

Ginkgo

Generated from remotes/origin/develop branch based on develop. Ginkgo version 1.0.0

Generated by Doxygen 1.8.13

Contents

1	Main Page	1
2	Installation Instructions	3
3	Testing Instructions	7
4	Running the benchmarks	9
5	Example programs	11
6	The custom-logger program	15
7	The custom-matrix-format program	27
8	The custom-stopping-criterion program	35
9	The external-lib-interfacing program	41
10	The ginkgo-overhead program	71
11	The ginkgo-ranges program	75
12	The inverse-iteration program	79
13	The minimal-cuda-solver program	85
14	The nine-pt-stencil-solver program	89
15	The papi-logging program	99
16	The poisson-solver program	105

17 The preconditioned-solver program	111
18 The simple-solver program	115
19 The simple-solver-logging program	121
20 The three-pt-stencil-solver program	129
21 The twentyseven-pt-stencil-solver program	137
22 Module Documentation	149
22.1 CUDA Executor	149
22.1.1 Detailed Description	149
22.2 Executors	150
22.2.1 Detailed Description	150
22.2.2 Executors in Ginkgo.	151
22.2.3 Macro Definition Documentation	151
22.2.3.1 GKO_REGISTER_OPERATION	151
22.3 Factorizations	153
22.3.1 Detailed Description	153
22.4 Linear Operators	154
22.4.1 Detailed Description	155
22.4.2 Advantages of this approach and usage	156
22.4.3 Linear operator as a concept	156
22.4.4 Macro Definition Documentation	157
22.4.4.1 GKO_CREATE_FACTORY_PARAMETERS	157
22.4.4.2 GKO_ENABLE_BUILD_METHOD	157
22.4.4.3 GKO_ENABLE_LIN_OP_FACTORY	158
22.4.4.4 GKO_FACTORY_PARAMETER	160
22.4.5 Typedef Documentation	160
22.4.5.1 EnableDefaultLinOpFactory	160
22.5 Logging	162
22.5.1 Detailed Description	162

22.6 SpMV employing different Matrix formats	163
22.6.1 Detailed Description	163
22.6.2 Function Documentation	164
22.6.2.1 initialize() [1/4]	164
22.6.2.2 initialize() [2/4]	164
22.6.2.3 initialize() [3/4]	165
22.6.2.4 initialize() [4/4]	166
22.7 OpenMP Executor	168
22.7.1 Detailed Description	168
22.8 Preconditioners	169
22.8.1 Detailed Description	169
22.9 Reference Executor	170
22.9.1 Detailed Description	170
22.10 Solvers	171
22.10.1 Detailed Description	171
22.11 Stopping criteria	172
22.11.1 Detailed Description	172
22.11.2 Macro Definition Documentation	172
22.11.2.1 GKO_ENABLE_CRITERION_FACTORY	173
22.11.3 Function Documentation	173
22.11.3.1 combine()	173

23 Namespace Documentation	175
23.1 gko Namespace Reference	175
23.1.1 Detailed Description	182
23.1.2 Typedef Documentation	182
23.1.2.1 is_complex_s	182
23.1.3 Enumeration Type Documentation	182
23.1.3.1 layout_type	182
23.1.4 Function Documentation	183
23.1.4.1 abs()	183
23.1.4.2 as() [1/2]	183
23.1.4.3 as() [2/2]	184
23.1.4.4 ceildiv()	184
23.1.4.5 clone() [1/2]	185
23.1.4.6 clone() [2/2]	186
23.1.4.7 conj()	186
23.1.4.8 copy_and_convert_to() [1/2]	187
23.1.4.9 copy_and_convert_to() [2/2]	187
23.1.4.10 get_significant_bit()	188
23.1.4.11 get_superior_power()	189
23.1.4.12 give()	189
23.1.4.13 imag()	190
23.1.4.14 is_complex()	190
23.1.4.15 isfinite()	191
23.1.4.16 lend() [1/2]	191
23.1.4.17 lend() [2/2]	192
23.1.4.18 make_temporary_clone()	193
23.1.4.19 max()	193
23.1.4.20 min()	194
23.1.4.21 one() [1/2]	194
23.1.4.22 one() [2/2]	195

23.1.4.23 operator!=()	[1/3]	195
23.1.4.24 operator!=()	[2/3]	196
23.1.4.25 operator!=()	[3/3]	196
23.1.4.26 operator<<()	[1/2]	197
23.1.4.27 operator<<()	[2/2]	197
23.1.4.28 operator==()	[1/2]	198
23.1.4.29 operator==()	[2/2]	198
23.1.4.30 read()		199
23.1.4.31 read_raw()		199
23.1.4.32 real()		200
23.1.4.33 round_down()		200
23.1.4.34 round_up()		201
23.1.4.35 share()		202
23.1.4.36 squared_norm()		202
23.1.4.37 transpose()		203
23.1.4.38 write()		203
23.1.4.39 write_raw()		204
23.1.4.40 zero()	[1/2]	205
23.1.4.41 zero()	[2/2]	205
23.2 gko::accessor Namespace Reference		205
23.2.1 Detailed Description		206
23.3 gko::factorization Namespace Reference		206
23.3.1 Detailed Description		206
23.4 gko::log Namespace Reference		206
23.4.1 Detailed Description		207
23.5 gko::matrix Namespace Reference		207
23.5.1 Detailed Description		207
23.6 gko::name_demangling Namespace Reference		207
23.6.1 Detailed Description		208
23.6.2 Function Documentation		208

23.6.2.1	get_dynamic_type()	208
23.6.2.2	get_static_type()	208
23.7	gko::preconditioner Namespace Reference	209
23.7.1	Detailed Description	209
23.8	gko::solver Namespace Reference	209
23.8.1	Detailed Description	210
23.9	gko::stop Namespace Reference	210
23.9.1	Detailed Description	210
23.9.2	Typedef Documentation	211
23.9.2.1	EnableDefaultCriterionFactory	211
23.10	gko::syn Namespace Reference	211
23.10.1	Detailed Description	211
23.11	gko::xstd Namespace Reference	211
23.11.1	Detailed Description	211
24	Class Documentation	213
24.1	gko::AbstractFactory< AbstractProductType, ComponentsType > Class Template Reference	213
24.1.1	Detailed Description	213
24.1.2	Member Function Documentation	214
24.1.2.1	generate()	214
24.2	gko::AllocationError Class Reference	214
24.2.1	Detailed Description	215
24.2.2	Constructor & Destructor Documentation	215
24.2.2.1	AllocationError()	215
24.3	gko::Array< ValueType > Class Template Reference	215
24.3.1	Detailed Description	217
24.3.2	Constructor & Destructor Documentation	217
24.3.2.1	Array() [1/11]	217
24.3.2.2	Array() [2/11]	217
24.3.2.3	Array() [3/11]	218
24.3.2.4	Array() [4/11]	218

24.3.2.5	Array() [5/11]	219
24.3.2.6	Array() [6/11]	219
24.3.2.7	Array() [7/11]	220
24.3.2.8	Array() [8/11]	221
24.3.2.9	Array() [9/11]	221
24.3.2.10	Array() [10/11]	221
24.3.2.11	Array() [11/11]	222
24.3.3	Member Function Documentation	222
24.3.3.1	clear()	222
24.3.3.2	get_const_data()	223
24.3.3.3	get_data()	223
24.3.3.4	get_executor()	224
24.3.3.5	get_num_elems()	224
24.3.3.6	operator=() [1/2]	224
24.3.3.7	operator=() [2/2]	225
24.3.3.8	resize_and_reset()	226
24.3.3.9	set_executor()	226
24.3.3.10	view()	227
24.4	gko::matrix::Hybrid< ValueType, IndexType >::automatic Class Reference	227
24.4.1	Detailed Description	228
24.4.2	Member Function Documentation	228
24.4.2.1	compute_ell_num_stored_elements_per_row()	228
24.5	gko::solver::Bicgstab< ValueType > Class Template Reference	229
24.5.1	Detailed Description	229
24.5.2	Member Function Documentation	229
24.5.2.1	get_preconditioner()	229
24.5.2.2	get_system_matrix()	230
24.6	gko::preconditioner::block_interleaved_storage_scheme< IndexType > Struct Template Reference	230
24.6.1	Detailed Description	231
24.6.2	Member Function Documentation	231

24.6.2.1	<code>compute_storage_space()</code>	231
24.6.2.2	<code>get_block_offset()</code>	232
24.6.2.3	<code>get_global_block_offset()</code>	232
24.6.2.4	<code>get_group_offset()</code>	233
24.6.2.5	<code>get_group_size()</code>	233
24.6.2.6	<code>get_stride()</code>	234
24.6.3	Member Data Documentation	234
24.6.3.1	<code>group_power</code>	234
24.7	<code>gko::solver::Cg< ValueType ></code> Class Template Reference	234
24.7.1	Detailed Description	235
24.7.2	Member Function Documentation	235
24.7.2.1	<code>get_preconditioner()</code>	235
24.7.2.2	<code>get_system_matrix()</code>	236
24.8	<code>gko::solver::Cgs< ValueType ></code> Class Template Reference	236
24.8.1	Detailed Description	236
24.8.2	Member Function Documentation	237
24.8.2.1	<code>get_preconditioner()</code>	237
24.8.2.2	<code>get_system_matrix()</code>	237
24.9	<code>gko::matrix::Hybrid< ValueType, IndexType >::column_limit</code> Class Reference	238
24.9.1	Detailed Description	238
24.9.2	Constructor & Destructor Documentation	238
24.9.2.1	<code>column_limit()</code>	238
24.9.3	Member Function Documentation	238
24.9.3.1	<code>compute_ell_num_stored_elements_per_row()</code>	239
24.10	<code>gko::Combination< ValueType ></code> Class Template Reference	239
24.10.1	Detailed Description	239
24.10.2	Member Function Documentation	240
24.10.2.1	<code>get_coefficients()</code>	240
24.10.2.2	<code>get_operators()</code>	240
24.11	<code>gko::stop::Combined</code> Class Reference	241

24.11.1 Detailed Description	241
24.12gko::Composition< ValueType > Class Template Reference	241
24.12.1 Detailed Description	241
24.12.2 Member Function Documentation	242
24.12.2.1 get_operators()	242
24.13gko::log::Convergence< ValueType > Class Template Reference	242
24.13.1 Detailed Description	243
24.13.2 Member Function Documentation	243
24.13.2.1 create()	243
24.13.2.2 get_num_iterations()	243
24.13.2.3 get_residual()	244
24.13.2.4 get_residual_norm()	244
24.14gko::ConvertibleTo< ResultType > Class Template Reference	244
24.14.1 Detailed Description	245
24.14.2 Member Function Documentation	245
24.14.2.1 convert_to()	245
24.14.2.2 move_to()	246
24.15gko::matrix::Coo< ValueType, IndexType > Class Template Reference	247
24.15.1 Detailed Description	247
24.15.2 Member Function Documentation	248
24.15.2.1 apply2() [1 / 4]	248
24.15.2.2 apply2() [2 / 4]	248
24.15.2.3 apply2() [3 / 4]	249
24.15.2.4 apply2() [4 / 4]	249
24.15.2.5 get_col_idxxs()	250
24.15.2.6 get_const_col_idxxs()	250
24.15.2.7 get_const_row_idxxs()	251
24.15.2.8 get_const_values()	251
24.15.2.9 get_num_stored_elements()	252
24.15.2.10get_row_idxxs()	252

24.15.2.11	get_values()	252
24.15.2.12	read()	252
24.15.2.13	write()	253
24.16	gko::copy_back_deleter< T > Class Template Reference	253
24.16.1	Detailed Description	253
24.16.2	Constructor & Destructor Documentation	254
24.16.2.1	copy_back_deleter()	254
24.16.3	Member Function Documentation	254
24.16.3.1	operator()()	254
24.17	gko::stop::Criterion Class Reference	255
24.17.1	Detailed Description	255
24.17.2	Member Function Documentation	255
24.17.2.1	check()	255
24.17.2.2	update()	256
24.18	gko::log::criterion_data Struct Reference	256
24.18.1	Detailed Description	257
24.19	gko::stop::CriterionArgs Struct Reference	257
24.19.1	Detailed Description	257
24.20	gko::matrix::Csr< ValueType, IndexType > Class Template Reference	257
24.20.1	Detailed Description	258
24.20.2	Member Function Documentation	259
24.20.2.1	conj_transpose()	259
24.20.2.2	get_col_idxxs()	259
24.20.2.3	get_const_col_idxxs()	259
24.20.2.4	get_const_row_ptrs()	260
24.20.2.5	get_const_srow()	260
24.20.2.6	get_const_values()	261
24.20.2.7	get_num_srow_elements()	261
24.20.2.8	get_num_stored_elements()	261
24.20.2.9	get_row_ptrs()	262

24.20.2.10	get_srow()	262
24.20.2.11	get_strategy()	262
24.20.2.12	get_values()	263
24.20.2.13	read()	263
24.20.2.14	transpose()	263
24.20.2.15	write()	263
24.21	gko::CublasError Class Reference	264
24.21.1	Detailed Description	264
24.21.2	Constructor & Destructor Documentation	264
24.21.2.1	CublasError()	264
24.22	gko::CudaError Class Reference	265
24.22.1	Detailed Description	265
24.22.2	Constructor & Destructor Documentation	265
24.22.2.1	CudaError()	265
24.23	gko::CudaExecutor Class Reference	266
24.23.1	Detailed Description	266
24.23.2	Member Function Documentation	267
24.23.2.1	create()	267
24.23.2.2	get_cublas_handle()	267
24.23.2.3	get_cuspars_handle()	267
24.23.2.4	get_master() [1/2]	268
24.23.2.5	get_master() [2/2]	268
24.23.2.6	run()	268
24.24	gko::CusparsError Class Reference	268
24.24.1	Detailed Description	269
24.24.2	Constructor & Destructor Documentation	269
24.24.2.1	CusparsError()	269
24.25	gko::default_converter< S, R > Struct Template Reference	269
24.25.1	Detailed Description	270
24.25.2	Member Function Documentation	270

24.25.2.1 operator()	270
24.26gko::matrix::Dense< ValueType > Class Template Reference	270
24.26.1 Detailed Description	271
24.26.2 Member Function Documentation	272
24.26.2.1 add_scaled()	272
24.26.2.2 at() [1/4]	272
24.26.2.3 at() [2/4]	273
24.26.2.4 at() [3/4]	274
24.26.2.5 at() [4/4]	274
24.26.2.6 compute_dot()	275
24.26.2.7 compute_norm2()	275
24.26.2.8 conj_transpose()	276
24.26.2.9 create_submatrix()	276
24.26.2.10create_with_config_of()	277
24.26.2.11get_const_values()	277
24.26.2.12get_num_stored_elements()	278
24.26.2.13get_stride()	278
24.26.2.14get_values()	278
24.26.2.15scale()	278
24.26.2.16transpose()	279
24.27gko::dim< Dimensionality, DimensionType > Struct Template Reference	279
24.27.1 Detailed Description	280
24.27.2 Constructor & Destructor Documentation	280
24.27.2.1 dim() [1/2]	280
24.27.2.2 dim() [2/2]	280
24.27.3 Member Function Documentation	281
24.27.3.1 operator bool()	281
24.27.3.2 operator[]() [1/2]	281
24.27.3.3 operator[]() [2/2]	282
24.27.4 Friends And Related Function Documentation	282

24.27.4.1 operator*	282
24.27.4.2 operator==	283
24.28gko::DimensionMismatch Class Reference	283
24.28.1 Detailed Description	283
24.28.2 Constructor & Destructor Documentation	283
24.28.2.1 DimensionMismatch()	284
24.29gko::matrix::Ell< ValueType, IndexType > Class Template Reference	284
24.29.1 Detailed Description	285
24.29.2 Member Function Documentation	286
24.29.2.1 col_at() [1/2]	286
24.29.2.2 col_at() [2/2]	286
24.29.2.3 get_col_idxes()	287
24.29.2.4 get_const_col_idxes()	287
24.29.2.5 get_const_values()	288
24.29.2.6 get_num_stored_elements()	288
24.29.2.7 get_num_stored_elements_per_row()	289
24.29.2.8 get_stride()	289
24.29.2.9 get_values()	289
24.29.2.10read()	289
24.29.2.11val_at() [1/2]	290
24.29.2.12val_at() [2/2]	290
24.29.2.13write()	291
24.30gko::enable_parameters_type< ConcreteParametersType, Factory > Struct Template Reference	291
24.30.1 Detailed Description	291
24.30.2 Member Function Documentation	292
24.30.2.1 on()	292
24.31gko::EnableAbstractPolymorphicObject< AbstractObject, PolymorphicBase > Class Template Reference	292
24.31.1 Detailed Description	292
24.32gko::EnableCreateMethod< ConcreteType > Class Template Reference	293
24.32.1 Detailed Description	293

24.33	gko::EnableDefaultFactory< ConcreteFactory, ProductType, ParametersType, PolymorphicBase > Class Template Reference	293
24.33.1	Detailed Description	294
24.33.2	Member Function Documentation	294
24.33.2.1	create()	294
24.33.2.2	get_parameters()	295
24.34	gko::EnableLinOp< ConcreteLinOp, PolymorphicBase > Class Template Reference	295
24.34.1	Detailed Description	295
24.35	gko::log::EnableLogging< ConcreteLoggable, PolymorphicBase > Class Template Reference	296
24.35.1	Detailed Description	296
24.35.2	Member Function Documentation	296
24.35.2.1	add_logger()	297
24.35.2.2	remove_logger()	297
24.36	gko::EnablePolymorphicAssignment< ConcreteType, ResultType > Class Template Reference	298
24.36.1	Detailed Description	298
24.36.2	Member Function Documentation	298
24.36.2.1	convert_to()	298
24.36.2.2	move_to()	299
24.37	gko::EnablePolymorphicObject< ConcreteObject, PolymorphicBase > Class Template Reference	299
24.37.1	Detailed Description	300
24.38	gko::Error Class Reference	300
24.38.1	Detailed Description	301
24.38.2	Constructor & Destructor Documentation	301
24.38.2.1	Error()	301
24.39	gko::Executor Class Reference	302
24.39.1	Detailed Description	302
24.39.2	Member Function Documentation	303
24.39.2.1	alloc()	303
24.39.2.2	copy_from()	304
24.39.2.3	free()	305
24.39.2.4	get_master() [1/2]	305

24.39.2.5 <code>get_master()</code> [2/2]	305
24.39.2.6 <code>run()</code> [1/2]	306
24.39.2.7 <code>run()</code> [2/2]	306
24.40 <code>gko::log::executor_data</code> Struct Reference	306
24.40.1 Detailed Description	307
24.41 <code>gko::executor_deleter< T ></code> Class Template Reference	307
24.41.1 Detailed Description	307
24.41.2 Constructor & Destructor Documentation	307
24.41.2.1 <code>executor_deleter()</code>	307
24.41.3 Member Function Documentation	308
24.41.3.1 <code>operator()()</code>	308
24.42 <code>gko::solver::Fcg< ValueType ></code> Class Template Reference	308
24.42.1 Detailed Description	309
24.42.2 Member Function Documentation	309
24.42.2.1 <code>get_preconditioner()</code>	309
24.42.2.2 <code>get_system_matrix()</code>	310
24.43 <code>gko::solver::Gmres< ValueType ></code> Class Template Reference	310
24.43.1 Detailed Description	310
24.43.2 Member Function Documentation	311
24.43.2.1 <code>get_krylov_dim()</code>	311
24.43.2.2 <code>get_preconditioner()</code>	311
24.43.2.3 <code>get_system_matrix()</code>	312
24.44 <code>gko::matrix::Hybrid< ValueType, IndexType ></code> Class Template Reference	312
24.44.1 Detailed Description	314
24.44.2 Member Function Documentation	314
24.44.2.1 <code>ell_col_at()</code> [1/2]	314
24.44.2.2 <code>ell_col_at()</code> [2/2]	314
24.44.2.3 <code>ell_val_at()</code> [1/2]	315
24.44.2.4 <code>ell_val_at()</code> [2/2]	315
24.44.2.5 <code>get_const_coo_col_idxs()</code>	316

24.44.2.6 <code>get_const_coo_row_idxs()</code>	316
24.44.2.7 <code>get_const_coo_values()</code>	317
24.44.2.8 <code>get_const_ell_col_idxs()</code>	317
24.44.2.9 <code>get_const_ell_values()</code>	318
24.44.2.10 <code>get_coo()</code>	318
24.44.2.11 <code>get_coo_col_idxs()</code>	318
24.44.2.12 <code>get_coo_num_stored_elements()</code>	319
24.44.2.13 <code>get_coo_row_idxs()</code>	319
24.44.2.14 <code>get_coo_values()</code>	319
24.44.2.15 <code>get_ell()</code>	320
24.44.2.16 <code>get_ell_col_idxs()</code>	320
24.44.2.17 <code>get_ell_num_stored_elements()</code>	320
24.44.2.18 <code>get_ell_num_stored_elements_per_row()</code>	321
24.44.2.19 <code>get_ell_stride()</code>	321
24.44.2.20 <code>get_ell_values()</code>	321
24.44.2.21 <code>get_num_stored_elements()</code>	322
24.44.2.22 <code>get_strategy()</code>	322
24.44.2.23 <code>operator=()</code>	322
24.44.2.24 <code>read()</code>	323
24.44.2.25 <code>write()</code>	323
24.45 <code>gko::matrix::Identity< ValueType ></code> Class Template Reference	324
24.45.1 Detailed Description	324
24.46 <code>gko::matrix::IdentityFactory< ValueType ></code> Class Template Reference	324
24.46.1 Detailed Description	325
24.46.2 Member Function Documentation	325
24.46.2.1 <code>create()</code>	325
24.47 <code>gko::matrix::Hybrid< ValueType, IndexType >::imbalance_bounded_limit</code> Class Reference	325
24.47.1 Detailed Description	326
24.47.2 Member Function Documentation	326
24.47.2.1 <code>compute_ell_num_stored_elements_per_row()</code>	326

24.48	gko::matrix::Hybrid< ValueType, IndexType >::imbalance_limit Class Reference	327
24.48.1	Detailed Description	327
24.48.2	Constructor & Destructor Documentation	327
24.48.2.1	imbalance_limit()	327
24.48.3	Member Function Documentation	328
24.48.3.1	compute_ell_num_stored_elements_per_row()	328
24.49	gko::solver::lr< ValueType > Class Template Reference	328
24.49.1	Detailed Description	329
24.49.2	Member Function Documentation	329
24.49.2.1	get_solver()	329
24.49.2.2	get_system_matrix()	330
24.50	gko::stop::iteration Class Reference	330
24.50.1	Detailed Description	330
24.51	gko::log::iteration_complete_data Struct Reference	331
24.51.1	Detailed Description	331
24.52	gko::preconditioner::Jacobi< ValueType, IndexType > Class Template Reference	331
24.52.1	Detailed Description	332
24.52.2	Member Function Documentation	332
24.52.2.1	convert_to()	332
24.52.2.2	get_blocks()	333
24.52.2.3	get_conditioning()	333
24.52.2.4	get_num_blocks()	334
24.52.2.5	get_num_stored_elements()	334
24.52.2.6	get_storage_scheme()	334
24.52.2.7	move_to()	335
24.52.2.8	write()	336
24.53	gko::KernelNotFound Class Reference	336
24.53.1	Detailed Description	336
24.53.2	Constructor & Destructor Documentation	337
24.53.2.1	KernelNotFound()	337

24.54	gko::log::linop_data Struct Reference	337
24.54.1	Detailed Description	337
24.55	gko::log::linop_factory_data Struct Reference	337
24.55.1	Detailed Description	338
24.56	gko::LinOpFactory Class Reference	338
24.56.1	Detailed Description	338
24.57	gko::log::Loggable Class Reference	339
24.57.1	Detailed Description	339
24.57.2	Member Function Documentation	339
24.57.2.1	add_logger()	339
24.57.2.2	remove_logger()	340
24.58	gko::log::Record::logged_data Struct Reference	340
24.58.1	Detailed Description	340
24.59	gko::matrix_data< ValueType, IndexType > Struct Template Reference	341
24.59.1	Detailed Description	342
24.59.2	Constructor & Destructor Documentation	342
24.59.2.1	matrix_data() [1/6]	342
24.59.2.2	matrix_data() [2/6]	343
24.59.2.3	matrix_data() [3/6]	343
24.59.2.4	matrix_data() [4/6]	344
24.59.2.5	matrix_data() [5/6]	344
24.59.2.6	matrix_data() [6/6]	345
24.59.3	Member Function Documentation	345
24.59.3.1	cond() [1/2]	346
24.59.3.2	cond() [2/2]	346
24.59.3.3	diag() [1/5]	347
24.59.3.4	diag() [2/5]	348
24.59.3.5	diag() [3/5]	349
24.59.3.6	diag() [4/5]	350
24.59.3.7	diag() [5/5]	351

24.59.4 Member Data Documentation	351
24.59.4.1 nonzeros	351
24.60gko::matrix::Hybrid< ValueType, IndexType >::minimal_storage_limit Class Reference	352
24.60.1 Detailed Description	352
24.60.2 Member Function Documentation	352
24.60.2.1 compute_ell_num_stored_elements_per_row()	352
24.61gko::matrix_data< ValueType, IndexType >::nonzero_type Struct Reference	353
24.61.1 Detailed Description	353
24.62gko::NotCompiled Class Reference	353
24.62.1 Detailed Description	353
24.62.2 Constructor & Destructor Documentation	353
24.62.2.1 NotCompiled()	353
24.63gko::NotImplemented Class Reference	354
24.63.1 Detailed Description	354
24.63.2 Constructor & Destructor Documentation	354
24.63.2.1 NotImplemented()	354
24.64gko::NotSupported Class Reference	355
24.64.1 Detailed Description	355
24.64.2 Constructor & Destructor Documentation	355
24.64.2.1 NotSupported()	355
24.65gko::null_deleter< T > Class Template Reference	356
24.65.1 Detailed Description	356
24.65.2 Member Function Documentation	356
24.65.2.1 operator()()	356
24.66gko::OmpExecutor Class Reference	357
24.66.1 Detailed Description	357
24.66.2 Member Function Documentation	357
24.66.2.1 get_master() [1/2]	358
24.66.2.2 get_master() [2/2]	358
24.67gko::Operation Class Reference	358

24.67.1 Detailed Description	359
24.67.2 Member Function Documentation	360
24.67.2.1 get_name()	360
24.68gko::log::operation_data Struct Reference	360
24.68.1 Detailed Description	360
24.69gko::OutOfBoundsError Class Reference	360
24.69.1 Detailed Description	361
24.69.2 Constructor & Destructor Documentation	361
24.69.2.1 OutOfBoundsError()	361
24.70gko::factorization::Parllu< ValueType, IndexType > Class Template Reference	361
24.70.1 Detailed Description	362
24.71gko::log::polymorphic_object_data Struct Reference	362
24.71.1 Detailed Description	362
24.72gko::PolymorphicObject Class Reference	363
24.72.1 Detailed Description	363
24.72.2 Member Function Documentation	363
24.72.2.1 clear()	364
24.72.2.2 clone() [1/2]	364
24.72.2.3 clone() [2/2]	365
24.72.2.4 copy_from() [1/2]	365
24.72.2.5 copy_from() [2/2]	366
24.72.2.6 create_default() [1/2]	366
24.72.2.7 create_default() [2/2]	367
24.72.2.8 get_executor()	367
24.73gko::precision_reduction Class Reference	368
24.73.1 Detailed Description	368
24.73.2 Constructor & Destructor Documentation	369
24.73.2.1 precision_reduction() [1/2]	369
24.73.2.2 precision_reduction() [2/2]	369
24.73.3 Member Function Documentation	370

24.73.3.1 autodetect()	370
24.73.3.2 common()	370
24.73.3.3 get_nonpreserving()	371
24.73.3.4 get_preserving()	371
24.73.3.5 operator storage_type()	371
24.74gko::Preconditionable Class Reference	372
24.74.1 Detailed Description	372
24.74.2 Member Function Documentation	372
24.74.2.1 get_preconditioner()	372
24.75gko::range< Accessor > Class Template Reference	372
24.75.1 Detailed Description	373
24.75.2 Constructor & Destructor Documentation	376
24.75.2.1 range()	376
24.75.3 Member Function Documentation	376
24.75.3.1 get_accessor()	376
24.75.3.2 length()	377
24.75.3.3 operator>()	377
24.75.3.4 operator->()	378
24.75.3.5 operator=() [1/2]	378
24.75.3.6 operator=() [2/2]	378
24.76gko::ReadableFromMatrixData< ValueType, IndexType > Class Template Reference	379
24.76.1 Detailed Description	379
24.76.2 Member Function Documentation	380
24.76.2.1 read()	380
24.77gko::log::Record Class Reference	381
24.77.1 Detailed Description	381
24.77.2 Member Function Documentation	382
24.77.2.1 create()	382
24.77.2.2 get() [1/2]	382
24.77.2.3 get() [2/2]	383

24.78	gko::ReferenceExecutor Class Reference	383
24.78.1	Detailed Description	383
24.78.2	Member Function Documentation	383
24.78.2.1	run()	383
24.79	gko::stop::ResidualNormReduction< ValueType > Class Template Reference	384
24.79.1	Detailed Description	384
24.80	gko::accessor::row_major< ValueType, Dimensionality > Class Template Reference	384
24.80.1	Detailed Description	385
24.80.2	Member Function Documentation	386
24.80.2.1	copy_from()	386
24.80.2.2	length()	386
24.80.2.3	operator>() [1/2]	387
24.80.2.4	operator>() [2/2]	387
24.81	gko::matrix::Sellp< ValueType, IndexType > Class Template Reference	388
24.81.1	Detailed Description	389
24.81.2	Member Function Documentation	389
24.81.2.1	col_at() [1/2]	389
24.81.2.2	col_at() [2/2]	390
24.81.2.3	get_col_idxes()	391
24.81.2.4	get_const_col_idxes()	391
24.81.2.5	get_const_slice_lengths()	392
24.81.2.6	get_const_slice_sets()	392
24.81.2.7	get_const_values()	393
24.81.2.8	get_num_stored_elements()	393
24.81.2.9	get_slice_lengths()	394
24.81.2.10	get_slice_sets()	394
24.81.2.11	get_slice_size()	394
24.81.2.12	get_stride_factor()	395
24.81.2.13	get_total_cols()	395
24.81.2.14	get_values()	395

24.81.2.15	<code>read()</code>	395
24.81.2.16	<code>val_at()</code> [1/2]	396
24.81.2.17	<code>val_at()</code> [2/2]	396
24.81.2.18	<code>write()</code>	397
24.82	<code>gko::span</code> Struct Reference	397
24.82.1	Detailed Description	398
24.82.2	Constructor & Destructor Documentation	398
24.82.2.1	<code>span()</code> [1/2]	398
24.82.2.2	<code>span()</code> [2/2]	399
24.82.3	Member Function Documentation	399
24.82.3.1	<code>is_valid()</code>	399
24.83	<code>gko::stopping_status</code> Class Reference	399
24.83.1	Detailed Description	400
24.83.2	Member Function Documentation	400
24.83.2.1	<code>converge()</code>	400
24.83.2.2	<code>get_id()</code>	401
24.83.2.3	<code>has_converged()</code>	401
24.83.2.4	<code>has_stopped()</code>	402
24.83.2.5	<code>is_finalized()</code>	402
24.83.2.6	<code>stop()</code>	402
24.83.3	Friends And Related Function Documentation	403
24.83.3.1	<code>operator!=</code>	403
24.83.3.2	<code>operator==</code>	403
24.84	<code>gko::matrix::Hybrid< ValueType, IndexType >::strategy_type</code> Class Reference	404
24.84.1	Detailed Description	404
24.84.2	Member Function Documentation	404
24.84.2.1	<code>compute_ell_num_stored_elements_per_row()</code>	404
24.84.2.2	<code>compute_hybrid_config()</code>	405
24.84.2.3	<code>get_coo_nnz()</code>	406
24.84.2.4	<code>get_ell_num_stored_elements_per_row()</code>	406

24.85	gko::log::Stream< ValueType > Class Template Reference	406
24.85.1	Detailed Description	406
24.85.2	Member Function Documentation	407
24.85.2.1	create()	407
24.86	gko::StreamError Class Reference	407
24.86.1	Detailed Description	408
24.86.2	Constructor & Destructor Documentation	408
24.86.2.1	StreamError()	408
24.87	gko::temporary_clone< T > Class Template Reference	408
24.87.1	Detailed Description	409
24.87.2	Constructor & Destructor Documentation	409
24.87.2.1	temporary_clone()	409
24.87.3	Member Function Documentation	410
24.87.3.1	get()	410
24.87.3.2	operator->()	410
24.88	gko::stop::Time Class Reference	410
24.88.1	Detailed Description	411
24.89	gko::Transposable Class Reference	411
24.89.1	Detailed Description	411
24.89.2	Member Function Documentation	411
24.89.2.1	conj_transpose()	412
24.89.2.2	transpose()	412
24.90	gko::stop::Criterion::Updater Class Reference	412
24.90.1	Detailed Description	413
24.90.2	Member Function Documentation	413
24.90.2.1	check()	413
24.91	gko::version Struct Reference	413
24.91.1	Detailed Description	414
24.91.2	Member Data Documentation	414
24.91.2.1	tag	414
24.92	gko::version_info Class Reference	414
24.92.1	Detailed Description	415
24.92.2	Member Function Documentation	415
24.92.2.1	get()	415
24.92.3	Member Data Documentation	415
24.92.3.1	core_version	415
24.92.3.2	cuda_version	416
24.92.3.3	omp_version	416
24.92.3.4	reference_version	416
24.93	gko::WritableToMatrixData< ValueType, IndexType > Class Template Reference	416
24.93.1	Detailed Description	416
24.93.2	Member Function Documentation	416
24.93.2.1	write()	416

Chapter 1

Main Page

This is the main page for the Ginkgo library pdf documentation. The repository is hosted on [github](#). Documentation on aspects such as the build system, can be found at the [Installation Instructions](#) page. The [Example programs](#) can help you get started with using Ginkgo.

Modules

The Ginkgo library can be grouped into [modules](#) and these modules form the basic building blocks of Ginkgo. The modules can be summarized as follows:

- [Executors](#) : Where do you want your code to be executed ?
- [Linear Operators](#) : What kind of operation do you want Ginkgo to perform ?
 - [Solvers](#) : Solve a linear system for a given matrix.
 - [Preconditioners](#) : Precondition a system for a solve.
 - [SpMV employing different Matrix formats](#) : Perform a sparse matrix vector multiplication with a particular matrix format.
- [Logging](#) : Monitor your code execution.
- [Stopping criteria](#) : Manage your iteration stopping criteria.

Chapter 2

Installation Instructions

Building

Use the standard cmake build procedure:

```
mkdir build; cd build
cmake -G "Unix Makefiles" [OPTIONS] .. && make
```

Replace [OPTIONS] with desired cmake options for your build. Ginkgo adds the following additional switches to control what is being built:

- `-DGINKGO_DEVEL_TOOLS={ON, OFF}` sets up the build system for development (requires clang-format, will also download git-cmake-format), default is ON
- `-DGINKGO_BUILD_TESTS={ON, OFF}` builds Ginkgo's tests (will download googletest), default is ON
- `-DGINKGO_BUILD_BENCHMARKS={ON, OFF}` builds Ginkgo's benchmarks (will download gflags and rapidjson), default is ON
- `-DGINKGO_BUILD_EXAMPLES={ON, OFF}` builds Ginkgo's examples, default is ON
- `-DGINKGO_BUILD_EXTLIB_EXAMPLE={ON, OFF}` builds the interfacing example with deal.II, default is OFF
- `-DGINKGO_BUILD_REFERENCE={ON, OFF}` build reference implementations of the kernels, useful for testing, default is ON
- `-DGINKGO_BUILD_OMP={ON, OFF}` builds optimized OpenMP versions of the kernels, default is OFF
- `-DGINKGO_BUILD_CUDA={ON, OFF}` builds optimized cuda versions of the kernels (requires CUDA), default is OFF
- `-DGINKGO_BUILD_DOC={ON, OFF}` creates an HTML version of Ginkgo's documentation from inline comments in the code. The default is OFF.
- `-DGINKGO_DOC_GENERATE_EXAMPLES={ON, OFF}` generates the documentation of examples in Ginkgo. The default is ON.
- `-DGINKGO_DOC_GENERATE_PDF={ON, OFF}` generates a PDF version of Ginkgo's documentation from inline comments in the code. The default is OFF.
- `-DGINKGO_DOC_GENERATE_DEV={ON, OFF}` generates the developer version of Ginkgo's documentation. The default is OFF.

- `-DGINKGO_EXPORT_BUILD_DIR={ON, OFF}` adds the Ginkgo build directory to the CMake package registry. The default is `OFF`.
- `-DGINKGO_WITH_CLANG_TIDY={ON, OFF}` makes Ginkgo call `clang-tidy` to find programming issues. The path can be manually controlled with the CMake variable `-DGINKGO_CLANG_TIDY_PATH=<path>`.
- `-DGINKGO_WITH_IWYU={ON, OFF}` makes Ginkgo call `iwyu` to find include issues. The path can be manually controlled with the CMake variable `-DGINKGO_IWYU_PATH=<path>`.
- `-DGINKGO_VERBOSE_LEVEL=integer` sets the verbosity of Ginkgo.
 - 0 disables all output in the main libraries,
 - 1 enables a few important messages related to unexpected behavior (default).
- `-DCMAKE_INSTALL_PREFIX=path` sets the installation path for `make install`. The default value is usually something like `/usr/local`
- `-DCMAKE_BUILD_TYPE=type` specifies which configuration will be used for this build of Ginkgo. The default is `RELEASE`. Supported values are CMake's standard build types such as `DEBUG` and `RELEASE` and the Ginkgo specific `COVERAGE`, `ASAN` (AddressSanitizer) and `TSAN` (ThreadSanitizer) types.
- `-DBUILD_SHARED_LIBS={ON, OFF}` builds ginkgo as shared libraries (`OFF`) or as dynamic libraries (`ON`), default is `ON`
- `-DGINKGO_JACOBI_FULL_OPTIMIZATIONS={ON, OFF}` use all the optimizations for the CUDA Jacobi algorithm. `OFF` by default. Setting this option to `ON` may lead to very slow compile time (>20 minutes) for the `jacobi_generate_kernels.cu` file and high memory usage.
- `-DCMAKE_CUDA_HOST_COMPILER=path` instructs the build system to explicitly set CUDA's host compiler to the path given as argument. By default, CUDA uses its toolchain's host compiler. Setting this option may help if you're experiencing linking errors due to ABI incompatibilities. This option is supported since [CMake 3.8](#) but [documented starting from 3.10](#).
- `-DGINKGO_CUDA_ARCHITECTURES=<list>` where `<list>` is a semicolon (;) separated list of architectures. Supported values are:
 - Auto
 - Kepler,Maxwell,Pascal,Volta
 - CODE, CODE (COMPUTE), (COMPUTE)

Auto will automatically detect the present CUDA-enabled GPU architectures in the system. Kepler, Maxwell, Pascal and Volta will add flags for all architectures of that particular NVIDIA GPU generation. COMPUTE and CODE are placeholders that should be replaced with compute and code numbers (e.g. for `compute_70` and `sm_70` COMPUTE and CODE should be replaced with 70. Default is Auto. For a more detailed explanation of this option see the [ARCHITECTURES specification list](#) section in the documentation of the `CudaArchitectureSelector` CMake module.

For example, to build everything (in debug mode), use:

```
cmake -G "Unix Makefiles" -H. -BDebug -DCMAKE_BUILD_TYPE=Debug -DGINKGO_DEVEL_TOOLS=ON \
      -DGINKGO_BUILD_TESTS=ON -DGINKGO_BUILD_REFERENCE=ON -DGINKGO_BUILD_OMP=ON \
      -DGINKGO_BUILD_CUDA=ON
cmake --build Debug
```

NOTE: Ginkgo is known to work with the `Unix Makefiles` and `Ninja` based generators. Other CMake generators are untested.

Third party libraries and packages

Ginkgo relies on third party packages in different cases. These third party packages can be turned off by disabling the relevant options.

- GINKGO_BUILD_CUDA=ON: [CudaArchitectureSelector](#) (CAS) is a CMake helper to manage C↵UDA architecture settings;
- GINKGO_BUILD_TESTS=ON: Our tests are implemented with [Google Test](#);
- GINKGO_BUILD_BENCHMARKS=ON: For argument management we use [gflags](#) and for JSON parsing we use [RapidJSON](#);
- GINKGO_DEVEL_TOOLS=ON: [git-cmake-format](#) is our CMake helper for code formatting.

By default, Ginkgo uses the internal version of each package. For each of the packages `GTEST`, `GFLAGS`, `R↵APIDJSON` and `CAS`, it is possible to force Ginkgo to try to use an external version of a package. For this, Ginkgo provides two ways to find packages. To rely on the CMake `find_package` command, use the CMake option `-DGINKGO_USE_EXTERNAL_<package>=ON`. Note that, if the external packages were not installed to the default location, the CMake option `-DCMAKE_PREFIX_PATH=<path-list>` needs to be set to the semicolon (;) separated list of install paths of these external packages. For more information, see the [CMake documentation for CMAKE_PREFIX_PATH](#) for details.

To manually configure the paths Ginkgo relies on the [standard xSDK Installation policies](#) for all packages except `CAS` (as it is neither a library nor a header, it cannot be expressed through the TPL format):

- `-DTPL_ENABLE_<package>=ON`
- `-DTPL_<package>_LIBRARIES=/path/to/libraries.{so|a}`
- `-DTPL_<package>_INCLUDE_DIRS=/path/to/header/directory`

When applicable (e.g. for `GTest` libraries), a ; separated list can be given to the `TPL_<package>_{LIBR↵ARIES|INCLUDE_DIRS}` variables.

Installing Ginkgo

To install Ginkgo into the specified folder, execute the following command in the build folder

```
make install
```

If the installation prefix (see `CMAKE_INSTALL_PREFIX`) is not writable for your user, e.g. when installing Ginkgo system-wide, it might be necessary to prefix the call with `sudo`.

After the installation, CMake can find ginkgo with `find_package(Ginkgo)`. An example can be found in the [test_install](#).

Chapter 3

Testing Instructions

Running the unit tests

You need to compile ginkgo with `-DGINKGO_BUILD_TESTS=ON` option to be able to run the tests.

Using make test

After configuring Ginkgo, use the following command inside the build folder to run all tests:

```
make test
```

The output should contain several lines of the form:

```
      Start   1: path/to/test
1/13 Test   #1: path/to/test ..... Passed    0.01 sec
```

To run only a specific test and see more details results (e.g. if a test failed) run the following from the build folder:

```
./path/to/test
```

where `path/to/test` is the path returned by `make test`.

Using CTest

The tests can also be ran through CTest from the command line, for example when in a configured build directory:

```
ctest -T start -T build -T test -T submit
```

Will start a new test campaign (usually in `Experimental` mode), build Ginkgo with the set configuration, run the tests and submit the results to our CDash dashboard.

Another option is to use Ginkgo's CTest script which is configured to build Ginkgo with default settings, runs the tests and submits the test to our CDash dashboard automatically.

To run the script, use the following command:

```
ctest -S cmake/CTestScript.cmake
```

The default settings are for our own CI system. Feel free to configure the script before launching it through variables or by directly changing its values. A documentation can be found in the script itself.

Chapter 4

Running the benchmarks

In addition to the unit tests designed to verify correctness, Ginkgo also includes a benchmark suite for checking its performance on the system. To compile the benchmarks, the flag `-DGINKGO_BUILD_BENCHMARKS=ON` has to be set during the `cmake` step. In addition, the `ssget` command-line utility has to be installed on the system.

The benchmark suite tests Ginkgo's performance using the `SuiteSparse matrix collection` and artificially generated matrices. The suite sparse collection will be downloaded automatically when the benchmarks are run. Please note that the entire collection requires roughly 100GB of disk storage in its compressed format, and roughly 25GB of additional disk space for intermediate data (such as uncompressing the archive). Additionally, the benchmark runs usually take a long time (SpMV benchmarks on the complete collection take roughly 24h using the K20 GPU), and will stress the system.

The benchmark suite is invoked using the `make benchmark` command in the build directory. The behavior of the suite can be modified using environment variables. Assuming the `bash` shell is used, these can either be specified via the `export` command to persist between multiple runs:

```
export VARIABLE="value"
...
make benchmark
```

or specified on the fly, on the same line as the `make benchmark` command:

```
env VARIABLE="value" ... make benchmark
```

Since `make` sets any variables passed to it as temporary environment variables, the following shorthand can also be used:

```
make benchmark VARIABLE="value" ...
```

A combination of the above approaches is also possible (e.g. it may be useful to `export` the `SYSTEM_NAME` variable, and specify the others at every benchmark run).

Supported environment variables are described in the following list:

- `BENCHMARK={spmv, solver, preconditioner}` - The benchmark set to run. Default is `spmv`.
 - `spmv` - Runs the sparse matrix-vector product benchmarks on the SuiteSparse collection.

- `solver` - Runs the solver benchmarks on the SuiteSparse collection. The matrix format is determined by running the `spmv` benchmarks first, and using the fastest format determined by that benchmark. The maximum number of iterations for the iterative solvers is set to 10,000 and the requested residual reduction factor to 1e-6.
- `preconditioner` - Runs the preconditioner benchmarks on artificially generated block-diagonal matrices.
- `DRY_RUN={true, false}` - If set to `true`, prepares the system for the benchmark runs (downloads the collections, creates the result structure, etc.) and outputs the list of commands that would normally be run, but does not run the benchmarks themselves. Default is `false`.
- `EXECUTOR={reference, cuda, omp}` - The executor used for running the benchmarks. Default is `cuda`.
- `SEGMENTS=<N>` - Splits the benchmark suite into `<N>` segments. This option is useful for running the benchmarks on an HPC system with a batch scheduler, as it enables partitioning of the benchmark suite and running it concurrently on multiple nodes of the system. If specified, `SEGMENT_ID` also has to be set. Default is 1.
- `SEGMENT_ID=<I>` - used in combination with the `SEGMENTS` variable. `<I>` should be an integer between 1 and `<N>`. If specified, only the `<I>`-th segment of the benchmark suite will be run. Default is 1.
- `SYSTEM_NAME=<name>` - the name of the system where the benchmarks are being run. This option only changes the directory where the benchmark results are stored. It can be used to avoid overwriting the benchmarks if multiple systems share the same filesystem, or when copying the results between systems. Default is `unknown`.

Once `make benchmark` completes, the results can be found in `<Ginkgo build directory>/benchmark/results/<SYSTEM_NAME>/. The files are written in the JSON format, and can be analyzed using any of the data analysis tools that support JSON. Alternatively, they can be uploaded to an online repository, and analyzed using Ginkgo's free web tool Ginkgo Performance Explorer \(GPE\). (Make sure to change the "Performance data URL" to your repository if using GPE.)`

Chapter 5

Example programs

Here you can find example programs that demonstrate the usage of Ginkgo.

Some examples are built on one another and some are stand-alone and demonstrate a concept of Ginkgo, which can be used in your own code.

You can browse the available example programs

1. as [a graph](#) that shows how example programs build upon each other.
2. as [a list](#) that provides a short synopsis of each program.
3. or [grouped by topic](#).

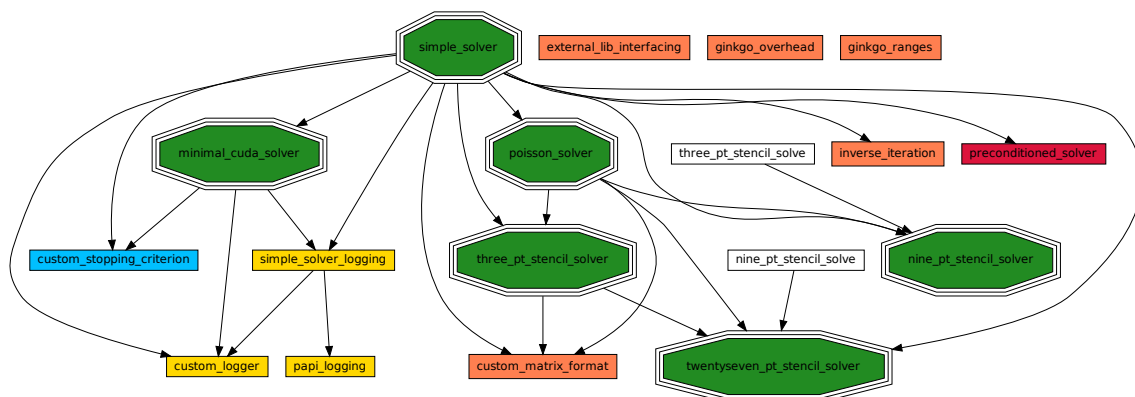
By default, all Ginkgo examples are built using CMake.

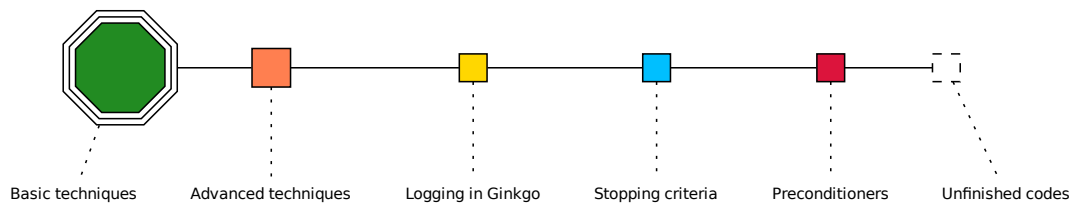
An example for building the examples and using Ginkgo as an external library without CMake can be found in the script provided for each example, which should be called with the form: `./build.sh PATH_TO_GINKGO_BUILD_DIR`

By default, Ginkgo is compiled with at least `-DGINKGO_BUILD_REFERENCE=ON`. To execute on a GPU, you need to have a GPU on the system and must have compiled Ginkgo with the `-DGINKGO_BUILD_CUDA=ON` option.

Connections between example programs

The following graph shows the connections between example programs and how they build on each other. Click on any of the boxes to go to one of the programs. If you hover your mouse pointer over a box, a brief description of the program should appear.



Legend:**Example programs**

The simple-solver program	A minimal CG solver in Ginkgo, which reads a matrix from a file.
The minimal-cuda-solver program	A minimal solver on the CUDA executor than can be run on NVIDIA GPU's.
The poisson-solver program	Solve an actual physically relevant problem, the poisson problem. The matrix is generated within Ginkgo.
The preconditioned-solver program	Using a Jacobi preconditioner to solve a linear system.
The three-pt-stencil-solver program	Using a three point stencil to solve the poisson equation with array views.
The nine-pt-stencil-solver program	Using a nine point 2D stencil to solve the poisson equation with array views.
The twentyseven-pt-stencil-solver program	Using a twentyseven point 3D stencil to solve the poisson equation with array views.
The external-lib-interfacing program	Using Ginkgo's solver with the external library deal.II.
The custom-logger program	Creating a custom logger specifically for comparing the recurrent and the real residual norms.
The custom-matrix-format program	Creating a matrix-free stencil solver by using Ginkgo's advanced methods to build your own custom matrix format.
The inverse-iteration program	Using Ginkgo to compute eigenvalues of a matrix with the inverse iteration method.
The simple-solver-logging program	Using the logging functionality in Ginkgo to get solver and other information to diagnose and debug your code.
The papi-logging program	Using the PAPI logging library in Ginkgo to get advanced information about your code and its behaviour.
The ginkgo-overhead program	Measuring the overhead of the Ginkgo library.
The custom-stopping-criterion program	Creating a custom stopping criterion for the iterative solution process.
The ginkgo-ranges program	Using the ranges concept to factorize a matrix with the LU factorization.

Example programs grouped by topics

Basic techniques

Solving a simple linear system with choice of executors.	The simple-solver program
Using the CUDA executor	The minimal-cuda-solver program
Using preconditioners	The preconditioned-solver program
Solving a physically relevant problem	The poisson-solver program , The three-pt-stencil-solver program , The nine-pt-stencil-solver program , The twentyseven-pt-stencil-solver program , The custom-matrix-format program
Reading in a matrix and right hand side from a file.	The simple-solver program , The minimal-cuda-solver program , The preconditioned-solver program , The inverse-iteration program , The simple-solver-logging program , The papi-logging program , The custom-stopping-criterion program , The custom-logger program

Advanced techniques

Using Ginkgo with external libraries.	The external-lib-interfacing program
Customizing Ginkgo	The custom-logger program , The custom-stopping-criterion program , The custom-matrix-format program
Writing your own matrix format	The custom-matrix-format program
Using Ginkgo to construct more complex linear algebra routines.	The inverse-iteration program
Logging within Ginkgo.	The simple-solver-logging program , The papi-logging program , The custom-logger program
Constructing your own stopping criterion.	The custom-stopping-criterion program
Using ranges in Ginkgo.	The ginkgo-ranges program

Chapter 6

The custom-logger program

The simple solver with a custom logger example.

This example depends on simple-solver, simple-solver-logging, minimal-cuda-solver.

Introduction

The custom-logger example shows how Ginkgo's API can be leveraged to implement application-specific callbacks for Ginkgo's events. This is the most basic way of extending Ginkgo and a good first step for any application developer who wants to adapt Ginkgo to his specific needs.

Ginkgo's `gko::log::Logger` abstraction provides hooks to the events that happen during the library execution. These hooks concern any low-level event such as memory allocations, deallocations, copies and kernel launches up to high-level events such as linear operator applications and completion of solver iterations.

In this example, a simple logger is implemented to track the solver's recurrent residual norm and compute the true residual norm. At the end of the solver execution, a comparison table is shown on-screen.

About the example

Each example has the following sections:

1. **Introduction:** This gives an overview of the example and mentions any interesting aspects in the example that might help the reader.
2. **The commented program:** This section is intended for you to understand the details of the example so that you can play with it and understand Ginkgo and its features better.
3. **Results:** This section shows the results of the code when run. Though the results may not be completely the same, you can expect the behaviour to be similar.
4. **The plain program:** This is the complete code without any comments to have an complete overview of the code.

The commented program

Include files

This is the main ginkgo header file.

```
#include <ginkgo/ginkgo.hpp>
```

Add the fstream header to read from data from files.

```
#include <fstream>
```

Add the C++ iomanip header to prettify the output.

```
#include <iomanip>
```

Add the C++ iostream header to output information to the console.

```
#include <iostream>
```

Add the string manipulation header to handle strings.

```
#include <string>
```

Add the vector header for storing the logger's data

```
#include <vector>
```

Utility function which gets the scalar value of a Ginkgo `gko::matrix::Dense` matrix representing the norm of a vector.

```
template <typename ValueType>
double get_norm(const gko::matrix::Dense<ValueType> *norm)
{
```

Put the value on CPU thanks to the master executor

```
auto cpu_norm = clone(norm->get_executor()->get_master(), norm);
```

Return the scalar value contained at position (0, 0)

```
    return cpu_norm->at(0, 0);
}
```

Utility function which computes the norm of a Ginkgo `gko::matrix::Dense` vector.

```
template <typename ValueType>
double compute_norm(const gko::matrix::Dense<ValueType> *b)
{
```

Get the executor of the vector

```
auto exec = b->get_executor();
```

Initialize a result scalar containing the value 0.0.

```
auto b_norm = gko::initialize<gko::matrix::Dense<ValueType>>({0.0}, exec);
```

Use the dense `compute_norm2` function to compute the norm.

```
b->compute_norm2(lend(b_norm));
```

Use the other utility function to return the norm contained in `b_norm`

```
    return get_norm(lend(b_norm));
}
```

Custom logger class which intercepts the residual norm scalar and solution vector in order to print a table of real vs recurrent (internal to the solvers) residual norms.

```
template <typename ValueType>
struct ResidualLogger : gko::log::Logger {
```

Output the logger's data in a table format

```
void write() const
{
```

Print a header for the table

```
std::cout << "Recurrent vs real residual norm:" << std::endl;
std::cout << '|' << std::setw(10) << "Iteration" << '|' << std::setw(25)
    << "Recurrent Residual Norm" << '|' << std::setw(25)
    << "Real Residual Norm" << '|' << std::endl;
```

Print a separation line. Note that for creating 10 characters `std::setw()` should be set to 11.

```
std::cout << '|' << std::setfill('-') << std::setw(11) << '|'
    << std::setw(26) << '|' << std::setw(26) << '|'
    << std::setfill(' ') << std::endl;
```

Print the data one by one in the form

```
for (std::size_t i = 0; i < iterations.size(); i++) {
    std::cout << std::scientific << '|' << std::setw(10)
        << iterations[i] << '|' << std::setw(25)
        << recurrent_norms[i] << '|' << std::setw(25)
        << real_norms[i] << '|' << std::defaultfloat << std::endl;
}
```

Print a separation line

```

        std::cout << '|' << std::setfill('-') << std::setw(11) << '|'
        << std::setw(26) << '|' << std::setw(26) << '|'
        << std::setfill(' ') << std::endl;
    }

    using gko_dense = gko::matrix::Dense<ValueType>;

```

Customize the logging hook which is called everytime an iteration is completed

```

void on_iteration_complete(const gko::LinOp *,
                          const gko::size_type &iteration,
                          const gko::LinOp *residual,
                          const gko::LinOp *solution,
                          const gko::LinOp *residual_norm) const override
{

```

If the solver shares a residual norm, log its value

```

if (residual_norm) {
    auto dense_norm = gko::as<gko_dense>(residual_norm);

```

Add the norm to the `recurrent_norms` vector

```

recurrent_norms.push_back(get_norm(dense_norm));

```

Otherwise, use the recurrent residual vector

```

} else {
    auto dense_residual = gko::as<gko_dense>(residual);

```

Compute the residual vector's norm

```

auto norm = compute_norm(gko::lend(dense_residual));

```

Add the computed norm to the `recurrent_norms` vector

```

    recurrent_norms.push_back(norm);
}

```

If the solver shares the current solution vector

```

if (solution) {

```

Store the matrix's executor

```

auto exec = matrix->get_executor();

```

Create a scalar containing the value 1.0

```

auto one = gko::initialize<gko_dense>({1.0}, exec);

```

Create a scalar containing the value -1.0

```
auto neg_one = gko::initialize<gko_dense>({-1.0}, exec);
```

Instantiate a temporary result variable

```
auto res = gko::clone(b);
```

Compute the real residual vector by calling apply on the system matrix

```
matrix->apply(gko::lend(one), gko::lend(solution),
             gko::lend(neg_one), gko::lend(res));
```

Compute the norm of the residual vector and add it to the `real_norms` vector

```
    real_norms.push_back(compute_norm(gko::lend(res)));
} else {
```

Add to the `real_norms` vector the value -1.0 if it could not be computed

```
    real_norms.push_back(-1.0);
}
```

Add the current iteration number to the `iterations` vector

```
    iterations.push_back(iteration);
}
```

Construct the logger and store the system matrix and b vectors

```
ResidualLogger(std::shared_ptr<const gko::Executor> exec,
               const gko::LinOp *matrix, const gko_dense *b)
: gko::log::Logger(exec, gko::log::Logger::iteration_complete_mask),
  matrix{matrix},
  b{b}
{}

private:
```

Pointer to the system matrix

```
const gko::LinOp *matrix;
```

Pointer to the right hand sides

```
const gko_dense *b;
```

Vector which stores all the recurrent residual norms

```
mutable std::vector<ValueType> recurrent_norms{};
```

Vector which stores all the real residual norms

```
mutable std::vector<ValueType> real_norms{};
```

Vector which stores all the iteration numbers

```
    mutable std::vector<std::size_t> iterations{};
};

int main(int argc, char *argv[])
{
```

Use some shortcuts. In Ginkgo, vectors are seen as a `gko::matrix::Dense` with one column/one row. The advantage of this concept is that using multiple vectors is now a natural extension of adding columns/rows are necessary.

```
using vec = gko::matrix::Dense<>;
```

The `gko::matrix::Csr` class is used here, but any other matrix class such as `gko::matrix::Coo`, `gko::matrix::Hybrid`, `gko::matrix::Ell` or `gko::matrix::Sellp` could also be used.

```
using mtx = gko::matrix::Csr<>;
```

The `gko::solver::Cg` is used here, but any other solver class can also be used.

```
using cg = gko::solver::Cg<>;
```

Print the ginkgo version information.

```
std::cout << gko::version_info::get() << std::endl;
```

Where do you want to run your solver ?

The `gko::Executor` class is one of the cornerstones of Ginkgo. Currently, we have support for an `gko::OmpExecutor`, which uses OpenMP multi-threading in most of its kernels, a `gko::ReferenceExecutor`, a single threaded specialization of the OpenMP executor and a `gko::CudaExecutor` which runs the code on a NVIDIA GPU if available.

Note

With the help of C++, you see that you only ever need to change the executor and all the other functions/ routines within Ginkgo should automatically work and run on the executor with any other changes.

```
std::shared_ptr<gko::Executor> exec;
if (argc == 1 || std::string(argv[1]) == "reference") {
    exec = gko::ReferenceExecutor::create();
} else if (argc == 2 && std::string(argv[1]) == "omp") {
    exec = gko::OmpExecutor::create();
} else if (argc == 2 && std::string(argv[1]) == "cuda" &&
           gko::CudaExecutor::get_num_devices() > 0) {
    exec = gko::CudaExecutor::create(0,
                                     gko::OmpExecutor::create());
} else {
    std::cerr << "Usage: " << argv[0] << " [executor]" << std::endl;
    std::exit(-1);
}
```


Reading your data and transfer to the proper device.

Read the matrix, right hand side and the initial solution using the read function.

Note

Ginkgo uses C++ smart pointers to automatically manage memory. To this end, we use our own object ownership transfer functions that under the hood call the required smart pointer functions to manage object ownership. The `gko::share`, `gko::give` and `gko::lend` are the functions that you would need to use.

```
auto A = share(gko::read<mtx>(std::ifstream("data/A.mtx"), exec));
auto b = gko::read<vec>(std::ifstream("data/b.mtx"), exec);
auto x = gko::read<vec>(std::ifstream("data/x0.mtx"), exec);
```

Creating the solver

Generate the `gko::solver` factory. Ginkgo uses the concept of Factories to build solvers with certain properties. Observe the Fluent interface used here. Here a cg solver is generated with a stopping criteria of maximum iterations of 20 and a residual norm reduction of 1e-15. You also observe that the stopping criteria(`gko::stop`) are also generated from factories using their build methods. You need to specify the executors which each of the object needs to be built on.

```
auto solver_gen =
    cg::build()
        .with_criteria(
            gko::stop::Iteration::build().with_max_iters(20u).on(exec),
            gko::stop::ResidualNormReduction<>::build()
                .with_reduction_factor(1e-15)
                .on(exec))
        .on(exec);
```

Instantiate a ResidualLogger logger.

```
auto logger = std::make_shared<ResidualLogger<double>>(exec, gko::lend(A),
                                                         gko::lend(b));
```

Add the previously created logger to the solver factory. The logger will be automatically propagated to all solvers created from this factory.

```
solver_gen->add_logger(logger);
```

Generate the solver from the matrix. The solver factory built in the previous step takes a "matrix"(a `gko::LinOp` to be more general) as an input. In this case we provide it with a full matrix that we previously read, but as the solver only effectively uses the `apply()` method within the provided "matrix" object, you can effectively create a `gko::LinOp` class with your own `apply` implementation to accomplish more tasks. We will see an example of how this can be done in the custom-matrix-format example

```
auto solver = solver_gen->generate(A);
```

Finally, solve the system. The solver, being a `gko::LinOp`, can be applied to a right hand side, `b` to obtain the solution, `x`.

```
solver->apply(lend(b), lend(x));
```

Print the solution to the command line.

```
std::cout << "Solution (x): \n";
write(std::cout, lend(x));
```

Print the table of the residuals obtained from the logger

```
logger->write();
```

To measure if your solution has actually converged, you can measure the error of the solution. `one`, `neg_one` are objects that represent the numbers which allow for a uniform interface when computing on any device. To compute the residual, all you need to do is call the `apply` method, which in this case is an `spmv` and equivalent to the LAPACK `z_spmv` routine. Finally, you compute the euclidean 2-norm with the `compute_norm2` function.

```
auto one = gko::initialize<vec>({1.0}, exec);
auto neg_one = gko::initialize<vec>({-1.0}, exec);
auto res = gko::initialize<vec>({0.0}, exec);
A->apply(lend(one), lend(x), lend(neg_one), lend(b));
b->compute_norm2(lend(res));

std::cout << "Residual norm sqrt(r^T r): \n";
write(std::cout, lend(res));
}
```

Results

The following is the expected result:

```
Solution (x):
%%MatrixMarket matrix array real general
19 1
0.252218
0.108645
0.0662811
0.0630433
0.0384088
0.0396536
0.0402648
0.0338935
0.0193098
0.0234653
0.0211499
0.0196413
0.0199151
0.0181674
0.0162722
0.0150714
0.0107016
0.0121141
0.0123025
Recurrent vs real residual norm:
| Iteration| Recurrent Residual Norm| Real Residual Norm|
|-----|-----|-----|
| 0| 4.358899e+00| 4.358899e+00|
| 1| 2.304548e+00| 2.304548e+00|
| 2| 1.467706e+00| 1.467706e+00|
| 3| 9.848751e-01| 9.848751e-01|
| 4| 7.418330e-01| 7.418330e-01|
| 5| 5.136231e-01| 5.136231e-01|
| 6| 3.841650e-01| 3.841650e-01|
| 7| 3.164394e-01| 3.164394e-01|
| 8| 2.277088e-01| 2.277088e-01|
| 9| 1.703121e-01| 1.703121e-01|
| 10| 9.737220e-02| 9.737220e-02|
| 11| 6.168306e-02| 6.168306e-02|
| 12| 4.541231e-02| 4.541231e-02|
| 13| 3.195304e-02| 3.195304e-02|
| 14| 1.616058e-02| 1.616058e-02|
| 15| 6.570152e-03| 6.570152e-03|
| 16| 2.643669e-03| 2.643669e-03|
| 17| 8.588089e-04| 8.588089e-04|
| 18| 2.864613e-04| 2.864613e-04|
| 19| 1.641952e-15| 2.107881e-15|
|-----|-----|-----|
Residual norm sqrt(r^T r):
%%MatrixMarket matrix array real general
1 1
2.10788e-15
```

Comments about programming and debugging

The plain program

```

/*****<GINKGO LICENSE>*****/
Copyright (c) 2017-2019, the Ginkgo authors
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its
contributors may be used to endorse or promote products derived from
this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*****/

#include <ginkgo/ginkgo.hpp>

#include <fstream>
#include <iomanip>
#include <iostream>
#include <string>
#include <vector>

template <typename ValueType>
double get_norm(const gko::matrix::Dense<ValueType> *norm)
{
    auto cpu_norm = clone(norm->get_executor()->get_master(), norm);
    return cpu_norm->at(0, 0);
}

template <typename ValueType>
double compute_norm(const gko::matrix::Dense<ValueType> *b)
{
    auto exec = b->get_executor();
    auto b_norm = gko::initialize<gko::matrix::Dense<ValueType>>({0.0}, exec);
    b->compute_norm2(lend(b_norm));
    return get_norm(lend(b_norm));
}

template <typename ValueType>
struct ResidualLogger : gko::log::Logger {
    void write() const
    {
        std::cout << "Recurrent vs real residual norm:" << std::endl;
        std::cout << '|' << std::setw(10) << "Iteration" << '|' << std::setw(25)
        << "Recurrent Residual Norm" << '|' << std::setw(25)
        << "Real Residual Norm" << '|' << std::endl;
        std::cout << '|' << std::setfill('-') << std::setw(11) << '|'
        << std::setw(26) << '|' << std::setw(26) << '|'
        << std::setfill(' ') << std::endl;
        for (std::size_t i = 0; i < iterations.size(); i++) {
            std::cout << std::scientific << '|' << std::setw(10)
            << iterations[i] << '|' << std::setw(25)
            << recurrent_norms[i] << '|' << std::setw(25)
            << real_norms[i] << '|' << std::defaultfloat << std::endl;
        }
        std::cout << '|' << std::setfill('-') << std::setw(11) << '|'
        << std::setw(26) << '|' << std::setw(26) << '|'
        << std::setfill(' ') << std::endl;
    }
}

```

```

using gko_dense = gko::matrix::Dense<ValueType>;

void on_iteration_complete(const gko::LinOp *,
                          const gko::size_type &iteration,
                          const gko::LinOp *residual,
                          const gko::LinOp *solution,
                          const gko::LinOp *residual_norm) const override
{
    if (residual_norm) {
        auto dense_norm = gko::as<gko_dense>(residual_norm);
        recurrent_norms.push_back(get_norm(dense_norm));
    } else {
        auto dense_residual = gko::as<gko_dense>(residual);
        auto norm = compute_norm(gko::lend(dense_residual));
        recurrent_norms.push_back(norm);
    }

    if (solution) {
        auto exec = matrix->get_executor();
        auto one = gko::initialize<gko_dense>({1.0}, exec);
        auto neg_one = gko::initialize<gko_dense>({-1.0}, exec);
        auto res = gko::clone(b);
        matrix->apply(gko::lend(one), gko::lend(solution),
                    gko::lend(neg_one), gko::lend(res));

        real_norms.push_back(compute_norm(gko::lend(res)));
    } else {
        real_norms.push_back(-1.0);
    }

    iterations.push_back(iteration);
}

ResidualLogger(std::shared_ptr<const gko::Executor> exec,
               const gko::LinOp *matrix, const gko_dense *b)
: gko::log::Logger(exec, gko::log::Logger::iteration_complete_mask),
  matrix{matrix},
  b{b}
{}

private:
const gko::LinOp *matrix;
const gko_dense *b;
mutable std::vector<ValueType> recurrent_norms{};
mutable std::vector<ValueType> real_norms{};
mutable std::vector<std::size_t> iterations{};
};

int main(int argc, char *argv[])
{
    using vec = gko::matrix::Dense<>;
    using mtx = gko::matrix::Csr<>;
    using cg = gko::solver::Cg<>;

    std::cout << gko::version_info::get() << std::endl;

    std::shared_ptr<gko::Executor> exec;
    if (argc == 1 || std::string(argv[1]) == "reference") {
        exec = gko::ReferenceExecutor::create();
    } else if (argc == 2 && std::string(argv[1]) == "omp") {
        exec = gko::OmpExecutor::create();
    } else if (argc == 2 && std::string(argv[1]) == "cuda" &&
               gko::CudaExecutor::get_num_devices() > 0) {
        exec = gko::CudaExecutor::create(0,
        gko::OmpExecutor::create());
    } else {
        std::cerr << "Usage: " << argv[0] << " [executor]" << std::endl;
        std::exit(-1);
    }

    auto A = share(gko::read<mtx>(std::ifstream("data/A.mtx"), exec));
    auto b = gko::read<vec>(std::ifstream("data/b.mtx"), exec);
    auto x = gko::read<vec>(std::ifstream("data/x0.mtx"), exec);

    auto solver_gen =
        cg::build()
            .with_criteria(
                gko::stop::Iteration::build().with_max_iters(20u).on(exec),
                gko::stop::ResidualNormReduction<>::build()
                    .with_reduction_factor(1e-15)
                    .on(exec))
            .on(exec);

    auto logger = std::make_shared<ResidualLogger<double>>(exec, gko::lend(A),
                                                         gko::lend(b));

```

```
solver_gen->add_logger(logger);  
  
auto solver = solver_gen->generate(A);  
  
solver->apply(lend(b), lend(x));  
  
std::cout << "Solution (x): \n";  
write(std::cout, lend(x));  
  
logger->write();  
  
auto one = gko::initialize<vec>({1.0}, exec);  
auto neg_one = gko::initialize<vec>({-1.0}, exec);  
auto res = gko::initialize<vec>({0.0}, exec);  
A->apply(lend(one), lend(x), lend(neg_one), lend(b));  
b->compute_norm2(lend(res));  
  
std::cout << "Residual norm sqrt(r^T r): \n";  
write(std::cout, lend(res));  
}
```


Chapter 7

The custom-matrix-format program

The custom matrix format example.

