as Kr = A_(last_mode)(l,:);
163 //         Matrix Kr(rank,1);
164 //         Kr = factors[last_mode].col((block_idx / rows_offset[_TnsSize - 3]) % tnsDims[last_mode]);
165 //         Matrix PartialKR(rank, tnsDims[first_mode]);
166
167 //         // compute "partial" KhatriRao as a recursive Hadamard Product : Kr = Kr . * A_(j)(...,;)
168 //         for (std::size_t i = _TnsSize - 4, j = last_mode - 1; i >= 0; i--, j--)
169 //         {
170 //             if (j == mode)
171 //             {
172 //                 j--;
173 //             }
174 //             Kr = (factors[j].col((block_idx / rows_offset[j]) % tnsDims[j])).cwiseProduct(Kr);
175 //         }
176
177 //         // Compute block of KhatriRao, using "partial" vector Kr and first Factor A_(first_mode),
as : KhatriRao(Kr, A_(first_mode)(:,:))
178 //         for (int col = 0; col < tnsDims[first_mode]; col++)
179 //         {
180 //             PartialKR.col(col) = ((factors[first_mode].col(col)).cwiseProduct(Kr));
181 //         }
182
183 //         result.noalias() += tns_mat.block(0, block_idx * tnsDims[first_mode], tnsDims[mode],
tnsDims[first_mode]) * PartialKR.transpose();
184 //     }
185 // }
186 // #ifndef EIGEN_DONT_PARALLELIZE
187 //     Eigen::setNbThreads(num_threads);
188 // #endif
189
190 omp_set_nested(1);

```

```

191     #pragma omp parallel for num_threads(NUM_SOCKETS) proc_bind(spread)
192     for (std::size_t sock_id=0; sock_id<NUM_SOCKETS; sock_id++)
193     {
194         #pragma omp parallel for reduction(sum: result) schedule(static, STATIC_SCHEDULE_CHUNK_SIZE)
num_threads(inner_num_threads) proc_bind(close)
195         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++)
196             // for (int block_idx = 0; block_idx < num_of_blocks; block_idx++)
197             {
198                 // Compute Kr = KhatriRao(A_(last_mode)(l,:), A_(last_mode-1)(k,:), ..., A_(2)(j,:))
199                 // Initialize vector Kr as Kr = A_(last_mode)(l,:)
200                 Matrix Kr(1, rank);
201                 Kr = factors[last_mode].row((block_idx / rows_offset[_TnsSize - 3]) % tnsDims[last_mode]);
202                 Matrix PartialKR(tnsDims[first_mode], rank);
203                 // compute "partial" KhatriRao as a recursive Hadamard Product : Kr = Kr . * A_(j)(...,:)
204                 for (int i = static_cast<int>(_TnsSize - 4), j = last_mode - 1; i >= 0; i--, j--)
205                 {
206                     if (j == static_cast<int>(mode))
207                     {
208                         j--;
209                     }
210                     Kr = (factors[j].row((block_idx / rows_offset[j]) % tnsDims[j])).cwiseProduct(Kr);
211                 }
212
213                 // Compute block of KhatriRao, using "partial" vector Kr and first Factor A_(first_mode), as :
KhatriRao(Kr, A_(first_mode)(:,:))
214                 for (int row = 0; row < tnsDims[first_mode]; row++)
215                 {
216                     PartialKR.row(row) = ((factors[first_mode].row(row)).cwiseProduct(Kr));
217                 }
218
219                 result.noalias() += tns_mat.block(0, block_idx * tnsDims[first_mode], tnsDims[mode],
tnsDims[first_mode]) * PartialKR;
220             }
221         }
222     #ifndef EIGEN_DONT_PARALLELIZE
223         Eigen::setNbThreads(num_threads);
224     #endif
225 }
226
227 // trasposed_v
228 // Serial (using std::array for tns_dimensions)
229 template<std::size_t _TnsSize>
230 void SparseMTTKRP(const std::array<int, _TnsSize> &tns_dimensions,
231                  const SparseMatrix &sparse_tns,
232                  const std::array<Matrix, _TnsSize> &factors,
233                  const int rank,
234                  const std::array<int, _TnsSize-1> &offsets,
235                  const int last_mode,
236                  const int cur_mode,
237                  Matrix &MTTKRP)
238 {
239     Matrix temp_R_1(rank, 1);
240     Matrix MTTKRP_col(rank, 1);
241
242     for (long int i = 0; i < sparse_tns.outerSize(); ++i)
243     {
244         MTTKRP_col = Matrix::Zero(rank, 1);
245         for (typename SparseMatrix::InnerIterator it(sparse_tns, i); it; ++it)
246         {
247             temp_R_1 = Matrix::Ones(rank, 1);
248             int row;
249             // Select rows of each factor and compute the respective row of the Khatri-Rao product.
250             for (int mode_i = last_mode, kr_counter = static_cast<int>(_TnsSize) - 2; mode_i >= 0 &&
kr_counter >= 0; mode_i--)
251             {
252                 if (mode_i == cur_mode)
253                 {
254                     continue;
255                 }
256                 row = ((it.row()) / offsets[kr_counter]) % (tns_dimensions[mode_i]);
257                 temp_R_1 = temp_R_1.cwiseProduct(factors[mode_i].col(row));
258                 kr_counter--;
259             }
260             // Subtract from the previous row the respective row of W, according to relation (9).
261             // MTTKRP.row(it.col()) -= it.value() * temp_R_1;
262             MTTKRP_col -= it.value() * temp_R_1;
263         }
264         MTTKRP.col(i) = MTTKRP_col;
265     }
266 }
267
268 // Parallel OpenMP (using std::array for tns_dimensions)
269 template<std::size_t _TnsSize>
270 void SparseMTTKRP_omp(const std::array<int, _TnsSize> &tns_dimensions,

```

```

272             const SparseMatrix          &sparse_tns,
273             const std::array<Matrix, _TnsSize> &factors,
274             const int                          rank,
275             const std::array<int, _TnsSize-1> &offsets,
276             const int                          last_mode,
277             const int                          cur_mode,
278             Matrix                             &MTTKRP)
279 {
280     Matrix temp_R_1(rank, 1);
281     Matrix MTTKRP_col(rank, 1);
282
283     #pragma omp for schedule(dynamic) nowait
284     for (long int i = 0; i < sparse_tns.outerSize(); ++i)
285     {
286         MTTKRP_col = Matrix::Zero(rank, 1);
287         for (typename SparseMatrix::InnerIterator it(sparse_tns, i); it; ++it)
288         {
289             temp_R_1 = Matrix::Ones(rank, 1);
290             long int row;
291             // Select rows of each factor an compute the respective row of the Khatri-Rao product.
292             for (int mode_i = last_mode, kr_counter = static_cast<int>(_TnsSize) - 2; mode_i >= 0 &&
kr_counter >= 0; mode_i--)
293             {
294
295                 if (mode_i == cur_mode)
296                 {
297                     continue;
298                 }
299                 row = ((it.row()) / offsets[kr_counter]) % (tns_dimensions[mode_i]);
300                 temp_R_1 = temp_R_1.cwiseProduct(factors[mode_i].col(row));
301                 kr_counter--;
302             }
303             // Subtract from the previous row the respective row of W, according to relation (9).
304             // MTTKRP.row(it.col()) -= it.value() * temp_R_1;
305             MTTKRP_col -= it.value() * temp_R_1;
306         }
307         MTTKRP.col(i) = MTTKRP_col;
308     }
309 }
310
311 // Parallel (using std::vector for tns_dimensions)
312 template<std::size_t _TnsSize>
313 void SparseMTTKRP(const std::vector<std::vector<int>> &tns_dimensions,
314                  const std::array<int, _TnsSize> &fiber_rank,
315                  const SparseMatrix          &sparse_tns,
316                  const std::array<Matrix, _TnsSize> &factors,
317                  const int                          rank,
318                  const std::array<int, _TnsSize-1> &offsets,
319                  const int                          last_mode,
320                  const int                          cur_mode,
321                  Matrix                             &MTTKRP)
322 {
323     Matrix temp_R_1(rank, 1);
324     Matrix MTTKRP_col(rank, 1);
325
326     for (long long int i = 0; i < sparse_tns.outerSize(); ++i)
327     {
328         MTTKRP_col.setZero();
329         for (typename SparseMatrix::InnerIterator it(sparse_tns, i); it; ++it)
330         {
331             temp_R_1 = Matrix::Ones(rank, 1);
332             long long int row;
333             // Select rows of each factor an compute the respective row of the Khatri-Rao product.
334             for (int mode_i = last_mode, kr_counter = static_cast<int>(_TnsSize) - 2; mode_i >= 0 &&
kr_counter >= 0; mode_i--)
335             {
336                 if (mode_i == cur_mode)
337                 {
338                     continue;
339                 }
340                 row = ((it.row()) / offsets[kr_counter]) %
(tns_dimensions[mode_i][fiber_rank[mode_i]]);
341
342                 temp_R_1 = temp_R_1.cwiseProduct(factors[mode_i].col(row));
343                 kr_counter--;
344             }
345             // Subtract from the previous row the respective row of W, according to relation (9).
346             MTTKRP_col -= it.value() * temp_R_1;
347         }
348         MTTKRP.col(i) = MTTKRP_col;
349     }
350 }
351
352 } // end namespace v1
353
354 namespace v2 // std_v
355 {

```

```

356 // Serial (using std::array for tns_dimensions)
357 template<std::size_t _TnsSize>
358 void SparseMTTKRP(const std::array<int, _TnsSize> &tns_dimensions,
359                 const SparseMatrix &sparse_tns,
360                 const std::array<Matrix, _TnsSize> &factors,
361                 const int rank,
362                 const std::array<int, _TnsSize> &offsets,
363                 const int last_mode,
364                 const int cur_mode,
365                 Matrix &MTTKRP)
366 {
367     Matrix temp_1_R(1, rank);
368     Matrix MTTKRP_row(1, rank);
369
370     MTTKRP.setZero();
371
372     for (long int i = 0; i < sparse_tns.outerSize(); ++i)
373     {
374         MTTKRP_row = Matrix::Zero(1, rank);
375         for (typename SparseMatrix::InnerIterator it(sparse_tns, i); it; ++it)
376         {
377             temp_1_R = Matrix::Ones(1, rank);
378             int row;
379             // Select rows of each factor an compute the respective row of the Khatri-Rao product.
380             for (int mode_i = last_mode, kr_counter = static_cast<int>(_TnsSize) - 2; mode_i >= 0 &&
381                 kr_counter >= 0; mode_i--)
382             {
383                 if (mode_i == cur_mode)
384                 {
385                     continue;
386                 }
387                 row = ((it.row()) / offsets[kr_counter]) % (tns_dimensions[mode_i]);
388                 temp_1_R = temp_1_R.cwiseProduct(factors[mode_i].row(row));
389                 kr_counter--;
390             }
391             // Subtract from the previous row the respective row of W, according to relation (9).
392             MTTKRP_row -= it.value() * temp_1_R;
393         }
394         MTTKRP.row(i) = MTTKRP_row;
395     }
396 }
397
398 // Parallel (using std::vector for tns_dimensions)
399 template<std::size_t _TnsSize>
400 void SparseMTTKRP(const std::vector<int> &tns_dimensions,
401                 const std::array<int, _TnsSize> &fiber_rank,
402                 const SparseMatrix &sparse_tns,
403                 const std::array<Matrix, _TnsSize> &factors,
404                 const int rank,
405                 const std::array<int, _TnsSize> &offsets,
406                 const int last_mode,
407                 const int cur_mode,
408                 Matrix &MTTKRP)
409 {
410     Matrix temp_1_R(1, rank);
411     Matrix MTTKRP_row(1, rank);
412
413     MTTKRP.setZero();
414
415     for (long int i = 0; i < sparse_tns.outerSize(); ++i)
416     {
417         MTTKRP_row = Matrix::Zero(1, rank);
418         for (typename SparseMatrix::InnerIterator it(sparse_tns, i); it; ++it)
419         {
420             temp_1_R = Matrix::Ones(1, rank);
421             int row;
422             // Select rows of each factor an compute the respective row of the Khatri-Rao product.
423             for (int mode_i = last_mode, kr_counter = static_cast<int>(_TnsSize) - 2; mode_i >= 0 &&
424                 kr_counter >= 0; mode_i--)
425             {
426                 if (mode_i == cur_mode)
427                 {
428                     continue;
429                 }
430                 row = ((it.row()) / offsets[kr_counter]) % (tns_dimensions[fiber_rank[mode_i]]);
431                 temp_1_R = temp_1_R.cwiseProduct(factors[mode_i].row(row));
432                 kr_counter--;
433             }
434             // Subtract from the previous row the respective row of W, according to relation (9).
435             MTTKRP_row -= it.value() * temp_1_R;
436         }
437         MTTKRP.row(i) = MTTKRP_row;
438     }
439 }
440

```

```

441 } // end namespace v2
442
443 } // end namespace partensor
444
445 #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)
SVDdecomposition 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 )

```

SVDdecomposition 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 $m \times n$, A belongs in R with dimensions $m \times r$, B belongs in R with dimensions $n \times r$ and consider the minimization problem $0.5 \text{ Frobenius_norm } (X-AB)^2$. We can use the following function in order to solve this problem.

Parameters

mat1	[in] The covariance Matrix $B^T B$.
mat2	[in] A Matrix containing ther resultt of $-(X^T B)$.
delta_1	[in] If Y is the updated matrix at each iteration, then delta_1 is the maximum tolerance for the value $\text{abs}(\text{gradient}(Y))$.
delta_2	[in] If Y is the updated matrix at each iteration, then delta_2 is the minimum tolerance for the value $\text{gradient}(Y)$.
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.](#)

```
1 #ifndef DOXYGEN_SHOULD_SKIP_THIS
15 #endif // DOXYGEN_SHOULD_SKIP_THIS
16 /*****
25 #ifndef PARTENSOR_NESTEROV_MNLS_HPP
26 #define PARTENSOR_NESTEROV_MNLS_HPP
27
28 #include "PARTENSOR_basic.hpp"
29 #include "MTTKRP.hpp"
30
31 namespace partensor
32 {
33
34     inline namespace v1 {
35         void ComputeSVD( Matrix const &mat,
36                         double &L,
37                         double &mu )
38         {
39             Eigen::JacobiSVD<Matrix> svd(mat, Eigen::ComputeThinU | Eigen::ComputeThinV);
40             L = svd.singularValues().maxCoeff();
41             mu = svd.singularValues().minCoeff();
42         }
43
44         void ComputeEIG(Matrix const &mat,
45                        double &L,
46                        double &mu)
47         {
48             Eigen::EigenSolver<Matrix> eig(mat);
49             L = (eig.eigenvalues().real()).maxCoeff();
50             mu = (eig.eigenvalues().real()).minCoeff();
51         }
52
53         inline void GLambda( double mu,
54                             double &L,
55                             double &lambda,
56                             double &q )
57         {
58             q = mu/L;
59             if (1/q>1e6)
60                 lambda = 10 * mu;
61             else if (1/q>1e3)
62                 lambda = mu;
63             else
64                 lambda = mu/10;
65
66             L += lambda;
67             mu += lambda;
68             q = mu/L;
69         }
70     }
71 }
72
```



```

103     inline double UpdateAlpha( double const alpha,
104                                double const q
105                                )
106     {
107         double a, b, c, D;
108         a = 1;
109         b = alpha * alpha - q;
110         c = -alpha * alpha;
111         D = b*b - 4 * a*c;
112
113         return (-b+sqrt(D))/2;
114     }
115
116     void TuneLambda(const double &L,
117                   double &lambda,
118                   const double &ratio)
119     {
120         // if(ratio < 1)
121         // {
122         //     lambda = 0.01;
123         // }
124         // else
125         // {
126         //     lambda = (L * ratio) / (1 - ratio);
127         // }
128         lambda = L/ratio;
129     }
130
131     // Returns maximum eigenvalue L of square matrix mat
132     inline double PowerMethod( Matrix &mat, const double epsilon)
133     {
134         Matrix x_init = Matrix::Random(mat.cols(),1);
135         Matrix x_new = x_init;
136         Matrix Ax = x_init;
137         double norm_Ax;
138         double lambda_max = 0;;
139
140         int iter = 0;
141         int MAX_ITER = 1e+4;
142
143         while (1)
144         {
145             Ax.noalias() = mat * x_init;
146             norm_Ax = Ax.norm();
147             x_new.noalias() = 1/(norm_Ax + 1e-12) * Ax;
148
149             if ((x_new - x_init).norm() <= epsilon || iter >= MAX_ITER)
150             {
151                 lambda_max = (x_new.transpose() * (mat * x_new))(0);
152                 break;
153             }
154
155             x_init = x_new;
156             iter++;
157         }
158
159         return lambda_max;
160     }
161
162     void NesterovMMLS( Matrix const &mat1,
163                      Matrix const &mat2,
164                      double const delta_1,
165                      double const delta_2,
166                      Matrix &res
167                      )
168     {
169         const int max_inner = 50;
170
171         int m = res.rows();
172         int r = res.cols();
173         int iter = 0;
174         double L, mu, lambda, q, alpha, new_alpha, beta;
175
176         Matrix A(m, r);
177         Matrix Y(m, r);
178         Matrix new_A(m, r);
179         Matrix grad_Y(m, r);
180         Matrix _mat1 = mat1;
181         Matrix _mat2 = mat2;
182         Matrix Zero_Matrix = Matrix::Zero(m, r);
183
184         ComputeSVD(mat1, L, mu);
185         GLambda(mu, L, lambda, q);
186
187         _mat1 += lambda * Matrix::Identity(r, r);
188         _mat2 += lambda * res;
189         // q = mu/L;
190         alpha = 1;

```

```

209         A           = res;
210         Y           = res;
211
212         while(1)
213         {
214             grad_Y   = -_mat2;
215             grad_Y.noalias() += Y * _mat1; // | grad_Y = W + Y *
Z.transpose();
216
217         if ((grad_Y.cwiseProduct(Y).cwiseAbs().maxCoeff() <= delta_1 &&
grad_Y.minCoeff() >= -delta_2) || (iter >= max_inner))
218             break;
219
220         new_A       = (Y - grad_Y/L).cwiseMax(Zero_Matrix);
221
222         // if ( ( (new_A-Y).norm()/Y.norm())<=delta_1 ) || (iter >= max_inner))
223         //     break;
224
225         new_alpha = UpdateAlpha(alpha, q);
226         beta     = alpha * (1 - alpha) / (alpha * alpha + new_alpha);
227
228         Y       = (1 + beta) * new_A - beta * A;
229         A       = new_A;
230         alpha   = new_alpha;
231         iter++;
232     }
233     res = A;
234 }
235
236 // GTC Serial
237 // transposed_v
238 template<std::size_t _TnsSize>
239 void NesterovMNLs(Matrix &mat1,
240                  std::array<Matrix, _TnsSize> &factors,
241                  std::array<int, _TnsSize> const &tns_dimensions,
242                  SparseMatrix const &tns_spMat,
243                  std::array<int, _TnsSize-1> const &offsets,
244                  int const max_nest_iter,
245                  double const ratio,
246                  int const cur_mode,
247                  Constraint const constraint_i,
248                  Matrix &MTTKRP_T)
249 {
250     int m = factors[cur_mode].cols();
251     int r = factors[cur_mode].rows();
252     double L, mu, q, alpha, new_alpha, beta, lambda;
253     int iter = 0;
254
255     Matrix grad_Y(r, m);
256     Matrix Y(r, m);
257     Matrix new_A(r, m);
258     Matrix A(r, m);
259     Matrix Zero_Matrix = Matrix::Zero(r, m);
260
261     ComputeEIG(mat1, L, mu);
262
263     lambda = ratio;
264     L = L + lambda;
265     q = lambda / L;
266     alpha = 1;
267
268     A = factors[cur_mode];
269     Y = factors[cur_mode];
270
271     int last_mode = (cur_mode == static_cast<int>(_TnsSize) - 1) ?
static_cast<int>(_TnsSize) - 2 : static_cast<int>(_TnsSize) - 1;
272
273     Matrix temp_R_1(r, 1);
274     Matrix temp_col = Matrix::Zero(r, 1);
275     while (1)
276     {
277         grad_Y.setZero();
278
279         if (iter >= max_nest_iter)
280         {
281             break;
282         }
283
284         // Compute grad_Y
285         for (long int i = 0; i < tns_spMat.outerSize(); ++i)
286         {
287             temp_col.setZero();
288             for (SparseMatrix::InnerIterator it(tns_spMat, i); it; ++it)
289             {
290                 temp_R_1 = Matrix::Ones(r, 1);
291                 // Select rows of each factor an compute the respective
row of the Khatri-Rao product.

```

```

292         for (int mode_i = last_mode, kr_counter =
static_cast<int>(_TnsSize) - 2; mode_i >= 0 && kr_counter >= 0; mode_i--)
293         {
294             if (mode_i == cur_mode)
295             {
296                 continue;
297             }
298             long int row;
299             row = ((it.row()) / offsets[kr_counter]) %
(tns_dimensions[mode_i]);
300             temp_R_1 =
temp_R_1.cwiseProduct(factors[mode_i].col(row));
301             kr_counter--;
302         }
303         // Computation of row of Z according the relation (10)
of the paper.
304         temp_col += (temp_R_1.transpose() * Y.col(i))(0) *
temp_R_1;
305     }
306     grad_Y.col(i) = temp_col;
307 }
308
309 // Add proximal term.
310 grad_Y += MTTKRP_T + lambda * Y;
311 if (constraint_i == Constraint::unconstrained)
312 {
313     new_A = (Y - grad_Y / L);
314 }
315 else // Use projection
316 {
317     new_A = (Y - grad_Y / L).cwiseMax(Zero_Matrix);
318 }
319
320 new_alpha = UpdateAlpha(alpha, q);
321 beta = alpha * (1 - alpha) / (alpha * alpha + new_alpha);
322
323 Y = (1 + beta) * new_A - beta * A;
324
325 // Update Y
326
327 A = new_A;
328 alpha = new_alpha;
329 iter++;
330 }
331 factors[cur_mode] = A;
332
333 }
334
335 // OpenMP
336 template<std::size_t _TnsSize>
337 void NesterovMNL(Matrix
338                 &mat1,
339                 std::array<Matrix, _TnsSize> &factors,
340                 std::array<int, _TnsSize> const &tns_dimensions,
341                 SparseMatrix const &tns_spMat,
342                 const std::array<int, _TnsSize-1> &offsets,
343                 Matrix &Y,
344                 int const max_nest_iter,
345                 double const ratio,
346                 int const cur_mode,
347                 Constraint const constraint_i,
348                 Matrix &MTTKRP_T)
349 {
350     int r = factors[cur_mode].rows();
351     double L, mu, q, alpha, new_alpha, beta, lambda;
352     int iter = 0;
353     long int row;
354
355     Matrix new_A_vec(r, 1);
356     Matrix Zero_Vec = Matrix::Zero(r, 1);
357
358     ComputeEIG(mat1, L, mu);
359
360     lambda = ratio;
361     L = L + lambda;
362     q = lambda / L;
363     alpha = 1;
364
365     #pragma omp master
366     Y = factors[cur_mode];
367     #pragma omp barrier
368
369     int last_mode = (cur_mode == static_cast<int>(_TnsSize) - 1) ?
static_cast<int>(_TnsSize) - 2 : static_cast<int>(_TnsSize) - 1;
370     Matrix temp_R_1(r, 1);
371
372     #pragma omp barrier

```

```

373         while (1)
374         {
375             if (iter >= max_nest_iter)
376             {
377                 break;
378             }
379
380             Matrix temp_col = Matrix::Zero(r, 1);
381             new_alpha      = UpdateAlpha(alpha, q);
382             beta           = alpha * (1 - alpha) / (alpha * alpha + new_alpha);
383
384             // Compute grad_Y
385             #pragma omp for schedule(dynamic) //ordered
386             for (long int i = 0; i < tns_spMat.outerSize(); ++i)
387             {
388                 temp_col.setZero();
389                 for (SparseMatrix::InnerIterator it(tns_spMat, i); it; ++it)
390                 {
391                     temp_R_1 = Matrix::Ones(r, 1);
392                     // Select rows of each factor and compute the respective
row of the Khatri-Rao product.
393                     for (int mode_j = last_mode, kr_counter =
static_cast<int>(_TnsSize) - 2; mode_j >= 0 && kr_counter >= 0; mode_j--)
394                     {
395                         if (mode_j == cur_mode)
396                         {
397                             continue;
398                         }
399                         row = ((it.row()) / offsets[kr_counter]) %
(tns_dimensions[mode_j]);
400                         temp_R_1 =
temp_R_1.cwiseProduct(factors[mode_j].col(row));
401                         kr_counter--;
402                     }
403                     // Computation of row of Z according the relation (10)
of the paper.
404                     temp_col += (temp_R_1.transpose() * Y.col(i))(0) *
temp_R_1;
405                 }
406                 temp_col.noalias() += MTKRP_T.col(i) + (lambda * Y.col(i));
407
408                 // Add proximal term.
409                 if (constraint_i == Constraint::unconstrained)
410                 {
411                     new_A_vec = (Y.col(i) - temp_col / L);
412                 }
413                 else // Use projection
414                 {
415                     new_A_vec = (Y.col(i) - temp_col /
L).cwiseMax(Zero_Vec);
416                 }
417                 Y.col(i) = (1 + beta) * new_A_vec - beta *
factors[cur_mode].col(i);
418                 factors[cur_mode].col(i) = new_A_vec;
419             }
420             alpha = new_alpha;
421             iter++;
422         }
423     }
424
425     #pragma omp barrier
426 }
427
428 namespace dynamic_blocksize
429 {
430     template <std::size_t _TnsSize>
431     void StochasticNesterovMNLS(std::array<Matrix, _TnsSize> &factors,
std::array<int,
_TnsSize> const &tns_dimensions,
std::array<int,
_TnsSize> 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)
432     {
433         int r = factors[cur_mode].rows();
434         double L2, inv_L2;
435         double sqrt_q = 0, beta = 0;
436         int iter = 0;
437     }
438 }

```

```

445         long int row;
446
447         const Matrix zero_vec = Matrix::Zero(r, 1);
448         Matrix new_A_vec = Matrix::Zero(r, 1);
449
450         Matrix Y = factors[cur_mode];
451
452         // int first_mode = (cur_mode == 0) ? 1 : 0;
453         int last_mode = (cur_mode == static_cast<int>(_TnsSize) - 1) ?
static_cast<int>(_TnsSize) - 2 : static_cast<int>(_TnsSize) - 1;
454
455         Matrix temp_R_1 = Matrix::Ones(r, 1);
456
457         Matrix temp_RxR(r, r);
458         Matrix temp_col = Matrix::Zero(r, 1);
459
460         std::srand(std::time(nullptr));
461
462         while (1)
463         {
464
465             if (iter >= max_nest_iter)
466             {
467                 break;
468             }
469
470             // Compute grad_Y
471             for (long int i = 0; i < tns_spMat.outerSize(); ++i)
472             {
473                 temp_col.setZero();
474                 // -- Pseudocode --
475                 // SparseMatrix::InnerIterator it(tns_spMat, i);
476                 // int length = tns_spMat.innerNonZeros(); // ??? check
if innerNonZeros, we want to count the nonzeros entries in the row
477                 // int random_pivot = rand(length - blocksize)
478                 // it += random_pivot;
479
480                 // Get the number of nnz per row of matricization.
481                 long int nnzs_per_col =
tns_spMat.innerVector(i).nonZeros();
482
483                 long int var_blocksize_i = static_cast<long
int>(c_stochastic_perc * nnzs_per_col);
484
485                 if (var_blocksize_i > 0)
486                 {
487                     // Choose a pivot from [0, nnzs_per_col -
blocksize].
488                     long int pivot = (std::rand() % (nnzs_per_col -
var_blocksize_i + 1));
489
490                     SparseMatrix::InnerIterator it(tns_spMat, i);
491                     // Iterate over [pivot, pivot + var_blocksize_i]
nnz elements per row.
492                     // it += pivot;
493                     for (long int acuum = 0; acuum < pivot; acuum++)
494                         ++it;
495
496                     // std::cout << pivot << "      nt" << nnzs_per_col <<
std::endl;
497                     temp_RxR.setZero();
498
499                     for (long int sample = 0; sample <
var_blocksize_i; sample++, ++it)
500                     {
501                         temp_R_1 = Matrix::Ones(r, 1);
502                         // Select rows of each factor an compute
the respective row of the Khatri-Rao product.
503                         for (int mode_i = last_mode, kr_counter
= static_cast<int>(_TnsSize) - 2; mode_i >= 0 && kr_counter >= 0; mode_i--)
504                         {
505                             if (mode_i == cur_mode)
506                             {
507                                 continue;
508                             }
509                             row = ((it.row()) /
offsets[kr_counter]) % (tns_dimensions[mode_i]);
510                             // temp_R_1 =
temp_R_1.cwiseProduct(factors[mode_i].row(row));
511                             temp_R_1 =
temp_R_1.cwiseProduct(factors[mode_i].col(row));
512                             kr_counter--;
513                         }
514                         // Computation of row of Z according the
relation (10) of the paper.
515                         // Subtract each term of MTKRP's row.
516                         temp_col.noalias() +=

```

```

517 ((temp_R_1.transpose() * Y.col(i))(0) - it.value()) * temp_R_1;
518 // Estimate Hessian for each row.
519 temp_RxR.noalias() += (temp_R_1 *
temp_R_1.transpose());
520 }
521 // Solve with l2-regularization term
522 // temp_col.noalias() += (lambda * Y.col(i));
523 // Solve with Proximal term
524 temp_col.noalias() += lambda * (Y.col(i) +
factors[cur_mode].col(i));
525
526 L2 = PowerMethod(temp_RxR, 1e-3);
527 L2 += lambda;
528 inv_L2 = 1 / L2;
529 new_A_vec.noalias() = (Y.col(i) - inv_L2 *
temp_col);
530 new_A_vec =
531 (new_A_vec).cwiseMax(zero_vec);
532
533 sqrt_q = sqrt(lambda * inv_L2);
534 beta = (1 - sqrt_q) / (1 + sqrt_q);
535 // Update Y
536 Y.col(i) = (1 + beta) * new_A_vec - beta *
factors[cur_mode].col(i);
537 // Update i-th column of current factor
538 factors[cur_mode].col(i) = new_A_vec;
539 }
540 // alpha = new_alpha;
541 iter++;
542 }
543 }
544 }
545 }
546 }
547 }
548 }
549 }
550 namespace local_L
551 {
552 // OpenMP
553 template<std::size_t _TnsSize>
554 void StochasticNesterovMNLS(std::array<Matrix, _TnsSize> &factors,
555 _TnsSize> const &tns_dimensions,
556 SparseMatrix
557 const
558 std::array<int, _TnsSize-1> &offsets,
559 double
560 Matrix
561 &Y,
562 int const
563 max_nest_iter,
564 double const
565 lambda,
566 int const
567 cur_mode) {
568 int r = factors[cur_mode].rows();
569 double sqrt_q = 0, beta = 0, L2 = 0, inv_L2 = 0;
570 int iter = 0;
571 long int row;
572 Matrix new_A_vec = Matrix::Zero(r, 1);
573 Matrix zero_vec = Matrix::Zero(r, 1);
574 #pragma omp master
575 Y = factors[cur_mode];
576 int last_mode = (cur_mode == static_cast<int>(_TnsSize) - 1) ?
577 static_cast<int>(_TnsSize) - 2 : static_cast<int>(_TnsSize) - 1;
578 Matrix temp_R_1 = Matrix::Ones(r, 1);
579 Matrix temp_RxR(r, r);
580 Matrix temp_col(r, 1);
581 std::srand(std::time(nullptr));
582 #pragma omp barrier
583 while (1)
584 {
585 if (iter >= max_nest_iter)
586 {
587

```

```

588                                     break;
589                                     }
590                                     // Compute grad_Y
591
592                                     // #pragma omp for schedule(dynamic, 8) ordered
593                                     // #pragma omp for schedule(guided) nowait
594                                     #pragma omp for schedule(dynamic) nowait
595                                     for (long int i = 0; i < tns_spMat.outerSize(); ++i)
596                                     {
597                                     // Get the number of nnz per row of
598                                     long int nnzs_per_col =
599                                     tns_spMat.innerVector(i).nonZeros();
600                                     long int var_blocksize_i = static_cast<long
601                                     int>(c_stochastic_perc * nnzs_per_col);
602                                     if (var_blocksize_i > 0)
603                                     {
604                                     temp_col.setZero();
605                                     // Choose a pivot from [0, nnzs_per_col
606                                     - blocksize].
607                                     long int pivot = (std::rand() %
608                                     (nnzs_per_col - var_blocksize_i + 1));
609                                     SparseMatrix::InnerIterator
610                                     // Iterate over [pivot, pivot +
611                                     var_blocksize_i] nnz elements per row.
612                                     // it += pivot;
613                                     for (long int acuum = 0; acuum < pivot;
614                                     acuum++)
615                                     ++it;
616                                     temp_RxR.setZero();
617                                     for (long int sample = 0; sample <
618                                     var_blocksize_i; sample++, ++it)
619                                     {
620                                     temp_R_1 = Matrix::Ones(r, 1);
621                                     // Select rows of each factor an
622                                     compute the respective row of the Khatri-Rao product.
623                                     for (int mode_i = last_mode,
624                                     kr_counter = static_cast<int>(_TnsSize) - 2; mode_i >= 0 && kr_counter >= 0; mode_i--,
625                                     kr_counter--)
626                                     {
627                                     if (mode_i == cur_mode)
628                                     {
629                                     continue;
630                                     }
631                                     row = ((it.row()) /
632                                     temp_R_1 =
633                                     kr_counter--);
634                                     // Computation of row of Z
635                                     temp_col.noalias() +=
636                                     // Estimate Hessian for each
637                                     temp_RxR.noalias() += (temp_R_1
638                                     * temp_R_1.transpose());
639                                     }
640                                     temp_col.noalias() += lambda * (Y.col(i)
641                                     + factors[cur_mode].col(i));
642                                     L2 = PowerMethod(temp_RxR, 1e-3);
643                                     L2 += lambda;
644                                     inv_L2 = 1 / L2;
645                                     new_A_vec.noalias() = (Y.col(i) - inv_L2
646                                     * temp_col);
647                                     new_A_vec =
648                                     (new_A_vec).cwiseMax(zero_vec);
649                                     sqrt_q = sqrt( lambda * inv_L2 );
650                                     beta = (1 - sqrt_q) / (1 + sqrt_q);
651                                     // Update Y
652                                     Y.col(i) = (1 + beta) * new_A_vec - beta
653

```

```

654 * factors[cur_mode].col(i);
655
656 // Update i-th column of current factor
657 factors[cur_mode].col(i) = new_A_vec;
658
659 }
660 }
661 }
662 }
663 }
664 }
665 } // end namespace local_L
666 } // end namespace dynamic_blocksize
667
668 } // end namespace v1
669
670 namespace std_V
671 {
672 // GTC Serial
673 template<std::size_t _TnsSize>
674 void NesterovMNL(Matrix &mat1,
675                  std::array<Matrix, _TnsSize> &factors,
676                  std::array<int, _TnsSize> const &tns_dimensions,
677                  SparseMatrix const &tns_spMat,
678                  const std::array<int, _TnsSize-1> &offsets,
679                  int const max_nest_iter,
680                  double const ratio,
681                  int const cur_mode,
682                  Constraint const constraint_i,
683                  Matrix const &MTTKRP)
684 {
685     int m = factors[cur_mode].rows();
686     int r = factors[cur_mode].cols();
687     double L, mu, q, alpha, new_alpha, beta, lambda;
688     int iter = 0;
689
690     Matrix grad_Y(m, r);
691     Matrix Y(m, r);
692     Matrix new_A(m, r);
693     Matrix A(m, r);
694     Matrix Zero_Matrix = Matrix::Zero(m, r);
695
696     ComputeEIG(mat1, L, mu);
697
698     lambda = ratio; // TuneLambda(L, lambda, ratio);
699     L = L + lambda;
700     q = lambda / L;
701     alpha = 1;
702
703     A = factors[cur_mode];
704     Y = factors[cur_mode]; // layer_factor
705
706     int last_mode = (cur_mode == static_cast<int>(_TnsSize) - 1) ?
707     static_cast<int>(_TnsSize) - 2 : static_cast<int>(_TnsSize) - 1;
708
709     Matrix temp_1_R(1, r);
710     while (1)
711     {
712         grad_Y.setZero();
713
714         if (iter >= max_nest_iter)
715         {
716             break;
717         }
718
719         // Compute grad_Y
720         Matrix temp_row = Matrix::Zero(1, r);
721         for (long int i = 0; i < tns_spMat.outerSize(); ++i)
722         {
723             temp_row.setZero();
724             for (SparseMatrix::InnerIterator it(tns_spMat, i); it; ++it)
725             {
726                 temp_1_R = Matrix::Ones(1, r);
727                 // Select rows of each factor and compute the respective
728                 for (int mode_i = last_mode, kr_counter =
729                 static_cast<int>(_TnsSize) - 2; mode_i >= 0 && kr_counter >= 0; mode_i--)
730                 {
731                     if (mode_i == cur_mode)
732                     {
733                         continue;
734                     }
735                     int row;
736                     row = ((it.row()) / offsets[kr_counter]) %
737                     (tns_dimensions[mode_i]);

```



```

736                                     temp_1_R =
737 temp_1_R.cwiseProduct(factors[mode_i].row(row));                               kr_counter--;
738                                                                                   }
739 // Computation of row of Z according the relation (10)
of the paper.
740                                     temp_row += (Y.row(i) * temp_1_R.transpose()) *
temp_1_R;
741                                     }
742                                     grad_Y.row(i) = temp_row;
743                                     }
744
745 // Add proximal term.
746 grad_Y += MTTKRP + lambda * Y;
747 if (constraint_i == Constraint::unconstrained)
748 {
749     new_A = (Y - grad_Y / L);
750 }
751 else // Use projection
752 {
753     new_A = (Y - grad_Y / L).cwiseMax(Zero_Matrix);
754 }
755
756 new_alpha = UpdateAlpha(alpha, q);
757 beta      = alpha * (1 - alpha) / (alpha * alpha + new_alpha);
758
759 Y = (1 + beta) * new_A - beta * A;
760
761 // Update Y
762
763 A      = new_A;
764 alpha = new_alpha;
765 iter++;
766 }
767     factors[cur_mode] = A;
768 }
769 // end of namespace std_V
770
771 namespace local_L
772 {
773     // GTC Serial
774     // transposed_v
775     template<std::size_t _TnsSize>
776     void NesterovMnls(std::array<Matrix, _TnsSize> &factors,
777                     std::array<int, _TnsSize> SparseMatrix, const &tns_dimensions,
778                     const std::array<int, _TnsSize-1> &offsets, const &tns_spMat,
779                     int const max_nest_iter,
780                     double const lambda,
781                     int const cur_mode,
782                     Matrix &MTTKRP_T)
783     {
784
785         int m = factors[cur_mode].cols();
786         int r = factors[cur_mode].rows();
787
788         Matrix inv_L2(tns_spMat.outerSize(),1);
789
790         double sqrt_q = 0, beta = 0;
791         double L2;
792         int iter = 0;
793         Matrix Y(r, m);
794         Matrix A(r, m);
795
796         const Matrix zero_vec = Matrix::Zero(r, 1);
797         Matrix new_A_vec = Matrix::Zero(r, 1);
798
799         A = factors[cur_mode];
800         Y = factors[cur_mode];
801
802         int last_mode = (cur_mode == static_cast<int>(_TnsSize) - 1) ?
803 static_cast<int>(_TnsSize) - 2 : static_cast<int>(_TnsSize) - 1;
804
805         Matrix temp_R_1 = Matrix::Ones(r, 1);
806         Matrix temp_RxR(r, r);
807
808         while (1)
809         {
810             if (iter >= max_nest_iter)
811             {
812                 break;
813             }
814
815             // Compute grad_Y
816             Matrix temp_col = Matrix::Zero(r, 1);
817             for (long int i = 0; i < tns_spMat.outerSize(); ++i)
818             {

```

```

819         temp_col.setZero();
820
821         if (iter < 1)
822         {
823             temp_RxR.setZero();
824         }
825
826         for (SparseMatrix::InnerIterator it(tns_spMat, i); it; ++it)
827         {
828             temp_R_1 = Matrix::Ones(r, 1);
829             // Select rows of each factor and compute the respective
row of the Khatri-Rao product.
830             for (int mode_i = last_mode, kr_counter =
static_cast<int>(_TnsSize) - 2; mode_i >= 0 && kr_counter >= 0; mode_i--)
831             {
832                 if (mode_i == cur_mode)
833                 {
834                     continue;
835                 }
836                 long long int row = ((it.row()) /
offsets[kr_counter]) % (tns_dimensions[mode_i]);
837                 temp_R_1 =
temp_R_1.cwiseProduct(factors[mode_i].col(row));
838                 kr_counter--;
839             }
840             // Computation of row of Z according the relation (10)
of the paper.
841             temp_col.noalias() += (temp_R_1.transpose() *
Y.col(i))(0) * temp_R_1;
842             // Estimate Hessian for each row.
843             if (iter < 1)
844             {
845                 temp_RxR.noalias() += (temp_R_1 *
temp_R_1.transpose());
846             }
847             temp_col.noalias() += MTTKRP_T.col(i) + (lambda * Y.col(i));
848
849             if (iter < 1)
850             {
851                 L2 = PowerMethod(temp_RxR, 1e-3);
852                 L2 += lambda;
853                 inv_L2(i) = 1 / L2;
854             }
855
856             new_A_vec = (Y.col(i) - inv_L2(i) *
temp_col).cwiseMax(zero_vec);
857
858             sqrt_q = sqrt( lambda * inv_L2(i) );
859
860             beta = (1 - sqrt_q) / (1 + sqrt_q);
861
862             // Update Y
863             Y.col(i) = (1 + beta) * new_A_vec - beta *
factors[cur_mode].col(i);
864
865             // Update i-th column of current factor
866             factors[cur_mode].col(i) = new_A_vec;
867         }
868         iter++;
869     }
870 }
871 }
872 }
873 }
874 }
875 }
876 }
877 // OpenMP
878 template<std::size_t _TnsSize>
879 void NesterovMNLN(Matrix
&inv_L2,
&factors,
std::array<Matrix, _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)
880 {
881     int r = factors[cur_mode].rows();
882     double sqrt_q = 0, beta = 0;
883     double L2;
884     int iter = 0;
885     long int row;
886     Matrix new_A_vec = Matrix::Zero(r, 1);

```

```

897         Matrix zero_vec = Matrix::Zero(r, 1);
898
899         #pragma omp master
900         Y = factors[cur_mode];
901         #pragma omp barrier
902
903         int last_mode = (cur_mode == static_cast<int>(_TnsSize) - 1) ?
static_cast<int>(_TnsSize) - 2 : static_cast<int>(_TnsSize) - 1;
904
905         Matrix temp_R_1 = Matrix::Ones(r, 1);
906         Matrix temp_RxR(r, r);
907         Matrix temp_col(r, 1);
908
909         #pragma omp barrier
910         while (1)
911         {
912             if (iter >= max_nest_iter)
913             {
914                 break;
915             }
916
917             // Compute grad_Y
918             #pragma omp for schedule(dynamic) nowait
919             for (long int i = 0; i < tns_spMat.outerSize(); ++i)
920             {
921                 temp_col.setZero();
922                 if (iter < 1)
923                 {
924                     temp_RxR.setZero();
925                 }
926                 for (SparseMatrix::InnerIterator it(tns_spMat, i); it; ++it)
927                 {
928                     temp_R_1 = Matrix::Ones(r, 1);
929                     // Select rows of each factor and compute the respective
row of the Khatri-Rao product.
930                     for (int mode_i = last_mode, kr_counter =
static_cast<int>(_TnsSize) - 2; mode_i >= 0 && kr_counter >= 0; mode_i--)
931                     {
932                         if (mode_i == cur_mode)
933                         {
934                             continue;
935                         }
936                         (tns_dimensions[mode_i];
937                         temp_R_1.cwiseProduct(factors[mode_i].col(row));
938                         kr_counter--;
939                     }
940                     // Computation of row of Z according the relation (10)
of the paper.
941                     temp_col.noalias() += (temp_R_1.transpose() *
Y.col(i))(0) * temp_R_1;
942
943                     // Estimate Hessian for each row.
944                     // Compute only once!
945                     if (iter < 1)
946                     {
947                         temp_RxR.noalias() += (temp_R_1 *
temp_R_1.transpose());
948                     }
949
950                     temp_col.noalias() += MTTKRP_T.col(i) + (lambda * Y.col(i));
951
952                     if (iter < 1)
953                     {
954                         L2 = PowerMethod(temp_RxR, 1e-3);
955                         L2 += lambda;
956                         inv_L2(i) = 1 / L2;
957                     }
958
959                     new_A_vec = (Y.col(i) - inv_L2(i) *
temp_col).cwiseMax(zero_vec);
960
961                     sqrt_q = sqrt( lambda * inv_L2(i) );
962
963                     beta = (1 - sqrt_q) / (1 + sqrt_q);
964
965                     // Update Y
966                     Y.col(i) = (1 + beta) * new_A_vec - beta *
factors[cur_mode].col(i);
967
968                     // Update i-th column of current factor
969                     factors[cur_mode].col(i) = new_A_vec;
970
971                 }
972             }
973             iter++;

```

```

974         }
975         #pragma omp barrier
976     }
977 } // end of namespace local_L
978 } // end namespace partensor
979
980 } // end namespace partensor
981
982 #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 rst 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

<code>_TnsSize</code>	Size of the factors and gramian arrays.
-----------------------	---

Parameters

<code>weight_factor</code>	[in] The facotr that the weights of the other factors will be loaded.
<code>R</code>	[in] Rank of factorization. (Number of columns in each Matrix).
<code>gramian</code>	[in,out] Quantity of factor T factor .
<code>factors</code>	[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 le.](#)

```
1 #ifndef DOXYGEN_SHOULD_SKIP_THIS
15 #endif // DOXYGEN_SHOULD_SKIP_THIS
16 / ***** /
23 #ifndef NORMALIZE_HPP
24 #define NORMALIZE_HPP
25
26 #include <math.h>
27 #include "PARTENSOR_basic.hpp"
28 #include "DimTrees.hpp"
29 #include "TensorOperations.hpp"
30 #include "Eigen/Core"
31
32 namespace partensor
33 {
34
35     inline namespace v1 {
36
37         template<typename Status>
38         void choose_normilization_factor(Status const &st,
39                                         bool &all_orthogonal=true,
40                                         int &weight_factor=0)
41         {
42             for(std::size_t i=0; i<st.options.constraints.size(); ++i)
43             {
44                 if(st.options.constraints[i] != Constraint::orthogonality)
45                 {
46                     all_orthogonal = false;
47                     weight_factor = i;
48                     break;
49                 }
50             }
51         }
52     }
53 }
54
55 #endif
```

```

67     }
68
69     template<std::size_t _TnsSize>
70     void Normalize( int weight_factor, const weight_factor,
71                   int const R,
72                   std::array<Matrix,_TnsSize> &gramian,
73                   std::array<Matrix,_TnsSize> &factors)
74     {
75         using Vector = Eigen::VectorXd;
76
77         constexpr std::size_t lastFactor = _TnsSize - 1;
78
79         int pass_flag = 0;
80         double cumul_power = 1;
81         bool nonZeroFlag = true;
82
83         std::array<Matrix,_TnsSize> normMatrixList;
84
85         std::array<Vector,lastFactor> lambda_fac;
86         std::array<double,lastFactor> norm_factor;
87
88         for(int i=0; i<static_cast<int>(_TnsSize); ++i)
89         {
90             if(i != weight_factor)
91             {
92                 norm_factor[i-pass_flag] = 1;
93                 lambda_fac[i-pass_flag] = Vector(static_cast<int>(R));
94                 lambda_fac[i-pass_flag] = gramian[i].diagonal();
95                 normMatrixList[i] = Matrix::Zero(static_cast<int>(R),static_cast<int>(R));
96             }
97             else
98             {
99                 normMatrixList[i] = Matrix::Ones(static_cast<int>(R),static_cast<int>(R));
100                pass_flag = 1;
101            }
102        }
103
104        for(int i=0; i<static_cast<int>(R); ++i)
105        {
106            for(int j=0; j<static_cast<int>(lastFactor); ++j)
107            {
108                norm_factor[j] = sqrt((lambda_fac[j])(i));
109                nonZeroFlag = !(norm_factor[j] == 0);
110            }
111
112            if(!nonZeroFlag)
113            {
114                for(int j=0; j<static_cast<int>(lastFactor); ++j)
115                    (lambda_fac[j])(i) = 1;
116            }
117            else
118            {
119                pass_flag = 0;
120                for(int j=0; j<static_cast<int>(lastFactor); ++j)
121                {
122                    if(j==weight_factor)
123                    {
124                        pass_flag = 1;
125                        continue;
126                    }
127                    factors[j].col(i) *= 1/norm_factor[j-pass_flag];
128                    cumul_power *= norm_factor[j-pass_flag];
129                    (lambda_fac[j-pass_flag])(i) = norm_factor[j-pass_flag];
130                }
131                factors[weight_factor].col(i) *= cumul_power;
132                cumul_power = 1;
133            }
134        }
135
136        pass_flag = 0;
137        for(int i=0; i<static_cast<int>(lastFactor); ++i)
138        {
139            if(i==weight_factor)
140            {
141                pass_flag = 1;
142                continue;
143            }
144            normMatrixList[i].noalias() = lambda_fac[i-pass_flag] * lambda_fac[i-pass_flag].transpose();
145            normMatrixList[weight_factor] = normMatrixList[weight_factor].cwiseProduct(normMatrixList[i]);
146            gramian[i] = gramian[i].cwiseQuotient(normMatrixList[i]);
147        }
148
149        gramian[weight_factor] = gramian[weight_factor].cwiseProduct(normMatrixList[weight_factor]);
150    }
151
152     template<std::size_t _TnsSize, typename DimensionType>

```

```

201 void Normalize( int                                const weight_factor,
202                int                                const R,
203                DimensionType                       const &tnsDims,
204                std::array<FactorDimTree,_TnsSize>   &factors)
205 {
206     using MatrixArray    = std::array<Matrix,_TnsSize>;
207     using FactorIterator = typename std::array<FactorDimTree,_TnsSize>::iterator;
208     using Vector         = Eigen::VectorXd;
209
210     constexpr std::size_t lastFactor = _TnsSize - 1;
211
212     int pass_flag = 0;
213     double cumul_power = 1;
214     bool nonZeroFlag = true;
215
216     MatrixArray factorsList;
217     MatrixArray gramMatrixList;
218     MatrixArray normMatrixList;
219
220     std::array<Vector,lastFactor> lambda_fac;
221     std::array<double,lastFactor> norm_factor;
222
223     FactorIterator factorPtr = factors.begin();
224
225     for(int i=0; i<static_cast<int>(_TnsSize); ++i)
226     {
227         factorsList[i] = Matrix(tnsDims[i],R);
228         factorsList[i] = tensorToMatrix(factorPtr->factor,tnsDims[i],static_cast<int>(R));
229
230         gramMatrixList[i] = Matrix(R,R);
231         gramMatrixList[i] = tensorToMatrix(factorPtr->gramian,static_cast<int>(R),static_cast<int>(R));
232
233         if(i != weight_factor)
234         {
235             norm_factor[i-pass_flag] = 1;
236             lambda_fac[i-pass_flag] = Vector(static_cast<int>(R));
237             lambda_fac[i-pass_flag] = gramMatrixList[i].diagonal();
238             normMatrixList[i] = Matrix::Zero(static_cast<int>(R),static_cast<int>(R));
239         }
240         else
241         {
242             normMatrixList[i] = Matrix::Ones(static_cast<int>(R),static_cast<int>(R));
243             pass_flag = 1;
244         }
245         factorPtr++;
246     }
247
248     for(int i=0; i<static_cast<int>(R); ++i)
249     {
250         for(int j=0; j<static_cast<int>(lastFactor); ++j)
251         {
252             norm_factor[j] = sqrt((lambda_fac[j])(i));
253             nonZeroFlag = !(norm_factor[j] == 0);
254         }
255
256         if(!nonZeroFlag)
257         {
258             for(int j=0; j<static_cast<int>(lastFactor); ++j)
259                 (lambda_fac[j])(i) = 1;
260         }
261         else
262         {
263             pass_flag = 0;
264             for(int j=0; j<static_cast<int>(lastFactor); ++j)
265             {
266                 if(j==weight_factor)
267                 {
268                     pass_flag = 1;
269                     continue;
270                 }
271                 factorsList[j].col(i) *= 1/norm_factor[j-pass_flag];
272                 cumul_power *= norm_factor[j-pass_flag];
273                 (lambda_fac[j-pass_flag])(i) = norm_factor[j-pass_flag];
274             }
275             factorsList[weight_factor].col(i) *= cumul_power;
276             cumul_power = 1;
277         }
278     }
279
280     pass_flag = 0;
281     for(int i=0; i<static_cast<int>(lastFactor); ++i)
282     {
283         if(i==weight_factor)
284         {
285             pass_flag = 1;
286             continue;
287         }

```



```

288     normMatrixList[i].noalias() = lambda_fac[i-pass_flag]          * lambda_fac[i-pass_flag].transpose();
289     normMatrixList[weight_factor] = normMatrixList[weight_factor].cwiseProduct(normMatrixList[i]);
290     gramMatrixList[i]           = gramMatrixList[i].cwiseQuotient(normMatrixList[i]);
291 }
292
293     gramMatrixList[weight_factor] =
gramMatrixList[weight_factor].cwiseProduct(normMatrixList[weight_factor]);
294     factorPtr
           = factors.begin();
295
296     for(int i=0; i<static_cast<int>(_TnsSize); ++i)
297     {
298         // Fill with the normalized factors.
299         factorPtr->factor = matrixToTensor(factorsList[i], tnsDims[i], static_cast<int>(R));
300         factorPtr->gramian = matrixToTensor(gramMatrixList[i], static_cast<int>(R),
static_cast<int>(R));
301         factorPtr++;
302     }
303 }
304
305 } // end namespace v1
306
307 } // end namespace partensor
308
309 #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, nalize, 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 overridden 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_ber_grid (cartesian_communicator const &grid, std::vector< cartesian_communicator > &ber_comm, std::array< int, _TnsSize > &ber_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 overridden with an output 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
```

Typedef for `cartesian_communicator` class from boost.