This example depends on simple-solver, poisson-solver, three-pt-stencil-solver, .

Introduction

This example solves a 1D Poisson equation:

$$\begin{aligned} u &: [0, 1] \rightarrow \mathbb{R} \\ u'' &= f \\ u(0) &= u_0 \\ u(1) &= u_1 \end{aligned}$$

using a finite difference method on an equidistant grid with K discretization points (K can be controlled with a command line parameter). The discretization is done via the second order Taylor polynomial:

$$\begin{aligned} u(x+h) &= u(x) + u'(x)h + 1/2u''(x)h^2 + O(h^3) \\ u(x-h) &= u(x) - u'(x)h + 1/2u''(x)h^2 + O(h^3) \\ \hline -u(x-h) + 2u(x) - u(x+h) &= -f(x)h^2 + O(h^3) \end{aligned}$$

For an equidistant grid with K "inner" discretization points x_1, \dots, x_K , and step size $h = 1/(K+1)$, the formula produces a system of linear equations

$$\begin{aligned} 2u_1 - u_2 &= -f_1h^2 + u_0 \\ -u_{k-1} + 2u_k - u_{k+1} &= -f_kh^2, k = 2, \dots, K-1 \\ -u_{K-1} + 2u_K &= -f_Kh^2 + u_1 \end{aligned}$$

which is then solved using Ginkgo's implementation of the CG method preconditioned with block-Jacobi. It is also possible to specify on which executor Ginkgo will solve the system via the command line. The function `f` is set to `f(x) = 6x` (making the solution `u(x) = x3`), but that can be changed in the `main` function.

The intention of this example is to show how a custom linear operator can be created and integrated into Ginkgo to achieve performance benefits.

About the example

The commented program

```
#include <iostream>
#include <map>
#include <string>

#include <omp.h>
#include <ginkgo/ginkgo.hpp>
```

A CUDA kernel implementing the stencil, which will be used if running on the CUDA executor. Unfortunately, NVCC has serious problems interpreting some parts of Ginkgo's code, so the kernel has to be compiled separately.

```
extern void stencil_kernel(std::size_t size, const double *coefs,
                          const double *b, double *x);
```

A stencil matrix class representing the 3pt stencil linear operator. We include the `gko::EnableLinOp` mixin which implements the entire `LinOp` interface, except the two `apply_impl` methods, which get called inside the default implementation of `apply` (after argument verification) to perform the actual application of the linear operator. In addition, it includes the implementation of the entire `PolymorphicObject` interface.

It also includes the `gko::EnableCreateMethod` mixin which provides a default implementation of the static `create` method. This method will forward all its arguments to the constructor to create the object, and return an `std::unique_ptr` to the created object.

```
class StencilMatrix : public gko::EnableLinOp<StencilMatrix>,
                     public gko::EnableCreateMethod<StencilMatrix> {
public:
```

This constructor will be called by the `create` method. Here we initialize the coefficients of the stencil.

```
    StencilMatrix(std::shared_ptr<const gko::Executor> exec,
                  gko::size_type size = 0, double left = -1.0,
                  double center = 2.0, double right = -1.0)
        : gko::EnableLinOp<StencilMatrix>(exec, gko::dim<2>{size}),
          coefficients(exec, {left, center, right})
    {}

protected:
    using vec = gko::matrix::Dense<>;
    using coef_type = gko::Array<double>;
```

Here we implement the application of the linear operator, $x = A * b$. `apply_impl` will be called by the `apply` method, after the arguments have been moved to the correct executor and the operators checked for conforming sizes.

For simplicity, we assume that there is always only one right hand side and the stride of consecutive elements in the vectors is 1 (both of these are always true in this example).

```
void apply_impl(const gko::LinOp *b, gko::LinOp *x) const override
{
```

we only implement the operator for dense RHS. `gko::as` will throw an exception if its argument is not `Dense`.

```
    auto dense_b = gko::as<vec>(b);
    auto dense_x = gko::as<vec>(x);
```


we need separate implementations depending on the executor, so we create an operation which maps the call to the correct implementation

```
struct stencil_operation : gko::Operation {
    stencil_operation(const coef_type &coefficients, const vec *b,
                     vec *x)
        : coefficients{coefficients}, b{b}, x{x}
    {}
}
```

OpenMP implementation

```
void run(std::shared_ptr<const gko::OmpExecutor>) const override
{
    auto b_values = b->get_const_values();
    auto x_values = x->get_values();
#pragma omp parallel for
    for (std::size_t i = 0; i < x->get_size()[0]; ++i) {
        auto coefs = coefficients.get_const_data();
        auto result = coefs[1] * b_values[i];
        if (i > 0) {
            result += coefs[0] * b_values[i - 1];
        }
        if (i < x->get_size()[0] - 1) {
            result += coefs[2] * b_values[i + 1];
        }
        x_values[i] = result;
    }
}
```

CUDA implementation

```
void run(std::shared_ptr<const gko::CudaExecutor>) const override
{
    stencil_kernel(x->get_size()[0], coefficients.get_const_data(),
                  b->get_const_values(), x->get_values());
}
```

We do not provide an implementation for reference executor. If not provided, Ginkgo will use the implementation for the OpenMP executor when calling it in the reference executor.

```
const coef_type &coefficients;
const vec *b;
vec *x;
};
this->get_executor()->run(
    stencil_operation(coefficients, dense_b, dense_x));
}
```

There is also a version of the apply function which does the operation $x = \alpha * A * b + \beta * x$. This function is commonly used and can often be better optimized than implementing it using $x = A * b$. However, for simplicity, we will implement it exactly like that in this example.

```
void apply_impl(const gko::LinOp *alpha, const gko::LinOp *b,
               const gko::LinOp *beta, gko::LinOp *x) const override
{
    auto dense_b = gko::as<vec>(b);
    auto dense_x = gko::as<vec>(x);
    auto tmp_x = dense_x->clone();
    this->apply_impl(b, lend(tmp_x));
    dense_x->scale(beta);
    dense_x->add_scaled(alpha, lend(tmp_x));
}

private:
    coef_type coefficients;
};
```

Creates a stencil matrix in CSR format for the given number of discretization points.

```
void generate_stencil_matrix(gko::matrix::Csr<> *matrix)
{
    const auto discretization_points = matrix->get_size()[0];
    auto row_ptrs = matrix->get_row_ptrs();
    auto col_idxxs = matrix->get_col_idxxs();
    auto values = matrix->get_values();
    int pos = 0;
    const double coefs[] = {-1, 2, -1};
    row_ptrs[0] = pos;
    for (int i = 0; i < discretization_points; ++i) {
        for (auto ofs : {-1, 0, 1}) {
            if (0 <= i + ofs && i + ofs < discretization_points) {
                values[pos] = coefs[ofs + 1];
                col_idxxs[pos] = i + ofs;
                ++pos;
            }
        }
        row_ptrs[i + 1] = pos;
    }
}
```

Generates the RHS vector given f and the boundary conditions.

```
template <typename Closure>
void generate_rhs(Closure f, double u0, double u1, gko::matrix::Dense<> *rhs)
{
    const auto discretization_points = rhs->get_size()[0];
    auto values = rhs->get_values();
    const auto h = 1.0 / (discretization_points + 1);
    for (int i = 0; i < discretization_points; ++i) {
        const auto xi = (i + 1) * h;
        values[i] = -f(xi) * h * h;
    }
    values[0] += u0;
    values[discretization_points - 1] += u1;
}
```

Prints the solution u .

```
void print_solution(double u0, double u1, const gko::matrix::Dense<> *u)
{
    std::cout << u0 << '\n';
    for (int i = 0; i < u->get_size()[0]; ++i) {
        std::cout << u->get_const_values()[i] << '\n';
    }
    std::cout << u1 << std::endl;
}
```

Computes the 1-norm of the error given the computed u and the correct solution function `correct_u`.

```
template <typename Closure>
double calculate_error(int discretization_points, const gko::matrix::Dense<> *u,
                      Closure correct_u)
{
    const auto h = 1.0 / (discretization_points + 1);
    auto error = 0.0;
    for (int i = 0; i < discretization_points; ++i) {
        using std::abs;
        const auto xi = (i + 1) * h;
        error +=
            abs(u->get_const_values()[i] - correct_u(xi)) /
            abs(correct_u(xi));
    }
    return error;
}

int main(int argc, char *argv[])
{
}
```

Some shortcuts

```
using vec = gko::matrix::Dense<double>;
using mtx = gko::matrix::Csr<double, int>;
using cg = gko::solver::Cg<double>;

if (argc < 2) {
    std::cerr << "Usage: " << argv[0] << " DISCRETIZATION_POINTS [executor]"
        << std::endl;
    std::exit(-1);
}
```

Get number of discretization points

```
const unsigned int discretization_points =
    argc >= 2 ? std::atoi(argv[1]) : 100u;
const auto executor_string = argc >= 3 ? argv[2] : "reference";
```

Figure out where to run the code

```
const auto omp = gko::OmpExecutor::create();
std::map<std::string, std::shared_ptr<gko::Executor>> exec_map{
    {"omp", omp},
    {"cuda", gko::CudaExecutor::create(0, omp)},
    {"reference", gko::ReferenceExecutor::create()}};
```

executor where Ginkgo will perform the computation

```
const auto exec = exec_map.at(executor_string); // throws if not valid
```

executor used by the application

```
const auto app_exec = exec_map["omp"];
```

problem:

```
auto correct_u = [](double x) { return x * x * x; };
auto f = [](double x) { return 6 * x; };
auto u0 = correct_u(0);
auto u1 = correct_u(1);
```

initialize vectors

```
auto rhs = vec::create(app_exec, gko::dim<2>(discretization_points, 1));
generate_rhs(f, u0, u1, lend(rhs));
auto u = vec::create(app_exec, gko::dim<2>(discretization_points, 1));
for (int i = 0; i < u->get_size()[0]; ++i) {
    u->get_values()[i] = 0.0;
}
```

Generate solver and solve the system

```
cg::build()
    .with_criteria(gko::stop::Iteration::build()
        .with_max_iters(discretization_points)
        .on(exec),
        gko::stop::ResidualNormReduction<>::build()
            .with_reduction_factor(1e-6)
            .on(exec))
    .on(exec)
```

notice how our custom StencilMatrix can be used in the same way as any built-in type

```
->generate(
    StencilMatrix::create(exec, discretization_points, -1, 2, -1))
->apply(lend(rhs), lend(u));

print_solution(u0, u1, lend(u));
std::cout << "The average relative error is "
    << calculate_error(discretization_points, lend(u), correct_u) /
        discretization_points
    << std::endl;
}
```

Results

Comments about programming and debugging

The plain program

```

/*****<GINKGO LICENSE>*****/
Copyright (c) 2017-2019, the Ginkgo authors
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its
contributors may be used to endorse or promote products derived from
this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*****/

#include <iostream>
#include <map>
#include <string>

#include <omp.h>
#include <ginkgo/ginkgo.hpp>

extern void stencil_kernel(std::size_t size, const double *coefs,
                          const double *b, double *x);

class StencilMatrix : public gko::EnableLinOp<StencilMatrix>,
                     public gko::EnableCreateMethod<StencilMatrix> {
public:
    StencilMatrix(std::shared_ptr<const gko::Executor> exec,
                  gko::size_type size = 0, double left = -1.0,
                  double center = 2.0, double right = -1.0)
        : gko::EnableLinOp<StencilMatrix>(exec, gko::dim<2>{size}),
          coefficients(exec, {left, center, right})
    {}

protected:
    using vec = gko::matrix::Dense<>;
    using coef_type = gko::Array<double>;

    void apply_impl(const gko::LinOp *b, gko::LinOp *x) const override
    {
        auto dense_b = gko::as<vec>(b);
        auto dense_x = gko::as<vec>(x);

        struct stencil_operation : gko::Operation {
            stencil_operation(const coef_type &coefficients, const vec *b,
                             vec *x)
                : coefficients{coefficients}, b{b}, x{x}
            {}

            void run(std::shared_ptr<const gko::OmpExecutor>) const override
            {
                auto b_values = b->get_const_values();
                auto x_values = x->get_values();
                #pragma omp parallel for

```

```

        for (std::size_t i = 0; i < x->get_size()[0]; ++i) {
            auto coefs = coefficients.get_const_data();
            auto result = coefs[1] * b_values[i];
            if (i > 0) {
                result += coefs[0] * b_values[i - 1];
            }
            if (i < x->get_size()[0] - 1) {
                result += coefs[2] * b_values[i + 1];
            }
            x_values[i] = result;
        }

        void run(std::shared_ptr<const gko::CudaExecutor>) const override
        {
            stencil_kernel(x->get_size()[0], coefficients.get_const_data(),
                          b->get_const_values(), x->get_values());
        }

        const coef_type &coefficients;
        const vec *b;
        vec *x;
    };
    this->get_executor()->run(
        stencil_operation(coefficients, dense_b, dense_x));
}

void apply_impl(const gko::LinOp *alpha, const gko::LinOp *b,
               const gko::LinOp *beta, gko::LinOp *x) const override
{
    auto dense_b = gko::as<vec>(b);
    auto dense_x = gko::as<vec>(x);
    auto tmp_x = dense_x->clone();
    this->apply_impl(b, lend(tmp_x));
    dense_x->scale(beta);
    dense_x->add_scaled(alpha, lend(tmp_x));
}

private:
    coef_type coefficients;
};

void generate_stencil_matrix(gko::matrix::Csr<> *matrix)
{
    const auto discretization_points = matrix->get_size()[0];
    auto row_ptrs = matrix->get_row_ptrs();
    auto col_idxs = matrix->get_col_idxs();
    auto values = matrix->get_values();
    int pos = 0;
    const double coefs[] = {-1, 2, -1};
    row_ptrs[0] = pos;
    for (int i = 0; i < discretization_points; ++i) {
        for (auto ofs : {-1, 0, 1}) {
            if (0 <= i + ofs && i + ofs < discretization_points) {
                values[pos] = coefs[ofs + 1];
                col_idxs[pos] = i + ofs;
                ++pos;
            }
        }
        row_ptrs[i + 1] = pos;
    }
}

template <typename Closure>
void generate_rhs(Closure f, double u0, double u1, gko::matrix::Dense<> *rhs)
{
    const auto discretization_points = rhs->get_size()[0];
    auto values = rhs->get_values();
    const auto h = 1.0 / (discretization_points + 1);
    for (int i = 0; i < discretization_points; ++i) {
        const auto xi = (i + 1) * h;
        values[i] = -f(xi) * h * h;
    }
    values[0] += u0;
    values[discretization_points - 1] += u1;
}

void print_solution(double u0, double u1, const gko::matrix::Dense<> *u)
{
    std::cout << u0 << '\n';
    for (int i = 0; i < u->get_size()[0]; ++i) {
        std::cout << u->get_const_values()[i] << '\n';
    }
}

```

```

    std::cout << u1 << std::endl;
}

template <typename Closure>
double calculate_error(int discretization_points, const gko::matrix::Dense<> *u,
                      Closure correct_u)
{
    const auto h = 1.0 / (discretization_points + 1);
    auto error = 0.0;
    for (int i = 0; i < discretization_points; ++i) {
        using std::abs;
        const auto xi = (i + 1) * h;
        error +=
            abs(u->get_const_values()[i] - correct_u(xi)) /
            abs(correct_u(xi));
    }
    return error;
}

int main(int argc, char *argv[])
{
    using vec = gko::matrix::Dense<double>;
    using mtx = gko::matrix::Csr<double, int>;
    using cg = gko::solver::Cg<double>;

    if (argc < 2) {
        std::cerr << "Usage: " << argv[0] << " DISCRETIZATION_POINTS [executor]"
                  << std::endl;
        std::exit(-1);
    }

    const unsigned int discretization_points =
        argc >= 2 ? std::atoi(argv[1]) : 100u;
    const auto executor_string = argc >= 3 ? argv[2] : "reference";

    const auto omp = gko::OmpExecutor::create();
    std::map<std::string, std::shared_ptr<gko::Executor>> exec_map{
        {"omp", omp},
        {"cuda", gko::CudaExecutor::create(0, omp)},
        {"reference", gko::ReferenceExecutor::create()}};

    const auto exec = exec_map.at(executor_string); // throws if not valid
    const auto app_exec = exec_map["omp"];

    auto correct_u = [](double x) { return x * x * x; };
    auto f = [](double x) { return 6 * x; };
    auto u0 = correct_u(0);
    auto u1 = correct_u(1);

    auto rhs = vec::create(app_exec, gko::dim<2>(discretization_points, 1));
    generate_rhs(f, u0, u1, lend(rhs));
    auto u = vec::create(app_exec, gko::dim<2>(discretization_points, 1));
    for (int i = 0; i < u->get_size()[0]; ++i) {
        u->get_values()[i] = 0.0;
    }

    cg::build()
        .with_criteria(gko::stop::Iteration::build()
                      .with_max_iters(discretization_points)
                      .on(exec),
                      gko::stop::ResidualNormReduction<>::build()
                      .with_reduction_factor(1e-6)
                      .on(exec))
        .on(exec)
        ->generate(
            StencilMatrix::create(exec, discretization_points, -1, 2, -1))
        ->apply(lend(rhs), lend(u));

    print_solution(u0, u1, lend(u));
    std::cout << "The average relative error is "
              << calculate_error(discretization_points, lend(u), correct_u) /
                discretization_points
              << std::endl;
}

```

Chapter 8

The custom-stopping-criterion program

The custom stopping criterion creation example.

This example depends on simple-solver, minimal-cuda-solver.

Introduction

About the example

The commented program

```
#include <ginkgo/ginkgo.hpp>

#include <fstream>
#include <iostream>
#include <string>
#include <thread>

/ **
 * The ByInteraction class is a criterion which asks for user input to stop
 * the iteration process. Using this criterion is slightly more complex than the
 * other ones, because it is asynchronous therefore requires the use of threads.
 */
class ByInteraction
: public gko::EnablePolymorphicObject<ByInteraction, gko::stop::Criterion>
{
    friend class gko::EnablePolymorphicObject<ByInteraction,
                                              gko::stop::Criterion>;
    using Criterion = gko::stop::Criterion;

public:
    GKO_CREATE_FACTORY_PARAMETERS(parameters, Factory)
    {
        / **
         * Boolean set by the user to stop the iteration process
         */
        std::add_pointer<volatile bool>::type GKO_FACTORY_PARAMETER(
            stop_iteration_process, nullptr);
    };
    GKO_ENABLE_CRITERION_FACTORY(ByInteraction, parameters, Factory);
    GKO_ENABLE_BUILD_METHOD(Factory);

protected:
    bool check_impl(gko::uint8 stoppingId, bool setFinalized,
                   gko::Array<gko::stopping_status> *stop_status,
                   bool *one_changed, const Criterion::Updater &) override
    {
        bool result = *(parameters_.stop_iteration_process);
```

```

        if (result) {
            this->set_all_statuses(stoppingId, setFinalized, stop_status);
            *one_changed = true;
        }
        return result;
    }

    explicit ByInteraction(std::shared_ptr<const gko::Executor> exec)
        : EnablePolymorphicObject<ByInteraction, Criterion>(std::move(exec))
    {}

    explicit ByInteraction(const Factory *factory,
                           const gko::stop::CriterionArgs &args)
        : EnablePolymorphicObject<ByInteraction, Criterion>(
            factory->get_executor()),
          parameters_{factory->get_parameters()}
    {}
};

void run_solver(volatile bool *stop_iteration_process,
                std::shared_ptr<gko::Executor> exec)
{

```

Some shortcuts

```

using mtx = gko::matrix::Csr<>;
using vec = gko::matrix::Dense<>;
using bicg = gko::solver::Bicgstab<>;

```

Read Data

```

auto A = share(gko::read<mtx>(std::ifstream("data/A.mtx"), exec));
auto b = gko::read<vec>(std::ifstream("data/b.mtx"), exec);
auto x = gko::read<vec>(std::ifstream("data/x0.mtx"), exec);

```

Create solver factory and solve system

```

auto solver = bicg::build()
    .with_criteria(ByInteraction::build()
        .with_stop_iteration_process(
            stop_iteration_process)
        .on(exec))
    .on(exec)
    ->generate(A);
solver->add_logger(gko::log::Stream<>::create(
    exec, gko::log::Logger::iteration_complete_mask, std::cout, true));
solver->apply(lend(b), lend(x));

std::cout << "Solver stopped" << std::endl;

```

Print solution

```

std::cout << "Solution (x): \n";
write(std::cout, lend(x));

```

Calculate residual

```

auto one = gko::initialize<vec>({1.0}, exec);
auto neg_one = gko::initialize<vec>({-1.0}, exec);
auto res = gko::initialize<vec>({0.0}, exec);
A->apply(lend(one), lend(x), lend(neg_one), lend(b));
b->compute_norm2(lend(res));

std::cout << "Residual norm sqrt(r^T r): \n";
write(std::cout, lend(res));
}

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

```


Print version information

```
std::cout << gko::version_info::get() << std::endl;
```

Figure out where to run the code

```
std::shared_ptr<gko::Executor> exec;
if (argc == 1 || std::string(argv[1]) == "reference") {
    exec = gko::ReferenceExecutor::create();
} else if (argc == 2 && std::string(argv[1]) == "omp") {
    exec = gko::OmpExecutor::create();
} else if (argc == 2 && std::string(argv[1]) == "cuda" &&
           gko::CudaExecutor::get_num_devices() > 0) {
    exec = gko::CudaExecutor::create(0,
                                     gko::OmpExecutor::create());
} else {
    std::cerr << "Usage: " << argv[0] << " [executor]" << std::endl;
    std::exit(-1);
}
```

Declare a user controlled boolean for the iteration process

```
volatile bool stop_iteration_process{};
```

Create a new a thread to launch the solver

```
std::thread t(run_solver, &stop_iteration_process, exec);
```

Look for an input command "stop" in the console, which sets the boolean to true

```
std::cout << "Type 'stop' to stop the iteration process" << std::endl;
std::string command;
while (std::cin >> command) {
    if (command == "stop") {
        break;
    } else {
        std::cout << "Unknown command" << std::endl;
    }
}
std::cout << "User input command 'stop' - The solver will stop!"
          << std::endl;
stop_iteration_process = true;
t.join();
}
```

Results

This is the expected output:

```
.
.
.
.
.
[LOG] >>> iteration 15331 completed with solver LinOp[gko::solver::Bicgstab<double>,0x7f2f38003c10] with
        residual LinOp[gko::matrix::Dense<double>,0x7f2f380048e0], solution LinOp[gko::LinOp const*,0] and
        residual_norm LinOp[gko::LinOp const*,0]
LinOp[gko::matrix::Dense<double>,0x7f2f380048e0] [
    6.21705e-164
    -1.18919e-164
    7.89129e-165
    -6.78013e-165
```

```

-2.42405e-164
-4.29503e-165
6.16567e-166
-3.34064e-164
6.38335e-165
7.86768e-165
-1.80969e-165
-4.17609e-166
2.5395e-165
-5.34283e-166
-4.10476e-166
-1.50132e-166
-1.25732e-165
-1.82819e-166
-2.0927e-165
]

// Typing 'stop' stops the solver.

User input command 'stop' - The solver will stop

LinOp[gko::matrix::Dense<double>,0x7f2f38004730][
  0.252218
  0.108645
  0.0662811
  0.0630433
  0.0384088
  0.0396536
  0.0402648
  0.0338935
  0.0193098
  0.0234653
  0.0211499
  0.0196413
  0.0199151
  0.0181674
  0.0162722
  0.0150714
  0.0107016
  0.0121141
  0.0123025
]

Solver stopped
Solution (x):
%%MatrixMarket matrix array real general
19 1
0.252218
0.108645
0.0662811
0.0630433
0.0384088
0.0396536
0.0402648
0.0338935
0.0193098
0.0234653
0.0211499
0.0196413
0.0199151
0.0181674
0.0162722
0.0150714
0.0107016
0.0121141
0.0123025
Residual norm sqrt(r^T r):
%%MatrixMarket matrix array real general
1 1
1.06135e-15

```

Comments about programming and debugging

The plain program

```

/*****<GINKGO LICENSE>*****/
Copyright (c) 2017-2019, the Ginkgo authors
All rights reserved.

Redistribution and use in source and binary forms, with or without

```

modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *****<GINKGO LICENSE>*****

```
#include <ginkgo/ginkgo.hpp>

#include <fstream>
#include <iostream>
#include <string>
#include <thread>

class ByInteraction
: public gko::EnablePolymorphicObject<ByInteraction, gko::stop::Criterion>
{
    friend class gko::EnablePolymorphicObject<ByInteraction,
                                              gko::stop::Criterion>;
    using Criterion = gko::stop::Criterion;

public:
    GKO_CREATE_FACTORY_PARAMETERS(parameters, Factory)
    {
        std::add_pointer<volatile bool>::type GKO_FACTORY_PARAMETER(
            stop_iteration_process, nullptr);
    };
    GKO_ENABLE_CRITERION_FACTORY(ByInteraction, parameters, Factory);
    GKO_ENABLE_BUILD_METHOD(Factory);

protected:
    bool check_impl(gko::uint8 stoppingId, bool setFinalized,
                   gko::Array<gko::stopping_status> *stop_status,
                   bool *one_changed, const Criterion::Updater &) override
    {
        bool result = *(parameters_.stop_iteration_process);
        if (result) {
            this->set_all_statuses(stoppingId, setFinalized, stop_status);
            *one_changed = true;
        }
        return result;
    }

    explicit ByInteraction(std::shared_ptr<const gko::Executor> exec)
        : EnablePolymorphicObject<ByInteraction, Criterion>(std::move(exec))
    {}

    explicit ByInteraction(const Factory *factory,
                           const gko::stop::CriterionArgs &args)
        : EnablePolymorphicObject<ByInteraction, Criterion>({
            factory->get_executor(),
            parameters_{factory->get_parameters()}
        })
    {}
};

void run_solver(volatile bool *stop_iteration_process,
               std::shared_ptr<gko::Executor> exec)
{
    using mtx = gko::matrix::Csr<>;
    using vec = gko::matrix::Dense<>;
    using bicg = gko::solver::Bicgstab<>;

    auto A = share(gko::read<mtx>(std::ifstream("data/A.mtx"), exec));
```

```

auto b = gko::read<vec>(std::ifstream("data/b.mtx"), exec);
auto x = gko::read<vec>(std::ifstream("data/x0.mtx"), exec);

auto solver = bicg::build()
    .with_criteria(ByInteraction::build()
        .with_stop_iteration_process(
            stop_iteration_process)
        .on(exec))
    .on(exec)
    ->generate(A);
solver->add_logger(gko::log::Stream<>::create(
    exec, gko::log::logger::iteration_complete_mask, std::cout, true));
solver->apply(lend(b), lend(x));

std::cout << "Solver stopped" << std::endl;

std::cout << "Solution (x): \n";
write(std::cout, lend(x));

auto one = gko::initialize<vec>({1.0}, exec);
auto neg_one = gko::initialize<vec>({-1.0}, exec);
auto res = gko::initialize<vec>({0.0}, exec);
A->apply(lend(one), lend(x), lend(neg_one), lend(b));
b->compute_norm2(lend(res));

std::cout << "Residual norm sqrt(r^T r): \n";
write(std::cout, lend(res));
}

int main(int argc, char *argv[])
{
    std::cout << gko::version_info::get() << std::endl;

    std::shared_ptr<gko::Executor> exec;
    if (argc == 1 || std::string(argv[1]) == "reference") {
        exec = gko::ReferenceExecutor::create();
    } else if (argc == 2 && std::string(argv[1]) == "omp") {
        exec = gko::OmpExecutor::create();
    } else if (argc == 2 && std::string(argv[1]) == "cuda" &&
        gko::CudaExecutor::get_num_devices() > 0) {
        exec = gko::CudaExecutor::create(0,
            gko::OmpExecutor::create());
    } else {
        std::cerr << "Usage: " << argv[0] << " [executor]" << std::endl;
        std::exit(-1);
    }

    volatile bool stop_iteration_process{};

    std::thread t(run_solver, &stop_iteration_process, exec);

    std::cout << "Type 'stop' to stop the iteration process" << std::endl;
    std::string command;
    while (std::cin >> command) {
        if (command == "stop") {
            break;
        } else {
            std::cout << "Unknown command" << std::endl;
        }
    }
    std::cout << "User input command 'stop' - The solver will stop!"
        << std::endl;
    stop_iteration_process = true;
    t.join();
}

```

Chapter 9

The external-lib-interfacing program

The external library(deal.II) interfacing example.

Introduction

About the example

The commented program

```
#include <deal.II/base/function.h>
#include <deal.II/base/logstream.h>
#include <deal.II/base/quadrature_lib.h>
#include <deal.II/dofs/dof_accessor.h>
#include <deal.II/dofs/dof_handler.h>
#include <deal.II/dofs/dof_tools.h>
#include <deal.II/fe/fe_q.h>
#include <deal.II/fe/fe_values.h>
#include <deal.II/grid/grid_generator.h>
#include <deal.II/grid/grid_out.h>
#include <deal.II/grid/grid_refinement.h>
#include <deal.II/grid/tria.h>
#include <deal.II/grid/tria_accessor.h>
#include <deal.II/grid/tria_iterator.h>
#include <deal.II/lac/constraint_matrix.h>
#include <deal.II/lac/dynamic_sparsity_pattern.h>
#include <deal.II/lac/full_matrix.h>
#include <deal.II/lac/precondition.h>
#include <deal.II/lac/solver_bicgstab.h>
#include <deal.II/lac/sparse_matrix.h>
#include <deal.II/lac/vector.h>
#include <deal.II/numerics/data_out.h>
#include <deal.II/numerics/matrix_tools.h>
#include <deal.II/numerics/vector_tools.h>
```

The following two files provide classes and information for multithreaded programs. In the first one, the classes and functions are declared which we need to do assembly in parallel (i.e. the `WorkStream` namespace). The second file has a class `MultithreadInfo` which can be used to query the number of processors in your system, which is often useful when deciding how many threads to start in parallel.

```
#include <deal.II/base/multithread_info.h>
#include <deal.II/base/work_stream.h>
```

The next new include file declares a base class `TensorFunction` not unlike the `Function` class, but with the difference that the return value is tensor-valued rather than scalar or vector-valued.

```
#include <deal.II/base/tensor_function.h>
#include <deal.II/numerics/error_estimator.h>
```

Ginkgo's header file

```
#include <ginkgo/ginkgo.hpp>
```

This is C++, as we want to write some output to disk:

```
#include <fstream>
#include <iostream>
```

The last step is as in previous programs:

```
namespace Step9 {
using namespace dealii;
```

AdvectionProblem class declaration

Following we declare the main class of this program. It is very much like the main classes of previous examples, so we again only comment on the differences.

```
template <int dim>
class AdvectionProblem {
public:
    AdvectionProblem();
    ~AdvectionProblem();
    void run();

private:
    void setup_system();
```

The next set of functions will be used to assemble the matrix. However, unlike in the previous examples, the `assemble_system()` function will not do the work itself, but rather will delegate the actual assembly to helper functions `assemble_local_system()` and `copy_local_to_global()`. The rationale is that matrix assembly can be parallelized quite well, as the computation of the local contributions on each cell is entirely independent of other cells, and we only have to synchronize when we add the contribution of a cell to the global matrix.

The strategy for parallelization we choose here is one of the possibilities mentioned in detail in the threads module in the documentation. Specifically, we will use the WorkStream approach discussed there. Since there is so much documentation in this module, we will not repeat the rationale for the design choices here (for example, if you read through the module mentioned above, you will understand what the purpose of the `AssemblyScratchData` and `AssemblyCopyData` structures is). Rather, we will only discuss the specific implementation.

If you read the page mentioned above, you will find that in order to parallelize assembly, we need two data structures – one that corresponds to data that we need during local integration ("scratch data", i.e., things we only need as temporary storage), and one that carries information from the local integration to the function that then adds the local contributions to the corresponding elements of the global matrix. The former of these typically contains the `FEValues` and `FEFaceValues` objects, whereas the latter has the local matrix, local right hand side, and information about which degrees of freedom live on the cell for which we are assembling a local contribution. With this information, the following should be relatively self-explanatory:

```

struct AssemblyScratchData {
    AssemblyScratchData(const FiniteElement<dim> &fe);
    AssemblyScratchData(const AssemblyScratchData &scratch_data);

    FEValues<dim> fe_values;
    FEFaceValues<dim> fe_face_values;
};

struct AssemblyCopyData {
    FullMatrix<double> cell_matrix;
    Vector<double> cell_rhs;
    std::vector<types::global_dof_index> local_dof_indices;
};

void assemble_system();
void local_assemble_system(
    const typename DoFHandler<dim>::active_cell_iterator &cell,
    AssemblyScratchData &scratch, AssemblyCopyData &copy_data);
void copy_local_to_global(const AssemblyCopyData &copy_data);

```

The following functions again are as in previous examples, as are the subsequent variables.

```

void solve();
void refine_grid();
void output_results(const unsigned int cycle) const;

Triangulation<dim> triangulation;
DoFHandler<dim> dof_handler;

FE_Q<dim> fe;

ConstraintMatrix hanging_node_constraints;

SparsityPattern sparsity_pattern;
SparseMatrix<double> system_matrix;

Vector<double> solution;
Vector<double> system_rhs;
};

```

Equation data declaration

Next we declare a class that describes the advection field. This, of course, is a vector field with as many components as there are space dimensions. One could now use a class derived from the `Function` base class, as we have done for boundary values and coefficients in previous examples, but there is another possibility in the library, namely a base class that describes tensor valued functions. In contrast to the usual `Function` objects, we provide the compiler with knowledge on the size of the objects of the return type. This enables the compiler to generate efficient code, which is not so simple for usual vector-valued functions where memory has to be allocated on the heap (thus, the `Function::vector_value` function has to be given the address of an object into which the result is to be written, in order to avoid copying and memory allocation and deallocation on the heap). In addition to the known size, it is possible not only to return vectors, but also tensors of higher rank; however, this is not very often requested by applications, to be honest...

The interface of the `TensorFunction` class is relatively close to that of the `Function` class, so there is probably no need to comment in detail the following declaration:

```

template <int dim>
class AdvectionField : public TensorFunction<1, dim> {
public:
    AdvectionField() : TensorFunction<1, dim>() {}

    virtual Tensor<1, dim> value(const Point<dim> &p) const;

    virtual void value_list(const std::vector<Point<dim>> &points,
                           std::vector<Tensor<1, dim>> &values) const;
};

```

In previous examples, we have used assertions that throw exceptions in several places. However, we have never seen how such exceptions are declared. This can be done as follows:

```
DeclException2(ExcDimensionMismatch, unsigned int, unsigned int,
    << "The vector has size " << arg1 << " but should have "
    << arg2 << " elements.");
```

The syntax may look a little strange, but is reasonable. The format is basically as follows: use the name of one of the macros `DeclExceptionN`, where `N` denotes the number of additional parameters which the exception object shall take. In this case, as we want to throw the exception when the sizes of two vectors differ, we need two arguments, so we use `DeclException2`. The first parameter then describes the name of the exception, while the following declare the data types of the parameters. The last argument is a sequence of output directives that will be piped into the `std::cerr` object, thus the strange format with the leading `<<` operator and the like. Note that we can access the parameters which are passed to the exception upon construction (i.e. within the `Assert` call) by using the names `arg1` through `argN`, where `N` is the number of arguments as defined by the use of the respective macro `DeclExceptionN`.

To learn how the preprocessor expands this macro into actual code, please refer to the documentation of the exception classes in the base library. Suffice it to say that by this macro call, the respective exception class is declared, which also has error output functions already implemented.

```
};
```

The following two functions implement the interface described above. The first simply implements the function as described in the introduction, while the second uses the same trick to avoid calling a virtual function as has already been introduced in the previous example program. Note the check for the right sizes of the arguments in the second function, which should always be present in such functions; it is our experience that many if not most programming errors result from incorrectly initialized arrays, incompatible parameters to functions and the like; using assertion as in this case can eliminate many of these problems.

```
template <int dim>
Tensor<1, dim> AdvectionField<dim>::value(const Point<dim> &p) const
{
    Point<dim> value;
    value[0] = 2;
    for (unsigned int i = 1; i < dim; ++i)
        value[i] = 1 + 0.8 * std::sin(8 * numbers::PI * p[0]);

    return value;
}

template <int dim>
void AdvectionField<dim>::value_list(const std::vector<Point<dim>> &points,
                                     std::vector<Tensor<1, dim>> &values) const
{
    Assert(values.size() == points.size(),
        ExcDimensionMismatch(values.size(), points.size()));

    for (unsigned int i = 0; i < points.size(); ++i)
        values[i] = AdvectionField<dim>::value(points[i]);
}
```

Besides the advection field, we need two functions describing the source terms (right hand side) and the boundary values. First for the right hand side, which follows the same pattern as in previous examples. As described in the introduction, the source is a constant function in the vicinity of a source point, which we denote by the constant static variable `center_point`. We set the values of this center using the same template tricks as we have shown in the step-7 example program. The rest is simple and has been shown previously, including the way to avoid virtual function calls in the `value_list` function.

```
template <int dim>
class RightHandSide : public Function<dim> {
public:
    RightHandSide() : Function<dim>() {}

    virtual double value(const Point<dim> &p,
                        const unsigned int component = 0) const;
```



```

        virtual void value_list(const std::vector<Point<dim>> &points,
                                std::vector<double> &values,
                                const unsigned int component = 0) const;

private:
    static const Point<dim> center_point;
};

template <>
const Point<1> RightHandSide<1>::center_point = Point<1>(-0.75);

template <>
const Point<2> RightHandSide<2>::center_point = Point<2>(-0.75, -0.75);

template <>
const Point<3> RightHandSide<3>::center_point = Point<3>(-0.75, -0.75, -0.75);

```

The only new thing here is that we check for the value of the `component` parameter. As this is a scalar function, it is obvious that it only makes sense if the desired component has the index zero, so we assert that this is indeed the case. `ExcIndexRange` is a global predefined exception (probably the one most often used, we therefore made it global instead of local to some class), that takes three parameters: the index that is outside the allowed range, the first element of the valid range and the one past the last (i.e. again the half-open interval so often used in the C++ standard library):

```

template <int dim>
double RightHandSide<dim>::value(const Point<dim> &p,
                                const unsigned int component) const
{
    (void)component;
    Assert(component == 0, ExcIndexRange(component, 0, 1));
    const double diameter = 0.1;
    return ((p - center_point).norm_square() < diameter * diameter
            ? .1 / std::pow(diameter, dim)
            : 0);
}

template <int dim>
void RightHandSide<dim>::value_list(const std::vector<Point<dim>> &points,
                                    std::vector<double> &values,
                                    const unsigned int component) const
{
    Assert(values.size() == points.size(),
           ExcDimensionMismatch(values.size(), points.size()));

    for (unsigned int i = 0; i < points.size(); ++i)
        values[i] = RightHandSide<dim>::value(points[i], component);
}

```

Finally for the boundary values, which is just another class derived from the `Function` base class:

```

template <int dim>
class BoundaryValues : public Function<dim> {
public:
    BoundaryValues() : Function<dim>() {}

    virtual double value(const Point<dim> &p,
                        const unsigned int component = 0) const;

    virtual void value_list(const std::vector<Point<dim>> &points,
                            std::vector<double> &values,
                            const unsigned int component = 0) const;
};

template <int dim>
double BoundaryValues<dim>::value(const Point<dim> &p,
                                const unsigned int component) const
{
    (void)component;
    Assert(component == 0, ExcIndexRange(component, 0, 1));

    const double sine_term =
        std::sin(16 * numbers::PI * std::sqrt(p.norm_square()));
    const double weight = std::exp(-5 * p.norm_square()) / std::exp(-5.);
    return sine_term * weight;
}

```

```

}

template <int dim>
void BoundaryValues<dim>::value_list(const std::vector<Point<dim>> &points,
                                     std::vector<double> &values,
                                     const unsigned int component) const
{
    Assert(values.size() == points.size(),
           ExcDimensionMismatch(values.size(), points.size()));

    for (unsigned int i = 0; i < points.size(); ++i)
        values[i] = BoundaryValues<dim>::value(points[i], component);
}

```

GradientEstimation class declaration

Now, finally, here comes the class that will compute the difference approximation of the gradient on each cell and weighs that with a power of the mesh size, as described in the introduction. This class is a simple version of the `DerivativeApproximation` class in the library, that uses similar techniques to obtain finite difference approximations of the gradient of a finite element field, or of higher derivatives.

The class has one public static function `estimate` that is called to compute a vector of error indicators, and a few private functions that do the actual work on all active cells. As in other parts of the library, we follow an informal convention to use vectors of floats for error indicators rather than the common vectors of doubles, as the additional accuracy is not necessary for estimated values.

In addition to these two functions, the class declares two exceptions which are raised when a cell has no neighbors in each of the space directions (in which case the matrix described in the introduction would be singular and can't be inverted), while the other one is used in the more common case of invalid parameters to a function, namely a vector of wrong size.

Two other comments: first, the class has no non-static member functions or variables, so this is not really a class, but rather serves the purpose of a `namespace` in C++. The reason that we chose a class over a namespace is that this way we can declare functions that are private. This can be done with namespaces as well, if one declares some functions in header files in the namespace and implements these and other functions in the implementation file. The functions not declared in the header file are still in the namespace but are not callable from outside. However, as we have only one file here, it is not possible to hide functions in the present case.

The second comment is that the dimension template parameter is attached to the function rather than to the class itself. This way, you don't have to specify the template parameter yourself as in most other cases, but the compiler can figure its value out itself from the dimension of the DoF handler object that one passes as first argument.

Before jumping into the fray with the implementation, let us also comment on the parallelization strategy. We have already introduced the necessary framework for using the `WorkStream` concept in the declaration of the main class of this program above. We will use it again here. In the current context, this means that we have to define (i) classes for scratch and copy objects, (ii) a function that does the local computation on one cell, and (iii) a function that copies the local result into a global object. Given this general framework, we will, however, deviate from it a bit. In particular, `WorkStream` was generally invented for cases where each local computation on a cell *adds* to a global object – for example, when assembling linear systems where we add local contributions into a global matrix and right hand side. `WorkStream` is designed to handle the potential conflict of multiple threads trying to do this addition at the same time, and consequently has to provide for some way to ensure that only thread gets to do this at a time. Here, however, the situation is slightly different: we compute contributions from every cell individually, but then all we need to do is put them into an element of an output vector that is unique to each cell. Consequently, there is no risk that the write operations from two cells might conflict, and the elaborate machinery of `WorkStream` to avoid conflicting writes is not necessary. Consequently, what we will do is this: We still need a scratch object that holds, for example, the `FEValues` object. However, we only create a fake, empty copy data structure. Likewise, we do need the function that computes local contributions, but since it can already put the result into its final location, we do not need a copy-local-to-global function and will instead give the `WorkStream::run()` function an empty function object – the equivalent to a `NULL` function pointer.

```

class GradientEstimation {
public:
    template <int dim>
    static void estimate(const DoFHandler<dim> &dof,
                       const Vector<double> &solution,
                       Vector<float> &error_per_cell);

    DeclException2(ExcInvalidVectorLength, int, int,
                  << "Vector has length " << arg1 << ", but should have "
                  << arg2);
    DeclException0(ExcInsufficientDirections);

private:
    template <int dim>
    struct EstimateScratchData {
        EstimateScratchData(const FiniteElement<dim> &fe,
                           const Vector<double> &solution,
                           Vector<float> &error_per_cell);
        EstimateScratchData(const EstimateScratchData &data);

        FEValues<dim> fe_midpoint_value;
        const Vector<double> &solution;
        Vector<float> &error_per_cell;
    };

    struct EstimateCopyData {};

    template <int dim>
    static void estimate_cell(
        const typename DoFHandler<dim>::active_cell_iterator &cell,
        EstimateScratchData<dim> &scratch_data,
        const EstimateCopyData &copy_data);
};

```

AdvectionProblem class implementation

Now for the implementation of the main class. Constructor, destructor and the function `setup_system` follow the same pattern that was used previously, so we need not comment on these three function:

```

template <int dim>
AdvectionProblem<dim>::AdvectionProblem() : dof_handler(triangulation), fe(1)
{}

template <int dim>
AdvectionProblem<dim>::~~AdvectionProblem()
{
    dof_handler.clear();
}

template <int dim>
void AdvectionProblem<dim>::setup_system()
{
    dof_handler.distribute_dofs(fe);
    hanging_node_constraints.clear();
    DoFTools::make_hanging_node_constraints(dof_handler,
                                           hanging_node_constraints);
    hanging_node_constraints.close();

    DynamicSparsityPattern dsp(dof_handler.n_dofs(), dof_handler.n_dofs());
    DoFTools::make_sparsity_pattern(dof_handler, dsp, hanging_node_constraints,
                                   /* keep_constrained_dofs = */ true);
    sparsity_pattern.copy_from(dsp);

    system_matrix.reinit(sparsity_pattern);

    solution.reinit(dof_handler.n_dofs());
    system_rhs.reinit(dof_handler.n_dofs());
}

```

In the following function, the matrix and right hand side are assembled. As stated in the documentation of the main class above, it does not do this itself, but rather delegates to the function following next, utilizing the WorkStream concept discussed in threads .

If you have looked through the threads module, you will have seen that assembling in parallel does not take an incredible amount of extra code as long as you diligently describe what the scratch and copy data objects are, and if you define suitable functions for the local assembly and the copy operation from local contributions to global objects. This done, the following will do all the heavy lifting to get these operations done on multiple threads on as many cores as you have in your system:

```
template <int dim>
void AdvectionProblem<dim>::assemble_system()
{
    WorkStream::run(dof_handler.begin_active(), dof_handler.end(), *this,
        &AdvectionProblem::local_assemble_system,
        &AdvectionProblem::copy_local_to_global,
        AssemblyScratchData(fe), AssemblyCopyData());
}
```

After the matrix has been assembled in parallel, we still have to eliminate hanging node constraints. This is something that can't be done on each of the threads separately, so we have to do it now. Note also, that unlike in previous examples, there are no boundary conditions to be applied to the system of equations. This, of course, is due to the fact that we have included them into the weak formulation of the problem.

```
hanging_node_constraints.condense(system_matrix);
hanging_node_constraints.condense(system_rhs);
}
```

As already mentioned above, we need to have scratch objects for the parallel computation of local contributions. These objects contain FEValues and FEFaceValues objects, and so we will need to have constructors and copy constructors that allow us to create them. In initializing them, note first that we use bilinear elements, so Gauss formulae with two points in each space direction are sufficient. For the cell terms we need the values and gradients of the shape functions, the quadrature points in order to determine the source density and the advection field at a given point, and the weights of the quadrature points times the determinant of the Jacobian at these points. In contrast, for the boundary integrals, we don't need the gradients, but rather the normal vectors to the cells. This determines which update flags we will have to pass to the constructors of the members of the class:

```
template <int dim>
AdvectionProblem<dim>::AssemblyScratchData::AssemblyScratchData(
    const FiniteElement<dim> &fe)
: fe_values(fe, QGauss<dim>(2),
    update_values | update_gradients | update_quadrature_points |
    update_JxW_values),
  fe_face_values(fe, QGauss<dim - 1>(2),
    update_values | update_quadrature_points |
    update_JxW_values | update_normal_vectors)
{}

template <int dim>
AdvectionProblem<dim>::AssemblyScratchData::AssemblyScratchData(
    const AssemblyScratchData &scratch_data)
: fe_values(scratch_data.fe_values.get_fe(),
    scratch_data.fe_values.get_quadrature(),
    update_values | update_gradients | update_quadrature_points |
    update_JxW_values),
  fe_face_values(scratch_data.fe_face_values.get_fe(),
    scratch_data.fe_face_values.get_quadrature(),
    update_values | update_quadrature_points |
    update_JxW_values | update_normal_vectors)
{}
}
```

Now, this is the function that does the actual work. It is not very different from the `assemble_system` functions of previous example programs, so we will again only comment on the differences. The mathematical stuff follows closely what we have said in the introduction.

There are a number of points worth mentioning here, though. The first one is that we have moved the FEValues and FEFaceValues objects into the ScratchData object. We have done so because the alternative would have been to simply create one every time we get into this function – i.e., on every cell. It now turns out that the FEValues classes were written with the explicit goal of moving everything that remains the same from cell to cell into the construction

of the object, and only do as little work as possible in `FEValues::reinit()` whenever we move to a new cell. What this means is that it would be very expensive to create a new object of this kind in this function as we would have to do it for every cell – exactly the thing we wanted to avoid with the `FEValues` class. Instead, what we do is create it only once (or a small number of times) in the scratch objects and then re-use it as often as we can.

This begs the question of whether there are other objects we create in this function whose creation is expensive compared to its use. Indeed, at the top of the function, we declare all sorts of objects. The `AdvectionField`, `RightHandSide` and `BoundaryValues` do not cost much to create, so there is no harm here. However, allocating memory in creating the `rhs_values` and similar variables below typically costs a significant amount of time, compared to just accessing the (temporary) values we store in them. Consequently, these would be candidates for moving into the `AssemblyScratchData` class. We will leave this as an exercise.

```
template <int dim>
void AdvectionProblem<dim>::local_assemble_system(
    const typename DoFHandler<dim>::active_cell_iterator &cell,
    AssemblyScratchData &scratch_data, AssemblyCopyData &copy_data)
{
```

First of all, we will need some objects that describe boundary values, right hand side function and the advection field. As we will only perform actions on these objects that do not change them, we declare them as constant, which can enable the compiler in some cases to perform additional optimizations.

```
const AdvectionField<dim> advection_field;
const RightHandSide<dim> right_hand_side;
const BoundaryValues<dim> boundary_values;
```

Then we define some abbreviations to avoid unnecessarily long lines:

```
const unsigned int dofs_per_cell = fe.dofs_per_cell;
const unsigned int n_q_points =
    scratch_data.fe_values.get_quadrature().size();
const unsigned int n_face_q_points =
    scratch_data.fe_face_values.get_quadrature().size();
```

We declare cell matrix and cell right hand side...

```
copy_data.cell_matrix.reinit(dofs_per_cell, dofs_per_cell);
copy_data.cell_rhs.reinit(dofs_per_cell);
```

... an array to hold the global indices of the degrees of freedom of the cell on which we are presently working...

```
copy_data.local_dof_indices.resize(dofs_per_cell);
```

... and array in which the values of right hand side, advection direction, and boundary values will be stored, for cell and face integrals respectively:

```
std::vector<double> rhs_values(n_q_points);
std::vector<Tensor<1, dim>> advection_directions(n_q_points);
std::vector<double> face_boundary_values(n_face_q_points);
std::vector<Tensor<1, dim>> face_advection_directions(n_face_q_points);
```

... then initialize the `FEValues` object...

```
scratch_data.fe_values.reinit(cell);
```

... obtain the values of right hand side and advection directions at the quadrature points...

```
advection_field.value_list(scratch_data.fe_values.get_quadrature_points(),
                           advection_directions);
right_hand_side.value_list(scratch_data.fe_values.get_quadrature_points(),
                           rhs_values);
```

... set the value of the streamline diffusion parameter as described in the introduction...

```
const double delta = 0.1 * cell->diameter();
```

... and assemble the local contributions to the system matrix and right hand side as also discussed above:

```
for (unsigned int q_point = 0; q_point < n_q_points; ++q_point)
  for (unsigned int i = 0; i < dofs_per_cell; ++i) {
    for (unsigned int j = 0; j < dofs_per_cell; ++j)
      copy_data.cell_matrix(i, j) +=
        ((advection_directions[q_point] *
          scratch_data.fe_values.shape_grad(j, q_point) *
          (scratch_data.fe_values.shape_value(i, q_point) +
            delta *
              (advection_directions[q_point] *
                scratch_data.fe_values.shape_grad(i, q_point)))) *
          scratch_data.fe_values.JxW(q_point));

    copy_data.cell_rhs(i) +=
      ((scratch_data.fe_values.shape_value(i, q_point) +
        delta * (advection_directions[q_point] *
          scratch_data.fe_values.shape_grad(i, q_point))) *
        rhs_values[q_point] * scratch_data.fe_values.JxW(q_point));
  }
```

Besides the cell terms which we have built up now, the bilinear form of the present problem also contains terms on the boundary of the domain. Therefore, we have to check whether any of the faces of this cell are on the boundary of the domain, and if so assemble the contributions of this face as well. Of course, the bilinear form only contains contributions from the `inflow` part of the boundary, but to find out whether a certain part of a face of the present cell is part of the inflow boundary, we have to have information on the exact location of the quadrature points and on the direction of flow at this point; we obtain this information using the `FEFaceValues` object and only decide within the main loop whether a quadrature point is on the inflow boundary.

```
for (unsigned int face = 0; face < GeometryInfo<dim>::faces_per_cell;
    ++face)
  if (cell->face(face)->at_boundary()) {
```

Ok, this face of the present cell is on the boundary of the domain. Just as for the usual `FEValues` object which we have used in previous examples and also above, we have to reinitialize the `FEFaceValues` object for the present face:

```
scratch_data.fe_face_values.reinit(cell, face);
```

For the quadrature points at hand, we ask for the values of the inflow function and for the direction of flow:

```
boundary_values.value_list(
  scratch_data.fe_face_values.get_quadrature_points(),
  face_boundary_values);
advection_field.value_list(
  scratch_data.fe_face_values.get_quadrature_points(),
  face_advection_directions);
```

Now loop over all quadrature points and see whether it is on the inflow or outflow part of the boundary. This is determined by a test whether the advection direction points inwards or outwards of the domain (note that the normal vector points outwards of the cell, and since the cell is at the boundary, the normal vector points outward of the domain, so if the advection direction points into the domain, its scalar product with the normal vector must be negative):

```
for (unsigned int q_point = 0; q_point < n_face_q_points; ++q_point)
    if (scratch_data.fe_face_values.normal_vector(q_point) *
        face_advection_directions[q_point] <
            0)
```

If the is part of the inflow boundary, then compute the contributions of this face to the global matrix and right hand side, using the values obtained from the FEFaceValues object and the formulae discussed in the introduction:

```
for (unsigned int i = 0; i < dofs_per_cell; ++i) {
    for (unsigned int j = 0; j < dofs_per_cell; ++j)
        copy_data.cell_matrix(i, j) -=
            (face_advection_directions[q_point] *
             scratch_data.fe_face_values.normal_vector(
                 q_point) *
             scratch_data.fe_face_values.shape_value(
                 i, q_point) *
             scratch_data.fe_face_values.shape_value(
                 j, q_point) *
             scratch_data.fe_face_values.JxW(q_point));

    copy_data.cell_rhs(i) -=
        (face_advection_directions[q_point] *
         scratch_data.fe_face_values.normal_vector(
             q_point) *
         face_boundary_values[q_point] *
         scratch_data.fe_face_values.shape_value(i,
                                                     q_point) *
         scratch_data.fe_face_values.JxW(q_point));
}
}
```

Now go on by transferring the local contributions to the system of equations into the global objects. The first step was to obtain the global indices of the degrees of freedom on this cell.

```
cell->get_dof_indices(copy_data.local_dof_indices);
}
```

The second function we needed to write was the one that copies the local contributions the previous function has computed and put into the copy data object, into the global matrix and right hand side vector objects. This is essentially what we always had as the last block of code when assembling something on every cell. The following should therefore be pretty obvious:

```
template <int dim>
void AdvectionProblem<dim>::copy_local_to_global(
    const AssemblyCopyData &copy_data)
{
    for (unsigned int i = 0; i < copy_data.local_dof_indices.size(); ++i) {
        for (unsigned int j = 0; j < copy_data.local_dof_indices.size(); ++j)
            system_matrix.add(copy_data.local_dof_indices[i],
                              copy_data.local_dof_indices[j],
                              copy_data.cell_matrix(i, j));

        system_rhs(copy_data.local_dof_indices[i]) += copy_data.cell_rhs(i);
    }
}
```

Following is the function that solves the linear system of equations. As the system is no more symmetric positive definite as in all the previous examples, we can't use the Conjugate Gradients method anymore. Rather, we use a solver that is tailored to nonsymmetric systems like the one at hand, the BiCGStab method. As preconditioner, we use the Block Jacobi method.

```
template <int dim>
void AdvectionProblem<dim>::solve()
{
```

Assert that the system be symmetric.

```
Assert(system_matrix.m() == system_matrix.n(), ExcNotQuadratic());
auto num_rows = system_matrix.m();
```

Make a copy of the rhs to use with Ginkgo.

```
std::vector<double> rhs(num_rows);
std::copy(system_rhs.begin(), system_rhs.begin() + num_rows, rhs.begin());
```

Ginkgo setup Some shortcuts: A vector is a Dense matrix with co-dimension 1. The matrix is setup in CSR. But various formats can be used. Look at Ginkgo's documentation.

```
using vec = gko::matrix::Dense<>;
using mtx = gko::matrix::Csr<>;
using bicgstab = gko::solver::Bicgstab<>;
using bj = gko::preconditioner::Jacobi<>;
using val_array = gko::Array<double>;
```

Where the code is to be executed. Can be changed to `omp` or `cuda` to run on multiple threads or on gpu's

```
std::shared_ptr<gko::Executor> exec = gko::ReferenceExecutor::create();
```

Setup Ginkgo's data structures

```
auto b = vec::create(exec, gko::dim<2>(num_rows, 1),
                    val_array::view(exec, num_rows, rhs.data()), 1);
auto x = vec::create(exec, gko::dim<2>(num_rows, 1));
auto A = mtx::create(exec, gko::dim<2>(num_rows),
                    system_matrix.n_nonzero_elements());
mtx::value_type *values = A->get_values();
mtx::index_type *row_ptr = A->get_row_ptrs();
mtx::index_type *col_idx = A->get_col_idxs();
```

Convert to standard CSR format As deal.ii does not expose its system matrix pointers, we construct them individually.

```
row_ptr[0] = 0;
for (auto row = 1; row <= num_rows; ++row) {
    row_ptr[row] = row_ptr[row - 1] + system_matrix.get_row_length(row - 1);
}

std::vector<mtx::index_type> ptrs(num_rows + 1);
std::copy(A->get_row_ptrs(), A->get_row_ptrs() + num_rows + 1,
          ptrs.begin());
for (auto row = 0; row < system_matrix.m(); ++row) {
    for (auto p = system_matrix.begin(row); p != system_matrix.end(row);
         ++p) {
```

write entry into the first free one for this row

```
col_idx[ptrs[row]] = p->column();
values[ptrs[row]] = p->value();
```

then move pointer ahead


```

        ++ptrs[row];
    }
}

```

Ginkgo solve The stopping criteria is set at maximum iterations of 1000 and a reduction factor of 1e-12. For other options, refer to Ginkgo's documentation.

```

auto solver_gen =
    bicgstab::build()
        .with_criteria(
            gko::stop::Iteration::build().with_max_iters(1000).on(exec),
            gko::stop::ResidualNormReduction<>::build()
                .with_reduction_factor(1e-12)
                .on(exec))
        .with_preconditioner(bj::build().on(exec))
        .on(exec);
auto solver = solver_gen->generate(gko::give(A));

```

Solve system

```

solver->apply(gko::lend(b), gko::lend(x));

```

Copy the solution vector back to deal.ii's data structures.

```

std::copy(x->get_values(), x->get_values() + num_rows, solution.begin());

/ *****
* deal.ii internal solver. Here for reference.
SolverControl      solver_control(1000, 1e-12);
SolverBicgstab<>    bicgstab(solver_control);

PreconditionJacobi<> preconditioner;
preconditioner.initialize(system_matrix, 1.0);

bicgstab.solve(system_matrix, solution, system_rhs,
               preconditioner);
***** /

```

Give the solution back to deal.ii

```

    hanging_node_constraints.distribute(solution);
}

```

The following function refines the grid according to the quantity described in the introduction. The respective computations are made in the class `GradientEstimation`. The only difference to previous examples is that we refine a little more aggressively (0.5 instead of 0.3 of the number of cells).

```

template <int dim>
void AdvectionProblem<dim>::refine_grid()
{
    Vector<float> estimated_error_per_cell(triangulation.n_active_cells());

    GradientEstimation::estimate(dof_handler, solution,
                                estimated_error_per_cell);

    GridRefinement::refine_and_coarsen_fixed_number(
        triangulation, estimated_error_per_cell, 0.5, 0.03);

    triangulation.execute_coarsening_and_refinement();
}

```

Writing output to disk is done in the same way as in the previous examples. Indeed, the function is identical to the one in step-6.

```

template <int dim>
void AdvectionProblem<dim>::output_results(const unsigned int cycle) const
{
    {
        GridOut grid_out;
        std::ofstream output("grid-" + std::to_string(cycle) + ".eps");
        grid_out.write_eps(triangulation, output);
    }

    {
        DataOut<dim> data_out;
        data_out.attach_dof_handler(dof_handler);
        data_out.add_data_vector(solution, "solution");
        data_out.build_patches();

        std::ofstream output("solution-" + std::to_string(cycle) + ".vtk");
        data_out.write_vtk(output);
    }
}

```

... as is the main loop (setup – solve – refine)

```

template <int dim>
void AdvectionProblem<dim>::run()
{
    for (unsigned int cycle = 0; cycle < 6; ++cycle) {
        std::cout << "Cycle " << cycle << ' ' << std::endl;

        if (cycle == 0) {
            GridGenerator::hyper_cube(triangulation, -1, 1);
            triangulation.refine_global(4);
        } else {
            refine_grid();
        }

        std::cout << "    Number of active cells:      "
                  << triangulation.n_active_cells() << std::endl;

        setup_system();

        std::cout << "    Number of degrees of freedom: " << dof_handler.n_dofs()
                  << std::endl;

        assemble_system();
        solve();
        output_results(cycle);
    }
}

```

GradientEstimation class implementation

Now for the implementation of the GradientEstimation class. Let us start by defining constructors for the EstimateScratchData class used by the estimate_cell() function:

```

template <int dim>
GradientEstimation::EstimateScratchData<dim>::EstimateScratchData(
    const FiniteElement<dim> &fe, const Vector<double> &solution,
    Vector<float> &error_per_cell)
    : fe_midpoint_value(fe, QMidpoint<dim>()),
      update_values | update_quadrature_points(),
      solution(solution),
      error_per_cell(error_per_cell)
{}

template <int dim>
GradientEstimation::EstimateScratchData<dim>::EstimateScratchData(
    const EstimateScratchData &scratch_data)
    : fe_midpoint_value(scratch_data.fe_midpoint_value.get_fe(),
                        scratch_data.fe_midpoint_value.get_quadrature(),
                        update_values | update_quadrature_points(),
                        solution(scratch_data.solution),
                        error_per_cell(scratch_data.error_per_cell))
{}

```

Next for the implementation of the `GradientEstimation` class. The first function does not much except for delegating work to the other function, but there is a bit of setup at the top.

Before starting with the work, we check that the vector into which the results are written has the right size. Programming mistakes in which one forgets to size arguments correctly at the calling site are quite common. Because the resulting damage from not catching such errors is often subtle (e.g., corruption of data somewhere in memory, or non-reproducible results), it is well worth the effort to check for such things.

```
template <int dim>
void GradientEstimation::estimate(const DoFHandler<dim> &dof_handler,
                                const Vector<double> &solution,
                                Vector<float> &error_per_cell)
{
    Assert(error_per_cell.size() ==
           dof_handler.get_triangulation().n_active_cells(),
           ExcInvalidVectorLength(
               error_per_cell.size(),
               dof_handler.get_triangulation().n_active_cells()));

    WorkStream::run(dof_handler.begin_active(), dof_handler.end(),
                    &GradientEstimation::template estimate_cell<dim>,
                    std::function<void(const EstimateCopyData &)>(),
                    EstimateScratchData<dim>(dof_handler.get_fe(), solution,
                                              error_per_cell),
                    EstimateCopyData());
}
```

Following now the function that actually computes the finite difference approximation to the gradient. The general outline of the function is to first compute the list of active neighbors of the present cell and then compute the quantities described in the introduction for each of the neighbors. The reason for this order is that it is not a one-liner to find a given neighbor with locally refined meshes. In principle, an optimized implementation would find neighbors and the quantities depending on them in one step, rather than first building a list of neighbors and in a second step their contributions but we will gladly leave this as an exercise. As discussed before, the worker function passed to `WorkStream::run` works on "scratch" objects that keep all temporary objects. This way, we do not need to create and initialize objects that are expensive to initialize within the function that does the work, every time it is called for a given cell. Such an argument is passed as the second argument. The third argument would be a "copy-data" object (see threads for more information) but we do not actually use any of these here. Because `WorkStream::run()` insists on passing three arguments, we declare this function with three arguments, but simply ignore the last one.

(This is unsatisfactory from an esthetic perspective. It can be avoided, at the cost of some other trickery. If you allow, let us here show how. First, assume that we had declared this function to only take two arguments by omitting the unused last one. Now, `WorkStream::run` still wants to call this function with three arguments, so we need to find a way to "forget" the third argument in the call. Simply passing `WorkStream::run` the pointer to the function as we do above will not do this – the compiler will complain that a function declared to have two arguments is called with three arguments. However, we can do this by passing the following as the third argument when calling `WorkStream::run()` above:

```
std::function<void (const typename DoFHandler<dim>::active_cell_iterator
&,
                    EstimateScratchData<dim> &,
                    EstimateCopyData &)>
(std::bind (&GradientEstimation::template estimate_cell<dim>,
           std::placeholders::_1,
           std::placeholders::_2))
```

This creates a function object taking three arguments, but when it calls the underlying function object, it simply only uses the first and second argument – we simply "forget" to use the third argument :-). In the end, this isn't completely obvious either, and so we didn't implement it, but hey – it can be done!

Now for the details:

```
template <int dim>
void GradientEstimation::estimate_cell(
    const typename DoFHandler<dim>::active_cell_iterator &cell,
    EstimateScratchData<dim> &scratch_data, const EstimateCopyData &)
{
```

We need space for the tensor Y , which is the sum of outer products of the y -vectors.

```
Tensor<2, dim> Y;
```

Then we allocate a vector to hold iterators to all active neighbors of a cell. We reserve the maximal number of active neighbors in order to avoid later reallocations. Note how this maximal number of active neighbors is computed here.

```
std::vector<typename DoFHandler<dim>::active_cell_iterator>
    active_neighbors;
active_neighbors.reserve(GeometryInfo<dim>::faces_per_cell *
    GeometryInfo<dim>::max_children_per_face);
```

First initialize the `FEValues` object, as well as the Y tensor:

```
scratch_data.fe_midpoint_value.reinit(cell);
```

Then allocate the vector that will be the sum over the y -vectors times the approximate directional derivative:

```
Tensor<1, dim> projected_gradient;
```

Now before going on first compute a list of all active neighbors of the present cell. We do so by first looping over all faces and see whether the neighbor there is active, which would be the case if it is on the same level as the present cell or one level coarser (note that a neighbor can only be once coarser than the present cell, as we only allow a maximal difference of one refinement over a face in deal.II). Alternatively, the neighbor could be on the same level and be further refined; then we have to find which of its children are next to the present cell and select these (note that if a child of a neighbor of an active cell that is next to this active cell, needs necessarily be active itself, due to the one-refinement rule cited above).

Things are slightly different in one space dimension, as there the one-refinement rule does not exist: neighboring active cells may differ in as many refinement levels as they like. In this case, the computation becomes a little more difficult, but we will explain this below.

Before starting the loop over all neighbors of the present cell, we have to clear the array storing the iterators to the active neighbors, of course.

```
active_neighbors.clear();
for (unsigned int face_no = 0; face_no < GeometryInfo<dim>::faces_per_cell;
    ++face_no)
    if (!cell->at_boundary(face_no)) {
```

First define an abbreviation for the iterator to the face and the neighbor

```
const typename DoFHandler<dim>::face_iterator face =
    cell->face(face_no);
const typename DoFHandler<dim>::cell_iterator neighbor =
    cell->neighbor(face_no);
```

Then check whether the neighbor is active. If it is, then it is on the same level or one level coarser (if we are not in 1D), and we are interested in it in any case.

```
if (neighbor->active())
    active_neighbors.push_back(neighbor);
else {
```

If the neighbor is not active, then check its children.

```
if (dim == 1) {
```

To find the child of the neighbor which bounds to the present cell, successively go to its right child if we are left of the present cell ($n==0$), or go to the left child if we are on the right ($n==1$), until we find an active cell.

```
typename DoFHandler<dim>::cell_iterator neighbor_child =
    neighbor;
while (neighbor_child->has_children())
    neighbor_child =
        neighbor_child->child(face_no == 0 ? 1 : 0);
```

As this used some non-trivial geometrical intuition, we might want to check whether we did it right, i.e. check whether the neighbor of the cell we found is indeed the cell we are presently working on. Checks like this are often useful and have frequently uncovered errors both in algorithms like the line above (where it is simple to involuntarily exchange $n==1$ for $n==0$ or the like) and in the library (the assumptions underlying the algorithm above could either be wrong, wrongly documented, or are violated due to an error in the library). One could in principle remove such checks after the program works for some time, but it might be a good things to leave it in anyway to check for changes in the library or in the algorithm above.

Note that if this check fails, then this is certainly an error that is irrecoverable and probably qualifies as an internal error. We therefore use a predefined exception class to throw here.

```
Assert(
    neighbor_child->neighbor(face_no == 0 ? 1 : 0) == cell,
    ExcInternalError());
```

If the check succeeded, we push the active neighbor we just found to the stack we keep:

```
    active_neighbors.push_back(neighbor_child);
} else
```

If we are not in 1d, we collect all neighbor children 'behind' the subfaces of the current face

```
for (unsigned int subface_no = 0;
     subface_no < face->n_children(); ++subface_no)
    active_neighbors.push_back(
        cell->neighbor_child_on_subface(face_no,
                                         subface_no));
}
```

OK, now that we have all the neighbors, lets start the computation on each of them. First we do some preliminaries: find out about the center of the present cell and the solution at this point. The latter is obtained as a vector of function values at the quadrature points, of which there are only one, of course. Likewise, the position of the center is the position of the first (and only) quadrature point in real space.

```
const Point<dim> this_center =
    scratch_data.fe_midpoint_value.quadrature_point(0);

std::vector<double> this_midpoint_value(1);
scratch_data.fe_midpoint_value.get_function_values(scratch_data.solution,
                                                    this_midpoint_value);
```

Now loop over all active neighbors and collect the data we need. Allocate a vector just like `this_midpoint_value` which we will use to store the value of the solution in the midpoint of the neighbor cell. We allocate it here already, since that way we don't have to allocate memory repeatedly in each iteration of this inner loop (memory allocation is a rather expensive operation):

```
std::vector<double> neighbor_midpoint_value(1);
typename std::vector<typename DoFHandler<dim>::active_cell_iterator>::
    const_iterator neighbor_ptr = active_neighbors.begin();
for (; neighbor_ptr != active_neighbors.end(); ++neighbor_ptr) {
```

First define an abbreviation for the iterator to the active neighbor cell:

```
const typename DoFHandler<dim>::active_cell_iterator neighbor =
    *neighbor_ptr;
```

Then get the center of the neighbor cell and the value of the finite element function thereon. Note that for this information we have to reinitialize the `FEValues` object for the neighbor cell.

```
scratch_data.fe_midpoint_value.reinit(neighbor);
const Point<dim> neighbor_center =
    scratch_data.fe_midpoint_value.quadrature_point(0);

scratch_data.fe_midpoint_value.get_function_values(
    scratch_data.solution, neighbor_midpoint_value);
```

Compute the vector \underline{y} connecting the centers of the two cells. Note that as opposed to the introduction, we denote by \underline{y} the normalized difference vector, as this is the quantity used everywhere in the computations.

```
Tensor<1, dim> y = neighbor_center - this_center;
const double distance = y.norm();
y /= distance;
```

Then add up the contribution of this cell to the \mathbf{Y} matrix...

```
for (unsigned int i = 0; i < dim; ++i)
    for (unsigned int j = 0; j < dim; ++j) Y[i][j] += y[i] * y[j];
```

... and update the sum of difference quotients:

```
projected_gradient +=
    (neighbor_midpoint_value[0] - this_midpoint_value[0]) / distance *
    y;
}
```

If now, after collecting all the information from the neighbors, we can determine an approximation of the gradient for the present cell, then we need to have passed over vectors \underline{y} which span the whole space, otherwise we would not have all components of the gradient. This is indicated by the invertibility of the matrix.

If the matrix should not be invertible, this means that the present cell had an insufficient number of active neighbors. In contrast to all previous cases, where we raised exceptions, this is, however, not a programming error: it is a runtime error that can happen in optimized mode even if it ran well in debug mode, so it is reasonable to try to catch this error also in optimized mode. For this case, there is the `AssertThrow` macro: it checks the condition like the `Assert` macro, but not only in debug mode; it then outputs an error message, but instead of terminating the program as in the case of the `Assert` macro, the exception is thrown using the `throw` command of C++. This way, one has the possibility to catch this error and take reasonable counter actions. One such measure would be to refine the grid globally, as the case of insufficient directions can not occur if every cell of the initial grid has been refined at least once.

```
AssertThrow(determinant(Y) != 0, ExcInsufficientDirections());
```

If, on the other hand the matrix is invertible, then invert it, multiply the other quantity with it and compute the estimated error using this quantity and the right powers of the mesh width:

```
const Tensor<2, dim> Y_inverse = invert(Y);
Tensor<1, dim> gradient = Y_inverse * projected_gradient;
```

The last part of this function is the one where we write into the element of the output vector what we have just computed. The address of this vector has been stored in the scratch data object, and all we have to do is know how to get at the correct element inside this vector – but we can ask the cell we're on the how-manyth active cell it is for this:

```
scratch_data.error_per_cell(cell->active_cell_index()) =
    (std::pow(cell->diameter(), 1 + 1.0 * dim / 2) *
     std::sqrt(gradient.norm_square()));
}
} // namespace Step9
```

Main function

The main function is similar to the previous examples. The main difference is that we use MultithreadInfo to set the maximum number of threads (see Parallel computing with multiple processors accessing shared memory" documentation module for more explanation). The number of threads used is the minimum of the environment variable DEAL_II_NUM_THREADS and the parameter of set_thread_limit. If no value is given to set_thread_limit, the default value from the Intel Threading Building Blocks (TBB) library is used. If the call to set_thread_limit is omitted, the number of threads will be chosen by TBB indepently of DEAL_II_NUM_THREADS.

```
int main()
{
    try {
        dealii::MultithreadInfo::set_thread_limit();

        Step9::AdvectionProblem<2> advection_problem_2d;
        advection_problem_2d.run();
    } catch (std::exception &exc) {
        std::cerr << std::endl
                  << std::endl
                  << "-----"
                  << std::endl;
        std::cerr << "Exception on processing: " << std::endl
                  << exc.what() << std::endl
                  << "Aborting!" << std::endl
                  << "-----"
                  << std::endl;

        return 1;
    } catch (...) {
        std::cerr << std::endl
                  << std::endl
                  << "-----"
                  << std::endl;
        std::cerr << "Unknown exception!" << std::endl
                  << "Aborting!" << std::endl
                  << "-----"
                  << std::endl;

        return 1;
    }

    return 0;
}
```

Results

Comments about programming and debugging

The plain program

```

/* -----
 *
 * Copyright (C) 2000 - 2018 by the deal.II authors
 *
 * This file is part of the deal.II library.
 *
 * The deal.II library is free software; you can use it, redistribute
 * it, and/or modify it under the terms of the GNU Lesser General
 * Public License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 * The full text of the license can be found in the file LICENSE at
 * the top level of the deal.II distribution.
 *
 * -----
 *
 * Author: Wolfgang Bangerth, University of Heidelberg, 2000
 */

/* -----
 *
 * This file has been taken verbatim from the deal.ii (version 9.0)
 * examples directory and modified.
 *
 * This example aims to demonstrate the ease with which Ginkgo can
 * be interfaced with other libraries. The only modification/ addition
 * has been to the AdvectionProblem::solve () function.
 *
 */

#include <deal.II/base/function.h>
#include <deal.II/base/logstream.h>
#include <deal.II/base/quadrature_lib.h>
#include <deal.II/dofs/dof_accessor.h>
#include <deal.II/dofs/dof_handler.h>
#include <deal.II/dofs/dof_tools.h>
#include <deal.II/fe/fe_q.h>
#include <deal.II/fe/fe_values.h>
#include <deal.II/grid/grid_generator.h>
#include <deal.II/grid/grid_out.h>
#include <deal.II/grid/grid_refinement.h>
#include <deal.II/grid/tria.h>
#include <deal.II/grid/tria_accessor.h>
#include <deal.II/grid/tria_iterator.h>
#include <deal.II/lac/constraint_matrix.h>
#include <deal.II/lac/dynamic_sparsity_pattern.h>
#include <deal.II/lac/full_matrix.h>
#include <deal.II/lac/precondition.h>
#include <deal.II/lac/solver_bicgstab.h>
#include <deal.II/lac/sparse_matrix.h>
#include <deal.II/lac/vector.h>
#include <deal.II/numerics/data_out.h>
#include <deal.II/numerics/matrix_tools.h>
#include <deal.II/numerics/vector_tools.h>

#include <deal.II/base/multithread_info.h>
#include <deal.II/base/work_stream.h>

#include <deal.II/base/tensor_function.h>

#include <deal.II/numerics/error_estimator.h>

#include <ginkgo/ginkgo.hpp>

#include <fstream>
#include <iostream>

namespace Step9 {
using namespace dealii;

template <int dim>
class AdvectionProblem {
public:

```

```

    AdvectionProblem();
    ~AdvectionProblem();
    void run();

private:
    void setup_system();

    struct AssemblyScratchData {
        AssemblyScratchData(const FiniteElement<dim> &fe);
        AssemblyScratchData(const AssemblyScratchData &scratch_data);

        FEValues<dim> fe_values;
        FEFaceValues<dim> fe_face_values;
    };

    struct AssemblyCopyData {
        FullMatrix<double> cell_matrix;
        Vector<double> cell_rhs;
        std::vector<types::global_dof_index> local_dof_indices;
    };

    void assemble_system();
    void local_assemble_system(
        const typename DoFHandler<dim>::active_cell_iterator &cell,
        AssemblyScratchData &scratch, AssemblyCopyData &copy_data);
    void copy_local_to_global(const AssemblyCopyData &copy_data);

    void solve();
    void refine_grid();
    void output_results(const unsigned int cycle) const;

    Triangulation<dim> triangulation;
    DoFHandler<dim> dof_handler;

    FE_Q<dim> fe;

    ConstraintMatrix hanging_node_constraints;

    SparsityPattern sparsity_pattern;
    SparseMatrix<double> system_matrix;

    Vector<double> solution;
    Vector<double> system_rhs;
};

template <int dim>
class AdvectionField : public TensorFunction<1, dim> {
public:
    AdvectionField() : TensorFunction<1, dim>() {}

    virtual Tensor<1, dim> value(const Point<dim> &p) const;

    virtual void value_list(const std::vector<Point<dim>> &points,
                           std::vector<Tensor<1, dim>> &values) const;

    DeclException2(ExcDimensionMismatch, unsigned int, unsigned int,
        << "The vector has size " << arg1 << " but should have "
        << arg2 << " elements.");
};

template <int dim>
Tensor<1, dim> AdvectionField<dim>::value(const Point<dim> &p) const
{
    Point<dim> value;
    value[0] = 2;
    for (unsigned int i = 1; i < dim; ++i)
        value[i] = 1 + 0.8 * std::sin(8 * numbers::PI * p[0]);

    return value;
}

template <int dim>
void AdvectionField<dim>::value_list(const std::vector<Point<dim>> &points,
                                     std::vector<Tensor<1, dim>> &values) const
{
    Assert(values.size() == points.size(),
        ExcDimensionMismatch(values.size(), points.size()));

    for (unsigned int i = 0; i < points.size(); ++i)
        values[i] = AdvectionField<dim>::value(points[i]);
}

```

```

template <int dim>
class RightHandSide : public Function<dim> {
public:
    RightHandSide() : Function<dim>() {}

    virtual double value(const Point<dim> &p,
                        const unsigned int component = 0) const;

    virtual void value_list(const std::vector<Point<dim>> &points,
                          std::vector<double> &values,
                          const unsigned int component = 0) const;

private:
    static const Point<dim> center_point;
};

template <>
const Point<1> RightHandSide<1>::center_point = Point<1>(-0.75);

template <>
const Point<2> RightHandSide<2>::center_point = Point<2>(-0.75, -0.75);

template <>
const Point<3> RightHandSide<3>::center_point = Point<3>(-0.75, -0.75, -0.75);

template <int dim>
double RightHandSide<dim>::value(const Point<dim> &p,
                                const unsigned int component) const
{
    (void)component;
    Assert(component == 0, ExcIndexRange(component, 0, 1));
    const double diameter = 0.1;
    return ((p - center_point).norm_square() < diameter * diameter
           ? .1 / std::pow(diameter, dim)
           : 0);
}

template <int dim>
void RightHandSide<dim>::value_list(const std::vector<Point<dim>> &points,
                                   std::vector<double> &values,
                                   const unsigned int component) const
{
    Assert(values.size() == points.size(),
           ExcDimensionMismatch(values.size(), points.size()));

    for (unsigned int i = 0; i < points.size(); ++i)
        values[i] = RightHandSide<dim>::value(points[i], component);
}

template <int dim>
class BoundaryValues : public Function<dim> {
public:
    BoundaryValues() : Function<dim>() {}

    virtual double value(const Point<dim> &p,
                        const unsigned int component = 0) const;

    virtual void value_list(const std::vector<Point<dim>> &points,
                          std::vector<double> &values,
                          const unsigned int component = 0) const;
};

template <int dim>
double BoundaryValues<dim>::value(const Point<dim> &p,
                                const unsigned int component) const
{
    (void)component;
    Assert(component == 0, ExcIndexRange(component, 0, 1));

    const double sine_term =
        std::sin(16 * numbers::PI * std::sqrt(p.norm_square()));
    const double weight = std::exp(-5 * p.norm_square()) / std::exp(-5.);
    return sine_term * weight;
}

template <int dim>
void BoundaryValues<dim>::value_list(const std::vector<Point<dim>> &points,
                                   std::vector<double> &values,
                                   const unsigned int component) const
{

```

```

    Assert(values.size() == points.size(),
           ExcDimensionMismatch(values.size(), points.size()));

    for (unsigned int i = 0; i < points.size(); ++i)
        values[i] = BoundaryValues<dim>::value(points[i], component);
}

class GradientEstimation {
public:
    template <int dim>
    static void estimate(const DoFHandler<dim> &dof,
                       const Vector<double> &solution,
                       Vector<float> &error_per_cell);

    DeclException2(ExcInvalidVectorLength, int, int,
                  << "Vector has length " << arg1 << ", but should have "
                  << arg2);
    DeclException0(ExcInsufficientDirections);

private:
    template <int dim>
    struct EstimateScratchData {
        EstimateScratchData(const FiniteElement<dim> &fe,
                           const Vector<double> &solution,
                           Vector<float> &error_per_cell);
        EstimateScratchData(const EstimateScratchData &data);

        FEValues<dim> fe_midpoint_value;
        const Vector<double> &solution;
        Vector<float> &error_per_cell;
    };

    struct EstimateCopyData {};

    template <int dim>
    static void estimate_cell(
        const typename DoFHandler<dim>::active_cell_iterator &cell,
        EstimateScratchData<dim> &scratch_data,
        const EstimateCopyData &copy_data);
};

template <int dim>
AdvectionProblem<dim>::AdvectionProblem() : dof_handler(triangulation), fe(1)
{}

template <int dim>
AdvectionProblem<dim>::~AdvectionProblem()
{
    dof_handler.clear();
}

template <int dim>
void AdvectionProblem<dim>::setup_system()
{
    dof_handler.distribute_dofs(fe);
    hanging_node_constraints.clear();
    DoFTools::make_hanging_node_constraints(dof_handler,
                                           hanging_node_constraints);
    hanging_node_constraints.close();

    DynamicSparsityPattern dsp(dof_handler.n_dofs(), dof_handler.n_dofs());
    DoFTools::make_sparsity_pattern(dof_handler, dsp, hanging_node_constraints,
                                   /*keep_constrained_dofs = */ true);
    sparsity_pattern.copy_from(dsp);

    system_matrix.reinit(sparsity_pattern);

    solution.reinit(dof_handler.n_dofs());
    system_rhs.reinit(dof_handler.n_dofs());
}

template <int dim>
void AdvectionProblem<dim>::assemble_system()
{
    WorkStream::run(dof_handler.begin_active(), dof_handler.end(), *this,
                   &AdvectionProblem::local_assemble_system,
                   &AdvectionProblem::copy_local_to_global,
                   AssemblyScratchData(fe), AssemblyCopyData());
}

```

```

    hanging_node_constraints.condense(system_matrix);
    hanging_node_constraints.condense(system_rhs);
}

template <int dim>
AdvectionProblem<dim>::AssemblyScratchData::AssemblyScratchData(
    const FiniteElement<dim> &fe)
: fe_values(fe, QGauss<dim>(2),
    update_values | update_gradients | update_quadrature_points |
    update_JxW_values),
  fe_face_values(fe, QGauss<dim - 1>(2),
    update_values | update_quadrature_points |
    update_JxW_values | update_normal_vectors)
{}

template <int dim>
AdvectionProblem<dim>::AssemblyScratchData::AssemblyScratchData(
    const AssemblyScratchData &scratch_data)
: fe_values(scratch_data.fe_values.get_fe(),
    scratch_data.fe_values.get_quadrature(),
    update_values | update_gradients | update_quadrature_points |
    update_JxW_values),
  fe_face_values(scratch_data.fe_face_values.get_fe(),
    scratch_data.fe_face_values.get_quadrature(),
    update_values | update_quadrature_points |
    update_JxW_values | update_normal_vectors)
{}

template <int dim>
void AdvectionProblem<dim>::local_assemble_system(
    const typename DoFHandler<dim>::active_cell_iterator &cell,
    AssemblyScratchData &scratch_data, AssemblyCopyData &copy_data)
{
    const AdvectionField<dim> advection_field;
    const RightHandSide<dim> right_hand_side;
    const BoundaryValues<dim> boundary_values;

    const unsigned int dofs_per_cell = fe.dofs_per_cell;
    const unsigned int n_q_points =
        scratch_data.fe_values.get_quadrature().size();
    const unsigned int n_face_q_points =
        scratch_data.fe_face_values.get_quadrature().size();

    copy_data.cell_matrix.reinit(dofs_per_cell, dofs_per_cell);
    copy_data.cell_rhs.reinit(dofs_per_cell);

    copy_data.local_dof_indices.resize(dofs_per_cell);

    std::vector<double> rhs_values(n_q_points);
    std::vector<Tensor<1, dim>> advection_directions(n_q_points);
    std::vector<double> face_boundary_values(n_face_q_points);
    std::vector<Tensor<1, dim>> face_advection_directions(n_face_q_points);

    scratch_data.fe_values.reinit(cell);

    advection_field.value_list(scratch_data.fe_values.get_quadrature_points(),
        advection_directions);
    right_hand_side.value_list(scratch_data.fe_values.get_quadrature_points(),
        rhs_values);

    const double delta = 0.1 * cell->diameter();

    for (unsigned int q_point = 0; q_point < n_q_points; ++q_point)
        for (unsigned int i = 0; i < dofs_per_cell; ++i) {
            for (unsigned int j = 0; j < dofs_per_cell; ++j)
                copy_data.cell_matrix(i, j) +=
                    ((advection_directions[q_point] *
                     scratch_data.fe_values.shape_grad(j, q_point) *
                     (scratch_data.fe_values.shape_value(i, q_point) +
                      delta *
                      (advection_directions[q_point] *
                       scratch_data.fe_values.shape_grad(i, q_point)))) *
                     scratch_data.fe_values.JxW(q_point));

            copy_data.cell_rhs(i) +=
                ((scratch_data.fe_values.shape_value(i, q_point) +
                 delta * (advection_directions[q_point] *
                  scratch_data.fe_values.shape_grad(i, q_point))) *
                 rhs_values[q_point] * scratch_data.fe_values.JxW(q_point));
        }

    for (unsigned int face = 0; face < GeometryInfo<dim>::faces_per_cell;

```

```

    ++face)
    if (cell->face(face)->at_boundary()) {
        scratch_data.fe_face_values.reinit(cell, face);

        boundary_values.value_list(
            scratch_data.fe_face_values.get_quadrature_points(),
            face_boundary_values);
        advection_field.value_list(
            scratch_data.fe_face_values.get_quadrature_points(),
            face_advection_directions);

        for (unsigned int q_point = 0; q_point < n_face_q_points; ++q_point)
            if (scratch_data.fe_face_values.normal_vector(q_point) *
                face_advection_directions[q_point] <
                    0)
                for (unsigned int i = 0; i < dofs_per_cell; ++i) {
                    for (unsigned int j = 0; j < dofs_per_cell; ++j)
                        copy_data.cell_matrix(i, j) -=
                            (face_advection_directions[q_point] *
                             scratch_data.fe_face_values.normal_vector(
                                 q_point) *
                             scratch_data.fe_face_values.shape_value(
                                 i, q_point) *
                             scratch_data.fe_face_values.shape_value(
                                 j, q_point) *
                             scratch_data.fe_face_values.JxW(q_point));

                    copy_data.cell_rhs(i) -=
                        (face_advection_directions[q_point] *
                         scratch_data.fe_face_values.normal_vector(
                             q_point) *
                         face_boundary_values[q_point] *
                         scratch_data.fe_face_values.shape_value(i,
                                                                     q_point) *
                         scratch_data.fe_face_values.JxW(q_point));
                }
            }

    cell->get_dof_indices(copy_data.local_dof_indices);
}

template <int dim>
void AdvectionProblem<dim>::copy_local_to_global(
    const AssemblyCopyData &copy_data)
{
    for (unsigned int i = 0; i < copy_data.local_dof_indices.size(); ++i) {
        for (unsigned int j = 0; j < copy_data.local_dof_indices.size(); ++j)
            system_matrix.add(copy_data.local_dof_indices[i],
                             copy_data.local_dof_indices[j],
                             copy_data.cell_matrix(i, j));

        system_rhs(copy_data.local_dof_indices[i]) += copy_data.cell_rhs(i);
    }
}

template <int dim>
void AdvectionProblem<dim>::solve()
{
    Assert(system_matrix.m() == system_matrix.n(), ExcNotQuadratic());
    auto num_rows = system_matrix.m();

    std::vector<double> rhs(num_rows);
    std::copy(system_rhs.begin(), system_rhs.begin() + num_rows, rhs.begin());

    using vec = gko::matrix::Dense<>;
    using mtx = gko::matrix::Csr<>;
    using bicgstab = gko::solver::Bicgstab<>;
    using bj = gko::preconditioner::Jacobi<>;
    using val_array = gko::Array<double>;

    std::shared_ptr<gko::Executor> exec = gko::ReferenceExecutor::create();

    auto b = vec::create(exec, gko::dim<2>(num_rows, 1),
                        val_array::view(exec, num_rows, rhs.data(), 1));
    auto x = vec::create(exec, gko::dim<2>(num_rows, 1));
    auto A = mtx::create(exec, gko::dim<2>(num_rows),
                        system_matrix.n_nonzero_elements());
    mtx::value_type *values = A->get_values();
    mtx::index_type *row_ptr = A->get_row_ptrs();
    mtx::index_type *col_idx = A->get_col_idxs();

    row_ptr[0] = 0;
    for (auto row = 1; row <= num_rows; ++row) {
        row_ptr[row] = row_ptr[row - 1] + system_matrix.get_row_length(row - 1);
    }
}

```

```

std::vector<mtx::index_type> ptrs(num_rows + 1);
std::copy(A->get_row_ptrs(), A->get_row_ptrs() + num_rows + 1,
          ptrs.begin());
for (auto row = 0; row < system_matrix.m(); ++row) {
    for (auto p = system_matrix.begin(row); p != system_matrix.end(row);
         ++p) {
        col_idx[ptrs[row]] = p->column();
        values[ptrs[row]] = p->value();

        ++ptrs[row];
    }
}

auto solver_gen =
    bicgstab::build()
        .with_criteria(
            gko::stop::Iteration::build().with_max_iters(1000).on(exec),
            gko::stop::ResidualNormReduction<>::build()
                .with_reduction_factor(1e-12)
                .on(exec))
        .with_preconditioner(bj::build().on(exec))
        .on(exec);
auto solver = solver_gen->generate(gko::give(A));

solver->apply(gko::lend(b), gko::lend(x));

std::copy(x->get_values(), x->get_values() + num_rows, solution.begin());

/*****
 * deal.ii internal solver. Here for reference.
 SolverControl          solver_control (1000, 1e-12);
 SolverBicgstab<>       bicgstab (solver_control);

 PreconditionJacobi<> preconditioner;
 preconditioner.initialize(system_matrix, 1.0);

 bicgstab.solve (system_matrix, solution, system_rhs,
                 preconditioner);
 *****/

hanging_node_constraints.distribute(solution);
}

template <int dim>
void AdvectionProblem<dim>::refine_grid()
{
    Vector<float> estimated_error_per_cell(triangulation.n_active_cells());

    GradientEstimation::estimate(dof_handler, solution,
                                estimated_error_per_cell);

    GridRefinement::refine_and_coarsen_fixed_number(
        triangulation, estimated_error_per_cell, 0.5, 0.03);

    triangulation.execute_coarsening_and_refinement();
}

template <int dim>
void AdvectionProblem<dim>::output_results(const unsigned int cycle) const
{
    {
        GridOut grid_out;
        std::ofstream output("grid-" + std::to_string(cycle) + ".eps");
        grid_out.write_eps(triangulation, output);
    }

    {
        DataOut<dim> data_out;
        data_out.attach_dof_handler(dof_handler);
        data_out.add_data_vector(solution, "solution");
        data_out.build_patches();

        std::ofstream output("solution-" + std::to_string(cycle) + ".vtk");
        data_out.write_vtk(output);
    }
}

template <int dim>
void AdvectionProblem<dim>::run()
{
    for (unsigned int cycle = 0; cycle < 6; ++cycle) {
        std::cout << "Cycle " << cycle << " " << std::endl;
    }
}

```

```

    if (cycle == 0) {
        GridGenerator::hyper_cube(triangulation, -1, 1);
        triangulation.refine_global(4);
    } else {
        refine_grid();
    }

    std::cout << "    Number of active cells:      "
              << triangulation.n_active_cells() << std::endl;

    setup_system();

    std::cout << "    Number of degrees of freedom: " << dof_handler.n_dofs()
              << std::endl;

    assemble_system();
    solve();
    output_results(cycle);
}

template <int dim>
GradientEstimation::EstimateScratchData<dim>::EstimateScratchData(
    const FiniteElement<dim> &fe, const Vector<double> &solution,
    Vector<float> &error_per_cell)
: fe_midpoint_value(fe, QMidpoint<dim>()),
  update_values | update_quadrature_points(),
  solution(solution),
  error_per_cell(error_per_cell)
{}

template <int dim>
GradientEstimation::EstimateScratchData<dim>::EstimateScratchData(
    const EstimateScratchData &scratch_data)
: fe_midpoint_value(scratch_data.fe_midpoint_value.get_fe(),
                    scratch_data.fe_midpoint_value.get_quadrature(),
                    update_values | update_quadrature_points(),
                    solution(scratch_data.solution),
                    error_per_cell(scratch_data.error_per_cell))
{}

template <int dim>
void GradientEstimation::estimate(const DoFHandler<dim> &dof_handler,
                                const Vector<double> &solution,
                                Vector<float> &error_per_cell)
{
    Assert(error_per_cell.size() ==
           dof_handler.get_triangulation().n_active_cells(),
           ExcInvalidVectorLength(
               error_per_cell.size(),
               dof_handler.get_triangulation().n_active_cells()));

    WorkStream::run(dof_handler.begin_active(), dof_handler.end(),
                    &GradientEstimation::template estimate_cell<dim>,
                    std::function<void(const EstimateCopyData &)>(),
                    EstimateScratchData<dim>(dof_handler.get_fe(), solution,
                                             error_per_cell),
                    EstimateCopyData());
}

template <int dim>
void GradientEstimation::estimate_cell(
    const typename DoFHandler<dim>::active_cell_iterator &cell,
    EstimateScratchData<dim> &scratch_data, const EstimateCopyData &)
{
    Tensor<2, dim> Y;

    std::vector<typename DoFHandler<dim>::active_cell_iterator>
        active_neighbors;
    active_neighbors.reserve(GeometryInfo<dim>::faces_per_cell *
                           GeometryInfo<dim>::max_children_per_face);

    scratch_data.fe_midpoint_value.reinit(cell);

    Tensor<1, dim> projected_gradient;

    active_neighbors.clear();
    for (unsigned int face_no = 0; face_no < GeometryInfo<dim>::faces_per_cell;
         ++face_no)

```

```

    if (!cell->at_boundary(face_no)) {
        const typename DoFHandler<dim>::face_iterator face =
            cell->face(face_no);
        const typename DoFHandler<dim>::cell_iterator neighbor =
            cell->neighbor(face_no);

        if (neighbor->active())
            active_neighbors.push_back(neighbor);
        else {
            if (dim == 1) {
                typename DoFHandler<dim>::cell_iterator neighbor_child =
                    neighbor;
                while (neighbor_child->has_children())
                    neighbor_child =
                        neighbor_child->child(face_no == 0 ? 1 : 0);

                Assert(
                    neighbor_child->neighbor(face_no == 0 ? 1 : 0) == cell,
                    ExcInternalError());

                active_neighbors.push_back(neighbor_child);
            } else
                for (unsigned int subface_no = 0;
                     subface_no < face->n_children(); ++subface_no)
                    active_neighbors.push_back(
                        cell->neighbor_child_on_subface(face_no,
                                                         subface_no));
        }
    }

    const Point<dim> this_center =
        scratch_data.fe_midpoint_value.quadrature_point(0);

    std::vector<double> this_midpoint_value(1);
    scratch_data.fe_midpoint_value.get_function_values(scratch_data.solution,
                                                       this_midpoint_value);

    std::vector<double> neighbor_midpoint_value(1);
    typename std::vector<typename DoFHandler<dim>::active_cell_iterator>::
        const_iterator neighbor_ptr = active_neighbors.begin();
    for (; neighbor_ptr != active_neighbors.end(); ++neighbor_ptr) {
        const typename DoFHandler<dim>::active_cell_iterator neighbor =
            *neighbor_ptr;

        scratch_data.fe_midpoint_value.reinit(neighbor);
        const Point<dim> neighbor_center =
            scratch_data.fe_midpoint_value.quadrature_point(0);

        scratch_data.fe_midpoint_value.get_function_values(
            scratch_data.solution, neighbor_midpoint_value);

        Tensor<1, dim> y = neighbor_center - this_center;
        const double distance = y.norm();
        y /= distance;

        for (unsigned int i = 0; i < dim; ++i)
            for (unsigned int j = 0; j < dim; ++j) Y[i][j] += y[i] * y[j];

        projected_gradient +=
            (neighbor_midpoint_value[0] - this_midpoint_value[0]) / distance *
            y;
    }

    AssertThrow(determinant(Y) != 0, ExcInsufficientDirections());

    const Tensor<2, dim> Y_inverse = invert(Y);

    Tensor<1, dim> gradient = Y_inverse * projected_gradient;

    scratch_data.error_per_cell(cell->active_cell_index()) =
        (std::pow(cell->diameter(), 1 + 1.0 * dim / 2) *
         std::sqrt(gradient.norm_square()));
}
// namespace Step9

int main()
{
    try {
        dealii::MultithreadInfo::set_thread_limit();

        Step9::AdvectionProblem<2> advection_problem_2d;
        advection_problem_2d.run();
    } catch (std::exception &exc) {
        std::cerr << std::endl

```



```
        << std::endl
        << "-----"
        << std::endl;
std::cerr << "Exception on processing: " << std::endl
<< exc.what() << std::endl
<< "Aborting!" << std::endl
<< "-----"
<< std::endl;
    return 1;
} catch (...) {
    std::cerr << std::endl
    << std::endl
    << "-----"
    << std::endl;
    std::cerr << "Unknown exception!" << std::endl
    << "Aborting!" << std::endl
    << "-----"
    << std::endl;
    return 1;
}
return 0;
}
```


Chapter 10

The ginkgo-overhead program

The ginkgo overhead measurement example.

Introduction

About the example

The commented program

```
#include <ginkgo/ginkgo.hpp>

#include <chrono>
#include <cmath>
#include <iostream>

[[noreturn]] void print_usage_and_exit(const char *name)
{
    std::cerr << "Usage: " << name << " [NUM_ITERS]" << std::endl;
    std::exit(-1);
}

int main(int argc, char *argv[])
{
    using vec = gko::matrix::Dense<>;
    using mtx = gko::matrix::Dense<>;
    using cg = gko::solver::Cg<>;

    long unsigned num_iters = 1000000;
    if (argc > 2) {
        print_usage_and_exit(argv[0]);
    }
    if (argc == 2) {
        num_iters = std::atol(argv[1]);
        if (num_iters == 0) {
            print_usage_and_exit(argv[0]);
        }
    }

    std::cout << gko::version_info::get() << std::endl;

    auto exec = gko::ReferenceExecutor::create();

    auto cg_factory =
        cg::build()
            .with_criteria(
                gko::stop::Iteration::build().with_max_iters(num_iters).on(
                    exec))
            .on(exec);
```

```

auto A = gko::initialize<mtx>({1.0}, exec);
auto b = gko::initialize<vec>({std::nan("")}, exec);
auto x = gko::initialize<vec>({0.0}, exec);

auto tic = std::chrono::steady_clock::now();

auto solver = cg_factory->generate(gko::give(A));
solver->apply(lend(x), lend(b));
exec->synchronize();

auto tac = std::chrono::steady_clock::now();

auto time = std::chrono::duration_cast<std::chrono::nanoseconds>(tac - tic);
std::cout << "Running " << num_iters
  << " iterations of the CG solver took a total of "
  << 1.0 * time.count() / std::nano::den << " seconds." << std::endl
  << "\tAverage library overhead: "
  << 1.0 * time.count() / num_iters << " [nanoseconds / iteration]"
  << std::endl;
}

```

Results

This is the expected output:

```

Running 1000000 iterations of the CG solver took a total of 1.50535 seconds.
Average library overhead:      1505.35 [nanoseconds / iteration]

```

Comments about programming and debugging

The plain program

```

/*****<GINKGO LICENSE>*****/
Copyright (c) 2017-2019, the Ginkgo authors
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its
contributors may be used to endorse or promote products derived from
this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*****/

#include <ginkgo/ginkgo.hpp>

#include <chrono>
#include <cmath>
#include <iostream>

```

```

[[noreturn]] void print_usage_and_exit(const char *name)
{
    std::cerr << "Usage: " << name << " [NUM_ITERS]" << std::endl;
    std::exit(-1);
}

int main(int argc, char *argv[])
{
    using vec = gko::matrix::Dense<>;
    using mtx = gko::matrix::Dense<>;
    using cg = gko::solver::Cg<>;

    long unsigned num_iters = 1000000;
    if (argc > 2) {
        print_usage_and_exit(argv[0]);
    }
    if (argc == 2) {
        num_iters = std::atol(argv[1]);
        if (num_iters == 0) {
            print_usage_and_exit(argv[0]);
        }
    }

    std::cout << gko::version_info::get() << std::endl;

    auto exec = gko::ReferenceExecutor::create();

    auto cg_factory =
        cg::build()
            .with_criteria(
                gko::stop::Iteration::build().with_max_iters(num_iters).on(
                    exec))
            .on(exec);
    auto A = gko::initialize<mtx>({1.0}, exec);
    auto b = gko::initialize<vec>({std::nan("")}, exec);
    auto x = gko::initialize<vec>({0.0}, exec);

    auto tic = std::chrono::steady_clock::now();

    auto solver = cg_factory->generate(gko::give(A));
    solver->apply(lend(x), lend(b));
    exec->synchronize();

    auto tac = std::chrono::steady_clock::now();

    auto time = std::chrono::duration_cast<std::chrono::nanoseconds>(tac - tic);
    std::cout << "Running " << num_iters
        << " iterations of the CG solver took a total of "
        << 1.0 * time.count() / std::nano::den << " seconds." << std::endl
        << "\tAverage library overhead: "
        << 1.0 * time.count() / num_iters << " [nanoseconds / iteration]"
        << std::endl;
}

```


Chapter 11

The ginkgo-ranges program

The ranges and accessor example.

Introduction

About the example

The commented program

```
#include <ginkgo/ginkgo.hpp>
#include <iomanip>
#include <iostream>
```

LU factorization implementation using Ginkgo ranges For simplicity, we only consider square matrices, and no pivoting.

```
template <typename Accessor>
void factorize(const gko::range<Accessor> &A)
```

note: const means that the range (i.e. the data handler) is constant, not that the underlying data is constant!

```
{
    using gko::span;
    assert(A.length(0) == A.length(1));
    for (gko::size_type i = 0; i < A.length(0) - 1; ++i) {
        const auto trail = span{i + 1, A.length(0)};
```

note: neither of the lines below need additional memory to store intermediate arrays, all computation is done at the point of assignment

```
A(trail, i) = A(trail, i) / A(i, i);
```

caveat: operator * is element-wise multiplication, mmul is matrix multiplication

```

        A(trail, trail) = A(trail, trail) - mmul(A(trail, i), A(i, trail));
    }
}

```

a utility function for printing the factorization on screen

```

template <typename Accessor>
void print_lu(const gko::range<Accessor> &A)
{
    std::cout << std::setprecision(2) << std::fixed;
    std::cout << "L = [";
    for (int i = 0; i < A.length(0); ++i) {
        std::cout << "\n ";
        for (int j = 0; j < A.length(1); ++j) {
            std::cout << (i > j ? A(i, j) : (i == j) * 1.) << " ";
        }
    }
    std::cout << "\n]\n\nU = [";
    for (int i = 0; i < A.length(0); ++i) {
        std::cout << "\n ";
        for (int j = 0; j < A.length(1); ++j) {
            std::cout << (i <= j ? A(i, j) : 0.) << " ";
        }
    }
    std::cout << "\n]" << std::endl;
}

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

```

Print version information

```
std::cout << gko::version_info::get() << std::endl;
```

Create some test data, add some padding just to demonstrate how to use it with ranges. clang-format off

```

double data[] = {
    2., 4., 5., -1.0,
    4., 11., 12., -1.0,
    6., 24., 24., -1.0
};

```

clang-format on

Create a 3-by-3 range, with a 2D row-major accessor using data as the underlying storage. Set the stride (a.k.a. "LDA") to 4.

```

auto A = gko::range<gko::accessor::row_major<double, 2>>(data
    , 3u, 3u, 4u);

```

use the LU factorization routine defined above to factorize the matrix

```
factorize(A);
```

print the factorization on screen

```

    print_lu(A);
}

```


Results

This is the expected output:

```
L = [
  1.00 0.00 0.00
  2.00 1.00 0.00
  3.00 4.00 1.00
]

U = [
  2.00 4.00 5.00
  0.00 3.00 2.00
  0.00 0.00 1.00
]
```

Comments about programming and debugging

The plain program

```
/******<GINKGO LICENSE>*****
Copyright (c) 2017-2019, the Ginkgo authors
All rights reserved.
```

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

```
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*****<GINKGO LICENSE>*****
```

```
#include <ginkgo/ginkgo.hpp>
#include <iomanip>
#include <iostream>
```

```
template <typename Accessor>
void factorize(const gko::range<Accessor> &A)
{
    using gko::span;
    assert(A.length(0) == A.length(1));
    for (gko::size_type i = 0; i < A.length(0) - 1; ++i) {
        const auto trail = span(i + 1, A.length(0));
        A(trail, i) = A(trail, i) / A(i, i);
        A(trail, trail) = A(trail, trail) - mmul(A(trail, i), A(i, trail));
    }
}
```

```
template <typename Accessor>
void print_lu(const gko::range<Accessor> &A)
{
    std::cout << std::setprecision(2) << std::fixed;
    std::cout << "L = [";
    for (int i = 0; i < A.length(0); ++i) {
        std::cout << "\n ";
    }
}
```

```

        for (int j = 0; j < A.length(1); ++j) {
            std::cout << (i > j ? A(i, j) : (i == j) * 1.) << " ";
        }
    }
    std::cout << "\n\nU = [";
    for (int i = 0; i < A.length(0); ++i) {
        std::cout << "\n ";
        for (int j = 0; j < A.length(1); ++j) {
            std::cout << (i <= j ? A(i, j) : 0.) << " ";
        }
    }
    std::cout << "\n]" << std::endl;
}

int main(int argc, char *argv[])
{
    std::cout << gko::version_info::get() << std::endl;

    double data[] = {
        2., 4., 5., -1.0,
        4., 11., 12., -1.0,
        6., 24., 24., -1.0
    };

    auto A = gko::range<gko::accessor::row_major<double, 2>>(
        data, 3u, 3u, 4u);

    factorize(A);

    print_lu(A);
}

```

Chapter 12

The inverse-iteration program

The inverse iteration example.

This example depends on simple-solver, .

Introduction

This example shows how components available in Ginkgo can be used to implement higher-level numerical methods. The method used here will be the shifted inverse iteration method for eigenvalue computation which find the eigenvalue and eigenvector of A closest to z , for some scalar z . The method requires repeatedly solving the shifted linear system $(A - zI)x = b$, as well as performing matrix-vector products with the matrix A . Here is the complete pseudocode of the method:

```
x_0 = initial guess
for i = 0 .. max_iterations:
    solve (A - zI) y_i = x_i for y_i+1
    x_(i+1) = y_i / || y_i ||      # compute next eigenvector approximation
    g_(i+1) = x_(i+1)^* A x_(i+1)  # approximate eigenvalue (Rayleigh quotient)
    if ||A x_(i+1) - g_(i+1)x_(i+1)|| < tol * g_(i+1): # check convergence
        break
```

About the example

The commented program

```
#include <ginkgo/ginkgo.hpp>

#include <cmath>
#include <complex>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <string>

int main(int argc, char *argv[])
{
```

Some shortcuts

```
using precision = std::complex<double>;
using real_precision = double;
using vec = gko::matrix::Dense<precision>;
using mtx = gko::matrix::Csr<precision>;
using solver_type = gko::solver::Bicgstab<precision>;

using std::abs;
using std::sqrt;
```

Print version information

```
std::cout << gko::version_info::get() << std::endl;

std::cout << std::scientific << std::setprecision(8) << std::showpos;
```

Figure out where to run the code

```
std::shared_ptr<gko::Executor> exec;
if (argc == 1 || std::string(argv[1]) == "reference") {
    exec = gko::ReferenceExecutor::create();
} else if (argc == 2 && std::string(argv[1]) == "omp") {
    exec = gko::OmpExecutor::create();
} else if (argc == 2 && std::string(argv[1]) == "cuda" &&
           gko::CudaExecutor::get_num_devices() > 0) {
    exec = gko::CudaExecutor::create(0,
                                     gko::OmpExecutor::create());
} else {
    std::cerr << "Usage: " << argv[0] << " [executor]" << std::endl;
    std::exit(-1);
}

auto this_exec = exec->get_master();
```

linear system solver parameters

```
auto system_max_iterations = 100u;
auto system_residual_goal = real_precision{1e-16};
```

eigensolver parameters

```
auto max_iterations = 20u;
auto residual_goal = real_precision{1e-8};
auto z = precision{20.0, 2.0};
```

Read data

```
auto A = share(gko::read<mtx>(std::ifstream("data/A.mtx"), exec));
```

Generate shifted matrix A - zI

- we avoid duplicating memory by not storing both A and A - zI, but compute A - zI on the fly by using Ginkgo's utilities for creating linear combinations of operators

```
auto one = share(gko::initialize<vec>({precision{1.0}}, exec));
auto neg_one = share(gko::initialize<vec>({-precision{1.0}}, exec));
auto neg_z = gko::initialize<vec>({-z}, exec);

auto system_matrix = share(gko::Combination<precision>::create(
    one, A, gko::initialize<vec>({-z}, exec),
    gko::matrix::Identity<precision>::create(exec, A->get_size()[0])
));
```

Generate solver operator $(A - zI)^{-1}$

```
auto solver =
    solver_type::build()
        .with_criteria(gko::stop::Iteration::build()
            .with_max_iters(system_max_iterations)
            .on(exec),
            gko::stop::ResidualNormReduction<precision>::build
                .with_reduction_factor(system_residual_goal)
                .on(exec))
        .on(exec)
    ->generate(system_matrix);
```

inverse iterations

start with guess $[1, 1, \dots, 1]$

```
auto x = [&] {
    auto work = vec::create(this_exec, gko::dim<2>{A->get_size()[0], 1});
    const auto n = work->get_size()[0];
    for (int i = 0; i < n; ++i) {
        work->get_values()[i] = precision{1.0} / sqrt(n);
    }
    return clone(exec, work);
}();
auto y = clone(x);
auto tmp = clone(x);
auto norm = clone(one);
auto inv_norm = clone(this_exec, one);
auto g = clone(one);

for (auto i = 0u; i < max_iterations; ++i) {
    std::cout << "i ";
```

$(A - zI)y = x$

```
solver->apply(lend(x), lend(y));
system_matrix->apply(lend(one), lend(y), lend(neg_one), lend(x));
x->compute_norm2(lend(norm));
std::cout << "\"system_residual\": "
    << clone(this_exec, norm)->get_values()[0] << ", ";
x->copy_from(lend(y));
```

$x = y / \|y\|$

```
x->compute_norm2(lend(norm));
inv_norm->get_values()[0] =
    precision{1.0} / clone(this_exec, norm)->get_values()[0];
x->scale(lend(clone(exec, inv_norm)));
```

$g = x^A x$

```
A->apply(lend(x), lend(tmp));
x->compute_dot(lend(tmp), lend(g));
auto g_val = clone(this_exec, g)->get_values()[0];
std::cout << "\"eigenvalue\": " << g_val << ", ";
```

$\|Ax - gx\| < \text{tol} * g$

```
auto v = gko::initialize<vec>({-g_val}, exec);
tmp->add_scaled(lend(v), lend(x));
tmp->compute_norm2(lend(norm));
auto res_val = clone(exec->get_master(), norm)->get_values()[0];
std::cout << "\"residual\": " << res_val / g_val << " }, " << std::endl;
if (abs(res_val) < residual_goal * abs(g_val)) {
    break;
}
}
```

Results

This is the expected output:

```
{ "system_residual": (+1.59066966e-14,+0.00000000e+00), "eigenvalue": (+2.03741410e+01,-5.42101086e-18), "
  residual": (+2.92231055e-01,+7.77548230e-20) },
{ "system_residual": (+6.38877157e-15,+0.00000000e+00), "eigenvalue": (+1.94878474e+01,-4.34534678e-16), "
  residual": (+7.94370276e-02,+1.77126506e-18) },
{ "system_residual": (+6.79215294e-15,+0.00000000e+00), "eigenvalue": (+1.93282121e+01,-3.68988781e-16), "
  residual": (+4.11149623e-02,+7.84912734e-19) },
{ "system_residual": (+3.54015578e-15,+0.00000000e+00), "eigenvalue": (+1.92638912e+01,+2.03949917e-16), "
  residual": (+2.34717040e-02,-2.48498708e-19) },
{ "system_residual": (+2.12400044e-15,+0.00000000e+00), "eigenvalue": (+1.92409166e+01,-7.59991100e-16), "
  residual": (+1.34709547e-02,+5.32085134e-19) },
{ "system_residual": (+3.29202859e-15,+0.00000000e+00), "eigenvalue": (+1.92331106e+01,+2.90110055e-15), "
  residual": (+7.72060707e-03,-1.16456760e-18) },
{ "system_residual": (+3.99088304e-15,+0.00000000e+00), "eigenvalue": (+1.92305014e+01,-3.21058733e-16), "
  residual": (+4.42106625e-03,+7.38109682e-20) },
{ "system_residual": (+2.02648035e-15,+0.00000000e+00), "eigenvalue": (+1.92296339e+01,+5.11222288e-16), "
  residual": (+2.53081312e-03,-6.72819919e-20) },
{ "system_residual": (+1.83840397e-15,+0.00000000e+00), "eigenvalue": (+1.92293461e+01,+3.51208924e-16), "
  residual": (+1.44862114e-03,-2.64579289e-20) },
{ "system_residual": (+1.60253167e-15,+0.00000000e+00), "eigenvalue": (+1.92292506e+01,-2.02284978e-15), "
  residual": (+8.29183451e-04,+8.72271932e-20) },
{ "system_residual": (+1.96758490e-15,+0.00000000e+00), "eigenvalue": (+1.92292190e+01,+8.90545453e-16), "
  residual": (+4.74636702e-04,-2.19814209e-20) },
{ "system_residual": (+1.53327380e-14,+0.00000000e+00), "eigenvalue": (+1.92292085e+01,-8.25871947e-17), "
  residual": (+2.71701077e-04,+1.16692425e-21) },
{ "system_residual": (+3.42985865e-15,+0.00000000e+00), "eigenvalue": (+1.92292051e+01,+1.63122796e-16), "
  residual": (+1.55539937e-04,-1.31945701e-21) },
{ "system_residual": (+3.30861071e-11,+0.00000000e+00), "eigenvalue": (+1.92292039e+01,-5.49102025e-16), "
  residual": (+8.90457139e-05,+2.54275643e-21) },
{ "system_residual": (+7.11155374e-14,+0.00000000e+00), "eigenvalue": (+1.92292035e+01,+1.16689376e-15), "
  residual": (+5.09805252e-05,-3.09367244e-21) },
{ "system_residual": (+2.68204494e-15,+0.00000000e+00), "eigenvalue": (+1.92292034e+01,-4.07084034e-17), "
  residual": (+2.91887365e-05,+6.17928281e-23) },
{ "system_residual": (+5.78377594e-13,+0.00000000e+00), "eigenvalue": (+1.92292034e+01,-3.38561848e-17), "
  residual": (+1.67126561e-05,+2.94253882e-23) },
{ "system_residual": (+6.26422040e-12,+0.00000000e+00), "eigenvalue": (+1.92292034e+01,-3.14429218e-18), "
  residual": (+9.56961199e-06,+1.56478953e-24) },
{ "system_residual": (+1.41104829e-12,+0.00000000e+00), "eigenvalue": (+1.92292033e+01,-6.54656730e-16), "
  residual": (+5.47975753e-06,+1.86557918e-22) },
{ "system_residual": (+1.97926842e-10,+0.00000000e+00), "eigenvalue": (+1.92292033e+01,+1.58008702e-16), "
  residual": (+3.13794996e-06,-2.57849164e-23) },
```

Comments about programming and debugging

The plain program

```
/******<GINKGO LICENSE>*****
Copyright (c) 2017-2019, the Ginkgo authors
All rights reserved.
```

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT

```

(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*****<GINKGO LICENSE>*****/

#include <ginkgo/ginkgo.hpp>

#include <cmath>
#include <complex>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <string>

int main(int argc, char *argv[])
{
    using precision = std::complex<double>;
    using real_precision = double;
    using vec = gko::matrix::Dense<precision>;
    using mtx = gko::matrix::Csr<precision>;
    using solver_type = gko::solver::Bicgstab<precision>;

    using std::abs;
    using std::sqrt;

    std::cout << gko::version_info::get() << std::endl;

    std::cout << std::scientific << std::setprecision(8) << std::showpos;

    std::shared_ptr<gko::Executor> exec;
    if (argc == 1 || std::string(argv[1]) == "reference") {
        exec = gko::ReferenceExecutor::create();
    } else if (argc == 2 && std::string(argv[1]) == "omp") {
        exec = gko::OmpExecutor::create();
    } else if (argc == 2 && std::string(argv[1]) == "cuda" &&
               gko::CudaExecutor::get_num_devices() > 0) {
        exec = gko::CudaExecutor::create(0,
        gko::OmpExecutor::create());
    } else {
        std::cerr << "Usage: " << argv[0] << " [executor]" << std::endl;
        std::exit(-1);
    }

    auto this_exec = exec->get_master();

    auto system_max_iterations = 100u;
    auto system_residual_goal = real_precision{1e-16};

    auto max_iterations = 20u;
    auto residual_goal = real_precision{1e-8};
    auto z = precision{20.0, 2.0};

    auto A = share(gko::read<mtx>(std::ifstream("data/A.mtx"), exec));

    auto one = share(gko::initialize<vec>({precision{1.0}}, exec));
    auto neg_one = share(gko::initialize<vec>({-precision{1.0}}, exec));
    auto neg_z = gko::initialize<vec>({-z}, exec);

    auto system_matrix = share(gko::Combination<precision>::create(
        one, A, gko::initialize<vec>({-z}, exec),
        gko::matrix::Identity<precision>::create(exec, A->get_size(
        ) [0])));

    auto solver =
        solver_type::build()
            .with_criteria(gko::stop::Iteration::build()
                .with_max_iters(system_max_iterations)
                .on(exec),
                gko::stop::ResidualNormReduction<precision>::build
                ()
                    .with_reduction_factor(system_residual_goal)
                    .on(exec))
            .on(exec)
            ->generate(system_matrix);

    auto x = [&] {
        auto work = vec::create(this_exec, gko::dim<2>{A->get_size() [0], 1});
        const auto n = work->get_size() [0];
        for (int i = 0; i < n; ++i) {
            work->get_values() [i] = precision{1.0} / sqrt(n);
        }
        return clone(exec, work);
    }();
    auto y = clone(x);
    auto tmp = clone(x);

```

```

auto norm = clone(one);
auto inv_norm = clone(this_exec, one);
auto g = clone(one);

for (auto i = 0u; i < max_iterations; ++i) {
    std::cout << "{ ";
    solver->apply(lend(x), lend(y));
    system_matrix->apply(lend(one), lend(y), lend(neg_one), lend(x));
    x->compute_norm2(lend(norm));
    std::cout << "\"system_residual\": "
                << clone(this_exec, norm)->get_values()[0] << ", ";
    x->copy_from(lend(y));
    x->compute_norm2(lend(norm));
    inv_norm->get_values()[0] =
        precision{1.0} / clone(this_exec, norm)->get_values()[0];
    x->scale(lend(clone(exec, inv_norm)));
    A->apply(lend(x), lend(tmp));
    x->compute_dot(lend(tmp), lend(g));
    auto g_val = clone(this_exec, g)->get_values()[0];
    std::cout << "\"eigenvalue\": " << g_val << ", ";
    auto v = gko::initialize<vec>({-g_val}, exec);
    tmp->add_scaled(lend(v), lend(x));
    tmp->compute_norm2(lend(norm));
    auto res_val = clone(exec->get_master(), norm)->get_values()[0];
    std::cout << "\"residual\": " << res_val / g_val << " }," << std::endl;
    if (abs(res_val) < residual_goal * abs(g_val)) {
        break;
    }
}
}

```


Chapter 13

The minimal-cuda-solver program

The minimal CUDA solver example.

This example depends on simple-solver.

Introduction

This is a minimal example that solves a system with Ginkgo. The matrix, right hand side and initial guess are read from standard input, and the result is written to standard output. The system matrix is stored in CSR format, and the system solved using the CG method, preconditioned with the block-Jacobi preconditioner. All computations are done on the GPU.

The easiest way to use the example data from the `data/` folder is to concatenate the matrix, the right hand side and the initial solution (in that exact order), and pipe the result to the `minimal_solver_cuda` executable:

```
cat data/A.mtx data/b.mtx data/x0.mtx | ./minimal_solver_cuda
```

About the example

The commented program

```
#include <ginkgo/ginkgo.hpp>
#include <iostream>

int main()
{
```

Instantiate a CUDA executor

```
auto gpu = gko::CudaExecutor::create(0,
    gko::OmpExecutor::create());
```

Read data

```

auto A = gko::read<gko::matrix::Csr<>>(std::cin, gpu);
auto b = gko::read<gko::matrix::Dense<>>(std::cin, gpu);
auto x = gko::read<gko::matrix::Dense<>>(std::cin, gpu);

```

Create the solver

```

auto solver =
    gko::solver::Cg<>::build()
        .with_preconditioner(gko::preconditioner::Jacobi<>::build().on(
            gpu))
        .with_criteria(
            gko::stop::Iteration::build().with_max_iters(20u).on(gpu),
            gko::stop::ResidualNormReduction<>::build()
                .with_reduction_factor(1e-15)
                .on(gpu))
        .on(gpu);

```

Solve system

```

solver->generate(give(A))>apply(lend(b), lend(x));

```

Write result

```

    write(std::cout, lend(x));
}

```

Results

The following is the expected result when using the data contained in the folder `data` as input:

```

%%MatrixMarket matrix array real general
19 1
0.252218
0.108645
0.0662811
0.0630433
0.0384088
0.0396536
0.0402648
0.0338935
0.0193098
0.0234653
0.0211499
0.0196413
0.0199151
0.0181674
0.0162722
0.0150714
0.0107016
0.0121141
0.0123025

```

Comments about programming and debugging

The plain program

```

/*****<GINKGO LICENSE>*****/
Copyright (c) 2017-2019, the Ginkgo authors
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its
contributors may be used to endorse or promote products derived from
this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*****/

#include <ginkgo/ginkgo.hpp>
#include <iostream>

int main()
{
    auto gpu = gko::CudaExecutor::create(0,
        gko::OmpExecutor::create());
    auto A = gko::read<gko::matrix::Csr<>>(std::cin, gpu);
    auto b = gko::read<gko::matrix::Dense<>>(std::cin, gpu);
    auto x = gko::read<gko::matrix::Dense<>>(std::cin, gpu);
    auto solver =
        gko::solver::Cg<>::build()
        .with_preconditioner(gko::preconditioner::Jacobi<>::build()
            .on(gpu))
        .with_criteria(
            gko::stop::Iteration::build().with_max_iters(20u).on(gpu),
            gko::stop::ResidualNormReduction<>::build()
                .with_reduction_factor(1e-15)
                .on(gpu))
        .on(gpu);
    solver->generate(give(A))->apply(lend(b), lend(x));
    write(std::cout, lend(x));
}

```


Chapter 14

The nine-pt-stencil-solver program

The 9-point stencil example.

This example depends on simple-solver, poisson-solver, three-pt-stencil-solve.

Introduction

This example solves a 2D Poisson equation:

$$\Omega = (0,1)^2 \setminus [0,1]^2 \setminus \{ \text{(with boundary)} \} \setminus \partial \Omega : -\Delta u = f \text{ in } \Omega, u = u_D \text{ on } \partial \Omega$$

using a finite difference method on an equidistant grid with K discretization points (K can be controlled with a command line parameter). The discretization may be done by any order Taylor polynomial. For an equidistant grid with K "inner" discretization points $((x_1, y_1), \dots, (x_K, y_1), (x_1, y_2), \dots, (x_K, y_2), \dots, (x_K, y_K))$ step size $(h = 1 / (K + 1))$ and a stencil $(\{R\}^3)$, the formula produces a system of linear equations

$$(\{a, b = -1\}^1 \text{ stencil}(a, b) * u_{\{(i+a, j+b)\}} = -f_k h^2), \text{ on any inner node with a neighborhood of inner nodes}$$

On any node, where neighbor is on the border, the neighbor is replaced with a $(-\text{stencil}(a, b) * u_{\{(i+a, j+b)\}})$ and added to the right hand side vector. For example a node with a neighborhood of only edge nodes may look like this

$$[\{a, b = -1\}^1(1, 0) \text{ stencil}(a, b) * u_{\{(i+a, j+b)\}} = -f_k h^2 - \{a = -1\}^1 \text{ stencil}(a, 1) * u_{\{(i+a, j+1)\}}]$$

which is then solved using Ginkgo's implementation of the CG method preconditioned with block-Jacobi. It is also possible to specify on which executor Ginkgo will solve the system via the command line. The function f is set to $f(x, y) = 6x + 6y$ (making the solution $(u(x, y) = x^3$

- $y^3)$), but that can be changed in the `main` function. Also the stencil values for the core, the faces, the edge and the corners can be changed when passing additional parameters.

The intention of this is to show how generation of stencil values and the right hand side vector changes when increasing the dimension.

About the example

The commented program

```
/ *****<DESCRIPTION>*****
This example solves a 2D Poisson equation:

\Omega = (0,1)^2
\Omega_b = [0,1]^2 (with boundary)
\partial\Omega = \Omega_b \backslash \Omega
u : \Omega_b \rightarrow \mathbb{R}
u'' = f in \Omega
u = u_D on \partial\Omega

using a finite difference method on an equidistant grid with 'K' discretization
points ('K' can be controlled with a command line parameter). The discretization
may be done by any order Taylor polynomial.
For an equidistant grid with K "inner" discretization points (x1,y1), ...,
(xk,y1), (x1,y2), ..., (xk,yk) step size h = 1 / (K + 1) and a stencil \in
\mathbb{R}^{3 \times 3}, the formula produces a system of linear equations

\sum_{a,b=-1}^1 stencil(a,b) * u_{(i+a,j+b)} = -f_k h^2, on any inner node with
a neighborhood of inner nodes

On any node, where neighbor is on the border, the neighbor is replaced with a
'-stencil(a,b) * u_{(i+a,j+b)}' and added to the right hand side vector. For
example a node with a neighborhood of only edge nodes may look like this

\sum_{a,b=-1}^1 stencil(a,b) * u_{(i+a,j+b)} = -f_k h^2 - \sum_{a=-1}^1
stencil(a,1) * u_{(i+a,j+1)}

which is then solved using Ginkgo's implementation of the CG method
preconditioned with block-Jacobi. It is also possible to specify on which
executor Ginkgo will solve the system via the command line.
The function 'f' is set to 'f(x,y) = 6x + 6y' (making the solution 'u(x,y) = x^3
+ y^3'), but that can be changed in the 'main' function. Also the stencil values
for the core, the faces, the edge and the corners can be changed when passing
additional parameters.

The intention of this is to show how generation of stencil values and the right
hand side vector changes when increasing the dimension.
*****<DESCRIPTION>***** /

#include <array>
#include <chrono>
#include <ginkgo/ginkgo.hpp>
#include <iostream>
#include <map>
#include <string>
#include <vector>
```

Stencil values. Ordering can be seen in the main function Can also be changed by passing additional parameter when executing

```
constexpr double default_alpha = 10.0 / 3.0;
constexpr double default_beta = -2.0 / 3.0;
constexpr double default_gamma = -1.0 / 6.0;

/ * Possible alternative default values are for example
* default_alpha = 8.0;
* default_beta = -1.0;
* default_gamma = -1.0;
* /
```

Creates a stencil matrix in CSR format for the given number of discretization points.

```
void generate_stencil_matrix(int dp, int *row_ptrs, int *col_idxes,
                           double *values, double *coefs)
{
    int pos = 0;
    const size_t dp_2 = dp * dp;
    row_ptrs[0] = pos;
    for (int k = 0; k < dp; ++k) {
        for (int i = 0; i < dp; ++i) {
            const size_t index = i + k * dp;
```

```

        for (int j = -1; j <= 1; ++j) {
            for (int l = -1; l <= 1; ++l) {
                const int64_t offset = l + l + 3 * (j + 1);
                if ((k + j) >= 0 && (k + j) < dp && (i + 1) >= 0 &&
                    (i + 1) < dp) {
                    values[pos] = coefs[offset];
                    col_idxes[pos] = index + l + dp * j;
                    ++pos;
                }
            }
            row_ptrs[index + 1] = pos;
        }
    }
}

```

Generates the RHS vector given f and the boundary conditions.

```

template <typename Closure, typename ClosureT>
void generate_rhs(int dp, Closure f, ClosureT u, double *rhs, double *coefs)
{
    const size_t dp_2 = dp * dp;
    const auto h = 1.0 / (dp + 1.0);
    for (int i = 0; i < dp; ++i) {
        const auto yi = (i + 1) * h;
        for (int j = 0; j < dp; ++j) {
            const auto xi = (j + 1) * h;
            const auto index = i * dp + j;
            rhs[index] = -f(xi, yi) * h * h;
        }
    }
}

```

Iterating over the edges to add boundary values and adding the overlapping 3x1 to the rhs

```

for (size_t i = 0; i < dp; ++i) {
    const auto xi = (i + 1) * h;
    const auto index_top = i;
    const auto index_bot = i + dp * (dp - 1);

    rhs[index_top] -= u(xi - h, 0.0) * coefs[0];
    rhs[index_top] -= u(xi, 0.0) * coefs[1];
    rhs[index_top] -= u(xi + h, 0.0) * coefs[2];

    rhs[index_bot] -= u(xi - h, 1.0) * coefs[6];
    rhs[index_bot] -= u(xi, 1.0) * coefs[7];
    rhs[index_bot] -= u(xi + h, 1.0) * coefs[8];
}
for (size_t i = 0; i < dp; ++i) {
    const auto yi = (i + 1) * h;
    const auto index_left = i * dp;
    const auto index_right = i * dp + (dp - 1);

    rhs[index_left] -= u(0.0, yi - h) * coefs[0];
    rhs[index_left] -= u(0.0, yi) * coefs[3];
    rhs[index_left] -= u(0.0, yi + h) * coefs[6];

    rhs[index_right] -= u(1.0, yi - h) * coefs[2];
    rhs[index_right] -= u(1.0, yi) * coefs[5];
    rhs[index_right] -= u(1.0, yi + h) * coefs[8];
}

```

remove the double corner values

```

rhs[0] += u(0.0, 0.0) * coefs[0];
rhs[(dp - 1)] += u(1.0, 0.0) * coefs[2];
rhs[(dp - 1) * dp] += u(0.0, 1.0) * coefs[6];
rhs[dp * dp - 1] += u(1.0, 1.0) * coefs[8];
}

```

Prints the solution u .

```
void print_solution(int dp, const double *u)
{
    for (int i = 0; i < dp; ++i) {
        for (int j = 0; j < dp; ++j) {
            std::cout << u[i * dp + j] << ' ';
        }
        std::cout << '\n';
    }
    std::cout << std::endl;
}
```

Computes the 1-norm of the error given the computed `u` and the correct solution function `correct_u`.

```
template <typename Closure>
double calculate_error(int dp, const double *u, Closure correct_u)
{
    const auto h = 1.0 / (dp + 1);
    auto error = 0.0;
    for (int j = 0; j < dp; ++j) {
        const auto xi = (j + 1) * h;
        for (int i = 0; i < dp; ++i) {
            using std::abs;
            const auto yi = (i + 1) * h;
            error +=
                abs(u[i * dp + j] - correct_u(xi, yi)) / abs(correct_u(xi, yi));
        }
    }
    return error;
}

void solve_system(const std::string &executor_string,
                  unsigned int discretization_points, int *row_ptrs,
                  int *col_idxes, double *values, double *rhs, double *u,
                  double accuracy)
{

```

Some shortcuts

```
using vec = gko::matrix::Dense<double>;
using mtx = gko::matrix::Csr<double, int>;
using cg = gko::solver::Cg<double>;
using bj = gko::preconditioner::Jacobi<double, int>;
using val_array = gko::Array<double>;
using idx_array = gko::Array<int>;
const auto &dp = discretization_points;
const size_t dp_2 = dp * dp;
```

Figure out where to run the code

```
const auto omp = gko::OmpExecutor::create();
std::map<std::string, std::shared_ptr<gko::Executor>> exec_map{
    {"omp", omp},
    {"cuda", gko::CudaExecutor::create(0, omp)},
    {"reference", gko::ReferenceExecutor::create()}};
```

executor where Ginkgo will perform the computation

```
const auto exec = exec_map.at(executor_string); // throws if not valid
```

executor where the application initialized the data

```
const auto app_exec = exec_map["omp"];
```


Tell Ginkgo to use the data in our application

Matrix: we have to set the executor of the matrix to the one where we want SpMV's to run (in this case `exec`). When creating array views, we have to specify the executor where the data is (in this case `app_exec`).