8.47.2.2 Boost_CartDimension

```
using Boost_CartDimension = boost::mpi::cartesian_dimension
```

Typedef for `cartesian_dimension` class from boost.

8.47.2.3 Boost_CartTopology

```
using Boost_CartTopology = boost::mpi::cartesian_topology
```

Typedef for `cartesian_topology` class from boost.

8.47.2.4 Boost_Communicator

```
using Boost_Communicator = boost::mpi::communicator
```

Typedef 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

<code>comm</code>	[in] The communicator over which the reduction will occur.
<code>value</code>	[in] The local value to be combined with the local values of every other process. For reducing arrays, <code>in_values</code> is a pointer to the local values to be reduced and <code>n</code> is the number of values to reduce. See <code>reduce</code> for more information.

If wrapped in an `inplace_t` object, combine the usage of both input and `out_value` and the local value will be overwritten (a convenience function `inplace` is provided for the wrapping).

Parameters

<code>out_value</code>	[in,out] Will receive the result of the reduction operation. If this parameter is omitted, the outgoing value will instead be returned.
<code>n</code>	[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_ber_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 ber grid in a cartesian_communicator .

Template Parameters

_TnsSize	Order of the Tensor.
----------	----------------------

Parameters

grid	[in] The MPI_COMM_WORLD implemented in a cartesian communicator .
ber_comm	[in,out] An stl vector containing the newly created layer communicator, that still belong in grid .
ber_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

<code>dis</code>	[in,out] The number of "lines" to skip per processor.
<code>count</code>	[in,out] The number of "lines" to read per processor.
<code>size</code>	[in] Number of processors.
<code>dim</code>	[in] Tensor dimensions.
<code>rank</code>	[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 overridden with an output value.

Parameters

<code>inout</code>	the contributing input value, it will be overridden 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.](#)

```
1 #ifndef DOXYGEN_SHOULD_SKIP_THIS
15 #endif // DOXYGEN_SHOULD_SKIP_THIS
16 / *****
24 #ifndef PARTENSOR_PARALLEL_WRAPPER_HPP
25 #define PARTENSOR_PARALLEL_WRAPPER_HPP
26
27 #include <vector>
28 #include "boost/mpi/communicator.hpp"
29 #include "boost/mpi/collectives.hpp"
30 #include "boost/mpi/environment.hpp"
31 #include "boost/mpi/cartesian_communicator.hpp"
32
33 namespace partensor {
```

```

34
35 inline namespace v1 {
40 using Boost_Environment = boost::mpi::environment;
41 using Boost_Communicator = boost::mpi::communicator;
42 using Boost_CartDimension = boost::mpi::cartesian_dimension;
43 using Boost_CartTopology = boost::mpi::cartesian_topology;
44 using Boost_CartCommunicator = boost::mpi::cartesian_communicator;
61 struct environment : public Boost_Environment
62 {
76     environment(int &argc, char ** &argv, bool abort_on_exception = true) : Boost_Environment(argc,
    argv, abort_on_exception) {};
77
90     ~environment() = default;
91
92     using Boost_Environment::abort;
93 };
94
105 struct communicator : public Boost_Communicator
106 {
113     communicator() : Boost_Communicator() { }
114
115     using Boost_Communicator::rank;
116     using Boost_Communicator::size;
117     using Boost_Communicator::barrier;
118 };
119
123 struct cartesian_dimension : public Boost_CartDimension
124 {
129     cartesian_dimension(int sz = 0, bool p = true) : Boost_CartDimension(sz,p) {}
130 };
131
138 struct cartesian_topology : public Boost_CartTopology
139 {
144     template<class InitArr_>
145     cartesian_topology(InitArr_ dims) : Boost_CartTopology(dims) {};
146 };
147
156 struct cartesian_communicator : public Boost_CartCommunicator
157 {
158     friend struct communicator;
159     using communicator::rank;
160
181     cartesian_communicator(communicator const &comm,
182                            cartesian_topology const &dims,
183                            bool reorder = true) : Boost_CartCommunicator(comm,
    cartesian_topology(dims), reorder) {}
184
192     cartesian_communicator(cartesian_communicator const &comm,
193                            std::vector<int> const &keep) : Boost_CartCommunicator(comm, keep)
194 {}
195 };
196
200 template<typename T>
201 using inplace_t = typename boost::mpi::inplace_t<T>;
202
212 // template<typename T>
213 // inplace_t<T> inplace(T& inout) {
214 //     return inplace_t<T>(inout);
215 // }
216
217 template<typename T>
218 inplace_t<T *> inplace(T * inout) {
219     return inplace_t<T *>(inout);
220 }
221
225 template<typename T>
226 using maximum = typename boost::mpi::maximum<T>;
227
231 template<typename T>
232 using minimum = typename boost::mpi::minimum<T>;
233
273 template<typename T, typename Op>
274 inline void all_reduce(const cartesian_communicator &comm,
275                      inplace_t<T *> value,
276                      int n,
277                      Op op)
278 {
279     boost::mpi::all_reduce(static_cast<const Boost_CartCommunicator &>(comm), value, n, op);
280 }
281
282 template<typename T, typename Op>
283 inline void all_reduce(const cartesian_communicator &comm,
284                      T &value,
285                      T &out_value,
286                      Op op)
287 {
288     boost::mpi::all_reduce(static_cast<const Boost_CartCommunicator &>(comm), value, out_value, op);

```

```

289 }
290
291 template<typename T, typename Op>
292 inline void all_reduce( const cartesian_communicator &comm,
293                       const T * value,
294                       int n,
295                       T * out_value,
296                       Op op)
297 {
298     boost::mpi::all_reduce(static_cast<const Boost_CartCommunicator &>(comm), value, n, out_value,
299 op);
300 }
301
302 /*
303  * @brief Combine the values stored by each process into a single
304  * value at the root.
305  *
306  * @c reduce is a collective algorithm that combines the values
307  * stored by each process into a single value at the @c root. The
308  * values can be combined arbitrarily, specified via a function
309  * object. The type @c T of the values may be any type that is
310  * serializable or has an associated MPI data type. One can think of
311  * this operation as a @c gather to the @p root, followed by an @c
312  * std::accumulate() over the gathered values and using the operation
313  * @c op.
314  *
315  * When the type @c T has an associated MPI data type, this routine
316  * invokes @c MPI_Reduce to perform the reduction. If possible,
317  * built-in MPI operations will be used; otherwise, @c reduce() will
318  * create a custom MPI_Op for the call to MPI_Reduce.
319  *
320  * @param comm [in] The communicator over which the reduction will
321  * occur.
322  *
323  * @param in_values [in] The local value to be combined with the local
324  * values of every other process. For reducing arrays, @c in_values
325  * contains a pointer to the local values. In this case, @c n is
326  * the number of values that will be reduced. Reduction occurs
327  * independently for each of the @p n values referenced by @p
328  * in_values, e.g., calling reduce on an array of @p n values is
329  * like calling @c reduce @p n separate times, one for each
330  * location in @p in_values and @p out_values.
331  *
332  * @param out_values [in,out] Will receive the result of the reduction
333  * operation, but only for the @p root process. Non-root processes
334  * may omit if parameter; if they choose to supply the parameter,
335  * it will be unchanged. For reducing arrays, @c out_values
336  * contains a pointer to the storage for the output values.
337  *
338  * @param op [in] The binary operation that combines two values of type
339  * @c T into a third value of type @c T. For types @c T that has
340  * associated MPI data types, @c op will either be translated into
341  * an @c MPI_Op (via @c MPI_Op_create) or, if possible, mapped
342  * directly to a built-in MPI operation. See @c is_mpi_op in the @c
343  * operations.hpp header for more details on this mapping. For any
344  * non-built-in operation, commutativity will be determined by the
345  * @c is_commutative trait (also in @c operations.hpp): users are
346  * encouraged to mark commutative operations as such, because it
347  * gives the implementation additional latitude to optimize the
348  * reduction operation.
349  *
350  * @param root [in] The process ID number that will receive the final,
351  * combined value. This value must be the same on all processes.
352  */
353 template<typename T, typename Op>
354 void reduce(const cartesian_communicator &comm,
355            const T * in_values,
356            int n,
357            T* out_values,
358            Op op,
359            int root )
360 {
361     boost::mpi::reduce(static_cast<const Boost_CartCommunicator &>(comm), in_values, n, out_values,
362 op, root );
363 }
364
365 /*
366  * @brief Similar to boost::mpi::scatter with the difference that the number
367  * of values stored at the root process does not need to be a multiple of
368  * the communicator's size.
369  *
370  * @param comm [in] The communicator over which the scatter will occur.
371  *
372  * @param in_values [in] A vector or pointer to storage that will contain
373  * the values to send to each process, indexed by the process rank.
374  * For non-root processes, this parameter may be omitted. If it is
375  * still provided, however, it will be unchanged.

```

```

374 *
375 * @param sizes [in] A vector containing the number of elements each non-root
376 * process will receive.
377 *
378 * @param out_values [in,out] The array of values received by each process.
379 *
380 * @param root [in] The process ID number that will scatter the
381 * values. This value must be the same on all processes.
382 */
383 template<typename T>
384 void scatterv(const cartesian_communicator &comm,
385             const T * in_values,
386             const std::vector<int> &sizes,
387             T* out_values,
388             int root )
389 {
390     boost::mpi::scatterv(static_cast<const Boost_CartCommunicator &>(comm), in_values, sizes,
391 out_values, root);
392 }
393
394 /*
395 * @brief Gather the values stored at every process into a vector of
396 * values from each process.
397 *
398 * @c all_gatherv is a collective algorithm that collects the values
399 * stored at each process into a vector of values at each
400 * process. This vector is indexed by the process number that the
401 * value came from. The type @c T of the values may be any type that
402 * is serializable or has an associated MPI data type.
403 *
404 * @param comm The communicator over which the gather will occur.
405 *
406 * @param in_values The array of values to be transmitted by each process.
407 *
408 * @param in_size For each process this specifies the size of @p in_values.
409 *
410 * @param out_values A pointer to storage that will be populated with
411 * the values from each process.
412 *
413 * @param sizes A vector containing the number of elements each
414 * process will send.
415 *
416 * @param displs A vector such that the i-th entry specifies the
417 * displacement (relative to @p out_values) from which to take the ingoing
418 * data at the @p root process. Overloaded versions for which @p displs is
419 * omitted assume that the data is to be placed contiguously at each process.
420 */
421 template<typename T>
422 void all_gatherv( const cartesian_communicator &comm,
423                const T * in_values,
424                int in_size,
425                T* out_values,
426                const std::vector<int> &sizes,
427                const std::vector<int> &displs )
428 {
429     for(int layerRank = 0; layerRank<comm.size(); layerRank++)
430         boost::mpi::gatherv(static_cast<const Boost_CartCommunicator &>(comm), in_values, in_size,
431 out_values, sizes, displs, layerRank);
432 }
433
434 template<std::size_t _TnsSize>
435 void create_layer_grid( cartesian_communicator &grid,
436                      std::vector<cartesian_communicator> &layer_comm,
437                      std::array<int, _TnsSize> &layer_rank )
438 {
439     std::vector<int> layer_dims(_TnsSize-1);
440     std::array<int, _TnsSize> free_coords;
441
442     for (std::size_t i = 0; i < _TnsSize; ++i)
443     {
444         std::fill(free_coords.begin(), free_coords.end(), 1);
445         free_coords[i] = 0;
446         int pos = 0;
447         for (std::size_t j = 0; j < _TnsSize; ++j) {
448             if (free_coords[j]) {
449                 layer_dims[pos++] = free_coords[j] * j;
450             }
451         }
452         // create the sub communicator in the cartesian communicator for layers
453         layer_comm.push_back(cartesian_communicator(grid, layer_dims));
454         // ID for each processor in layers sub communicator
455         layer_rank[i] = layer_comm[i].rank();
456     }
457 }
458
459 template<std::size_t _TnsSize>

```



```

483 void create_fiber_grid( cartesian_communicator const &grid,
484                       std::vector<cartesian_communicator> &fiber_comm,
485                       std::array<int, _TnsSize> &fiber_rank )
486 {
487     std::vector<int> fiber_dims(1);
488
489     for (std::size_t i = 0; i < _TnsSize; ++i)
490     {
491         fiber_dims[0] = i;
492         // create the sub communicator in the cartesian communicator for fibers
493         fiber_comm.push_back(cartesian_communicator(grid, fiber_dims));
494         // ID for each processor in fibers sub communicator
495         fiber_rank[i] = fiber_comm[i].rank();
496     }
497 }
498
499 void DisCount(int *dis, int *count, int const size, int const dim, std::size_t const rank)
500 {
501     int x = dim / size;
502     int y = dim % size;
503     for (int i=0; i<size; i++)
504     {
505         count[i] = (i >= y) ? x * rank : (x+1) * rank;
506         dis[i] = 0;
507
508         for (int j=0; j<i; j++)
509             dis[i] += count[j];
510     }
511 }
512
513 void DisCount(std::vector<int> &dis, std::vector<int> &count, int const size, int const dim,
514 std::size_t const rank)
515 {
516     int x = dim / size;
517     int y = dim % size;
518
519     for (int i=0; i<size; i++)
520     {
521         count.push_back((i >= y) ? x * rank : (x+1) * rank);
522         dis.push_back(0);
523
524         for (int j=0; j<i; j++)
525             dis[i] += count[j];
526     }
527 }
528 } // end namespace v1
529
530 #ifndef DOXYGEN_SHOULD_SKIP_THIS
531 namespace v2 {
532
533     /*
534     * Wrapper for MPI_Init, which initializes the MPI execution environment.
535     *
536     * @param argc [in] Pointer to the number of arguments.
537     * @param argv [in] Argument vector.
538     */
539 void Init(int argc, char ** argv)
540 {
541     MPI_Init(&argc, &argv);
542 }
543
544     /*
545     * Wrapper for MPI_Comm_rank, which determines the rank of the calling
546     * process in the communicator.
547     *
548     * @param comm [in] Communicator.
549     * @param rank [in,out] Rank of the calling process in group of comm.
550     */
551 void Comm_Rank( MPI_Comm const &comm,
552                int &rank )
553 {
554     MPI_Comm_rank(comm, &rank);
555 }
556
557     /*
558     * Wrapper for MPI_Comm_size, which returns the size of the group
559     * associated with a communicator.
560     *
561     * @param comm [in] Communicator.
562     * @param size [int,out] Number of processes in the group of comm.
563     */
564 void Comm_Size( MPI_Comm const &comm,
565                int &size )
566 {
567     MPI_Comm_size(comm, &size);
568 }
569 }

```

```

580
581 / *
582  * Wrapper for MPI_Abort, which terminates MPI execution environment.
583  *
584  * @param comm [in] Communicator.
585  * @param errorCode [in] Error code to return to invoking environment.
586  */
587 void Abort( MPI_Comm const &comm,
588            int const errorCode )
589 {
590     MPI_Abort(comm, errorCode);
591 }
592
593 template <typename DataType_>
594 void all_reduce( MPI_Comm const &comm,
595                double const size,
596                DataType_ &dt )
597 {
598     // In case of @c Eigen Matrix
599     MPI_Allreduce(MPI_IN_PLACE, dt.data(), size, MPI_DOUBLE, MPI_SUM, comm);
600 }
601
602 template <typename DataType_>
603 void all_reduce ( MPI_Comm const &comm,
604                 DataType_ &value,
605                 DataType_ &out_value,
606                 int const size )
607 {
608     MPI_Allreduce(value.data(), out_value.data(), size, MPI_DOUBLE, MPI_SUM, comm);
609 }
610
611 template <typename DataType_>
612 void gather( MPI_Comm const &comm,
613            DataType_ const &sendBuf,
614            int const sendSize,
615            int const &recvSize,
616            int const &displs,
617            int const root,
618            DataType_ &recvBuf )
619 {
620     MPI_Gather(sendBuf.data(), sendSize, MPI_DOUBLE, recvBuf.data(), &recvSize, &displs, MPI_DOUBLE,
621               root, comm);
622 }
623
624 template <typename DataType_>
625 void all_gather( MPI_Comm const &comm,
626                DataType_ const &sendBuf,
627                int const sendSize,
628                int const &recvSize,
629                int const &displs,
630                DataType_ &recvBuf )
631 {
632     MPI_Allgather(sendBuf.data(), sendSize, MPI_DOUBLE, recvBuf.data(), &recvSize, &displs,
633                  MPI_DOUBLE, comm);
634 }
635
636 template <typename DataType_>
637 void reduce_scatter( MPI_Comm const &comm,
638                    DataType_ const &sendBuf,
639                    int const &recvCounts,
640                    DataType_ &recvBuf )
641 {
642     MPI_Reduce_scatter(sendBuf.data(), recvBuf.data(), &recvCounts, MPI_DOUBLE,
643                       MPI_SUM, comm);
644 }
645
646 / *
647  * Wrapper for MPI_Cart_create, which makes a new communicator to which
648  * Cartesian topology information has been attached.
649  * @tparam _Size Number of dimensions of Cartesian grid.
650  * @param dims [in] Array specifying the number of processes in each dimension.
651  * @param periods [in] Logical array specifying whether the grid is periodic (1 = true) or not
652  * (0 = false) in each dimension.
653  * @param reorder [in] Ranking may be reordered (1 = true) or not (0 = false).
654  * @param comm [in,out] Communicator with new Cartesian topology.
655  */
656 template<std::size_t _Size>
657 void Cart_Create( std::array<int,_Size> const &dims,
658                 std::array<int,_Size> const &periods,
659                 int const reorder,
660                 MPI_Comm &comm )
661 {
662     MPI_Cart_create(MPI_COMM_WORLD, static_cast<int>(_Size), dims.data(), periods.data(), reorder,
663                   &comm);
664 }
665
666 / *

```

```

704     * Wrapper for MPI_Cart_coords, which determines process coords in
705     * Cartesian topology given rank in group.
706     * @tparam _Size      Number of dimensions of Cartesian grid.
707     * @param comm [in]   Communicator with Cartesian structure.
708     * @param rank [in]   Rank of a process within group of comm.
709     * @param coords [in,out] Array containing the Cartesian coordinates of specified process.
710     */
711     template<std::size_t _Size>
712     void Cart_Coords( MPI_Comm          const &comm,
713                    int                const rank,
714                    std::array<int, _Size> &coords)
715     {
716         MPI_Cart_coords(comm, rank, static_cast<int>(_Size), coords.data());
717     }
718
719     /*
720     * Wrapper for MPI_Cart_sub, which partitions a communicator into subgroups,
721     * and form lower-dimensional Cartesian subgrids.
722     * @tparam _Size      Number of dimensions of Cartesian grid.
723     * @param comm [in]   Communicator with Cartesian structure.
724     * @param rDims [in]   The ith entry of rDims specifies whether the ith dimension is kept in
725     *                     the subgrid (1 = true) or is dropped (0 = false).
726     * @param subComm [in,out] Communicator containing the subgrid that includes the calling process.
727     */
728     template<std::size_t _Size>
729     void Cart_Sub( MPI_Comm          const &comm,
730                 std::array<int, _Size> const &rDims,
731                 MPI_Comm          &subComm )
732     {
733         MPI_Cart_sub(comm, rDims.data(), &subComm);
734     }
735
736     /*
737     * Wrapper for MPI_Barrier, for synchronization between MPI processes
738     * @param comm [in] Communicator.
739     */
740     void Barrier(MPI_Comm const &comm)
741     {
742         MPI_Barrier(comm);
743     }
744
745     /*
746     * Wrapper for Comm_Free, which marks a communicator object for deallocation.
747     * @param comm [in] Communicator.
748     */
749     void Comm_Free(MPI_Comm &comm)
750     {
751         MPI_Comm_free(&comm);
752     }
753
754     /*
755     * Wrapper for MPI_Finalize, that checks whether MPI has been finalized.
756     */
757     void Finalize()
758     {
759         MPI_Finalize();
760     }
761 } // end namespace v2
762 #endif // DOXYGEN_SHOULD_SKIP_THIS
763
764 } // end namespace partensor
765
766 #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

Contains all the header files created for PARTENSOR Project.

8.50 PARTENSOR.hpp

[Go to the documentation of this file.](#)

```
1 #ifndef DOXYGEN_SHOULD_SKIP_THIS
15 #endif // DOXYGEN_SHOULD_SKIP_THIS
16 / ***** /
22 #ifndef PARTENSOR_HPP
23 #define PARTENSOR_HPP
24
25 #include "PARTENSOR_basic.hpp"
26 #include "Constants.hpp"
27 #include "CwiseProd.hpp"
28 #include "KhatriRao.hpp"
29 #include "Kronecker.hpp"
30 #include "DataGeneration.hpp"
31 #include "ReadWrite.hpp"
32 #include "Matricization.hpp"
33 #include "Tensor.hpp"
34 #include "TensorOperations.hpp"
35 #include "Timers.hpp"
36 #include "DimTrees.hpp"
37 #include "Cpd.hpp"
38 #include "Gtc.hpp"
39 #include "GtcStochastic.hpp"
40 #include "CpdDimTree.hpp"
41 #include "ParallelWrapper.hpp"
42
43 #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 Default - Values_ = DefaultValues> Options< Tensor_, ExecutionPolicy_, DefaultValues_ > MakeOptions ()`
- `template< typename Tensor_ , typename ExecutionPolicy_ = execution::sequenced_policy, template< typename T > class Default - Values_ = 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 Default - Values_ = SparseDefaultValues> SparseOptions< _TnsSize, ExecutionPolicy_, DefaultValues_ > MakeSparseOptions ()`
- `template< std::size_t _TnsSize, typename ExecutionPolicy_ = execution::sequenced_policy, template< typename T > class Default - Values_ = SparseDefaultValues> SparseOptions< _TnsSize, ExecutionPolicy_, DefaultValues_ > MakeSparseOptions (DefaultValues_ - < partensor::SparseTensor< _TnsSize > > &&dv, ExecutionPolicy_ &&xp)`

8.51.1 Detailed Description

Contains 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 [le](#).

```

1 #ifndef DOXYGEN_SHOULD_SKIP_THIS
15 #endif // DOXYGEN_SHOULD_SKIP_THIS
16 / ..... /
23 #ifndef PARTENSOR_BASIC_HPP
24 #define PARTENSOR_BASIC_HPP
25
26 #define EIGEN_PERMANENTLY_DISABLE_STUPID_WARNINGS 1
27
28 #if defined __GNUC__ && __GNUC__ >=6
29 # pragma GCC diagnostic push
30 # pragma GCC diagnostic ignored "-Wignored-attributes"
31 # pragma GCC diagnostic ignored "-Wunknown-pragmas"
32 #endif
33
34 #include "Config.hpp"
35
36 #include <mutex>
37
38 #if USE_MPI
39 #include "ParallelWrapper.hpp"
40 #endif / * USE_MPI */
41
42 #include <Eigen/Dense>
43 #include <Eigen/Sparse>
44 #include <unsupported/Eigen/CXX11/Tensor>
45 #include "spdlog/spdlog.h"
46 #include "spdlog/sinks/basic_file_sink.h"
47 #include "execution.hpp"
48 #include "Constants.hpp"
49 #include "Tensor.hpp"
50
51 namespace partensor
52 {
53 #if USE_MPI
54 using MPI_Environment = partensor::environment;
55 using MPI_Communicator = partensor::communicator;
56 #endif / * USE_MPI */
57
58 using Clock = std::chrono::high_resolution_clock;
59 using Duration = std::chrono::nanoseconds; // microseconds??
60
61 class Environment
62 {
63 public:
64 Environment(int argc, char * argv[], char * envp[])
65 #if USE_MPI
66 : mMPIEnv(argc,argv,true)
67 #endif / * USE_MPI */
68 {
69 (void) argc;
70 (void) argv;
71 (void) envp;
72
73 // create an spdlog object with pattern:
74 // %l = The log level of the message (eg info, warn)
75 // %Y-%m-%d = Year in 4 digits - Month 01-12 - Day of month 01-31 (eg 2019-09-19)
76 // %r = 12 hour clock (eg 02:55:02 pm)
77 // %F = Nanosecond part of the current second 000000000-999999999 (eg 256789123)
78 // %n = Logger's name (eg some logger name)
79 // %v = The actual text to log (eg "some user text")
80 spdlog::set_pattern("<%=l> : [%Y-%m-%d %r] [%F] [%n] %v");
81 mLogger = spdlog::basic_logger_mt("Partensor", "../log/partensor.txt");
82 }
83
84 Environment(int argc, char * argv[]) : Environment(argc,argv,nullptr)
85 { }
86
87 Environment() : Environment(0, nullptr,nullptr)
88 { }
89
90 ~Environment()
91 {
92 mLogger->flush();
93 // TODO spdlog::drop("Partensor");
94 }
95
96 static Environment *Partensor(int argc, char ** argv, char ** envp)
97 {
98 static std::mutex l_mutex;
99 static std::unique_ptr<Environment> l_partensor(nullptr);
100
101 if (!l_partensor)

```

```

102     {
103         std::lock_guard<std::mutex> lock(_mutex);
104
105         if (!_partensor)
106         {
107             if (argc == 0)
108                 _partensor.reset(new Environment());
109             else if (envp == nullptr)
110                 _partensor.reset(new Environment(argc,argv));
111             else
112                 _partensor.reset(new Environment(argc,argv,envp));
113         }
114     }
115
116     return _partensor.get();
117 }
118
119 std::shared_ptr<spdlog::logger> Logger()
120 {
121     return mLogger;
122 }
123
124 #if USE_MPI
125 MPI_Environment &MpiEnvironment()
126 {
127     return mMpiEnv;
128 }
129
130 MPI_Communicator &MpiCommunicator()
131 {
132     return mMpiCom;
133 }
134
135 #endif / * USE_MPI */
136 private:
137     std::shared_ptr<spdlog::logger> mLogger;
138
139 #if USE_MPI
140     MPI_Environment mMpiEnv;
141     MPI_Communicator mMpiCom;
142 #endif / * USE_MPI */
143 };
144
145 inline Environment *Partensor(int argc=0, char ** argv=nullptr, char ** envp=nullptr)
146 {
147     return Environment::Partensor(argc,argv,envp);
148 }
149
150 inline void Init(int argc, char ** argv, char ** envp)
151 {
152     Partensor(argc,argv,envp);
153 }
154
155 inline void Init(int argc, char ** argv)
156 {
157     Partensor(argc,argv);
158 }
159
160 inline void Init()
161 {
162     Partensor();
163 }
164
165 template<typename Tensor_>
166 struct DefaultValues {
167
168     static std::size_t constexpr TnsSize = TensorTraits<Tensor_>::TnsSize;
169
170     using DoubleArray = typename TensorTraits<Tensor_>::DoubleArray;
171     using Constraints = typename TensorTraits<Tensor_>::Constraints;
172     using IntArray = typename TensorTraits<Tensor_>::IntArray;
173
174     static Method constexpr DefaultMethod = Method::als;
175     // static Constraint constexpr DefaultConstraint = Constraint::unconstrained; / * < Default value
176     for Constraint is unconstrained. */
177
178     static Constraint constexpr DefaultConstraint = Constraint::nonnegativity;
179     static double constexpr DefaultThresholdError = 1e-3;
180     static double constexpr DefaultNesterovTolerance = 1e-2;
181     static unsigned constexpr DefaultMaxIter = 20;
182     static Duration constexpr DefaultMaxDuration = Duration(10000);
183     static double constexpr DefaultLambda = 0.1;
184     static double constexpr DefaultProcessorPerMode = 2;
185     static int constexpr DefaultAccelerationCoefficient = 3;
186     static int constexpr DefaultAccelerationFail = 0;
187     static bool constexpr DefaultAcceleration = true;
188     static bool constexpr DefaultNormalization = true;
189     static bool constexpr DefaultWriteToFile = false;