If the two do not match, Ginkgo will automatically create a copy of the data on `exec` (however, it will not copy the data back once it is done

- here this is not important since we are not modifying the matrix).

```
auto matrix = mtx::create(
    exec, gko::dim<2>(dp_2),
    val_array::view(app_exec, (3 * dp - 2) * (3 * dp - 2), values),
    idx_array::view(app_exec, (3 * dp - 2) * (3 * dp - 2), col_idx),
    idx_array::view(app_exec, dp_2 + 1, row_ptrs));
```

RHS: similar to matrix

```
auto b = vec::create(exec, gko::dim<2>(dp_2, 1),
    val_array::view(app_exec, dp_2, rhs), 1);
```

Solution: we have to be careful here - if the executors are different, once we compute the solution the array will not be automatically copied back to the original memory locations. Fortunately, whenever `apply` is called on a linear operator (e.g. matrix, solver) the arguments automatically get copied to the executor where the operator is, and copied back once the operation is completed. Thus, in this case, we can just define the solution on `app_exec`, and it will be automatically transferred to/from `exec` if needed.

```
auto x = vec::create(app_exec, gko::dim<2>(dp_2, 1),
    val_array::view(app_exec, dp_2, u), 1);
```

Generate solver

```
auto solver_gen =
    cg::build()
        .with_criteria(
            gko::stop::Iteration::build().with_max_iters(dp_2).on(exec),
            gko::stop::ResidualNormReduction<>::build()
                .with_reduction_factor(accuracy)
                .on(exec))
        .with_preconditioner(bj::build().on(exec))
        .on(exec);
auto solver = solver_gen->generate(gko::give(matrix));
```

Solve system

```
solver->apply(gko::lend(b), gko::lend(x));
}

int main(int argc, char *argv[])
{
    if (argc < 2) {
        std::cerr << "Usage: " << argv[0] << " DISCRETIZATION_POINTS [executor]"
            << " [stencil_alpha] [stencil_beta] [stencil_gamma]"
            << std::endl;
        std::exit(-1);
    }

    const int discretization_points = argc >= 2 ? std::atoi(argv[1]) : 100;
    const auto executor_string = argc >= 3 ? argv[2] : "reference";
    const double alpha_c = argc >= 4 ? std::atof(argv[3]) : default_alpha;
    const double beta_c = argc >= 5 ? std::atof(argv[4]) : default_beta;
    const double gamma_c = argc >= 6 ? std::atof(argv[5]) : default_gamma;
```

clang-format off

```
std::array<double, 9> coeffs{
    gamma_c, beta_c, gamma_c,
    beta_c, alpha_c, beta_c,
    gamma_c, beta_c, gamma_c};
```

clang-format on

```
const auto dp = discretization_points;
const size_t dp_2 = dp * dp;
```

problem:

```
auto correct_u = [](double x, double y) { return x * x * x + y * y * y; };
auto f = [](double x, double y) { return 6 * x + 6 * y; };
```

matrix

```
std::vector<int> row_ptrs(dp_2 + 1);
std::vector<int> col_idxes((3 * dp - 2) * (3 * dp - 2));
std::vector<double> values((3 * dp - 2) * (3 * dp - 2));
```

right hand side

```
std::vector<double> rhs(dp_2);
```

solution

```
std::vector<double> u(dp_2, 0.0);
generate_stencil_matrix(dp, row_ptrs.data(), col_idxes.data(), values.data(),
    coeffs.data());
```

looking for solution $u = x^3$: $f = 6x$, $u(0) = 0$, $u(1) = 1$

```
generate_rhs(dp, f, correct_u, rhs.data(), coeffs.data());

auto start_time = std::chrono::steady_clock::now();

solve_system(executor_string, dp, row_ptrs.data(), col_idxes.data(),
    values.data(), rhs.data(), u.data(), 1e-12);

auto stop_time = std::chrono::steady_clock::now();
double runtime_duration =
    std::chrono::duration_cast<std::chrono::nanoseconds>(stop_time -
        start_time)
        .count() *
    1e-6;

print_solution(dp, u.data());
std::cout << "The average relative error is "
    << calculate_error(dp, u.data(), correct_u) / dp_2 << std::endl;
std::cout << "The runtime is " << std::to_string(runtime_duration) << " ms"
    << std::endl;
}
```

Results

The expected output of the relative error at $K=10$ should be

The average relative error is $1.45687e-13$

Comments about programming and debugging

The plain program

```

/*****<GINKGO LICENSE>*****/
Copyright (c) 2017-2019, the Ginkgo authors
All rights reserved.

```

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

```

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*****/

```

```

/*****<DESCRIPTION>*****/
This example solves a 2D Poisson equation:

```

```

\Omega = (0,1)^2
\Omega_b = [0,1]^2 (with boundary)
\partial\Omega = \Omega_b \backslash \Omega
u : \Omega_b -> R
u'' = f in \Omega
u = u_D on \partial\Omega

```

using a finite difference method on an equidistant grid with 'K' discretization points ('K' can be controlled with a command line parameter). The discretization may be done by any order Taylor polynomial.

For an equidistant grid with K "inner" discretization points (x1,y1), ..., (xk,y1), (x1,y2), ..., (xk,yk) step size $h = 1 / (K + 1)$ and a stencil $\setminus R^{3 \times 3}$, the formula produces a system of linear equations

$\sum_{a,b=-1}^1 \text{stencil}(a,b) * u_{\{i+a,j+b\}} = -f_k h^2$, on any inner node with a neighborhood of inner nodes

On any node, where neighbor is on the border, the neighbor is replaced with a $'-\text{stencil}(a,b) * u_{\{i+a,j+b\}}'$ and added to the right hand side vector. For example a node with a neighborhood of only edge nodes may look like this

$\sum_{a,b=-1}^1 \text{stencil}(a,b) * u_{\{i+a,j+b\}} = -f_k h^2 - \sum_{a=-1}^1 \text{stencil}(a,1) * u_{\{i+a,j+1\}}$

which is then solved using Ginkgo's implementation of the CG method preconditioned with block-Jacobi. It is also possible to specify on which executor Ginkgo will solve the system via the command line. The function 'f' is set to $f(x,y) = 6x + 6y$ (making the solution $u(x,y) = x^3 + y^3$), but that can be changed in the 'main' function. Also the stencil values for the core, the faces, the edge and the corners can be changed when passing additional parameters.

The intention of this is to show how generation of stencil values and the right hand side vector changes when increasing the dimension.

*****<DESCRIPTION>*****

```
#include <array>
#include <chrono>
#include <ginkgo/ginkgo.hpp>
#include <iostream>
#include <map>
#include <string>
#include <vector>

constexpr double default_alpha = 10.0 / 3.0;
constexpr double default_beta = -2.0 / 3.0;
constexpr double default_gamma = -1.0 / 6.0;

/* Possible alternative default values are for example
 * default_alpha = 8.0;
 * default_beta = -1.0;
 * default_gamma = -1.0;
 */

void generate_stencil_matrix(int dp, int *row_ptrs, int *col_idxxs,
                           double *values, double *coefs)
{
    int pos = 0;
    const size_t dp_2 = dp * dp;
    row_ptrs[0] = pos;
    for (int k = 0; k < dp; ++k) {
        for (int i = 0; i < dp; ++i) {
            const size_t index = i + k * dp;
            for (int j = -1; j <= 1; ++j) {
                for (int l = -1; l <= 1; ++l) {
                    const int64_t offset = 1 + 1 + 3 * (j + 1);
                    if ((k + j) >= 0 && (k + j) < dp && (i + l) >= 0 &&
                        (i + l) < dp) {
                        values[pos] = coefs[offset];
                        col_idxxs[pos] = index + 1 + dp * j;
                        ++pos;
                    }
                }
            }
            row_ptrs[index + 1] = pos;
        }
    }
}

template <typename Closure, typename ClosureT>
void generate_rhs(int dp, Closure f, ClosureT u, double *rhs, double *coefs)
{
    const size_t dp_2 = dp * dp;
    const auto h = 1.0 / (dp + 1.0);
    for (int i = 0; i < dp; ++i) {
        const auto yi = (i + 1) * h;
        for (int j = 0; j < dp; ++j) {
            const auto xi = (j + 1) * h;
            const auto index = i * dp + j;
            rhs[index] = -f(xi, yi) * h * h;
        }
    }

    for (size_t i = 0; i < dp; ++i) {
        const auto xi = (i + 1) * h;
        const auto index_top = i;
        const auto index_bot = i + dp * (dp - 1);

        rhs[index_top] -= u(xi - h, 0.0) * coefs[0];
        rhs[index_top] -= u(xi, 0.0) * coefs[1];
        rhs[index_top] -= u(xi + h, 0.0) * coefs[2];

        rhs[index_bot] -= u(xi - h, 1.0) * coefs[6];
        rhs[index_bot] -= u(xi, 1.0) * coefs[7];
        rhs[index_bot] -= u(xi + h, 1.0) * coefs[8];
    }

    for (size_t i = 0; i < dp; ++i) {
        const auto yi = (i + 1) * h;
        const auto index_left = i * dp;
        const auto index_right = i * dp + (dp - 1);

        rhs[index_left] -= u(0.0, yi - h) * coefs[0];
        rhs[index_left] -= u(0.0, yi) * coefs[3];
        rhs[index_left] -= u(0.0, yi + h) * coefs[6];

        rhs[index_right] -= u(1.0, yi - h) * coefs[2];
        rhs[index_right] -= u(1.0, yi) * coefs[5];
        rhs[index_right] -= u(1.0, yi + h) * coefs[8];
    }
}
```

```

    }

    rhs[0] += u(0.0, 0.0) * coefs[0];
    rhs[(dp - 1)] += u(1.0, 0.0) * coefs[2];
    rhs[(dp - 1) * dp] += u(0.0, 1.0) * coefs[6];
    rhs[dp * dp - 1] += u(1.0, 1.0) * coefs[8];
}

void print_solution(int dp, const double *u)
{
    for (int i = 0; i < dp; ++i) {
        for (int j = 0; j < dp; ++j) {
            std::cout << u[i * dp + j] << ' ';
        }
        std::cout << '\n';
    }
    std::cout << std::endl;
}

template <typename Closure>
double calculate_error(int dp, const double *u, Closure correct_u)
{
    const auto h = 1.0 / (dp + 1);
    auto error = 0.0;
    for (int j = 0; j < dp; ++j) {
        const auto xi = (j + 1) * h;
        for (int i = 0; i < dp; ++i) {
            using std::abs;
            const auto yi = (i + 1) * h;
            error +=
                abs(u[i * dp + j] - correct_u(xi, yi)) / abs(correct_u(xi, yi));
        }
    }
    return error;
}

void solve_system(const std::string &executor_string,
                  unsigned int discretization_points, int *row_ptrs,
                  int *col_idxxs, double *values, double *rhs, double *u,
                  double accuracy)
{
    using vec = gko::matrix::Dense<double>;
    using mtx = gko::matrix::Csr<double, int>;
    using cg = gko::solver::Cg<double>;
    using bj = gko::preconditioner::Jacobi<double, int>;
    using val_array = gko::Array<double>;
    using idx_array = gko::Array<int>;
    const auto &dp = discretization_points;
    const size_t dp_2 = dp * dp;

    const auto omp = gko::OmpExecutor::create();
    std::map<std::string, std::shared_ptr<gko::Executor>> exec_map{
        {"omp", omp},
        {"cuda", gko::CudaExecutor::create(0, omp)},
        {"reference", gko::ReferenceExecutor::create()};
    const auto exec = exec_map.at(executor_string); // throws if not valid
    const auto app_exec = exec_map["omp"];

    auto matrix = mtx::create(
        exec, gko::dim<2>(dp_2),
        val_array::view(app_exec, (3 * dp - 2) * (3 * dp - 2), values),
        idx_array::view(app_exec, (3 * dp - 2) * (3 * dp - 2), col_idxxs),
        idx_array::view(app_exec, dp_2 + 1, row_ptrs));

    auto b = vec::create(exec, gko::dim<2>(dp_2, 1),
        val_array::view(app_exec, dp_2, rhs), 1);

    auto x = vec::create(app_exec, gko::dim<2>(dp_2, 1),
        val_array::view(app_exec, dp_2, u), 1);

    auto solver_gen =
        cg::build()
            .with_criteria(
                gko::stop::Iteration::build().with_max_iters(dp_2).on(exec),
                gko::stop::ResidualNormReduction<>::build()
                    .with_reduction_factor(accuracy)
                    .on(exec))
            .with_preconditioner(bj::build().on(exec))
            .on(exec);
    auto solver = solver_gen->generate(gko::give(matrix));

    solver->apply(gko::lend(b), gko::lend(x));
}

```

```

int main(int argc, char *argv[])
{
    if (argc < 2) {
        std::cerr << "Usage: " << argv[0] << " DISCRETIZATION_POINTS [executor]"
        << " [stencil_alpha] [stencil_beta] [stencil_gamma]"
        << std::endl;
        std::exit(-1);
    }

    const int discretization_points = argc >= 2 ? std::atoi(argv[1]) : 100;
    const auto executor_string = argc >= 3 ? argv[2] : "reference";
    const double alpha_c = argc >= 4 ? std::atof(argv[3]) : default_alpha;
    const double beta_c = argc >= 5 ? std::atof(argv[4]) : default_beta;
    const double gamma_c = argc >= 6 ? std::atof(argv[5]) : default_gamma;

    std::array<double, 9> coefs{
        gamma_c, beta_c, gamma_c,
        beta_c, alpha_c, beta_c,
        gamma_c, beta_c, gamma_c};

    const auto dp = discretization_points;
    const size_t dp_2 = dp * dp;

    auto correct_u = [](double x, double y) { return x * x * x + y * y * y; };
    auto f = [](double x, double y) { return 6 * x + 6 * y; };

    std::vector<int> row_ptrs(dp_2 + 1);
    std::vector<int> col_idxs((3 * dp - 2) * (3 * dp - 2));
    std::vector<double> values((3 * dp - 2) * (3 * dp - 2));
    std::vector<double> rhs(dp_2);
    std::vector<double> u(dp_2, 0.0);

    generate_stencil_matrix(dp, row_ptrs.data(), col_idxs.data(), values.data(),
        coefs.data());
    generate_rhs(dp, f, correct_u, rhs.data(), coefs.data());

    auto start_time = std::chrono::steady_clock::now();

    solve_system(executor_string, dp, row_ptrs.data(), col_idxs.data(),
        values.data(), rhs.data(), u.data(), 1e-12);

    auto stop_time = std::chrono::steady_clock::now();
    double runtime_duration =
        std::chrono::duration_cast<std::chrono::nanoseconds>(stop_time -
            start_time)
        .count() *
        1e-6;

    print_solution(dp, u.data());
    std::cout << "The average relative error is "
        << calculate_error(dp, u.data(), correct_u) / dp_2 << std::endl;
    std::cout << "The runtime is " << std::to_string(runtime_duration) << " ms"
        << std::endl;
}

```

Chapter 15

The papi-logging program

The papi logging example.

This example depends on simple-solver-logging.

Introduction

About the example

The commented program

```
#include <ginkgo/ginkgo.hpp>

#include <papi.h>
#include <fstream>
#include <iostream>
#include <string>
#include <thread>

namespace {

void papi_add_event(const std::string &event_name, int &eventset)
{
    int code;
    int ret_val = PAPI_event_name_to_code(event_name.c_str(), &code);
    if (PAPI_OK != ret_val) {
        std::cerr << "Error at PAPI_name_to_code()" << std::endl;
        std::exit(-1);
    }

    ret_val = PAPI_add_event(eventset, code);
    if (PAPI_OK != ret_val) {
        std::cerr << "Error at PAPI_name_to_code()" << std::endl;
        std::exit(-1);
    }
}

template <typename T>
std::string to_string(T *ptr)
{
    std::ostringstream os;
    os << reinterpret_cast<gko::uintptr>(ptr);
    return os.str();
}

} // namespace

int init_papi_counters(std::string solver_name, std::string A_name)
{

```

Initialize PAPI, add events and start it up

```
int eventset = PAPI_NULL;
int ret_val = PAPI_library_init(PAPI_VER_CURRENT);
if (ret_val != PAPI_VER_CURRENT) {
    std::cerr << "Error at PAPI_library_init()" << std::endl;
    std::exit(-1);
}
ret_val = PAPI_create_eventset(&eventset);
if (PAPI_OK != ret_val) {
    std::cerr << "Error at PAPI_create_eventset()" << std::endl;
    std::exit(-1);
}

std::string simple_apply_string("sde::ginkgo0::linop_apply_completed:");
std::string advanced_apply_string(
    "sde::ginkgo0::linop_advanced_apply_completed:");
papi_add_event(simple_apply_string + solver_name, eventset);
papi_add_event(simple_apply_string + A_name, eventset);
papi_add_event(advanced_apply_string + A_name, eventset);

ret_val = PAPI_start(eventset);
if (PAPI_OK != ret_val) {
    std::cerr << "Error at PAPI_start()" << std::endl;
    std::exit(-1);
}
return eventset;
}

void print_papi_counters(int eventset)
{
```

Stop PAPI and read the linop_apply_completed event for all of them

```
long long int values[3];
int ret_val = PAPI_stop(eventset, values);
if (PAPI_OK != ret_val) {
    std::cerr << "Error at PAPI_stop()" << std::endl;
    std::exit(-1);
}

PAPI_shutdown();
```

Print all values returned from PAPI

```
std::cout << "PAPI SDE counters:" << std::endl;
std::cout << "solver did " << values[0] << " applies." << std::endl;
std::cout << "A did " << values[1] << " simple applies." << std::endl;
std::cout << "A did " << values[2] << " advanced applies." << std::endl;
}

int main(int argc, char *argv[])
{
```

Some shortcuts

```
using vec = gko::matrix::Dense<>;
using mtx = gko::matrix::Csr<>;
using cg = gko::solver::Cg<>;
```

Print version information

```
std::cout << gko::version_info::get() << std::endl;
```

Figure out where to run the code


```
std::shared_ptr<gko::Executor> exec;
if (argc == 1 || std::string(argv[1]) == "reference") {
    exec = gko::ReferenceExecutor::create();
} else if (argc == 2 && std::string(argv[1]) == "omp") {
    exec = gko::OmpExecutor::create();
} else if (argc == 2 && std::string(argv[1]) == "cuda" &&
           gko::CudaExecutor::get_num_devices() > 0) {
    exec = gko::CudaExecutor::create(0,
    gko::OmpExecutor::create());
} else {
    std::cerr << "Usage: " << argv[0] << " [executor]" << std::endl;
    std::exit(-1);
}
```

Read data

```
auto A = share(gko::read<mtx>(std::ifstream("data/A.mtx"), exec));
auto b = gko::read<vec>(std::ifstream("data/b.mtx"), exec);
auto x = gko::read<vec>(std::ifstream("data/x0.mtx"), exec);
```

Generate solver

```
auto solver_gen =
    cg::build()
        .with_criteria(
            gko::stop::Iteration::build().with_max_iters(20u).on(exec),
            gko::stop::ResidualNormReduction<>::build()
                .with_reduction_factor(1e-20)
                .on(exec))
        .on(exec);
auto solver = solver_gen->generate(A);
```

In this example, we split as much as possible the Ginkgo solver/logger and the PAPI interface. Note that the PAPI ginkgo namespaces are of the form `sde::ginkgo<x>` where `<x>` starts from 0 and is incremented with every new PAPI logger.

```
int eventset =
    init_papi_counters(to_string(solver.get()), to_string(A.get()));
```

Create a PAPI logger and add it to relevant LinOps

```
auto logger = gko::log::Papi<>::create(
    exec, gko::log::Logger::linop_apply_completed_mask |
        gko::log::Logger::linop_advanced_apply_completed_mask);
solver->add_logger(logger);
A->add_logger(logger);
```

Solve system

```
solver->apply(lend(b), lend(x));
```

Stop PAPI event gathering and print the counters

```
print_papi_counters(eventset);
```

Print solution

```
std::cout << "Solution (x): \n";
write(std::cout, lend(x));
```

Calculate residual

```
auto one = gko::initialize<vec>({1.0}, exec);
auto neg_one = gko::initialize<vec>({-1.0}, exec);
auto res = gko::initialize<vec>({0.0}, exec);
A->apply(lend(one), lend(x), lend(neg_one), lend(b));
b->compute_norm2(lend(res));

std::cout << "Residual norm sqrt(r^T r): \n";
write(std::cout, lend(res));
}
```

Results

The following is the expected result:

```
PAPI SDE counters:
solver did 1 applies.
A did 20 simple applies.
A did 1 advanced applies.
Solution (x):
%%MatrixMarket matrix array real general
19 1
0.252218
0.108645
0.0662811
0.0630433
0.0384088
0.0396536
0.0402648
0.0338935
0.0193098
0.0234653
0.0211499
0.0196413
0.0199151
0.0181674
0.0162722
0.0150714
0.0107016
0.0121141
0.0123025
Residual norm sqrt(r^T r):
%%MatrixMarket matrix array real general
1 1
8.87107e-16
```

Comments about programming and debugging

The plain program

```
/******<GINKGO LICENSE>*****
Copyright (c) 2017-2019, the Ginkgo authors
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its
contributors may be used to endorse or promote products derived from
this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*****<GINKGO LICENSE>*****/

#include <ginkgo/ginkgo.hpp>

#include <papi.h>
#include <fstream>
#include <iostream>
```

```

#include <string>
#include <thread>

namespace {

void papi_add_event(const std::string &event_name, int &eventset)
{
    int code;
    int ret_val = PAPI_event_name_to_code(event_name.c_str(), &code);
    if (PAPI_OK != ret_val) {
        std::cerr << "Error at PAPI_name_to_code()" << std::endl;
        std::exit(-1);
    }

    ret_val = PAPI_add_event(eventset, code);
    if (PAPI_OK != ret_val) {
        std::cerr << "Error at PAPI_name_to_code()" << std::endl;
        std::exit(-1);
    }
}

template <typename T>
std::string to_string(T *ptr)
{
    std::ostringstream os;
    os << reinterpret_cast<gko::uintptr>(ptr);
    return os.str();
}

} // namespace

int init_papi_counters(std::string solver_name, std::string A_name)
{
    int eventset = PAPI_NULL;
    int ret_val = PAPI_library_init(PAPI_VER_CURRENT);
    if (ret_val != PAPI_VER_CURRENT) {
        std::cerr << "Error at PAPI_library_init()" << std::endl;
        std::exit(-1);
    }
    ret_val = PAPI_create_eventset(&eventset);
    if (PAPI_OK != ret_val) {
        std::cerr << "Error at PAPI_create_eventset()" << std::endl;
        std::exit(-1);
    }

    std::string simple_apply_string("sde::ginkgo0::linop_apply_completed::");
    std::string advanced_apply_string(
        "sde::ginkgo0::linop_advanced_apply_completed::");
    papi_add_event(simple_apply_string + solver_name, eventset);
    papi_add_event(simple_apply_string + A_name, eventset);
    papi_add_event(advanced_apply_string + A_name, eventset);

    ret_val = PAPI_start(eventset);
    if (PAPI_OK != ret_val) {
        std::cerr << "Error at PAPI_start()" << std::endl;
        std::exit(-1);
    }
    return eventset;
}

void print_papi_counters(int eventset)
{
    long long int values[3];
    int ret_val = PAPI_stop(eventset, values);
    if (PAPI_OK != ret_val) {
        std::cerr << "Error at PAPI_stop()" << std::endl;
        std::exit(-1);
    }

    PAPI_shutdown();

    std::cout << "PAPI SDE counters:" << std::endl;
    std::cout << "solver did " << values[0] << " applies." << std::endl;
    std::cout << "A did " << values[1] << " simple applies." << std::endl;
    std::cout << "A did " << values[2] << " advanced applies." << std::endl;
}

int main(int argc, char *argv[])
{
    using vec = gko::matrix::Dense<>;

```

```

using mtx = gko::matrix::Csr<>;
using cg = gko::solver::Cg<>;

std::cout << gko::version_info::get() << std::endl;

std::shared_ptr<gko::Executor> exec;
if (argc == 1 || std::string(argv[1]) == "reference") {
    exec = gko::ReferenceExecutor::create();
} else if (argc == 2 && std::string(argv[1]) == "omp") {
    exec = gko::OmpExecutor::create();
} else if (argc == 2 && std::string(argv[1]) == "cuda" &&
    gko::CudaExecutor::get_num_devices() > 0) {
    exec = gko::CudaExecutor::create(0,
    gko::OmpExecutor::create());
} else {
    std::cerr << "Usage: " << argv[0] << " [executor]" << std::endl;
    std::exit(-1);
}

auto A = share(gko::read<mtx>(std::ifstream("data/A.mtx"), exec));
auto b = gko::read<vec>(std::ifstream("data/b.mtx"), exec);
auto x = gko::read<vec>(std::ifstream("data/x0.mtx"), exec);

auto solver_gen =
    cg::build()
        .with_criteria(
            gko::stop::Iteration::build().with_max_iters(20u).on(exec),
            gko::stop::ResidualNormReduction<>::build()
                .with_reduction_factor(1e-20)
                .on(exec))
        .on(exec);
auto solver = solver_gen->generate(A);

int eventset =
    init_papi_counters(to_string(solver.get()), to_string(A.get()));

auto logger = gko::log::Papi<>::create(
    exec, gko::log::Logger::linop_apply_completed_mask |
    gko::log::Logger::linop_advanced_apply_completed_mask);
solver->add_logger(logger);
A->add_logger(logger);

solver->apply(lend(b), lend(x));

print_papi_counters(eventset);

std::cout << "Solution (x): \n";
write(std::cout, lend(x));

auto one = gko::initialize<vec>({1.0}, exec);
auto neg_one = gko::initialize<vec>({-1.0}, exec);
auto res = gko::initialize<vec>({0.0}, exec);
A->apply(lend(one), lend(x), lend(neg_one), lend(b));
b->compute_norm2(lend(res));

std::cout << "Residual norm sqrt(r^T r): \n";
write(std::cout, lend(res));
}

```

Chapter 16

The poisson-solver program

The poisson solver example.

This example depends on simple-solver.

Introduction

This example solves a 1D Poisson equation:

$$\begin{aligned} u &: [0, 1] \rightarrow \mathbb{R} \\ u'' &= f \\ u(0) &= u_0 \\ u(1) &= u_1 \end{aligned}$$

using a finite difference method on an equidistant grid with K discretization points (K can be controlled with a command line parameter). The discretization is done via the second order Taylor polynomial:

$$\begin{aligned} u(x+h) &= u(x) + u'(x)h + 1/2u''(x)h^2 + O(h^3) \\ u(x-h) &= u(x) - u'(x)h + 1/2u''(x)h^2 + O(h^3) \\ \hline -u(x-h) + 2u(x) - u(x+h) &= -f(x)h^2 + O(h^3) \end{aligned}$$

For an equidistant grid with K "inner" discretization points x_1, \dots, x_K , and step size $h = 1/(K+1)$, the formula produces a system of linear equations

$$\begin{aligned} 2u_1 - u_2 &= -f_1 h^2 + u_0 \\ -u_{k-1} + 2u_k - u_{k+1} &= -f_k h^2, k = 2, \dots, K-1 \\ -u_{K-1} + 2u_K &= -f_K h^2 + u_1 \end{aligned}$$

which is then solved using Ginkgo's implementation of the CG method preconditioned with block-Jacobi. It is also possible to specify on which executor Ginkgo will solve the system via the command line. The function `f` is set to `f(x) = 6x` (making the solution `u(x) = x3`), but that can be changed in the `main` function.

The intention of the example is to show how Ginkgo can be used to build an application solving a real-world problem, which includes a solution of a large, sparse linear system as a component.

About the example

The commented program

```
#include <ginkgo/ginkgo.hpp>
#include <iostream>
#include <map>
#include <string>
#include <vector>
```

Creates a stencil matrix in CSR format for the given number of discretization points.

```
void generate_stencil_matrix(gko::matrix::Csr<> *matrix)
{
    const auto discretization_points = matrix->get_size()[0];
    auto row_ptrs = matrix->get_row_ptrs();
    auto col_idxs = matrix->get_col_idxs();
    auto values = matrix->get_values();
    int pos = 0;
    const double coefs[] = {-1, 2, -1};
    row_ptrs[0] = pos;
    for (int i = 0; i < discretization_points; ++i) {
        for (auto ofs : {-1, 0, 1}) {
            if (0 <= i + ofs && i + ofs < discretization_points) {
                values[pos] = coefs[ofs + 1];
                col_idxs[pos] = i + ofs;
                ++pos;
            }
        }
        row_ptrs[i + 1] = pos;
    }
}
```

Generates the RHS vector given f and the boundary conditions.

```
template <typename Closure>
void generate_rhs(Closure f, double u0, double u1, gko::matrix::Dense<> *rhs)
{
    const auto discretization_points = rhs->get_size()[0];
    auto values = rhs->get_values();
    const auto h = 1.0 / (discretization_points + 1);
    for (int i = 0; i < discretization_points; ++i) {
        const auto xi = (i + 1) * h;
        values[i] = -f(xi) * h * h;
    }
    values[0] += u0;
    values[discretization_points - 1] += u1;
}
```

Prints the solution u .

```
void print_solution(double u0, double u1, const gko::matrix::Dense<> *u)
{
    std::cout << u0 << '\n';
    for (int i = 0; i < u->get_size()[0]; ++i) {
        std::cout << u->get_const_values()[i] << '\n';
    }
    std::cout << u1 << std::endl;
}
```

Computes the 1-norm of the error given the computed u and the correct solution function `correct_u`.

```

template <typename Closure>
double calculate_error(int discretization_points, const gko::matrix::Dense<> *u,
                      Closure correct_u)
{
    const auto h = 1.0 / (discretization_points + 1);
    auto error = 0.0;
    for (int i = 0; i < discretization_points; ++i) {
        using std::abs;
        const auto xi = (i + 1) * h;
        error +=
            abs(u->get_const_values()[i] - correct_u(xi)) /
            abs(correct_u(xi));
    }
    return error;
}

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

```

Some shortcuts

```

using vec = gko::matrix::Dense<double>;
using mtx = gko::matrix::Csr<double, int>;
using cg = gko::solver::Cg<double>;
using bj = gko::preconditioner::Jacobi<>;

if (argc < 2) {
    std::cerr << "Usage: " << argv[0] << " DISCRETIZATION_POINTS [executor]"
               << std::endl;
    std::exit(-1);
}

```

Get number of discretization points

```

const unsigned int discretization_points =
    argc >= 2 ? std::atoi(argv[1]) : 100;
const auto executor_string = argc >= 3 ? argv[2] : "reference";

```

Figure out where to run the code

```

const auto omp = gko::OmpExecutor::create();
std::map<std::string, std::shared_ptr<gko::Executor>> exec_map{
    {"omp", omp},
    {"cuda", gko::CudaExecutor::create(0, omp)},
    {"reference", gko::ReferenceExecutor::create()}};

```

executor where Ginkgo will perform the computation

```

const auto exec = exec_map.at(executor_string); // throws if not valid

```

executor used by the application

```

const auto app_exec = exec_map["omp"];

```

problem:

```

auto correct_u = [](double x) { return x * x * x; };
auto f = [](double x) { return 6 * x; };
auto u0 = correct_u(0);
auto u1 = correct_u(1);

```

initialize matrix and vectors

```

auto matrix = mtx::create(app_exec, gko::dim<2>(discretization_points),
                          3 * discretization_points - 2);
generate_stencil_matrix(lend(matrix));
auto rhs = vec::create(app_exec, gko::dim<2>(discretization_points, 1));
generate_rhs(f, u0, u1, lend(rhs));
auto u = vec::create(app_exec, gko::dim<2>(discretization_points, 1));
for (int i = 0; i < u->get_size()[0]; ++i) {
    u->get_values()[i] = 0.0;
}

```

Generate solver and solve the system

```

cg::build()
    .with_criteria(gko::stop::Iteration::build()
                  .with_max_iters(discretization_points)
                  .on(exec),
                  gko::stop::ResidualNormReduction<>::build()
                  .with_reduction_factor(1e-6)
                  .on(exec))
    .with_preconditioner(bj::build().on(exec))
    .on(exec)
    ->generate(clone(exec, matrix)) // copy the matrix to the executor
    ->apply(lend(rhs), lend(u));

print_solution(u0, u1, lend(u));
std::cout << "The average relative error is "
            << calculate_error(discretization_points, lend(u), correct_u) /
              discretization_points
            << std::endl;
}

```

Results

This is the expected output:

```

0
0.00010798
0.000863838
0.00291545
0.0069107
0.0134975
0.0233236
0.037037
0.0552856
0.0787172
0.10798
0.143721
0.186589
0.237231
0.296296
0.364431
0.442285
0.530504
0.629738
0.740633
0.863838
1
The average relative error is 1.87318e-15

```

Comments about programming and debugging

The plain program

```

/*****<GINKGO LICENSE>*****/
Copyright (c) 2017-2019, the Ginkgo authors
All rights reserved.

Redistribution and use in source and binary forms, with or without

```


modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *****<GINKGO LICENSE>*****

```
#include <ginkgo/ginkgo.hpp>
#include <iostream>
#include <map>
#include <string>
#include <vector>

void generate_stencil_matrix(gko::matrix::Csr<> *matrix)
{
    const auto discretization_points = matrix->get_size()[0];
    auto row_ptrs = matrix->get_row_ptrs();
    auto col_idxs = matrix->get_col_idxs();
    auto values = matrix->get_values();
    int pos = 0;
    const double coefs[] = {-1, 2, -1};
    row_ptrs[0] = pos;
    for (int i = 0; i < discretization_points; ++i) {
        for (auto ofs : {-1, 0, 1}) {
            if (0 <= i + ofs && i + ofs < discretization_points) {
                values[pos] = coefs[ofs + 1];
                col_idxs[pos] = i + ofs;
                ++pos;
            }
        }
        row_ptrs[i + 1] = pos;
    }
}

template <typename Closure>
void generate_rhs(Closure f, double u0, double u1, gko::matrix::Dense<> *rhs)
{
    const auto discretization_points = rhs->get_size()[0];
    auto values = rhs->get_values();
    const auto h = 1.0 / (discretization_points + 1);
    for (int i = 0; i < discretization_points; ++i) {
        const auto xi = (i + 1) * h;
        values[i] = -f(xi) * h * h;
    }
    values[0] += u0;
    values[discretization_points - 1] += u1;
}

void print_solution(double u0, double u1, const gko::matrix::Dense<> *u)
{
    std::cout << u0 << '\n';
    for (int i = 0; i < u->get_size()[0]; ++i) {
        std::cout << u->get_const_values()[i] << '\n';
    }
    std::cout << u1 << std::endl;
}

template <typename Closure>
double calculate_error(int discretization_points, const gko::matrix::Dense<> *u,
    Closure correct_u)
{
    const auto h = 1.0 / (discretization_points + 1);
    auto error = 0.0;

```

```

    for (int i = 0; i < discretization_points; ++i) {
        using std::abs;
        const auto xi = (i + 1) * h;
        error +=
            abs(u->get_const_values()[i] - correct_u(xi)) /
            abs(correct_u(xi));
    }
    return error;
}

int main(int argc, char *argv[])
{
    using vec = gko::matrix::Dense<double>;
    using mtx = gko::matrix::Csr<double, int>;
    using cg = gko::solver::Cg<double>;
    using bj = gko::preconditioner::Jacobi<>;

    if (argc < 2) {
        std::cerr << "Usage: " << argv[0] << " DISCRETIZATION_POINTS [executor]"
            << std::endl;
        std::exit(-1);
    }

    const unsigned int discretization_points =
        argc >= 2 ? std::atoi(argv[1]) : 100;
    const auto executor_string = argc >= 3 ? argv[2] : "reference";

    const auto omp = gko::OmpExecutor::create();
    std::map<std::string, std::shared_ptr<gko::Executor>> exec_map{
        {"omp", omp},
        {"cuda", gko::CudaExecutor::create(0, omp)},
        {"reference", gko::ReferenceExecutor::create()}};

    const auto exec = exec_map.at(executor_string); // throws if not valid
    const auto app_exec = exec_map["omp"];

    auto correct_u = [](double x) { return x * x * x; };
    auto f = [](double x) { return 6 * x; };
    auto u0 = correct_u(0);
    auto u1 = correct_u(1);

    auto matrix = mtx::create(app_exec, gko::dim<2>(discretization_points),
        3 * discretization_points - 2);
    generate_stencil_matrix(lend(matrix));
    auto rhs = vec::create(app_exec, gko::dim<2>(discretization_points, 1));
    generate_rhs(f, u0, u1, lend(rhs));
    auto u = vec::create(app_exec, gko::dim<2>(discretization_points, 1));
    for (int i = 0; i < u->get_size()[0]; ++i) {
        u->get_values()[i] = 0.0;
    }

    cg::build()
        .with_criteria(gko::stop::Iteration::build()
            .with_max_iters(discretization_points)
            .on(exec),
            gko::stop::ResidualNormReduction<>::build()
                .with_reduction_factor(1e-6)
                .on(exec))
        .with_preconditioner(bj::build().on(exec))
        .on(exec)
        ->generate(clone(exec, matrix)) // copy the matrix to the executor
        ->apply(lend(rhs), lend(u));

    print_solution(u0, u1, lend(u));
    std::cout << "The average relative error is "
        << calculate_error(discretization_points, lend(u), correct_u) /
            discretization_points
        << std::endl;
}

```

Chapter 17

The preconditioned-solver program

The preconditioned solver example.

This example depends on simple-solver.

Introduction

About the example

The commented program

```
#include <ginkgo/ginkgo.hpp>

#include <fstream>
#include <iostream>
#include <string>

int main(int argc, char *argv[])
{
```

Some shortcuts

```
using vec = gko::matrix::Dense<>;
using mtx = gko::matrix::Csr<>;
using cg = gko::solver::Cg<>;
using bj = gko::preconditioner::Jacobi<>;
```

Print version information

```
std::cout << gko::version_info::get() << std::endl;
```

Figure out where to run the code

```
std::shared_ptr<gko::Executor> exec;
if (argc == 1 || std::string(argv[1]) == "reference") {
    exec = gko::ReferenceExecutor::create();
} else if (argc == 2 && std::string(argv[1]) == "omp") {
    exec = gko::OmpExecutor::create();
} else if (argc == 2 && std::string(argv[1]) == "cuda" &&
           gko::CudaExecutor::get_num_devices() > 0) {
    exec = gko::CudaExecutor::create(0,
    gko::OmpExecutor::create());
} else {
    std::cerr << "Usage: " << argv[0] << " [executor]" << std::endl;
    std::exit(-1);
}
```

Read data

```
auto A = share(gko::read<mtx>(std::ifstream("data/A.mtx"), exec));
auto b = gko::read<vec>(std::ifstream("data/b.mtx"), exec);
auto x = gko::read<vec>(std::ifstream("data/x0.mtx"), exec);
```

Create solver factory

```
auto solver_gen =
    cg::build()
        .with_criteria(
            gko::stop::Iteration::build().with_max_iters(20u).on(exec),
            gko::stop::ResidualNormReduction<>::build()
                .with_reduction_factor(1e-20)
                .on(exec))
```

Add preconditioner, these 2 lines are the only difference from the simple solver example

```
.with_preconditioner(bj::build().with_max_block_size(8u).on(exec))
.on(exec);
```

Create solver

```
auto solver = solver_gen->generate(A);
```

Solve system

```
solver->apply(lend(b), lend(x));
```

Print solution

```
std::cout << "Solution (x): \n";
write(std::cout, lend(x));
```

Calculate residual

```
auto one = gko::initialize<vec>({1.0}, exec);
auto neg_one = gko::initialize<vec>({-1.0}, exec);
auto res = gko::initialize<vec>({0.0}, exec);
A->apply(lend(one), lend(x), lend(neg_one), lend(b));
b->compute_norm2(lend(res));

std::cout << "Residual norm sqrt(r^T r): \n";
write(std::cout, lend(res));
}
```

Results

This is the expected output:

```
Solution (x):
%%MatrixMarket matrix array real general
19 1
0.252218
0.108645
0.0662811
0.0630433
0.0384088
0.0396536
0.0402648
0.0338935
0.0193098
0.0234653
0.0211499
0.0196413
0.0199151
0.0181674
0.0162722
0.0150714
0.0107016
0.0121141
0.0123025
Residual norm sqrt(r^T r):
%%MatrixMarket matrix array real general
1 1
9.08137e-16
```

Comments about programming and debugging

The plain program

```
/******<GINKGO LICENSE>*****
Copyright (c) 2017-2019, the Ginkgo authors
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its
contributors may be used to endorse or promote products derived from
this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*****<GINKGO LICENSE>*****/

#include <ginkgo/ginkgo.hpp>

#include <fstream>
#include <iostream>
#include <string>

int main(int argc, char *argv[])
```

```

{
    using vec = gko::matrix::Dense<>;
    using mtx = gko::matrix::Csr<>;
    using cg = gko::solver::Cg<>;
    using bj = gko::preconditioner::Jacobi<>;

    std::cout << gko::version_info::get() << std::endl;

    std::shared_ptr<gko::Executor> exec;
    if (argc == 1 || std::string(argv[1]) == "reference") {
        exec = gko::ReferenceExecutor::create();
    } else if (argc == 2 && std::string(argv[1]) == "omp") {
        exec = gko::OmpExecutor::create();
    } else if (argc == 2 && std::string(argv[1]) == "cuda" &&
               gko::CudaExecutor::get_num_devices() > 0) {
        exec = gko::CudaExecutor::create(0,
        gko::OmpExecutor::create());
    } else {
        std::cerr << "Usage: " << argv[0] << " [executor]" << std::endl;
        std::exit(-1);
    }

    auto A = share(gko::read<mtx>(std::ifstream("data/A.mtx"), exec));
    auto b = gko::read<vec>(std::ifstream("data/b.mtx"), exec);
    auto x = gko::read<vec>(std::ifstream("data/x0.mtx"), exec);

    auto solver_gen =
        cg::build()
            .with_criteria(
                gko::stop::Iteration::build().with_max_iters(20u).on(exec),
                gko::stop::ResidualNormReduction<>::build()
                    .with_reduction_factor(1e-20)
                    .on(exec))
            .with_preconditioner(bj::build().with_max_block_size(8u).on(exec))
            .on(exec);
    auto solver = solver_gen->generate(A);

    solver->apply(lend(b), lend(x));

    std::cout << "Solution (x): \n";
    write(std::cout, lend(x));

    auto one = gko::initialize<vec>({1.0}, exec);
    auto neg_one = gko::initialize<vec>({-1.0}, exec);
    auto res = gko::initialize<vec>({0.0}, exec);
    A->apply(lend(one), lend(x), lend(neg_one), lend(b));
    b->compute_norm2(lend(res));

    std::cout << "Residual norm sqrt(r^T r): \n";
    write(std::cout, lend(res));
}

```

Chapter 18

The simple-solver program

The simple solver example.

Introduction

This simple solver example should help you get started with Ginkgo. This example is meant for you to understand how Ginkgo works and how you can solve a simple linear system with Ginkgo. We encourage you to play with the code, change the parameters and see what is best suited for your purposes.

About the example

Each example has the following sections:

1. **Introduction:** This gives an overview of the example and mentions any interesting aspects in the example that might help the reader.
2. **The commented program:** This section is intended for you to understand the details of the example so that you can play with it and understand Ginkgo and its features better.
3. **Results:** This section shows the results of the code when run. Though the results may not be completely the same, you can expect the behaviour to be similar.
4. **The plain program:** This is the complete code without any comments to have an complete overview of the code.

The commented program

Include files

This is the main ginkgo header file.

```
#include <ginkgo/ginkgo.hpp>
```

Add the fstream header to read from data from files.

```
#include <fstream>
```

Add the C++ iostream header to output information to the console.

```
#include <iostream>
```

Add the string manipulation header to handle strings.

```
#include <string>
```

```
int main(int argc, char *argv[])
{
```

Use some shortcuts. In Ginkgo, vectors are seen as a `gko::matrix::Dense` with one column/one row. The advantage of this concept is that using multiple vectors is a now a natural extension of adding columns/rows are necessary.

```
using vec = gko::matrix::Dense<>;
```

The `gko::matrix::Csr` class is used here, but any other matrix class such as `gko::matrix::Coo`, `gko::matrix::Hybrid`, `gko::matrix::Ell` or `gko::matrix::Sellp` could also be used.

```
using mtx = gko::matrix::Csr<>;
```

The `gko::solver::Cg` is used here, but any other solver class can also be used.

```
using cg = gko::solver::Cg<>;
```

Print the ginkgo version information.

```
std::cout << gko::version_info::get() << std::endl;
```

Where do you want to run your solver ?

The `gko::Executor` class is one of the cornerstones of Ginkgo. Currently, we have support for an `gko::OmpExecutor`, which uses OpenMP multi-threading in most of its kernels, a `gko::ReferenceExecutor`, a single threaded specialization of the OpenMP executor and a `gko::CudaExecutor` which runs the code on a NVIDIA GPU if available.

Note

With the help of C++, you see that you only ever need to change the executor and all the other functions/routines within Ginkgo should automatically work and run on the executor with any other changes.

```
std::shared_ptr<gko::Executor> exec;
if (argc == 1 || std::string(argv[1]) == "reference") {
    exec = gko::ReferenceExecutor::create();
} else if (argc == 2 && std::string(argv[1]) == "omp") {
    exec = gko::OmpExecutor::create();
} else if (argc == 2 && std::string(argv[1]) == "cuda" &&
           gko::CudaExecutor::get_num_devices() > 0) {
    exec = gko::CudaExecutor::create(0,
                                     gko::OmpExecutor::create());
} else {
    std::cerr << "Usage: " << argv[0] << " [executor]" << std::endl;
    std::exit(-1);
}
```


Reading your data and transfer to the proper device.

Read the matrix, right hand side and the initial solution using the read function.

Note

Ginkgo uses C++ smart pointers to automatically manage memory. To this end, we use our own object ownership transfer functions that under the hood call the required smart pointer functions to manage object ownership. The `gko::share`, `gko::give` and `gko::lend` are the functions that you would need to use.

```
auto A = share(gko::read<mtx>(std::ifstream("data/A.mtx"), exec));
auto b = gko::read<vec>(std::ifstream("data/b.mtx"), exec);
auto x = gko::read<vec>(std::ifstream("data/x0.mtx"), exec);
```

Creating the solver

Generate the `gko::solver` factory. Ginkgo uses the concept of Factories to build solvers with certain properties. Observe the Fluent interface used here. Here a cg solver is generated with a stopping criteria of maximum iterations of 20 and a residual norm reduction of $1e-15$. You also observe that the stopping criteria(`gko::stop`) are also generated from factories using their build methods. You need to specify the executors which each of the object needs to be built on.

```
auto solver_gen =
    cg::build()
        .with_criteria(
            gko::stop::Iteration::build().with_max_iters(20u).on(exec),
            gko::stop::ResidualNormReduction<>::build()
                .with_reduction_factor(1e-15)
                .on(exec))
        .on(exec);
```

Generate the solver from the matrix. The solver factory built in the previous step takes a "matrix"(a `gko::LinOp` to be more general) as an input. In this case we provide it with a full matrix that we previously read, but as the solver only effectively uses the `apply()` method within the provided "matrix" object, you can effectively create a `gko::LinOp` class with your own `apply` implementation to accomplish more tasks. We will see an example of how this can be done in the custom-matrix-format example

```
auto solver = solver_gen->generate(A);
```

Finally, solve the system. The solver, being a `gko::LinOp`, can be applied to a right hand side, `b` to obtain the solution, `x`.

```
solver->apply(lend(b), lend(x));
```

Print the solution to the command line.

```
std::cout << "Solution (x): \n";
write(std::cout, lend(x));
```

To measure if your solution has actually converged, you can measure the error of the solution. `one`, `neg_one` are objects that represent the numbers which allow for a uniform interface when computing on any device. To compute the residual, all you need to do is call the `apply` method, which in this case is an `spmv` and equivalent to the LAPACK `z_spmv` routine. Finally, you compute the euclidean 2-norm with the `compute_norm2` function.

```
auto one = gko::initialize<vec>({1.0}, exec);
auto neg_one = gko::initialize<vec>({-1.0}, exec);
auto res = gko::initialize<vec>({0.0}, exec);
A->apply(lend(one), lend(x), lend(neg_one), lend(b));
b->compute_norm2(lend(res));

std::cout << "Residual norm sqrt(r^T r): \n";
write(std::cout, lend(res));
}
```

Results

The following is the expected result:

```
Solution (x):
%%MatrixMarket matrix array real general
19 1
0.252218
0.108645
0.0662811
0.0630433
0.0384088
0.0396536
0.0402648
0.0338935
0.0193098
0.0234653
0.0211499
0.0196413
0.0199151
0.0181674
0.0162722
0.0150714
0.0107016
0.0121141
0.0123025
Residual norm sqrt(r^T r):
%%MatrixMarket matrix array real general
1 1
2.10788e-15
```

Comments about programming and debugging

The plain program

```
/******<GINKGO LICENSE>*****
Copyright (c) 2017-2019, the Ginkgo authors
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its
contributors may be used to endorse or promote products derived from
this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*****<GINKGO LICENSE>*****/

#include <ginkgo/ginkgo.hpp>

#include <fstream>
#include <iostream>
#include <string>

int main(int argc, char *argv[])
{
```

```

using vec = gko::matrix::Dense<>;
using mtx = gko::matrix::Csr<>;
using cg = gko::solver::Cg<>;

std::cout << gko::version_info::get() << std::endl;

std::shared_ptr<gko::Executor> exec;
if (argc == 1 || std::string(argv[1]) == "reference") {
    exec = gko::ReferenceExecutor::create();
} else if (argc == 2 && std::string(argv[1]) == "omp") {
    exec = gko::OmpExecutor::create();
} else if (argc == 2 && std::string(argv[1]) == "cuda" &&
    gko::CudaExecutor::get_num_devices() > 0) {
    exec = gko::CudaExecutor::create(0,
    gko::OmpExecutor::create());
} else {
    std::cerr << "Usage: " << argv[0] << " [executor]" << std::endl;
    std::exit(-1);
}

auto A = share(gko::read<mtx>(std::ifstream("data/A.mtx"), exec));
auto b = gko::read<vec>(std::ifstream("data/b.mtx"), exec);
auto x = gko::read<vec>(std::ifstream("data/x0.mtx"), exec);

auto solver_gen =
    cg::build()
        .with_criteria(
            gko::stop::Iteration::build().with_max_iters(20u).on(exec),
            gko::stop::ResidualNormReduction<>::build()
                .with_reduction_factor(1e-15)
                .on(exec))
        .on(exec);
auto solver = solver_gen->generate(A);

solver->apply(lend(b), lend(x));

std::cout << "Solution (x): \n";
write(std::cout, lend(x));

auto one = gko::initialize<vec>({1.0}, exec);
auto neg_one = gko::initialize<vec>({-1.0}, exec);
auto res = gko::initialize<vec>({0.0}, exec);
A->apply(lend(one), lend(x), lend(neg_one), lend(b));
b->compute_norm2(lend(res));

std::cout << "Residual norm sqrt(r^T r): \n";
write(std::cout, lend(res));
}

```


Chapter 19

The simple-solver-logging program

The simple solver with logging example.

This example depends on simple-solver, minimal-cuda-solver.

Introduction

About the example

The commented program

```
#include <ginkgo/ginkgo.hpp>

#include <fstream>
#include <iomanip>
#include <iostream>
#include <string>

namespace {

void print_vector(const std::string &name, const gko::matrix::Dense<> *vec)
{
    std::cout << name << " = [" << std::endl;
    for (int i = 0; i < vec->get_size()[0]; ++i) {
        std::cout << "      " << vec->at(i, 0) << std::endl;
    }
    std::cout << "];" << std::endl;
}

} // namespace

int main(int argc, char *argv[])
{
```

Some shortcuts

```
using vec = gko::matrix::Dense<>;
using mtx = gko::matrix::Csr<>;
using cg = gko::solver::Cg<>;
```

Print version information

```
std::cout << gko::version_info::get() << std::endl;
```

Figure out where to run the code

```
std::shared_ptr<gko::Executor> exec;
if (argc == 1 || std::string(argv[1]) == "reference") {
    exec = gko::ReferenceExecutor::create();
} else if (argc == 2 && std::string(argv[1]) == "omp") {
    exec = gko::OmpExecutor::create();
} else if (argc == 2 && std::string(argv[1]) == "cuda" &&
    gko::CudaExecutor::get_num_devices() > 0) {
    exec = gko::CudaExecutor::create(0,
    gko::OmpExecutor::create());
} else {
    std::cerr << "Usage: " << argv[0] << " [executor]" << std::endl;
    std::exit(-1);
}
```

Read data

```
auto A = share(gko::read<mtx>(std::ifstream("data/A.mtx"), exec));
auto b = gko::read<vec>(std::ifstream("data/b.mtx"), exec);
auto x = gko::read<vec>(std::ifstream("data/x0.mtx"), exec);
```

Let's declare a logger which prints to `std::cout` instead of printing to a file. We log all events except for all linop factory and polymorphic object events. Events masks are group of events which are provided for convenience.

```
std::shared_ptr<gko::log::Stream<>> stream_logger =
    gko::log::Stream<>::create(
        exec,
        gko::log::Logger::all_events_mask ^
        gko::log::Logger::linop_factory_events_mask ^
        gko::log::Logger::polymorphic_object_events_mask,
        std::cout);
```

Add stream_logger to the executor

```
exec->add_logger(stream_logger);
```

Add `stream_logger` only to the `ResidualNormReduction` criterion Factory Note that the logger will get automatically propagated to every criterion generated from this factory.

```
using ResidualCriterionFactory =
    gko::stop::ResidualNormReduction<>::Factory;
std::shared_ptr<ResidualCriterionFactory> residual_criterion =
    ResidualCriterionFactory::create().with_reduction_factor(1e-20).on(
        exec);
residual_criterion->add_logger(stream_logger);
```

Generate solver

```
auto solver_gen =
    cg::build()
        .with_criteria(
            residual_criterion,
            gko::stop::Iteration::build().with_max_iters(20u).on(exec))
        .on(exec);
auto solver = solver_gen->generate(A);
```

First we add facilities to only print to a file. It's possible to select events, using masks, e.g. only iterations mask: `gko::log::Logger::iteration_complete_mask`. See the documentation of `Logger` class for more information.

```
std::ofstream filestream("my_file.txt");
solver->add_logger(gko::log::Stream<>::create(
    exec, gko::log::Logger::all_events_mask, filestream));
solver->add_logger(stream_logger);
```

Add another logger which puts all the data in an object, we can later retrieve this object in our code. Here we only have want `Executor` and `criterion check completed` events.

```
std::shared_ptr<gko::log::Record> record_logger = gko::log::Record::create(
    exec, gko::log::Logger::executor_events_mask |
        gko::log::Logger::criterion_check_completed_mask);
exec->add_logger(record_logger);
residual_criterion->add_logger(record_logger);
```

Solve system

```
solver->apply(lend(b), lend(x));
```

Finally, get some data from `record_logger` and print the last memory location copied

```
auto &last_copy = record_logger->get().copy_completed.back();
std::cout << "Last memory copied was of size " << std::hex
    << std::get<0>(*last_copy).num_bytes << " FROM executor "
    << std::get<0>(*last_copy).exec << " pointer "
    << std::get<0>(*last_copy).location << " TO executor "
    << std::get<1>(*last_copy).exec << " pointer "
    << std::get<1>(*last_copy).location << std::dec << std::endl;
```

Also print the residual of the last criterion check event (where convergence happened)

```
auto residual =
    record_logger->get().criterion_check_completed.back()->residual.get();
auto residual_d = gko::as<gko::matrix::Dense<>>(residual);
print_vector("Residual", residual_d);
```

Print solution

```
std::cout << "Solution (x): \n";
write(std::cout, lend(x));
```

Calculate residual

```
auto one = gko::initialize<vec>({1.0}, exec);
auto neg_one = gko::initialize<vec>({-1.0}, exec);
auto res = gko::initialize<vec>({0.0}, exec);
A->apply(lend(one), lend(x), lend(neg_one), lend(b));
b->compute_norm2(lend(res));

std::cout << "Residual norm sqrt(r^T r): \n";
write(std::cout, lend(res));
}
```

Results

This is the expected output:

```
[LOG] >>> apply started on A LinOp[gko::solver::Cg<double>,0x55ae09d923f0] with b LinOp[
gko::matrix::Dense<double>,0x55ae09d928b0] and x LinOp[gko::matrix::Dense<double>,0x55ae09d92f10]
[LOG] >>> allocation started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] with Bytes[8]
[LOG] >>> allocation completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[
0x55ae09d91750] with Bytes[8]
[LOG] >>> allocation started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] with Bytes[8]
[LOG] >>> allocation completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[
0x55ae09d977e0] with Bytes[8]
[LOG] >>> allocation started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] with Bytes[152]
[LOG] >>> allocation completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[
0x55ae09d97b40] with Bytes[152]
[LOG] >>> allocation started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] with Bytes[152]
[LOG] >>> allocation completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[
0x55ae09d97d10] with Bytes[152]
[LOG] >>> allocation started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] with Bytes[152]
[LOG] >>> allocation completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[
0x55ae09d97ee0] with Bytes[152]
[LOG] >>> allocation started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] with Bytes[152]
[LOG] >>> allocation completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[
0x55ae09d980b0] with Bytes[152]
[LOG] >>> allocation started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] with Bytes[8]
[LOG] >>> allocation completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[
0x55ae09d98090] with Bytes[8]
[LOG] >>> allocation started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] with Bytes[8]
[LOG] >>> allocation completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[
0x55ae09d98260] with Bytes[8]
[LOG] >>> allocation started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] with Bytes[8]
[LOG] >>> allocation completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[
0x55ae09d983b0] with Bytes[8]
[LOG] >>> allocation started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] with Bytes[8]
[LOG] >>> allocation completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[
0x55ae09d98500] with Bytes[8]
[LOG] >>> allocation started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] with Bytes[1]
[LOG] >>> allocation completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[
0x55ae09d98650] with Bytes[1]
[LOG] >>> Operation[gko::solver::cg::initialize_operation<gko::matrix::Dense<double> const*&,
gko::matrix::Dense<double>*, gko::matrix::Dense<double>*, gko::matrix::Dense<double>*, gko::matrix::Dense<double>*,
gko::matrix::Dense<double>*, gko::matrix::Dense<double>*, gko:
:Array<gko::stopping_status>*>,0x7ffcab765d60] started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0]
[LOG] >>> Operation[gko::solver::cg::initialize_operation<gko::matrix::Dense<double> const*&,
gko::matrix::Dense<double>*, gko::matrix::Dense<double>*, gko:matrix::Dense<double>*, gko::matrix::Dense<double>*,
gko::matrix::Dense<double>*, gko::matrix::Dense<double>*, gko:
:Array<gko::stopping_status>*>,0x7ffcab765d60] completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0]
[LOG] >>> Operation[gko::matrix::csr::advanced_spmv_operation<gko::matrix::Dense<double> const*,
gko::matrix::Csr<double, int> const*, gko::matrix::Dense<double> const*, gko::matrix::Dense<double> const*,
gko::matrix::Dense<double>*>,0x7ffcab765980] started on Executor[gk
o::ReferenceExecutor,0x55ae09d8f2a0]
[LOG] >>> Operation[gko::matrix::csr::advanced_spmv_operation<gko::matrix::Dense<double> const*,
gko::matrix::Csr<double, int> const*, gko::matrix::Dense<double> const*, gko::matrix::Dense<double> const*,
gko::matrix::Dense<double>*>,0x7ffcab765980] completed on Executor[
gko::ReferenceExecutor,0x55ae09d8f2a0]
[LOG] >>> allocation started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] with Bytes[2]
[LOG] >>> allocation completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[
0x55ae09d98690] with Bytes[2]
[LOG] >>> allocation started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] with Bytes[8]
[LOG] >>> allocation completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[
0x55ae09d99310] with Bytes[8]
[LOG] >>> allocation started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] with Bytes[8]
[LOG] >>> allocation completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[
0x55ae09d99350] with Bytes[8]
[LOG] >>> Operation[gko::matrix::dense::compute_norm2_operation<gko::matrix::Dense<double> const*,
gko::matrix::Dense<double>*>,0x7ffcab7657c0] started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0]
[LOG] >>> Operation[gko::matrix::dense::compute_norm2_operation<gko::matrix::Dense<double> const*,
gko::matrix::Dense<double>*>,0x7ffcab7657c0] completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0]
[LOG] >>> copy started from Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] to Executor[
gko::ReferenceExecutor,0x55ae09d8f2a0] from Location[0x55ae09d97b40] to Location[0x55ae09d97d10] with Bytes[152]
[LOG] >>> copy completed from Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] to Executor[
gko::ReferenceExecutor,0x55ae09d8f2a0] from Location[0x55ae09d97b40] to Location[0x55ae09d97d10] with Bytes[152]
[LOG] >>> Operation[gko::matrix::dense::compute_dot_operation<gko::matrix::Dense<double> const*,
gko::matrix::Dense<double> const*, gko::matrix::Dense<double>*>,0x7ffcab7659a0] started on Executor[
gko::ReferenceExecutor,0x55ae09d8f2a0]
[LOG] >>> Operation[gko::matrix::dense::compute_dot_operation<gko::matrix::Dense<double> const*,
gko::matrix::Dense<double> const*, gko::matrix::Dense<double>*>,0x7ffcab7659a0] completed on Executor[
gko::ReferenceExecutor,0x55ae09d8f2a0]
[LOG] >>> iteration 0 completed with solver LinOp[gko::solver::Cg<double>,0x55ae09d923f0] with residual
LinOp[gko::matrix::Dense<double>,0x55ae09d930c0], solution LinOp[gko::matrix::Dense<double>,0x55ae09d92f10] and
residual_norm LinOp[gko::LinOp const*,0]
[LOG] >>> check started for stop::Criterion[gko::stop::ResidualNormReduction<double>,0x55ae09d99260] at
iteration 0 with ID 1 and finalized set to 1
```



```

[LOG] >>> Operation[gko::matrix::dense::compute_norm2_operation<gko::matrix::Dense<double> const*,
gko::matrix::Dense<double>*>,0x7ffcab765740] started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0]
[LOG] >>> Operation[gko::matrix::dense::compute_norm2_operation<gko::matrix::Dense<double> const*,
gko::matrix::Dense<double>*>,0x7ffcab765740] completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0]
[LOG] >>> Operation[
gko::stop::residual_norm_reduction::residual_norm_reduction_operation<gko::matrix::Dense<double> const*&, gko::matrix::Dense<double>*>,
gko::Array<gko::stopping_status>*&, gko::Array<bool>*&, bool*&, bool*&>,0x7ffcab765980]
started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0]
[LOG] >>> Operation[
gko::stop::residual_norm_reduction::residual_norm_reduction_operation<gko::matrix::Dense<double> const*&, gko::matrix::Dense<double>*>,
gko::Array<gko::stopping_status>*&, gko::Array<bool>*&, bool*&, bool*&>,0x7ffcab765980]
completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0]
[LOG] >>> check completed for stop::Criterion[gko::stop::ResidualNormReduction<double>,0x55ae09d99260] at
iteration 0 with ID 1 and finalized set to 1. It changed one RHS 0, stopped the iteration process 0
[LOG] >>> allocation started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] with Bytes[152]
[LOG] >>> allocation completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[
0x55ae09d99820] with Bytes[152]

.
.
.
.
.

[LOG] >>> free started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d98690]
[LOG] >>> free completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d98690]
[LOG] >>> free started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d99350]
[LOG] >>> free completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d99350]
[LOG] >>> free started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d99310]
[LOG] >>> free completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d99310]
[LOG] >>> free started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d98650]
[LOG] >>> free completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d98650]
[LOG] >>> free started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d98500]
[LOG] >>> free completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d98500]
[LOG] >>> free started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d983b0]
[LOG] >>> free completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d983b0]
[LOG] >>> free started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d98260]
[LOG] >>> free completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d98260]
[LOG] >>> free started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d98090]
[LOG] >>> free completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d98090]
[LOG] >>> free started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d980b0]
[LOG] >>> free completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d980b0]
[LOG] >>> free started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d97ee0]
[LOG] >>> free completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d97ee0]
[LOG] >>> free started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d97d10]
[LOG] >>> free completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d97d10]
[LOG] >>> free started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d97b40]
[LOG] >>> free completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d97b40]
[LOG] >>> free started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d977e0]
[LOG] >>> free completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d977e0]
[LOG] >>> free started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d91750]
[LOG] >>> free completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d91750]
[LOG] >>> apply completed on A LinOp[gko::solver::Cg<double>,0x55ae09d923f0] with b LinOp[
gko::matrix::Dense<double>,0x55ae09d928b0] and x LinOp[gko::matrix::Dense<double>,0x55ae09d92f10]
Last memory copied was of size 98 FROM executor 0x55ae09d8f2a0 pointer 55ae09d931d0 TO executor
0x55ae09d8f2a0 pointer 55ae09d998c0
Residual = [
1.3067e-18
-1.34263e-18
-2.7754e-19
2.35392e-20
-2.25114e-19
-1.35474e-20
-1.82049e-19
-2.48092e-19
-4.57754e-19
-1.28163e-18
-1.04918e-18
-5.88231e-19
-8.463e-19
-2.87785e-18
-4.06072e-18
-9.40979e-18
-1.07071e-17
-4.14666e-17
-2.75923e-17
];
Solution (x):
%%MatrixMarket matrix array real general
19 1
0.252218
0.108645
0.0662811
0.0630433
0.0384088
0.0396536

```

```

0.0402648
0.0338935
0.0193098
0.0234653
0.0211499
0.0196413
0.0199151
0.0181674
0.0162722
0.0150714
0.0107016
0.0121141
0.0123025
[LOG] >>> allocation started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] with Bytes[8]
[LOG] >>> allocation completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0
x55ae09d9a310] with Bytes[8]
[LOG] >>> allocation started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] with Bytes[8]
[LOG] >>> allocation completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0
x55ae09d97ec0] with Bytes[8]
[LOG] >>> allocation started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] with Bytes[8]
[LOG] >>> allocation completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0
x55ae09d99370] with Bytes[8]
[LOG] >>> Operation[gko::matrix::csr::advanced_spmv_operation<gko::matrix::Dense<double> const*,
gko::matrix::Csr<double, int> const*, gko::matrix::Dense<double> const*,
gko::matrix::Dense<double> const*, gko::matrix::Dense<double>*>,0x7ffcab765d50] started on Executor[
gk
o::ReferenceExecutor,0x55ae09d8f2a0]
[LOG] >>> Operation[gko::matrix::csr::advanced_spmv_operation<gko::matrix::Dense<double> const*,
gko::matrix::Csr<double, int> const*, gko::matrix::Dense<double> const*,
gko::matrix::Dense<double> const*, gko::matrix::Dense<double>*>,0x7ffcab765d50] completed on Executor
[
gko::ReferenceExecutor,0x55ae09d8f2a0]
[LOG] >>> Operation[gko::matrix::dense::compute_norm2_operation<gko::matrix::Dense<double> const*,
gko::matrix::Dense<double>*>,0x7ffcab765dc0] started on Executor[gko::ReferenceExecutor,0
x55ae09d8f2a0]
[LOG] >>> Operation[gko::matrix::dense::compute_norm2_operation<gko::matrix::Dense<double> const*,
gko::matrix::Dense<double>*>,0x7ffcab765dc0] completed on Executor[gko::ReferenceExecutor,0
x55ae09d8f2a0]
Residual norm sqrt(r^T r):
%%MatrixMarket matrix array real general
1 1
8.87107e-16
[LOG] >>> free started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d99370]
[LOG] >>> free completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d99370]
[LOG] >>> free started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d97ec0]
[LOG] >>> free completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d97ec0]
[LOG] >>> free started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d9a310]
[LOG] >>> free completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d9a310]
[LOG] >>> free started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d931d0]
[LOG] >>> free completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d931d0]
[LOG] >>> free started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d93020]
[LOG] >>> free completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d93020]
[LOG] >>> free started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d92830]
[LOG] >>> free completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d92830]
[LOG] >>> free started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d925b0]
[LOG] >>> free completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d925b0]
[LOG] >>> free started on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d93360]
[LOG] >>> free completed on Executor[gko::ReferenceExecutor,0x55ae09d8f2a0] at Location[0x55ae09d93360]

```

Comments about programming and debugging

The plain program

```

/*****<GINKGO LICENSE>*****/
Copyright (c) 2017-2019, the Ginkgo authors
All rights reserved.

```

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from

this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 ****<GINKGO LICENSE>****/

```
#include <ginkgo/ginkgo.hpp>

#include <fstream>
#include <iomanip>
#include <iostream>
#include <string>

namespace {

void print_vector(const std::string &name, const gko::matrix::Dense<> *vec)
{
    std::cout << name << " = [" << std::endl;
    for (int i = 0; i < vec->get_size()[0]; ++i) {
        std::cout << "      " << vec->at(i, 0) << std::endl;
    }
    std::cout << "];" << std::endl;
}

} // namespace

int main(int argc, char *argv[])
{
    using vec = gko::matrix::Dense<>;
    using mtx = gko::matrix::Csr<>;
    using cg = gko::solver::Cg<>;

    std::cout << gko::version_info::get() << std::endl;

    std::shared_ptr<gko::Executor> exec;
    if (argc == 1 || std::string(argv[1]) == "reference") {
        exec = gko::ReferenceExecutor::create();
    } else if (argc == 2 && std::string(argv[1]) == "omp") {
        exec = gko::OmpExecutor::create();
    } else if (argc == 2 && std::string(argv[1]) == "cuda" &&
               gko::CudaExecutor::get_num_devices() > 0) {
        exec = gko::CudaExecutor::create(0,
                                          gko::OmpExecutor::create());
    } else {
        std::cerr << "Usage: " << argv[0] << " [executor]" << std::endl;
        std::exit(-1);
    }

    auto A = share(gko::read<mtx>(std::ifstream("data/A.mtx"), exec));
    auto b = gko::read<vec>(std::ifstream("data/b.mtx"), exec);
    auto x = gko::read<vec>(std::ifstream("data/x0.mtx"), exec);

    std::shared_ptr<gko::log::Stream<>> stream_logger =
        gko::log::Stream<>::create(
            exec,
            gko::log::Logger::all_events_mask ^
            gko::log::Logger::linop_factory_events_mask ^
            gko::log::Logger::polymorphic_object_events_mask,
            std::cout);

    exec->add_logger(stream_logger);

    using ResidualCriterionFactory =
        gko::stop::ResidualNormReduction<>::Factory;
    std::shared_ptr<ResidualCriterionFactory> residual_criterion =
        ResidualCriterionFactory::create().with_reduction_factor(1e-20).on(
            exec);
    residual_criterion->add_logger(stream_logger);

    auto solver_gen =
        cg::build()
            .with_criteria(
                residual_criterion,
```

```

        gko::stop::Iteration::build().with_max_iters(20u).on(exec)
        .on(exec);
    auto solver = solver_gen->generate(A);

    std::ofstream filestream("my_file.txt");
    solver->add_logger(gko::log::Stream<>::create(
        exec, gko::log::Logger::all_events_mask, filestream));
    solver->add_logger(stream_logger);

    std::shared_ptr<gko::log::Record> record_logger = gko::log::Record::create(
        exec, gko::log::Logger::executor_events_mask |
            gko::log::Logger::criterion_check_completed_mask);
    exec->add_logger(record_logger);
    residual_criterion->add_logger(record_logger);

    solver->apply(lend(b), lend(x));

    auto &last_copy = record_logger->get().copy_completed.back();
    std::cout << "Last memory copied was of size " << std::hex
        << std::get<0>(*last_copy).num_bytes << " FROM executor "
        << std::get<0>(*last_copy).exec << " pointer "
        << std::get<0>(*last_copy).location << " TO executor "
        << std::get<1>(*last_copy).exec << " pointer "
        << std::get<1>(*last_copy).location << std::dec << std::endl;
    auto residual =
        record_logger->get().criterion_check_completed.back()->residual.get();
    auto residual_d = gko::as<gko::matrix::Dense<>>(residual);
    print_vector("Residual", residual_d);

    std::cout << "Solution (x): \n";
    write(std::cout, lend(x));

    auto one = gko::initialize<vec>({1.0}, exec);
    auto neg_one = gko::initialize<vec>({-1.0}, exec);
    auto res = gko::initialize<vec>({0.0}, exec);
    A->apply(lend(one), lend(x), lend(neg_one), lend(b));
    b->compute_norm2(lend(res));

    std::cout << "Residual norm sqrt(r^T r): \n";
    write(std::cout, lend(res));
}

```

Chapter 20

The three-pt-stencil-solver program

The 3-point stencil example.

This example depends on simple-solver, poisson-solver.

Introduction

This example solves a 1D Poisson equation:

$$\begin{aligned}u &: [0, 1] \rightarrow R \\ u'' &= f \\ u(0) &= u0 \\ u(1) &= u1\end{aligned}$$

using a finite difference method on an equidistant grid with K discretization points (K can be controlled with a command line parameter). The discretization is done via the second order Taylor polynomial:

$$\begin{aligned}u(x+h) &= u(x) + u'(x)h + 1/2u''(x)h^2 + O(h^3) \\ u(x-h) &= u(x) - u'(x)h + 1/2u''(x)h^2 + O(h^3) \\ \hline -u(x-h) + 2u(x) - u(x+h) &= -f(x)h^2 + O(h^3)\end{aligned}$$

For an equidistant grid with K "inner" discretization points x_1, \dots, x_K , and step size $h = 1/(K+1)$, the formula produces a system of linear equations

$$\begin{aligned}2u_1 - u_2 &= -f_1h^2 + u0 \\ -u_{(k-1)} + 2u_k - u_{(k+1)} &= -f_kh^2, k = 2, \dots, K-1 \\ -u_{(K-1)} + 2u_K &= -f_Kh^2 + u1\end{aligned}$$

which is then solved using Ginkgo's implementation of the CG method preconditioned with block-Jacobi. It is also possible to specify on which executor Ginkgo will solve the system via the command line. The function ' f ' is set to ' $f(x) = 6x$ ' (making the solution ' $u(x) = x^3$ '), but that can be changed in the `main` function.

The intention of the example is to show how Ginkgo can be integrated into existing software - the `generate_stencil_matrix`, `generate_rhs`, `print_solution`, `compute_error` and `main` function do not reference Ginkgo at all (i.e. they could have been there before the application developer decided to use Ginkgo, and the only part where Ginkgo is introduced is inside the `solve_system` function).

About the example

The commented program

```

/ *****<DECSRIPTION>*****
This example solves a 1D Poisson equation:

    u : [0, 1] -> R
    u'' = f
    u(0) = u0
    u(1) = u1

using a finite difference method on an equidistant grid with 'K' discretization
points ('K' can be controlled with a command line parameter). The discretization
is done via the second order Taylor polynomial:

u(x + h) = u(x) - u'(x)h + 1/2 u''(x)h^2 + O(h^3)
u(x - h) = u(x) + u'(x)h + 1/2 u''(x)h^2 + O(h^3)  / +
-----
-u(x - h) + 2u(x) - u(x + h) = -f(x)h^2 + O(h^3)

For an equidistant grid with K "inner" discretization points x1, ..., xk, and
step size h = 1 / (K + 1), the formula produces a system of linear equations

      2u_1 - u_2      = -f_1 h^2 + u0
-u_(k-1) + 2u_k - u_(k+1) = -f_k h^2,      k = 2, ..., K - 1
-u_(K-1) + 2u_K      = -f_K h^2 + u1

which is then solved using Ginkgo's implementation of the CG method
preconditioned with block-Jacobi. It is also possible to specify on which
executor Ginkgo will solve the system via the command line.
The function 'f' is set to 'f(x) = 6x' (making the solution 'u(x) = x^3'), but
that can be changed in the 'main' function.

The intention of the example is to show how Ginkgo can be integrated into
existing software - the 'generate_stencil_matrix', 'generate_rhs',
'print_solution', 'compute_error' and 'main' function do not reference Ginkgo at
all (i.e. they could have been there before the application developer decided to
use Ginkgo, and the only part where Ginkgo is introduced is inside the
'solve_system' function.
*****<DECSRIPTION>***** /

#include <ginkgo/ginkgo.hpp>
#include <iostream>
#include <map>
#include <string>
#include <vector>

```

Creates a stencil matrix in CSR format for the given number of discretization points.

```

void generate_stencil_matrix(int discretization_points, int *row_ptrs,
                           int *col_idxs, double *values)
{
    int pos = 0;
    const double coefs[] = {-1, 2, -1};
    row_ptrs[0] = pos;
    for (int i = 0; i < discretization_points; ++i) {
        for (auto ofs : {-1, 0, 1}) {
            if (0 <= i + ofs && i + ofs < discretization_points) {
                values[pos] = coefs[ofs + 1];
                col_idxs[pos] = i + ofs;
                ++pos;
            }
        }
        row_ptrs[i + 1] = pos;
    }
}

```

Generates the RHS vector given f and the boundary conditions.

```

template <typename Closure>
void generate_rhs(int discretization_points, Closure f, double u0, double u1,
                double *rhs)
{

```

```

const auto h = 1.0 / (discretization_points + 1);
for (int i = 0; i < discretization_points; ++i) {
    const auto xi = (i + 1) * h;
    rhs[i] = -f(xi) * h * h;
}
rhs[0] += u0;
rhs[discretization_points - 1] += u1;
}

```

Prints the solution `u`.

```

void print_solution(int discretization_points, double u0, double u1,
                   const double *u)
{
    std::cout << u0 << '\n';
    for (int i = 0; i < discretization_points; ++i) {
        std::cout << u[i] << '\n';
    }
    std::cout << u1 << std::endl;
}

```

Computes the 1-norm of the error given the computed `u` and the correct solution function `correct_u`.

```

template <typename Closure>
double calculate_error(int discretization_points, const double *u,
                      Closure correct_u)
{
    const auto h = 1.0 / (discretization_points + 1);
    auto error = 0.0;
    for (int i = 0; i < discretization_points; ++i) {
        using std::abs;
        const auto xi = (i + 1) * h;
        error += abs(u[i] - correct_u(xi)) / abs(correct_u(xi));
    }
    return error;
}

void solve_system(const std::string &executor_string,
                  unsigned int discretization_points, int *row_ptrs,
                  int *col_idxs, double *values, double *rhs, double *u,
                  double accuracy)
{

```

Some shortcuts

```

using vec = gko::matrix::Dense<double>;
using mtx = gko::matrix::Csr<double, int>;
using cg = gko::solver::Cg<double>;
using bj = gko::preconditioner::Jacobi<double, int>;
using val_array = gko::Array<double>;
using idx_array = gko::Array<int>;
const auto &dp = discretization_points;

```

Figure out where to run the code

```

const auto omp = gko::OmpExecutor::create();
std::map<std::string, std::shared_ptr<gko::Executor>> exec_map{
    {"omp", omp},
    {"cuda", gko::CudaExecutor::create(0, omp)},
    {"reference", gko::ReferenceExecutor::create()}};

```

executor where Ginkgo will perform the computation

```

const auto exec = exec_map.at(executor_string); // throws if not valid

```

executor where the application initialized the data

```
const auto app_exec = exec_map["omp"];
```

Tell Ginkgo to use the data in our application

Matrix: we have to set the executor of the matrix to the one where we want SpMV's to run (in this case `exec`). When creating array views, we have to specify the executor where the data is (in this case `app_exec`).

If the two do not match, Ginkgo will automatically create a copy of the data on `exec` (however, it will not copy the data back once it is done

- here this is not important since we are not modifying the matrix).

```
auto matrix = mtx::create(exec, gko::dim<2>(dp),
    val_array::view(app_exec, 3 * dp - 2, values),
    idx_array::view(app_exec, 3 * dp - 2, col_idxs),
    idx_array::view(app_exec, dp + 1, row_ptrs));
```

RHS: similar to matrix

```
auto b = vec::create(exec, gko::dim<2>(dp, 1),
    val_array::view(app_exec, dp, rhs), 1);
```

Solution: we have to be careful here - if the executors are different, once we compute the solution the array will not be automatically copied back to the original memory locations. Fortunately, whenever `apply` is called on a linear operator (e.g. matrix, solver) the arguments automatically get copied to the executor where the operator is, and copied back once the operation is completed. Thus, in this case, we can just define the solution on `app_exec`, and it will be automatically transferred to/from `exec` if needed.

```
auto x = vec::create(app_exec, gko::dim<2>(dp, 1),
    val_array::view(app_exec, dp, u), 1);
```

Generate solver

```
auto solver_gen =
    cg::build()
        .with_criteria(
            gko::stop::Iteration::build().with_max_iters(dp).on(exec),
            gko::stop::ResidualNormReduction<>::build()
                .with_reduction_factor(accuracy)
                .on(exec))
        .with_preconditioner(bj::build().on(exec))
        .on(exec);
auto solver = solver_gen->generate(gko::give(matrix));
```

Solve system

```
solver->apply(gko::lend(b), gko::lend(x));
}

int main(int argc, char *argv[])
{
    if (argc < 2) {
        std::cerr << "Usage: " << argv[0] << " DISCRETIZATION_POINTS [executor]"
            << std::endl;
        std::exit(-1);
    }

    const int discretization_points = argc >= 2 ? std::atoi(argv[1]) : 100;
    const auto executor_string = argc >= 3 ? argv[2] : "reference";
```


problem:

```
auto correct_u = [](double x) { return x * x * x; };
auto f = [](double x) { return 6 * x; };
auto u0 = correct_u(0);
auto u1 = correct_u(1);
```

matrix

```
std::vector<int> row_ptrs(discretization_points + 1);
std::vector<int> col_idxs(3 * discretization_points - 2);
std::vector<double> values(3 * discretization_points - 2);
```

right hand side

```
std::vector<double> rhs(discretization_points);
```

solution

```
std::vector<double> u(discretization_points, 0.0);
generate_stencil_matrix(discretization_points, row_ptrs.data(),
                        col_idxs.data(), values.data());
```

looking for solution $u = x^3$: $f = 6x$, $u(0) = 0$, $u(1) = 1$

```
generate_rhs(discretization_points, f, u0, u1, rhs.data());

solve_system(executor_string, discretization_points, row_ptrs.data(),
             col_idxs.data(), values.data(), rhs.data(), u.data(), 1e-12);

print_solution(discretization_points, 0, 1, u.data());
std::cout << "The average relative error is "
           << calculate_error(discretization_points, u.data(), correct_u) /
             discretization_points
           << std::endl;
}
```

Results

This is the expected output:

```
0
0.00010798
0.000863838
0.00291545
0.0069107
0.0134975
0.0233236
0.037037
0.0552856
0.0787172
0.10798
0.143721
0.186589
0.237231
0.296296
0.364431
0.442285
0.530504
0.629738
0.740633
0.863838
1
The average relative error is 1.87318e-15
```

Comments about programming and debugging

The plain program

```

/*****<GINKGO LICENSE>*****/
Copyright (c) 2017-2019, the Ginkgo authors
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its
contributors may be used to endorse or promote products derived from
this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*****/

/*****<DESCRIPTION>*****/
This example solves a 1D Poisson equation:

    u : [0, 1] -> R
    u' = f
    u(0) = u0
    u(1) = u1

using a finite difference method on an equidistant grid with 'K' discretization
points ('K' can be controlled with a command line parameter). The discretization
is done via the second order Taylor polynomial:

u(x + h) = u(x) + u'(x)h + 1/2 u''(x)h^2 + O(h^3)
u(x - h) = u(x) - u'(x)h + 1/2 u''(x)h^2 + O(h^3)  / +
-----
-u(x - h) + 2u(x) - u(x + h) = -f(x)h^2 + O(h^3)

For an equidistant grid with K "inner" discretization points x1, ..., xk, and
step size h = 1 / (K + 1), the formula produces a system of linear equations

    2u1 - u2      = -f1 h^2 + u0
-u_(k-1) + 2u_k - u_(k+1) = -f_k h^2,      k = 2, ..., K - 1
-u_(K-1) + 2u_K    = -f_K h^2 + u1

which is then solved using Ginkgo's implementation of the CG method
preconditioned with block-Jacobi. It is also possible to specify on which
executor Ginkgo will solve the system via the command line.
The function 'f' is set to 'f(x) = 6x' (making the solution 'u(x) = x^3'), but
that can be changed in the 'main' function.

The intention of the example is to show how Ginkgo can be integrated into
existing software - the 'generate_stencil_matrix', 'generate_rhs',
'print_solution', 'compute_error' and 'main' function do not reference Ginkgo at
all (i.e. they could have been there before the application developer decided to
use Ginkgo, and the only part where Ginkgo is introduced is inside the
'solve_system' function.
*****/

#include <ginkgo/ginkgo.hpp>
#include <iostream>
#include <map>
#include <string>
#include <vector>

void generate_stencil_matrix(int discretization_points, int *row_ptrs,

```

```

        int *col_idxxs, double *values)
{
    int pos = 0;
    const double coefs[] = {-1, 2, -1};
    row_ptrs[0] = pos;
    for (int i = 0; i < discretization_points; ++i) {
        for (auto ofs : {-1, 0, 1}) {
            if (0 <= i + ofs && i + ofs < discretization_points) {
                values[pos] = coefs[ofs + 1];
                col_idxxs[pos] = i + ofs;
                ++pos;
            }
        }
        row_ptrs[i + 1] = pos;
    }
}

template <typename Closure>
void generate_rhs(int discretization_points, Closure f, double u0, double u1,
                 double *rhs)
{
    const auto h = 1.0 / (discretization_points + 1);
    for (int i = 0; i < discretization_points; ++i) {
        const auto xi = (i + 1) * h;
        rhs[i] = -f(xi) * h * h;
    }
    rhs[0] += u0;
    rhs[discretization_points - 1] += u1;
}

void print_solution(int discretization_points, double u0, double u1,
                   const double *u)
{
    std::cout << u0 << '\n';
    for (int i = 0; i < discretization_points; ++i) {
        std::cout << u[i] << '\n';
    }
    std::cout << u1 << std::endl;
}

template <typename Closure>
double calculate_error(int discretization_points, const double *u,
                      Closure correct_u)
{
    const auto h = 1.0 / (discretization_points + 1);
    auto error = 0.0;
    for (int i = 0; i < discretization_points; ++i) {
        using std::abs;
        const auto xi = (i + 1) * h;
        error += abs(u[i] - correct_u(xi)) / abs(correct_u(xi));
    }
    return error;
}

void solve_system(const std::string &executor_string,
                 unsigned int discretization_points, int *row_ptrs,
                 int *col_idxxs, double *values, double *rhs, double *u,
                 double accuracy)
{
    using vec = gko::matrix::Dense<double>;
    using mtx = gko::matrix::Csr<double, int>;
    using cg = gko::solver::Cg<double>;
    using bj = gko::preconditioner::Jacobi<double, int>;
    using val_array = gko::Array<double>;
    using idx_array = gko::Array<int>;
    const auto &dp = discretization_points;

    const auto omp = gko::OmpExecutor::create();
    std::map<std::string, std::shared_ptr<gko::Executor>> exec_map{
        {"omp", omp},
        {"cuda", gko::CudaExecutor::create(0, omp)},
        {"reference", gko::ReferenceExecutor::create()};
    const auto exec = exec_map.at(executor_string); // throws if not valid
    const auto app_exec = exec_map["omp"];

    auto matrix = mtx::create(exec, gko::dim<2>(dp),
                              val_array::view(app_exec, 3 * dp - 2, values),
                              idx_array::view(app_exec, 3 * dp - 2, col_idxxs),
                              idx_array::view(app_exec, dp + 1, row_ptrs));

    auto b = vec::create(exec, gko::dim<2>(dp, 1),
                          val_array::view(app_exec, dp, rhs), 1);
}

```

```

    auto x = vec::create(app_exec, gko::dim<2>(dp, 1),
        val_array::view(app_exec, dp, u), 1);

    auto solver_gen =
        cg::build()
            .with_criteria(
                gko::stop::Iteration::build().with_max_iters(dp).on(exec),
                gko::stop::ResidualNormReduction<>::build()
                    .with_reduction_factor(accuracy)
                    .on(exec))
            .with_preconditioner(bj::build().on(exec))
            .on(exec);
    auto solver = solver_gen->generate(gko::give(matrix));

    solver->apply(gko::lend(b), gko::lend(x));
}

int main(int argc, char *argv[])
{
    if (argc < 2) {
        std::cerr << "Usage: " << argv[0] << " DISCRETIZATION_POINTS [executor]"
            << std::endl;
        std::exit(-1);
    }

    const int discretization_points = argc >= 2 ? std::atoi(argv[1]) : 100;
    const auto executor_string = argc >= 3 ? argv[2] : "reference";

    auto correct_u = [] (double x) { return x * x * x; };
    auto f = [] (double x) { return 6 * x; };
    auto u0 = correct_u(0);
    auto u1 = correct_u(1);

    std::vector<int> row_ptrs(discretization_points + 1);
    std::vector<int> col_idxs(3 * discretization_points - 2);
    std::vector<double> values(3 * discretization_points - 2);
    std::vector<double> rhs(discretization_points);
    std::vector<double> u(discretization_points, 0.0);

    generate_stencil_matrix(discretization_points, row_ptrs.data(),
        col_idxs.data(), values.data());
    generate_rhs(discretization_points, f, u0, u1, rhs.data());

    solve_system(executor_string, discretization_points, row_ptrs.data(),
        col_idxs.data(), values.data(), rhs.data(), u.data(), 1e-12);

    print_solution(discretization_points, 0, 1, u.data());
    std::cout << "The average relative error is "
        << calculate_error(discretization_points, u.data(), correct_u) /
            discretization_points
        << std::endl;
}

```

Chapter 21

The twentyseven-pt-stencil-solver program

The 27-point stencil example.

This example depends on simple-solver, poisson-solver, three-pt-stencil-solver, nine-pt-stencil-solve.

Introduction

This example solves a 3D Poisson equation:

$$\Omega = (0,1)^3 \setminus [0,1]^3 \setminus \{ \text{(with boundary)} \} \setminus \partial \Omega : \Delta u = f \text{ on } \Omega, u = u_D \text{ on } \partial \Omega$$

using a finite difference method on an equidistant grid with K discretization points (K can be controlled with a command line parameter). The discretization may be done by any order Taylor polynomial. For an equidistant grid with K "inner" discretization points $((x_1, y_1, z_1), \dots, (x_K, y_1, z_1), (x_1, y_2, z_1), \dots, (x_K, y_2, z_1), (x_1, y_1, z_2), \dots, (x_K, y_1, z_2), \dots, (x_K, y_K, z_K))$, step size $(h = 1 / (K + 1))$ and a stencil $(\{R\}^3 \setminus \{0\})$, the formula produces a system of linear equations

$$(\{a,b,c=-1\}^3 \setminus \{0\}) \text{ stencil}(a,b,c) * u_{\{(i+a,j+b,k+c)\}} = -f_k h^2, \text{ on any inner node with a neighborhood of inner nodes}$$

On any node, where neighbor is on the border, the neighbor is replaced with a $(-\text{stencil}(a,b,c) * u_{\{(i+a,j+b,k+c)\}})$ and added to the right hand side vector. For example a node with a neighborhood of only face nodes may look like this

$$(\{a,b,c=-1\}^3 \setminus \{(1,1,0)\}) \text{ stencil}(a,b,c) * u_{\{(i+a,j+b,k+c)\}} = -f_k h^2$$

$$\bullet \{a,b=-1\}^3 \setminus \{(1,1,1)\} \text{ stencil}(a,b,1) * u_{\{(i+a,j+b,k+1)\}}$$

which is then solved using Ginkgo's implementation of the CG method preconditioned with block-Jacobi. It is also possible to specify on which executor Ginkgo will solve the system via the command line. The function f is set to $f(x,y,z) = 6x + 6y + 6z$ (making the solution $(u(x,y,z) = x^3 + y^3 + z^3)$), but that can be changed in the `main` function. Also the stencil values for the core, the faces, the edge and the corners can be changed when passing additional parameters.

The intention of this is to show how generation of stencil values and the right hand side vector changes when increasing the dimension.

About the example

The commented program

```
/ *****<DECSRIPTION>*****
This example solves a 3D Poisson equation:

\Omega = (0,1)^3
\Omega_b = [0,1]^3 (with boundary)
\partial\Omega = \Omega_b \backslash \Omega
u : \Omega_b -> R
u'' = f in \Omega
u = u_D on \partial\Omega

using a finite difference method on an equidistant grid with 'K' discretization
points ('K' can be controlled with a command line parameter). The discretization
may be done by any order Taylor polynomial.
For an equidistant grid with K "inner" discretization points (x1,y1,z1), ...,
(xk,y1,z1), (x1,y2,z1), ..., (xk,yk,z1), (x1,y1,z2), ..., (xk,yk,zk), step size h
= 1 / (K + 1) and a stencil \in \mathbb{R}^{3 \times 3 \times 3}, the formula produces a system of
linear equations

\sum_{a,b,c=-1}^1 stencil(a,b,c) * u_{(i+a,j+b,k+c)} = -f_k h^2, on any inner
node with a neighborhood of inner nodes

On any node, where neighbor is on the border, the neighbor is replaced with a
'-stencil(a,b,c) * u_{(i+a,j+b,k+c)}' and added to the right hand side vector.
For example a node with a neighborhood of only face nodes may look like this

\sum_{a,b,c=-1}^1 stencil(a,b,c) * u_{(i+a,j+b,k+c)} = -f_k h^2 -
\sum_{a,b=-1}^1 stencil(a,b,1) * u_{(i+a,j+b,k+1)}

which is then solved using Ginkgo's implementation of the CG method
preconditioned with block-Jacobi. It is also possible to specify on which
executor Ginkgo will solve the system via the command line.
The function 'f' is set to 'f(x,y,z) = 6x + 6y + 6z' (making the solution
'u(x,y,z) = x^3 + y^3 + z^3'), but that can be changed in the 'main' function.
Also the stencil values for the core, the faces, the edge and the corners can be
changed when passing additional parameters.

The intention of this is to show how generation of stencil values and the right
hand side vector changes when increasing the dimension.
*****<DECSRIPTION>***** /

#include <array>
#include <chrono>
#include <ginkgo/ginkgo.hpp>
#include <iostream>
#include <map>
#include <string>
#include <vector>
```

Can be changed by passing additional parameters when executing the program

```
constexpr double default_alpha = 38 / 6.0;
constexpr double default_beta = -4.0 / 6.0;
constexpr double default_gamma = -1.0 / 6.0;
constexpr double default_delta = -1.0 / 24.0;

/ * Possible alternative values can be for example
* default_alpha = 28.0;
* default_beta = -1.0;
* default_gamma = -1.0;
* default_delta = -1.0;
* /
```

Creates a stencil matrix in CSR format for the given number of discretization points.

```
void generate_stencil_matrix(int dp, int *row_ptrs, int *col_idxs,
                           double *values, double *coefs)
{
    int pos = 0;
    size_t dp_2 = dp * dp;
```

```

row_ptrs[0] = pos;
for (int64_t z = 0; z < dp; ++z) {
    for (int64_t y = 0; y < dp; ++y) {
        for (int64_t x = 0; x < dp; ++x) {
            const auto index = x + dp * (y + dp * z);
            for (int k = -1; k <= 1; ++k) {
                for (int j = -1; j <= 1; ++j) {
                    for (int i = -1; i <= 1; ++i) {
                        const int64_t offset =
                            i + 1 + 3 * (j + 1 + 3 * (k + 1));
                        if ((x + i) >= 0 && (x + i) < dp && (y + j) >= 0 &&
                            (y + j) < dp && (z + k) >= 0 && (z + k) < dp) {
                            values[pos] = coefs[offset];
                            col_idxes[pos] = index + i + dp * (j + dp * k);
                            ++pos;
                        }
                    }
                }
            }
            row_ptrs[index + 1] = pos;
        }
    }
}

```

Generates the RHS vector given f and the boundary conditions.

```

template <typename Closure, typename ClosureT>
void generate_rhs(int dp, Closure f, ClosureT u, double *rhs, double *coefs)
{
    const size_t dp_2 = dp * dp;
    const auto h = 1.0 / (dp + 1.0);
    for (size_t k = 0; k < dp; ++k) {
        const auto zi = (k + 1) * h;
        for (size_t j = 0; j < dp; ++j) {
            const auto yi = (j + 1) * h;
            for (size_t i = 0; i < dp; ++i) {
                const auto xi = (i + 1) * h;
                const auto index = i + dp * (j + dp * k);
                rhs[index] = -f(xi, yi, zi) * h * h;
            }
        }
    }
}

```

This is the iteration over the surface of left and right side of the cube x - ortho to left, right y - ortho to top, bottom z - ortho to front, back

```

for (size_t j = 0; j < dp; ++j) {
    for (size_t k = 0; k < dp; ++k) {
        const auto yi = (j + 1) * h;
        const auto zi = (k + 1) * h;
        const auto index_left = dp * j + dp * dp * k;
        const auto index_right = dp * j + dp * dp * k + (dp - 1);

        for (int b = -1; b <= 1; ++b) {
            for (int c = -1; c <= 1; ++c) {
                rhs[index_left] -= u(0.0, yi + b * h, zi + c * h) *
                    coefs[3 * (b + 1) + 3 * 3 * (c + 1)];
                rhs[index_right] -=
                    u(1.0, yi + b * h, zi + c * h) *
                    coefs[3 * (b + 1) + 3 * 3 * (c + 1) + 2];
            }
        }
    }
}

```

To avoid double counting we have to check if our previous calculations included this case

```

for (size_t i = 0; i < dp; ++i) {
    for (size_t k = 0; k < dp; ++k) {
        const auto xi = (i + 1) * h;
        const auto zi = (k + 1) * h;
        const auto index_top = i + dp * dp * k;
        const auto index_bot = i + dp * dp * k + dp * (dp - 1);

        for (int a = -1; a <= 1; ++a) {

```

```

        if ((i < (dp - 1) || a < 1) && (i > 0 || a > -1)) {
            for (int c = -1; c <= 1; ++c) {
                rhs[index_top] -= u(xi + a * h, 0.0, zi + c * h) *
                    coefs[(a + 1) + 3 * 3 * (c + 1)];
                rhs[index_bot] -=
                    u(xi + a * h, 1.0, zi + c * h) *
                    coefs[(a + 1) + 3 * 3 * (c + 1) + 3 * 2];
            }
        }
    }
}

```

Now every side has to be checked

```

for (size_t i = 0; i < dp; ++i) {
    for (size_t j = 0; j < dp; ++j) {
        const auto xi = (i + 1) * h;
        const auto yi = (j + 1) * h;
        const auto index_front = i + dp * j;
        const auto index_back = i + dp * j + dp * dp * (dp - 1);

        for (int a = -1; a <= 1; ++a) {
            if ((i < (dp - 1) || a < 1) && (i > 0 || a > -1)) {
                for (int b = -1; b <= 1; ++b) {
                    if ((j < (dp - 1) || b < 1) && (j > 0 || b > -1)) {
                        rhs[index_front] -= u(xi + a * h, yi + b * h, 0.0) *
                            coefs[(a + 1) + 3 * (b + 1)];
                        rhs[index_back] -=
                            u(xi + a * h, yi + b * h, 1.0) *
                            coefs[(a + 1) + 3 * (b + 1) + 3 * 3 * 2];
                    }
                }
            }
        }
    }
}

```

Prints the solution u.

```

void print_solution(int dp, const double *u)
{
    for (size_t k = 0; k < dp; ++k) {
        for (size_t j = 0; j < dp; ++j) {
            for (size_t i = 0; i < dp; ++i) {
                std::cout << u[i + dp * (j + dp * k)] << ' ';
            }
            std::cout << '\n';
        }
        std::cout << ":\n";
    }
    std::cout << std::endl;
}

```

Computes the 1-norm of the error given the computed u and the correct solution function correct_u.

```

template <typename Closure>
double calculate_error(int dp, const double *u, Closure correct_u)
{
    using std::abs;
    const auto h = 1.0 / (dp + 1);
    auto error = 0.0;
    for (int k = 0; k < dp; ++k) {
        const auto zi = (k + 1) * h;
        for (int j = 0; j < dp; ++j) {
            const auto yi = (j + 1) * h;
            for (int i = 0; i < dp; ++i) {
                const auto xi = (i + 1) * h;
                error +=
                    abs(u[k * dp * dp + i * dp + j] - correct_u(xi, yi, zi)) /
                    abs(correct_u(xi, yi, zi));
            }
        }
    }
    return error;
}

```



```

}

void solve_system(const std::string &executor_string,
                 unsigned int discretization_points, int *row_ptrs,
                 int *col_idxes, double *values, double *rhs, double *u,
                 double accuracy)
{

```

Some shortcuts

```

using vec = gko::matrix::Dense<double>;
using mtx = gko::matrix::Csr<double, int>;
using cg = gko::solver::Cg<double>;
using bj = gko::preconditioner::Jacobi<double, int>;
using val_array = gko::Array<double>;
using idx_array = gko::Array<int>;
const auto &dp = discretization_points;
const size_t dp_2 = dp * dp;
const size_t dp_3 = dp * dp * dp;

```

Figure out where to run the code

```

const auto omp = gko::OmpExecutor::create();
std::map<std::string, std::shared_ptr<gko::Executor>> exec_map{
    {"omp", omp},
    {"cuda", gko::CudaExecutor::create(0, omp)},
    {"reference", gko::ReferenceExecutor::create()}};

```

executor where Ginkgo will perform the computation

```

const auto exec = exec_map.at(executor_string); // throws if not valid

```

executor where the application initialized the data

```

const auto app_exec = exec_map["omp"];

```

Tell Ginkgo to use the data in our application

Matrix: we have to set the executor of the matrix to the one where we want SpMV's to run (in this case `exec`). When creating array views, we have to specify the executor where the data is (in this case `app_exec`).

If the two do not match, Ginkgo will automatically create a copy of the data on `exec` (however, it will not copy the data back once it is done

- here this is not important since we are not modifying the matrix).

```

auto matrix = mtx::create(
    exec, gko::dim<2>(dp_3),
    val_array::view(app_exec, (3 * dp - 2) * (3 * dp - 2) * (3 * dp - 2),
        values),
    idx_array::view(app_exec, (3 * dp - 2) * (3 * dp - 2) * (3 * dp - 2),
        col_idxes),
    idx_array::view(app_exec, dp_3 + 1, row_ptrs));

```

RHS: similar to matrix

```

auto b = vec::create(exec, gko::dim<2>(dp_3, 1),
    val_array::view(app_exec, dp_3, rhs), 1);

```

Solution: we have to be careful here - if the executors are different, once we compute the solution the array will not be automatically copied back to the original memory locations. Fortunately, whenever `apply` is called on a linear operator (e.g. matrix, solver) the arguments automatically get copied to the executor where the operator is, and copied back once the operation is completed. Thus, in this case, we can just define the solution on `app_exec`, and it will be automatically transferred to/from `exec` if needed.

```
auto x = vec::create(app_exec, gko::dim<2>(dp_3, 1),
                    val_array::view(app_exec, dp_3, u), 1);
```

Generate solver

```
auto solver_gen =
    cg::build()
        .with_criteria(
            gko::stop::Iteration::build().with_max_iters(dp_3).on(exec),
            gko::stop::ResidualNormReduction<>::build()
                .with_reduction_factor(accuracy)
                .on(exec)
            .with_preconditioner(bj::build().on(exec))
            .on(exec);
auto solver = solver_gen->generate(gko::give(matrix));
```

Solve system

```
    solver->apply(gko::lend(b), gko::lend(x));
}

int main(int argc, char *argv[])
{
    if (argc < 2) {
        std::cerr
            << "Usage: " << argv[0] << " DISCRETIZATION_POINTS [executor]"
            << " [stencil_alpha] [stencil_beta] [stencil_gamma] [stencil_delta]"
            << std::endl;
        std::exit(-1);
    }

    const int discretization_points = argc >= 2 ? std::atoi(argv[1]) : 100;
    const auto executor_string = argc >= 3 ? argv[2] : "reference";
    const double alpha_c = argc >= 4 ? std::atof(argv[3]) : default_alpha;
    const double beta_c = argc >= 5 ? std::atof(argv[4]) : default_beta;
    const double gamma_c = argc >= 6 ? std::atof(argv[5]) : default_gamma;
    const double delta_c = argc >= 7 ? std::atof(argv[6]) : default_delta;
```

clang-format off

```
std::array<double,27> coefs{
    delta_c, gamma_c, delta_c,
    gamma_c, beta_c, gamma_c,
    delta_c, gamma_c, delta_c,

    gamma_c, beta_c, gamma_c,
    beta_c, alpha_c, beta_c,
    gamma_c, beta_c, gamma_c,

    delta_c, gamma_c, delta_c,
    gamma_c, beta_c, gamma_c,
    delta_c, gamma_c, delta_c
};
```

clang-format on

```
const auto dp = discretization_points;
const size_t dp_2 = dp * dp;
const size_t dp_3 = dp * dp * dp;
```

problem:

```

auto correct_u = [](double x, double y, double z) {
    return x * x * x + y * y * y + z * z * z;
};
auto f = [](double x, double y, double z) { return 6 * x + 6 * y + 6 * z; };

```

matrix

```

std::vector<int> row_ptrs(dp_3 + 1);
std::vector<int> col_idxes((3 * dp - 2) * (3 * dp - 2) * (3 * dp - 2));
std::vector<double> values((3 * dp - 2) * (3 * dp - 2) * (3 * dp - 2));

```

right hand side

```
std::vector<double> rhs(dp_3);
```

solution

```

std::vector<double> u(dp_3, 0.0);
generate_stencil_matrix(dp, row_ptrs.data(), col_idxes.data(), values.data(),
                        coefs.data());

```

looking for solution $u = x^3$: $f = 6x$, $u(0) = 0$, $u(1) = 1$

```

generate_rhs(dp, f, correct_u, rhs.data(), coefs.data());

auto start_time = std::chrono::steady_clock::now();

solve_system(executor_string, dp, row_ptrs.data(), col_idxes.data(),
             values.data(), rhs.data(), u.data(), 1e-12);

auto stop_time = std::chrono::steady_clock::now();
double runtime_duration =
    std::chrono::duration_cast<std::chrono::nanoseconds>(stop_time -
                                                         start_time)
        .count() *
    1e-6;

print_solution(dp, u.data());
std::cout << "The average relative error is "
           << calculate_error(dp, u.data(), correct_u) / dp_3 << std::endl;

std::cout << "The runtime is " << std::to_string(runtime_duration) << " ms"
           << std::endl;
}

```

Results

The expected output of the relative error at $K=10$ should be

```
The average relative error is 1.87318e-15
```

Comments about programming and debugging

The plain program

```

/*****<GINKGO LICENSE>*****/
Copyright (c) 2017-2019, the Ginkgo authors
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its
contributors may be used to endorse or promote products derived from
this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*****/

/*****<DECSRIPTION>*****/
This example solves a 3D Poisson equation:

\Omega = (0,1)^3
\Omega_b = [0,1]^3 (with boundary)
\partial\Omega = \Omega_b \backslash \Omega
u : \Omega_b \rightarrow R
u'' = f in \Omega
u = u_D on \partial\Omega

using a finite difference method on an equidistant grid with 'K' discretization
points ('K' can be controlled with a command line parameter). The discretization
may be done by any order Taylor polynomial.
For an equidistant grid with K "inner" discretization points (x1,y1,z1), ...,
(xk,y1,z1), (x1,y2,z1), ..., (xk,y2,z1), (x1,y1,z2), ..., (xk,yk,zk), step size h
= 1 / (K + 1) and a stencil \in R^{3 x 3 x 3}, the formula produces a system of
linear equations

\sum_{a,b,c=-1}^1 stencil(a,b,c) * u_{(i+a,j+b,k+c)} = -f_k h^2, on any inner
node with a neighborhood of inner nodes

On any node, where neighbor is on the border, the neighbor is replaced with a
'-stencil(a,b,c) * u_{(i+a,j+b,k+c)}' and added to the right hand side vector.
For example a node with a neighborhood of only face nodes may look like this

\sum_{a,b,c=-1}^1 stencil(a,b,c) * u_{(i+a,j+b,k+c)} = -f_k h^2 -
\sum_{a,b=-1}^1 stencil(a,b,1) * u_{(i+a,j+b,k+1)}

which is then solved using Ginkgo's implementation of the CG method
preconditioned with block-Jacobi. It is also possible to specify on which
executor Ginkgo will solve the system via the command line.
The function 'f' is set to 'f(x,y,z) = 6x + 6y + 6z' (making the solution
'u(x,y,z) = x^3 + y^3 + z^3'), but that can be changed in the 'main' function.
Also the stencil values for the core, the faces, the edge and the corners can be
changed when passing additional parameters.

The intention of this is to show how generation of stencil values and the right
hand side vector changes when increasing the dimension.
*****/

#include <array>
#include <chrono>
#include <ginkgo/ginkgo.hpp>
#include <iostream>
#include <map>
#include <string>
#include <vector>

```

```

constexpr double default_alpha = 38 / 6.0;
constexpr double default_beta = -4.0 / 6.0;
constexpr double default_gamma = -1.0 / 6.0;
constexpr double default_delta = -1.0 / 24.0;

/* Possible alternative values can be for example
 * default_alpha = 28.0;
 * default_beta = -1.0;
 * default_gamma = -1.0;
 * default_delta = -1.0;
 */

void generate_stencil_matrix(int dp, int *row_ptrs, int *col_idx,
                           double *values, double *coefs)
{
    int pos = 0;
    size_t dp_2 = dp * dp;

    row_ptrs[0] = pos;
    for (int64_t z = 0; z < dp; ++z) {
        for (int64_t y = 0; y < dp; ++y) {
            for (int64_t x = 0; x < dp; ++x) {
                const auto index = x + dp * (y + dp * z);
                for (int k = -1; k <= 1; ++k) {
                    for (int j = -1; j <= 1; ++j) {
                        for (int i = -1; i <= 1; ++i) {
                            const int64_t offset =
                                i + 1 + 3 * (j + 1 + 3 * (k + 1));
                            if ((x + i) >= 0 && (x + i) < dp && (y + j) >= 0 &&
                                (y + j) < dp && (z + k) >= 0 && (z + k) < dp) {
                                values[pos] = coefs[offset];
                                col_idx[pos] = index + i + dp * (j + dp * k);
                                ++pos;
                            }
                        }
                    }
                }
                row_ptrs[index + 1] = pos;
            }
        }
    }
}

template <typename Closure, typename ClosureT>
void generate_rhs(int dp, Closure f, ClosureT u, double *rhs, double *coefs)
{
    const size_t dp_2 = dp * dp;
    const auto h = 1.0 / (dp + 1.0);
    for (size_t k = 0; k < dp; ++k) {
        const auto zi = (k + 1) * h;
        for (size_t j = 0; j < dp; ++j) {
            const auto yi = (j + 1) * h;
            for (size_t i = 0; i < dp; ++i) {
                const auto xi = (i + 1) * h;
                const auto index = i + dp * (j + dp * k);
                rhs[index] = -f(xi, yi, zi) * h * h;
            }
        }
    }

    for (size_t j = 0; j < dp; ++j) {
        for (size_t k = 0; k < dp; ++k) {
            const auto yi = (j + 1) * h;
            const auto zi = (k + 1) * h;
            const auto index_left = dp * j + dp * dp * k;
            const auto index_right = dp * j + dp * dp * k + (dp - 1);

            for (int b = -1; b <= 1; ++b) {
                for (int c = -1; c <= 1; ++c) {
                    rhs[index_left] -= u(0.0, yi + b * h, zi + c * h) *
                                       coefs[3 * (b + 1) + 3 * 3 * (c + 1)];
                    rhs[index_right] -=
                        u(1.0, yi + b * h, zi + c * h) *
                        coefs[3 * (b + 1) + 3 * 3 * (c + 1) + 2];
                }
            }
        }
    }

    for (size_t i = 0; i < dp; ++i) {
        for (size_t k = 0; k < dp; ++k) {
            const auto xi = (i + 1) * h;
            const auto zi = (k + 1) * h;
            const auto index_top = i + dp * dp * k;

```

```

    const auto index_bot = i + dp * dp * k + dp * (dp - 1);

    for (int a = -1; a <= 1; ++a) {
        if ((i < (dp - 1) || a < 1) && (i > 0 || a > -1)) {
            for (int c = -1; c <= 1; ++c) {
                rhs[index_top] -= u(xi + a * h, 0.0, zi + c * h) *
                    coefs[(a + 1) + 3 * 3 * (c + 1)];
                rhs[index_bot] -=
                    u(xi + a * h, 1.0, zi + c * h) *
                    coefs[(a + 1) + 3 * 3 * (c + 1) + 3 * 2];
            }
        }
    }
}

for (size_t i = 0; i < dp; ++i) {
    for (size_t j = 0; j < dp; ++j) {
        const auto xi = (i + 1) * h;
        const auto yi = (j + 1) * h;
        const auto index_front = i + dp * j;
        const auto index_back = i + dp * j + dp * dp * (dp - 1);

        for (int a = -1; a <= 1; ++a) {
            if ((i < (dp - 1) || a < 1) && (i > 0 || a > -1)) {
                for (int b = -1; b <= 1; ++b) {
                    if ((j < (dp - 1) || b < 1) && (j > 0 || b > -1)) {
                        rhs[index_front] -= u(xi + a * h, yi + b * h, 0.0) *
                            coefs[(a + 1) + 3 * (b + 1)];
                        rhs[index_back] -=
                            u(xi + a * h, yi + b * h, 1.0) *
                            coefs[(a + 1) + 3 * (b + 1) + 3 * 3 * 2];
                    }
                }
            }
        }
    }
}

void print_solution(int dp, const double *u)
{
    for (size_t k = 0; k < dp; ++k) {
        for (size_t j = 0; j < dp; ++j) {
            for (size_t i = 0; i < dp; ++i) {
                std::cout << u[i + dp * (j + dp * k)] << ' ';
            }
            std::cout << '\n';
        }
        std::cout << ":\n";
    }
    std::cout << std::endl;
}

template <typename Closure>
double calculate_error(int dp, const double *u, Closure correct_u)
{
    using std::abs;
    const auto h = 1.0 / (dp + 1);
    auto error = 0.0;
    for (int k = 0; k < dp; ++k) {
        const auto zi = (k + 1) * h;
        for (int j = 0; j < dp; ++j) {
            const auto yi = (j + 1) * h;
            for (int i = 0; i < dp; ++i) {
                const auto xi = (i + 1) * h;
                error +=
                    abs(u[k * dp * dp + i * dp + j] - correct_u(xi, yi, zi)) /
                    abs(correct_u(xi, yi, zi));
            }
        }
    }
    return error;
}

void solve_system(const std::string &executor_string,
    unsigned int discretization_points, int *row_ptrs,
    int *col_idxs, double *values, double *rhs, double *u,
    double accuracy)
{
    using vec = gko::matrix::Dense<double>;
    using mtx = gko::matrix::Csr<double, int>;
    using cg = gko::solver::Cg<double>;
    using bj = gko::preconditioner::Jacobi<double, int>;

```

```

using val_array = gko::Array<double>;
using idx_array = gko::Array<int>;
const auto &dp = discretization_points;
const size_t dp_2 = dp * dp;
const size_t dp_3 = dp * dp * dp;

const auto omp = gko::OmpExecutor::create();
std::map<std::string, std::shared_ptr<gko::Executor>> exec_map{
    {"omp", omp},
    {"cuda", gko::CudaExecutor::create(0, omp)},
    {"reference", gko::ReferenceExecutor::create()};
const auto exec = exec_map.at(executor_string); // throws if not valid
const auto app_exec = exec_map["omp"];

auto matrix = mtx::create(
    exec, gko::dim<2>(dp_3),
    val_array::view(app_exec, (3 * dp - 2) * (3 * dp - 2) * (3 * dp - 2),
        values),
    idx_array::view(app_exec, (3 * dp - 2) * (3 * dp - 2) * (3 * dp - 2),
        col_idxs),
    idx_array::view(app_exec, dp_3 + 1, row_ptrs));

auto b = vec::create(exec, gko::dim<2>(dp_3, 1),
    val_array::view(app_exec, dp_3, rhs), 1);

auto x = vec::create(app_exec, gko::dim<2>(dp_3, 1),
    val_array::view(app_exec, dp_3, u), 1);

auto solver_gen =
    cg::build()
        .with_criteria(
            gko::stop::Iteration::build().with_max_iters(dp_3).on(exec),
            gko::stop::ResidualNormReduction<>::build()
                .with_reduction_factor(accuracy)
                .on(exec))
        .with_preconditioner(bj::build().on(exec))
        .on(exec);
auto solver = solver_gen->generate(gko::give(matrix));

solver->apply(gko::lend(b), gko::lend(x));
}

int main(int argc, char *argv[])
{
    if (argc < 2) {
        std::cerr
            << "Usage: " << argv[0] << " DISCRETIZATION_POINTS [executor]"
            << " [stencil_alpha] [stencil_beta] [stencil_gamma] [stencil_delta]"
            << std::endl;
        std::exit(-1);
    }

    const int discretization_points = argc >= 2 ? std::atoi(argv[1]) : 100;
    const auto executor_string = argc >= 3 ? argv[2] : "reference";
    const double alpha_c = argc >= 4 ? std::atof(argv[3]) : default_alpha;
    const double beta_c = argc >= 5 ? std::atof(argv[4]) : default_beta;
    const double gamma_c = argc >= 6 ? std::atof(argv[5]) : default_gamma;
    const double delta_c = argc >= 7 ? std::atof(argv[6]) : default_delta;

    std::array<double, 27> coefs{
        delta_c, gamma_c, delta_c,
        gamma_c, beta_c, gamma_c,
        delta_c, gamma_c, delta_c,

        gamma_c, beta_c, gamma_c,
        beta_c, alpha_c, beta_c,
        gamma_c, beta_c, gamma_c,

        delta_c, gamma_c, delta_c,
        gamma_c, beta_c, gamma_c,
        delta_c, gamma_c, delta_c
    };

    const auto dp = discretization_points;
    const size_t dp_2 = dp * dp;
    const size_t dp_3 = dp * dp * dp;

    auto correct_u = [](double x, double y, double z) {
        return x * x * x + y * y * y + z * z * z;
    };
    auto f = [](double x, double y, double z) { return 6 * x + 6 * y + 6 * z; };

    std::vector<int> row_ptrs(dp_3 + 1);
    std::vector<int> col_idxs((3 * dp - 2) * (3 * dp - 2) * (3 * dp - 2));
    std::vector<double> values((3 * dp - 2) * (3 * dp - 2) * (3 * dp - 2));
    std::vector<double> rhs(dp_3);

```

```
std::vector<double> u(dp_3, 0.0);

generate_stencil_matrix(dp, row_ptrs.data(), col_idxes.data(), values.data(),
                      coefs.data());
generate_rhs(dp, f, correct_u, rhs.data(), coefs.data());

auto start_time = std::chrono::steady_clock::now();

solve_system(executor_string, dp, row_ptrs.data(), col_idxes.data(),
            values.data(), rhs.data(), u.data(), 1e-12);

auto stop_time = std::chrono::steady_clock::now();
double runtime_duration =
    std::chrono::duration_cast<std::chrono::nanoseconds>(stop_time -
                                                         start_time)
        .count() *
    1e-6;

print_solution(dp, u.data());
std::cout << "The average relative error is "
    << calculate_error(dp, u.data(), correct_u) / dp_3 << std::endl;

std::cout << "The runtime is " << std::to_string(runtime_duration) << " ms"
    << std::endl;
}
```


Chapter 22

Module Documentation

22.1 CUDA Executor

A module dedicated to the implementation and usage of the CUDA executor in Ginkgo.

Classes

- class [gko::CudaExecutor](#)

This is the [Executor](#) subclass which represents the CUDA device.

22.1.1 Detailed Description

A module dedicated to the implementation and usage of the CUDA executor in Ginkgo.

22.2 Executors

A module dedicated to the implementation and usage of the executors in Ginkgo.

Modules

- [CUDA Executor](#)
A module dedicated to the implementation and usage of the CUDA executor in Ginkgo.
- [OpenMP Executor](#)
A module dedicated to the implementation and usage of the OpenMP executor in Ginkgo.
- [Reference Executor](#)
A module dedicated to the implementation and usage of the Reference executor in Ginkgo.

Classes

- class [gko::Operation](#)
Operations can be used to define functionalities whose implementations differ among devices.
- class [gko::Executor](#)
The first step in using the Ginkgo library consists of creating an executor.
- class [gko::executor_deleter< T >](#)
This is a deleter that uses an executor's `free` method to deallocate the data.
- class [gko::OmpExecutor](#)
This is the [Executor](#) subclass which represents the OpenMP device (typically CPU).
- class [gko::ReferenceExecutor](#)
This is a specialization of the [OmpExecutor](#), which runs the reference implementations of the kernels used for debugging purposes.
- class [gko::CudaExecutor](#)
This is the [Executor](#) subclass which represents the CUDA device.

Macros

- `#define GKO_REGISTER_OPERATION(_name, _kernel)`
Binds a set of device-specific kernels to an Operation.

22.2.1 Detailed Description

A module dedicated to the implementation and usage of the executors in Ginkgo.

Below, we provide a brief introduction to executors in Ginkgo, how they have been implemented, how to best make use of them and how to add new executors.

22.2.2 Executors in Ginkgo.

The first step in using the Ginkgo library consists of creating an executor. Executors are used to specify the location for the data of linear algebra objects, and to determine where the operations will be executed. Ginkgo currently supports three different executor types:

- [OpenMP Executor](#) specifies that the data should be stored and the associated operations executed on an OpenMP-supporting device (e.g. host CPU);
- [CUDA Executor](#) specifies that the data should be stored and the operations executed on the NVIDIA GPU accelerator;
- [Reference Executor](#) executes a non-optimized reference implementation, which can be used to debug the library.

22.2.3 Macro Definition Documentation

22.2.3.1 GKO_REGISTER_OPERATION

```
#define GKO_REGISTER_OPERATION(
    _name,
    _kernel )
```

Binds a set of device-specific kernels to an Operation.

It also defines a helper function which creates the associated operation. Any input arguments passed to the helper function are forwarded to the kernel when the operation is executed.

The kernels used to bind the operation are searched in `kernels::DEV_TYPE` namespace, where `DEV_TYPE` is replaced by `omp`, `cuda` and `reference`.

Parameters

<code>_name</code>	operation name
<code>_kernel</code>	kernel which will be bound to the operation

Example

```
{c++}
// define the omp, cuda and reference kernels which will be bound to the
// operation
namespace kernels {
namespace omp {
void my_kernel(int x) {
    // omp code
}
}
namespace cuda {
void my_kernel(int x) {
    // cuda code
}
}
namespace reference {
```

```
void my_kernel(int x) {  
    // reference code  
}  
  
// Bind the kernels to the operation  
GKO_REGISTER_OPERATION(my_op, my_kernel);  
  
int main() {  
    // create executors  
    auto omp = OmpExecutor::create();  
    auto cuda = CudaExecutor::create(omp, 0);  
    auto ref = ReferenceExecutor::create();  
  
    // create the operation  
    auto op = make_my_op(5); // x = 5  
  
    omp->run(op); // run omp kernel  
    cuda->run(op); // run cuda kernel  
    ref->run(op); // run reference kernel  
}
```

22.3 Factorizations

A module dedicated to the implementation and usage of the Factorizations in Ginkgo.

Namespaces

- [gko::factorization](#)

The Factorization namespace.

Classes

- class [gko::factorization::Parllu< ValueType, IndexType >](#)

ParlLU is an incomplete LU factorization which is computed in parallel.

22.3.1 Detailed Description

A module dedicated to the implementation and usage of the Factorizations in Ginkgo.

22.4 Linear Operators

A module dedicated to the implementation and usage of the Linear operators in Ginkgo.

Modules

- [Factorizations](#)
A module dedicated to the implementation and usage of the Factorizations in Ginkgo.
- [SpMV employing different Matrix formats](#)
A module dedicated to the implementation and usage of the various Matrix Formats in Ginkgo.
- [Preconditioners](#)
A module dedicated to the implementation and usage of the Preconditioners in Ginkgo.
- [Solvers](#)
A module dedicated to the implementation and usage of the Solvers in Ginkgo.

Classes

- class [gko::Combination< ValueType >](#)
*The [Combination](#) class can be used to construct a linear combination of multiple linear operators $c_1 * op_1 + c_2 * op_2 + \dots$*
- class [gko::Composition< ValueType >](#)
*The [Composition](#) class can be used to compose linear operators op_1, op_2, \dots, op_n and obtain the operator $op_1 * op_2 * \dots$*
- class [gko::LinOpFactory](#)
A [LinOpFactory](#) represents a higher order mapping which transforms one linear operator into another.
- class [gko::ReadableFromMatrixData< ValueType, IndexType >](#)
A [LinOp](#) implementing this interface can read its data from a [matrix_data](#) structure.
- class [gko::WritableToMatrixData< ValueType, IndexType >](#)
A [LinOp](#) implementing this interface can write its data to a [matrix_data](#) structure.
- class [gko::Preconditionable](#)
A [LinOp](#) implementing this interface can be preconditioned.
- class [gko::EnableLinOp< ConcreteLinOp, PolymorphicBase >](#)
The [EnableLinOp](#) mixin can be used to provide sensible default implementations of the majority of the [LinOp](#) and [PolymorphicObject](#) interface.
- class [gko::matrix::Coo< ValueType, IndexType >](#)
COO stores a matrix in the coordinate matrix format.
- class [gko::matrix::Csr< ValueType, IndexType >](#)
CSR is a matrix format which stores only the nonzero coefficients by compressing each row of the matrix (compressed sparse row format).
- class [gko::matrix::Dense< ValueType >](#)
[Dense](#) is a matrix format which explicitly stores all values of the matrix.
- class [gko::matrix::Ell< ValueType, IndexType >](#)
ELL is a matrix format where stride with explicit zeros is used such that all rows have the same number of stored elements.
- class [gko::matrix::Hybrid< ValueType, IndexType >](#)
HYBRID is a matrix format which splits the matrix into ELLPACK and COO format.
- class [gko::matrix::Identity< ValueType >](#)
This class is a utility which efficiently implements the identity matrix (a linear operator which maps each vector to itself).
- class [gko::matrix::IdentityFactory< ValueType >](#)

This factory is a utility which can be used to generate [Identity](#) operators.

- class [gko::matrix::Sellp< ValueType, IndexType >](#)
SELL-P is a matrix format similar to ELL format.
- struct [gko::preconditioner::block_interleaved_storage_scheme< IndexType >](#)
Defines the parameters of the interleaved block storage scheme used by block-Jacobi blocks.
- class [gko::preconditioner::Jacobi< ValueType, IndexType >](#)
A block-Jacobi preconditioner is a block-diagonal linear operator, obtained by inverting the diagonal blocks of the source operator.
- class [gko::solver::Bicgstab< ValueType >](#)
BiCGSTAB or the Bi-Conjugate Gradient-Stabilized is a Krylov subspace solver.
- class [gko::solver::Cg< ValueType >](#)
CG or the conjugate gradient method is an iterative type Krylov subspace method which is suitable for symmetric positive definite methods.
- class [gko::solver::Cgs< ValueType >](#)
CGS or the conjugate gradient square method is an iterative type Krylov subspace method which is suitable for general systems.
- class [gko::solver::Fcg< ValueType >](#)
FCG or the flexible conjugate gradient method is an iterative type Krylov subspace method which is suitable for symmetric positive definite methods.
- class [gko::solver::Gmres< ValueType >](#)
GMRES or the generalized minimal residual method is an iterative type Krylov subspace method which is suitable for nonsymmetric linear systems.

Macros

- #define [GKO_CREATE_FACTORY_PARAMETERS](#)(_parameters_name, _factory_name)
This Macro will generate a new type containing the parameters for the factory `_factory_name`.
- #define [GKO_ENABLE_LIN_OP_FACTORY](#)(_lin_op, _parameters_name, _factory_name)
This macro will generate a default implementation of a `LinOpFactory` for the `LinOp` subclass it is defined in.
- #define [GKO_ENABLE_BUILD_METHOD](#)(_factory_name)
Defines a build method for the factory, simplifying its construction by removing the repetitive typing of factory's name.
- #define [GKO_FACTORY_PARAMETER](#)(_name, ...)
Creates a factory parameter in the factory parameters structure.

Typedefs

- template<typename ConcreteFactory , typename ConcreteLinOp , typename ParametersType , typename PolymorphicBase = [LinOpFactory](#)>
using [gko::EnableDefaultLinOpFactory](#) = [EnableDefaultFactory](#)< ConcreteFactory, ConcreteLinOp, ParametersType, PolymorphicBase >
This is an alias for the [EnableDefaultFactory](#) mixin, which correctly sets the template parameters to enable a subclass of [LinOpFactory](#).

22.4.1 Detailed Description

A module dedicated to the implementation and usage of the Linear operators in Ginkgo.

Below we elaborate on one of the most important concepts of Ginkgo, the linear operator. The linear operator (`LinOp`) is a base class for all linear algebra objects in Ginkgo. The main benefit of having a single base class for the entire collection of linear algebra objects (as opposed to having separate hierarchies for matrices, solvers and preconditioners) is the generality it provides.

22.4.2 Advantages of this approach and usage

A common interface often allows for writing more generic code. If a user's routine requires only operations provided by the LinOp interface, the same code can be used for any kind of linear operators, independent of whether these are matrices, solvers or preconditioners. This feature is also extensively used in Ginkgo itself. For example, a preconditioner used inside a Krylov solver is a LinOp. This allows the user to supply a wide variety of preconditioners: either the ones which were designed to be used in this scenario (like ILU or block-Jacobi), a user-supplied matrix which is known to be a good preconditioner for the specific problem, or even another solver (e.g., if constructing a flexible GMRES solver).

For example, a matrix free implementation would require the user to provide an apply implementation and instead of passing the generated matrix to the solver, they would have to provide their apply implementation for all the executors needed and no other code needs to be changed. See [The custom-matrix-format program](#) example for more details.

22.4.3 Linear operator as a concept

The linear operator (LinOp) is a base class for all linear algebra objects in Ginkgo. The main benefit of having a single base class for the entire collection of linear algebra objects (as opposed to having separate hierarchies for matrices, solvers and preconditioners) is the generality it provides.

First, since all subclasses provide a common interface, the library users are exposed to a smaller set of routines. For example, a matrix-vector product, a preconditioner application, or even a system solve are just different terms given to the operation of applying a certain linear operator to a vector. As such, Ginkgo uses the same routine name, `LinOp::apply()` for each of these operations, where the actual operation performed depends on the type of linear operator involved in the operation.

Second, a common interface often allows for writing more generic code. If a user's routine requires only operations provided by the LinOp interface, the same code can be used for any kind of linear operators, independent of whether these are matrices, solvers or preconditioners. This feature is also extensively used in Ginkgo itself. For example, a preconditioner used inside a Krylov solver is a LinOp. This allows the user to supply a wide variety of preconditioners: either the ones which were designed to be used in this scenario (like ILU or block-Jacobi), a user-supplied matrix which is known to be a good preconditioner for the specific problem, or even another solver (e.g., if constructing a flexible GMRES solver).

A key observation for providing a unified interface for matrices, solvers, and preconditioners is that the most common operation performed on all of them can be expressed as an application of a linear operator to a vector:

- the sparse matrix-vector product with a matrix A is a linear operator application $y = Ax$;
- the application of a preconditioner is a linear operator application $y = M^{-1}x$, where M is an approximation of the original system matrix A (thus a preconditioner represents an "approximate inverse" operator M^{-1}).
- the system solve $Ax = b$ can be viewed as linear operator application $x = A^{-1}b$ (it goes without saying that the implementation of linear system solves does not follow this conceptual idea), so a linear system solver can be viewed as a representation of the operator A^{-1} .

Finally, direct manipulation of LinOp objects is rarely required in simple scenarios. As an illustrative example, one could construct a fixed-point iteration routine $x_{k+1} = Lx_k + b$ as follows:

```
std::unique_ptr<matrix::Dense<>> calculate_fixed_point(
    int iters, const LinOp *L, const matrix::Dense<> *x0
    const matrix::Dense<> *b)
{
    auto x = gko::clone(x0);
    auto tmp = gko::clone(x0);
    auto one = Dense<>::create(L->get_executor(), {1.0,});
    for (int i = 0; i < iters; ++i) {
        L->apply(gko::lend(tmp), gko::lend(x));
        x->add_scaled(gko::lend(one), gko::lend(b));
        tmp->copy_from(gko::lend(x));
    }
    return x;
}
```


Here, if L is a matrix, `LinOp::apply()` refers to the matrix vector product, and `L->apply(a, b)` computes $b = L \cdot a$. `x->add_scaled(one.get(), b.get())` is the axpy vector update $x := x + b$.

The interesting part of this example is the `apply()` routine at line 4 of the function body. Since this routine is part of the `LinOp` base class, the fixed-point iteration routine can calculate a fixed point not only for matrices, but for any type of linear operator.

Linear Operators

22.4.4 Macro Definition Documentation

22.4.4.1 GKO_CREATE_FACTORY_PARAMETERS

```
#define GKO_CREATE_FACTORY_PARAMETERS (
    _parameters_name,
    _factory_name )
```

Value:

```
public:
    class _factory_name;
    struct _parameters_name##_type
        : ::gko::enable_parameters_type<_parameters_name##_type,
            _factory_name>
```

This Macro will generate a new type containing the parameters for the factory `_factory_name`.

For more details, see [GKO_ENABLE_LIN_OP_FACTORY\(\)](#). It is required to use this macro **before** calling the macro [GKO_ENABLE_LIN_OP_FACTORY\(\)](#). It is also required to use the same names for all parameters between both macros.

Parameters

<code>_parameters_name</code>	name of the parameters member in the class
<code>_factory_name</code>	name of the generated factory type

Referenced by `gko::solver::Gmres< ValueType >::get_krylov_dim()`, `gko::preconditioner::Jacobi< ValueType, IndexType >::get_num_stored_elements()`, `gko::solver::Cgs< ValueType >::get_preconditioner()`, `gko::solver::Cg< ValueType >::get_preconditioner()`, `gko::solver::Bicgstab< ValueType >::get_preconditioner()`, `gko::solver::Fcg< ValueType >::get_preconditioner()`, and `gko::solver::lr< ValueType >::get_solver()`.

22.4.4.2 GKO_ENABLE_BUILD_METHOD

```
#define GKO_ENABLE_BUILD_METHOD (
    _factory_name )
```

Value:

```

static auto build()->decltype(_factory_name::create())
{
    return _factory_name::create();
}
static_assert(true,
    "This assert is used to counter the false positive extra "
    "semi-colon warnings")

```

Defines a build method for the factory, simplifying its construction by removing the repetitive typing of factory's name.

Parameters

<code>_factory_name</code>	the factory for which to define the method
----------------------------	--

Referenced by `gko::solver::Gmres< ValueType >::get_krylov_dim()`, `gko::preconditioner::Jacobi< ValueType, IndexType >::get_num_stored_elements()`, `gko::solver::Cgs< ValueType >::get_preconditioner()`, `gko::solver::Cg< ValueType >::get_preconditioner()`, `gko::solver::Bicgstab< ValueType >::get_preconditioner()`, `gko::solver::Fcg< ValueType >::get_preconditioner()`, and `gko::solver::lr< ValueType >::get_solver()`.

22.4.4.3 GKO_ENABLE_LIN_OP_FACTORY

```

#define GKO_ENABLE_LIN_OP_FACTORY(
    _lin_op,
    _parameters_name,
    _factory_name )

```

Value:

```

public:
    const _parameters_name##_type &get_##_parameters_name() const
    {
        return _parameters_name##_;
    }

    class _factory_name
    : public ::gko::EnableDefaultLinOpFactory<_factory_name,
        _lin_op, \
            _parameters_name##_type> {
        friend class ::gko::EnablePolymorphicObject<_factory_name,
            ::gko::LinOpFactory>;
        friend class ::gko::enable_parameters_type<_parameters_name##_type,
            _factory_name>;
        using ::gko::EnableDefaultLinOpFactory<
            _factory_name, _lin_op,
            _parameters_name##_type>::EnableDefaultLinOpFactory;
    };
    friend ::gko::EnableDefaultLinOpFactory<_factory_name,
        _lin_op, \
            _parameters_name##_type>;

private:
    _parameters_name##_type _parameters_name##_;

public:
    static_assert(true,
        "This assert is used to counter the false positive extra "
        "semi-colon warnings")

```

This macro will generate a default implementation of a `LinOpFactory` for the `LinOp` subclass it is defined in.

It is required to first call the macro `GKO_CREATE_FACTORY_PARAMETERS()` before this one in order to instantiate the parameters type first.

The list of parameters for the factory should be defined in a code block after the macro definition, and should contain a list of `GKO_FACTORY_PARAMETER` declarations. The class should provide a constructor with signature `_lin_op(const _factory_name *, std::shared_ptr<const LinOp>)` which the factory will use a callback to construct the object.