```



```

200     static DoubleArray constexpr DefaultLambdas = []() constexpr -> auto {
201         DoubleArray c{};
202         for (auto &e : c) e = DefaultLambda;
203         return c;
204     } ();
205
206     static Constraints constexpr DefaultConstraints = []() constexpr -> auto {
207         Constraints c{};
208         for (auto &e : c) e = DefaultConstraint;
209         return c;
210     } ();
211
212     static IntArray constexpr DefaultProcessorsPerMode = []() constexpr -> auto {
213         IntArray c{};
214         for(auto &e : c) e = DefaultProcessorPerMode;
215         return c;
216     } ();
217
218 };
219
220 template<typename SparseTensor_>
221 struct SparseDefaultValues {
222
223     static std::size_t constexpr TnsSize = SparseTensorTraits<SparseTensor_>::TnsSize;
224
225     using DoubleArray = typename SparseTensorTraits<SparseTensor_>::DoubleArray;
226     using Constraints = typename SparseTensorTraits<SparseTensor_>::Constraints;
227     using IntArray = typename SparseTensorTraits<SparseTensor_>::IntArray;
228
229     static Method constexpr DefaultMethod = Method::als;
230     // static Constraint constexpr DefaultConstraint = Constraint::unconstrained; / ** < Default value
231     for Constraint is unconstrained. */
232     static Constraint constexpr DefaultConstraint = Constraint::nonnegativity;
233     static double constexpr DefaultThresholdError = 1e-3;
234     static double constexpr DefaultNesterovTolerance = 1e-2;
235     static int constexpr DefaultMaxNesterovIter = 20;
236     static unsigned constexpr DefaultMaxIter = 20;
237     static Duration constexpr DefaultMaxDuration = Duration(10000);
238     static double constexpr DefaultC_stochastic_perc = 0.5;
239     static double constexpr DefaultLambda = 0.01;
240     static double constexpr DefaultProcessorPerMode = 2;
241     static int constexpr DefaultAccelerationCoefficient = 3;
242     static int constexpr DefaultAccelerationFail = 0;
243     static bool constexpr DefaultAcceleration = false;
244     static bool constexpr DefaultAveraging = false;
245     static bool constexpr DefaultNormalization = false;
246     static bool constexpr DefaultInitializeFactors = false;
247     static bool constexpr DefaultReadFactorsFromFile = false;
248     static bool constexpr DefaultWriteToFile = false;
249     static int constexpr DefaultNonZeros = 1000;
250     static DoubleArray constexpr DefaultLambdas = []() constexpr -> auto {
251         DoubleArray c{};
252         for (auto &e : c) e = DefaultLambda;
253         return c;
254     } ();
255
256     static Constraints constexpr DefaultConstraints = []() constexpr -> auto {
257         Constraints c{};
258         for (auto &e : c) e = DefaultConstraint;
259         return c;
260     } ();
261
262     static IntArray constexpr DefaultProcessorsPerMode = []() constexpr -> auto {
263         IntArray c{};
264         for(auto &e : c) e = DefaultProcessorPerMode;
265         return c;
266     } ();
267
268 };
269
270
271
272
273 template < typename Tensor_,
274             typename ExecutionPolicy_ = execution::sequenced_policy,
275             template <typename T> class DefaultValues_ = DefaultValues >
276 struct Options
277 {
278     static constexpr std::size_t TnsSize = TensorTraits<Tensor_>::TnsSize;
279     using DataType = typename TensorTraits<Tensor_>::DataType;
280     using MatrixType = typename TensorTraits<Tensor_>::MatrixType;
281     using Constraints = typename TensorTraits<Tensor_>::Constraints;
282     using DoubleArray = typename TensorTraits<Tensor_>::DoubleArray;
283     using StringArray = std::array<std::string, TnsSize>;
284
285     Method method;
286     Constraints constraints;
287     double threshold_error;
288     double nesterov_delta_1;
289     double nesterov_delta_2;

```

```

301     DoubleArray    lambdas;
302     unsigned      max_iter;
303     Duration      max_duration;
304     int           accel_coeff;
305     int           accel_fail;
306     bool          acceleration;
307     bool          normalization;
308     bool          writeToFile;
309     StringArray   final_factors_paths;
310
311     Options() : method(DefaultValues_<Tensor_>::DefaultMethod),
312               constraints(DefaultValues_<Tensor_>::DefaultConstraints),
313               threshold_error(DefaultValues_<Tensor_>::DefaultThresholdError),
314               nesterov_delta_1(DefaultValues_<Tensor_>::DefaultNesterovTolerance),
315               nesterov_delta_2(DefaultValues_<Tensor_>::DefaultNesterovTolerance),
316               lambdas(DefaultValues_<Tensor_>::DefaultLambdas),
317               max_iter(DefaultValues_<Tensor_>::DefaultMaxIter),
318               max_duration(DefaultValues_<Tensor_>::DefaultMaxDuration),
319               accel_coeff(DefaultValues_<Tensor_>::DefaultAccelerationCoefficient),
320               accel_fail(DefaultValues_<Tensor_>::DefaultAccelerationFail),
321               acceleration(DefaultValues_<Tensor_>::DefaultAcceleration),
322               normalization(DefaultValues_<Tensor_>::DefaultNormalization),
323               writeToFile(DefaultValues_<Tensor_>::DefaultWriteToFile)//,           /           ** < Default
value for write final factors to files.          */
324               // final_factors_paths(DefaultValues_<Tensor_>::DefaultFinalFactorsPaths) /           ** < Default
value for path for factors at the end of the algorithm.          */
325     {
326         for(std::size_t i=0; i<TnsSize; ++i)
327         {
328             final_factors_paths[i] = "final_" + std::to_string(i) + ".bin";
329         }
330     }
331     Options(Options const &) = default;
332     Options(Options &&) = default;
333
334     Options &operator=(Options const &) = default;
335     Options &operator=(Options &&) = default;
336 };
337
338 / *
339     Sparse Options
340 */
341 template < std::size_t _TnsSize,
342            typename ExecutionPolicy_ = execution::sequenced_policy,
343            template <typename T> class DefaultValues_ = SparseDefaultValues    >
344 struct SparseOptions
345 {
346     using SparseTensor = typename partensor::SparseTensor<_TnsSize>;
347     static constexpr std::size_t TnsSize = SparseTensorTraits<SparseTensor>::TnsSize;
348     using DataType     = typename SparseTensorTraits<SparseTensor>::DataType;
349     using MatrixType   = typename SparseTensorTraits<SparseTensor>::MatrixType;
350     using Constraints   = typename SparseTensorTraits<SparseTensor>::Constraints;
351     using DoubleArray  = typename SparseTensorTraits<SparseTensor>::DoubleArray;
352     using SparseMatrixType = typename SparseTensorTraits<SparseTensor>::SparseMatrixType;
353     using LongMatrixType = typename SparseTensorTraits<SparseTensor>::LongMatrixType;
354     using Dimensions   = typename SparseTensorTraits<SparseTensor>::Dimensions;
355     using SparseTensor = typename SparseTensorTraits<SparseTensor>::SparseTensor;
356     using MatrixArray  = typename SparseTensorTraits<SparseTensor>::MatrixArray;
357     using StringArray  = std::array<std::string, TnsSize>;
358
359     int rank;
360     std::array<int, TnsSize> tnsDims;
361     int nonZeros;
362     bool initialized_factors;
363     bool read_factors_from_file;
364     MatrixArray factorsInit;
365
366     Method method;
367     Constraints constraints;
368     double threshold_error;
369     double nesterov_delta_1;
370     double nesterov_delta_2;
371     int max_nesterov_iter;
372     double c_stochastic_perc;
373     DoubleArray lambdas;
374     unsigned max_iter;
375     Duration max_duration;
376     int accel_coeff;
377     int accel_fail;
378     bool acceleration;
379     bool averaging;
380     bool normalization;
381     bool writeToFile;
382
383     std::string ratings_path;
384     StringArray initial_factors_paths;
385     StringArray final_factors_paths;
386

```

```

387
388 SparseOptions() : initialized_factors(DefaultValues_<SparseTensor>::DefaultInitializeFactors),
389                 read_factors_from_file(DefaultValues_<SparseTensor>::DefaultReadFactorsFromFile),
390                 method(DefaultValues_<SparseTensor>::DefaultMethod),
391                 constraints(DefaultValues_<SparseTensor>::DefaultConstraints),
392                 threshold_error(DefaultValues_<SparseTensor>::DefaultThresholdError),
393                 nesterov_delta_1(DefaultValues_<SparseTensor>::DefaultNesterovTolerance),
394                 nesterov_delta_2(DefaultValues_<SparseTensor>::DefaultNesterovTolerance),
395                 max_nesterov_iter(DefaultValues_<SparseTensor>::DefaultMaxNesterovIter),
396                 c_stochastic_perc(DefaultValues_<SparseTensor>::DefaultC_stochastic_perc),
397                 lambdas(DefaultValues_<SparseTensor>::DefaultLambdas),
398                 max_iter(DefaultValues_<SparseTensor>::DefaultMaxIter),
399                 max_duration(DefaultValues_<SparseTensor>::DefaultMaxDuration),
400                 accel_coeff(DefaultValues_<SparseTensor>::DefaultAccelerationCoefficient),
401                 accel_fail(DefaultValues_<SparseTensor>::DefaultAccelerationFail),
402                 acceleration(DefaultValues_<SparseTensor>::DefaultAcceleration),
403                 averaging(DefaultValues_<SparseTensor>::DefaultAveraging),
404                 normalization(DefaultValues_<SparseTensor>::DefaultNormalization),
405                 writeToFile(DefaultValues_<SparseTensor>::DefaultWriteToFile)
406                 // final_factors_paths(DefaultValues_<SparseTensor>::DefaultFinalFactorsPaths)
407
408 /* < Default value for path for factors at the end of the algorithm. */
409 {
410     for(std::size_t i=0; i<TnsSize; ++i)
411     {
412         final_factors_paths[i] = "final_" + std::to_string(i) + ".bin";
413     }
414     SparseOptions(SparseOptions const &) = default;
415     SparseOptions(SparseOptions &&) = default;
416
417     SparseOptions &operator=(SparseOptions const &) = default;
418     SparseOptions &operator=(SparseOptions &&) = default;
419 };
420
421 template < typename Tensor_,
422           typename ExecutionPolicy_ = execution::sequenced_policy,
423           template <typename T> class DefaultValues_ = DefaultValues_ >
424 Options<Tensor_, ExecutionPolicy_, DefaultValues_> MakeOptions()
425 {
426     Options<Tensor_, ExecutionPolicy_, DefaultValues_> options;
427     static constexpr std::size_t TnsSize = TensorTraits<Tensor_>::TnsSize;
428     options.method = DefaultValues_<Tensor_>::DefaultMethod; // Default
429     value for Method is als. = DefaultValues_<Tensor_>::DefaultConstraints; // Default
430     value for Constraint is no negative.
431     options.threshold_error = DefaultValues_<Tensor_>::DefaultThresholdError; // Default
432     value for cost function's threshold.
433     options.nesterov_delta_1 = DefaultValues_<Tensor_>::DefaultNesterovTolerance; // Default
434     value for Nesterov's tolerance.
435     options.nesterov_delta_2 = DefaultValues_<Tensor_>::DefaultNesterovTolerance; // Default
436     value for Nesterov's tolerance.
437     options.lambdas = DefaultValues_<Tensor_>::DefaultLambdas; // Default
438     value for lambda.
439     options.max_iter = DefaultValues_<Tensor_>::DefaultMaxIter; // Default
440     value outer loop maximum iterations.
441     options.max_duration = DefaultValues_<Tensor_>::DefaultMaxDuration; // Default
442     value outer loop maximum duration.
443     options.accel_coeff = DefaultValues_<Tensor_>::DefaultAccelerationCoefficient; // Default
444     value for acceleration coefficient.
445     options.accel_fail = DefaultValues_<Tensor_>::DefaultAccelerationFail; // Default
446     value for acceleration fail.
447     options.acceleration = DefaultValues_<Tensor_>::DefaultAcceleration; // Default
448     value for acceleration.
449     // options.averaging = DefaultValues_<Tensor_>::DefaultAveraging; //
450     Default value for averaging.
451     options.normalization = DefaultValues_<Tensor_>::DefaultNormalization; // Default
452     value for normalization.
453     options.writeToFile = DefaultValues_<Tensor_>::DefaultWriteToFile; // Default
454     value for write final factors to files.
455     // options.final_factors_paths = DefaultValues_<Tensor_>::DefaultFinalFactorsPaths; //
456     Default value for path for factors at the end of the algorithm.
457     for(std::size_t i=0; i<TnsSize; ++i)
458     {
459         options.final_factors_paths[i] = "final_" + std::to_string(i) + ".bin";
460     }
461     return options;
462 }
463
464 template < std::size_t _TnsSize,
465           typename ExecutionPolicy_ = execution::sequenced_policy,
466           template <typename T> class DefaultValues_ = SparseDefaultValues_ >
467 SparseOptions<_TnsSize, ExecutionPolicy_, DefaultValues_> MakeSparseOptions()
468 {
469     SparseOptions<_TnsSize, ExecutionPolicy_, DefaultValues_> options;
470     using SparseTensor = typename partensor::SparseTensor<_TnsSize>;
471     static constexpr std::size_t TnsSize = SparseTensorTraits<SparseTensor>::TnsSize;
472     options.method = DefaultValues_<SparseTensor>::DefaultMethod; //
473 }

```

```

460     options.constraints      = DefaultValues_<SparseTensor>::DefaultConstraints;           //
Default value for Constraint is no negative.
461     options.threshold_error = DefaultValues_<SparseTensor>::DefaultThresholdError;       //
Default value for cost function's threshold.
462     options.nesterov_delta_1 = DefaultValues_<SparseTensor>::DefaultNesterovTolerance;   //
Default value for Nesterov's tolerance.
463     options.nesterov_delta_2 = DefaultValues_<SparseTensor>::DefaultNesterovTolerance;   //
Default value for Nesterov's tolerance.
464     options.c_stochastic_perc = DefaultValues_<SparseTensor>::DefaultC_stochastic_perc;
465     options.lambdas         = DefaultValues_<SparseTensor>::DefaultLambdas;             //
Default value for lambda.
466     options.max_iter        = DefaultValues_<SparseTensor>::DefaultMaxIter;             //
Default value outer loop maximum iterations.
467     options.max_duration    = DefaultValues_<SparseTensor>::DefaultMaxDuration;         //
Default value outer loop maximum duration.
468     options.accel_coeff     = DefaultValues_<SparseTensor>::DefaultAccelerationCoefficient; //
Default value for acceleration coefficient.
469     options.accel_fail      = DefaultValues_<SparseTensor>::DefaultAccelerationFail;     //
Default value for acceleration fail.
470     options.acceleration    = DefaultValues_<SparseTensor>::DefaultAcceleration;         //
Default value for acceleration.
471     options.averaging       = DefaultValues_<SparseTensor>::DefaultAveraging;           //
Default value for averaging.
472     options.normalization   = DefaultValues_<SparseTensor>::DefaultNormalization;       //
Default value for normalization.
473     options.writeToFile     = DefaultValues_<SparseTensor>::DefaultWriteToFile;         //
Default value for write final factors to files.
474     // options.final_factors_paths = DefaultValues_<SparseTensor>::DefaultFinalFactorsPaths; //
Default value for path for factors at the end of the algorithm.
475     for(std::size_t i=0; i<TnsSize; ++i)
476     {
477         options.final_factors_paths[i] = "final_" + std::to_string(i) + ".bin";
478     }
479     return options;
480 }
481
482 template < typename Tensor_,
483           typename ExecutionPolicy_ = execution::sequenced_policy,
484           template <typename T> class DefaultValues_ = DefaultValues_ >
485 Options<Tensor_, ExecutionPolicy_, DefaultValues_> MakeOptions(DefaultValues_<Tensor_> &&dv,
ExecutionPolicy_ &&xp)
486 {
487     Options<Tensor_, ExecutionPolicy_, DefaultValues_> options;
488     static constexpr std::size_t TnsSize = TensorTraits<Tensor_>::TnsSize;
489     options.method      = DefaultValues_<Tensor_>::DefaultMethod;           // Default
490
491     value for Method is als.
491     options.constraints      = DefaultValues_<Tensor_>::DefaultConstraints;           // Default
492     value for Constraint is no negative.
492     options.threshold_error = DefaultValues_<Tensor_>::DefaultThresholdError;       // Default
493     value for cost function's threshold.
493     options.nesterov_delta_1 = DefaultValues_<Tensor_>::DefaultNesterovTolerance;   // Default
494     value for Nesterov's tolerance.
494     options.nesterov_delta_2 = DefaultValues_<Tensor_>::DefaultNesterovTolerance;   // Default
495     value for Nesterov's tolerance.
495     options.lambdas         = DefaultValues_<Tensor_>::DefaultLambdas;             // Default
496     value for lambda.
496     options.max_iter        = DefaultValues_<Tensor_>::DefaultMaxIter;             // Default
497     value outer loop maximum iterations.
497     options.max_duration    = DefaultValues_<Tensor_>::DefaultMaxDuration;         // Default
498     value outer loop maximum duration.
498     options.accel_coeff     = DefaultValues_<Tensor_>::DefaultAccelerationCoefficient; // Default
499     value for acceleration coefficient.
499     options.accel_fail      = DefaultValues_<Tensor_>::DefaultAccelerationFail;     // Default
500     value for acceleration fail.
500     options.acceleration    = DefaultValues_<Tensor_>::DefaultAcceleration;         // Default
501     value for acceleration.
501     // options.averaging       = DefaultValues_<Tensor_>::DefaultAveraging;           //
Default value for averaging.
502     options.normalization   = DefaultValues_<Tensor_>::DefaultNormalization;       // Default
503     value for normalization.
503     options.writeToFile     = DefaultValues_<Tensor_>::DefaultWriteToFile;         // Default
504     value for write final factors to files.
504     // options.final_factors_paths = DefaultValues_<Tensor_>::DefaultFinalFactorsPaths; //
Default value for path for factors at the end of the algorithm.
505     for(std::size_t i=0; i<TnsSize; ++i)
506     {
507         options.final_factors_paths[i] = "final_" + std::to_string(i) + ".bin";
508     }
509     return options;
510 }
511
512 template < std::size_t _TnsSize,
513           typename ExecutionPolicy_ = execution::sequenced_policy,
514           template <typename T> class DefaultValues_ = SparseDefaultValues_ >
515 SparseOptions<_TnsSize, ExecutionPolicy_, DefaultValues_>
MakeSparseOptions(DefaultValues_<partensor::SparseTensor<_TnsSize> &&dv, ExecutionPolicy_ &&xp)

```

```

516 {
517     SparseOptions<_TnsSize,ExecutionPolicy_,DefaultValues_> options;
518     using SparseTensor = typename partensor::SparseTensor<_TnsSize>;
519     static constexpr std::size_t TnsSize = SparseTensorTraits<SparseTensor>::TnsSize;
520     options.method = DefaultValues_<SparseTensor>::DefaultMethod; //
521     Default value for Method is als.
522     options.constraints = DefaultValues_<SparseTensor>::DefaultConstraints; //
523     Default value for Constraint is no negative.
524     options.threshold_error = DefaultValues_<SparseTensor>::DefaultThresholdError; //
525     Default value for cost function's threshold.
526     options.nesterov_delta_1 = DefaultValues_<SparseTensor>::DefaultNesterovTolerance; //
527     Default value for Nesterov's tolerance.
528     options.nesterov_delta_2 = DefaultValues_<SparseTensor>::DefaultNesterovTolerance; //
529     Default value for Nesterov's tolerance.
530     options.c_stochastic_perc = DefaultValues_<SparseTensor>::DefaultC_stochastic_perc; //
531     options.lambdas = DefaultValues_<SparseTensor>::DefaultLambdas; //
532     Default value for lambda.
533     options.max_iter = DefaultValues_<SparseTensor>::DefaultMaxIter; //
534     Default value outer loop maximum iterations.
535     options.max_duration = DefaultValues_<SparseTensor>::DefaultMaxDuration; //
536     Default value outer loop maximum duration.
537     options.accel_coeff = DefaultValues_<SparseTensor>::DefaultAccelerationCoefficient; //
538     Default value for acceleration coefficient.
539     options.accel_fail = DefaultValues_<SparseTensor>::DefaultAccelerationFail; //
540     Default value for acceleration fail.
541     options.acceleration = DefaultValues_<SparseTensor>::DefaultAcceleration; //
542     Default value for acceleration.
543     options.averaging = DefaultValues_<SparseTensor>::DefaultAveraging; //
544     // Default value for averaging.
545     options.normalization = DefaultValues_<SparseTensor>::DefaultNormalization; //
546     Default value for normalization.
547     options.writeToFile = DefaultValues_<SparseTensor>::DefaultWriteToFile; //
548     Default value for write final factors to files.
549     // options.final_factors_paths = DefaultValues_<SparseTensor>::DefaultFinalFactorsPaths; //
550     Default value for path for factors at the end of the algorithm.
551     for(std::size_t i=0; i<TnsSize; ++i)
552     {
553         options.final_factors_paths[i] = "final_" + std::to_string(i) + ".bin";
554     }
555     return options;
556 }
557
558 template < typename Tensor_,
559             typename ExecutionPolicy_ = execution::sequenced_policy,
560             template <typename T> class DefaultValues_ = DefaultValues >
561 struct Status
562 {
563     using MatrixArray = typename TensorTraits<Tensor_>::MatrixArray;
564     Options<Tensor_,ExecutionPolicy_,DefaultValues_> options;
565     double frob_tns = 0.0;
566     double f_value = 0.0;
567     double rel_costFunction = 0.0;
568     unsigned ao_iter = 0;
569     MatrixArray factors;
570     Status() = default;
571     Status(Status const &) = default;
572     Status(Status &&) = default;
573
574     Status(Options<Tensor_,ExecutionPolicy_,DefaultValues_> const &opt) : options(opt)
575     { }
576
577     Status &operator=(Status const &) = default;
578     Status &operator=(Status &&) = default;
579
580     /*
581     Constructor called, in case the user decides to change one or more
582     of the options and use them in the factorization.
583     */
584     Status &operator=(Options<Tensor_,ExecutionPolicy_,DefaultValues_> const &opts)
585     {
586         options = opts;
587     }
588     return * this;
589 }
590
591 explicit operator Options<Tensor_,ExecutionPolicy_,DefaultValues_>& ()
592 {
593     return options;
594 }
595 };
596
597 template < std::size_t _TnsSize,
598             typename ExecutionPolicy_ = execution::sequenced_policy,
599             template <typename T> class DefaultValues_ = SparseDefaultValues >
600 struct SparseStatus
601 {
602     using SparseTensor = typename partensor::SparseTensor<_TnsSize>;

```

```

606     using MatrixArray = typename SparseTensorTraits<SparseTensor>::MatrixArray;
607     SparseOptions<_TnsSize,ExecutionPolicy_,DefaultValues_> options;
608     double                frob_tns           = 0.0;
609     double                f_value           = 0.0;
610     double                rel_costFunction  = 0.0;
611     unsigned              ao_iter          = 0;
612     MatrixArray           factors;
613     SparseStatus() = default;
614     SparseStatus(SparseStatus const &) = default;
615     SparseStatus(SparseStatus const &&) = default;
616     SparseStatus(SparseOptions<_TnsSize,ExecutionPolicy_,DefaultValues_> const &opt) : options(opt)
617     { }
618
619     SparseStatus &operator=(SparseStatus const &) = default;
620     SparseStatus &operator=(SparseStatus const &&) = default;
621
622     /*
623     Constructor called, in case the user decides to change one or more
624     of the options and use them in the factorization.
625     */
626     SparseStatus &operator=(SparseOptions<_TnsSize,ExecutionPolicy_,DefaultValues_> const &opts)
627     {
628         options = opts;
629         return *this;
630     }
631
632     explicit operator SparseOptions<_TnsSize,ExecutionPolicy_,DefaultValues_> & ()
633     {
634         return options;
635     }
636 };
637
638 template < typename Tensor_,
639           typename ExecutionPolicy_ = execution::sequenced_policy,
640           template <typename T> class DefaultValues_ = DefaultValues_ >
641 Status<Tensor_> MakeStatus()
642 {
643     Status<Tensor_,ExecutionPolicy_,DefaultValues_> status;
644
645     status.options = MakeOptions<Tensor_,ExecutionPolicy_,DefaultValues_>();
646
647     return status;
648 }
649
650 template <typename Tensor_, typename ExecutionPolicy_, template <typename T> class DefaultValues_>
651 Status<Tensor_,std::remove_cv_t<std::remove_reference_t<ExecutionPolicy_>,DefaultValues_>
652 MakeStatus(DefaultValues_<Tensor_> &&dv, ExecutionPolicy_ &&xp)
653 {
654     using ExecutionPolicy_t = std::remove_cv_t<std::remove_reference_t<ExecutionPolicy_>;
655
656     Status<Tensor_,ExecutionPolicy_t,DefaultValues_> status;
657
658     status.options = MakeOptions<Tensor_>(std::forward<ExecutionPolicy_>(xp));
659
660     return status;
661 }
662
663 template < std::size_t _TnsSize,
664           typename ExecutionPolicy_ = execution::sequenced_policy,
665           template <typename T> class DefaultValues_ = SparseDefaultValues_ >
666 SparseStatus<_TnsSize> MakeSparseStatus()
667 {
668     SparseStatus<_TnsSize,ExecutionPolicy_,DefaultValues_> status;
669
670     status.options = MakeSparseOptions<_TnsSize,ExecutionPolicy_,DefaultValues_>();
671
672     return status;
673 }
674
675 template < std::size_t _TnsSize,
676           typename ExecutionPolicy_,
677           template <typename T> class DefaultValues_ >
678 SparseStatus<_TnsSize,std::remove_cv_t<std::remove_reference_t<ExecutionPolicy_>,DefaultValues_>
679 MakeSparseStatus(DefaultValues_<partensor::SparseTensor<_TnsSize> &&dv, ExecutionPolicy_ &&xp)
680 {
681     SparseStatus<_TnsSize,ExecutionPolicy_,DefaultValues_> status;
682
683     status.options = MakeSparseOptions<_TnsSize,ExecutionPolicy_,DefaultValues_>();
684
685     return status;
686 }
687
688 template <typename Tensor_>
689 struct Options<Tensor_,execution::openmpi_policy,DefaultValues> : public Options<Tensor_>
690 {
691