A minimal example of a linear operator is the following:

```
{c++}
struct MyLinOp : public EnableLinOp<MyLinOp> {
    GKO_ENABLE_LIN_OP_FACTORY(MyLinOp, my_parameters, Factory) {
        // a factory parameter named "my_value", of type int and default
        // value of 5
        int GKO_FACTORY_PARAMETER(my_value, 5);
    };
    // constructor needed by EnableLinOp
    explicit MyLinOp(std::shared_ptr<const Executor> exec) {
        : EnableLinOp<MyLinOp>(exec) {}
    }
    // constructor needed by the factory
    explicit MyLinOp(const Factory *factory,
                    std::shared_ptr<const LinOp> matrix)
        : EnableLinOp<MyLinOp>(factory->get_executor()), matrix->get_size(),
        // store factory's parameters locally
        my_parameters_{factory->get_parameters()} {
    }
    {
        int value = my_parameters_.my_value;
        // do something with value
    }
}
```

MyLinOp can then be created as follows:

```
{c++}
auto exec = gko::ReferenceExecutor::create();
// create a factory with default 'my_value' parameter
auto fact = MyLinOp::build().on(exec);
// create a operator using the factory:
auto my_op = fact->generate(gko::matrix::Identity::create(exec, 2));
std::cout << my_op->get_my_parameters().my_value; // prints 5

// create a factory with custom 'my_value' parameter
auto fact = MyLinOp::build().with_my_value(0).on(exec);
// create a operator using the factory:
auto my_op = fact->generate(gko::matrix::Identity::create(exec, 2));
std::cout << my_op->get_my_parameters().my_value; // prints 0
```

Note

It is possible to combine both the `#GKO_CREATE_FACTORY_PARAMETER()` macro with this one in a unique macro for class **templates** (not with regular classes). Splitting this into two distinct macros allows to use them in all contexts. See <https://stackoverflow.com/q/50202718/9385966> for more details.

Parameters

<code>_lin_op</code>	concrete operator for which the factory is to be created [CRTP parameter]
<code>_parameters_name</code>	name of the parameters member in the class (its type is <code><_parameters_name>_type</code> , the protected member's name is <code><_parameters_name>_</code> , and the public getter's name is <code>get_<_parameters_name>()</code>)
<code>_factory_name</code>	name of the generated factory type

Referenced by `gko::solver::Gmres< ValueType >::get_krylov_dim()`, `gko::preconditioner::Jacobi< ValueType, IndexType >::get_num_stored_elements()`, `gko::solver::Cgs< ValueType >::get_preconditioner()`, `gko::solver::Cg< ValueType >::get_preconditioner()`, `gko::solver::Bicgstab< ValueType >::get_preconditioner()`, `gko::solver::Fcg< ValueType >::get_preconditioner()`, and `gko::solver::Ir< ValueType >::get_solver()`.

22.4.4.4 GKO_FACTORY_PARAMETER

```
#define GKO_FACTORY_PARAMETER(
    _name,
    ... )
```

Value:

```
mutable _name{__VA_ARGS__};

template <typename... Args>
auto with_##_name(Args &&... _value)
    const->const ::gko::xstd::decay_t<decltype(*this)> &
{
    using type = decltype(this->_name);
    this->_name = type{std::forward<Args>(_value)...};
    return *this;
}
static_assert(true,
    "This assert is used to counter the false positive extra " \
    "semi-colon warnings")
```

Creates a factory parameter in the factory parameters structure.

Parameters

<code>_name</code>	name of the parameter
<code>VA_ARGS</code>	default value of the parameter

See also

[GKO_ENABLE_LIN_OP_FACTORY](#) for more details, and usage example

Referenced by `gko::solver::Gmres< ValueType >::get_krylov_dim()`, `gko::preconditioner::Jacobi< ValueType, IndexType >::get_num_stored_elements()`, `gko::solver::Cgs< ValueType >::get_preconditioner()`, `gko::solver::Cg< ValueType >::get_preconditioner()`, `gko::solver::Bicgstab< ValueType >::get_preconditioner()`, `gko::solver::Fcg< ValueType >::get_preconditioner()`, and `gko::solver::Ir< ValueType >::get_solver()`.

22.4.5 Typedef Documentation

22.4.5.1 EnableDefaultLinOpFactory

```
template<typename ConcreteFactory , typename ConcreteLinOp , typename ParametersType , typename
PolymorphicBase = LinOpFactory>
using gko::EnableDefaultLinOpFactory = typedef EnableDefaultFactory<ConcreteFactory, Concrete↵
LinOp, ParametersType, PolymorphicBase>
```

This is an alias for the [EnableDefaultFactory](#) mixin, which correctly sets the template parameters to enable a subclass of [LinOpFactory](#).

Template Parameters

<i>ConcreteFactory</i>	the concrete factory which is being implemented [CRTP parmeter]
<i>ConcreteLinOp</i>	the concrete LinOp type which this factory produces, needs to have a constructor which takes a const ConcreteFactory *, and an std::shared_ptr<const LinOp> as parameters.
<i>ParametersType</i>	a subclass of enable_parameters_type template which defines all of the parameters of the factory
<i>PolymorphicBase</i>	parent of ConcreteFactory in the polymorphic hierarchy, has to be a subclass of LinOpFactory

22.5 Logging

A module dedicated to the implementation and usage of the Logging in Ginkgo.

Namespaces

- [gko::log](#)

The logger namespace .

Classes

- class [gko::log::Convergence< ValueType >](#)

Convergence is a Logger which logs data strictly from the `criterion_check_completed` event.

- class [gko::log::Stream< ValueType >](#)

Stream is a Logger which logs every event to a stream.

22.5.1 Detailed Description

A module dedicated to the implementation and usage of the Logging in Ginkgo.

The Logger class represents a simple Logger object.

It comprises all masks and events internally. Every new logging event addition should be done here. The Logger class also provides a default implementation for most events which do nothing, therefore it is not an obligation to change all classes which derive from Logger, although it is good practice. The logger class is built using event masks to control which events should be logged, and which should not.

22.6 SpMV employing different Matrix formats

A module dedicated to the implementation and usage of the various Matrix Formats in Ginkgo.

Classes

- class `gko::matrix::Coo< ValueType, IndexType >`
COO stores a matrix in the coordinate matrix format.
- class `gko::matrix::Csr< ValueType, IndexType >`
CSR is a matrix format which stores only the nonzero coefficients by compressing each row of the matrix (compressed sparse row format).
- class `gko::matrix::Dense< ValueType >`
Dense is a matrix format which explicitly stores all values of the matrix.
- class `gko::matrix::Ell< ValueType, IndexType >`
ELL is a matrix format where stride with explicit zeros is used such that all rows have the same number of stored elements.
- class `gko::matrix::Hybrid< ValueType, IndexType >`
HYBRID is a matrix format which splits the matrix into ELLPACK and COO format.
- class `gko::matrix::Identity< ValueType >`
This class is a utility which efficiently implements the identity matrix (a linear operator which maps each vector to itself).
- class `gko::matrix::IdentityFactory< ValueType >`
This factory is a utility which can be used to generate Identity operators.
- class `gko::matrix::Sellp< ValueType, IndexType >`
SELL-P is a matrix format similar to ELL format.

Functions

- `template<typename Matrix, typename... TArgs>`
`std::unique_ptr< Matrix > gko::initialize (size_type stride, std::initializer_list< typename Matrix::value_type > vals, std::shared_ptr< const Executor > exec, TArgs &&... create_args)`
Creates and initializes a column-vector.
- `template<typename Matrix, typename... TArgs>`
`std::unique_ptr< Matrix > gko::initialize (std::initializer_list< typename Matrix::value_type > vals, std::shared_ptr< const Executor > exec, TArgs &&... create_args)`
Creates and initializes a column-vector.
- `template<typename Matrix, typename... TArgs>`
`std::unique_ptr< Matrix > gko::initialize (size_type stride, std::initializer_list< std::initializer_list< typename Matrix::value_type > > vals, std::shared_ptr< const Executor > exec, TArgs &&... create_args)`
Creates and initializes a matrix.
- `template<typename Matrix, typename... TArgs>`
`std::unique_ptr< Matrix > gko::initialize (std::initializer_list< std::initializer_list< typename Matrix::value_type > > vals, std::shared_ptr< const Executor > exec, TArgs &&... create_args)`
Creates and initializes a matrix.

22.6.1 Detailed Description

A module dedicated to the implementation and usage of the various Matrix Formats in Ginkgo.

22.6.2 Function Documentation

22.6.2.1 initialize() [1/4]

```
template<typename Matrix , typename... TArgs>
std::unique_ptr<Matrix> gko::initialize (
    size_type stride,
    std::initializer_list< typename Matrix::value_type > vals,
    std::shared_ptr< const Executor > exec,
    TArgs &&... create_args )
```

Creates and initializes a column-vector.

This function first creates a temporary Dense matrix, fills it with passed in values, and then converts the matrix to the requested type.

Template Parameters

<i>Matrix</i>	matrix type to initialize (Dense has to implement the <code>ConvertibleTo<Matrix></code> interface)
<i>TArgs</i>	argument types for <code>Matrix::create</code> method (not including the implied Executor as the first argument)

Parameters

<i>stride</i>	row stride for the temporary Dense matrix
<i>vals</i>	values used to initialize the vector
<i>exec</i>	Executor associated to the vector
<i>create_args</i>	additional arguments passed to <code>Matrix::create</code> , not including the Executor , which is passed as the first argument

References `gko::matrix::Dense< ValueType >::at()`.

```
518 {
519     using dense = matrix::Dense<typename Matrix::value_type>;
520     size_type num_rows = vals.size();
521     auto tmp = dense::create(exec->get_master(), dim<2>{num_rows, 1}, stride);
522     size_type idx = 0;
523     for (const auto &elem : vals) {
524         tmp->at(idx) = elem;
525         ++idx;
526     }
527     auto mtx = Matrix::create(exec, std::forward<TArgs>(create_args)...);
528     tmp->move_to(mtx.get());
529     return mtx;
530 }
```

22.6.2.2 initialize() [2/4]

```
template<typename Matrix , typename... TArgs>
std::unique_ptr<Matrix> gko::initialize (
```



```

std::initializer_list< typename Matrix::value_type > vals,
std::shared_ptr< const Executor > exec,
TArgs &&... create_args )

```

Creates and initializes a column-vector.

This function first creates a temporary Dense matrix, fills it with passed in values, and then converts the matrix to the requested type. The stride of the intermediate Dense matrix is set to 1.

Template Parameters

<i>Matrix</i>	matrix type to initialize (Dense has to implement the ConvertibleTo<Matrix> interface)
<i>TArgs</i>	argument types for Matrix::create method (not including the implied Executor as the first argument)

Parameters

<i>vals</i>	values used to initialize the vector
<i>exec</i>	Executor associated to the vector
<i>create_args</i>	additional arguments passed to Matrix::create, not including the Executor , which is passed as the first argument

```

557 {
558     return initialize<Matrix>(1, vals, std::move(exec),
559                             std::forward<TArgs>(create_args)...);
560 }

```

22.6.2.3 initialize() [3/4]

```

template<typename Matrix , typename... TArgs>
std::unique_ptr<Matrix> gko::initialize (
    size_type stride,
    std::initializer_list< std::initializer_list< typename Matrix::value_type >>
    vals,
    std::shared_ptr< const Executor > exec,
    TArgs &&... create_args )

```

Creates and initializes a matrix.

This function first creates a temporary Dense matrix, fills it with passed in values, and then converts the matrix to the requested type.

Template Parameters

<i>Matrix</i>	matrix type to initialize (Dense has to implement the ConvertibleTo<Matrix> interface)
<i>TArgs</i>	argument types for Matrix::create method (not including the implied Executor as the first argument)

Parameters

<i>stride</i>	row stride for the temporary Dense matrix
---------------	---

Parameters

<i>vals</i>	values used to initialize the matrix
<i>exec</i>	Executor associated to the matrix
<i>create_args</i>	additional arguments passed to <code>Matrix::create</code> , not including the Executor , which is passed as the first argument

References `gko::matrix::Dense< ValueType >::at()`.

```

590 {
591     using dense = matrix::Dense<typename Matrix::value_type>;
592     size_type num_rows = vals.size();
593     size_type num_cols = num_rows > 0 ? begin(vals)->size() : 1;
594     auto tmp =
595         dense::create(exec->get_master(), dim<2>(num_rows, num_cols), stride);
596     size_type ridx = 0;
597     for (const auto &row : vals) {
598         size_type cidx = 0;
599         for (const auto &elem : row) {
600             tmp->at(ridx, cidx) = elem;
601             ++cidx;
602         }
603         ++ridx;
604     }
605     auto mtx = Matrix::create(exec, std::forward<TArgs>(create_args)...);
606     tmp->move_to(mtx.get());
607     return mtx;
608 }
```

22.6.2.4 initialize() [4/4]

```

template<typename Matrix , typename... TArgs>
std::unique_ptr<Matrix> gko::initialize (
    std::initializer_list< std::initializer_list< typename Matrix::value_type >>
    vals,
    std::shared_ptr< const Executor > exec,
    TArgs &&... create_args )
```

Creates and initializes a matrix.

This function first creates a temporary Dense matrix, fills it with passed in values, and then converts the matrix to the requested type. The stride of the intermediate Dense matrix is set to the number of columns of the initializer list.

Template Parameters

<i>Matrix</i>	matrix type to initialize (Dense has to implement the <code>ConvertibleTo<Matrix></code> interface)
<i>TArgs</i>	argument types for <code>Matrix::create</code> method (not including the implied Executor as the first argument)

Parameters

<i>vals</i>	values used to initialize the matrix
<i>exec</i>	Executor associated to the matrix
<i>create_args</i>	additional arguments passed to <code>Matrix::create</code> , not including the Executor , which is passed as the first argument

```
638 {  
639     return initialize<Matrix>(vals.size() > 0 ? begin(vals)->size() : 0, vals,  
640                             std::move(exec),  
641                             std::forward<TArgs>(create_args)...);  
642 }
```

22.7 OpenMP Executor

A module dedicated to the implementation and usage of the OpenMP executor in Ginkgo.

Classes

- class [gko::OmpExecutor](#)

This is the [Executor](#) subclass which represents the OpenMP device (typically CPU).

22.7.1 Detailed Description

A module dedicated to the implementation and usage of the OpenMP executor in Ginkgo.

22.8 Preconditioners

A module dedicated to the implementation and usage of the Preconditioners in Ginkgo.

Namespaces

- [gko::preconditioner](#)
The Preconditioner namespace.

Classes

- class [gko::Preconditionable](#)
A LinOp implementing this interface can be preconditioned.
- struct [gko::preconditioner::block_interleaved_storage_scheme< IndexType >](#)
Defines the parameters of the interleaved block storage scheme used by block-Jacobi blocks.
- class [gko::preconditioner::Jacobi< ValueType, IndexType >](#)
A block-Jacobi preconditioner is a block-diagonal linear operator, obtained by inverting the diagonal blocks of the source operator.

22.8.1 Detailed Description

A module dedicated to the implementation and usage of the Preconditioners in Ginkgo.

22.9 Reference Executor

A module dedicated to the implementation and usage of the Reference executor in Ginkgo.

Classes

- class [gko::ReferenceExecutor](#)

This is a specialization of the [OmpExecutor](#), which runs the reference implementations of the kernels used for debugging purposes.

22.9.1 Detailed Description

A module dedicated to the implementation and usage of the Reference executor in Ginkgo.

22.10 Solvers

A module dedicated to the implementation and usage of the Solvers in Ginkgo.

Namespaces

- [gko::solver](#)
The ginkgo Solve namespace.

Classes

- class [gko::solver::Bicgstab< ValueType >](#)
BiCGSTAB or the Bi-Conjugate Gradient-Stabilized is a Krylov subspace solver.
- class [gko::solver::Cg< ValueType >](#)
CG or the conjugate gradient method is an iterative type Krylov subspace method which is suitable for symmetric positive definite methods.
- class [gko::solver::Cgs< ValueType >](#)
CGS or the conjugate gradient square method is an iterative type Krylov subspace method which is suitable for general systems.
- class [gko::solver::Fcg< ValueType >](#)
FCG or the flexible conjugate gradient method is an iterative type Krylov subspace method which is suitable for symmetric positive definite methods.
- class [gko::solver::Gmres< ValueType >](#)
GMRES or the generalized minimal residual method is an iterative type Krylov subspace method which is suitable for nonsymmetric linear systems.

22.10.1 Detailed Description

A module dedicated to the implementation and usage of the Solvers in Ginkgo.

22.11 Stopping criteria

A module dedicated to the implementation and usage of the Stopping Criteria in Ginkgo.

Namespaces

- [gko::stop](#)

The Stopping criterion namespace.

Classes

- class [gko::stop::Combined](#)

The [Combined](#) class is used to combine multiple criterions together through an OR operation.

- class [gko::stop::Iteration](#)

The [Iteration](#) class is a stopping criterion which stops the iteration process after a preset number of iterations.

- class [gko::stop::ResidualNormReduction< ValueType >](#)

The [ResidualNormReduction](#) class is a stopping criterion which stops the iteration process when the relative residual norm is below a certain threshold.

- class [gko::stopping_status](#)

This class is used to keep track of the stopping status of one vector.

- class [gko::stop::Time](#)

The [Time](#) class is a stopping criterion which stops the iteration process after a certain amount of time has passed.

Macros

- `#define GKO_ENABLE_CRITERION_FACTORY(_criterion, _parameters_name, _factory_name)`

This macro will generate a default implementation of a [CriterionFactory](#) for the [Criterion](#) subclass it is defined in.

Functions

- `template<typename FactoryContainer >
std::shared_ptr< const CriterionFactory > gko::stop::combine (FactoryContainer &&factories)`

Combines multiple criterion factories into a single combined criterion factory.

22.11.1 Detailed Description

A module dedicated to the implementation and usage of the Stopping Criteria in Ginkgo.

22.11.2 Macro Definition Documentation

22.11.2.1 GKO_ENABLE_CRITERION_FACTORY

```
#define GKO_ENABLE_CRITERION_FACTORY(
    _criterion,
    _parameters_name,
    _factory_name )
```

Value:

```
public:
    const _parameters_name##_type &get_##_parameters_name() const
    {
        return _parameters_name##_;
    }

    class _factory_name
    : public ::gko::stop::EnableDefaultCriterionFactory<
        _factory_name, _criterion, _parameters_name##_type> {
    friend class ::gko::EnablePolymorphicObject<
        _factory_name, ::gko::stop::CriterionFactory>;
    friend class ::gko::enable_parameters_type<_parameters_name##_type,
        _factory_name>;
    using ::gko::stop::EnableDefaultCriterionFactory<
        _factory_name, _criterion,
        _parameters_name##_type>::EnableDefaultCriterionFactory;
    };
    friend ::gko::stop::EnableDefaultCriterionFactory<
        _factory_name, _criterion, _parameters_name##_type>;

private:
    _parameters_name##_type _parameters_name##_;

public:
    static_assert(true,
        "This assert is used to counter the false positive extra " \
        "semi-colon warnings")
```

This macro will generate a default implementation of a CriterionFactory for the Criterion subclass it is defined in.

This macro is very similar to the macro #ENABLE_LIN_OP_FACTORY(). A more detailed description of the use of these type of macros can be found there.

Parameters

<code>_criterion</code>	concrete operator for which the factory is to be created [CRTP parameter]
<code>_parameters_name</code>	name of the parameters member in the class (its type is <code><_parameters_name>_type</code> , the protected member's name is <code><_parameters_name>_</code> , and the public getter's name is <code>get_<_parameters_name>()</code>)
<code>_factory_name</code>	name of the generated factory type

22.11.3 Function Documentation

22.11.3.1 combine()

```
template<typename FactoryContainer >
std::shared_ptr<const CriterionFactory> gko::stop::combine (
    FactoryContainer && factories )
```

Combines multiple criterion factories into a single combined criterion factory.

This function treats a singleton container as a special case and avoids creating an additional object and just returns the input factory.

Template Parameters

<i>FactoryContainer</i>	a random access container type
-------------------------	--------------------------------

Parameters

<i>factories</i>	a list of factories to combined
------------------	---------------------------------

Returns

a combined criterion factory if the input contains multiple factories or the input factory if the input contains only one factory

Referenced by `gko::solver::Gmres< ValueType >::get_krylov_dim()`, `gko::solver::Cgs< ValueType >::get_preconditioner()`, `gko::solver::Cg< ValueType >::get_preconditioner()`, `gko::solver::Fcg< ValueType >::get_preconditioner()`, `gko::solver::Bicgstab< ValueType >::get_preconditioner()`, and `gko::solver::lr< ValueType >::get_solver()`.

```

117 {
118     switch (factories.size()) {
119     case 0:
120         GKO_NOT_SUPPORTED(nullptr);
121         return nullptr;
122     case 1:
123         return factories[0];
124     default:
125         auto exec = factories[0]->get_executor();
126         return Combined::build()
127             .with_criteria(std::forward<FactoryContainer>(factories))
128             .on(exec);
129     }
130 }
```

Chapter 23

Namespace Documentation

23.1 gko Namespace Reference

The Ginkgo namespace.

Namespaces

- [accessor](#)
The accessor namespace.
- [factorization](#)
The Factorization namespace.
- [log](#)
The logger namespace .
- [matrix](#)
The matrix namespace.
- [name_demangling](#)
The name demangling namespace.
- [preconditioner](#)
The Preconditioner namespace.
- [solver](#)
The ginkgo Solve namespace.
- [stop](#)
The Stopping criterion namespace.
- [syn](#)
The Synthesizer namespace.
- [xstd](#)
The namespace for functionalities after C++11 standard.

Classes

- class [AbstractFactory](#)
The [AbstractFactory](#) is a generic interface template that enables easy implementation of the abstract factory design pattern.
- class [AllocationError](#)
[AllocationError](#) is thrown if a memory allocation fails.
- class [Array](#)
An [Array](#) is a container which encapsulates fixed-sized arrays, stored on the [Executor](#) tied to the [Array](#).
- class [Combination](#)
*The [Combination](#) class can be used to construct a linear combination of multiple linear operators $c_1 * op_1 + c_2 * op_2 + \dots$*
- class [Composition](#)
*The [Composition](#) class can be used to compose linear operators op_1, op_2, \dots, op_n and obtain the operator $op_1 * op_2 * \dots$*
- class [ConvertibleTo](#)
[ConvertibleTo](#) interface is used to mark that the implementer can be converted to the object of ResultType.
- class [copy_back_deleter](#)
A [copy_back_deleter](#) is a type of deleter that copies the data to an internally referenced object before performing the deletion.
- class [CublasError](#)
[CublasError](#) is thrown when a cuBLAS routine throws a non-zero error code.
- class [CudaError](#)
[CudaError](#) is thrown when a CUDA routine throws a non-zero error code.
- class [CudaExecutor](#)
This is the [Executor](#) subclass which represents the CUDA device.
- class [CusparsError](#)
[CusparsError](#) is thrown when a cuSPARSE routine throws a non-zero error code.
- struct [default_converter](#)
Used to convert objects of type S to objects of type R using `static_cast`.
- struct [dim](#)
A type representing the dimensions of a multidimensional object.
- class [DimensionMismatch](#)
[DimensionMismatch](#) is thrown if an operation is being applied to LinOps of incompatible size.
- struct [enable_parameters_type](#)
The [enable_parameters_type](#) mixin is used to create a base implementation of the factory parameters structure.
- class [EnableAbstractPolymorphicObject](#)
This mixin inherits from (a subclass of) [PolymorphicObject](#) and provides a base implementation of a new abstract object.
- class [EnableCreateMethod](#)
This mixin implements a static `create()` method on `ConcreteType` that dynamically allocates the memory, uses the passed-in arguments to construct the object, and returns an `std::unique_ptr` to such an object.
- class [EnableDefaultFactory](#)
This mixin provides a default implementation of a concrete factory.
- class [EnableLinOp](#)
The [EnableLinOp](#) mixin can be used to provide sensible default implementations of the majority of the LinOp and [PolymorphicObject](#) interface.
- class [EnablePolymorphicAssignment](#)
This mixin is used to enable a default [PolymorphicObject::copy_from\(\)](#) implementation for objects that have implemented conversions between them.
- class [EnablePolymorphicObject](#)
This mixin inherits from (a subclass of) [PolymorphicObject](#) and provides a base implementation of a new concrete polymorphic object.

- class [Error](#)
The [Error](#) class is used to report exceptional behaviour in library functions.
- class [Executor](#)
The first step in using the Ginkgo library consists of creating an executor.
- class [executor_deleter](#)
This is a deleter that uses an executor's `free` method to deallocate the data.
- class [KernelNotFound](#)
[KernelNotFound](#) is thrown if Ginkgo cannot find a kernel which satisfies the criteria imposed by the input arguments.
- class [LinOpFactory](#)
A [LinOpFactory](#) represents a higher order mapping which transforms one linear operator into another.
- struct [matrix_data](#)
This structure is used as an intermediate data type to store a sparse matrix.
- class [NotCompiled](#)
[NotCompiled](#) is thrown when attempting to call an operation which is a part of a module that was not compiled on the system.
- class [NotImplemented](#)
[NotImplemented](#) is thrown in case an operation has not yet been implemented (but will be implemented in the future).
- class [NotSupported](#)
[NotSupported](#) is thrown in case it is not possible to perform the requested operation on the given object type.
- class [null_deleter](#)
This is a deleter that does not delete the object.
- class [OmpExecutor](#)
This is the [Executor](#) subclass which represents the OpenMP device (typically CPU).
- class [Operation](#)
Operations can be used to define functionalities whose implementations differ among devices.
- class [OutOfBoundsError](#)
[OutOfBoundsError](#) is thrown if a memory access is detected to be out-of-bounds.
- class [PolymorphicObject](#)
A [PolymorphicObject](#) is the abstract base for all "heavy" objects in Ginkgo that behave polymorphically.
- class [precision_reduction](#)
This class is used to encode storage precisions of low precision algorithms.
- class [Preconditionable](#)
A [LinOp](#) implementing this interface can be preconditioned.
- class [range](#)
A range is a multidimensional view of the memory.
- class [ReadableFromMatrixData](#)
A [LinOp](#) implementing this interface can read its data from a [matrix_data](#) structure.
- class [ReferenceExecutor](#)
This is a specialization of the [OmpExecutor](#), which runs the reference implementations of the kernels used for debugging purposes.
- struct [span](#)
A span is a lightweight structure used to create sub-ranges from other ranges.
- class [stopping_status](#)
This class is used to keep track of the stopping status of one vector.
- class [StreamError](#)
[StreamError](#) is thrown if accessing a stream failed.
- class [temporary_clone](#)
A [temporary_clone](#) is a special smart pointer-like object that is designed to hold an object temporarily copied to another executor.
- class [Transposable](#)
Linear operators which support transposition should implement the [Transposable](#) interface.

- struct [version](#)
This structure is used to represent versions of various Ginkgo modules.
- class [version_info](#)
Ginkgo uses version numbers to label new features and to communicate backward compatibility guarantees:
- class [WritableToMatrixData](#)
A LinOp implementing this interface can write its data to a [matrix_data](#) structure.

Typedefs

- `template<typename ConcreteFactory, typename ConcreteLinOp, typename ParametersType, typename PolymorphicBase = LinOpFactory>`
`using EnableDefaultLinOpFactory = EnableDefaultFactory< ConcreteFactory, ConcreteLinOp, ParametersType, PolymorphicBase >`
This is an alias for the [EnableDefaultFactory](#) mixin, which correctly sets the template parameters to enable a subclass of [LinOpFactory](#).
- `template<typename T >`
`using remove_complex = typename detail::remove_complex_impl< T >::type`
Obtains a real counterpart of a `std::complex` type, and leaves the type unchanged if it is not a complex type.
- `template<typename T >`
`using is_complex_s = detail::is_complex_impl< T >`
Allows to check if `T` is a complex value during compile time by accessing the `value` attribute of this struct.
- `template<typename T >`
`using reduce_precision = typename detail::reduce_precision_impl< T >::type`
Obtains the next type in the hierarchy with lower precision than `T`.
- `template<typename T >`
`using increase_precision = typename detail::increase_precision_impl< T >::type`
Obtains the next type in the hierarchy with higher precision than `T`.
- `template<typename T, size_type Limit = sizeof(uint16) * byte_size>`
`using truncate_type = xstd::conditional_t< detail::type_size_impl< T >::value >= 2 * Limit, typename detail::truncate_type_impl< T >::type, T >`
Truncates the type by half (by dropping bits), but ensures that it is at least `Limit` bits wide.
- `using size_type = std::size_t`
Integral type used for allocation quantities.
- `using int8 = std::int8_t`
8-bit signed integral type.
- `using int16 = std::int16_t`
16-bit signed integral type.
- `using int32 = std::int32_t`
32-bit signed integral type.
- `using int64 = std::int64_t`
64-bit signed integral type.
- `using uint8 = std::uint8_t`
8-bit unsigned integral type.
- `using uint16 = std::uint16_t`
16-bit unsigned integral type.
- `using uint32 = std::uint32_t`
32-bit unsigned integral type.
- `using uint64 = std::uint64_t`
64-bit unsigned integral type.
- `using float16 = half`
Half precision floating point type.

- using `float32` = float
Single precision floating point type.
- using `float64` = double
Double precision floating point type.
- using `full_precision` = double
The most precise floating-point type.
- using `default_precision` = double
Precision used if no precision is explicitly specified.

Enumerations

- enum `layout_type` { `layout_type::array`, `layout_type::coordinate` }
Specifies the layout type when writing data in matrix market format.

Functions

- template<size_type Dimensionality, typename DimensionType >
constexpr bool `operator!=` (const `dim`< Dimensionality, DimensionType > &x, const `dim`< Dimensionality, DimensionType > &y)
Checks if two dim objects are different.
- template<typename DimensionType >
constexpr `dim`< 2, DimensionType > `transpose` (const `dim`< 2, DimensionType > &dimensions) noexcept
Returns a `dim`<2> object with its dimensions swapped.
- template<typename T >
constexpr bool `is_complex` ()
Checks if T is a complex type.
- template<typename T >
constexpr `reduce_precision`< T > `round_down` (T val)
Reduces the precision of the input parameter.
- template<typename T >
constexpr `increase_precision`< T > `round_up` (T val)
Increases the precision of the input parameter.
- constexpr `int64 ceildiv` (`int64` num, `int64` den)
Performs integer division with rounding up.
- template<typename T >
constexpr T `zero` ()
Returns the additive identity for T.
- template<typename T >
constexpr T `zero` (const T &)
Returns the additive identity for T.
- template<typename T >
constexpr T `one` ()
Returns the multiplicative identity for T.
- template<typename T >
constexpr T `one` (const T &)
Returns the multiplicative identity for T.
- template<typename T >
constexpr T `abs` (const T &x)
Returns the absolute value of the object.
- template<typename T >
constexpr T `max` (const T &x, const T &y)

- Returns the larger of the arguments.*

 - `template<typename T >`
`constexpr T min (const T &x, const T &y)`

Returns the smaller of the arguments.

 - `template<typename T >`
`constexpr T real (const T &x)`

Returns the real part of the object.

 - `template<typename T >`
`constexpr T imag (const T &)`

Returns the imaginary part of the object.

 - `template<typename T >`
`T conj (const T &x)`

Returns the conjugate of an object.

 - `template<typename T >`
`constexpr auto squared_norm (const T &x) -> decltype(real(conj(x) *x))`

Returns the squared norm of the object.

 - `template<typename T >`
`constexpr uint32 get_significant_bit (const T &n, uint32 hint=0u) noexcept`

Returns the position of the most significant bit of the number.

 - `template<typename T >`
`constexpr T get_superior_power (const T &base, const T &limit, const T &hint=T{1}) noexcept`

*Returns the smallest power of *base* not smaller than *limit*.*

 - `template<typename T >`
`xstd::enable_if_t< is_complex_s< T >::value, bool > isfinite (const T &value)`

Checks if a given component of a complex value is positive infinity, negative infinity, or NaN.

 - `template<typename ValueType = default_precision, typename IndexType = int32>`
`matrix_data< ValueType, IndexType > read_raw (std::istream &is)`

Reads a matrix stored in matrix market format from an input stream.

 - `template<typename ValueType , typename IndexType >`
`void write_raw (std::ostream &os, const matrix_data< ValueType, IndexType > &data, layout_type layout=layout_type::array)`

Writes a [matrix_data](#) structure to a stream in matrix market format.

 - `template<typename MatrixType , typename StreamType , typename... MatrixArgs>`
`std::unique_ptr< MatrixType > read (StreamType &&is, MatrixArgs &&... args)`

Reads a matrix stored in matrix market format from an input stream.

 - `template<typename MatrixType , typename StreamType >`
`void write (StreamType &&os, MatrixType *matrix, layout_type layout=layout_type::array)`

Reads a matrix stored in matrix market format from an input stream.

 - `template<typename R , typename T >`
`std::unique_ptr< R, std::function< void(R *)> > copy_and_convert_to (std::shared_ptr< const Executor > exec, T *obj)`

Converts the object to R and places it on [Executor](#) exec.

 - `template<typename R , typename T >`
`std::unique_ptr< const R, std::function< void(const R *)> > copy_and_convert_to (std::shared_ptr< const Executor > exec, const T *obj)`

Converts the object to R and places it on [Executor](#) exec.

 - `constexpr bool operator== (precision_reduction x, precision_reduction y) noexcept`

Checks if two [precision_reduction](#) encodings are equal.

 - `constexpr bool operator!= (precision_reduction x, precision_reduction y) noexcept`

Checks if two [precision_reduction](#) encodings are different.

 - `template<typename Pointer >`
`detail::cloned_type< Pointer > clone (const Pointer &p)`

Creates a unique clone of the object pointed to by p.

- `template<typename Pointer >`
`detail::cloned_type< Pointer > clone (std::shared_ptr< const Executor > exec, const Pointer &p)`
Creates a unique clone of the object pointed to by `p` on `Executor` `exec`.
- `template<typename OwningPointer >`
`detail::shared_type< OwningPointer > share (OwningPointer &&p)`
Marks the object pointed to by `p` as shared.
- `template<typename OwningPointer >`
`std::remove_reference< OwningPointer >::type && give (OwningPointer &&p)`
Marks that the object pointed to by `p` can be given to the callee.
- `template<typename Pointer >`
`std::enable_if< detail::have_ownership< Pointer >, detail::pointee< Pointer > * >::type lend (const Pointer &p)`
Returns a non-owning (plain) pointer to the object pointed to by `p`.
- `template<typename Pointer >`
`std::enable_if<!detail::have_ownership< Pointer >, detail::pointee< Pointer > * >::type lend (const Pointer &p)`
Returns a non-owning (plain) pointer to the object pointed to by `p`.
- `template<typename T , typename U >`
`std::decay< T >::type * as (U *obj)`
Performs polymorphic type conversion.
- `template<typename T , typename U >`
`const std::decay< T >::type * as (const U *obj)`
Performs polymorphic type conversion.
- `template<typename T >`
`temporary_clone< T > make_temporary_clone (std::shared_ptr< const Executor > exec, T *ptr)`
Creates a `temporary_clone`.
- `std::ostream & operator<< (std::ostream &os, const version &ver)`
Prints version information to a stream.
- `std::ostream & operator<< (std::ostream &os, const version_info &ver_info)`
Prints library version information in human-readable format to a stream.
- `template<typename Matrix , typename... TArgs>`
`std::unique_ptr< Matrix > initialize (size_type stride, std::initializer_list< typename Matrix::value_type > vals, std::shared_ptr< const Executor > exec, TArgs &&... create_args)`
Creates and initializes a column-vector.
- `template<typename Matrix , typename... TArgs>`
`std::unique_ptr< Matrix > initialize (std::initializer_list< typename Matrix::value_type > vals, std::shared_ptr< const Executor > exec, TArgs &&... create_args)`
Creates and initializes a column-vector.
- `template<typename Matrix , typename... TArgs>`
`std::unique_ptr< Matrix > initialize (size_type stride, std::initializer_list< std::initializer_list< typename Matrix::value_type >> vals, std::shared_ptr< const Executor > exec, TArgs &&... create_args)`
Creates and initializes a matrix.
- `template<typename Matrix , typename... TArgs>`
`std::unique_ptr< Matrix > initialize (std::initializer_list< std::initializer_list< typename Matrix::value_type >> vals, std::shared_ptr< const Executor > exec, TArgs &&... create_args)`
Creates and initializes a matrix.
- `bool operator== (const stopping_status &x, const stopping_status &y) noexcept`
Checks if two stopping statuses are equivalent.
- `bool operator!= (const stopping_status &x, const stopping_status &y) noexcept`
Checks if two stopping statuses are different.

Variables

- constexpr `size_type byte_size` = CHAR_BIT
Number of bits in a byte.

23.1.1 Detailed Description

The Ginkgo namespace.

23.1.2 Typedef Documentation

23.1.2.1 `is_complex_s`

```
template<typename T >
using gko::is_complex_s = typedef detail::is_complex_impl<T>
```

Allows to check if T is a complex value during compile time by accessing the `value` attribute of this struct.

If `value` is `true`, T is a complex type, if it is `false`, T is not a complex type.

Template Parameters

<code>T</code>	type to check
----------------	---------------

23.1.3 Enumeration Type Documentation

23.1.3.1 `layout_type`

```
enum gko::layout_type [strong]
```

Specifies the layout type when writing data in matrix market format.

Enumerator

<code>array</code>	The matrix should be written as dense matrix in column-major order.
<code>coordinate</code>	The matrix should be written as a sparse matrix in coordinate format.

```
67                                     {
71     array,
75     coordinate
76 };
```

23.1.4 Function Documentation

23.1.4.1 `abs()`

```
template<typename T >
constexpr T gko::abs (
    const T & x ) [inline]
```

Returns the absolute value of the object.

Template Parameters

<i>T</i>	the type of the object
----------	------------------------

Parameters

<i>x</i>	the object
----------	------------

Returns

```
x >= zero<T>() ? x : -x;
```

```
362 {
363     return x >= zero<T>() ? x : -x;
364 }
```

23.1.4.2 `as()` [1/2]

```
template<typename T , typename U >
std::decay<T>::type* gko::as (
    U * obj ) [inline]
```

Performs polymorphic type conversion.

Template Parameters

<i>T</i>	requested result type
<i>U</i>	static type of the passed object

Parameters

<i>obj</i>	the object which should be converted
------------	--------------------------------------

Returns

If successful, returns a pointer to the subtype, otherwise throws [NotSupported](#).

```

286 {
287     if (auto p = dynamic_cast<typename std::decay<T>::type *>(obj)) {
288         return p;
289     } else {
290         throw NotSupported(__FILE__, __LINE__, __func__, typeid(obj).name());
291     }
292 }
```

23.1.4.3 as() [2/2]

```

template<typename T , typename U >
const std::decay<T>::type* gko::as (
    const U * obj ) [inline]
```

Performs polymorphic type conversion.

This is the constant version of the function.

Template Parameters

<i>T</i>	requested result type
<i>U</i>	static type of the passed object

Parameters

<i>obj</i>	the object which should be converted
------------	--------------------------------------

Returns

If successful, returns a pointer to the subtype, otherwise throws [NotSupported](#).

```

309 {
310     if (auto p = dynamic_cast<const typename std::decay<T>::type *>(obj)) {
311         return p;
312     } else {
313         throw NotSupported(__FILE__, __LINE__, __func__, typeid(obj).name());
314     }
315 }
```

23.1.4.4 ceildiv()

```

constexpr int64 gko::ceildiv (
    int64 num,
    int64 den ) [inline]
```

Performs integer division with rounding up.

Parameters

<i>num</i>	numerator
<i>den</i>	denominator

Returns

returns the ceiled quotient.

Referenced by `gko::matrix::Sellp< ValueType, IndexType >::col_at()`, and `gko::preconditioner::block_interleaved_storage_scheme< index_type >::compute_storage_space()`.

```
292 {
293     return (num + den - 1) / den;
294 }
```

23.1.4.5 clone() [1/2]

```
template<typename Pointer >
detail::cloned_type<Pointer> gko::clone (
    const Pointer & p ) [inline]
```

Creates a unique clone of the object pointed to by `p`.

The pointee (i.e. `*p`) needs to have a clone method that returns a `std::unique_ptr` in order for this method to work.

Template Parameters

<i>Pointer</i>	type of pointer to the object (plain or smart pointer)
----------------	--

Parameters

<i>p</i>	a pointer to the object
----------	-------------------------

Note

The difference between this function and directly calling `LinOp::clone()` is that this one preserves the static type of the object.

Referenced by `gko::temporary_clone< T >::temporary_clone()`.

```
157 {
158     static_assert(detail::is_clonable<detail::pointee<Pointer>>>(),
159                 "Object is not clonable");
160     return detail::cloned_type<Pointer>(
161         static_cast<typename std::remove_cv<detail::pointee<Pointer>>::type*>(
162             p->clone().release()));
163 }
```

23.1.4.6 clone() [2/2]

```
template<typename Pointer >
detail::cloned_type<Pointer> gko::clone (
    std::shared_ptr< const Executor > exec,
    const Pointer & p ) [inline]
```

Creates a unique clone of the object pointed to by *p* on [Executor](#) *exec*.

The pointee (i.e. **p*) needs to have a clone method that takes an executor and returns a `std::unique_ptr` in order for this method to work.

Template Parameters

<i>Pointer</i>	type of pointer to the object (plain or smart pointer)
----------------	--

Parameters

<i>exec</i>	the executor where the cloned object should be stored
<i>p</i>	a pointer to the object

Note

The difference between this function and directly calling `LinOp::clone()` is that this one preserves the static type of the object.

```
184 {
185     static_assert(detail::is_clonable_to<detail::pointee<Pointer>>>(),
186                 "Object is not clonable");
187     return detail::cloned_type<Pointer>(
188         static_cast<typename std::remove_cv<detail::pointee<Pointer>>::type *>(
189             p->clone(std::move(exec)).release()));
190 }
```

23.1.4.7 conj()

```
template<typename T >
T gko::conj (
    const T & x ) [inline]
```

Returns the conjugate of an object.

Parameters

<i>x</i>	the number to conjugate
----------	-------------------------

Returns

conjugate of the object (by default, the object itself)

Referenced by `gko::matrix_data< ValueType, IndexType >::ensure_row_major_order()`, and `squared_norm()`.

```

451 {
452     return x;
453 }

```

23.1.4.8 copy_and_convert_to() [1/2]

```

template<typename R , typename T >
std::unique_ptr<R, std::function<void(R *)> > gko::copy_and_convert_to (
    std::shared_ptr< const Executor > exec,
    T * obj )

```

Converts the object to R and places it on [Executor](#) exec.

If the object is already of the requested type and on the requested executor, the copy and conversion is avoided and a reference to the original object is returned instead.

Template Parameters

<i>R</i>	the type to which the object should be converted
<i>T</i>	the type of the input object

Parameters

<i>exec</i>	the executor where the result should be placed
<i>obj</i>	the object that should be converted

Returns

a unique pointer (with dynamically bound deleter) to the converted object

```

451 {
452     return detail::copy_and_convert_to_impl<R>(std::move(exec), obj);
453 }

```

23.1.4.9 copy_and_convert_to() [2/2]

```

template<typename R , typename T >
std::unique_ptr<const R, std::function<void(const R *)> > gko::copy_and_convert_to (
    std::shared_ptr< const Executor > exec,
    const T * obj )

```

Converts the object to R and places it on [Executor](#) exec.

If the object is already of the requested type and on the requested executor, the copy and conversion is avoided and a reference to the original object is returned instead.

Template Parameters

<i>R</i>	the type to which the object should be converted
<i>T</i>	the type of the input object

Parameters

<i>exec</i>	the executor where the result should be placed
<i>obj</i>	the object that should be converted

Returns

a unique pointer (with dynamically bound deleter) to the converted object

Note

This is a version of the function which adds the `const` qualifier to the result if the input had the same qualifier.

```

465 {
466     return detail::copy_and_convert_to_impl<const R>(std::move(exec), obj);
467 }
```

23.1.4.10 get_significant_bit()

```

template<typename T >
constexpr uint32 gko::get_significant_bit (
    const T & n,
    uint32 hint = 0u ) [inline], [noexcept]
```

Returns the position of the most significant bit of the number.

This is the same as the rounded down base-2 logarithm of the number.

Template Parameters

<i>T</i>	a numeric type supporting bit shift and comparison
----------	--

Parameters

<i>n</i>	a number
<i>hint</i>	a lower bound for the position of the significant bit

Returns

maximum of `hint` and the significant bit position of `n`

Referenced by `gko::preconditioner::Jacobi< ValueType, IndexType >::get_num_stored_elements()`.


```

489 {
490     return (T{1} << (hint + 1)) > n ? hint : get\_significant\_bit(n, hint + 1u);
491 }

```

23.1.4.11 `get_superior_power()`

```

template<typename T >
constexpr T gko::get_superior_power (
    const T & base,
    const T & limit,
    const T & hint = T{1} ) [inline], [noexcept]

```

Returns the smallest power of `base` not smaller than `limit`.

Template Parameters

<i>T</i>	a numeric type supporting multiplication and comparison
----------	---

Parameters

<i>base</i>	the base of the power to be returned
<i>limit</i>	the lower limit on the size of the power returned
<i>hint</i>	a lower bound on the result, has to be a power of <code>base</code>

Returns

the smallest power of `base` not smaller than `limit`

Referenced by `gko::preconditioner::Jacobi< ValueType, IndexType >::get_num_stored_elements()`.

```

507                                     {1}) noexcept
508 {
509     return hint >= limit ? hint : get\_superior\_power(base, limit, hint * base);
510 }

```

23.1.4.12 `give()`

```

template<typename OwningPointer >
std::remove_reference<OwningPointer>::type&& gko::give (
    OwningPointer && p ) [inline]

```

Marks that the object pointed to by `p` can be given to the callee.

Effectively calls `std::move(p)`.

Template Parameters

<i>OwningPointer</i>	type of pointer with ownership to the object (has to be a smart pointer)
----------------------	--

Parameters

<i>p</i>	a pointer to the object
----------	-------------------------

Note

The original pointer *p* becomes invalid after this call.

```

229 {
230     static_assert(detail::have_ownership<OwningPointer>(),
231                   "OwningPointer does not have ownership of the object");
232     return std::move(p);
233 }
```

23.1.4.13 imag()

```

template<typename T >
constexpr T gko::imag (
    const T & ) [inline]
```

Returns the imaginary part of the object.

Template Parameters

<i>T</i>	type of the object
----------	--------------------

Parameters

<i>x</i>	the object
----------	------------

Returns

imaginary part of the object (by default, [zero<T>\(\)](#))

```

437 {
438     return zero<T>();
439 }
```

23.1.4.14 is_complex()

```

template<typename T >
constexpr bool gko::is_complex ( ) [inline]
```

Checks if *T* is a complex type.

Template Parameters

<i>T</i>	type to check
----------	---------------

Returns

true if *T* is a complex type, false otherwise

```

116 {
117     return detail::is_complex_impl<T>::value;
118 }
```

23.1.4.15 isfinite()

```

template<typename T >
xstd::enable_if_t<is_complex_s<T>::value, bool> gko::isfinite (
    const T & value ) [inline]
```

Checks if a given component of a complex value is positive infinity, negative infinity, or NaN.

Template Parameters

<i>T</i>	complex type of the value to check
----------	------------------------------------

Parameters

<i>value</i>	complex value to check
--------------	------------------------

returns true if a component of value is either positive or negative infinity or NaN. Otherwise false

```

601 {
602     return isfinite(value.real()) || isfinite(value.imag());
603 }
```

23.1.4.16 lend() [1/2]

```

template<typename Pointer >
std::enable_if<detail::have_ownership<Pointer>>, detail::pointee<Pointer> *>::type gko::lend
(
    const Pointer & p ) [inline]
```

Returns a non-owning (plain) pointer to the object pointed to by *p*.

Template Parameters

<i>Pointer</i>	type of pointer to the object (plain or smart pointer)
----------------	--

Parameters

<i>p</i>	a pointer to the object
----------	-------------------------

Note

This is the overload for owning (smart) pointers, that behaves the same as calling `.get()` on the smart pointer.

Referenced by `gko::preconditioner::Jacobi< ValueType, IndexType >::get_num_stored_elements()`, and `gko::log↵
::EnableLogging< Executor >::remove_logger()`.

```
250 {
251     return p.get();
252 }
```

23.1.4.17 `lend()` [2/2]

```
template<typename Pointer >
std::enable_if<!detail::have_ownership<Pointer>(), detail::pointee<Pointer>*>::type gko↵
::lend (
    const Pointer & p ) [inline]
```

Returns a non-owning (plain) pointer to the object pointed to by `p`.

Template Parameters

<i>Pointer</i>	type of pointer to the object (plain or smart pointer)
----------------	--

Parameters

<i>p</i>	a pointer to the object
----------	-------------------------

Note

This is the overload for non-owning (plain) pointers, that just returns `p`.

```
268 {
269     return p;
270 }
```

23.1.4.18 make_temporary_clone()

```
template<typename T >
temporary_clone<T> gko::make_temporary_clone (
    std::shared_ptr< const Executor > exec,
    T * ptr )
```

Creates a [temporary_clone](#).

This is a helper function which avoids the need to explicitly specify the type of the object, as would be the case if using the constructor of [temporary_clone](#).

Parameters

<i>exec</i>	the executor where the clone will be created
<i>ptr</i>	a pointer to the object of which the clone will be created

Referenced by `gko::matrix::Dense< ValueType >::add_scaled()`, `gko::matrix::Coo< ValueType, IndexType >::apply2()`, `gko::matrix::Dense< ValueType >::compute_dot()`, `gko::matrix::Dense< ValueType >::compute_norm2()`, and `gko::matrix::Dense< ValueType >::scale()`.

```
476 {
477     return temporary_clone<T>(std::move(exec), ptr);
478 }
```

23.1.4.19 max()

```
template<typename T >
constexpr T gko::max (
    const T & x,
    const T & y ) [inline]
```

Returns the larger of the arguments.

Template Parameters

<i>T</i>	type of the arguments
----------	-----------------------

Parameters

<i>x</i>	first argument
<i>y</i>	second argument

Returns

$x \geq y ? x : y$

Note

C++11 version of this function is not `constexpr`, thus we provide our own implementation.

```
385 {
386     return x >= y ? x : y;
387 }
```

23.1.4.20 min()

```
template<typename T >
constexpr T gko::min (
    const T & x,
    const T & y ) [inline]
```

Returns the smaller of the arguments.

Template Parameters

<i>T</i>	type of the arguments
----------	-----------------------

Parameters

<i>x</i>	first argument
<i>y</i>	second argument

Returns

$x \leq y ? x : y$

Note

C++11 version of this function is not `constexpr`, thus we provide our own implementation.

```
405 {
406     return x <= y ? x : y;
407 }
```

23.1.4.21 one() [1/2]

```
template<typename T >
constexpr T gko::one ( ) [inline]
```

Returns the multiplicative identity for T.

Returns

the multiplicative identity for T

Referenced by `gko::matrix_data< ValueType, IndexType >::ensure_row_major_order()`, and `gko::Combination< ValueType >::get_operators()`.

```
331 {
332     return T(1);
333 }
```

23.1.4.22 one() [2/2]

```
template<typename T >
constexpr T gko::one (
    const T & ) [inline]
```

Returns the multiplicative identity for T.

Returns

the multiplicative identity for T

Note

This version takes an unused reference argument to avoid complicated calls like `one<decltype(x)>()`. Instead, it allows `one(x)`.

```
346 {
347     return one<T>();
348 }
```

23.1.4.23 operator!=() [1/3]

```
bool gko::operator!=(
    const stopping_status & x,
    const stopping_status & y ) [inline], [noexcept]
```

Checks if two stopping statuses are different.

Parameters

<i>x</i>	a stopping status
<i>y</i>	a stopping status

Returns

true if and only if `! (x == y)`

```
179 {
180     return x.data_ != y.data_;
181 }
```

23.1.4.24 operator"!="() [2/3]

```
template<size_type Dimensionality, typename DimensionType >
constexpr bool gko::operator!= (
    const dim< Dimensionality, DimensionType > & x,
    const dim< Dimensionality, DimensionType > & y ) [inline]
```

Checks if two dim objects are different.

Template Parameters

<i>Dimensionality</i>	number of dimensions of the dim objects
<i>DimensionType</i>	datatype used to represent each dimension

Parameters

<i>x</i>	first object
<i>y</i>	second object

Returns

`! (x == y)`

```
219 {
220     return !(x == y);
221 }
```

23.1.4.25 operator"!="() [3/3]

```
constexpr bool gko::operator!= (
    precision_reduction x,
    precision_reduction y ) [noexcept]
```

Checks if two `precision_reduction` encodings are different.

Parameters

<i>x</i>	an encoding
<i>y</i>	an encoding

Returns

true if and only if `x` and `y` are different encodings.

```

368 {
369     using st = precision_reduction::storage_type;
370     return static_cast<st>(x) != static_cast<st>(y);
371 }
```

23.1.4.26 operator<<() [1/2]

```

std::ostream& gko::operator<< (
    std::ostream & os,
    const version & ver ) [inline]
```

Prints version information to a stream.

Parameters

<i>os</i>	output stream
<i>ver</i>	version structure

Returns

os

References `gko::version::major`, `gko::version::minor`, `gko::version::patch`, and `gko::version::tag`.

```

110 {
111     os << ver.major << "." << ver.minor << "." << ver.patch;
112     if (ver.tag) {
113         os << " (" << ver.tag << ")";
114     }
115     return os;
116 }
```

23.1.4.27 operator<<() [2/2]

```

std::ostream& gko::operator<< (
    std::ostream & os,
    const version_info & ver_info )
```

Prints library version information in human-readable format to a stream.

Parameters

<i>os</i>	output stream
<i>ver_info</i>	version information

Returns

OS

23.1.4.28 operator==([1/2]

```
bool gko::operator== (
    const stopping_status & x,
    const stopping_status & y ) [inline], [noexcept]
```

Checks if two stopping statuses are equivalent.

Parameters

<i>x</i>	a stopping status
<i>y</i>	a stopping status

Returns

true if and only if both *x* and *y* have the same mask and converged and finalized state

```
164 {
165     return x.data_ == y.data_;
166 }
```

23.1.4.29 operator==([2/2]

```
constexpr bool gko::operator== (
    precision_reduction x,
    precision_reduction y ) [noexcept]
```

Checks if two [precision_reduction](#) encodings are equal.

Parameters

<i>x</i>	an encoding
<i>y</i>	an encoding

Returns

true if and only if *x* and *y* are the same encodings

```
352 {
353     using st = precision_reduction::storage_type;
354     return static_cast<st>(x) == static_cast<st>(y);
355 }
```

23.1.4.30 read()

```
template<typename MatrixType , typename StreamType , typename... MatrixArgs>
std::unique_ptr<MatrixType> gko::read (
    StreamType && is,
    MatrixArgs &&... args ) [inline]
```

Reads a matrix stored in matrix market format from an input stream.

Template Parameters

<i>MatrixType</i>	a ReadableFromMatrixData LinOp type used to store the matrix once it's been read from disk.
<i>StreamType</i>	type of stream used to write the data to
<i>MatrixArgs</i>	additional argument types passed to MatrixType constructor

Parameters

<i>is</i>	input stream from which to read the data
<i>args</i>	additional arguments passed to MatrixType constructor

Returns

A MatrixType LinOp filled with data from filename

References `read_raw()`.

```
114 {
115     auto mtx = MatrixType::create(std::forward<MatrixArgs>(args)...);
116     mtx->read(read_raw<typename MatrixType::value_type,
117               typename MatrixType::index_type>(is));
118     return mtx;
119 }
```

23.1.4.31 read_raw()

```
template<typename ValueType = default_precision, typename IndexType = int32>
matrix_data<ValueType, IndexType> gko::read_raw (
    std::istream & is )
```

Reads a matrix stored in matrix market format from an input stream.

Template Parameters

<i>ValueType</i>	type of matrix values
<i>IndexType</i>	type of matrix indexes

Parameters

<i>is</i>	input stream from which to read the data
-----------	--

Returns

A [matrix_data](#) structure containing the matrix. The nonzero elements are sorted in lexicographic order of their (row, column) indexes.

Note

This is an advanced routine that will return the raw matrix data structure. Consider using [gko::read](#) instead.

Referenced by [read\(\)](#).

23.1.4.32 real()

```
template<typename T >
constexpr T gko::real (
    const T & x ) [inline]
```

Returns the real part of the object.

Template Parameters

<i>T</i>	type of the object
----------	--------------------

Parameters

<i>x</i>	the object
----------	------------

Returns

real part of the object (by default, the object itself)

Referenced by [squared_norm\(\)](#).

```
421 {
422     return x;
423 }
```

23.1.4.33 round_down()

```
template<typename T >
constexpr reduce\_precision<T> gko::round_down (
    T val ) [inline]
```

Reduces the precision of the input parameter.

Template Parameters

<i>T</i>	the original precision
----------	------------------------

Parameters

<i>val</i>	the value to round down
------------	-------------------------

Returns

the rounded down value

```

194 {
195     return static_cast<reduce_precision<T>>(val);
196 }
```

23.1.4.34 round_up()

```

template<typename T >
constexpr increase_precision<T> gko::round_up (
    T val ) [inline]
```

Increases the precision of the input parameter.

Template Parameters

<i>T</i>	the original precision
----------	------------------------

Parameters

<i>val</i>	the value to round up
------------	-----------------------

Returns

the rounded up value

References `byte_size`.

```

210 {
211     return static_cast<increase_precision<T>>(val);
212 }
```

23.1.4.35 share()

```
template<typename OwingPointer >
detail::shared_type<OwingPointer> gko::share (
    OwingPointer && p ) [inline]
```

Marks the object pointed to by `p` as shared.

Effectively converts a pointer with ownership to `std::shared_ptr`.

Template Parameters

<i>OwingPointer</i>	type of pointer with ownership to the object (has to be a smart pointer)
---------------------	--

Parameters

<i>p</i>	a pointer to the object
----------	-------------------------

Note

The original pointer `p` becomes invalid after this call.

```
207 {
208     static_assert(detail::have_ownership<OwingPointer>(),
209                 "OwingPointer does not have ownership of the object");
210     return detail::shared_type<OwingPointer>(std::move(p));
211 }
```

23.1.4.36 squared_norm()

```
template<typename T >
constexpr auto gko::squared_norm (
    const T & x ) -> decltype(real(conj(x) * x)) [inline]
```

Returns the squared norm of the object.

Template Parameters

<i>T</i>	type of the object.
----------	---------------------

Returns

The squared norm of the object.

References `conj()`, and `real()`.

```
469 {
470     return real(conj(x) * x);
471 }
```

23.1.4.37 transpose()

```
template<typename DimensionType >
constexpr dim<2, DimensionType> gko::transpose (
    const dim< 2, DimensionType > & dimensions ) [inline], [noexcept]
```

Returns a [dim](#)<2> object with its dimensions swapped.

Template Parameters

<i>DimensionType</i>	datatype used to represent each dimension
----------------------	---

Parameters

<i>dimensions</i>	original object
-------------------	-----------------

Returns

a [dim](#)<2> object with its dimensions swapped

Referenced by `gko::matrix_data< ValueType, IndexType >::ensure_row_major_order()`, `gko::solver::Gmres< ValueType >::get_krylov_dim()`, `gko::preconditioner::Jacobi< ValueType, IndexType >::get_num_stored_↵
_elements()`, `gko::solver::Cgs< ValueType >::get_preconditioner()`, `gko::solver::Cg< ValueType >::get_↵
preconditioner()`, `gko::solver::Bicgstab< ValueType >::get_preconditioner()`, `gko::solver::Fcg< ValueType >↵
::get_preconditioner()`, and `gko::solver::lr< ValueType >::get_solver()`.

```
236 {
237     return {dimensions[1], dimensions[0]};
238 }
```

23.1.4.38 write()

```
template<typename MatrixType , typename StreamType >
void gko::write (
    StreamType && os,
    MatrixType * matrix,
    layout\_type layout = layout\_type::array ) [inline]
```

Reads a matrix stored in matrix market format from an input stream.

Template Parameters

<i>MatrixType</i>	a ReadableFromMatrixData LinOp type used to store the matrix once it's been read from disk.
<i>StreamType</i>	type of stream used to write the data to

Parameters

<i>os</i>	output stream where the data is to be written
-----------	---

Parameters

<i>matrix</i>	the matrix to write
<i>layout</i>	the layout used in the output

References `write_raw()`.

Referenced by `gko::preconditioner::Jacobi< ValueType, IndexType >::get_num_stored_elements()`.

```

136 {
137     matrix_data<typename MatrixType::value_type,
138               typename MatrixType::index_type>
139         data{};
140     matrix->write(data);
141     write_raw(os, data, layout);
142 }
```

23.1.4.39 `write_raw()`

```

template<typename ValueType , typename IndexType >
void gko::write_raw (
    std::ostream & os,
    const matrix_data< ValueType, IndexType > & data,
    layout_type layout = layout_type::array )
```

Writes a `matrix_data` structure to a stream in matrix market format.

Template Parameters

<i>ValueType</i>	type of matrix values
<i>IndexType</i>	type of matrix indexes

Parameters

<i>os</i>	output stream where the data is to be written
<i>data</i>	the matrix data to write
<i>layout</i>	the layout used in the output

Note

This is an advanced routine that writes the raw matrix data structure. If you are trying to write an existing matrix, consider using `gko::write` instead.

Referenced by `write()`.

23.1.4.40 zero() [1/2]

```
template<typename T >
constexpr T gko::zero ( ) [inline]
```

Returns the additive identity for T.

Returns

additive identity for T

Referenced by `gko::matrix_data< ValueType, IndexType >::ensure_row_major_order()`, and `gko::Combination< ValueType >::get_operators()`.

```
304 {
305     return T(0);
306 }
```

23.1.4.41 zero() [2/2]

```
template<typename T >
constexpr T gko::zero (
    const T & ) [inline]
```

Returns the additive identity for T.

Returns

additive identity for T

Note

This version takes an unused reference argument to avoid complicated calls like `zero<decltype(x)>()`. Instead, it allows `zero(x)`.

```
319 {
320     return zero<T>();
321 }
```

23.2 gko::accessor Namespace Reference

The accessor namespace.

Classes

- class [row_major](#)

A [row_major](#) accessor is a bridge between a range and the row-major memory layout.

23.2.1 Detailed Description

The accessor namespace.

23.3 gko::factorization Namespace Reference

The Factorization namespace.

Classes

- class [Parllu](#)
ParLLU is an incomplete LU factorization which is computed in parallel.

23.3.1 Detailed Description

The Factorization namespace.

23.4 gko::log Namespace Reference

The logger namespace .

Classes

- class [Convergence](#)
Convergence is a Logger which logs data strictly from the `criterion_check_completed` event.
- struct [criterion_data](#)
Struct representing Criterion related data.
- class [EnableLogging](#)
EnableLogging is a mixin which should be inherited by any class which wants to enable logging.
- struct [executor_data](#)
Struct representing Executor related data.
- struct [iteration_complete_data](#)
Struct representing iteration complete related data.
- struct [linop_data](#)
Struct representing LinOp related data.
- struct [linop_factory_data](#)
Struct representing LinOp factory related data.
- class [Loggable](#)
Loggable class is an interface which should be implemented by classes wanting to support logging.
- struct [operation_data](#)
Struct representing Operator related data.
- struct [polymorphic_object_data](#)
Struct representing PolymorphicObject related data.
- class [Record](#)
Record is a Logger which logs every event to an object.
- class [Stream](#)
Stream is a Logger which logs every event to a stream.

23.4.1 Detailed Description

The logger namespace .

The Logging namespace.

[Logging](#)

23.5 gko::matrix Namespace Reference

The matrix namespace.

Classes

- class [Coo](#)
COO stores a matrix in the coordinate matrix format.
- class [Csr](#)
CSR is a matrix format which stores only the nonzero coefficients by compressing each row of the matrix (compressed sparse row format).
- class [Dense](#)
Dense is a matrix format which explicitly stores all values of the matrix.
- class [Ell](#)
ELL is a matrix format where stride with explicit zeros is used such that all rows have the same number of stored elements.
- class [Hybrid](#)
HYBRID is a matrix format which splits the matrix into ELLPACK and COO format.
- class [Identity](#)
This class is a utility which efficiently implements the identity matrix (a linear operator which maps each vector to itself).
- class [IdentityFactory](#)
This factory is a utility which can be used to generate [Identity](#) operators.
- class [Sellp](#)
SELL-P is a matrix format similar to ELL format.

23.5.1 Detailed Description

The matrix namespace.

23.6 gko::name_demangling Namespace Reference

The name demangling namespace.

Functions

- `template<typename T >`
`std::string get_static_type (const T &)`
*This function uses name demangling facilities to get the name of the static type (*T*) of the object passed in arguments.*
- `template<typename T >`
`std::string get_dynamic_type (const T &t)`
This function uses name demangling facilities to get the name of the dynamic type of the object passed in arguments.

23.6.1 Detailed Description

The name demangling namespace.

23.6.2 Function Documentation

23.6.2.1 `get_dynamic_type()`

```
template<typename T >
std::string gko::name_demangling::get_dynamic_type (
    const T & t )
```

This function uses name demangling facilities to get the name of the dynamic type of the object passed in arguments.

Template Parameters

<i>T</i>	the type of the object to demangle
----------	------------------------------------

Parameters

<i>t</i>	the object we get the dynamic type of
----------	---------------------------------------

```
100 {
101     return get_type_name(typeid(t));
102 }
```

23.6.2.2 `get_static_type()`

```
template<typename T >
std::string gko::name_demangling::get_static_type (
    const T & )
```

This function uses name demangling facilities to get the name of the static type (*T*) of the object passed in arguments.

Template Parameters

<i>T</i>	the type of the object to demangle
----------	------------------------------------

Parameters

<i>unused</i>	
---------------	--

```

85 {
86     return get_type_name (typeid(T) );
87 }
```

23.7 gko::preconditioner Namespace Reference

The Preconditioner namespace.

Classes

- struct [block_interleaved_storage_scheme](#)
Defines the parameters of the interleaved block storage scheme used by block-Jacobi blocks.
- class [Jacobi](#)
A block-Jacobi preconditioner is a block-diagonal linear operator, obtained by inverting the diagonal blocks of the source operator.

23.7.1 Detailed Description

The Preconditioner namespace.

23.8 gko::solver Namespace Reference

The ginkgo Solve namespace.

Classes

- class [Bicgstab](#)
BiCGSTAB or the Bi-Conjugate Gradient-Stabilized is a Krylov subspace solver.
- class [Cg](#)
CG or the conjugate gradient method is an iterative type Krylov subspace method which is suitable for symmetric positive definite methods.
- class [Cgs](#)
CGS or the conjugate gradient square method is an iterative type Krylov subspace method which is suitable for general systems.
- class [Fcg](#)
FCG or the flexible conjugate gradient method is an iterative type Krylov subspace method which is suitable for symmetric positive definite methods.
- class [Gmres](#)
GMRES or the generalized minimal residual method is an iterative type Krylov subspace method which is suitable for nonsymmetric linear systems.
- class [lr](#)
Iterative refinement (IR) is an iterative method that uses another coarse method to approximate the error of the current solution via the current residual.

23.8.1 Detailed Description

The ginkgo Solve namespace.

23.9 gko::stop Namespace Reference

The Stopping criterion namespace.

Classes

- class [Combined](#)
The [Combined](#) class is used to combine multiple criterions together through an OR operation.
- class [Criterion](#)
The [Criterion](#) class is a base class for all stopping criteria.
- struct [CriterionArgs](#)
This struct is used to pass parameters to the `EnableDefaultCriterionFactory::generate()` method.
- class [Iteration](#)
The [Iteration](#) class is a stopping criterion which stops the iteration process after a preset number of iterations.
- class [ResidualNormReduction](#)
The [ResidualNormReduction](#) class is a stopping criterion which stops the iteration process when the relative residual norm is below a certain threshold.
- class [Time](#)
The [Time](#) class is a stopping criterion which stops the iteration process after a certain amount of time has passed.

Typedefs

- using [CriterionFactory](#) = [AbstractFactory](#)< [Criterion](#), [CriterionArgs](#) >
Declares an Abstract Factory specialized for [Criteria](#)s.
- template<typename ConcreteFactory , typename ConcreteCriterion , typename ParametersType , typename PolymorphicBase = [CriterionFactory](#)>
using [EnableDefaultCriterionFactory](#) = [EnableDefaultFactory](#)< ConcreteFactory, ConcreteCriterion, ParametersType, PolymorphicBase >
This is an alias for the [EnableDefaultFactory](#) mixin, which correctly sets the template parameters to enable a subclass of [CriterionFactory](#).

Functions

- template<typename FactoryContainer >
std::shared_ptr< const [CriterionFactory](#) > [combine](#) (FactoryContainer &&factories)
Combines multiple criterion factories into a single combined criterion factory.

23.9.1 Detailed Description

The Stopping criterion namespace.

[Stopping criteria](#)

23.9.2 Typedef Documentation

23.9.2.1 EnableDefaultCriterionFactory

```
template<typename ConcreteFactory , typename ConcreteCriterion , typename ParametersType ,
typename PolymorphicBase = CriterionFactory>
using gko::stop::EnableDefaultCriterionFactory = typedef EnableDefaultFactory<ConcreteFactory,
ConcreteCriterion, ParametersType, PolymorphicBase>
```

This is an alias for the [EnableDefaultFactory](#) mixin, which correctly sets the template parameters to enable a subclass of CriterionFactory.