```



```

693     using Options<Tensor_,execution::sequenced_policy,DefaultValues>::constraints;
694
695     using IntArray = typename TensorTraits<Tensor_>::IntArray;
696
697     IntArray proc_per_mode;
698
699     Options() : proc_per_mode(DefaultValues<Tensor_>::DefaultProcessorsPerMode)
700     { }
701     Options(Options const &) = default;
702     Options(Options &&) = default;
703
704     Options &operator=(Options const &) = default;
705     Options &operator=(Options &&) = default;
706 };
707
708 template <typename Tensor_>
709 Options<Tensor_,execution::openmpi_policy,DefaultValues> MakeOptions(execution::openmpi_policy &&)
710 {
711     Options<Tensor_,execution::openmpi_policy,DefaultValues> options;
712
713     static_cast<Options<Tensor_>&>(options) = MakeOptions<Tensor_>();
714
715     options.proc_per_mode = DefaultValues<Tensor_>::DefaultProcessorsPerMode;
716
717     return options;
718 }
719
720 template <std::size_t _TnsSize>
721 struct SparseOptions<_TnsSize,execution::openmpi_policy,SparseDefaultValues> : public
SparseOptions<_TnsSize>
722 {
723     using SparseOptions<_TnsSize,execution::sequenced_policy,SparseDefaultValues>::constraints;
724
725     using SparseTensor = typename partensor::SparseTensor<_TnsSize>;
726     using IntArray = typename SparseTensorTraits<SparseTensor>::IntArray;
727
728     IntArray proc_per_mode;
729
730     SparseOptions() : proc_per_mode(SparseDefaultValues<SparseTensor>::DefaultProcessorsPerMode)
731     { }
732     SparseOptions(SparseOptions const &) = default;
733     SparseOptions(SparseOptions &&) = default;
734
735     SparseOptions &operator=(SparseOptions const &) = default;
736     SparseOptions &operator=(SparseOptions &&) = default;
737 };
738
739 template <std::size_t _TnsSize>
740 SparseOptions<_TnsSize,execution::openmpi_policy,SparseDefaultValues>
MakeSparseOptions(execution::openmpi_policy &&)
741 {
742     using SparseTensor = typename partensor::SparseTensor<_TnsSize>;
743
744     SparseOptions<_TnsSize,execution::openmpi_policy,SparseDefaultValues> options;
745
746     static_cast<SparseOptions<_TnsSize>&>(options) = MakeSparseOptions<_TnsSize>();
747
748     options.proc_per_mode = SparseDefaultValues<SparseTensor>::DefaultProcessorsPerMode;
749
750     return options;
751 }
752
753 template <typename Tensor_>
754 using MpiOptions = Options<Tensor_,execution::openmpi_policy,DefaultValues>;
755
756 template <typename Tensor_>
757 using MpiStatus = Status<Tensor_,execution::openmpi_policy,DefaultValues>;
758
759 template < std::size_t _TnsSize>
760 using MpiSparseOptions = SparseOptions<_TnsSize,execution::openmpi_policy,SparseDefaultValues>;
761
762 template < std::size_t _TnsSize>
763 using MpiSparseStatus = SparseStatus<_TnsSize,execution::openmpi_policy,SparseDefaultValues>;
764
765 template <typename Tensor_>
766 using OmpOptions = Options<Tensor_,execution::openmp_policy,DefaultValues>;
767
768 template <typename Tensor_>
769 using OmpStatus = Status<Tensor_,execution::openmp_policy,DefaultValues>;
770
771 template <typename Tensor_>
772 using CudaOptions = Options<Tensor_, execution::cuda_policy, DefaultValues>;
773
774 template <typename Tensor_>
775 using CudaStatus = Status<Tensor_, execution::cuda_policy, DefaultValues>;
776
777 template <std::size_t _TnsSize>

```

```

778     using OmpSparseOptions = SparseOptions<_TnsSize,execution::openmp_policy,SparseDefaultValues>;
779
780     template <std::size_t _TnsSize>
781     using OmpSparseStatus = SparseStatus<_TnsSize,execution::openmp_policy,SparseDefaultValues>;
782
783 } // namespace partensor
784
785 namespace ptl = partensor;
786
787 #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 le.](#)

```

1 #ifndef DOXYGEN_SHOULD_SKIP_THIS
15 #endif // DOXYGEN_SHOULD_SKIP_THIS
16 / *****
28 #ifndef PARTENSOR_PARTIAL_CWISE_PROD_HPP
29 #define PARTENSOR_PARTIAL_CWISE_PROD_HPP
30
31 #include "CwiseProd.hpp"
32
33 namespace partensor
34 {
35     inline namespace v1 {
36         template <typename MatrixArray_>
37         auto PartialCwiseProd( MatrixArray_ const &matArray,
38                               std::size_t const mode )
39         {
40             using Matrix = typename MatrixArrayTraits<MatrixArray_>::value_type; // Type of
41             data of @c Eigen Matrix (e.g double, float, ..).
42
43             constexpr std::size_t TnsSize = MatrixArrayTraits<MatrixArray_>::Size; // Size of
44             matArray.
45
46             if constexpr ( TnsSize == 3 )
47             {
48                 switch ( mode )
49                 {
50                     case 0:
51                         return CwiseProd(matArray[2], matArray[1]);
52                         break;
53                     case 1:
54                         return CwiseProd(matArray[2], matArray[0]);
55                         break;
56                     case 2:
57                         return CwiseProd(matArray[1], matArray[0]);
58                         break;
59                     default:
60                         Matrix resMat;
61                         return resMat.setZero();
62                         break;
63                 }
64             }
65             else if constexpr ( TnsSize == 4 )
66             {
67                 switch ( mode )
68                 {
69                     case 0:
70                         return CwiseProd(matArray[3], matArray[2], matArray[1]);
71                         break;
72                     case 1:
73                         return CwiseProd(matArray[3], matArray[2], matArray[0]);
74                         break;
75                     case 2:
76                         return CwiseProd(matArray[3], matArray[1], matArray[0]);
77                         break;
78                     case 3:
79                         return CwiseProd(matArray[2], matArray[1], matArray[0]);
80                         break;
81                     default:
82

```

```

98                                     Matrix resMat;
99                                     return resMat.setZero();
100                                     break;
101                                 }
102                             }
103                             else if constexpr ( TnsSize == 5 )
104                             {
105                                 switch ( mode )
106                                 {
107                                     case 0:
108                                         return CwiseProd(matArray[4], matArray[3], matArray[2],
109 matArray[1]);
110                                         break;
111                                     case 1:
112                                         return CwiseProd(matArray[4], matArray[3], matArray[2],
113 matArray[0]);
114                                         break;
115                                     case 2:
116                                         return CwiseProd(matArray[4], matArray[3], matArray[1],
117 matArray[0]);
118                                         break;
119                                     case 3:
120                                         return CwiseProd(matArray[4], matArray[2], matArray[1],
121 matArray[0]);
122                                         break;
123                                     default:
124                                         Matrix resMat;
125                                         return resMat.setZero();
126                                         break;
127                                 }
128                             }
129                             else if constexpr ( TnsSize == 6 )
130                             {
131                                 switch ( mode )
132                                 {
133                                     case 0:
134                                         return CwiseProd(matArray[5], matArray[4], matArray[3],
135 matArray[2], matArray[1]);
136                                         break;
137                                     case 1:
138                                         return CwiseProd(matArray[5], matArray[4], matArray[3],
139 matArray[2], matArray[0]);
140                                         break;
141                                     case 2:
142                                         return CwiseProd(matArray[5], matArray[4], matArray[3],
143 matArray[1], matArray[0]);
144                                         break;
145                                     case 3:
146                                         return CwiseProd(matArray[5], matArray[4], matArray[2],
147 matArray[1], matArray[0]);
148                                         break;
149                                     case 4:
150                                         return CwiseProd(matArray[5], matArray[3], matArray[2],
151 matArray[1], matArray[0]);
152                                         break;
153                                     case 5:
154                                         return CwiseProd(matArray[4], matArray[3], matArray[2],
155 matArray[1], matArray[0]);
156                                         break;
157                                     default:
158                                         Matrix resMat;
159                                         return resMat.setZero();
160                                         break;
161                                 }
162                             }
163                             else if constexpr ( TnsSize == 7 )
164                             {
165                                 switch ( mode )
166                                 {
167                                     case 0:
168                                         return CwiseProd(matArray[6], matArray[5], matArray[4],
169 matArray[3], matArray[2], matArray[1]);
170                                         break;
171                                     case 1:
172                                         return CwiseProd(matArray[6], matArray[5], matArray[4],
173 matArray[3], matArray[2], matArray[0]);
174                                         break;
175                                     case 2:
176                                         return CwiseProd(matArray[6], matArray[5], matArray[4],
177 matArray[3], matArray[1], matArray[0]);
178                                         break;
179                                     case 3:
180                                         return CwiseProd(matArray[6], matArray[5], matArray[4],

```

```

    matArray[2], matArray[1], matArray[0]);
171                                     break;
172                                     case 4:
173                                     return CwiseProd(matArray[6], matArray[5], matArray[3],
174                                     matArray[2], matArray[1], matArray[0]);
175                                     break;
176                                     case 5:
177                                     return CwiseProd(matArray[6], matArray[4], matArray[3],
178                                     matArray[2], matArray[1], matArray[0]);
179                                     break;
180                                     case 6:
181                                     return CwiseProd(matArray[5], matArray[4], matArray[3],
182                                     matArray[2], matArray[1], matArray[0]);
183                                     break;
184                                     default:
185                                     Matrix resMat;
186                                     return resMat.setZero();
187                                     break;
188                                     }
189                                     }
190                                     else if constexpr ( TnsSize == 8 )
191                                     {
192                                     switch ( mode )
193                                     {
194                                     case 0:
195                                     return CwiseProd(matArray[7], matArray[6], matArray[5],
196                                     matArray[4], matArray[3], matArray[2], matArray[1]);
197                                     break;
198                                     case 1:
199                                     return CwiseProd(matArray[7], matArray[6], matArray[5],
200                                     matArray[4], matArray[3], matArray[2], matArray[0]);
201                                     break;
202                                     case 2:
203                                     return CwiseProd(matArray[7], matArray[6], matArray[5],
204                                     matArray[4], matArray[3], matArray[1], matArray[0]);
205                                     break;
206                                     case 3:
207                                     return CwiseProd(matArray[7], matArray[6], matArray[5],
208                                     matArray[4], matArray[2], matArray[1], matArray[0]);
209                                     break;
210                                     case 4:
211                                     return CwiseProd(matArray[7], matArray[6], matArray[5],
212                                     matArray[3], matArray[2], matArray[1], matArray[0]);
213                                     break;
214                                     case 5:
215                                     return CwiseProd(matArray[7], matArray[6], matArray[4],
216                                     matArray[3], matArray[2], matArray[1], matArray[0]);
217                                     break;
218                                     case 6:
219                                     return CwiseProd(matArray[7], matArray[5], matArray[4],
220                                     matArray[3], matArray[2], matArray[1], matArray[0]);
221                                     break;
222                                     case 7:
223                                     return CwiseProd(matArray[6], matArray[5], matArray[4],
224                                     matArray[3], matArray[2], matArray[1], matArray[0]);
225                                     break;
226                                     default:
227                                     Matrix resMat;
228                                     return resMat.setZero();
229                                     break;
230                                     }
231                                     }
232                                     }
233                                     }
234                                     } // end namespace v1
235                                     } // end namespace partensor
236 } // end namespace PARTENSOR_PARTIAL_CWISE_PROD_HPP
237 #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).
<code>MatrixArray</code>	An array container type.

Parameters

<code>execution_policy</code>	[in] Which Policy to execute.
<code>matArray</code>	[in] An array containing matrices to use in this operation.
<code>mode</code>	[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.](#)

```

1 #ifndef DOXYGEN_SHOULD_SKIP_THIS
15 #endif // DOXYGEN_SHOULD_SKIP_THIS
16 /***** /
28 #ifndef PARTENSOR_PARTIAL_KHATRI_RAO_HPP
29 #define PARTENSOR_PARTIAL_KHATRI_RAO_HPP
30
31 #include "KhatriRao.hpp"
32
33 namespace partensor
34 {
35     inline namespace v1 {
59         template <typename Policy, typename MatrixArray_>
60         auto PartialKhatriRao( Policy const &execution_policy,
61                               MatrixArray_ const &matArray,
62                               std::size_t const mode )
63         {
64             using Matrix = typename MatrixArrayTraits<MatrixArray_>::value_type; // Type of
data of @c Eigen Matrix (e.g double, float, ..).
65
66             constexpr std::size_t TnsSize = MatrixArrayTraits<MatrixArray_>::Size; // Size of
matArray.
67
68             if constexpr ( TnsSize == 3 )
69             {
70                 switch ( mode )
71                 {
72                     case 0:
73                         return KhatriRao(execution_policy, matArray[2],
matArray[1]);
74
75                         case 1:
76                             return KhatriRao(execution_policy, matArray[2],
matArray[0]);
77
78                             case 2:
79                                 return KhatriRao(execution_policy, matArray[1],
matArray[0]);
80
81                                 default:
82                                     Matrix resMat;
83                                     return resMat.setZero();
84                                     break;
85                                 }
86                             }
87                         else if constexpr ( TnsSize == 4 )
88                         {
89                             switch ( mode )
90                             {
91                                 case 0:
92                                     return KhatriRao(execution_policy, matArray[3],
matArray[2], matArray[1]);
93
94                                     case 1:
95                                         return KhatriRao(execution_policy, matArray[3],
matArray[2], matArray[0]);
96
97                                         case 2:
98                                             return KhatriRao(execution_policy, matArray[3],
matArray[1], matArray[0]);
99
100                                             case 3:
101                                                 return KhatriRao(execution_policy, matArray[2],
matArray[1], matArray[0]);
102
103                                                 default:
104                                                     Matrix resMat;
105                                                     return resMat.setZero();
106                                                     break;
107                                                 }
108                             }
109                         else if constexpr ( TnsSize == 5 )
110                         {
111                             switch ( mode )
112                             {
113                                 case 0:
114                                     return KhatriRao(execution_policy, matArray[4],
matArray[3], matArray[2], matArray[1]);
115
116                                     case 1:
117                                         return KhatriRao(execution_policy, matArray[4],
matArray[3], matArray[2], matArray[0]);
118
119                                     case 2:
120                                         return KhatriRao(execution_policy, matArray[4],
matArray[2], matArray[1], matArray[0]);
121
122                                     case 3:
123                                         return KhatriRao(execution_policy, matArray[4],
matArray[2], matArray[1], matArray[0]);
124
125                                     case 4:
126                                         return KhatriRao(execution_policy, matArray[4],
matArray[1], matArray[0], matArray[0]);
127
128                                     default:
129                                         Matrix resMat;
130                                         return resMat.setZero();
131                                         break;
132                                     }
133                             }
134                         }
135                     }
136                 }
137             }
138         }
139     }
140 }

```

```

119                                     case 2:
120     matArray[3], matArray[1], matArray[0];           return KhatriRao(execution_policy, matArray[4],
121                                                     break;
122                                     case 3:
123     matArray[2], matArray[1], matArray[0];           return KhatriRao(execution_policy, matArray[4],
124                                                     break;
125                                     case 4:
126     matArray[2], matArray[1], matArray[0];           return KhatriRao(execution_policy, matArray[3],
127                                                     break;
128                                     default:
129     Matrix resMat;
130     return resMat.setZero();
131     break;
132     }
133     }
134     else if constexpr ( TnsSize == 6 )
135     {
136         switch ( mode )
137         {
138             case 0:
139     matArray[4], matArray[3], matArray[2], matArray[1];   return KhatriRao(execution_policy, matArray[5],
140                                                     break;
141             case 1:
142     matArray[4], matArray[3], matArray[2], matArray[0];   return KhatriRao(execution_policy, matArray[5],
143                                                     break;
144             case 2:
145     matArray[4], matArray[3], matArray[1], matArray[0];   return KhatriRao(execution_policy, matArray[5],
146                                                     break;
147             case 3:
148     matArray[4], matArray[2], matArray[1], matArray[0];   return KhatriRao(execution_policy, matArray[5],
149                                                     break;
150             case 4:
151     matArray[3], matArray[2], matArray[1], matArray[0];   return KhatriRao(execution_policy, matArray[5],
152                                                     break;
153             case 5:
154     matArray[3], matArray[2], matArray[1], matArray[0];   return KhatriRao(execution_policy, matArray[4],
155                                                     break;
156             default:
157     Matrix resMat;
158     return resMat.setZero();
159     break;
160         }
161     }
162     else if constexpr ( TnsSize == 7 )
163     {
164         switch ( mode )
165         {
166             case 0:
167     matArray[5], matArray[4], matArray[3], matArray[2], matArray[1];   return KhatriRao(execution_policy, matArray[6],
168                                                     break;
169             case 1:
170     matArray[5], matArray[4], matArray[3], matArray[2], matArray[0];   return KhatriRao(execution_policy, matArray[6],
171                                                     break;
172             case 2:
173     matArray[5], matArray[4], matArray[3], matArray[1], matArray[0];   return KhatriRao(execution_policy, matArray[6],
174                                                     break;
175             case 3:
176     matArray[5], matArray[4], matArray[2], matArray[1], matArray[0];   return KhatriRao(execution_policy, matArray[6],
177                                                     break;
178             case 4:
179     matArray[5], matArray[3], matArray[2], matArray[1], matArray[0];   return KhatriRao(execution_policy, matArray[6],
180                                                     break;
181             case 5:
182     matArray[4], matArray[3], matArray[2], matArray[1], matArray[0];   return KhatriRao(execution_policy, matArray[6],
183                                                     break;
184             case 6:
185     matArray[4], matArray[3], matArray[2], matArray[1], matArray[0];   return KhatriRao(execution_policy, matArray[5],
186                                                     break;
187             default:
188     Matrix resMat;
189     return resMat.setZero();

```

```

190                                     break;
191                                 }
192                             }
193                             else if constexpr ( TnsSize == 8 )
194                             {
195                                 switch ( mode )
196                                 {
197                                     case 0:
198                                         return KhatriRao(execution_policy, matArray[7],
199 matArray[6], matArray[5], matArray[4], matArray[3], matArray[2], matArray[1]);
200                                         break;
201                                     case 1:
202                                         return KhatriRao(execution_policy, matArray[7],
203 matArray[6], matArray[5], matArray[4], matArray[3], matArray[2], matArray[0]);
204                                         break;
205                                     case 2:
206                                         return KhatriRao(execution_policy, matArray[7],
207 matArray[6], matArray[5], matArray[4], matArray[3], matArray[1], matArray[0]);
208                                         break;
209                                     case 3:
210                                         return KhatriRao(execution_policy, matArray[7],
211 matArray[6], matArray[5], matArray[4], matArray[2], matArray[1], matArray[0]);
212                                         break;
213                                     case 4:
214                                         return KhatriRao(execution_policy, matArray[7],
215 matArray[6], matArray[5], matArray[3], matArray[2], matArray[1], matArray[0]);
216                                         break;
217                                     case 5:
218                                         return KhatriRao(execution_policy, matArray[7],
219 matArray[6], matArray[4], matArray[3], matArray[2], matArray[1], matArray[0]);
220                                         break;
221                                     case 6:
222                                         return KhatriRao(execution_policy, matArray[7],
223 matArray[5], matArray[4], matArray[3], matArray[2], matArray[1], matArray[0]);
224                                         break;
225                                     case 7:
226                                         return KhatriRao(execution_policy, matArray[6],
227 matArray[5], matArray[4], matArray[3], matArray[2], matArray[1], matArray[0]);
228                                         break;
229                                     default:
230                                         Matrix resMat;
231                                         return resMat.setZero();
232                                         break;
233                                 }
234                             }
235                         }
236                     }
237
238     template <typename MatrixArray_>
239     auto PartialKhatriRao(MatrixArray_ const &matArray, std::size_t const mode)
240     {
241         return PartialKhatriRao(execution::seq, matArray, mode);
242     }
243 } // end namespace v1
244 } // end namespace partensor
245 #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 & lName, 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

leName	[in] Specify the path to the le, 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 le 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 les, in a speci c way. In each le, a single row is stored, from the 3D data and the pattern of the le names are like : leName_00000, leName_00001, etc. Then use readFMRI_matrix to read from all les and store the data in a Matrix , equivalent to the third order Tensor matricization in rst mode.

Template Parameters

FileName	Type of input le, either std::string or char .
Dimensions	Array type for Tensor dimensions.

Parameters

leName	[in] Specify the path to the le, 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 le.

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 : `leName_00000`, `leName_00001`, etc. Then use `readFMRIFromFiles` 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 <code>dat</code> . Also, the size of the <code>std::array</code> <code>tnsDims</code> , <code>local_tnsDims</code> and <code>local_skip</code> .
FileName	Input file type. Either <code>std::string</code> or <code>char</code> .

Parameters

leName	[in] Specify the path to the file, where the data are located. Variable can be of <code>FileName</code> type.
tnsDims	[in] <code>std::array</code> with initial lengths for each Tensor dimension.
local_tnsDims	[in] <code>std::array</code> with lengths for each Tensor dimension per processor.
local_skip	[in] <code>std::array</code> 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 : `leName_00000`, `leName_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 le type. Either std::string or char .

Parameters

leName	[in] Specify the path to the le, where the data are located. Variable can be of FileName type.
skip	[in] Number of elements to skip.
dat	[in,out] Data read from le.

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 le. 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 le, either std::string or char .

Parameters

leName	[in] Specify the path to the le, 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 le.

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 `std::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 [le](#).

```

1  #ifndef DOXYGEN_SHOULD_SKIP_THIS
15 #endif // DOXYGEN_SHOULD_SKIP_THIS
16 / ..... /
26 #ifndef PARTENSOR_READ_WRITE_HPP
27 #define PARTENSOR_READ_WRITE_HPP
28
29 #include "PARTENSOR_basic.hpp"
30 #include "boost/interprocess/file_mapping.hpp"
31 #include "boost/interprocess/mapped_region.hpp"
32
33 #include <iostream>
34 #include <fstream>
35 // Open Function Sys Call
36 #include <sys/types.h>
37 #include <sys/stat.h>
38 #include <fcntl.h>
39 // Read-Write Functions Sys Call
40 #include <unistd.h>
41
42 namespace partensor {
43
44     #ifndef DOXYGEN_SHOULD_SKIP_THIS
45     template<typename T>
46     inline constexpr bool is_c_str = std::is_same<char const          *, typename std::decay<T>::type>::value
47     ||
48     std::is_same<char const          *, typename
49     std::decay<T>::type>::value;
50     #endif // end of DOXYGEN_SHOULD_SKIP_THIS
51
52
53     // =====
54     // ===== WRITE FUNCTIONS =====
55     // =====
56
57     template<typename FileName>
58     void writeToFile_append( double const fvalue,
59                             FileName const &fileName )
60     {
61         std::ofstream os;
62         os.exceptions(std::ofstream::badbit | std::ofstream::failbit);
63         try
64         {
65             os.open(fileName, std::ios::app);
66             os << fvalue << " nn";
67             os.close();
68         }
69         catch (std::ofstream::failure const &ex)
70         {
71             std::cerr << "Exception opening/writing/closing file: " << fileName << std::endl;
72         }
73     }
74
75     #ifndef DOXYGEN_SHOULD_SKIP_THIS
76     template<typename Data, typename FileName>
77     void writeToFile_cppStream( Data const &dat,
78                                FileName const &fileName,
79                                std::size_t const size)
80     {
81         std::ofstream os;
82         os.exceptions(std::ofstream::badbit | std::ofstream::failbit);
83         try
84         {
85             os.open(fileName, std::ios::binary | std::ios::trunc);
86
87             std::size_t count = 0;
88             if constexpr (is_matrix<Data>)
89             {
90                 using DataType = typename MatrixTraits<Data>::DataType;
91                 Matrix dat_T;
92                 dat_T = dat.transpose();
93                 count = size * sizeof(DataType);
94                 os.write(reinterpret_cast<char*>(dat_T.data()), count);
95             }
96             else
97             {
98                 using DataType = typename TensorTraits<Data>::DataType;
99                 count = size * sizeof(DataType);
100                os.write((char *)dat.data(), count);
101            }
102            os.close();
103        }
104        catch (std::ofstream::failure const &ex)
105        {
106
107        }
108    }
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128

```

```

129         {
130             std::cerr << "Exception opening/writing/closing file: " << fileName << std::endl;
131         }
132     }
133     #endif // end of DOXYGEN_SHOULD_SKIP_THIS
134
135     #ifndef DOXYGEN_SHOULD_SKIP_THIS
151     template<typename Data, typename FileName>
152     void writeTo_cStream( Data const &dat,
153                         FileName const &fileName,
154                         std::size_t const size )
155     {
156         try
157         {
158             FILE *of;
159             if constexpr (! is_c_str<FileName>)
160             {
161                 of = fopen(fileName.c_str(), "wb");
162             }
163             else
164             {
165                 of = fopen(fileName, "wb");
166             }
167
168             std::size_t fw = 0;
169             if constexpr (is_matrix<Data>)
170             {
171                 using DataType = typename MatrixTraits<Data>::DataType;
172                 Matrix dat_T;
173                 dat_T = dat.transpose();
174                 fw = fwrite(dat_T.data(), sizeof(DataType), size, of);
175             }
176             else
177             {
178                 using DataType = typename TensorTraits<Data>::DataType;
179                 fw = fwrite(dat.data(), sizeof(DataType), size, of);
180             }
181             if(fw != size)
182                 throw 0;
183
184             fclose(of);
185         }
186         catch(...)
187         {
188             std::cerr << "Exception opening/writing/closing file: " << fileName << std::endl;
189         }
190     }
191     #endif // end of DOXYGEN_SHOULD_SKIP_THIS
192
193     #ifndef DOXYGEN_SHOULD_SKIP_THIS
209     template<typename Data, typename FileName>
210     void writeTo_sysCalls( Data const &dat,
211                           FileName const &fileName,
212                           std::size_t const size )
213     {
214         // equal to open() with flags = O_CREAT|O_WRONLY|O_TRUNC
215         // S_IRWXU - user (file owner) has write only permission
216         try
217         {
218             int fileDescriptor;
219             if constexpr (! is_c_str<FileName>)
220             {
221                 fileDescriptor = creat(fileName.c_str(), S_IWUSR);
222             }
223             else
224             {
225                 fileDescriptor = creat(fileName, S_IWUSR);
226             }
227             if(fileDescriptor > 0)
228             {
229                 std::size_t count = 0;
230                 ssize_t writeToFile;
231                 if constexpr (is_matrix<Data>)
232                 {
233                     using DataType = typename MatrixTraits<Data>::DataType;
234                     Matrix dat_T;
235                     dat_T = dat.transpose();
236                     count = size * sizeof(DataType);
237                     writeToFile = write(fileDescriptor, dat_T.data(), count);
238                 }
239                 else
240                 {
241                     using DataType = typename TensorTraits<Data>::DataType;
242                     count = size * sizeof(DataType);
243                     writeToFile = write(fileDescriptor, dat.data(), count);
244                 }
245                 if (writeToFile <= 0)

```

```

246         {
247             throw -1;
248         }
249     }
250     if(close(fileDescriptor) < 0)
251     {
252         throw 0;
253     }
254 }
255 catch(...)
256 {
257     std::cerr << "Exception opening/writing/closing file: " << fileName << std::endl;
258 }
259 }
260 #endif // end of DOXYGEN_SHOULD_SKIP_THIS
261
262 template<typename Data, typename FileName>
263 void write( Data const &dat,
264            FileName const &fileName,
265            std::size_t const size )
266 {
267     writeToCppStream(dat, fileName, size);
268 }
269
270 // ===== READ FUNCTIONS =====
271 // =====
272
273 #ifndef DOXYGEN_SHOULD_SKIP_THIS
274 template<typename Data, typename FileName>
275 void readFromFile_cppStream( FileName const &fileName,
276                             std::size_t const size,
277                             std::size_t const skip,
278                             Data &dat )
279 {
280     std::ifstream is;
281     is.exceptions(std::ifstream::badbit | std::ifstream::failbit);
282     try
283     {
284         is.open(fileName, std::ios::binary);
285         std::size_t count = 0;
286         if constexpr (is_matrix<Data>)
287         {
288             using DataType = typename MatrixTraits<Data>::DataType;
289             Matrix dat_T(dat.cols(), dat.rows());
290             count = size * sizeof(DataType);
291             is.ignore(skip * sizeof(DataType)); // where to start
292             is.read(reinterpret_cast<char*>(dat_T.data()), count);
293             dat = dat_T.transpose();
294         }
295         else
296         {
297             using DataType = typename TensorTraits<Data>::DataType;
298             count = size * sizeof(DataType);
299             is.ignore(skip * sizeof(DataType)); // where to start
300             is.read(reinterpret_cast<char*>(dat.data()), count);
301         }
302         is.close();
303     }
304     catch (std::ifstream::failure const &ex)
305     {
306         std::cerr << "Exception opening/reading/closing file: " << fileName << std::endl;
307     }
308 }
309 #endif // end of DOXYGEN_SHOULD_SKIP_THIS
310
311 #ifndef DOXYGEN_SHOULD_SKIP_THIS
312 template<typename Data, typename FileName>
313 void readFromFile_cStream( FileName const &fileName,
314                           std::size_t const size,
315                           std::size_t const skip,
316                           Data &dat )
317 {
318     try
319     {
320         FILE *inputStream;
321         if constexpr (! is_c_str<FileName>)
322         {
323             inputStream = fopen(fileName.c_str(), "rb");
324         }
325         else
326         {
327             inputStream = fopen(fileName, "rb");
328         }
329         std::size_t fr = 0;
330         if constexpr (is_matrix<Data>)
331         {

```

```

382         using DataType = typename MatrixTraits<Data>::DataType;
383         fseek(inputStream, skip * sizeof(DataType), SEEK_SET); // where to start
384
385         Matrix dat_T(dat.cols(), dat.rows());
386         fr = fread(dat_T.data(), sizeof(DataType), size, inputStream);
387         dat = dat_T.transpose();
388     }
389     else
390     {
391         using DataType = typename TensorTraits<Data>::DataType;
392         fseek(inputStream, skip * sizeof(DataType), SEEK_SET); // where to start
393         fr = fread(dat.data(), sizeof(DataType), size, inputStream);
394     }
395     if(fr!=size)
396     {
397         throw 0;
398     }
399     fclose(inputStream);
400 }
401 catch(...)
402 {
403     std::cerr << "Exception opening/writing/closing file: " << fileName << std::endl;
404 }
405 }
406 #endif // end of DOXYGEN_SHOULD_SKIP_THIS
407
408 #ifndef DOXYGEN_SHOULD_SKIP_THIS
426 template<typename Data, typename FileName>
427 void readFromFile_sysCalls( FileName const &fileName,
428                             std::size_t const size,
429                             off_t const skip,
430                             Data &dat )
431 {
432     // S_IRWXU - user (file owner) has read only permission
433     try
434     {
435         int fileDescriptor;
436         if constexpr (! is_c_str<FileName>)
437         {
438             fileDescriptor = open(fileName.c_str(), S_IRUSR);
439         }
440         else
441         {
442             fileDescriptor = open(fileName, S_IRUSR);
443         }
444         if(fileDescriptor > 0)
445         {
446             std::size_t count = 0;
447             off_t seek = 0;
448             ssize_t readFromFile;
449             if constexpr (is_matrix<Data>)
450             {
451                 using DataType = typename MatrixTraits<Data>::DataType;
452                 count = size * sizeof(DataType);
453                 seek = lseek(fileDescriptor,
454                             skip * sizeof(DataType), SEEK_SET);
455
456                 Matrix dat_T(dat.cols(), dat.rows());
457                 readFromFile = read(fileDescriptor, dat_T.data(), count);
458                 dat = dat_T.transpose();
459             }
460             else
461             {
462                 using DataType = typename TensorTraits<Data>::DataType;
463                 count = size * sizeof(DataType);
464                 seek = lseek(fileDescriptor,
465                             skip * sizeof(DataType), SEEK_SET);
466                 readFromFile = read(fileDescriptor, dat.data(), count);
467             }
468             if(seek == -1 || readFromFile <= 0)
469             {
470                 throw -1;
471             }
472             if(close(fileDescriptor) < 0)
473             {
474                 throw 0;
475             }
476         }
477     }
478     catch(...)
479     {
480         std::cerr << "Exception opening/reading/closing file: " << fileName << std::endl;
481     }
482 #endif // end of DOXYGEN_SHOULD_SKIP_THIS
483

```



```

484     #ifndef DOXYGEN_SHOULD_SKIP_THIS
489     template<typename Data, typename FileName>
490     void readFromFile_memMap( FileName const &fileName,
491                             Data          &dat          )
492     {
493         try
494         {
495             boost::interprocess::mode_t      mode = boost::interprocess::read_only;
496             boost::interprocess::file_mapping fm;
497             if constexpr (is_c_str<FileName>)
498             {
499                 fm = boost::interprocess::file_mapping(fileName, mode);
500             }
501             else
502             {
503                 fm = boost::interprocess::file_mapping(fileName.c_str(), mode);
504             }
505             boost::interprocess::mapped_region region(fm, mode, 0, 0);
506             if constexpr (is_matrix<Data>)
507             {
508                 using DataType = typename MatrixTraits<Data>::DataType;
509                 DataType * addr = static_cast<DataType * >(region.get_address());
510
511                 Matrix dat_T(dat.cols(), dat.rows());
512                 dat_T      = Eigen::Map<Data>(addr, dat.cols(), dat.rows());
513                 dat        = dat_T.transpose();
514             }
515             else
516             {
517                 using DataType
518                 = typename TensorTraits<Data>::DataType;
519                 using Dimensions
520                 = typename TensorTraits<Data>::Dimensions;
521                 const Dimensions tnsDims = dat.dimensions();
522                 DataType * addr
523                 = static_cast<DataType * >(region.get_address());
524                 dat
525                 = Eigen::TensorMap<Data>(addr,
526                                         tnsDims);
527             }
528         }
529         catch(boost::interprocess::interprocess_exception &ex)
530         {
531             std::cerr << ex.what() << std::endl;
532         }
533         catch(...)
534         {
535             std::cerr << "Unknown Exception" << std::endl;
536         }
537     }
538     #endif // end of DOXYGEN_SHOULD_SKIP_THIS
539
540     template<typename Data, typename FileName>
541     void read( FileName const &fileName,
542              std::size_t const size,
543              std::size_t const skip,
544              Data          &dat          )
545     {
546         readFromFile_cppStream(fileName, size, skip, dat);
547     }
548
549     template<typename FileName, typename Dimensions>
550     void readfMRI_matrix( FileName const &fileName,
551                          Dimensions const tnsDims,
552                          std::size_t const skip,
553                          Matrix          &dat          )
554     {
555         std::string prefix      = fileName.substr(0, fileName.size()-4);
556         std::string extension  = fileName.substr(fileName.size()-4, 50);
557         std::string newFileName;
558         Matrix      _temp(tnsDims[0], 1);
559
560         for(auto fn=0; fn<tnsDims[1] * tnsDims[2]; fn++)
561         {
562             newFileName = prefix + "_";
563             newFileName += std::to_string(fn/10000) + std::to_string(fn/1000) +
564             std::to_string(fn/100);
565             newFileName += std::to_string(fn/10) + std::to_string(fn%10) + extension;
566             readFromFile_cppStream(newFileName, tnsDims[0], skip, _temp);
567             dat.col(fn) = _temp;
568         }
569     }
570
571     template<typename Data, typename FileName>
572     void readfMRI_tensor( FileName const &fileName,
573                          std::size_t const skip,
574                          Data          &dat          )

```

```

638     {
639         using Dimensions = typename TensorTraits<Data>::Dimensions;
640
641         const Dimensions&      tnsDims      = dat.dimensions();
642         std::string            prefix        =
643         fileName.substr(0,fileName.size()-4);
644         std::string            extension     =
645         fileName.substr(fileName.size()-4,50);
646         std::string            newFileName;
647         int                    fn           = 0;
648         Tensor<1>              _temp(tnsDims[0]);
649         for(int i=0; i<tnsDims[2]; i++)
650         {
651             for(int j=0; j<tnsDims[1]; j++, fn++)
652             {
653                 newFileName = prefix + "_";
654                 newFileName += std::to_string(fn/10000) + std::to_string(fn/1000) +
655                 std::to_string(fn/100);
656                 newFileName += std::to_string(fn/10) + std::to_string(fn%10) +
657                 extension;
658                 readFromFile_cppStream(newFileName, tnsDims[0], skip, _temp);
659                 dat.chip(i,2).chip(j,1) = _temp;
660             }
661         }
662         // =====
663         // ===== READ FUNCTIONS FOR MPI =====
664         // =====
665
666         #ifndef DOXYGEN_SHOULD_SKIP_THIS
667         template<std::size_t TnsSize>
668         void offset_calculation( std::array<int, TnsSize> const &dimensions,
669                                std::array<int, TnsSize> const &skip,
670                                std::array<int, TnsSize> &offset
671         )
672         {
673             for (int i=TnsSize-1; i>1; i--)
674             {
675                 // How many cubes to skip
676                 offset[i] = std::accumulate(dimensions.begin(), dimensions.end()-(TnsSize-i), 1,
677                 std::multiplies<int>())
678                 * skip[i];
679             }
680             offset[1] = dimensions[0] * skip[1]; // how many matrices to skip
681             offset[0] = skip[0]; // how many rows to skip
682         }
683         #endif // end of DOXYGEN_SHOULD_SKIP_THIS
684
685         #ifndef DOXYGEN_SHOULD_SKIP_THIS
686         template<std::size_t idx, std::size_t TnsSize>
687         void readTensor_util( std::array<int, TnsSize> const &tnsDims,
688                             std::array<int, TnsSize> const &local_tnsDims,
689                             std::array<int, TnsSize> const &offset,
690                             std::ifstream &is,
691                             Tensor<static_cast<int>(TnsSize)> &dat
692         )
693         {
694             using DT = typename TensorTraits<Tensor<static_cast<int>(TnsSize)>::DataType;
695             if constexpr (idx>0)
696             {
697                 // e.g. I * J * K in 4D tensor
698                 int dim_product = std::accumulate(tnsDims.begin(), tnsDims.begin()+idx, 1,
699                 std::multiplies<int>());
700
701                 // ignore
702                 is.ignore(offset[idx] * sizeof(DT));
703
704                 for (int i=0; i<local_tnsDims[idx]; i++)
705                 {
706                     readTensor_util<idx-1, TnsSize>(tnsDims, local_tnsDims, offset, is,
707                     dat);
708                 }
709                 is.ignore( dim_product * (tnsDims[idx] - local_tnsDims[idx] -
710                 offset[idx]) * sizeof(DT) );
711             }
712             else
713             {
714                 Tensor<1> tns_col(local_tnsDims[0]);
715                 // ignore offset[0] rows
716                 is.ignore(offset[0] * sizeof(DT));
717                 is.read(reinterpret_cast<char *>(tns_col.data()), local_tnsDims[0] * sizeof(DT));
718                 // use tensor indexing for reading from file
719                 for (int i=0; i<local_tnsDims[0]; i++)
720                 {
721                     static int idx_read = 0;
722                     dat(idx_read) = tns_col(i);
723                 }
724             }
725         }
726     }

```

```

746             idx_read++;
747         }
748         // ignore remaining column elements
749         is.ignore( (tnsDims[0] - local_tnsDims[0] - offset[0]) * sizeof(DT) );
750     }
751 }
752 #endif // end of DOXYGEN_SHOULD_SKIP_THIS
753
754 template<std::size_t TnsSize, typename FileName>
755 void readTensor( FileName fileName, const &fileName,
756                std::array<int, TnsSize> const &tnsDims,
757                std::array<int, TnsSize> const &local_tnsDims,
758                std::array<int, TnsSize> const &local_skip,
759                Tensor<static_cast<int>(TnsSize)> &dat )
760 {
761     std::ifstream is;
762     try
763     {
764         is.open(fileName, std::ios::binary);
765         std::array<int, TnsSize> offset;
766         offset_calculation( tnsDims, local_skip, offset );
767         readTensor_util<TnsSize-1, TnsSize>( tnsDims, local_tnsDims, offset, is, dat );
768     }
769     catch (std::ofstream::failure const &ex)
770     {
771         std::cerr << "Exception opening/reading/closing file: " << fileName << std::endl;
772     }
773 }
774
775 template<std::size_t TnsSize, typename FileName>
776 void readFMRI_mpi( FileName fileName, const &fileName,
777                  std::array<int, TnsSize> const &tnsDims,
778                  std::array<int, TnsSize> const &local_tnsDims,
779                  std::array<int, TnsSize> const &local_skip,
780                  Tensor<static_cast<int>(TnsSize)> &dat )
781 {
782     std::string prefix = fileName.substr(0, fileName.size()-4);
783     std::string extension = fileName.substr(fileName.size()-4, 5);
784     std::string newFileName;
785     int fn = local_skip[2] * tnsDims[1];
786     Tensor<1> _temp(local_tnsDims[0]);
787     for(int i=0; i<local_tnsDims[2]; i++)
788     {
789         fn += local_skip[1];
790         for(int j=0; j<local_tnsDims[1]; j++)
791         {
792             newFileName = prefix + "_";
793             newFileName += std::to_string(fn/10000) + std::to_string(fn/1000) +
794             std::to_string(fn/100);
795             newFileName += std::to_string(fn/10) + std::to_string(fn%10) +
796             extension;
797             readFromFile_cppStream(newFileName.c_str(), local_tnsDims[0],
798             local_skip[0], _temp);
799             dat.chip(i,2).chip(j,1) = _temp;
800             fn++;
801         }
802         fn += tnsDims[1] - local_tnsDims[1] - local_skip[1];
803     }
804 }
805 }
806 // end namespace partensor
807 #endif // end of PARTENSOR_READ_WRITE_HPP

```

8.59 temp.hpp

```

1 template<std::size_t _TnsSize>
2 double ComputeCostFun(SparseTensor const &sparse_tns,
3                       std::array<Matrix, _TnsSize> const &factors,
4                       std::array<int, _TnsSize> const &tns_dimensions)
5 {
6     int const rank = factors[0].cols();
7     Matrix temp_1_R(1, rank);
8     double temp_1_1 = 0;
9     double f_value = 0;
10    std::array<int, _TnsSize> offsets[tns_size - 1];
11    offsets[0] = 1;
12
13    for (int j = 1; j < static_cast<int>(_TnsSize) - 1; j++)

```

```

14     {
15         offsets[j] = offsets[j - 1]          * factors[j - 1].rows();
16     }
17
18     for (long int i = 0; i < tns_sparse.outerSize(); ++i)
19     {
20         int row;
21         for (SparseTensor::InnerIterator it(tns_sparse, i); it; ++it)
22         {
23             temp_1_R = factors[tns_size - 1].row(it.col());
24             // Select rows of each factor an compute the Hadamard product of the respective row of the
25             Khatri-Rao product, and the row of factor A_N.
26             for (int mode_i = static_cast<int>(_TnsSize) - 2; mode_i >= 0; mode_i--)
27             {
28                 row = ((it.row()) / offsets[mode_i]) % (tns_dimensions[mode_i]);
29                 temp_1_R.noalias() = temp_1_R.cwiseProduct(factors[mode_i].row(row));
30             }
31             temp_1_1 = it.value() - temp_1_R.sum();
32             f_value += temp_1_1 * temp_1_1;
33         }
34     }
35     return f_value;
36 }
37
38
39
40
41
42 template<std::size_t _TnsSize>
43 void UnconUpdate(std::array<int, _TnsSize> const &tns_dimensions,
44                 SparseMatrix const &sparse_tns,
45                 double const lambda,
46                 int const cur_mode,
47                 std::array<Matrix, _TnsSize> &factors)
48 {
49     // int m = factors[cur_mode].rows();
50     int r = factors[cur_mode].cols();
51
52     Matrix eye = lambda * Matrix::Identity(r, r);
53
54     std::array<int, _TnsSize> offsets[_TnsSize - 1];
55     offsets[0] = 1;
56     for (int j = 1, mode = 0; j < static_cast<int>(_TnsSize) - 1; j++, mode++)
57     {
58         if (cur_mode == mode)
59         {
60             mode++;
61         }
62         offsets[j] = offsets[j - 1]          * ((factors[mode]).rows());
63     }
64
65     int last_mode = (cur_mode == static_cast<int>(_TnsSize) - 1) ? static_cast<int>(_TnsSize) - 2 :
66     static_cast<int>(_TnsSize) - 1;
67
68     Matrix MTTKRP_row(1, r);
69     Matrix temp_RxR(r, r);
70     Matrix temp_1_R(1, r);
71
72     // Compute MTTKRP
73     for (long int i = 0; i < sparse_tns.outerSize(); ++i)
74     {
75         MTTKRP_row.setZero();
76         temp_RxR.setZero(); // is the Hadamard product of Grammians of the Factors, that correspond to
77         the nnz elements of the Tensor.
78         for (SparseTensor::InnerIterator it(sparse_tns, i); it; ++it)
79         {
80             temp_1_R = Matrix::Ones(1, r);
81             int row;
82             // Select rows of each factor an compute the respective row of the Khatri-Rao product.
83             for (int mode_i = last_mode, kr_counter = static_cast<int>(_TnsSize) - 2; mode_i >= 0 &&
84             kr_counter >= 0; mode_i--)
85             {
86                 if (mode_i == cur_mode)
87                 {
88                     continue;
89                 }
90                 row = ((it.row()) / offsets[kr_counter]) % (tns_dimensions[mode_i]);
91                 temp_1_R = temp_1_R.cwiseProduct(factors[mode_i].row(row));
92                 kr_counter--;
93             }
94             // Subtract from the previous row the respective row of W, according to relation (9).
95             MTTKRP_row.noalias() += it.value() * temp_1_R;
96             temp_RxR.noalias() += temp_1_R.transpose() * temp_1_R;
97             // std::cerr << "ERROR!!! proc_ID = " << my_rank << " " << nt << i << " " << nt << it.col() << " " << nn";
98         }
99         factors[cur_mode].row(i) = MTTKRP_row * (temp_RxR + eye).inverse();
100     }
101 }

```

```

100 namespace v2 // transposed_v
101 {
102     template<std::size_t _TnsSize>
103     void UnconUpdate(std::array<int, _TnsSize> const &tns_dimensions,
104                    SparseMatrix const &sparse_tns,
105                    double const lambda,
106                    int const cur_mode,
107                    std::array<Matrix, _TnsSize> &factors)
108     {
109         // int m = factors[cur_mode].rows();
110         int r = factors[cur_mode].rows();
111
112         Matrix eye = lambda * Matrix::Identity(r, r);
113
114         std::array<int, _TnsSize> offsets[_TnsSize - 1];
115         offsets[0] = 1;
116         for (int j = 1, mode = 0; j < static_cast<int>(_TnsSize) - 1; j++, mode++)
117         {
118             if (cur_mode == mode)
119             {
120                 mode++;
121             }
122             offsets[j] = offsets[j - 1] * ((factors[mode]).cols());
123         }
124
125         int last_mode = (cur_mode == static_cast<int>(_TnsSize) - 1) ? static_cast<int>(_TnsSize) - 2 :
static_cast<int>(_TnsSize) - 1;
126
127         Matrix MTTKRP_col(r, 1);
128         Matrix temp_RxR(r, r);
129         Matrix temp_R_1(r, 1);
130
131         // Compute MTTKRP
132         for (long int i = 0; i < sparse_tns.outerSize(); ++i)
133         {
134             MTTKRP_col.setZero();
135             temp_RxR.setZero(); // is the Hadamard product of Grammians of the Factors, that correspond
to the nnz elements of the Tensor.
136             for (SparseTensor::InnerIterator it(sparse_tns, i); it; ++it)
137             {
138                 temp_R_1 = Matrix::Ones(r, 1);
139                 int row;
140                 // Select rows of each factor an compute the respective row of the Khatri-Rao product.
141                 for (int mode_i = last_mode, kr_counter = static_cast<int>(_TnsSize) - 2; mode_i >= 0 &&
kr_counter >= 0; mode_i--)
142                 {
143                     if (mode_i == cur_mode)
144                     {
145                         continue;
146                     }
147                     row = ((it.row()) / offsets[kr_counter]) % (tns_dimensions[mode_i]);
148                     temp_R_1 = temp_R_1.cwiseProduct(factors[mode_i].col(row));
149                     kr_counter--;
150                 }
151                 // Subtract from the previous row the respective row of W, according to relation (9).
152                 MTTKRP_col.noalias() += it.value() * temp_R_1;
153                 temp_RxR.noalias() += temp_R_1.transpose() * temp_R_1;
154                 // std::cerr << "ERROR!!! proc_ID = " << my_rank << " " << nt << i << " " << nt << it.col() << " " << nn";
155             }
156             factors[cur_mode].col(i) = ((temp_RxR + eye).inverse()) * MTTKRP_col;
157         }
158     }
159
160     template<std::size_t _TnsSize>
161     double ComputeCostFun(SparseTensor const &sparse_tns,
162                          std::array<Matrix, _TnsSize> const &factors,
163                          std::array<int, _TnsSize> const &tns_dimensions)
164     {
165         int const rank = factors[0].rows();
166         Matrix temp_R_1(rank, 1);
167         double temp_1_1 = 0;
168         double f_value = 0;
169         std::array<int, _TnsSize> offsets[tns_size - 1];
170
171         offsets[0] = 1;
172         for (int j = 1; j < static_cast<int>(_TnsSize) - 1; j++)
173         {
174             offsets[j] = offsets[j - 1] * factors[j - 1].cols();
175         }
176
177         for (long int i = 0; i < tns_sparse.outerSize(); ++i)
178         {
179             int row;
180             for (SparseTensor::InnerIterator it(tns_sparse, i); it; ++it)
181             {
182                 temp_R_1 = factors[tns_size - 1].col(it.col());
183                 // 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.
184     for (int mode_i = static_cast<int>(_TnsSize) - 2; mode_i >= 0; mode_i--)
185     {
186         row = ((it.row()) / offsets[mode_i]) % (tns_dimensions[mode_i]);
187         temp_R_1.noalias() = temp_R_1.cwiseProduct(factors[mode_i].col(row));
188     }
189     temp_1_1 = it.value() - temp_R_1.sum();
190     f_value += temp_1_1 * temp_1_1;
191 }
192 }
193
194     return f_value;
195 }
196 } // 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

Contains 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

<code>_TnsSize</code>	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.](#)

```

1 #ifndef DOXYGEN_SHOULD_SKIP_THIS
15 #endif // DOXYGEN_SHOULD_SKIP_THIS
16 / *****/
25 #ifndef PARTENSOR_TYPES_HPP
26 #define PARTENSOR_TYPES_HPP
27
28 // #include "PARTENSOR_basic.hpp"
29 // #include "Constants.hpp"
30
31 namespace partensor {
32
36 using DefaultDataType = double;
37
42 using Matrix = Eigen::Matrix<DefaultDataType, Eigen::Dynamic, Eigen::Dynamic, Eigen::ColMajor>;
43
49 using LongMatrix = Eigen::Matrix<long int, Eigen::Dynamic, Eigen::Dynamic, Eigen::ColMajor>;
50
55 using SparseMatrix = Eigen::SparseMatrix<DefaultDataType, Eigen::ColMajor, long int>;
56
63 template<int _TnsSize>
64 using Tensor = Eigen::Tensor<DefaultDataType, _TnsSize>;
65
72 template <typename Tensor>
73 struct TensorTraits;
74
80 template <int _TnsSize>
81 struct TensorTraits<Tensor<_TnsSize>>
82 {
83     using DataType = DefaultDataType;
84     using MatrixType = partensor::Matrix;
85     using Dimensions = typename Tensor<_TnsSize>::Dimensions;
86     using Constraints = std::array<Constraint,_TnsSize>;
87     using MatrixArray = std::array<MatrixType,_TnsSize>;
88     using DoubleArray = std::array<double,_TnsSize>;
89     using IntArray = std::array<int,_TnsSize>;
91     static constexpr std::size_t TnsSize = _TnsSize;
92 };
93
94 template <typename Tensor>
95 using MatrixArray = typename TensorTraits<Tensor>::MatrixArray;
96
97 template <typename Tensor>
98 using Constraints = typename TensorTraits<Tensor>::Constraints;
99
100 template <typename Tensor>
101 using DoubleArray = typename TensorTraits<Tensor>::DoubleArray;
102
109 template <typename SparseTensor>
110 struct SparseTensorTraits;
111
112 template <std::size_t _TnsSize>
113 using SparseTensor = std::array<SparseMatrix, _TnsSize>;
114
120 template <std::size_t _TnsSize>
121 struct SparseTensorTraits<SparseTensor<_TnsSize>> // std::array<SparseMatrix, _TnsSize>
122 {
123     using DataType = DefaultDataType;
124     using MatrixType = partensor::Matrix;
125     using SparseMatrixType = partensor::SparseMatrix;
126     using LongMatrixType = partensor::LongMatrix;
127     using Dimensions = std::array<int,_TnsSize>;
128     using Constraints = std::array<Constraint,_TnsSize>;
129     using SparseTensor = std::array<SparseMatrix, _TnsSize>;
130     using MatrixArray = std::array<MatrixType,_TnsSize>;
131     using DoubleArray = std::array<double,_TnsSize>;

```



```

132     using IntArray          = std::array<int,_TnsSize>;
133     static constexpr std::size_t TnsSize = _TnsSize;
134 };
135
136
137
138
139
140
141
142
143 template <typename Matrix>
144 struct MatrixTraits;
145
146
147
148
149 template <>
150 // struct MatrixTraits<Eigen::Matrix<DefaultDataType, Eigen::Dynamic, Eigen::Dynamic, Eigen::ColMajor>
151 struct MatrixTraits<Matrix>
152 {
153     using DataType      = DefaultDataType;
154     using MatrixType    = partensor::Matrix;
155 };
156
157
158
159
160
161
162
163 template <typename MA>
164 struct MatrixArrayTraits
165 {
166     using value_type = typename MA::value_type;
167     using array_type = MA;
168
169     static constexpr unsigned Size = std::tuple_size<MA>::value;
170 };
171
172
173
174
175
176
177
178
179 template <typename T, std::size_t _Size>
180 struct MatrixArrayTraits<std::array<T,_Size>
181 {
182     using value_type = T;
183     using array_type = std::array<T,_Size>;
184
185     static constexpr std::size_t Size = _Size;
186 };
187
188 // template<typename MatrixArray>
189 // using MatrixArrayType = typename MatrixArrayTraits<MatrixArray>::value_type;
190
191
192
193
194
195
196
197
198     template<typename T>
199     inline constexpr bool is_matrix = std::is_same<Matrix, typename std::decay<T>::type>::value;
200
201
202
203
204
205
206
207
208
209 template<typename FactorType>
210 struct Factor
211 {
212     FactorType    factor;
213     FactorType    gramian;
214     Constraint    constraint;
215 };
216
217 } // end namespace partensor
218
219 #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 contains all nonzeros.
perm_tns_indices	[in,out] Stl array containing the Tensor indices (vector), which will be permuted.
Balanced_Ratings_Base_T	[in/out] The output matrix which contains 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 be 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 lled 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 be 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 contains all nonzeros.
perm_tns_indices	[in,out] Stl array containing the Tensor indices (vector), which will be permuted.
Balanced_Ratings_Base - _T	[in/out] The output matrix which contains 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 lled with the data.

Note

tnsX must be initialized before function call.

8.63 TensorOperations.hpp

[Go to the documentation of this le.](#)