Template Parameters

<i>ConcreteFactory</i>	the concrete factory which is being implemented [CRTP parameter]
<i>ConcreteCriterion</i>	the concrete Criterion type which this factory produces, needs to have a constructor which takes a const ConcreteFactory *, and a const CriterionArgs * as parameters.
<i>ParametersType</i>	a subclass of enable_parameters_type template which defines all of the parameters of the factory
<i>PolymorphicBase</i>	parent of ConcreteFactory in the polymorphic hierarchy, has to be a subclass of CriterionFactory

23.10 gko::syn Namespace Reference

The Synthesizer namespace.

23.10.1 Detailed Description

The Synthesizer namespace.

23.11 gko::xstd Namespace Reference

The namespace for functionalities after C++11 standard.

23.11.1 Detailed Description

The namespace for functionalities after C++11 standard.

Chapter 24

Class Documentation

24.1 gko::AbstractFactory< AbstractProductType, ComponentsType > Class Template Reference

The [AbstractFactory](#) is a generic interface template that enables easy implementation of the abstract factory design pattern.

```
#include <ginkgo/core/base/abstract_factory.hpp>
```

Public Member Functions

- `template<typename... Args>
std::unique_ptr< AbstractProductType > generate (Args &&... args) const`
Creates a new product from the given components.

24.1.1 Detailed Description

```
template<typename AbstractProductType, typename ComponentsType>  
class gko::AbstractFactory< AbstractProductType, ComponentsType >
```

The [AbstractFactory](#) is a generic interface template that enables easy implementation of the abstract factory design pattern.

The interface provides the [AbstractFactory::generate\(\)](#) method that can produce products of type `AbstractProductType` using an object of `ComponentsType` (which can be constructed on the fly from parameters to its constructors). The [generate\(\)](#) method is not declared as virtual, as this allows subclasses to hide the method with a variant that preserves the compile-time type of the objects. Instead, implementers should override the `generate_impl()` method, which is declared virtual.

Implementers of concrete factories should consider using the [EnableDefaultFactory](#) mixin to obtain default implementations of utility methods of [PolymorphicObject](#) and [AbstractFactory](#).

Template Parameters

<i>AbstractProductType</i>	the type of products the factory produces
<i>ComponentsType</i>	the type of components the factory needs to produce the product

24.1.2 Member Function Documentation

24.1.2.1 generate()

```
template<typename AbstractProductType, typename ComponentsType>
template<typename... Args>
std::unique_ptr<AbstractProductType> gko::AbstractFactory< AbstractProductType, ComponentsType >::generate (
    Args &&... args ) const [inline]
```

Creates a new product from the given components.

The method will create an ComponentsType object from the arguments of this method, and pass it to the generate_impl() function which will create a new AbstractProductType.

Template Parameters

<i>Args</i>	types of arguments passed to the constructor of ComponentsType
-------------	--

Parameters

<i>args</i>	arguments passed to the constructor of ComponentsType
-------------	---

Returns

an instance of AbstractProductType

```
93     {
94         auto product = this->generate_impl({std::forward<Args>(args)...});
95         for (auto logger : this->loggers_) {
96             product->add_logger(logger);
97         }
98         return product;
99     }
```

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

- ginkgo/core/base/abstract_factory.hpp (8045ac75)

24.2 gko::AllocationError Class Reference

[AllocationError](#) is thrown if a memory allocation fails.

```
#include <ginkgo/core/base/exception.hpp>
```

Public Member Functions

- [AllocationError](#) (const std::string &file, int line, const std::string &device, [size_type](#) bytes)
Initializes an allocation error.

24.2.1 Detailed Description

[AllocationError](#) is thrown if a memory allocation fails.

24.2.2 Constructor & Destructor Documentation

24.2.2.1 AllocationError()

```
gko::AllocationError::AllocationError (
    const std::string & file,
    int line,
    const std::string & device,
    size_type bytes ) [inline]
```

Initializes an allocation error.

Parameters

<i>file</i>	The name of the offending source file
<i>line</i>	The source code line number where the error occurred
<i>device</i>	The device on which the error occurred
<i>bytes</i>	The size of the memory block whose allocation failed.

```
286         : Error(file, line,
287               device + ": failed to allocate memory block of " +
288                 std::to_string(bytes) + "B")
289     {}
```

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

- ginkgo/core/base/exception.hpp (8045ac75)

24.3 gko::Array< ValueType > Class Template Reference

An [Array](#) is a container which encapsulates fixed-sized arrays, stored on the [Executor](#) tied to the [Array](#).

```
#include <ginkgo/core/base/array.hpp>
```

Public Types

- using [value_type](#) = ValueType
The type of elements stored in the array.
- using [default_deleter](#) = [executor_deleter](#)< [value_type](#)[]>
The default deleter type used by Array.
- using [view_deleter](#) = [null_deleter](#)< [value_type](#)[]>
The deleter type used for views.

Public Member Functions

- [Array](#) () noexcept
Creates an empty [Array](#) not tied to any executor.
- [Array](#) (std::shared_ptr< const [Executor](#) > exec) noexcept
Creates an empty [Array](#) tied to the specified [Executor](#).
- [Array](#) (std::shared_ptr< const [Executor](#) > exec, [size_type](#) num_elems)
Creates an [Array](#) on the specified [Executor](#).
- template<typename DeleterType >
[Array](#) (std::shared_ptr< const [Executor](#) > exec, [size_type](#) num_elems, [value_type](#) *data, DeleterType deleter)
Creates an [Array](#) from existing memory.
- [Array](#) (std::shared_ptr< const [Executor](#) > exec, [size_type](#) num_elems, [value_type](#) *data)
Creates an [Array](#) from existing memory.
- template<typename RandomAccessIterator >
[Array](#) (std::shared_ptr< const [Executor](#) > exec, RandomAccessIterator begin, RandomAccessIterator end)
Creates an array on the specified [Executor](#) and initializes it with values.
- template<typename T >
[Array](#) (std::shared_ptr< const [Executor](#) > exec, std::initializer_list< T > init_list)
Creates an array on the specified [Executor](#) and initializes it with values.
- [Array](#) (std::shared_ptr< const [Executor](#) > exec, const [Array](#) &other)
Creates a copy of another array on a different executor.
- [Array](#) (const [Array](#) &other)
Creates a copy of another array.
- [Array](#) (std::shared_ptr< const [Executor](#) > exec, [Array](#) &&other)
Moves another array to a different executor.
- [Array](#) ([Array](#) &&other)
Moves another array.
- [Array](#) & operator= (const [Array](#) &other)
Copies data from another array.
- [Array](#) & operator= ([Array](#) &&other)
Moves data from another array.
- void [clear](#) () noexcept
Deallocates all data used by the [Array](#).
- void [resize_and_reset](#) ([size_type](#) num_elems)
Resizes the array so it is able to hold the specified number of elements.
- [size_type](#) [get_num_elems](#) () const noexcept
Returns the number of elements in the [Array](#).
- [value_type](#) * [get_data](#) () noexcept
Returns a pointer to the block of memory used to store the elements of the [Array](#).
- const [value_type](#) * [get_const_data](#) () const noexcept
Returns a constant pointer to the block of memory used to store the elements of the [Array](#).
- std::shared_ptr< const [Executor](#) > [get_executor](#) () const noexcept
Returns the [Executor](#) associated with the array.
- void [set_executor](#) (std::shared_ptr< const [Executor](#) > exec)
Changes the [Executor](#) of the [Array](#), moving the allocated data to the new [Executor](#).

Static Public Member Functions

- static [Array](#) [view](#) (std::shared_ptr< const [Executor](#) > exec, [size_type](#) num_elems, [value_type](#) *data)
Creates an [Array](#) from existing memory.

24.3.1 Detailed Description

```
template<typename ValueType>
class gko::Array< ValueType >
```

An [Array](#) is a container which encapsulates fixed-sized arrays, stored on the [Executor](#) tied to the [Array](#).

The array stores and transfers its data as **raw** memory, which means that the constructors of its elements are not called when constructing, copying or moving the [Array](#). Thus, the [Array](#) class is most suitable for storing POD types.

Template Parameters

<i>ValueType</i>	the type of elements stored in the array
------------------	--

24.3.2 Constructor & Destructor Documentation

24.3.2.1 Array() [1/11]

```
template<typename ValueType>
gko::Array< ValueType >::Array ( ) [inline], [noexcept]
```

Creates an empty [Array](#) not tied to any executor.

An array without an assigned executor can only be empty. Attempts to change its size (e.g. via the `resize_and_reset` method) will result in an exception. If such an array is used as the right hand side of an assignment or move assignment expression, the data of the target array will be cleared, but its executor will not be modified.

The executor can later be set by using the `set_executor` method. If an [Array](#) with no assigned executor is assigned or moved to, it will inherit the executor of the source [Array](#).

```
94         : num_elems_(0),
95         data_(nullptr, default_deleter{nullptr}),
96         exec_(nullptr)
97     {}
```

24.3.2.2 Array() [2/11]

```
template<typename ValueType>
gko::Array< ValueType >::Array (
    std::shared_ptr< const Executor > exec ) [inline], [noexcept]
```

Creates an empty [Array](#) tied to the specified [Executor](#).

Parameters

<i>exec</i>	the Executor where the array data is allocated
-------------	--

```

105         : num_elems_(0),
106         data_(nullptr, default_deleter{exec}),
107         exec_(std::move(exec))
108     {}

```

24.3.2.3 [Array\(\)](#) [3/11]

```

template<typename ValueType>
gko::Array< ValueType >::Array (
    std::shared_ptr< const Executor > exec,
    size_type num_elems ) [inline]

```

Creates an [Array](#) on the specified [Executor](#).

Parameters

<i>exec</i>	the Executor where the array data will be allocated
<i>num_elems</i>	the amount of memory (expressed as the number of <code>value_type</code> elements) allocated on the Executor

```

118         : num_elems_(num_elems),
119         data_(nullptr, default_deleter{exec}),
120         exec_(std::move(exec))
121     {
122         if (num_elems > 0) {
123             data_.reset(exec->alloc<value_type>(num_elems));
124         }
125     }

```

24.3.2.4 [Array\(\)](#) [4/11]

```

template<typename ValueType>
template<typename DeleterType >
gko::Array< ValueType >::Array (
    std::shared_ptr< const Executor > exec,
    size_type num_elems,
    value_type * data,
    DeleterType deleter ) [inline]

```

Creates an [Array](#) from existing memory.

The memory will be managed by the array, and deallocated using the specified deleter (e.g. use `std::default_delete` for data allocated with `new`).

Template Parameters

<i>DeleterType</i>	type of the deleter
--------------------	---------------------

Parameters

<i>exec</i>	executor where <i>data</i> is located
<i>num_elems</i>	number of elements in <i>data</i>
<i>data</i>	chunk of memory used to create the array
<i>deleter</i>	the deleter used to free the memory

See also

[Array::view\(\)](#) to create an [array](#) that does not deallocate memory
[Array\(std::shared_ptr<cont Executor>, size_type, value_type*\)](#) to deallocate the memory using [Executor::free\(\)](#) method

```
148         : num_elems_{num_elems}, data_(data, deleter), exec_{exec}
149     {}
```

24.3.2.5 [Array\(\)](#) [5/11]

```
template<typename ValueType>
gko::Array< ValueType >::Array (
    std::shared_ptr< const Executor > exec,
    size_type num_elems,
    value_type * data ) [inline]
```

Creates an [Array](#) from existing memory.

The memory will be managed by the array, and deallocated using the [Executor::free](#) method.

Parameters

<i>exec</i>	executor where <i>data</i> is located
<i>num_elems</i>	number of elements in <i>data</i>
<i>data</i>	chunk of memory used to create the array

```
163         : Array(exec, num_elems, data, default_deleter(exec))
164     {}
```

24.3.2.6 [Array\(\)](#) [6/11]

```
template<typename ValueType>
template<typename RandomAccessIterator >
```

```
gko::Array< ValueType >::Array (
    std::shared_ptr< const Executor > exec,
    RandomAccessIterator begin,
    RandomAccessIterator end ) [inline]
```

Creates an array on the specified [Executor](#) and initializes it with values.

Template Parameters

<i>RandomAccessIterator</i>	type of the iterators
-----------------------------	-----------------------

Parameters

<i>exec</i>	the Executor where the array data will be allocated
<i>begin</i>	start of range of values
<i>end</i>	end of range of values

```
179         : Array(exec)
180     {
181         Array tmp(exec->get_master(), end - begin);
182         int i = 0;
183         for (auto it = begin; it != end; ++it, ++i) {
184             tmp.data_[i] = *it;
185         }
186         *this = std::move(tmp);
187     }
```

24.3.2.7 Array() [7/11]

```
template<typename ValueType>
template<typename T >
gko::Array< ValueType >::Array (
    std::shared_ptr< const Executor > exec,
    std::initializer_list< T > init_list ) [inline]
```

Creates an array on the specified [Executor](#) and initializes it with values.

Template Parameters

<i>T</i>	type of values used to initialize the array (T has to be implicitly convertible to value_type)
----------	--

Parameters

<i>exec</i>	the Executor where the array data will be allocated
<i>init_list</i>	list of values used to initialize the Array

```
202         : Array(exec, begin(init_list), end(init_list))
203     {}
```


24.3.2.8 Array() [8/11]

```
template<typename ValueType>
gko::Array< ValueType >::Array (
    std::shared_ptr< const Executor > exec,
    const Array< ValueType > & other ) [inline]
```

Creates a copy of another array on a different executor.

This does not invoke the constructors of the elements, instead they are copied as POD types.

Parameters

<i>exec</i>	the executor where the new array will be created
<i>other</i>	the Array to copy from

```
215         : Array(exec)
216     {
217         *this = other;
218     }
```

24.3.2.9 Array() [9/11]

```
template<typename ValueType>
gko::Array< ValueType >::Array (
    const Array< ValueType > & other ) [inline]
```

Creates a copy of another array.

This does not invoke the constructors of the elements, instead they are copied as POD types.

Parameters

<i>other</i>	the Array to copy from
--------------	--

```
228 : Array(other.get_executor(), other) {}
```

24.3.2.10 Array() [10/11]

```
template<typename ValueType>
gko::Array< ValueType >::Array (
    std::shared_ptr< const Executor > exec,
    Array< ValueType > && other ) [inline]
```

Moves another array to a different executor.

This does not invoke the constructors of the elements, instead they are copied as POD types.

Parameters

<i>exec</i>	the executor where the new array will be moved
<i>other</i>	the Array to move

```

239                                     : Array(exec)
240     {
241         *this = std::move(other);
242     }

```

24.3.2.11 [Array\(\)](#) [11/11]

```

template<typename ValueType>
gko::Array< ValueType >::Array (
    Array< ValueType > && other ) [inline]

```

Moves another array.

This does not invoke the constructors of the elements, instead they are copied as POD types.

Parameters

<i>other</i>	the Array to move
--------------	-----------------------------------

```

252 : Array(other.get_executor(), std::move(other)) {}

```

24.3.3 Member Function Documentation

24.3.3.1 [clear\(\)](#)

```

template<typename ValueType>
void gko::Array< ValueType >::clear ( ) [inline], [noexcept]

```

Deallocates all data used by the [Array](#).

The array is left in a valid, but empty state, so the same array can be used to allocate new memory. Calls to [Array::get_data\(\)](#) will return a `nullptr`.

```

351     {
352         num_elems_ = 0;
353         data_.reset(nullptr);
354     }

```

24.3.3.2 get_const_data()

```
template<typename ValueType>
const value_type* gko::Array< ValueType >::get_const_data ( ) const [inline], [noexcept]
```

Returns a constant pointer to the block of memory used to store the elements of the [Array](#).

Returns

a constant pointer to the block of memory used to store the elements of the [Array](#)

Referenced by gko::matrix::Dense< ValueType >::at(), gko::matrix::Sellp< ValueType, IndexType >::get_const_col_idxs(), gko::matrix::Ell< ValueType, IndexType >::get_const_col_idxs(), gko::matrix::Coo< ValueType, IndexType >::get_const_col_idxs(), gko::matrix::Coo< ValueType, IndexType >::get_const_row_idxs(), gko::matrix::Sellp< ValueType, IndexType >::get_const_slice_lengths(), gko::matrix::Sellp< ValueType, IndexType >::get_const_slice_sets(), gko::matrix::Sellp< ValueType, IndexType >::get_const_values(), gko::matrix::Ell< ValueType, IndexType >::get_const_values(), gko::matrix::Coo< ValueType, IndexType >::get_const_values(), gko::matrix::Dense< ValueType >::get_const_values(), gko::matrix::Hybrid< ValueType, IndexType >::strategy_type::get_coo_nnz(), gko::Array< index_type >::operator=(), gko::matrix::Ell< ValueType, IndexType >::val_at(), and gko::matrix::Sellp< ValueType, IndexType >::val_at().

```
406 { return data_.get(); }
```

24.3.3.3 get_data()

```
template<typename ValueType>
value_type* gko::Array< ValueType >::get_data ( ) [inline], [noexcept]
```

Returns a pointer to the block of memory used to store the elements of the [Array](#).

Returns

a pointer to the block of memory used to store the elements of the [Array](#)

Referenced by gko::matrix::Dense< ValueType >::at(), gko::matrix::Hybrid< ValueType, IndexType >::imbalance_limit::compute_ell_num_stored_elements_per_row(), gko::matrix::Sellp< ValueType, IndexType >::get_col_idxs(), gko::matrix::Ell< ValueType, IndexType >::get_col_idxs(), gko::matrix::Coo< ValueType, IndexType >::get_col_idxs(), gko::matrix::Coo< ValueType, IndexType >::get_row_idxs(), gko::matrix::Sellp< ValueType, IndexType >::get_slice_lengths(), gko::matrix::Sellp< ValueType, IndexType >::get_slice_sets(), gko::matrix::Sellp< ValueType, IndexType >::get_values(), gko::matrix::Ell< ValueType, IndexType >::get_values(), gko::matrix::Coo< ValueType, IndexType >::get_values(), gko::matrix::Dense< ValueType >::get_values(), gko::Array< index_type >::operator=(), gko::matrix::Ell< ValueType, IndexType >::val_at(), and gko::matrix::Sellp< ValueType, IndexType >::val_at().

```
397 { return data_.get(); }
```

24.3.3.4 get_executor()

```
template<typename ValueType>
std::shared_ptr<const Executor> gko::Array< ValueType >::get_executor ( ) const [inline],
[noexcept]
```

Returns the [Executor](#) associated with the array.

Returns

the [Executor](#) associated with the array

Referenced by `gko::matrix::Hybrid< ValueType, IndexType >::strategy_type::compute_hybrid_config()`, and `gko::Array< index_type >::operator=()`.

```
414     {
415         return exec_;
416     }
```

24.3.3.5 get_num_elems()

```
template<typename ValueType>
size_type gko::Array< ValueType >::get_num_elems ( ) const [inline], [noexcept]
```

Returns the number of elements in the [Array](#).

Returns

the number of elements in the [Array](#)

Referenced by `gko::matrix::Coo< ValueType, IndexType >::apply2()`, `gko::matrix::Ell< ValueType, IndexType >::col_at()`, `gko::matrix::Hybrid< ValueType, IndexType >::imbalance_limit::compute_ell_num_stored_elements_per_row()`, `gko::matrix::Hybrid< ValueType, IndexType >::imbalance_bounded_limit::compute_ell_num_stored_elements_per_row()`, `gko::matrix::Hybrid< ValueType, IndexType >::strategy_type::compute_hybrid_config()`, `gko::matrix::Dense< ValueType >::create_submatrix()`, `gko::matrix::Hybrid< ValueType, IndexType >::strategy_type::get_coo_nnz()`, `gko::matrix::Ell< ValueType, IndexType >::get_num_stored_elements()`, `gko::matrix::Coo< ValueType, IndexType >::get_num_stored_elements()`, `gko::matrix::Sellp< ValueType, IndexType >::get_num_stored_elements()`, `gko::matrix::Dense< ValueType >::get_num_stored_elements()`, `gko::preconditioner::Jacobi< ValueType, IndexType >::get_num_stored_elements()`, and `gko::Array< index_type >::operator=()`.

```
388 { return num_elems; }
```

24.3.3.6 operator=() [1/2]

```
template<typename ValueType>
Array& gko::Array< ValueType >::operator= (
    const Array< ValueType > & other ) [inline]
```

Copies data from another array.

This does not invoke the constructors of the elements, instead they are copied as POD types.

The executor of this is preserved. In case this does not have an assigned executor, it will inherit the executor of other.

Parameters

<i>other</i>	the Array to copy from
--------------	--

Returns

this

```

286     {
287         if (&other == this) {
288             return *this;
289         }
290         if (exec_ == nullptr) {
291             exec_ = other.get_executor();
292             data_ = data_manager{nullptr, other.data_.get_deleter()};
293         }
294         if (other.get_executor() == nullptr) {
295             this->resize_and_reset(0);
296             return *this;
297         }
298         this->resize_and_reset(other.get_num_elems());
299         exec_->copy_from(other.get_executor().get(), num_elems_,
300                        other.get_const_data(), this->get_data());
301         return *this;
302     }

```

24.3.3.7 operator=() [2/2]

```

template<typename ValueType>
Array& gko::Array< ValueType >::operator= (
    Array< ValueType > && other ) [inline]

```

Moves data from another array.

This does not invoke the constructors of the elements, instead they are copied as POD types.

The executor of this is preserved. In case this does not have an assigned executor, it will inherit the executor of other.

Parameters

<i>other</i>	the Array to move data from
--------------	---

Returns

this

```

318     {
319         if (&other == this) {
320             return *this;
321         }
322         if (exec_ == nullptr) {
323             exec_ = other.get_executor();
324             data_ = data_manager{nullptr, other.data_.get_deleter()};
325         }
326         if (other.get_executor() == nullptr) {
327             this->resize_and_reset(0);
328             return *this;
329         }

```

```

330         if (exec_ == other.get_executor() &&
331             data_.get_deleter().target_type() != typeid(view_deleter)) {
332             // same device and not a view, only move the pointer
333             using std::swap;
334             swap(data_, other.data_);
335             swap(num_elems_, other.num_elems_);
336         } else {
337             // different device or a view, copy the data
338             *this = other;
339         }
340         return *this;
341     }

```

24.3.3.8 `resize_and_reset()`

```

template<typename ValueType>
void gko::Array< ValueType >::resize_and_reset (
    size_type num_elems ) [inline]

```

Resizes the array so it is able to hold the specified number of elements.

All data stored in the array will be lost.

If the [Array](#) is not assigned an executor, an exception will be thrown.

Parameters

<code>num_elems</code>	the amount of memory (expressed as the number of <code>value_type</code> elements) allocated on the Executor
------------------------	--

Referenced by `gko::Array< index_type >::operator=()`.

```

367     {
368         if (num_elems == num_elems_) {
369             return;
370         }
371         if (exec_ == nullptr) {
372             throw gko::NotSupported(__FILE__, __LINE__, __func__,
373                                     "gko::Executor (nullptr)");
374         }
375         num_elems_ = num_elems;
376         if (num_elems > 0) {
377             data_.reset(exec_->alloc<value_type>(num_elems));
378         } else {
379             data_.reset(nullptr);
380         }
381     }

```

24.3.3.9 `set_executor()`

```

template<typename ValueType>
void gko::Array< ValueType >::set_executor (
    std::shared_ptr< const Executor > exec ) [inline]

```

Changes the [Executor](#) of the [Array](#), moving the allocated data to the new [Executor](#).

Parameters

<i>exec</i>	the Executor where the data will be moved to
-------------	--

```

425     {
426         if (exec == exec_) {
427             // moving to the same executor, no-op
428             return;
429         }
430         Array tmp(std::move(exec));
431         tmp = *this;
432         exec_ = std::move(tmp.exec_);
433         data_ = std::move(tmp.data_);
434     }

```

24.3.3.10 view()

```

template<typename ValueType>
static Array gko::Array< ValueType >::view (
    std::shared_ptr< const Executor > exec,
    size_type num_elems,
    value_type * data ) [inline], [static]

```

Creates an [Array](#) from existing memory.

The [Array](#) does not take ownership of the memory, and will not deallocate it once it goes out of scope.

Parameters

<i>exec</i>	executor where data is located
<i>num_elems</i>	number of elements in data
<i>data</i>	chunk of memory used to create the array

Returns

an [Array](#) constructed from data

Referenced by gko::matrix::Dense< ValueType >::create_submatrix().

```

268     {
269         return Array(exec, num_elems, data, view_deleter{});
270     }

```

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

- ginkgo/core/base/array.hpp (8045ac75)

24.4 gko::matrix::Hybrid< ValueType, IndexType >::automatic Class Reference

automatic is a stratgy_type which decides the number of stored elements per row of the ell part automatically.

```
#include <ginkgo/core/matrix/hybrid.hpp>
```

Public Member Functions

- [automatic](#) ()
Creates an automatic strategy.
- [size_type compute_ell_num_stored_elements_per_row](#) (Array< [size_type](#) > *row_nnz) const override
Computes the number of stored elements per row of the ell part.

24.4.1 Detailed Description

```
template<typename ValueType = default_precision, typename IndexType = int32>
class gko::matrix::Hybrid< ValueType, IndexType >::automatic
```

automatic is a stratgy_type which decides the number of stored elements per row of the ell part automatically.

24.4.2 Member Function Documentation

24.4.2.1 compute_ell_num_stored_elements_per_row()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Hybrid< ValueType, IndexType >::automatic::compute_ell_num_stored_↵
elements_per_row (
    Array< size_type > * row_nnz ) const [inline], [override], [virtual]
```

Computes the number of stored elements per row of the ell part.

Parameters

<i>row_nnz</i>	the number of nonzeros of each row
----------------	------------------------------------

Returns

the number of stored elements per row of the ell part

Implements [gko::matrix::Hybrid< ValueType, IndexType >::strategy_type](#).

References [gko::matrix::Hybrid< ValueType, IndexType >::read\(\)](#), and [gko::matrix::Hybrid< ValueType, IndexType >::write\(\)](#).

```
325         {
326             return strategy_.compute_ell_num_stored_elements_per_row
327         (row_nnz);
327         }
```

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

- ginkgo/core/matrix/hybrid.hpp (3e51a52b)

24.5 gko::solver::Bicgstab< ValueType > Class Template Reference

BiCGSTAB or the Bi-Conjugate Gradient-Stabilized is a Krylov subspace solver.

```
#include <ginkgo/core/solver/bicgstab.hpp>
```

Public Member Functions

- `std::shared_ptr< const LinOp > get_system_matrix () const`
Gets the system operator (matrix) of the linear system.
- `std::shared_ptr< const LinOp > get_preconditioner () const override`
Returns the preconditioner operator used by the solver.

24.5.1 Detailed Description

```
template<typename ValueType = default_precision>
class gko::solver::Bicgstab< ValueType >
```

BiCGSTAB or the Bi-Conjugate Gradient-Stabilized is a Krylov subspace solver.

Being a generic solver, it is capable of solving general matrices, including non-s.p.d matrices. Though, the memory and the computational requirement of the BiCGSTAB solver are higher than of its s.p.d solver counterpart, it has the capability to solve generic systems. It was developed by stabilizing the BiCG method.

Template Parameters

<i>ValueType</i>	precision of the elements of the system matrix.
------------------	---

24.5.2 Member Function Documentation

24.5.2.1 [get_preconditioner\(\)](#)

```
template<typename ValueType = default_precision>
std::shared_ptr<const LinOp> gko::solver::Bicgstab< ValueType >::get\_preconditioner ( ) const
[inline], [override], [virtual]
```

Returns the preconditioner operator used by the solver.

Returns

the preconditioner operator used by the solver

Implements [gko::Preconditionable](#).

References [gko::stop::combine\(\)](#), [gko::PolymorphicObject::get_executor\(\)](#), [GKO_CREATE_FACTORY_PARAMETERS](#), [GKO_ENABLE_BUILD_METHOD](#), [GKO_ENABLE_LIN_OP_FACTORY](#), [GKO_FACTORY_PARAMETER](#), and [gko::transpose\(\)](#).

```

99     {
100         return preconditioner_;
101     }

```

24.5.2.2 get_system_matrix()

```

template<typename ValueType = default_precision>
std::shared_ptr<const LinOp> gko::solver::Bicgstab< ValueType >::get_system_matrix ( ) const
[inline]

```

Gets the system operator (matrix) of the linear system.

Returns

the system operator (matrix)

```

89     {
90         return system_matrix_;
91     }

```

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

- ginkgo/core/solver/bicgstab.hpp (f1a4eb68)

24.6 gko::preconditioner::block_interleaved_storage_scheme< IndexType > Struct Template Reference

Defines the parameters of the interleaved block storage scheme used by block-Jacobi blocks.

```
#include <ginkgo/core/preconditioner/jacobi.hpp>
```

Public Member Functions

- IndexType [get_group_size](#) () const noexcept
Returns the number of elements in the group.
- [size_type compute_storage_space](#) ([size_type](#) num_blocks) const noexcept
Computes the storage space required for the requested number of blocks.
- IndexType [get_group_offset](#) (IndexType block_id) const noexcept
Returns the offset of the group belonging to the block with the given ID.
- IndexType [get_block_offset](#) (IndexType block_id) const noexcept
Returns the offset of the block with the given ID within its group.
- IndexType [get_global_block_offset](#) (IndexType block_id) const noexcept
Returns the offset of the block with the given ID.
- IndexType [get_stride](#) () const noexcept
Returns the stride between columns of the block.

Public Attributes

- IndexType [block_offset](#)
The offset between consecutive blocks within the group.
- IndexType [group_offset](#)
The offset between two block groups.
- uint32 [group_power](#)
Then base 2 power of the group.

24.6.1 Detailed Description

```
template<typename IndexType>
struct gko::preconditioner::block_interleaved_storage_scheme< IndexType >
```

Defines the parameters of the interleaved block storage scheme used by block-Jacobi blocks.

Template Parameters

<i>IndexType</i>	type used for storing indices of the matrix
------------------	---

24.6.2 Member Function Documentation

24.6.2.1 compute_storage_space()

```
template<typename IndexType>
size_type gko::preconditioner::block_interleaved_storage_scheme< IndexType >::compute_storage↵
_space (
    size_type num_blocks ) const [inline], [noexcept]
```

Computes the storage space required for the requested number of blocks.

Parameters

<i>num_blocks</i>	the total number of blocks that needs to be stored
-------------------	--

Returns

the total memory (as the number of elements) that need to be allocated for the scheme

Note

To simplify using the method in situations where the number of blocks is not known, for a special input `size↵_type{} - 1` the method returns 0 to avoid overallocation of memory.

```

105     {
106         return (num_blocks + 1 == size_type{0})
107             ? size_type{0}
108             : ceildiv(num_blocks, this->get_group_size()) *
109         group_offset;
110     }

```

24.6.2.2 get_block_offset()

```

template<typename IndexType>
IndexType gko::preconditioner::block_interleaved_storage_scheme< IndexType >::get_block_offset
(
    IndexType block_id ) const [inline], [noexcept]

```

Returns the offset of the block with the given ID within its group.

Parameters

<i>block_id</i>	the ID of the block
-----------------	---------------------

Returns

the offset of the block with ID `block_id` within its group

Referenced by `gko::preconditioner::block_interleaved_storage_scheme< index_type >::get_global_block_offset()`.

```

131     {
132         return block_offset * (block_id & (this->get_group_size() - 1));
133     }

```

24.6.2.3 get_global_block_offset()

```

template<typename IndexType>
IndexType gko::preconditioner::block_interleaved_storage_scheme< IndexType >::get_global_↵
block_offset (
    IndexType block_id ) const [inline], [noexcept]

```

Returns the offset of the block with the given ID.

Parameters

<i>block_id</i>	the ID of the block
-----------------	---------------------

Returns

the offset of the block with ID `block_id`

```

144     {
145         return this->get_group_offset(block_id) +
146            this->get_block_offset(block_id);
147     }

```

24.6.2.4 get_group_offset()

```

template<typename IndexType>
IndexType gko::preconditioner::block_interleaved_storage_scheme< IndexType >::get_group_offset
(
    IndexType block_id ) const [inline], [noexcept]

```

Returns the offset of the group belonging to the block with the given ID.

Parameters

<i>block_id</i>	the ID of the block
-----------------	---------------------

Returns

the offset of the group belonging to block with ID `block_id`

Referenced by `gko::preconditioner::block_interleaved_storage_scheme< index_type >::get_global_block_offset()`.

```

119     {
120         return group_offset * (block_id >> group_power);
121     }

```

24.6.2.5 get_group_size()

```

template<typename IndexType>
IndexType gko::preconditioner::block_interleaved_storage_scheme< IndexType >::get_group_size (
) const [inline], [noexcept]

```

Returns the number of elements in the group.

Returns

the number of elements in the group

Referenced by `gko::preconditioner::block_interleaved_storage_scheme< index_type >::compute_storage_space()`, and `gko::preconditioner::block_interleaved_storage_scheme< index_type >::get_block_offset()`.

```

87     {
88         return one<IndexType>() << group_power;
89     }

```

24.6.2.6 get_stride()

```
template<typename IndexType>
IndexType gko::preconditioner::block_interleaved_storage_scheme< IndexType >::get_stride ( )
const [inline], [noexcept]
```

Returns the stride between columns of the block.

Returns

stride between columns of the block

```
155     {
156         return block_offset << group_power;
157     }
```

24.6.3 Member Data Documentation

24.6.3.1 group_power

```
template<typename IndexType>
uint32 gko::preconditioner::block_interleaved_storage_scheme< IndexType >::group_power
```

Then base 2 power of the group.

I.e. the group contains $1 \ll \text{group_power}$ elements.

Referenced by `gko::preconditioner::block_interleaved_storage_scheme< index_type >::get_group_offset()`, and `gko::preconditioner::block_interleaved_storage_scheme< index_type >::get_stride()`.

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

- `ginkgo/core/preconditioner/jacobi.hpp` (8045ac75)

24.7 gko::solver::Cg< ValueType > Class Template Reference

CG or the conjugate gradient method is an iterative type Krylov subspace method which is suitable for symmetric positive definite methods.

```
#include <ginkgo/core/solver/cg.hpp>
```

Public Member Functions

- `std::shared_ptr< const LinOp > get_system_matrix () const`
Gets the system operator (matrix) of the linear system.
- `std::shared_ptr< const LinOp > get_preconditioner () const override`
Returns the preconditioner operator used by the solver.

24.7.1 Detailed Description

```
template<typename ValueType = default_precision>
class gko::solver::Cg< ValueType >
```

CG or the conjugate gradient method is an iterative type Krylov subspace method which is suitable for symmetric positive definite methods.

Though this method performs very well for symmetric positive definite matrices, it is in general not suitable for general matrices.

The implementation in Ginkgo makes use of the merged kernel to make the best use of data locality. The inner operations in one iteration of CG are merged into 2 separate steps.

Template Parameters

<i>ValueType</i>	precision of matrix elements
------------------	------------------------------

24.7.2 Member Function Documentation

24.7.2.1 get_preconditioner()

```
template<typename ValueType = default_precision>
std::shared_ptr<const LinOp> gko::solver::Cg< ValueType >::get_preconditioner ( ) const [inline],
[override], [virtual]
```

Returns the preconditioner operator used by the solver.

Returns

the preconditioner operator used by the solver

Implements [gko::Preconditionable](#).

References [gko::stop::combine\(\)](#), [gko::PolymorphicObject::get_executor\(\)](#), [GKO_CREATE_FACTORY_PARAMETERS](#), [GKO_ENABLE_BUILD_METHOD](#), [GKO_ENABLE_LIN_OP_FACTORY](#), [GKO_FACTORY_PARAMETER](#), and [gko::transpose\(\)](#).

```
94     {
95         return preconditioner_;
96     }
```

24.7.2.2 `get_system_matrix()`

```
template<typename ValueType = default_precision>
std::shared_ptr<const LinOp> gko::solver::Cg< ValueType >::get_system_matrix ( ) const [inline]
```

Gets the system operator (matrix) of the linear system.

Returns

the system operator (matrix)

```
84     {
85         return system_matrix_;
86     }
```

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

- ginkgo/core/solver/cg.hpp (f1a4eb68)

24.8 `gko::solver::Cgs< ValueType >` Class Template Reference

CGS or the conjugate gradient square method is an iterative type Krylov subspace method which is suitable for general systems.

```
#include <ginkgo/core/solver/cgs.hpp>
```

Public Member Functions

- `std::shared_ptr< const LinOp > get_system_matrix () const`
Gets the system operator (matrix) of the linear system.
- `std::shared_ptr< const LinOp > get_preconditioner () const override`
Returns the preconditioner operator used by the solver.

24.8.1 Detailed Description

```
template<typename ValueType = default_precision>
class gko::solver::Cgs< ValueType >
```

CGS or the conjugate gradient square method is an iterative type Krylov subspace method which is suitable for general systems.

The implementation in Ginkgo makes use of the merged kernel to make the best use of data locality. The inner operations in one iteration of CGS are merged into 3 separate steps.

Template Parameters

<i>ValueType</i>	precision of matrix elements
------------------	------------------------------

24.8.2 Member Function Documentation

24.8.2.1 get_preconditioner()

```
template<typename ValueType = default_precision>
std::shared_ptr<const LinOp> gko::solver::Cgs< ValueType >::get_preconditioner ( ) const
[inline], [override], [virtual]
```

Returns the preconditioner operator used by the solver.

Returns

the preconditioner operator used by the solver

Implements [gko::Preconditionable](#).

References [gko::stop::combine\(\)](#), [gko::PolymorphicObject::get_executor\(\)](#), [GKO_CREATE_FACTORY_PARAMETERS](#), [GKO_ENABLE_BUILD_METHOD](#), [GKO_ENABLE_LIN_OP_FACTORY](#), [GKO_FACTORY_PARAMETER](#), and [gko::transpose\(\)](#).

```
91     {
92         return preconditioner_;
93     }
```

24.8.2.2 get_system_matrix()

```
template<typename ValueType = default_precision>
std::shared_ptr<const LinOp> gko::solver::Cgs< ValueType >::get_system_matrix ( ) const [inline]
```

Gets the system operator (matrix) of the linear system.

Returns

the system operator (matrix)

```
81     {
82         return system_matrix_;
83     }
```

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

- [ginkgo/core/solver/cgs.hpp \(f1a4eb68\)](#)

24.9 gko::matrix::Hybrid< ValueType, IndexType >::column_limit Class Reference

`column_limit` is a [strategy_type](#) which decides the number of stored elements per row of the ell part by specifying the number of columns.

```
#include <ginkgo/core/matrix/hybrid.hpp>
```

Public Member Functions

- `column_limit` ([size_type](#) num_column=0)
Creates a [column_limit](#) strategy.
- `size_type compute_ell_num_stored_elements_per_row` ([Array](#)< [size_type](#) > *row_nnz) const override
Computes the number of stored elements per row of the ell part.

24.9.1 Detailed Description

```
template<typename ValueType = default_precision, typename IndexType = int32>
class gko::matrix::Hybrid< ValueType, IndexType >::column_limit
```

`column_limit` is a [strategy_type](#) which decides the number of stored elements per row of the ell part by specifying the number of columns.

24.9.2 Constructor & Destructor Documentation

24.9.2.1 column_limit()

```
template<typename ValueType = default_precision, typename IndexType = int32>
gko::matrix::Hybrid< ValueType, IndexType >::column_limit (
    size\_type num_column = 0 ) [inline], [explicit]
```

Creates a [column_limit](#) strategy.

Parameters

<code>num_column</code>	the specified number of columns of the ell part
-------------------------	---

```
203         : num_columns_(num_column)
204         {}
```

24.9.3 Member Function Documentation

24.9.3.1 compute_ell_num_stored_elements_per_row()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Hybrid< ValueType, IndexType >::column_limit::compute_ell_num_stored_
elements_per_row (
    Array< size_type > * row_nnz ) const [inline], [override], [virtual]
```

Computes the number of stored elements per row of the ell part.

Parameters

<code>row_nnz</code>	the number of nonzeros of each row
----------------------	------------------------------------

Returns

the number of stored elements per row of the ell part

Implements [gko::matrix::Hybrid< ValueType, IndexType >::strategy_type](#).

```
208     {
209         return num_columns_;
210     }
```

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

- ginkgo/core/matrix/hybrid.hpp (3e51a52b)

24.10 gko::Combination< ValueType > Class Template Reference

The [Combination](#) class can be used to construct a linear combination of multiple linear operators 'c1 * op1 + c2 * op2 + ...

```
#include <ginkgo/core/base/combination.hpp>
```

Public Member Functions

- const std::vector< std::shared_ptr< const LinOp > > & [get_coefficients](#) () const noexcept
Returns a list of coefficients of the combination.
- const std::vector< std::shared_ptr< const LinOp > > & [get_operators](#) () const noexcept
Returns a list of operators of the combination.

24.10.1 Detailed Description

```
template<typename ValueType = default_precision>
class gko::Combination< ValueType >
```

The [Combination](#) class can be used to construct a linear combination of multiple linear operators 'c1 * op1 + c2 * op2 + ...

- ck * opk'.

Template Parameters

<i>ValueType</i>	precision of input and result vectors
------------------	---------------------------------------

24.10.2 Member Function Documentation

24.10.2.1 `get_coefficients()`

```
template<typename ValueType = default_precision>
const std::vector<std::shared_ptr<const LinOp> > & gko::Combination< ValueType >::get_↵
coefficients ( ) const [inline], [noexcept]
```

Returns a list of coefficients of the combination.

Returns

a list of coefficients

```
70     {
71         return coefficients_;
72     }
```

24.10.2.2 `get_operators()`

```
template<typename ValueType = default_precision>
const std::vector<std::shared_ptr<const LinOp> > & gko::Combination< ValueType >::get_↵
operators ( ) const [inline], [noexcept]
```

Returns a list of operators of the combination.

Returns

a list of operators

References `gko::one()`, and `gko::zero()`.

```
81     {
82         return operators_;
83     }
```

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

- `ginkgo/core/base/combination.hpp` (f1a4eb68)

24.11 gko::stop::Combined Class Reference

The [Combined](#) class is used to combine multiple criterions together through an OR operation.

```
#include <ginkgo/core/stop/combined.hpp>
```

24.11.1 Detailed Description

The [Combined](#) class is used to combine multiple criterions together through an OR operation.

The typical use case is to stop the iteration process if any of the criteria is fulfilled, e.g. a number of iterations, the relative residual norm has reached a threshold, etc.

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

- ginkgo/core/stop/combined.hpp (f1a4eb68)

24.12 gko::Composition< ValueType > Class Template Reference

The [Composition](#) class can be used to compose linear operators op_1, op_2, \dots, op_n and obtain the operator $op_1 * op_2 * \dots$

```
#include <ginkgo/core/base/composition.hpp>
```

Public Member Functions

- `const std::vector< std::shared_ptr< const LinOp > > & get_operators () const noexcept`
Returns a list of operators of the composition.

24.12.1 Detailed Description

```
template<typename ValueType = default_precision>
class gko::Composition< ValueType >
```

The [Composition](#) class can be used to compose linear operators op_1, op_2, \dots, op_n and obtain the operator $op_1 * op_2 * \dots$

- op_n .

Template Parameters

<i>ValueType</i>	precision of input and result vectors
------------------	---------------------------------------

24.12.2 Member Function Documentation

24.12.2.1 get_operators()

```
template<typename ValueType = default_precision>
const std::vector<std::shared_ptr<const LinOp> >& gko::Composition< ValueType >::get_↵
operators ( ) const [inline], [noexcept]
```

Returns a list of operators of the composition.

Returns

a list of operators

```
70     {
71         return operators_;
72     }
```

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

- ginkgo/core/base/composition.hpp (f1a4eb68)

24.13 gko::log::Convergence< ValueType > Class Template Reference

[Convergence](#) is a Logger which logs data strictly from the `criterion_check_completed` event.

```
#include <ginkgo/core/log/convergence.hpp>
```

Public Member Functions

- const [size_type](#) & [get_num_iterations](#) () const noexcept
Returns the number of iterations.
- const LinOp * [get_residual](#) () const noexcept
Returns the residual.
- const LinOp * [get_residual_norm](#) () const noexcept
Returns the residual norm.

Static Public Member Functions

- static std::unique_ptr< [Convergence](#) > [create](#) (std::shared_ptr< const [Executor](#) > exec, const mask_type &enabled_events=Logger::all_events_mask)
Creates a convergence logger.

24.13.1 Detailed Description

```
template<typename ValueType = default_precision>
class gko::log::Convergence< ValueType >
```

[Convergence](#) is a Logger which logs data strictly from the `criterion_check_completed` event.

The purpose of this logger is to give a simple access to standard data generated by the solver once it has converged with minimal overhead.

This logger also computes the residual norm from the residual when the residual norm was not available. This can add some slight overhead.

24.13.2 Member Function Documentation

24.13.2.1 `create()`

```
template<typename ValueType = default_precision>
static std::unique_ptr<Convergence> gko::log::Convergence< ValueType >::create (
    std::shared_ptr< const Executor > exec,
    const mask_type & enabled_events = Logger::all_events_mask ) [inline], [static]
```

Creates a convergence logger.

This dynamically allocates the memory, constructs the object and returns an `std::unique_ptr` to this object.

Parameters

<code>exec</code>	the executor
<code>enabled_events</code>	the events enabled for this logger. By default all events.

Returns

an `std::unique_ptr` to the the constructed object

```
94     {
95         return std::unique_ptr<Convergence>(
96             new Convergence(exec, enabled_events));
97     }
```

24.13.2.2 `get_num_iterations()`

```
template<typename ValueType = default_precision>
const size_type& gko::log::Convergence< ValueType >::get_num_iterations ( ) const [inline],
[noexcept]
```

Returns the number of iterations.

Returns

the number of iterations

```

105     {
106         return num_iterations_;
107     }

```

24.13.2.3 get_residual()

```

template<typename ValueType = default_precision>
const LinOp* gko::log::Convergence< ValueType >::get_residual ( ) const [inline], [noexcept]

```

Returns the residual.

Returns

the residual

```

114 { return residual_.get(); }

```

24.13.2.4 get_residual_norm()

```

template<typename ValueType = default_precision>
const LinOp* gko::log::Convergence< ValueType >::get_residual_norm ( ) const [inline], [noexcept]

```

Returns the residual norm.

Returns

the residual norm

```

122     {
123         return residual_norm_.get();
124     }

```

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

- ginkgo/core/log/convergence.hpp (f1a4eb68)

24.14 gko::ConvertibleTo< ResultType > Class Template Reference

[ConvertibleTo](#) interface is used to mark that the implementer can be converted to the object of ResultType.

```
#include <ginkgo/core/base/polymorphic_object.hpp>
```


Public Member Functions

- virtual void [convert_to](#) (result_type *result) const =0
Converts the implementer to an object of type result_type.
- virtual void [move_to](#) (result_type *result)=0
Converts the implementer to an object of type result_type by moving data from this object.

24.14.1 Detailed Description

```
template<typename ResultType>
class gko::ConvertibleTo< ResultType >
```

[ConvertibleTo](#) interface is used to mark that the implementer can be converted to the object of ResultType.

This interface is used to enable conversions between polymorphic objects. To mark that an object of type *U* can be converted to an object of type *V*, *U* should implement [ConvertibleTo<V>](#). Then, the implementation of [PolymorphicObject::copy_from](#) automatically generated by [EnablePolymorphicObject](#) mixin will use RTTI to figure out that *U* implements the interface and convert it using the [convert_to](#) / [move_to](#) methods of the interface.

As an example, the following function:

```
{c++}
void my_function(const U *u, V *v) {
    v->copy_from(u);
}
```

will convert object *u* to object *v* by checking that *u* can be dynamically casted to [ConvertibleTo<V>](#), and calling [ConvertibleTo<V>::convert_to\(V*\)](#) to do the actual conversion.

In case *u* is passed as a unique_ptr, call to [convert_to](#) will be replaced by a call to [move_to](#) and trigger move semantics.

Template Parameters

<i>ResultType</i>	the type to which the implementer can be converted to, has to be a subclass of PolymorphicObject
-------------------	--

24.14.2 Member Function Documentation

24.14.2.1 convert_to()

```
template<typename ResultType>
virtual void gko::ConvertibleTo< ResultType >::convert\_to (
    result_type * result ) const [pure virtual]
```

Converts the implementer to an object of type result_type.

Parameters

<i>result</i>	the object used to store the result of the conversion
---------------	---

Implemented in `gko::EnablePolymorphicAssignment< ConcreteType, ResultType >`, `gko::EnablePolymorphicAssignment< Dense< ValueType >, gko::EnablePolymorphicAssignment< Hybrid< ValueType, IndexType >, gko::EnablePolymorphicAssignment< Identity< ValueType >, gko::EnablePolymorphicAssignment< ConcreteLinOp >, gko::EnablePolymorphicAssignment< Composition< ValueType >, gko::EnablePolymorphicAssignment< Bicgstab< ValueType >, gko::EnablePolymorphicAssignment< Combination< ValueType >, gko::EnablePolymorphicAssignment< Gmres< ValueType >, gko::EnablePolymorphicAssignment< Csr< ValueType, IndexType >, gko::EnablePolymorphicAssignment< Ir< ValueType >, gko::EnablePolymorphicAssignment< Coo< ValueType, IndexType >, gko::EnablePolymorphicAssignment< Fcg< ValueType >, gko::EnablePolymorphicAssignment< ConcreteFactory >, gko::EnablePolymorphicAssignment< Cgs< ValueType >, gko::EnablePolymorphicAssignment< Ell< ValueType, IndexType >, gko::EnablePolymorphicAssignment< Jacobi< ValueType, IndexType >, gko::EnablePolymorphicAssignment< Cg< ValueType >, gko::EnablePolymorphicAssignment< Sellp< ValueType, IndexType >, and gko::preconditioner::Jacobi< ValueType, IndexType >.`

24.14.2.2 `move_to()`

```
template<typename ResultType>
virtual void gko::ConvertibleTo< ResultType >::move_to (
    result_type * result ) [pure virtual]
```

Converts the implementer to an object of type `result_type` by moving data from this object.

This method is used when the implementer is a temporary object, and move semantics can be used.

Parameters

<i>result</i>	the object used to emplace the result of the conversion
---------------	---

Note

`ConvertibleTo::move_to` can be implemented by simply calling `ConvertibleTo::convert_to`. However, this operation can often be optimized by exploiting the fact that implementer's data can be moved to the result.

Implemented in `gko::EnablePolymorphicAssignment< ConcreteType, ResultType >`, `gko::EnablePolymorphicAssignment< Dense< ValueType >, gko::EnablePolymorphicAssignment< Hybrid< ValueType, IndexType >, gko::EnablePolymorphicAssignment< Identity< ValueType >, gko::EnablePolymorphicAssignment< ConcreteLinOp >, gko::EnablePolymorphicAssignment< Composition< ValueType >, gko::EnablePolymorphicAssignment< Bicgstab< ValueType >, gko::EnablePolymorphicAssignment< Combination< ValueType >, gko::EnablePolymorphicAssignment< Gmres< ValueType >, gko::EnablePolymorphicAssignment< Csr< ValueType, IndexType >, gko::EnablePolymorphicAssignment< Ir< ValueType >, gko::EnablePolymorphicAssignment< Coo< ValueType, IndexType >, gko::EnablePolymorphicAssignment< Fcg< ValueType >, gko::EnablePolymorphicAssignment< ConcreteFactory >, gko::EnablePolymorphicAssignment< Cgs< ValueType >, gko::EnablePolymorphicAssignment< Ell< ValueType, IndexType >, gko::EnablePolymorphicAssignment< Jacobi< ValueType, IndexType >, gko::EnablePolymorphicAssignment< Cg< ValueType >, gko::EnablePolymorphicAssignment< Sellp< ValueType, IndexType >, and gko::preconditioner::Jacobi< ValueType, IndexType >.`

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

- `ginkgo/core/base/polymorphic_object.hpp` (4bde4271)

24.15 gko::matrix::Coo< ValueType, IndexType > Class Template Reference

COO stores a matrix in the coordinate matrix format.

```
#include <ginkgo/core/matrix/coo.hpp>
```

Public Member Functions

- void [read](#) (const [mat_data](#) &data) override
Reads a matrix from a [matrix_data](#) structure.
- void [write](#) ([mat_data](#) &data) const override
Writes a matrix to a [matrix_data](#) structure.
- value_type * [get_values](#) () noexcept
Returns the values of the matrix.
- const value_type * [get_const_values](#) () const noexcept
Returns the values of the matrix.
- index_type * [get_col_idxs](#) () noexcept
Returns the column indexes of the matrix.
- const index_type * [get_const_col_idxs](#) () const noexcept
Returns the column indexes of the matrix.
- index_type * [get_row_idxs](#) () noexcept
Returns the row indexes of the matrix.
- const index_type * [get_const_row_idxs](#) () const noexcept
- size_type [get_num_stored_elements](#) () const noexcept
Returns the number of elements explicitly stored in the matrix.
- LinOp * [apply2](#) (const LinOp *b, LinOp *x)
Applies [Coo](#) matrix axpy to a vector (or a sequence of vectors).
- const LinOp * [apply2](#) (const LinOp *b, LinOp *x) const
- LinOp * [apply2](#) (const LinOp *alpha, const LinOp *b, LinOp *x)
*Performs the operation $x = \text{alpha} * \text{Coo} * b + x$.*
- const LinOp * [apply2](#) (const LinOp *alpha, const LinOp *b, LinOp *x) const
*Performs the operation $x = \text{alpha} * \text{Coo} * b + x$.*

24.15.1 Detailed Description

```
template<typename ValueType = default_precision, typename IndexType = int32>
class gko::matrix::Coo< ValueType, IndexType >
```

COO stores a matrix in the coordinate matrix format.

The nonzero elements are stored in an array row-wise (but not necessarily sorted by column index within a row). Two extra arrays contain the row and column indexes of each nonzero element of the matrix.

Template Parameters

<i>ValueType</i>	precision of matrix elements
<i>IndexType</i>	precision of matrix indexes

24.15.2 Member Function Documentation

24.15.2.1 `apply2()` [1/4]

```
template<typename ValueType = default_precision, typename IndexType = int32>
LinOp* gko::matrix::Coo< ValueType, IndexType >::apply2 (
    const LinOp * b,
    LinOp * x ) [inline]
```

Applies `Coo` matrix axpy to a vector (or a sequence of vectors).

Performs the operation $x = \text{Coo} * b + x$

Parameters

<i>b</i>	the input vector(s) on which the operator is applied
<i>x</i>	the output vector(s) where the result is stored

Returns

`this`

References `gko::PolymorphicObject::get_executor()`, and `gko::make_temporary_clone()`.

```
182     {
183         this->validate_application_parameters(b, x);
184         auto exec = this->get_executor();
185         this->apply2_impl(make_temporary_clone(exec, b).get(),
186                        make_temporary_clone(exec, x).get());
187         return this;
188     }
```

24.15.2.2 `apply2()` [2/4]

```
template<typename ValueType = default_precision, typename IndexType = int32>
const LinOp* gko::matrix::Coo< ValueType, IndexType >::apply2 (
    const LinOp * b,
    LinOp * x ) const [inline]
```

References `gko::PolymorphicObject::get_executor()`, and `gko::make_temporary_clone()`.

```
194     {
195         this->validate_application_parameters(b, x);
196         auto exec = this->get_executor();
197         this->apply2_impl(make_temporary_clone(exec, b).get(),
198                        make_temporary_clone(exec, x).get());
199         return this;
200     }
```

24.15.2.3 apply2() [3/4]

```
template<typename ValueType = default_precision, typename IndexType = int32>
LinOp* gko::matrix::Coo< ValueType, IndexType >::apply2 (
    const LinOp * alpha,
    const LinOp * b,
    LinOp * x ) [inline]
```

Performs the operation $x = \text{alpha} * \text{Coo} * b + x$.

Parameters

<i>alpha</i>	scaling of the result of <code>Coo * b</code>
<i>b</i>	vector(s) on which the operator is applied
<i>x</i>	output vector(s)

Returns

this

References `gko::PolymorphicObject::get_executor()`, and `gko::make_temporary_clone()`.

```
212 {
213     this->validate_application_parameters(b, x);
214     GKO_ASSERT_EQUAL_DIMENSIONS(alpha, dim<2>(1, 1));
215     auto exec = this->get_executor();
216     this->apply2_impl(make_temporary_clone(exec, alpha).get(),
217                     make_temporary_clone(exec, b).get(),
218                     make_temporary_clone(exec, x).get());
219     return this;
220 }
```

24.15.2.4 apply2() [4/4]

```
template<typename ValueType = default_precision, typename IndexType = int32>
const LinOp* gko::matrix::Coo< ValueType, IndexType >::apply2 (
    const LinOp * alpha,
    const LinOp * b,
    LinOp * x ) const [inline]
```

Performs the operation $x = \text{alpha} * \text{Coo} * b + x$.

Parameters

<i>alpha</i>	scaling of the result of <code>Coo * b</code>
<i>b</i>	vector(s) on which the operator is applied
<i>x</i>	output vector(s)

Returns

this

References `gko::PolymorphicObject::get_executor()`, `gko::Array< ValueType >::get_num_elems()`, and `gko::Array< ValueType >::make_temporary_clone()`.

```

226     {
227         this->validate_application_parameters(b, x);
228         GKO_ASSERT_EQUAL_DIMENSIONS(alpha, dim<2>(1, 1));
229         auto exec = this->get_executor();
230         this->apply2_impl(make_temporary_clone(exec, alpha).get(),
231                        make_temporary_clone(exec, b).get(),
232                        make_temporary_clone(exec, x).get());
233         return this;
234     }

```

24.15.2.5 get_col_idxs()

```

template<typename ValueType = default_precision, typename IndexType = int32>
index_type* gko::matrix::Coo< ValueType, IndexType >::get_col_idxs ( ) [inline], [noexcept]

```

Returns the column indexes of the matrix.

Returns

the column indexes of the matrix.

References `gko::Array< ValueType >::get_data()`.

```

128 { return col_idxs_.get_data(); }

```

24.15.2.6 get_const_col_idxs()

```

template<typename ValueType = default_precision, typename IndexType = int32>
const index_type* gko::matrix::Coo< ValueType, IndexType >::get_const_col_idxs ( ) const [inline],
[noexcept]

```

Returns the column indexes of the matrix.

Returns

the column indexes of the matrix.

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

References `gko::Array< ValueType >::get_const_data()`.

```

138     {
139         return col_idxs_.get_const_data();
140     }

```

24.15.2.7 get_const_row_idxs()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const index_type* gko::matrix::Coo< ValueType, IndexType >::get_const_row_idxs ( ) const [inline],
[noexcept]
```

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

References gko::Array< ValueType >::get_const_data().

```
157     {
158         return row_idxs_.get_const_data();
159     }
```

24.15.2.8 get_const_values()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const value_type* gko::matrix::Coo< ValueType, IndexType >::get_const_values ( ) const [inline],
[noexcept]
```

Returns the values of the matrix.

Returns

the values of the matrix.

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

References gko::Array< ValueType >::get_const_data().

```
119     {
120         return values_.get_const_data();
121     }
```

24.15.2.9 get_num_stored_elements()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Coo< ValueType, IndexType >::get_num_stored_elements ( ) const [inline],
[noexcept]
```

Returns the number of elements explicitly stored in the matrix.

Returns

the number of elements explicitly stored in the matrix

References `gko::Array< ValueType >::get_num_elems()`.

```
167 {
168     return values_.get_num_elems();
169 }
```

24.15.2.10 get_row_idxs()

```
template<typename ValueType = default_precision, typename IndexType = int32>
index_type* gko::matrix::Coo< ValueType, IndexType >::get_row_idxs ( ) [inline], [noexcept]
```

Returns the row indexes of the matrix.

Returns

the row indexes of the matrix.

References `gko::Array< ValueType >::get_data()`.

```
147 { return row_idxs_.get_data(); }
```

24.15.2.11 get_values()

```
template<typename ValueType = default_precision, typename IndexType = int32>
value_type* gko::matrix::Coo< ValueType, IndexType >::get_values ( ) [inline], [noexcept]
```

Returns the values of the matrix.

Returns

the values of the matrix.

References `gko::Array< ValueType >::get_data()`.

```
109 { return values_.get_data(); }
```

24.15.2.12 read()

```
template<typename ValueType = default_precision, typename IndexType = int32>
void gko::matrix::Coo< ValueType, IndexType >::read (
    const mat_data & data ) [override], [virtual]
```

Reads a matrix from a `mat_data` structure.

Parameters

<i>data</i>	the matrix_data structure
-------------	---

Implements [gko::ReadableFromMatrixData< ValueType, IndexType >](#).

24.15.2.13 write()

```
template<typename ValueType = default_precision, typename IndexType = int32>
void gko::matrix::Coo< ValueType, IndexType >::write (
    mat_data & data ) const [override], [virtual]
```

Writes a matrix to a [matrix_data](#) structure.

Parameters

<i>data</i>	the matrix_data structure
-------------	---

Implements [gko::WritableToMatrixData< ValueType, IndexType >](#).

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

- ginkgo/core/matrix/coo.hpp (5545d209)

24.16 gko::copy_back_deleter< T > Class Template Reference

A [copy_back_deleter](#) is a type of deleter that copies the data to an internally referenced object before performing the deletion.

```
#include <ginkgo/core/base/utils.hpp>
```

Public Member Functions

- [copy_back_deleter](#) (pointer original)
Creates a new deleter object.
- void [operator\(\)](#) (pointer ptr) const
Deletes the object.

24.16.1 Detailed Description

```
template<typename T>
class gko::copy_back_deleter< T >
```

A [copy_back_deleter](#) is a type of deleter that copies the data to an internally referenced object before performing the deletion.

The deleter will use the `copy_from` method to perform the copy, and then delete the passed object using the `delete` keyword. This kind of deleter is useful when temporarily copying an object with the intent of copying it back once it goes out of scope.

There is also a specialization for constant objects that does not perform the copy, since a constant object couldn't have been changed.

Template Parameters

<i>T</i>	the type of object being deleted
----------	----------------------------------

24.16.2 Constructor & Destructor Documentation

24.16.2.1 `copy_back_deleter()`

```
template<typename T >
gko::copy_back_deleter< T >::copy_back_deleter (
    pointer original ) [inline]
```

Creates a new deleter object.

Parameters

<i>original</i>	the origin object where the data will be copied before deletion
-----------------	---

```
372 : original_{original} {}
```

24.16.3 Member Function Documentation

24.16.3.1 `operator()()`

```
template<typename T >
void gko::copy_back_deleter< T >::operator() (
    pointer ptr ) const [inline]
```

Deletes the object.

Parameters

<i>ptr</i>	pointer to the object being deleted
------------	-------------------------------------

```
380     {
381         original_>copy_from(ptr);
382         delete ptr;
383     }
```

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

- ginkgo/core/base/utils.hpp (4bde4271)

24.17 gko::stop::Criterion Class Reference

The [Criterion](#) class is a base class for all stopping criteria.

```
#include <ginkgo/core/stop/criterion.hpp>
```

Classes

- class [Updater](#)

The [Updater](#) class serves for convenient argument passing to the [Criterion](#)'s check function.

Public Member Functions

- [Updater](#) [update](#) ()
Returns the updater object.
- bool [check](#) (uint8 stoppingId, bool setFinalized, [Array](#)< [stopping_status](#) > *stop_status, bool *one_changed, const [Updater](#) &updater)
This checks whether convergence was reached for a certain criterion.

24.17.1 Detailed Description

The [Criterion](#) class is a base class for all stopping criteria.

It contains a factory to instantiate criteria. It is up to each specific stopping criterion to decide what to do with the data that is passed to it.

Note that depending on the criterion, convergence may not have happened after stopping.

24.17.2 Member Function Documentation

24.17.2.1 [check\(\)](#)

```
bool gko::stop::Criterion::check (  
    uint8 stoppingId,  
    bool setFinalized,  
    Array< stopping\_status > * stop_status,  
    bool * one_changed,  
    const Updater & updater ) [inline]
```

This checks whether convergence was reached for a certain criterion.

The actual implantation of the criterion goes here.

Parameters

<i>stoppingId</i>	id of the stopping criterion
<i>setFinalized</i>	Controls if the current version should count as finalized or not
<i>stop_status</i>	status of the stopping criterion
<i>one_changed</i>	indicates if one vector's status changed
<i>updater</i>	the Updater object containing all the information

Returns

whether convergence was completely reached

Referenced by `gko::stop::Criterion::Updater::check()`.

```

152     {
153         this->template log<log::Logger::criterion_check_started>(
154             this, updater.num_iterations_, updater.residual_,
155             updater.residual_norm_, updater.solution_, stoppingId,
156             setFinalized);
157         auto all_converged = this->check_impl(
158             stoppingId, setFinalized, stop_status, one_changed, updater);
159         this->template log<log::Logger::criterion_check_completed>(
160             this, updater.num_iterations_, updater.residual_,
161             updater.residual_norm_, updater.solution_, stoppingId, setFinalized,
162             stop_status, *one_changed, all_converged);
163         return all_converged;
164     }

```

24.17.2.2 update()

[Updater](#) `gko::stop::Criterion::update ()` [inline]

Returns the updater object.

Returns

the updater object

```

134 { return {this}; }

```

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

- `ginkgo/core/stop/criterion.hpp` (f0a50f96)

24.18 gko::log::criterion_data Struct Reference

Struct representing Criterion related data.

```
#include <ginkgo/core/log/record.hpp>
```

24.18.1 Detailed Description

Struct representing Criterion related data.

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

- ginkgo/core/log/record.hpp (f0a50f96)

24.19 gko::stop::CriterionArgs Struct Reference

This struct is used to pass parameters to the `EnableDefaultCriterionFactoryCriterionFactory::generate()` method.

```
#include <ginkgo/core/stop/criterion.hpp>
```

24.19.1 Detailed Description

This struct is used to pass parameters to the `EnableDefaultCriterionFactoryCriterionFactory::generate()` method.

It is the `ComponentsType` of `CriterionFactory`.

Note

Dependly on the use case, some of these parameters can be `nullptr` as only some stopping criterion require them to be set. An example is the [ResidualNormReduction](#) which really requires the `initial_residual` to be set.

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

- ginkgo/core/stop/criterion.hpp (f0a50f96)

24.20 gko::matrix::Csr< ValueType, IndexType > Class Template Reference

CSR is a matrix format which stores only the nonzero coefficients by compressing each row of the matrix (compressed sparse row format).

```
#include <ginkgo/core/matrix/csr.hpp>
```

Public Member Functions

- void [read](#) (const [mat_data](#) &data) override
Reads a matrix from a [matrix_data](#) structure.
- void [write](#) ([mat_data](#) &data) const override
Writes a matrix to a [matrix_data](#) structure.
- std::unique_ptr< LinOp > [transpose](#) () const override
Returns a LinOp representing the transpose of the [Transposable](#) object.
- std::unique_ptr< LinOp > [conj_transpose](#) () const override
Returns a LinOp representing the conjugate transpose of the [Transposable](#) object.
- void [sort_by_column_index](#) ()
Sorts all (value, col_idx) pairs in each row by column index.
- value_type * [get_values](#) () noexcept
Returns the values of the matrix.
- const value_type * [get_const_values](#) () const noexcept
Returns the values of the matrix.
- index_type * [get_col_idxs](#) () noexcept
Returns the column indexes of the matrix.
- const index_type * [get_const_col_idxs](#) () const noexcept
Returns the column indexes of the matrix.
- index_type * [get_row_ptrs](#) () noexcept
Returns the row pointers of the matrix.
- const index_type * [get_const_row_ptrs](#) () const noexcept
Returns the row pointers of the matrix.
- index_type * [get_srow](#) () noexcept
Returns the starting rows.
- const index_type * [get_const_srow](#) () const noexcept
Returns the starting rows.
- size_type [get_num_srow_elements](#) () const noexcept
Returns the number of the srow stored elements (involved warps)
- size_type [get_num_stored_elements](#) () const noexcept
Returns the number of elements explicitly stored in the matrix.
- std::shared_ptr< strategy_type > [get_strategy](#) () const noexcept
Returns the strategy.

24.20.1 Detailed Description

```
template<typename ValueType = default_precision, typename IndexType = int32>
class gko::matrix::Csr< ValueType, IndexType >
```

CSR is a matrix format which stores only the nonzero coefficients by compressing each row of the matrix (compressed sparse row format).

The nonzero elements are stored in a 1D array row-wise, and accompanied with a row pointer array which stores the starting index of each row. An additional column index array is used to identify the column of each nonzero element.

Template Parameters

<i>ValueType</i>	precision of matrix elements
<i>IndexType</i>	precision of matrix indexes

24.20.2 Member Function Documentation

24.20.2.1 conj_transpose()

```
template<typename ValueType = default_precision, typename IndexType = int32>
std::unique_ptr<LinOp> gko::matrix::Csr< ValueType, IndexType >::conj_transpose ( ) const
[override], [virtual]
```

Returns a LinOp representing the conjugate transpose of the [Transposable](#) object.

Returns

a pointer to the new conjugate transposed object

Implements [gko::Transposable](#).

24.20.2.2 get_col_idxs()

```
template<typename ValueType = default_precision, typename IndexType = int32>
index_type* gko::matrix::Csr< ValueType, IndexType >::get_col_idxs ( ) [inline], [noexcept]
```

Returns the column indexes of the matrix.

Returns

the column indexes of the matrix.