```

1  #ifndef DOXYGEN_SHOULD_SKIP_THIS
15 #endif // DOXYGEN_SHOULD_SKIP_THIS
16 / ****
24 #ifndef PARTENSOR_TENSOR_OPERATIONS_HPP
25 #define PARTENSOR_TENSOR_OPERATIONS_HPP
26
27 #include <iostream>
28 #include "PARTENSOR_basic.hpp"
29
30 namespace partensor
31 {
32
47     template<typename Tensor_>
48     typename TensorTraits<Tensor_>::MatrixType tensorToMatrix( Tensor_ const &tnsX,
49                                                                 int const rows,
50                                                                 int const cols )
51     {
52         using Matrix = typename TensorTraits<Tensor_>::MatrixType;
53         return Eigen::Map<const Matrix>( tnsX.data(), rows, cols);
54     }
55
69     auto matrixToTensor( Matrix const &mtx,
70                         int const dim0,
71                         int const dim1 )
72     {
73         return Eigen::TensorMap<Eigen::Tensor<const DefaultDataType,2>(mtx.data(), {dim0,dim1});
74     }
75
90     template<typename Dimensions>
91     auto matrixToTensor( Matrix const &mtx,
92                         Dimensions const tnsDims )
93     {
94         static constexpr std::size_t TnsSize = tnsDims.size();
95         using tensormap = typename Eigen::TensorMap<Eigen::Tensor<const DefaultDataType,TnsSize>;
96
97         if constexpr (TnsSize == 3) {
98             return tensormap(mtx.data(), {tnsDims[0],tnsDims[1],tnsDims[2]});
99         }
100        else if constexpr (TnsSize == 4) {
101            return tensormap(mtx.data(), {tnsDims[0],tnsDims[1],tnsDims[2],tnsDims[3]});
102        }
103        else if constexpr (TnsSize == 5) {
104            return tensormap(mtx.data(),
105                             {tnsDims[0],tnsDims[1],tnsDims[2],tnsDims[3],tnsDims[4]});
106        }
107        else if constexpr (TnsSize == 6) {
108            return tensormap(mtx.data(),
109                             {tnsDims[0],tnsDims[1],tnsDims[2],tnsDims[3],tnsDims[4],tnsDims[5]});
110        }
111        else if constexpr (TnsSize == 7) {
112            return tensormap(mtx.data(),
113                             {tnsDims[0],tnsDims[1],tnsDims[2],tnsDims[3],tnsDims[4],tnsDims[5],tnsDims[6]});
114        }
115        else {
116            return tensormap(mtx.data(),
117                             {tnsDims[0],tnsDims[1],tnsDims[2],tnsDims[3],tnsDims[4],tnsDims[5],tnsDims[6],tnsDims[7]});
118        }
119    }
120
127     template<typename Tensor_>
128     double norm( Tensor_ const &tnsX )
129     {
130         Tensor<0> frob_norm_tens = tnsX.square().sum().sqrt();
131         return frob_norm_tens.coeff();
132     }
133
144     template<typename Tensor_>
145     double square_norm( Tensor_ const &tnsX )
146     {
147         Tensor<0> frob_norm_tens = tnsX.square().sum();
148         return frob_norm_tens.coeff();
149     }
150
151     #ifndef DOXYGEN_SHOULD_SKIP_THIS
163     template<std::size_t _TnsSize>
164     void TensorContraction( Tensor<static_cast<int>(_TnsSize)> const &parentTensor,
165                             Tensor<2> const
166                             &factor,
167                             int const
168                             contractDim1,
169                             int const
170                             contractDim2,
171                             Tensor<static_cast<int>(_TnsSize)>
172                             &contractionRes )
173     {
174         std::array<Eigen::IndexPair<int>, 1> product_dims = {

```

```

Eigen::IndexPair<int>(contractDim1, contractDim2) );
171     std::array<int, _TnsSize>          shuffles;
172
173     Tensor<static_cast<int>(_TnsSize)>  _temp;
174
175     for(std::size_t i=0; i<_TnsSize; i++) { shuffles[i] = i+1; }
176     shuffles[_TnsSize-1] = 0;
177
178     _temp          = factor.contract(parentTensor, product_dims);
179     contractionRes = _temp.shuffle(shuffles);
180 }
181 #endif // DOXYGEN_SHOULD_SKIP_THIS
182
183 #ifndef DOXYGEN_SHOULD_SKIP_THIS
184 template<std::size_t _ParTnsSize, std::size_t _TnsSize>
185 void TensorPartialProduct_R(Tensor<static_cast<int>(_ParTnsSize)> const &parentTensor,
186                             Tensor<2>
187                             const &factor,
188                             int
189                             chipDim,
190                             int
191                             contractDim,
192                             Tensor<static_cast<int>(_TnsSize)>
193                             * contractionRes )
194 {
195     static_assert(_ParTnsSize == _TnsSize+1,"Wrong call!");
196     const int R = factor.dimension(1);
197     std::array<Eigen::IndexPair<int>, 1> product_dims = { Eigen::IndexPair<int>(contractDim,
198 0) };
199
200     for (int i=0; i<R; i++)
201     {
202         contractionRes->chip(i,chipDim) =
203         (parentTensor.chip(i,chipDim)).contract(factor.chip(i,1),product_dims);
204     }
205 }
206 #endif // DOXYGEN_SHOULD_SKIP_THIS
207
208 template<typename Array_, typename Tensor_>
209 void RandomTensorGen(Array_ const tnsDims, Tensor_ &tnsX)
210 {
211     std::srand((unsigned int) time(NULL)+std::rand());
212     tnsX.resize(tnsDims);
213     tnsX.template setRandom<Eigen::internal::UniformRandomGenerator<double>>();
214 }
215
216 template<typename Array_, typename Tensor_>
217 void ZeroTensorGen(Array_ const tnsDims, Tensor_ &tnsX)
218 {
219     tnsX.resize(tnsDims);
220     tnsX.setZero();
221 }
222
223 template<typename Array_, typename Tensor_>
224 void IdentityTensorGen(Array_ const tnsDims, Tensor_ &tnsX)
225 {
226     static constexpr std::size_t TnsSize = TensorTraits<Tensor_>::TnsSize;
227
228     const int dim0 = tnsDims[0];
229     ZeroTensorGen(tnsDims, tnsX);
230
231     if constexpr(TnsSize == 2) {
232         for (int i=0; i<dim0; i++) {
233             tnsX(i,i) = 1;
234         }
235     }
236     else if constexpr(TnsSize == 3) {
237         for (int i=0; i<dim0; i++) {
238             tnsX(i,i,i) = 1;
239         }
240     }
241     else if constexpr(TnsSize == 4) {
242         for (int i=0; i<dim0; i++) {
243             tnsX(i,i,i,i) = 1;
244         }
245     }
246     else if constexpr(TnsSize == 5) {
247         for (int i=0; i<dim0; i++) {
248             tnsX(i,i,i,i,i) = 1;
249         }
250     }
251     else if constexpr(TnsSize == 6) {
252         for (int i=0; i<dim0; i++) {
253             tnsX(i,i,i,i,i,i) = 1;
254         }
255     }
256 }

```

```

313     }
314     else if constexpr(TnsSize == 7) {
315         for (int i=0; i<dim0; i++) {
316             tnsX(i,i,i,i,i,i) = 1;
317         }
318     }
319     else {
320         for (int i=0; i<dim0; i++) {
321             tnsX(i,i,i,i,i,i) = 1;
322         }
323     }
324 }
325
326 /* Parallel Version of ReserveSparseTensor */
327 template <std::size_t _TnsSize>
328 void ReserveSparseTensor(std::array<SparseMatrix, _TnsSize> &layer_tns_sparse,
329                          std::vector<std::vector<int>> &local_tns_dimensions,
330                          std::array<int, _TnsSize> &fiber_rank,
331                          int grid_size,
332                          long int nnz)
333 {
334     for (std::size_t i = 0; i < _TnsSize; i++)
335     {
336         long int col = 1;
337         for (std::size_t j = 0; j < _TnsSize; j++)
338         {
339             if (j == i)
340                 continue;
341
342             col = col * local_tns_dimensions[j][fiber_rank[j]];
343         }
344         layer_tns_sparse[i].resize(col, local_tns_dimensions[i][fiber_rank[i]]);
345         layer_tns_sparse[i].reserve((long int)(nnz / grid_size) + 1);
346     }
347 }
348
349 /* Serial Version of ReserveSparseTensor */
350 template<std::size_t _TnsSize>
351 void ReserveSparseTensor(std::array<SparseMatrix, _TnsSize> &tns_sparse,
352                          std::array<int, _TnsSize> &tns_dimensions,
353                          const long int nnz)
354 {
355     for (std::size_t i = 0; i < _TnsSize; i++)
356     {
357         long int col = 1;
358         for (std::size_t j = 0; j < _TnsSize; j++)
359         {
360             if (j == i)
361                 continue;
362
363             col = col * tns_dimensions[j];
364         }
365         tns_sparse[i].resize(col, tns_dimensions[i]);
366         tns_sparse[i].reserve(nnz);
367     }
368 }
369
370 // Assign nonzeros to the respective layer_tns_sparse subtensor.
371 template <std::size_t _TnsSize>
372 void FillSparseTensor(std::array<SparseMatrix, _TnsSize> &tns_sparse,
373                      long int Matrix,
374                      const nnz,
375                      const
376                      &Ratings_Base_T,
377                      std::array<int, _TnsSize> &tns_dimensions)
378 {
379     LongMatrix matr_mapping(static_cast<int>(_TnsSize), static_cast<int>(_TnsSize));
380     for (int i = 0; i < static_cast<int>(_TnsSize); i++)
381     {
382         for (int j = 0, first = 1, prev = 0; j < static_cast<int>(_TnsSize); j++)
383         {
384             if (j == i)
385             {
386                 matr_mapping(i, j) = 0;
387                 continue;
388             }
389             if (first == 1)
390             {
391                 matr_mapping(i, j) = 1;
392                 first = 0;
393                 prev = j;

```

```

394         }
395         else
396         {
397             matr_mapping(i, j) = matr_mapping(i, prev) *
tns_dimensions[prev];
398             prev = j;
399         }
400     }
401 }
402
403 LongMatrix tuple(1, _TnsSize);
404
405 for (long int nnz_k = 0; nnz_k < nnz; nnz_k++)
406 {
407     for (int column_idx = 0; column_idx < static_cast<int>(_TnsSize); column_idx++)
408     {
409         tuple(0, column_idx) = static_cast<long int>(Ratings_Base_T(column_idx,
nnz_k));
410     }
411
412     for (int mode_i = 0; mode_i < static_cast<int>(_TnsSize); mode_i++)
413     {
414         long int linear_col =
((matr_mapping.row(mode_i)).cwiseProduct(tuple)).sum());
415         long int row = tuple(0, mode_i);
416         if (tns_sparse[mode_i].outerSize() < row ||
tns_sparse[mode_i].innerSize() < linear_col)
417         {
418             std::cerr << "error!" << tns_sparse[mode_i].outerSize() << " " <<
row << " " << tns_sparse[mode_i].innerSize() << " " << linear_col << std::endl;
419         }
420
421         if (tns_sparse[mode_i].coeff(linear_col, row) == 0)
422         {
423             // Using insert to fill sparse matrix
424             tns_sparse[mode_i].insert(linear_col, row) =
Ratings_Base_T(_TnsSize, nnz_k);
425         }
426     }
427 }
428 }
429 }
430 }
431
432 template<std::size_t _TnsSize>
433 void FillSparseMatricization(std::array<SparseMatrix, _TnsSize> &tns_sparse,
434                             const long int
nnz,
435                             Matrix
&Ratings_Base_T,
436                             std::array<int, _TnsSize> const
&tns_dimensions,
437                             const int
cur_mode)
438 {
439     LongMatrix matr_mapping(static_cast<int>(_TnsSize), static_cast<int>(_TnsSize));
440     for (int i = 0; i < static_cast<int>(_TnsSize); i++)
441     {
442         for (int j = 0, first = 1, prev = 0; j < static_cast<int>(_TnsSize); j++)
443         {
444             if (j == i)
445             {
446                 matr_mapping(i, j) = 0;
447                 continue;
448             }
449             if (first == 1)
450             {
451                 matr_mapping(i, j) = 1;
452                 first = 0;
453                 prev = j;
454             }
455             else
456             {
457                 matr_mapping(i, j) = matr_mapping(i, prev) *
tns_dimensions[prev];
458                 prev = j;
459             }
460         }
461     }
462
463     LongMatrix tuple(1, static_cast<int>(_TnsSize));
464
465     for (int nnz_k = 0; nnz_k < nnz; nnz_k++)
466     {
467         for (int column_idx = 0; column_idx < static_cast<int>(_TnsSize); column_idx++)
468         {
469             tuple(0, column_idx) = static_cast<long int>(Ratings_Base_T(column_idx,

```

```

nnz_k));
470         }
471
472         long linear_col = ((matr_mapping.row(cur_mode)).cwiseProduct(tuple)).sum();
473
474         long row = tuple(0, cur_mode);
475
476         if (tns_sparse[cur_mode].outerSize() < row || tns_sparse[cur_mode].innerSize() <
linear_col)
477         {
478             std::cerr << "error!" << tns_sparse[cur_mode].outerSize() << " " << row << "
" << tns_sparse[cur_mode].innerSize() << " " << linear_col << std::endl;
479         }
480
481         if (tns_sparse[cur_mode].coeff(linear_col, row) == 0)
482         {
483             // Using insert to fill sparse matrix
484             tns_sparse[cur_mode].insert(linear_col, row) =
Ratings_Base_T(static_cast<int>(_TnsSize), nnz_k);
485         }
486     }
487 }
488
489 template <std::size_t _TnsSize>
490 void Dist_NNZ(std::array<SparseMatrix, _TnsSize> &layer_tns_sparse,
491              long int const nnz,
492              std::vector<std::vector<int>> const &skip_rows,
493              std::array<int, _TnsSize> const &fiber_rank,
494              Matrix const &Ratings_Base_T,
495              std::vector<std::vector<int>> const &local_tns_dimensions)
496 {
497     LongMatrix matr_mapping(static_cast<int>(_TnsSize), static_cast<int>(_TnsSize));
498     for (int i = 0; i < static_cast<int>(_TnsSize); i++)
499     {
500         for (int j = 0, first = 1, prev = 0; j < static_cast<int>(_TnsSize); j++)
501         {
502             if (j == i)
503             {
504                 matr_mapping(i, j) = 0;
505                 continue;
506             }
507             if (first == 1)
508             {
509                 matr_mapping(i, j) = 1;
510                 first = 0;
511                 prev = j;
512             }
513             else
514             {
515                 matr_mapping(i, j) = matr_mapping(i, prev) *
local_tns_dimensions[prev][fiber_rank[prev]];
516                 prev = j;
517             }
518         }
519     }
520
521     long int local_nnz_counter = 0;
522
523     LongMatrix tuple(1, static_cast<int>(_TnsSize));
524
525     for (int nnz_k = 0; nnz_k < nnz; nnz_k++)
526     {
527         for (int column_idx = 0, insert_tuple_flag = 0; column_idx <
static_cast<int>(_TnsSize) && insert_tuple_flag == column_idx; column_idx++)
528         {
529             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]]))
530             {
531                 tuple(0, column_idx) = (long int)(Ratings_Base_T(column_idx,
nnz_k)) - skip_rows[column_idx][fiber_rank[column_idx]];
532                 insert_tuple_flag++;
533             }
534             else
535             {
536                 break;
537             }
538             if (column_idx == static_cast<int>(_TnsSize) - 1)
539             {
540                 local_nnz_counter++;
541                 for (int mode_i = 0; mode_i < static_cast<int>(_TnsSize);
mode_i++)
542                 {
543                     long int linear_col =
((matr_mapping.row(mode_i)).cwiseProduct(tuple)).sum();
544                     long int row = tuple(0, mode_i);

```

```

545
546                                     if (layer_tns_sparse[mode_i].coeff(linear_col, row) ==
547 0)
548                                     {
549                                     layer_tns_sparse[mode_i].insert(linear_col, row)
550 = Ratings_Base_T(static_cast<int>(_TnsSize), nnz_k);
551                                     }
552                                     }
553                                     }
554     }
555
556     template <std::size_t _TnsSize>
557     void Dist_NNZ_sorted(std::array<SparseMatrix, _TnsSize> &layer_tns_sparse,
558                         long int                                const nnz,
559                         std::vector<std::vector<int>>          const &skip_rows,
560                         std::array<int, _TnsSize>              const &fiber_rank,
561                         Matrix                                  const
562 &Ratings_Base_T,
563                         std::vector<std::vector<int>>          const
564 &local_tns_dimensions,
565                         int                                      const cur_mode)
566     {
567         LongMatrix matr_mapping(static_cast<int>(_TnsSize), static_cast<int>(_TnsSize));
568         for (int i = 0; i < static_cast<int>(_TnsSize); i++)
569         {
570             for (int j = 0, first = 1, prev = 0; j < static_cast<int>(_TnsSize); j++)
571             {
572                 if (j == i)
573                 {
574                     matr_mapping(i, j) = 0;
575                     continue;
576                 }
577                 if (first == 1)
578                 {
579                     matr_mapping(i, j) = 1;
580                     first = 0;
581                     prev = j;
582                 }
583                 else
584                 {
585                     matr_mapping(i, j) = matr_mapping(i, prev) *
586 local_tns_dimensions[prev][fiber_rank[prev]];
587                     prev = j;
588                 }
589             }
590         }
591         long int local_nnz_counter = 0;
592         LongMatrix tuple(1, static_cast<int>(_TnsSize));
593         for (int nnz_k = 0; nnz_k < nnz; nnz_k++)
594         {
595             for (int column_idx = 0, insert_tuple_flag = 0; column_idx <
596 static_cast<int>(_TnsSize) && insert_tuple_flag == column_idx; column_idx++)
597             {
598                 if ((Ratings_Base_T(column_idx, nnz_k) >=
599 skip_rows[column_idx][fiber_rank[column_idx]] && (Ratings_Base_T(column_idx, nnz_k) <
600 local_tns_dimensions[column_idx][fiber_rank[column_idx]] +
601 skip_rows[column_idx][fiber_rank[column_idx]]))
602                 {
603                     tuple(0, column_idx) = (long int)(Ratings_Base_T(column_idx,
604 nnz_k) - skip_rows[column_idx][fiber_rank[column_idx]]);
605                     insert_tuple_flag++;
606                 }
607                 else
608                 {
609                     break;
610                 }
611                 if (column_idx == static_cast<int>(_TnsSize) - 1)
612                 {
613                     local_nnz_counter++;
614                 }
615                 long int linear_col =
616 ((matr_mapping.row(cur_mode)).cwiseProduct(tuple)).sum();
617                 long int row = tuple(0, cur_mode);
618                 if (layer_tns_sparse[cur_mode].coeff(linear_col, row) == 0)
619                 {
620                     layer_tns_sparse[cur_mode].insert(linear_col, row) =
621 Ratings_Base_T(static_cast<int>(_TnsSize), nnz_k);
622                 }
623             }
624         }
625     }

```

```

620     }
621
622     template <int TnsSize, int mode, typename Type>
623     bool SortRows(const std::vector<Type> &v1, const std::vector<Type> &v2)
624     {
625         std::array<int, TnsSize> sort_direction;
626         bool final_expr{false}; // final criterion for sorting
627         bool prev_equal{false}; // keep history of equal comparisons between v1,v2
628
629         sort_direction[0] = mode;
630         sort_direction[1] = (mode < TnsSize - 1) ? TnsSize - 1 : TnsSize - 2;
631         for (int i = 2; i < TnsSize; i++)
632         {
633             sort_direction[i] = sort_direction[i - 1] - 1;
634             if (sort_direction[i] == mode)
635             {
636                 sort_direction[i]--;
637             }
638         }
639         std::array<bool, TnsSize> expr;
640         for (int i = 0; i < TnsSize; i++)
641         {
642             if (i > 0)
643             {
644                 expr[i] = prev_equal && (v1[sort_direction[i]] < v2[sort_direction[i]]);
645                 final_expr = final_expr || expr[i];
646                 prev_equal = prev_equal && (v1[sort_direction[i]] ==
v2[sort_direction[i]]);
647
648                 if (final_expr)
649                 {
650                     return final_expr;
651                 }
652             }
653             else
654             {
655                 expr[i] = v1[sort_direction[i]] < v2[sort_direction[i]];
656                 final_expr = final_expr || expr[i];
657                 prev_equal = (v1[sort_direction[i]] == v2[sort_direction[i]]);
658                 if (final_expr)
659                 {
660                     return final_expr;
661                 }
662             }
663         }
664         return final_expr;
665     }
666 }
667
668 void PermutedModeN(long int nnz, const int cur_mode,
669                       int tns_dim, const Matrix &Ratings_Base_T,
670                       std::vector<long int> &perm_tns_indices,
671                       Matrix &Balanced_Ratings_Base_T)
672 {
673     // Copy values
674     Balanced_Ratings_Base_T = Ratings_Base_T;
675
676     // Allocate & Initialize permuted dims
677     perm_tns_indices.reserve(tns_dim);
678
679     for (int i_j = 0; i_j < tns_dim; i_j++)
680     {
681         perm_tns_indices.push_back(i_j);
682     }
683
684     // Permute dims
685     std::random_device rd;
686     std::mt19937 g(rd());
687     std::shuffle(perm_tns_indices.begin(), perm_tns_indices.end(), g);
688     double prev = -1;
689     long int idx;
690     std::vector<long int>::iterator it;
691     long int perm_idx = 0;
692     for (long int nnz_i = 0; nnz_i < nnz; nnz_i++)
693     {
694         if(Ratings_Base_T(cur_mode, nnz_i) != prev)
695         {
696             idx = static_cast<long int>(Ratings_Base_T(cur_mode, nnz_i));
697             it = std::find(perm_tns_indices.begin(), perm_tns_indices.end(), idx);
698             perm_idx = it - perm_tns_indices.begin();
699             Balanced_Ratings_Base_T(cur_mode, nnz_i) =
static_cast<double>(perm_idx);
700             prev = Ratings_Base_T(cur_mode, nnz_i);
701         }
702     }
703 }

```



```

716             else
717             {
718                 Balanced_Ratings_Base_T(cur_mode, nnz_i) =
static_cast<double>(perm_idx);
719                 prev = Ratings_Base_T(cur_mode, nnz_i);
720             }
721         }
722     }
723
724     template <std::size_t TnsSize>
725     void BalanceDataset(long int
726     const nnz,
727     std::array<int, TnsSize>
728     const tns_dimensions,
729     Matrix
730     const &Ratings_Base_T,
731     std::array<std::vector<long int>, TnsSize>
732     &perm_tns_indices,
733     Matrix
734     &Balanced_Ratings_Base_T)
735     {
736         for (int i = 0; i < static_cast<int>(TnsSize); i++)
737         {
738             PermuteModeN(nnz, i, tns_dimensions[i], Ratings_Base_T, perm_tns_indices[i],
739             Balanced_Ratings_Base_T);
740         }
741     }
742
743     template <std::size_t TnsSize>
744     void PermuteFactors(std::array<Matrix, TnsSize> const &depermuted_factors,
745     std::array<std::vector<long int>, TnsSize> const
746     &perm_tns_indices,
747     std::array<Matrix, TnsSize>
748     &permuted_factors_T)
749     {
750         Matrix temp_permuted_factor;
751         for (int i = 0; i < static_cast<int>(TnsSize); i++)
752         {
753             temp_permuted_factor = Matrix::Zero(depermuted_factors[i].rows(),
754             depermuted_factors[i].cols());
755             for (int row = 0; row < temp_permuted_factor.rows(); row++)
756             {
757                 temp_permuted_factor.row(row) =
758                 depermuted_factors[i].row(perm_tns_indices[i][row]);
759             }
760             permuted_factors_T[i] = temp_permuted_factor.transpose();
761         }
762     }
763
764     template <std::size_t TnsSize>
765     void DepermuteFactors(std::array<Matrix, TnsSize> const &permuted_factors,
766     std::array<std::vector<long int>, TnsSize> const
767     &perm_tns_indices,
768     std::array<Matrix, TnsSize>
769     &depermuted_factors)
770     {
771         for (int i = 0; i < static_cast<int>(TnsSize); i++)
772         {
773             depermuted_factors[i] = Matrix::Zero(permuted_factors[i].rows(),
774             permuted_factors[i].cols());
775             for (int row = 0; row < permuted_factors[i].rows(); row++)
776             {
777                 depermuted_factors[i].row(perm_tns_indices[i][row]) =
778                 permuted_factors[i].row(row);
779             }
780         }
781     }
782 } // end namespace partensor
783
784 #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

Contains 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

<code>MatrixArray</code>	An array container type.
--------------------------	--------------------------

Parameters

<code>origMatrTns</code>	[in] The matricization of the original/true Tensor. It must be in Eigen Matrix format.
<code>factors</code>	[in] Contains the factors generated from a factorization algorithm, like <code>cpd</code> , <code>cpdDimTree</code> , etc.
<code>mode</code>	[in] Specify in which mode, is <code>origMatrTns</code> 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 than 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.](#)

```
1 #ifndef DOXYGEN_SHOULD_SKIP_THIS
15 #endif // DOXYGEN_SHOULD_SKIP_THIS
16 / *****
24 #ifndef PARTENSOR_TERMINATION_CONDITIONS_HPP
25 #define PARTENSOR_TERMINATION_CONDITIONS_HPP
26
27 #include "DataGeneration.hpp"
28 #include "math.h"
29
30 namespace partensor {
31
32     struct Conditions {
33         int const max_iter = 500; // Maximum Number of iterations
34         double const ao_tol = 1e-2; // Tolerance for AO Algorithm
35         double const delta_1 = 1e-2; // | Tolerance for Nesterov
36         Algorithm
37         double const delta_2 = 1e-2; // |
38     };
39
40     static inline Conditions con;
41     inline int maxIterations(int const countIter)
42     {
43         return (countIter >= con.max_iter) ? 1 : 0;
44     }
45
46     inline int relativeCostFunction(double const newValue, double const trueValue)
47     {
48         return (newValue/sqrt(trueValue) <= con.ao_tol) ? 1 : 0;
49     }
50 }
```

```

71     }
72
73     inline int relativeError(double const newValue, double const trueValue)
74     {
75         return (abs(trueValue-newValue)/trueValue <= con.ao_tol) ? 1 : 0;
76     }
77
78     inline int objectiveValueError(double const newValue, double const pastValue)
79     {
80         return (abs(newValue-pastValue) <= con.ao_tol) ? 1 : 0;
81     }
82
83     template<typename MatrixArray>
84     double error( typename MatrixArrayTraits<MatrixArray>::value_type const &origMatrTns,
85                 MatrixArray const &factors,
86                 std::size_t const mode )
87     {
88         using Matrix = typename MatrixArrayTraits<MatrixArray>::value_type;
89
90         Matrix localMatrTns = generateTensor(mode, factors);
91         return (origMatrTns - localMatrTns).squaredNorm();
92     }
93 } // end namespace partensor
94
95 #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.](#)

```

1 #ifndef DOXYGEN_SHOULD_SKIP_THIS
15 #endif // DOXYGEN_SHOULD_SKIP_THIS
16 / *****
27 #ifndef PARTENSOR_TIMERS_HPP
28 #define PARTENSOR_TIMERS_HPP
29
30 #include <chrono>
31 #include "mpi.h"
32
33 namespace partensor {
34
35     struct Timers
36     {
37     public:
38         using ClockSteady = std::chrono::time_point<std::chrono::steady_clock>;
39         using ClockHigh = std::chrono::time_point<std::chrono::high_resolution_clock>;
40         clock_t      cpu_current_time;
41         double       cpu_elapsed_time;
42
43         ClockHigh    chrono_high_current_time;
44         double       chrono_high_elapsed_time;
45
46         ClockSteady  chrono_steady_current_time;
47         double       chrono_steady_elapsed_time;
48
49         double       mpi_current_time;
50         double       mpi_elapsed_time;
51
52         void startCpuTimer ()
53         {
54             cpu_current_time = clock();
55         }
56
57         double endCpuTimer ()
58         {
59             auto finish = clock();
60             cpu_elapsed_time = (float)(finish - cpu_current_time)/(float)CLOCKS_PER_SEC ;
61             return cpu_elapsed_time;
62         }
63
64         void startChronoHighTimer ()
65         {
66             chrono_high_current_time = std::chrono::high_resolution_clock::now();
67         }
68
69         double endChronoHighTimer ()
70         {
71             std::chrono::duration<double> finish = std::chrono::high_resolution_clock::now() -
72             chrono_high_current_time;
73             chrono_high_elapsed_time = finish.count();
74             return chrono_high_elapsed_time;
75         }
76
77         void startChronoSteadyTimer ()
78         {
79             chrono_steady_current_time = std::chrono::steady_clock::now();
80         }
81
82         double endChronoSteadyTimer ()
83         {
84             std::chrono::duration<double> finish = std::chrono::steady_clock::now() -
85             chrono_steady_current_time;
86             chrono_steady_elapsed_time = finish.count();
87             return chrono_steady_elapsed_time;
88         }
89
90         void startMpiTimer ()
91         {
92             mpi_current_time = MPI_Wtime();
93         }
94     };
95
96 }
97
98 #endif

```

```
126
133     double endMpiTimer ()
134     {
135         auto finish = MPI_Wtime();
136         mpi_elapsed_time = (finish - mpi_current_time);
137         return mpi_elapsed_time;
138     }
139
140 };
141
142 static inline Timers timer;
144 } // end namespace partensor
145
146
147 #endif // end of PARTENSOR_TIMERS_HPP
```


Index

- environment
 - environment, [74](#)
- all_reduce
 - ParallelWrapper.hpp, [323](#)
- ao_iter
 - SparseStatus< _TnsSize, ExecutionPolicy_, DefaultValues_ >, [99](#)
 - Status< Tensor_, ExecutionPolicy_, DefaultValues_ >, [102](#)
- BalanceDataset
 - TensorOperations.hpp, [370](#)
- Boost_CartCommunicator
 - ParallelWrapper.hpp, [322](#)
- Boost_CartDimension
 - ParallelWrapper.hpp, [322](#)
- Boost_CartTopology
 - ParallelWrapper.hpp, [322](#)
- Boost_Communicator
 - ParallelWrapper.hpp, [322](#)
- Boost_Environment
 - ParallelWrapper.hpp, [322](#)
- BrotherLabelSetSize
 - ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >, [80](#)
- cartesian_communicator, [49](#)
 - cartesian_communicator, [49, 50](#)
- cartesian_dimension, [50](#)
 - cartesian_dimension, [50](#)
- cartesian_topology, [51](#)
 - cartesian_topology, [51](#)
- choose_normilization_factor
 - Normalize.hpp, [316](#)
- Clock
 - PARTENSOR_basic.hpp, [333](#)
- ClockHigh
 - Timers, [105](#)
- ClockSteady
 - Timers, [105](#)
- communicator, [52](#)
 - communicator, [52](#)
- ComputeSVD
 - NesterovMNLS.hpp, [302](#)
- Conditions, [52](#)
- Con g.hpp, [117](#)
- constant
 - Constants.hpp, [118](#)
- Constants.hpp, [117, 119](#)
 - constant, [118](#)
 - Constraint, [118](#)
 - Distribution, [118](#)
 - Method, [118](#)
 - nonnegativity, [118](#)
 - orthogonality, [118](#)
 - ProblemType, [118](#)
 - sparsity, [118](#)
 - symmetric, [118](#)
 - symmetric_nonnegativity, [118](#)
 - unconstrained, [118](#)
- Constraint
 - Constants.hpp, [118](#)
- Constraints
 - SparseTensorTraits< SparseTensor< _TnsSize > >, [101](#)
 - TensorTraits< Tensor< _TnsSize > >, [104](#)
- cpd
 - Cpd.hpp, [120–126](#)
- CPD< Tensor_, execution::openmp_policy >, [53](#)
 - operator(), [53–58](#)
- CPD< Tensor_, execution::openmpi_policy >, [58](#)
 - IntArray, [59](#)
 - operator(), [59, 61–64](#)
- Cpd.hpp, [119, 127](#)
 - cpd, [120–126](#)
- CPD_DIMTREE< Tensor_, execution::openmpi_policy >, [65](#)
 - operator(), [66–69](#)
- cpdDimTree
 - CpdDimTree.hpp, [142–147](#)
- CpdDimTree.hpp, [141, 148](#)
 - cpdDimTree, [142–147](#)
- CpdDimTreeMpi.hpp, [161](#)
- CpdMpi.hpp, [173, 174](#)
- CpdOpenMP.hpp, [187](#)
- Create
 - ExprTree< _TnsSize >, [84](#)
- create_ber_grid
 - ParallelWrapper.hpp, [324](#)
- create_layer_grid
 - ParallelWrapper.hpp, [324](#)
- CwiseProd
 - CwiseProd.hpp, [194](#)
- CwiseProd.hpp, [194, 195](#)
 - CwiseProd, [194](#)
- DataGeneration.hpp, [195, 199](#)
 - generateRandomMatrix, [196](#)
 - generateRandomTensor, [196](#)

- generateTensor, [197](#)
- makeFactors, [198](#)
- makeTensor, [199](#)
- DataType
 - Options< Tensor_, ExecutionPolicy_, DefaultValues_ >, [97](#)
- DefaultAcceleration
 - DefaultValues< Tensor_ >, [71](#)
- DefaultAccelerationCoefficient
 - DefaultValues< Tensor_ >, [71](#)
- DefaultAccelerationFail
 - DefaultValues< Tensor_ >, [71](#)
- DefaultConstraint
 - DefaultValues< Tensor_ >, [71](#)
- DefaultDataType
 - Tensor.hpp, [366](#)
- DefaultLambda
 - DefaultValues< Tensor_ >, [71](#)
- DefaultMaxDuration
 - DefaultValues< Tensor_ >, [71](#)
- DefaultMaxIter
 - DefaultValues< Tensor_ >, [72](#)
- DefaultMethod
 - DefaultValues< Tensor_ >, [72](#)
- DefaultNesterovTolerance
 - DefaultValues< Tensor_ >, [72](#)
- DefaultNormalization
 - DefaultValues< Tensor_ >, [72](#)
- DefaultProcessorPerMode
 - DefaultValues< Tensor_ >, [72](#)
- DefaultThresholdError
 - DefaultValues< Tensor_ >, [72](#)
- DefaultValues< Tensor_ >, [70](#)
 - DefaultAcceleration, [71](#)
 - DefaultAccelerationCoefficient, [71](#)
 - DefaultAccelerationFail, [71](#)
 - DefaultConstraint, [71](#)
 - DefaultLambda, [71](#)
 - DefaultMaxDuration, [71](#)
 - DefaultMaxIter, [72](#)
 - DefaultMethod, [72](#)
 - DefaultNesterovTolerance, [72](#)
 - DefaultNormalization, [72](#)
 - DefaultProcessorPerMode, [72](#)
 - DefaultThresholdError, [72](#)
 - DefaultWriteToFile, [72](#)
- DefaultWriteToFile
 - DefaultValues< Tensor_ >, [72](#)
- DeltaSet
 - ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >, [76](#)
 - I_TnsNode, [91](#)
 - TnsNode< 0 >, [112](#)
- DepermuteFactors
 - TensorOperations.hpp, [371](#)
- DIM_HALF_SIZE
 - ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >, [81](#)
- DIM_LEFT_SIZE
 - ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >, [81](#)
- DIM_RIGHT_SIZE
 - ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >, [81](#)
- DimTrees.hpp, [206](#), [207](#)
 - search_leaf, [207](#)
- DisCount
 - ParallelWrapper.hpp, [324](#)
- Distribution
 - Constants.hpp, [118](#)
- DoubleArray
 - SparseTensorTraits< SparseTensor< _TnsSize > >, [101](#)
 - TensorTraits< Tensor< _TnsSize > >, [104](#)
- Duration
 - PARTENSOR_basic.hpp, [333](#)
- endChronoHighTimer
 - Timers, [106](#)
- endChronoSteadyTimer
 - Timers, [106](#)
- endCpuTimer
 - Timers, [106](#)
- endMpiTimer
 - Timers, [106](#)
- environment, [73](#)
 - environment, [74](#)
 - environment, [73](#)
- error
 - TerminationConditions.hpp, [386](#)
- execution.hpp, [215](#), [216](#)
- ExprNode
 - ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >, [76](#)
- ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >, [74](#)
 - BrotherLabelSetSize, [80](#)
 - DeltaSet, [76](#)
 - DIM_HALF_SIZE, [81](#)
 - DIM_LEFT_SIZE, [81](#)
 - DIM_RIGHT_SIZE, [81](#)
 - ExprNode, [76](#)
 - IsFirstChild, [81](#)
 - IsLeaf, [81](#)
 - IsRoot, [81](#)
 - LabelSet, [76](#)
 - LabelSetSize, [81](#)
 - Left, [76](#)
 - left, [81](#)
 - mDeltaSet, [82](#)
 - mGramian, [82](#)
 - mKey, [82](#)
 - mLabelSet, [82](#)
 - mTnsDims, [82](#)
 - mTnsX, [82](#)
 - mUpdated, [82](#)
 - Parent, [77](#)

- parent, [82](#)
- ParLabelSetSize, [83](#)
- ParTnsSize, [83](#)
- Right, [77](#)
- right, [83](#)
- RootSize, [83](#)
- SearchKey, [77](#)
- TnsDims, [78](#)
- TnsSize, [83](#)
- TreeMode_N_Product, [78](#)
- TTVs, [79](#)
- TTVs_util, [79](#)
- UpdateTree, [80](#)
- ExprTree< _TnsSize >, [83](#)
 - Create, [84](#)
 - IsNull, [85](#)
 - root, [85](#)
- f_value
 - SparseStatus< _TnsSize, ExecutionPolicy_, DefaultValues_ >, [99](#)
 - Status< Tensor_, ExecutionPolicy_, DefaultValues_ >, [102](#)
- Factor< FactorType >, [85](#)
- FactorDimTree, [86](#)
- factors
 - SparseStatus< _TnsSize, ExecutionPolicy_, DefaultValues_ >, [99](#)
 - Status< Tensor_, ExecutionPolicy_, DefaultValues_ >, [102](#)
- frob_tns
 - SparseStatus< _TnsSize, ExecutionPolicy_, DefaultValues_ >, [99](#)
 - Status< Tensor_, ExecutionPolicy_, DefaultValues_ >, [103](#)
- generateRandomMatrix
 - DataGeneration.hpp, [196](#)
- generateRandomTensor
 - DataGeneration.hpp, [196](#)
- generateTensor
 - DataGeneration.hpp, [197](#)
- GLambda
 - NesterovMNLS.hpp, [303](#)
- Gramian
 - I_TnsNode, [91](#)
 - TnsNode< 0 >, [112](#)
 - TnsNode< _TnsSize >, [108](#)
- gtc
 - Gtc.hpp, [220](#), [221](#)
- GTC< TnsSize_, execution::openmpi_policy >, [86](#)
 - initialize_factors, [87](#)
 - IntArray, [87](#)
 - operator(), [87](#), [88](#)
- Gtc.hpp, [219](#), [221](#)
 - gtc, [220](#), [221](#)
- gtc_stochastic
 - GtcStochastic.hpp, [250](#), [251](#)
- GTC_STOCHASTIC< TnsSize_, execution::openmpi_policy >, [88](#)
 - initialize_factors, [89](#)
 - IntArray, [89](#)
 - operator(), [89](#), [90](#)
- GtcMpi.hpp, [229](#), [230](#)
- GtcOpenMP.hpp, [242](#)
- GtcStochastic.hpp, [250](#), [252](#)
 - gtc_stochastic, [250](#), [251](#)
- GtcStochasticMpi.hpp, [260](#)
- GtcStochasticOpenMP.hpp, [270](#)
- I_TnsNode, [90](#)
 - DeltaSet, [91](#)
 - Gramian, [91](#)
 - I_TnsNode, [91](#)
 - Key, [92](#)
 - LabelSet, [92](#)
 - Left, [92](#)
 - Parent, [92](#)
 - Right, [92](#)
 - SearchKey, [93](#)
 - SetOutdated, [93](#)
 - TensorX, [93](#)
 - TnsDims, [93](#)
 - TnsSize, [94](#)
 - Updated, [94](#)
 - UpdateTree, [94](#)
- IdentityTensorGen
 - TensorOperations.hpp, [371](#)
- initialize_factors
 - GTC< TnsSize_, execution::openmpi_policy >, [87](#)
 - GTC_STOCHASTIC< TnsSize_, execution::openmpi_policy >, [89](#)
- inplace
 - ParallelWrapper.hpp, [325](#)
- IntArray
 - CPD< Tensor_, execution::openmpi_policy >, [59](#)
 - GTC< TnsSize_, execution::openmpi_policy >, [87](#)
 - GTC_STOCHASTIC< TnsSize_, execution::openmpi_policy >, [89](#)
 - SparseTensorTraits< SparseTensor< _TnsSize > >, [101](#)
 - TensorTraits< Tensor< _TnsSize > >, [104](#)
- is_matrix
 - Tensor.hpp, [367](#)
- IsFirstChild
 - ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >, [81](#)
- IsLeaf
 - ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >, [81](#)
- IsNull
 - ExprTree< _TnsSize >, [85](#)
 - TnsNode< 0 >, [115](#)
 - TnsNode< _TnsSize >, [110](#)
- IsRoot
 - ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >, [81](#)

- Key
 - I_TnsNode, [92](#)
 - TnsNode< 0 >, [113](#)
 - TnsNode< _TnsSize >, [109](#)
- KhatriRao
 - KhatriRao.hpp, [277](#)
- KhatriRao.hpp, [277](#), [278](#)
 - KhatriRao, [277](#)
- Kronecker
 - Kronecker.hpp, [285](#)
- Kronecker.hpp, [285](#), [287](#)
 - Kronecker, [285](#)
- LabelSet
 - ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >, [76](#)
 - I_TnsNode, [92](#)
 - TnsNode< 0 >, [113](#)
- LabelSetSize
 - ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >, [81](#)
- Left
 - ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >, [76](#)
 - I_TnsNode, [92](#)
 - TnsNode< 0 >, [113](#)
- left
 - ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >, [81](#)
- LongMatrix
 - Tensor.hpp, [367](#)
- makeFactors
 - DataGeneration.hpp, [198](#)
- MakeOptions
 - PARTENSOR_basic.hpp, [334](#)
- MakeSparseOptions
 - PARTENSOR_basic.hpp, [334](#)
- makeTensor
 - DataGeneration.hpp, [199](#)
- Matricization
 - Matricization.hpp, [288](#)
- Matricization.hpp, [287](#), [289](#)
 - Matricization, [288](#)
- Matrix
 - Tensor.hpp, [367](#)
- MatrixArray
 - SparseStatus< _TnsSize, ExecutionPolicy_, DefaultValues_ >, [99](#)
 - SparseTensorTraits< SparseTensor< _TnsSize > >, [101](#)
 - Status< Tensor_, ExecutionPolicy_, DefaultValues_ >, [102](#)
 - TensorTraits< Tensor< _TnsSize > >, [104](#)
- MatrixArrayTraits< MA >, [95](#)
- MatrixArrayTraits< std::array< T, _Size > >, [95](#)
- matrixToTensor
 - TensorOperations.hpp, [372](#)
- MatrixTraits< Matrix >, [96](#)
- MatrixType
 - Options< Tensor_, ExecutionPolicy_, DefaultValues_ >, [97](#)
- maxIterations
 - TerminationConditions.hpp, [387](#)
- mDeltaSet
 - ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >, [82](#)
- Method
 - Constants.hpp, [118](#)
- mGramian
 - ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >, [82](#)
 - TnsNode< _TnsSize >, [110](#)
- mKey
 - ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >, [82](#)
 - TnsNode< _TnsSize >, [110](#)
- mLabelSet
 - ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >, [82](#)
- mTnsDims
 - ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >, [82](#)
- mTnsX
 - ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >, [82](#)
 - TnsNode< _TnsSize >, [110](#)
- mttkrp
 - MTTKRP.hpp, [296](#)
- MTTKRP.hpp, [295](#), [297](#)
 - mttkrp, [296](#)
- mUpdated
 - ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >, [82](#)
 - TnsNode< _TnsSize >, [110](#)
- NesterovMNLS
 - NesterovMNLS.hpp, [303](#)
- NesterovMNLS.hpp, [302](#), [304](#)
 - ComputeSVD, [302](#)
 - GLambda, [303](#)
 - NesterovMNLS, [303](#)
 - UpdateAlpha, [303](#)
- nonnegativity
 - Constants.hpp, [118](#)
- norm
 - TensorOperations.hpp, [373](#)
- Normalize
 - Normalize.hpp, [317](#)
- Normalize.hpp, [316](#), [318](#)
 - choose_normilization_factor, [316](#)
 - Normalize, [317](#)
- objectiveValueError
 - TerminationConditions.hpp, [387](#)
- operator()
 - CPD< Tensor_, execution::openmp_policy >, [53–58](#)

- CPD< Tensor_, execution::openmpi_policy > , 59, 61–64
- CPD_DIMTREE< Tensor_, execution::openmpi_policy > , 66–69
- GTC< TnsSize_, execution::openmpi_policy > , 87, 88
- GTC_STOCHASTIC< TnsSize_, execution::openmpi_policy > , 89, 90
- Options
 - Options< Tensor_, ExecutionPolicy_, DefaultValues_ > , 97
- Options< Tensor_, ExecutionPolicy_, DefaultValues_ > , 96
 - DataType, 97
 - MatrixType, 97
 - Options, 97
 - TnsSize, 98
- orthogonality
 - Constants.hpp, 118
- ParallelWrapper.hpp, 321, 325
 - all_reduce, 323
 - Boost_CartCommunicator, 322
 - Boost_CartDimension, 322
 - Boost_CartTopology, 322
 - Boost_Communicator, 322
 - Boost_Environment, 322
 - create_ber_grid, 324
 - create_layer_grid, 324
 - DisCount, 324
 - inplace, 325
- Parent
 - ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize > , 77
 - I_TnsNode, 92
 - TnsNode< 0 > , 113
- parent
 - ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize > , 82
- ParLabelSetSize
 - ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize > , 83
- PARTENSOR.hpp, 331, 332
- PARTENSOR_basic.hpp, 332, 335
 - Clock, 333
 - Duration, 333
 - MakeOptions, 334
 - MakeSparseOptions, 334
- PartialCwiseProd
 - PartialCwiseProd.hpp, 344
- PartialCwiseProd.hpp, 344, 345
 - PartialCwiseProd, 344
- PartialKhatriRao
 - PartialKhatriRao.hpp, 348
- PartialKhatriRao.hpp, 347, 349
 - PartialKhatriRao, 348
- ParTnsSize
 - ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize > , 83
- PermuteFactors
 - TensorOperations.hpp, 373
- PermuteModeN
 - TensorOperations.hpp, 374
- ProblemType
 - Constants.hpp, 118
- RandomTensorGen
 - TensorOperations.hpp, 374
- read
 - ReadWrite.hpp, 352
- readFMRI_matrix
 - ReadWrite.hpp, 353
- readFMRI_mpi
 - ReadWrite.hpp, 353
- readFMRI_tensor
 - ReadWrite.hpp, 354
- readTensor
 - ReadWrite.hpp, 355
- ReadWrite.hpp, 351, 357
 - read, 352
 - readFMRI_matrix, 353
 - readFMRI_mpi, 353
 - readFMRI_tensor, 354
 - readTensor, 355
 - write, 356
 - writeToFile_append, 356
- rel_costFunction
 - SparseStatus< _TnsSize, ExecutionPolicy_, DefaultValues_ > , 99
 - Status< Tensor_, ExecutionPolicy_, DefaultValues_ > , 103
- relativeCostFunction
 - TerminationConditions.hpp, 387
- relativeError
 - TerminationConditions.hpp, 388
- Right
 - ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize > , 77
 - I_TnsNode, 92
 - TnsNode< 0 > , 113
- right
 - ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize > , 83
- root
 - ExprTree< _TnsSize > , 85
- RootSize
 - ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize > , 83
- search_leaf
 - DimTrees.hpp, 207
- SearchKey
 - ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize > , 77
 - I_TnsNode, 93
 - TnsNode< 0 > , 114
- SetOutdated
 - I_TnsNode, 93

- TnsNode < 0 >, 114
- TnsNode < _TnsSize >, 109
- SparseMatrix
 - Tensor.hpp, 367
- SparseStatus < _TnsSize, ExecutionPolicy_, DefaultValues_ >, 98
 - ao_iter, 99
 - f_value, 99
 - factors, 99
 - frob_tns, 99
 - MatrixArray, 99
 - rel_costFunction, 99
- SparseTensorTraits < SparseTensor >, 100
- SparseTensorTraits < SparseTensor < _TnsSize > >, 100
 - Constraints, 101
 - DoubleArray, 101
 - IntArray, 101
 - MatrixArray, 101
- sparsity
 - Constants.hpp, 118
- square_norm
 - TensorOperations.hpp, 376
- startChronoHighTimer
 - Timers, 107
- startChronoSteadyTimer
 - Timers, 107
- startCpuTimer
 - Timers, 107
- startMpiTimer
 - Timers, 107
- Status < Tensor_, ExecutionPolicy_, DefaultValues_ >, 101
 - ao_iter, 102
 - f_value, 102
 - factors, 102
 - frob_tns, 103
 - MatrixArray, 102
 - rel_costFunction, 103
- symmetric
 - Constants.hpp, 118
- symmetric_nonnegativity
 - Constants.hpp, 118
- temp.hpp, 363
- Tensor
 - Tensor.hpp, 367
- Tensor.hpp, 366, 368
 - DefaultDataType, 366
 - is_matrix, 367
 - LongMatrix, 367
 - Matrix, 367
 - SparseMatrix, 367
 - Tensor, 367
- Tensor_Type
 - TnsNode < 0 >, 112
- TensorOperations.hpp, 369, 377
 - BalanceDataset, 370
 - DepermuteFactors, 371
 - IdentityTensorGen, 371
 - matrixToTensor, 372
 - norm, 373
 - PermuteFactors, 373
 - PermuteModeN, 374
 - RandomTensorGen, 374
 - square_norm, 376
 - tensorToMatrix, 376
 - ZeroTensorGen, 377
- tensorToMatrix
 - TensorOperations.hpp, 376
- TensorTraits < Tensor >, 103
- TensorTraits < Tensor < _TnsSize > >, 103
 - Constraints, 104
 - DoubleArray, 104
 - IntArray, 104
 - MatrixArray, 104
- TensorX
 - I_TnsNode, 93
 - TnsNode < 0 >, 114
 - TnsNode < _TnsSize >, 109
- TerminationConditions.hpp, 385, 388
 - error, 386
 - maxIterations, 387
 - objectiveValueError, 387
 - relativeCostFunction, 387
 - relativeError, 388
- timer
 - Timers.hpp, 389
- Timers, 105
 - ClockHigh, 105
 - ClockSteady, 105
 - endChronoHighTimer, 106
 - endChronoSteadyTimer, 106
 - endCpuTimer, 106
 - endMpiTimer, 106
 - startChronoHighTimer, 107
 - startChronoSteadyTimer, 107
 - startCpuTimer, 107
 - startMpiTimer, 107
- Timers.hpp, 389, 390
 - timer, 389
- TnsDims
 - ExprNode < _LabelSetSize, _ParLabelSetSize, _RootSize >, 78
 - I_TnsNode, 93
 - TnsNode < 0 >, 114
- TnsNode
 - TnsNode < 0 >, 112
 - TnsNode < _TnsSize >, 108
- TnsNode < 0 >, 111
 - DeltaSet, 112
 - Gramian, 112
 - IsNull, 115
 - Key, 113
 - LabelSet, 113
 - Left, 113
 - Parent, 113

- Right, [113](#)
- SearchKey, [114](#)
- SetOutdated, [114](#)
- Tensor_Type, [112](#)
- TensorX, [114](#)
- TnsDims, [114](#)
- TnsNode, [112](#)
- TnsSize, [115](#)
- Updated, [115](#)
- UpdateTree, [115](#)
- TnsNode< _TnsSize >, [107](#)
 - Gramian, [108](#)
 - IsNull, [110](#)
 - Key, [109](#)
 - mGramian, [110](#)
 - mKey, [110](#)
 - mTnsX, [110](#)
 - mUpdated, [110](#)
 - SetOutdated, [109](#)
 - TensorX, [109](#)
 - TnsNode, [108](#)
 - TnsSize, [110](#)
 - Updated, [109](#)
- TnsSize
 - ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >, [83](#)
 - I_TnsNode, [94](#)
 - Options< Tensor_, ExecutionPolicy_, DefaultValues_ >, [98](#)
 - TnsNode< 0 >, [115](#)
 - TnsNode< _TnsSize >, [110](#)
- TreeMode_N_Product
 - ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >, [78](#)
- TTVs
 - ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >, [79](#)
- TTVs_util
 - ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >, [79](#)
- unconstrained
 - Constants.hpp, [118](#)
- UpdateAlpha
 - NesterovMNLS.hpp, [303](#)
- Updated
 - I_TnsNode, [94](#)
 - TnsNode< 0 >, [115](#)
 - TnsNode< _TnsSize >, [109](#)
- UpdateTree
 - ExprNode< _LabelSetSize, _ParLabelSetSize, _RootSize >, [80](#)
 - I_TnsNode, [94](#)
 - TnsNode< 0 >, [115](#)
- write
 - ReadWrite.hpp, [356](#)
- writeToFile_append
 - ReadWrite.hpp, [356](#)
- ZeroTensorGen
 - TensorOperations.hpp, [377](#)