```
347 { return col_idxs_.get_data(); }
```

24.20.2.3 get_const_col_idxs()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const index_type* gko::matrix::Csr< ValueType, IndexType >::get_const_col_idxs ( ) const [inline],
[noexcept]
```

Returns the column indexes of the matrix.

Returns

the column indexes of the matrix.

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

```
357 {
358     return col_idxs_.get_const_data();
359 }
```

24.20.2.4 get_const_row_ptrs()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const index_type* gko::matrix::Csr< ValueType, IndexType >::get_const_row_ptrs ( ) const [inline],
[noexcept]
```

Returns the row pointers of the matrix.

Returns

the row pointers of the matrix.

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

```
376     {
377         return row_ptrs_.get_const_data();
378     }
```

24.20.2.5 get_const_srow()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const index_type* gko::matrix::Csr< ValueType, IndexType >::get_const_srow ( ) const [inline],
[noexcept]
```

Returns the starting rows.

Returns

the starting rows.

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

```
395     {
396         return srow_.get_const_data();
397     }
```


24.20.2.6 get_const_values()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const value_type* gko::matrix::Csr< ValueType, IndexType >::get_const_values ( ) const [inline],
[noexcept]
```

Returns the values of the matrix.

Returns

the values of the matrix.

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

```
338     {
339         return values_.get_const_data();
340     }
```

24.20.2.7 get_num_srow_elements()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Csr< ValueType, IndexType >::get_num_srow_elements ( ) const [inline],
[noexcept]
```

Returns the number of the srow stored elements (involved warps)

Returns

the number of the srow stored elements (involved warps)

```
405     {
406         return srow_.get_num_elems();
407     }
```

24.20.2.8 get_num_stored_elements()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Csr< ValueType, IndexType >::get_num_stored_elements ( ) const [inline],
[noexcept]
```

Returns the number of elements explicitly stored in the matrix.

Returns

the number of elements explicitly stored in the matrix

```
415     {
416         return values_.get_num_elems();
417     }
```

24.20.2.9 get_row_ptrs()

```
template<typename ValueType = default_precision, typename IndexType = int32>
index_type* gko::matrix::Csr< ValueType, IndexType >::get_row_ptrs ( ) [inline], [noexcept]
```

Returns the row pointers of the matrix.

Returns

the row pointers of the matrix.

```
366 { return row_ptrs_.get_data(); }
```

24.20.2.10 get_srow()

```
template<typename ValueType = default_precision, typename IndexType = int32>
index_type* gko::matrix::Csr< ValueType, IndexType >::get_srow ( ) [inline], [noexcept]
```

Returns the starting rows.

Returns

the starting rows.

```
385 { return srow_.get_data(); }
```

24.20.2.11 get_strategy()

```
template<typename ValueType = default_precision, typename IndexType = int32>
std::shared_ptr<strategy_type> gko::matrix::Csr< ValueType, IndexType >::get_strategy ( )
const [inline], [noexcept]
```

Returns the strategy.

Returns

the strategy

```
424 {
425     return strategy_;
426 }
```

24.20.2.12 get_values()

```
template<typename ValueType = default_precision, typename IndexType = int32>
value_type* gko::matrix::Csr< ValueType, IndexType >::get_values ( ) [inline], [noexcept]
```

Returns the values of the matrix.

Returns

the values of the matrix.

```
328 { return values_.get_data(); }
```

24.20.2.13 read()

```
template<typename ValueType = default_precision, typename IndexType = int32>
void gko::matrix::Csr< ValueType, IndexType >::read (
    const mat_data & data ) [override], [virtual]
```

Reads a matrix from a [matrix_data](#) structure.

Parameters

<i>data</i>	the matrix_data structure
-------------	---

Implements [gko::ReadableFromMatrixData< ValueType, IndexType >](#).

24.20.2.14 transpose()

```
template<typename ValueType = default_precision, typename IndexType = int32>
std::unique_ptr<LinOp> gko::matrix::Csr< ValueType, IndexType >::transpose ( ) const [override],
[virtual]
```

Returns a LinOp representing the transpose of the [Transposable](#) object.

Returns

a pointer to the new transposed object

Implements [gko::Transposable](#).

24.20.2.15 write()

```
template<typename ValueType = default_precision, typename IndexType = int32>
void gko::matrix::Csr< ValueType, IndexType >::write (
    mat_data & data ) const [override], [virtual]
```

Writes a matrix to a [matrix_data](#) structure.

Parameters

<i>data</i>	the matrix_data structure
-------------	---

Implements [gko::WritableToMatrixData< ValueType, IndexType >](#).

The documentation for this class was generated from the following files:

- [ginkgo/core/matrix/coo.hpp](#) (5545d209)
- [ginkgo/core/matrix/csr.hpp](#) (ec6dbb34)

24.21 gko::CublasError Class Reference

[CublasError](#) is thrown when a cuBLAS routine throws a non-zero error code.

```
#include <ginkgo/core/base/exception.hpp>
```

Public Member Functions

- [CublasError](#) (const std::string &file, int line, const std::string &func, [int64](#) error_code)
Initializes a cuBLAS error.

24.21.1 Detailed Description

[CublasError](#) is thrown when a cuBLAS routine throws a non-zero error code.

24.21.2 Constructor & Destructor Documentation

24.21.2.1 CublasError()

```
gko::CublasError::CublasError (
    const std::string & file,
    int line,
    const std::string & func,
    int64 error_code ) [inline]
```

Initializes a cuBLAS error.

Parameters

<i>file</i>	The name of the offending source file
<i>line</i>	The source code line number where the error occurred
<i>func</i>	The name of the cuBLAS routine that failed
<i>error_code</i>	The resulting cuBLAS error code

```

208         : Error(file, line, func + ": " + get_error(error_code))
209     {}

```

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

- ginkgo/core/base/exception.hpp (8045ac75)

24.22 gko::CudaError Class Reference

[CudaError](#) is thrown when a CUDA routine throws a non-zero error code.

```
#include <ginkgo/core/base/exception.hpp>
```

Public Member Functions

- [CudaError](#) (const std::string &file, int line, const std::string &func, [int64](#) error_code)
Initializes a CUDA error.

24.22.1 Detailed Description

[CudaError](#) is thrown when a CUDA routine throws a non-zero error code.

24.22.2 Constructor & Destructor Documentation

24.22.2.1 CudaError()

```

gko::CudaError::CudaError (
    const std::string & file,
    int line,
    const std::string & func,
    int64 error_code ) [inline]

```

Initializes a CUDA error.

Parameters

<i>file</i>	The name of the offending source file
<i>line</i>	The source code line number where the error occurred
<i>func</i>	The name of the CUDA routine that failed
<i>error_code</i>	The resulting CUDA error code

```

186         : Error(file, line, func + ": " + get_error(error_code))
187     {}

```

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

- ginkgo/core/base/exception.hpp (8045ac75)

24.23 gko::CudaExecutor Class Reference

This is the [Executor](#) subclass which represents the CUDA device.

```
#include <ginkgo/core/base/executor.hpp>
```

Public Member Functions

- `std::shared_ptr< Executor > get_master ()` noexcept override
Returns the master [OmpExecutor](#) of this [Executor](#).
- `std::shared_ptr< const Executor > get_master ()` const noexcept override
Returns the master [OmpExecutor](#) of this [Executor](#).
- `void synchronize ()` const override
Synchronize the operations launched on the executor with its master.
- `void run (const Operation &op)` const override
Runs the specified [Operation](#) using this [Executor](#).
- `int get_device_id ()` const noexcept
Get the CUDA device id of the device associated to this executor.
- `int get_num_cores_per_sm ()` const noexcept
Get the number of cores per SM of this executor.
- `int get_num_multiprocessor ()` const noexcept
Get the number of multiprocessor of this executor.
- `int get_num_warps ()` const noexcept
Get the number of warps of this executor.
- `int get_major_version ()` const noexcept
Get the major verion of compute capability.
- `int get_minor_version ()` const noexcept
Get the minor verion of compute capability.
- `cublasContext * get_cublas_handle ()` const
Get the cublas handle for this executor.
- `cusparseContext * get_cusparse_handle ()` const
Get the cusparse handle for this executor.

Static Public Member Functions

- `static std::shared_ptr< CudaExecutor > create (int device_id, std::shared_ptr< Executor > master)`
Creates a new [CudaExecutor](#).
- `static int get_num_devices ()`
Get the number of devices present on the system.

24.23.1 Detailed Description

This is the [Executor](#) subclass which represents the CUDA device.

24.23.2 Member Function Documentation

24.23.2.1 create()

```
static std::shared_ptr<CudaExecutor> gko::CudaExecutor::create (
    int device_id,
    std::shared_ptr< Executor > master ) [static]
```

Creates a new [CudaExecutor](#).

Parameters

<i>device_id</i>	the CUDA device id of this device
<i>master</i>	an executor on the host that is used to invoke the device kernels

24.23.2.2 get_cublas_handle()

```
cublasContext* gko::CudaExecutor::get_cublas_handle ( ) const [inline]
```

Get the cublas handle for this executor.

Returns

the cublas handle (cublasContext*) for this executor

```
874 { return cublas_handle_.get (); }
```

24.23.2.3 get_cuspars_handle()

```
cusparsContext* gko::CudaExecutor::get_cuspars_handle ( ) const [inline]
```

Get the cuspars handle for this executor.

Returns

the cuspars handle (cusparsContext*) for this executor

```
882 {
883     return cuspars_handle_.get ();
884 }
```

24.23.2.4 `get_master()` [1/2]

```
std::shared_ptr<Executor> gko::CudaExecutor::get_master ( ) [override], [virtual], [noexcept]
```

Returns the master [OmpExecutor](#) of this [Executor](#).

Returns

the master [OmpExecutor](#) of this [Executor](#).

Implements [gko::Executor](#).

24.23.2.5 `get_master()` [2/2]

```
std::shared_ptr<const Executor> gko::CudaExecutor::get_master ( ) const [override], [virtual], [noexcept]
```

Returns the master [OmpExecutor](#) of this [Executor](#).

Returns

the master [OmpExecutor](#) of this [Executor](#).

Implements [gko::Executor](#).

24.23.2.6 `run()`

```
void gko::CudaExecutor::run (
    const Operation & op ) const [override], [virtual]
```

Runs the specified [Operation](#) using this [Executor](#).

Parameters

<i>op</i>	the operation to run
-----------	----------------------

Implements [gko::Executor](#).

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

- `ginkgo/core/base/executor.hpp` (f1a4eb68)

24.24 `gko::CusparsError` Class Reference

[CusparsError](#) is thrown when a cuSPARSE routine throws a non-zero error code.

```
#include <ginkgo/core/base/exception.hpp>
```


Public Member Functions

- [CusparsedError](#) (const std::string &file, int line, const std::string &func, [int64](#) error_code)
Initializes a cuSPARSE error.

24.24.1 Detailed Description

[CusparsedError](#) is thrown when a cuSPARSE routine throws a non-zero error code.

24.24.2 Constructor & Destructor Documentation

24.24.2.1 CusparsedError()

```
gko::CusparsedError::CusparsedError (
    const std::string & file,
    int line,
    const std::string & func,
    int64 error_code ) [inline]
```

Initializes a cuSPARSE error.

Parameters

<i>file</i>	The name of the offending source file
<i>line</i>	The source code line number where the error occurred
<i>func</i>	The name of the cuSPARSE routine that failed
<i>error_code</i>	The resulting cuSPARSE error code

```
230         : Error(file, line, func + ": " + get_error(error_code))
231     {}
```

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

- ginkgo/core/base/exception.hpp (8045ac75)

24.25 gko::default_converter< S, R > Struct Template Reference

Used to convert objects of type *S* to objects of type *R* using `static_cast`.

```
#include <ginkgo/core/base/math.hpp>
```

Public Member Functions

- [R operator\(\)](#) (*S* val)
Converts the object to result type.

24.25.1 Detailed Description

```
template<typename S, typename R>
struct gko::default_converter< S, R >
```

Used to convert objects of type *S* to objects of type *R* using `static_cast`.

Template Parameters

<i>S</i>	source type
<i>R</i>	result type

24.25.2 Member Function Documentation

24.25.2.1 `operator()`

```
template<typename S , typename R >
R gko::default_converter< S, R >::operator() (
    S val ) [inline]
```

Converts the object to result type.

Parameters

<i>val</i>	the object to convert
------------	-----------------------

Returns

the converted object

```
276 { return static_cast<R>(val); }
```

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

- `ginkgo/core/base/math.hpp` (664d978f)

24.26 `gko::matrix::Dense< ValueType >` Class Template Reference

`Dense` is a matrix format which explicitly stores all values of the matrix.

```
#include <ginkgo/core/matrix/dense.hpp>
```

Public Member Functions

- `std::unique_ptr< LinOp > transpose ()` const override
Returns a LinOp representing the transpose of the [Transposable](#) object.
- `std::unique_ptr< LinOp > conj_transpose ()` const override
Returns a LinOp representing the conjugate transpose of the [Transposable](#) object.
- `value_type * get_values ()` noexcept
Returns a pointer to the array of values of the matrix.
- `const value_type * get_const_values ()` const noexcept
Returns a pointer to the array of values of the matrix.
- `size_type get_stride ()` const noexcept
Returns the stride of the matrix.
- `size_type get_num_stored_elements ()` const noexcept
Returns the number of elements explicitly stored in the matrix.
- `value_type & at (size_type row, size_type col)` noexcept
Returns a single element of the matrix.
- `value_type at (size_type row, size_type col)` const noexcept
Returns a single element of the matrix.
- `ValueType & at (size_type idx)` noexcept
Returns a single element of the matrix.
- `ValueType at (size_type idx)` const noexcept
Returns a single element of the matrix.
- `void scale (const LinOp *alpha)`
Scales the matrix with a scalar (aka: BLAS scal).
- `void add_scaled (const LinOp *alpha, const LinOp *b)`
Adds b scaled by α to the matrix (aka: BLAS axpy).
- `void compute_dot (const LinOp *b, LinOp *result)` const
Computes the column-wise dot product of this matrix and b .
- `void compute_norm2 (LinOp *result)` const
Computes the Euclidian (L^2) norm of this matrix.
- `std::unique_ptr< Dense > create_submatrix (const span &rows, const span &columns, const size_type stride)`
Create a submatrix from the original matrix.

Static Public Member Functions

- `static std::unique_ptr< Dense > create_with_config_of (const Dense *other)`
Creates a [Dense](#) matrix with the configuration of another [Dense](#) matrix.

24.26.1 Detailed Description

```
template<typename ValueType = default_precision>
class gko::matrix::Dense< ValueType >
```

[Dense](#) is a matrix format which explicitly stores all values of the matrix.

The values are stored in row-major format (values belonging to the same row appear consecutive in the memory). Optionally, rows can be padded for better memory access.

Template Parameters

<i>ValueType</i>	precision of matrix elements
------------------	------------------------------

Note

While this format is not very useful for storing sparse matrices, it is often suitable to store vectors, and sets of vectors.

24.26.2 Member Function Documentation

24.26.2.1 `add_scaled()`

```
template<typename ValueType = default_precision>
void gko::matrix::Dense< ValueType >::add_scaled (
    const LinOp * alpha,
    const LinOp * b ) [inline]
```

Adds `b` scaled by `alpha` to the matrix (aka: BLAS axpy).

Parameters

<i>alpha</i>	If <code>alpha</code> is 1x1 Dense matrix, the entire matrix is scaled by <code>alpha</code> . If it is a Dense row vector of values, then <code>i</code> -th column of the matrix is scaled with the <code>i</code> -th element of <code>alpha</code> (the number of columns of <code>alpha</code> has to match the number of columns of the matrix).
<i>b</i>	a matrix of the same dimension as this

References `gko::PolymorphicObject::get_executor()`, and `gko::make_temporary_clone()`.

```
306     {
307         auto exec = this->get_executor();
308         this->add_scaled_impl(make_temporary_clone(exec, alpha).get(),
309                             make_temporary_clone(exec, b).get());
310     }
```

24.26.2.2 `at()` [1/4]

```
template<typename ValueType = default_precision>
value_type& gko::matrix::Dense< ValueType >::at (
    size_type row,
    size_type col ) [inline], [noexcept]
```

Returns a single element of the matrix.

Parameters

<i>row</i>	the row of the requested element
<i>col</i>	the column of the requested element

Note

the method has to be called on the same [Executor](#) the matrix is stored at (e.g. trying to call this method on a GPU matrix from the OMP results in a runtime error)

References `gko::Array< ValueType >::get_data()`.

Referenced by `gko::initialize()`.

```

241     {
242         return values_.get_data()[linearize_index(row, col)];
243     }

```

24.26.2.3 `at()` [2/4]

```

template<typename ValueType = default_precision>
value_type gko::matrix::Dense< ValueType >::at (
    size_type row,
    size_type col ) const [inline], [noexcept]

```

Returns a single element of the matrix.

Parameters

<i>row</i>	the row of the requested element
<i>col</i>	the column of the requested element

Note

the method has to be called on the same [Executor](#) the matrix is stored at (e.g. trying to call this method on a GPU matrix from the OMP results in a runtime error)

References `gko::Array< ValueType >::get_const_data()`.

```

249     {
250         return values_.get_const_data()[linearize_index(row, col)];
251     }

```

24.26.2.4 at() [3/4]

```
template<typename ValueType = default_precision>
ValueType& gko::matrix::Dense< ValueType >::at (
    size_type idx ) [inline], [noexcept]
```

Returns a single element of the matrix.

Useful for iterating across all elements of the matrix. However, it is less efficient than the two-parameter variant of this method.

Parameters

<i>idx</i>	a linear index of the requested element (ignoring the stride)
------------	---

Note

the method has to be called on the same [Executor](#) the matrix is stored at (e.g. trying to call this method on a GPU matrix from the OMP results in a runtime error)

References `gko::Array< ValueType >::get_data()`.

```
268     {
269         return values_.get_data() [linearize_index(idx)];
270     }
```

24.26.2.5 at() [4/4]

```
template<typename ValueType = default_precision>
ValueType gko::matrix::Dense< ValueType >::at (
    size_type idx ) const [inline], [noexcept]
```

Returns a single element of the matrix.

Useful for iterating across all elements of the matrix. However, it is less efficient than the two-parameter variant of this method.

Parameters

<i>idx</i>	a linear index of the requested element (ignoring the stride)
------------	---

Note

the method has to be called on the same [Executor](#) the matrix is stored at (e.g. trying to call this method on a GPU matrix from the OMP results in a runtime error)

References `gko::Array< ValueType >::get_const_data()`.

```
276     {
277         return values_.get_const_data() [linearize_index(idx)];
278     }
```

24.26.2.6 compute_dot()

```
template<typename ValueType = default_precision>
void gko::matrix::Dense< ValueType >::compute_dot (
    const LinOp * b,
    LinOp * result ) const [inline]
```

Computes the column-wise dot product of this matrix and b.

The conjugate of this is taken.

Parameters

<i>b</i>	a Dense matrix of same dimension as this
<i>result</i>	a Dense row vector, used to store the dot product (the number of column in the vector must match the number of columns of this)

References [gko::PolymorphicObject::get_executor\(\)](#), and [gko::make_temporary_clone\(\)](#).

```
322     {
323         auto exec = this->get_executor();
324         this->compute_dot_impl(make_temporary_clone(exec, b).get(),
325                               make_temporary_clone(exec, result).get());
326     }
```

24.26.2.7 compute_norm2()

```
template<typename ValueType = default_precision>
void gko::matrix::Dense< ValueType >::compute_norm2 (
    LinOp * result ) const [inline]
```

Computes the Euclidian (L^2) norm of this matrix.

Parameters

<i>result</i>	a Dense row vector, used to store the norm (the number of columns in the vector must match the number of columns of this)
---------------	---

References [gko::PolymorphicObject::get_executor\(\)](#), and [gko::make_temporary_clone\(\)](#).

```
336     {
337         auto exec = this->get_executor();
338         this->compute_norm2_impl(make_temporary_clone(exec, result).get());
339     }
```

24.26.2.8 conj_transpose()

```
template<typename ValueType = default_precision>
std::unique_ptr<LinOp> gko::matrix::Dense< ValueType >::conj_transpose ( ) const [override],
[virtual]
```

Returns a LinOp representing the conjugate transpose of the [Transposable](#) object.

Returns

a pointer to the new conjugate transposed object

Implements [gko::Transposable](#).

Referenced by `gko::matrix::Dense< ValueType >::create_with_config_of()`.

24.26.2.9 create_submatrix()

```
template<typename ValueType = default_precision>
std::unique_ptr<Dense> gko::matrix::Dense< ValueType >::create_submatrix (
    const span & rows,
    const span & columns,
    const size_type stride ) [inline]
```

Create a submatrix from the original matrix.

Warning: defining stride for this `create_submatrix` method might cause wrong memory access. Better use the `create_submatrix(rows, columns)` method instead.

Parameters

<i>rows</i>	row span
<i>columns</i>	column span
<i>stride</i>	stride of the new submatrix.

References `gko::span::begin`, `gko::PolymorphicObject::get_executor()`, `gko::Array< ValueType >::get_num_elems()`, `gko::matrix::Dense< ValueType >::get_stride()`, `gko::matrix::Dense< ValueType >::get_values()`, and `gko::Array< ValueType >::view()`.

```
354     {
355         row_major_range range_this{this->get_values(), this->get_size()[0],
356                                     this->get_size()[1], this->get_stride()};
357         auto range_result = range_this(rows, columns);
358         // TODO: can result in HUGE padding - which will be copied with the
359         // vector
360         return Dense::create(
361             this->get_executor(),
362             dim<2>{range_result.length(0), range_result.length(1)},
363             Array<ValueType>::view(
364                 this->get_executor(),
365                 range_result.length(0) * range_this.length(1) - columns.begin,
366                 range_result->data(),
367                 stride);
368     }
```


24.26.2.10 create_with_config_of()

```
template<typename ValueType = default_precision>
static std::unique_ptr<Dense> gko::matrix::Dense< ValueType >::create_with_config_of (
    const Dense< ValueType > * other ) [inline], [static]
```

Creates a [Dense](#) matrix with the configuration of another [Dense](#) matrix.

Parameters

<i>other</i>	The other matrix whose configuration needs to copied.
--------------	---

References [gko::matrix::Dense< ValueType >::conj_transpose\(\)](#), and [gko::matrix::Dense< ValueType >::transpose\(\)](#).

```
132     {
133         // De-referencing 'other' before calling the functions (instead of
134         // using operator '->') is currently required to be compatible with
135         // CUDA 10.1.
136         // Otherwise, it results in a compile error.
137         // TODO Check if the compiler error is fixed and revert to 'operator->'.
138         return Dense::create((*other).get_executor(), (*other).get_size(),
139                             (*other).get_stride());
140     }
```

24.26.2.11 get_const_values()

```
template<typename ValueType = default_precision>
const value_type* gko::matrix::Dense< ValueType >::get_const_values ( ) const [inline], [noexcept]
```

Returns a pointer to the array of values of the matrix.

Returns

the pointer to the array of values

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

References [gko::Array< ValueType >::get_const_data\(\)](#).

```
209     {
210         return values_.get_const_data();
211     }
```

24.26.2.12 get_num_stored_elements()

```
template<typename ValueType = default_precision>
size_type gko::matrix::Dense< ValueType >::get_num_stored_elements ( ) const [inline], [noexcept]
```

Returns the number of elements explicitly stored in the matrix.

Returns

the number of elements explicitly stored in the matrix

References `gko::Array< ValueType >::get_num_elems()`.

```
226 {
227     return values_.get_num_elems();
228 }
```

24.26.2.13 get_stride()

```
template<typename ValueType = default_precision>
size_type gko::matrix::Dense< ValueType >::get_stride ( ) const [inline], [noexcept]
```

Returns the stride of the matrix.

Returns

the stride of the matrix.

Referenced by `gko::matrix::Dense< ValueType >::create_submatrix()`.

```
218 { return stride_; }
```

24.26.2.14 get_values()

```
template<typename ValueType = default_precision>
value_type* gko::matrix::Dense< ValueType >::get_values ( ) [inline], [noexcept]
```

Returns a pointer to the array of values of the matrix.

Returns

the pointer to the array of values

References `gko::Array< ValueType >::get_data()`.

Referenced by `gko::matrix::Dense< ValueType >::create_submatrix()`.

```
199 { return values_.get_data(); }
```

24.26.2.15 scale()

```
template<typename ValueType = default_precision>
void gko::matrix::Dense< ValueType >::scale (
    const LinOp * alpha ) [inline]
```

Scales the matrix with a scalar (aka: BLAS scal).

Parameters

<i>alpha</i>	If alpha is 1x1 Dense matrix, the entire matrix is scaled by alpha. If it is a Dense row vector of values, then i-th column of the matrix is scaled with the i-th element of alpha (the number of columns of alpha has to match the number of columns of the matrix).
--------------	---

References `gko::PolymorphicObject::get_executor()`, and `gko::make_temporary_clone()`.

```

290     {
291         auto exec = this->get_executor();
292         this->scale_impl(make_temporary_clone(exec, alpha).get());
293     }

```

24.26.2.16 transpose()

```

template<typename ValueType = default_precision>
std::unique_ptr<LinOp> gko::matrix::Dense< ValueType >::transpose ( ) const [override],
[virtual]

```

Returns a `LinOp` representing the transpose of the [Transposable](#) object.

Returns

a pointer to the new transposed object

Implements [gko::Transposable](#).

Referenced by `gko::matrix::Dense< ValueType >::create_with_config_of()`.

The documentation for this class was generated from the following files:

- `ginkgo/core/matrix/coo.hpp` (5545d209)
- `ginkgo/core/matrix/dense.hpp` (664ab4d8)

24.27 gko::dim< Dimensionality, DimensionType > Struct Template Reference

A type representing the dimensions of a multidimensional object.

```
#include <ginkgo/core/base/dim.hpp>
```

Public Member Functions

- `constexpr dim (const dimension_type &size=dimension_type{})`
Creates a dimension object with all dimensions set to the same value.
- `template<typename... Rest>`
`constexpr dim (const dimension_type &first, const Rest &... rest)`
Creates a dimension object with the specified dimensions.
- `constexpr const dimension_type & operator[] (const size_type &dimension) const noexcept`
Returns the requested dimension.
- `dimension_type & operator[] (const size_type &dimension) noexcept`
- `constexpr operator bool () const`
Checks if all dimensions evaluate to true.

Friends

- constexpr bool `operator==` (const `dim` &x, const `dim` &y)
Checks if two dim objects are equal.
- constexpr `dim operator*` (const `dim` &x, const `dim` &y)
Multiplies two dim objects.

24.27.1 Detailed Description

```
template<size_type Dimensionality, typename DimensionType = size_type>
struct gko::dim< Dimensionality, DimensionType >
```

A type representing the dimensions of a multidimensional object.

Template Parameters

<i>Dimensionality</i>	number of dimensions of the object
<i>DimensionType</i>	datatype used to represent each dimension

24.27.2 Constructor & Destructor Documentation

24.27.2.1 `dim()` [1/2]

```
template<size_type Dimensionality, typename DimensionType = size_type>
constexpr gko::dim< Dimensionality, DimensionType >::dim (
    const dimension_type & size = dimension_type{} ) [inline]
```

Creates a dimension object with all dimensions set to the same value.

Parameters

<i>size</i>	the size of each dimension
-------------	----------------------------

```
63                                     {}
64         : first_{size}, rest_{size}
65         {}
```

24.27.2.2 `dim()` [2/2]

```
template<size_type Dimensionality, typename DimensionType = size_type>
template<typename... Rest>
```

```
constexpr gko::dim< Dimensionality, DimensionType >::dim (
    const dimension_type & first,
    const Rest &... rest ) [inline]
```

Creates a dimension object with the specified dimensions.

If the number of dimensions given is less than the dimensionality of the object, the remaining dimensions are set to the same value as the last value given.

For example, in the context of matrices `dim<2>{2, 3}` creates the dimensions for a 2-by-3 matrix.

Parameters

<i>first</i>	first dimension
<i>rest</i>	other dimensions

```
83         : first_{first}, rest_{static_cast<dimension_type>(rest)...}
84     {}
```

24.27.3 Member Function Documentation

24.27.3.1 operator bool()

```
template<size_type Dimensionality, typename DimensionType = size_type>
constexpr gko::dim< Dimensionality, DimensionType >::operator bool ( ) const [inline]
```

Checks if all dimensions evaluate to true.

For standard arithmetic types, this is equivalent to all dimensions being different than zero.

Returns

true if and only if all dimensions evaluate to true

```
121     {
122         return static_cast<bool>(first_) && static_cast<bool>(rest_);
123     }
```

24.27.3.2 operator[]() [1/2]

```
template<size_type Dimensionality, typename DimensionType = size_type>
constexpr const dimension_type& gko::dim< Dimensionality, DimensionType >::operator[] (
    const size_type & dimension ) const [inline], [noexcept]
```

Returns the requested dimension.

For example, if `d` is a `dim<2>` object representing matrix dimensions, `d[0]` returns the number of rows, and `d[1]` returns the number of columns.

Parameters

<i>dimension</i>	the requested dimension
------------------	-------------------------

Returns

the dimension-th dimension

```

99      {
100         return GKO_ASSERT(dimension < dimensionality), *(&first_ + dimension);
101     }
```

24.27.3.3 operator[]() [2/2]

```

template<size_type Dimensionality, typename DimensionType = size_type>
dimension_type& gko::dim< Dimensionality, DimensionType >::operator[] (
    const size_type & dimension ) [inline], [noexcept]
```

```

108     {
109         return GKO_ASSERT(dimension < dimensionality), *(&first_ + dimension);
110     }
```

24.27.4 Friends And Related Function Documentation**24.27.4.1 operator***

```

template<size_type Dimensionality, typename DimensionType = size_type>
constexpr dim operator* (
    const dim< Dimensionality, DimensionType > & x,
    const dim< Dimensionality, DimensionType > & y ) [friend]
```

Multiplies two dim objects.

Parameters

<i>x</i>	first object
<i>y</i>	second object

Returns

a dim object representing the size of the tensor product $x * y$

```

147     {
148         return dim(x.first_ * y.first_, x.rest_ * y.rest_);
149     }
```

24.27.4.2 operator==

```
template<size_type Dimensionality, typename DimensionType = size_type>
constexpr bool operator== (
    const dim< Dimensionality, DimensionType > & x,
    const dim< Dimensionality, DimensionType > & y ) [friend]
```

Checks if two dim objects are equal.

Parameters

<i>x</i>	first object
<i>y</i>	second object

Returns

true if and only if all dimensions of both objects are equal.

```
134     {
135         return x.first_ == y.first_ && x.rest_ == y.rest_;
136     }
```

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

- ginkgo/core/base/dim.hpp (f1a4eb68)

24.28 gko::DimensionMismatch Class Reference

[DimensionMismatch](#) is thrown if an operation is being applied to LinOps of incompatible size.

```
#include <ginkgo/core/base/exception.hpp>
```

Public Member Functions

- [DimensionMismatch](#) (const std::string &file, int line, const std::string &func, const std::string &first_name, [size_type](#) first_rows, [size_type](#) first_cols, const std::string &second_name, [size_type](#) second_rows, [size_type](#) second_cols, const std::string &clarification)

Initializes a dimension mismatch error.

24.28.1 Detailed Description

[DimensionMismatch](#) is thrown if an operation is being applied to LinOps of incompatible size.

24.28.2 Constructor & Destructor Documentation

24.28.2.1 DimensionMismatch()

```
gko::DimensionMismatch::DimensionMismatch (
    const std::string & file,
    int line,
    const std::string & func,
    const std::string & first_name,
    size_type first_rows,
    size_type first_cols,
    const std::string & second_name,
    size_type second_rows,
    size_type second_cols,
    const std::string & clarification ) [inline]
```

Initializes a dimension mismatch error.

Parameters

<i>file</i>	The name of the offending source file
<i>line</i>	The source code line number where the error occurred
<i>func</i>	The function name where the error occurred
<i>first_name</i>	The name of the first operator
<i>first_rows</i>	The output dimension of the first operator
<i>first_cols</i>	The input dimension of the first operator
<i>second_name</i>	The name of the second operator
<i>second_rows</i>	The output dimension of the second operator
<i>second_cols</i>	The input dimension of the second operator
<i>clarification</i>	An additional message describing the error further

```
262         : Error(file, line,
263               func + ": attempting to combine operators " + first_name +
264                 " [" + std::to_string(first_rows) + " x " +
265                 std::to_string(first_cols) + "] and " + second_name + " [" +
266                 std::to_string(second_rows) + " x " +
267                 std::to_string(second_cols) + "]: " + clarification)
268     {}
```

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

- ginkgo/core/base/exception.hpp (8045ac75)

24.29 gko::matrix::Ell< ValueType, IndexType > Class Template Reference

ELL is a matrix format where stride with explicit zeros is used such that all rows have the same number of stored elements.

```
#include <ginkgo/core/matrix/ell.hpp>
```


Public Member Functions

- void [read](#) (const [mat_data](#) &data) override
Reads a matrix from a [matrix_data](#) structure.
- void [write](#) ([mat_data](#) &data) const override
Writes a matrix to a [matrix_data](#) structure.
- value_type * [get_values](#) () noexcept
Returns the values of the matrix.
- const value_type * [get_const_values](#) () const noexcept
Returns the values of the matrix.
- index_type * [get_col_idxs](#) () noexcept
Returns the column indexes of the matrix.
- const index_type * [get_const_col_idxs](#) () const noexcept
Returns the column indexes of the matrix.
- size_type [get_num_stored_elements_per_row](#) () const noexcept
Returns the number of stored elements per row.
- size_type [get_stride](#) () const noexcept
Returns the stride of the matrix.
- size_type [get_num_stored_elements](#) () const noexcept
Returns the number of elements explicitly stored in the matrix.
- value_type & [val_at](#) (size_type row, size_type idx) noexcept
*Returns the *idx*-th non-zero element of the *row*-th row .*
- value_type [val_at](#) (size_type row, size_type idx) const noexcept
*Returns the *idx*-th non-zero element of the *row*-th row .*
- index_type & [col_at](#) (size_type row, size_type idx) noexcept
*Returns the *idx*-th column index of the *row*-th row .*
- index_type [col_at](#) (size_type row, size_type idx) const noexcept
*Returns the *idx*-th column index of the *row*-th row .*

24.29.1 Detailed Description

```
template<typename ValueType = default_precision, typename IndexType = int32>
class gko::matrix::Ell< ValueType, IndexType >
```

ELL is a matrix format where stride with explicit zeros is used such that all rows have the same number of stored elements.

The number of elements stored in each row is the largest number of nonzero elements in any of the rows (obtainable through [get_num_stored_elements_per_row\(\)](#) method). This removes the need of a row pointer like in the CSR format, and allows for SIMD processing of the distinct rows. For efficient processing, the nonzero elements and the corresponding column indices are stored in column-major fashion. The columns are padded to the length by user-defined stride parameter whose default value is the number of rows of the matrix.

Template Parameters

<i>ValueType</i>	precision of matrix elements
<i>IndexType</i>	precision of matrix indexes

24.29.2 Member Function Documentation

24.29.2.1 `col_at()` [1/2]

```
template<typename ValueType = default_precision, typename IndexType = int32>
index_type& gko::matrix::Ell< ValueType, IndexType >::col_at (
    size_type row,
    size_type idx ) [inline], [noexcept]
```

Returns the `idx`-th column index of the `row`-th row .

Parameters

<i>row</i>	the row of the requested element
<i>idx</i>	the <code>idx</code> -th stored element of the row

Note

the method has to be called on the same [Executor](#) the matrix is stored at (e.g. trying to call this method on a GPU matrix from the OMP results in a runtime error)

References `gko::matrix::Ell< ValueType, IndexType >::get_col_idxes()`.

```
201     {
202         return this->get_col_idxes() [this->linearize_index(row, idx)];
203     }
```

24.29.2.2 `col_at()` [2/2]

```
template<typename ValueType = default_precision, typename IndexType = int32>
index_type gko::matrix::Ell< ValueType, IndexType >::col_at (
    size_type row,
    size_type idx ) const [inline], [noexcept]
```

Returns the `idx`-th column index of the `row`-th row .

Parameters

<i>row</i>	the row of the requested element
<i>idx</i>	the <code>idx</code> -th stored element of the row

Note

the method has to be called on the same [Executor](#) the matrix is stored at (e.g. trying to call this method on a GPU matrix from the OMP results in a runtime error)

References gko::matrix::Ell< ValueType, IndexType >::get_const_col_idxs(), and gko::Array< ValueType >::get↵_num_elems().

```
209 {
210     return this->get_const_col_idxs() [this->linearize_index(row, idx)];
211 }
```

24.29.2.3 get_col_idxs()

```
template<typename ValueType = default_precision, typename IndexType = int32>
index_type* gko::matrix::Ell< ValueType, IndexType >::get_col_idxs ( ) [inline], [noexcept]
```

Returns the column indexes of the matrix.

Returns

the column indexes of the matrix.

References gko::Array< ValueType >::get_data().

Referenced by gko::matrix::Ell< ValueType, IndexType >::col_at().

```
126 { return col_idxs_.get_data(); }
```

24.29.2.4 get_const_col_idxs()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const index_type* gko::matrix::Ell< ValueType, IndexType >::get_const_col_idxs ( ) const [inline],
[noexcept]
```

Returns the column indexes of the matrix.

Returns

the column indexes of the matrix.

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

References gko::Array< ValueType >::get_const_data().

Referenced by gko::matrix::Ell< ValueType, IndexType >::col_at().

```
136 {
137     return col_idxs_.get_const_data();
138 }
```

24.29.2.5 get_const_values()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const value_type* gko::matrix::Ell< ValueType, IndexType >::get_const_values ( ) const [inline],
[noexcept]
```

Returns the values of the matrix.

Returns

the values of the matrix.

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

References `gko::Array< ValueType >::get_const_data()`.

```
117     {
118         return values_.get_const_data();
119     }
```

24.29.2.6 get_num_stored_elements()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Ell< ValueType, IndexType >::get_num_stored_elements ( ) const [inline],
[noexcept]
```

Returns the number of elements explicitly stored in the matrix.

Returns

the number of elements explicitly stored in the matrix

References `gko::Array< ValueType >::get_num_elems()`.

```
163     {
164         return values_.get_num_elems();
165     }
```

24.29.2.7 get_num_stored_elements_per_row()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Ell< ValueType, IndexType >::get_num_stored_elements_per_row ( ) const
[inline], [noexcept]
```

Returns the number of stored elements per row.

Returns

the number of stored elements per row.

```
146     {
147         return num_stored_elements_per_row_;
148     }
```

24.29.2.8 get_stride()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Ell< ValueType, IndexType >::get_stride ( ) const [inline], [noexcept]
```

Returns the stride of the matrix.

Returns

the stride of the matrix.

```
155 { return stride_; }
```

24.29.2.9 get_values()

```
template<typename ValueType = default_precision, typename IndexType = int32>
value_type* gko::matrix::Ell< ValueType, IndexType >::get_values ( ) [inline], [noexcept]
```

Returns the values of the matrix.

Returns

the values of the matrix.

References `gko::Array< ValueType >::get_data()`.

```
107 { return values_.get_data(); }
```

24.29.2.10 read()

```
template<typename ValueType = default_precision, typename IndexType = int32>
void gko::matrix::Ell< ValueType, IndexType >::read (
    const mat_data & data ) [override], [virtual]
```

Reads a matrix from a `matrix_data` structure.

Parameters

<i>data</i>	the matrix_data structure
-------------	---

Implements [gko::ReadableFromMatrixData< ValueType, IndexType >](#).

24.29.2.11 `val_at()` [1/2]

```
template<typename ValueType = default_precision, typename IndexType = int32>
value_type& gko::matrix::Ell< ValueType, IndexType >::val_at (
    size_type row,
    size_type idx ) [inline], [noexcept]
```

Returns the `idx`-th non-zero element of the `row`-th row .

Parameters

<i>row</i>	the row of the requested element
<i>idx</i>	the <code>idx</code> -th stored element of the row

Note

the method has to be called on the same [Executor](#) the matrix is stored at (e.g. trying to call this method on a GPU matrix from the OMP results in a runtime error)

References [gko::Array< ValueType >::get_data\(\)](#).

```
178     {
179         return values_.get\_data() [this->linearize_index(row, idx)];
180     }
```

24.29.2.12 `val_at()` [2/2]

```
template<typename ValueType = default_precision, typename IndexType = int32>
value_type gko::matrix::Ell< ValueType, IndexType >::val_at (
    size_type row,
    size_type idx ) const [inline], [noexcept]
```

Returns the `idx`-th non-zero element of the `row`-th row .

Parameters

<i>row</i>	the row of the requested element
<i>idx</i>	the <code>idx</code> -th stored element of the row

Note

the method has to be called on the same [Executor](#) the matrix is stored at (e.g. trying to call this method on a GPU matrix from the OMP results in a runtime error)

References `gko::Array< ValueType >::get_const_data()`.

```
186     {
187         return values_.get_const_data()[this->linearize_index(row, idx)];
188     }
```

24.29.2.13 write()

```
template<typename ValueType = default_precision, typename IndexType = int32>
void gko::matrix::Ell< ValueType, IndexType >::write (
    mat_data & data ) const [override], [virtual]
```

Writes a matrix to a `matrix_data` structure.

Parameters

<i>data</i>	the <code>matrix_data</code> structure
-------------	--

Implements `gko::WritableToMatrixData< ValueType, IndexType >`.

The documentation for this class was generated from the following files:

- ginkgo/core/matrix/csr.hpp (ec6dbb34)
- ginkgo/core/matrix/ell.hpp (8045ac75)

24.30 gko::enable_parameters_type< ConcreteParametersType, Factory > Struct Template Reference

The `enable_parameters_type` mixin is used to create a base implementation of the factory parameters structure.

```
#include <ginkgo/core/base/abstract_factory.hpp>
```

Public Member Functions

- `std::unique_ptr< Factory > on (std::shared_ptr< const Executor > exec) const`
Creates a new factory on the specified executor.

24.30.1 Detailed Description

```
template<typename ConcreteParametersType, typename Factory>
struct gko::enable_parameters_type< ConcreteParametersType, Factory >
```

The `enable_parameters_type` mixin is used to create a base implementation of the factory parameters structure.

It provides only the `on()` method which can be used to instantiate the factory give the parameters stored in the structure.

Template Parameters

<i>ConcreteParametersType</i>	the concrete parameters type which is being implemented [CRTP parameter]
<i>Factory</i>	the concrete factory for which these parameters are being used

24.30.2 Member Function Documentation

24.30.2.1 on()

```
template<typename ConcreteParametersType, typename Factory>
std::unique_ptr<Factory> gko::enable_parameters_type< ConcreteParametersType, Factory >::on (
    std::shared_ptr< const Executor > exec ) const [inline]
```

Creates a new factory on the specified executor.

Parameters

<i>exec</i>	the executor where the factory will be created
-------------	--

Returns

a new factory instance

```
280     {
281         return std::unique_ptr<Factory>(new Factory(exec, *self()));
282     }
```

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

- ginkgo/core/base/abstract_factory.hpp (8045ac75)

24.31 gko::EnableAbstractPolymorphicObject< AbstractObject, PolymorphicBase > Class Template Reference

This mixin inherits from (a subclass of) [PolymorphicObject](#) and provides a base implementation of a new abstract object.

```
#include <ginkgo/core/base/polymorphic_object.hpp>
```

24.31.1 Detailed Description

```
template<typename AbstractObject, typename PolymorphicBase = PolymorphicObject>
class gko::EnableAbstractPolymorphicObject< AbstractObject, PolymorphicBase >
```

This mixin inherits from (a subclass of) [PolymorphicObject](#) and provides a base implementation of a new abstract object.

It uses method hiding to update the parameter and return types from [PolymorphicObject](#) to `AbstractObject` wherever it makes sense. As opposed to [EnablePolymorphicObject](#), it does not implement [PolymorphicObject](#)'s virtual methods.

Template Parameters

<i>AbstractObject</i>	the abstract class which is being implemented [CRTP parameter]
<i>PolymorphicBase</i>	parent of AbstractObject in the polymorphic hierarchy, has to be a subclass of polymorphic object

See also

[EnablePolymorphicObject](#) for creating a concrete subclass of [PolymorphicObject](#).

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

- ginkgo/core/base/polymorphic_object.hpp (4bde4271)

24.32 gko::EnableCreateMethod< ConcreteType > Class Template Reference

This mixin implements a static `create()` method on `ConcreteType` that dynamically allocates the memory, uses the passed-in arguments to construct the object, and returns an `std::unique_ptr` to such an object.

```
#include <ginkgo/core/base/polymorphic_object.hpp>
```

24.32.1 Detailed Description

```
template<typename ConcreteType>
class gko::EnableCreateMethod< ConcreteType >
```

This mixin implements a static `create()` method on `ConcreteType` that dynamically allocates the memory, uses the passed-in arguments to construct the object, and returns an `std::unique_ptr` to such an object.

Template Parameters

<i>ConcreteObject</i>	the concrete type for which <code>create()</code> is being implemented [CRTP parameter]
-----------------------	---

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

- ginkgo/core/base/polymorphic_object.hpp (4bde4271)

24.33 gko::EnableDefaultFactory< ConcreteFactory, ProductType, ParametersType, PolymorphicBase > Class Template Reference

This mixin provides a default implementation of a concrete factory.

```
#include <ginkgo/core/base/abstract_factory.hpp>
```

Public Member Functions

- `const parameters_type & get_parameters () const noexcept`

Returns the parameters of the factory.

Static Public Member Functions

- `static parameters_type create ()`

Creates a new ParametersType object which can be used to instantiate a new ConcreteFactory.

24.33.1 Detailed Description

```
template<typename ConcreteFactory, typename ProductType, typename ParametersType, typename PolymorphicBase>
class gko::EnableDefaultFactory< ConcreteFactory, ProductType, ParametersType, PolymorphicBase >
```

This mixin provides a default implementation of a concrete factory.

It implements all the methods of [AbstractFactory](#) and [PolymorphicObject](#). Its implementation of the `generate_impl()` method delegates the creation of the product by calling the `ProductType::ProductType(const ConcreteFactory *, const components_type &)` constructor. The factory also supports parameters by using the `ParametersType` structure, which is defined by the user.

For a simple example, see `IntFactory` in `core/test/base/abstract_factory.cpp`.

Template Parameters

<i>ConcreteFactory</i>	the concrete factory which is being implemented [CRTP parameter]
<i>ProductType</i>	the concrete type of products which this factory produces, has to be a subclass of <code>PolymorphicBase::abstract_product_type</code>
<i>ParametersType</i>	a type representing the parameters of the factory, has to inherit from the enable_parameters_type mixin
<i>PolymorphicBase</i>	parent of <code>ConcreteFactory</code> in the polymorphic hierarchy, has to be a subclass of AbstractFactory

24.33.2 Member Function Documentation

24.33.2.1 create()

```
template<typename ConcreteFactory , typename ProductType , typename ParametersType , typename
PolymorphicBase >
static parameters_type gko::EnableDefaultFactory< ConcreteFactory, ProductType, ParametersType, PolymorphicBase >::create ( ) [inline], [static]
```

Creates a new `ParametersType` object which can be used to instantiate a new `ConcreteFactory`.

This method does not construct the factory directly, but returns a new `parameters_type` object, which can be used to set the parameters of the factory. Once the parameters have been set, the `parameters_type::on()` method can be used to obtain an instance of the factory with those parameters.

Returns

a default `parameters_type` object

```
192 { return {}; }
```

24.33.2.2 `get_parameters()`

```
template<typename ConcreteFactory , typename ProductType , typename ParametersType , typename
PolymorphicBase >
const parameters_type& gko::EnableDefaultFactory< ConcreteFactory, ProductType, ParametersType, PolymorphicBase >::get_parameters ( ) const [inline], [noexcept]
```

Returns the parameters of the factory.

Returns

the parameters of the factory

```
176 {
177     return parameters_;
178 };
```

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

- `ginkgo/core/base/abstract_factory.hpp` (8045ac75)

24.34 `gko::EnableLinOp< ConcreteLinOp, PolymorphicBase >` Class Template Reference

The [EnableLinOp](#) mixin can be used to provide sensible default implementations of the majority of the `LinOp` and `PolymorphicObject` interface.

```
#include <ginkgo/core/base/lin_op.hpp>
```

Additional Inherited Members**24.34.1 Detailed Description**

```
template<typename ConcreteLinOp, typename PolymorphicBase = LinOp>
class gko::EnableLinOp< ConcreteLinOp, PolymorphicBase >
```

The [EnableLinOp](#) mixin can be used to provide sensible default implementations of the majority of the `LinOp` and `PolymorphicObject` interface.

The goal of the mixin is to facilitate the development of new `LinOp`, by enabling the implementers to focus on the important parts of their operator, while the library takes care of generating the trivial utility functions. The mixin will provide default implementations for the entire [PolymorphicObject](#) interface, including a default implementation of `copy_from` between objects of the new `LinOp` type. It will also hide the default `LinOp::apply()` methods with versions that preserve the static type of the object.

Implementers of new `LinOps` are required to specify only the following aspects:

1. Creation of the `LinOp`: This can be facilitated via either [EnableCreateMethod](#) mixin (used mostly for matrix formats), or `GKO_ENABLE_LIN_OP_FACTORY` macro (used for operators created from other operators, like preconditioners and solvers).
2. Application of the `LinOp`: Implementers have to override the two overloads of the `LinOp::apply_impl()` virtual methods.

Template Parameters

<i>ConcreteLinOp</i>	the concrete LinOp which is being implemented [CRTP parameter]
<i>PolymorphicBase</i>	parent of ConcreteLinOp in the polymorphic hierarchy, has to be a subclass of LinOp

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

- ginkgo/core/base/lin_op.hpp (fb72cdf1)

24.35 gko::log::EnableLogging< ConcreteLoggable, PolymorphicBase > Class Template Reference

[EnableLogging](#) is a mixin which should be inherited by any class which wants to enable logging.

```
#include <ginkgo/core/log/logger.hpp>
```

Public Member Functions

- void [add_logger](#) (std::shared_ptr< const Logger > logger) override
Adds a new logger to the list of subscribed loggers.
- void [remove_logger](#) (const Logger *logger) override
Removes a logger from the list of subscribed loggers.

24.35.1 Detailed Description

```
template<typename ConcreteLoggable, typename PolymorphicBase = Loggable>
class gko::log::EnableLogging< ConcreteLoggable, PolymorphicBase >
```

[EnableLogging](#) is a mixin which should be inherited by any class which wants to enable logging.

All the received events are passed to the loggers this class contains.

Template Parameters

<i>ConcreteLoggable</i>	the object being logged [CRTP parameter]
<i>PolymorphicBase</i>	the polymorphic base of this class. By default it is Loggable . Change it if you want to use a new superclass of Loggable as polymorphic base of this class.

24.35.2 Member Function Documentation

24.35.2.1 add_logger()

```
template<typename ConcreteLoggable, typename PolymorphicBase = Loggable>
void gko::log::EnableLogging< ConcreteLoggable, PolymorphicBase >::add_logger (
    std::shared_ptr< const Logger > logger ) [inline], [override], [virtual]
```

Adds a new logger to the list of subscribed loggers.

Parameters

<i>logger</i>	the logger to add
---------------	-------------------

Implements [gko::log::Loggable](#).

```
524     {
525         loggers_.push_back(logger);
526     }
```

24.35.2.2 remove_logger()

```
template<typename ConcreteLoggable, typename PolymorphicBase = Loggable>
void gko::log::EnableLogging< ConcreteLoggable, PolymorphicBase >::remove_logger (
    const Logger * logger ) [inline], [override], [virtual]
```

Removes a logger from the list of subscribed loggers.

Parameters

<i>logger</i>	the logger to remove
---------------	----------------------

Note

The comparison is done using the logger's object unique identity. Thus, two loggers constructed in the same way are not considered equal.

Implements [gko::log::Loggable](#).

```
529     {
530         auto idx = find_if(begin(loggers_), end(loggers_),
531             [&logger](std::shared_ptr<const Logger> l) {
532                 return lend(l) == logger;
533             });
534         if (idx != end(loggers_)) {
535             loggers_.erase(idx);
536         } else {
537             throw OutOfBoundsError(__FILE__, __LINE__, loggers_.size(),
538                 loggers_.size());
539         }
540     }
```

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

- ginkgo/core/log/logger.hpp (0d7578c9)

24.36 `gko::EnablePolymorphicAssignment< ConcreteType, ResultType >` Class Template Reference

This mixin is used to enable a default `PolymorphicObject::copy_from()` implementation for objects that have implemented conversions between them.

```
#include <ginkgo/core/base/polymorphic_object.hpp>
```

Public Member Functions

- void `convert_to` (result_type *result) const override
Converts the implementer to an object of type result_type.
- void `move_to` (result_type *result) override
Converts the implementer to an object of type result_type by moving data from this object.

24.36.1 Detailed Description

```
template<typename ConcreteType, typename ResultType = ConcreteType>
class gko::EnablePolymorphicAssignment< ConcreteType, ResultType >
```

This mixin is used to enable a default `PolymorphicObject::copy_from()` implementation for objects that have implemented conversions between them.

The requirement is that there is either a conversion constructor from `ConcreteType` in `ResultType`, or a conversion operator to `ResultType` in `ConcreteType`.

Template Parameters

<i>ConcreteType</i>	the concrete type from which the copy_from is being enabled [CRTP parameter]
<i>ResultType</i>	the type to which copy_from is being enabled

24.36.2 Member Function Documentation

24.36.2.1 `convert_to()`

```
template<typename ConcreteType, typename ResultType = ConcreteType>
void gko::EnablePolymorphicAssignment< ConcreteType, ResultType >::convert_to (
    result_type * result ) const [inline], [override], [virtual]
```

Converts the implementer to an object of type result_type.

Parameters

<i>result</i>	the object used to store the result of the conversion
---------------	---

Implements [gko::ConvertibleTo< ResultType >](#).

```
558 { *result = *self(); }
```

24.36.2.2 move_to()

```
template<typename ConcreteType, typename ResultType = ConcreteType>
void gko::EnablePolymorphicAssignment< ConcreteType, ResultType >::move_to (
    result_type * result ) [inline], [override], [virtual]
```

Converts the implementer to an object of type result_type by moving data from this object.

This method is used when the implementer is a temporary object, and move semantics can be used.

Parameters

<i>result</i>	the object used to emplace the result of the conversion
---------------	---

Note

[ConvertibleTo::move_to](#) can be implemented by simply calling [ConvertibleTo::convert_to](#). However, this operation can often be optimized by exploiting the fact that implementer's data can be moved to the result.

Implements [gko::ConvertibleTo< ResultType >](#).

```
560 { *result = std::move(*self()); }
```

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

- ginkgo/core/base/polymorphic_object.hpp (4bde4271)

24.37 gko::EnablePolymorphicObject< ConcreteObject, PolymorphicBase > Class Template Reference

This mixin inherits from (a subclass of) [PolymorphicObject](#) and provides a base implementation of a new concrete polymorphic object.

```
#include <ginkgo/core/base/polymorphic_object.hpp>
```

24.37.1 Detailed Description

```
template<typename ConcreteObject, typename PolymorphicBase = PolymorphicObject>
class gko::EnablePolymorphicObject< ConcreteObject, PolymorphicBase >
```

This mixin inherits from (a subclass of) [PolymorphicObject](#) and provides a base implementation of a new concrete polymorphic object.

The mixin changes parameter and return types of appropriate public methods of [PolymorphicObject](#) in the same way [EnableAbstractPolymorphicObject](#) does. In addition, it also provides default implementations of [PolymorphicObject](#)'s virtual methods by using the *executor default constructor* and the assignment operator of `ConcreteObject`. Consequently, the following is a minimal example of [PolymorphicObject](#):

```
{c++}
struct MyObject : EnablePolymorphicObject<MyObject> {
    MyObject (std::shared_ptr<const Executor> exec)
        : EnablePolymorphicObject<MyObject> (std::move(exec))
    {}
};
```

In a way, this mixin can be viewed as an extension of default constructor/destructor/assignment operators.

Note

This mixin does not enable copying the polymorphic object to the object of the same type (i.e. it does not implement the `ConvertibleTo<ConcreteObject>` interface). To enable a default implementation of this interface see the [EnablePolymorphicAssignment](#) mixin.

Template Parameters

<i>ConcreteObject</i>	the concrete type which is being implemented [CRTP parameter]
<i>PolymorphicBase</i>	parent of <code>ConcreteObject</code> in the polymorphic hierarchy, has to be a subclass of polymorphic object

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

- `ginkgo/core/base/polymorphic_object.hpp` (4bde4271)

24.38 gko::Error Class Reference

The [Error](#) class is used to report exceptional behaviour in library functions.

```
#include <ginkgo/core/base/exception.hpp>
```

Public Member Functions

- [Error](#) (const std::string &file, int line, const std::string &what)
Initializes an error.
- virtual const char * [what](#) () const noexcept override
Returns a human-readable string with a more detailed description of the error.

24.38.1 Detailed Description

The [Error](#) class is used to report exceptional behaviour in library functions.

Ginkgo uses C++ exception mechanism to this end, and the [Error](#) class represents a base class for all types of errors. The exact list of errors which could occur during the execution of a certain library routine is provided in the documentation of that routine, along with a short description of the situation when that error can occur. During runtime, these errors can be detected by using standard C++ try-catch blocks, and a human-readable error description can be obtained by calling the [Error::what\(\)](#) method.

As an example, trying to compute a matrix-vector product with arguments of incompatible size will result in a [DimensionMismatch](#) error, which is demonstrated in the following program.

```
#include <ginkgo.h>
#include <iostream>

using namespace gko;

int main()
{
    auto omp = create<OmpExecutor>();
    auto A = randn_fill<matrix::Csr<float>>>(5, 5, 0f, 1f, omp);
    auto x = fill<matrix::Dense<float>>>(6, 1, 1f, omp);
    try {
        auto y = apply(A.get(), x.get());
    } catch(Error e) {
        // an error occurred, write the message to screen and exit
        std::cout << e.what() << std::endl;
        return -1;
    }
    return 0;
}
```

24.38.2 Constructor & Destructor Documentation

24.38.2.1 Error()

```
gko::Error::Error (
    const std::string & file,
    int line,
    const std::string & what ) [inline]
```

Initializes an error.

Parameters

<i>file</i>	The name of the offending source file
<i>line</i>	The source code line number where the error occurred
<i>what</i>	The error message

```
95         : what_(file + ":" + std::to_string(line) + ": " + what)
96     {}
```

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

- ginkgo/core/base/exception.hpp (8045ac75)

24.39 gko::Executor Class Reference

The first step in using the Ginkgo library consists of creating an executor.

```
#include <ginkgo/core/base/executor.hpp>
```

Public Member Functions

- virtual void [run](#) (const [Operation](#) &op) const =0
Runs the specified [Operation](#) using this [Executor](#).
- template<typename ClosureOmp , typename ClosureCuda >
void [run](#) (const ClosureOmp &op_omp, const ClosureCuda &op_cuda) const
Runs one of the passed in functors, depending on the [Executor](#) type.
- template<typename T >
T * [alloc](#) ([size_type](#) num_elems) const
Allocates memory in this [Executor](#).
- void [free](#) (void *ptr) const noexcept
Frees memory previously allocated with [Executor::alloc\(\)](#).
- template<typename T >
void [copy_from](#) (const [Executor](#) *src_exec, [size_type](#) num_elems, const T *src_ptr, T *dest_ptr) const
Copies data from another [Executor](#).
- virtual std::shared_ptr< [Executor](#) > [get_master](#) () noexcept=0
Returns the master [OmpExecutor](#) of this [Executor](#).
- virtual std::shared_ptr< const [Executor](#) > [get_master](#) () const noexcept=0
Returns the master [OmpExecutor](#) of this [Executor](#).
- virtual void [synchronize](#) () const =0
Synchronize the operations launched on the executor with its master.

24.39.1 Detailed Description

The first step in using the Ginkgo library consists of creating an executor.

Executors are used to specify the location for the data of linear algebra objects, and to determine where the operations will be executed. Ginkgo currently supports three different executor types:

- [OmpExecutor](#) specifies that the data should be stored and the associated operations executed on an OpenMP-supporting device (e.g. host CPU);
- [CudaExecutor](#) specifies that the data should be stored and the operations executed on the NVIDIA GPU accelerator;
- [ReferenceExecutor](#) executes a non-optimized reference implementation, which can be used to debug the library.

The following code snippet demonstrates the simplest possible use of the Ginkgo library:

```
auto omp = gko::create<gko::OmpExecutor>();
auto A = gko::read_from_mtx<gko::matrix::Csr<float>>("A.mtx", omp);
```

First, we create a OMP executor, which will be used in the next line to specify where we want the data for the matrix A to be stored. The second line will read a matrix from the matrix market file 'A.mtx', and store the data on the CPU in CSR format (`gko::matrix::Csr` is a Ginkgo matrix class which stores its data in CSR format). At this point, matrix A is bound to the CPU, and any routines called on it will be performed on the CPU. This approach is usually desired in sparse linear algebra, as the cost of individual operations is several orders of magnitude lower than the cost of copying the matrix to the GPU.

If matrix A is going to be reused multiple times, it could be beneficial to copy it over to the accelerator, and perform the operations there, as demonstrated by the next code snippet:

```
auto cuda = gko::create<gko::CudaExecutor>(0, omp);
auto dA = gko::copy_to<gko::matrix::Csr<float>>(A.get(), cuda);
```

The first line of the snippet creates a new CUDA executor. Since there may be multiple NVIDIA GPUs present on the system, the first parameter instructs the library to use the first device (i.e. the one with device ID zero, as in `cudaSetDevice()` routine from the CUDA runtime API). In addition, since GPUs are not stand-alone processors, it is required to pass a "master" `OmpExecutor` which will be used to schedule the requested CUDA kernels on the accelerator.

The second command creates a copy of the matrix A on the GPU. Notice the use of the `get()` method. As Ginkgo aims to provide automatic memory management of its objects, the result of calling `gko::read_from_mtx()` is a smart pointer (`std::unique_ptr`) to the created object. On the other hand, as the library will not hold a reference to A once the copy is completed, the input parameter for `gko::copy_to()` is a plain pointer. Thus, the `get()` method is used to convert from a `std::unique_ptr` to a plain pointer, as expected by `gko::copy_to()`.

As a side note, the `gko::copy_to` routine is far more powerful than just copying data between different devices. It can also be used to convert data between different formats. For example, if the above code used `gko::matrix::Ell` as the template parameter, dA would be stored on the GPU, in ELLPACK format.

Finally, if all the processing of the matrix is supposed to be done on the GPU, and a CPU copy of the matrix is not required, we could have read the matrix to the GPU directly:

```
auto omp = gko::create<gko::OmpExecutor>();
auto cuda = gko::create<gko::CudaExecutor>(0, omp);
auto dA = gko::read_from_mtx<gko::matrix::Csr<float>>("A.mtx", cuda);
```

Notice that even though reading the matrix directly from a file to the accelerator is not supported, the library is designed to abstract away the intermediate step of reading the matrix to the CPU memory. This is a general design approach taken by the library: in case an operation is not supported by the device, the data will be copied to the CPU, the operation performed there, and finally the results copied back to the device. This approach makes using the library more concise, as explicit copies are not required by the user. Nevertheless, this feature should be taken into account when considering performance implications of using such operations.

24.39.2 Member Function Documentation

24.39.2.1 alloc()

```
template<typename T>
T* gko::Executor::alloc (
    size_type num_elems ) const [inline]
```

Allocates memory in this `Executor`.

Template Parameters

<i>T</i>	datatype to allocate
----------	----------------------

Parameters

<i>num_elems</i>	number of elements of type <i>T</i> to allocate
------------------	---

Exceptions

<i>AllocationError</i>	if the allocation failed
--	--------------------------

Returns

pointer to allocated memory

```

460     {
461         this->template log<log::Logger::allocation_started>(
462             this, num_elems * sizeof(T));
463         T *allocated = static_cast<T *>(this->raw_alloc(num_elems * sizeof(T)));
464         this->template log<log::Logger::allocation_completed>(
465             this, num_elems * sizeof(T), reinterpret_cast<uintptr>(allocated));
466         return allocated;
467     }

```

24.39.2.2 copy_from()

```

template<typename T >
void gko::Executor::copy_from (
    const Executor * src_exec,
    size_type num_elems,
    const T * src_ptr,
    T * dest_ptr ) const [inline]

```

Copies data from another [Executor](#).

Template Parameters

<i>T</i>	datatype to copy
----------	------------------

Parameters

<i>src_exec</i>	Executor from which the memory will be copied
<i>num_elems</i>	number of elements of type <i>T</i> to copy
<i>src_ptr</i>	pointer to a block of memory containing the data to be copied
<i>dest_ptr</i>	pointer to an allocated block of memory where the data will be copied to

```

500     {

```

```

501         this->template log<log::Logger::copy_started>(
502             src_exec, this, reinterpret_cast<uintptr>(src_ptr),
503             reinterpret_cast<uintptr>(dest_ptr), num_elems * sizeof(T));
504         this->raw_copy_from(src_exec, num_elems * sizeof(T), src_ptr, dest_ptr);
505         this->template log<log::Logger::copy_completed>(
506             src_exec, this, reinterpret_cast<uintptr>(src_ptr),
507             reinterpret_cast<uintptr>(dest_ptr), num_elems * sizeof(T));
508     }

```

24.39.2.3 free()

```

void gko::Executor::free (
    void * ptr ) const [inline], [noexcept]

```

Frees memory previously allocated with [Executor::alloc\(\)](#).

If `ptr` is a `nullptr`, the function has no effect.

Parameters

<i>ptr</i>	pointer to the allocated memory block
------------	---------------------------------------

```

477     {
478         this->template log<log::Logger::free_started>(
479             this, reinterpret_cast<uintptr>(ptr));
480         this->raw_free(ptr);
481         this->template log<log::Logger::free_completed>(
482             this, reinterpret_cast<uintptr>(ptr));
483     }

```

24.39.2.4 get_master() [1/2]

```
virtual std::shared_ptr<Executor> gko::Executor::get_master ( ) [pure virtual], [noexcept]
```

Returns the master [OmpExecutor](#) of this [Executor](#).

Returns

the master [OmpExecutor](#) of this [Executor](#).

Implemented in [gko::CudaExecutor](#), and [gko::OmpExecutor](#).

24.39.2.5 get_master() [2/2]

```
virtual std::shared_ptr<const Executor> gko::Executor::get_master ( ) const [pure virtual],
[noexcept]
```

Returns the master [OmpExecutor](#) of this [Executor](#).

Returns

the master [OmpExecutor](#) of this [Executor](#).

Implemented in [gko::CudaExecutor](#), and [gko::OmpExecutor](#).

24.39.2.6 `run()` [1/2]

```
virtual void gko::Executor::run (
    const Operation & op ) const [pure virtual]
```

Runs the specified [Operation](#) using this [Executor](#).

Parameters

<i>op</i>	the operation to run
-----------	----------------------

Implemented in [gko::CudaExecutor](#), and [gko::ReferenceExecutor](#).

24.39.2.7 `run()` [2/2]

```
template<typename ClosureOmp , typename ClosureCuda >
void gko::Executor::run (
    const ClosureOmp & op_omp,
    const ClosureCuda & op_cuda ) const [inline]
```

Runs one of the passed in functors, depending on the [Executor](#) type.

Template Parameters

<i>ClosureOmp</i>	type of op_omp
<i>ClosureCuda</i>	type of op_cuda

Parameters

<i>op_omp</i>	functor to run in case of a OmpExecutor or ReferenceExecutor
<i>op_cuda</i>	functor to run in case of a CudaExecutor

```
442     {
443         LambdaOperation<ClosureOmp, ClosureCuda> op(op_omp, op_cuda);
444         this->run(op);
445     }
```

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

- `ginkgo/core/base/executor.hpp` (f1a4eb68)

24.40 `gko::log::executor_data` Struct Reference

Struct representing [Executor](#) related data.

```
#include <ginkgo/core/log/record.hpp>
```

24.40.1 Detailed Description

Struct representing [Executor](#) related data.

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

- ginkgo/core/log/record.hpp (f0a50f96)

24.41 gko::executor_deleter< T > Class Template Reference

This is a deleter that uses an executor's `free` method to deallocate the data.

```
#include <ginkgo/core/base/executor.hpp>
```

Public Member Functions

- [executor_deleter](#) (std::shared_ptr< const [Executor](#) > exec)
Creates a new deleter.
- void [operator\(\)](#) (pointer ptr) const
Deletes the object.

24.41.1 Detailed Description

```
template<typename T>
class gko::executor_deleter< T >
```

This is a deleter that uses an executor's `free` method to deallocate the data.

Template Parameters

<i>T</i>	the type of object being deleted
----------	----------------------------------

24.41.2 Constructor & Destructor Documentation

24.41.2.1 executor_deleter()

```
template<typename T >
gko::executor_deleter< T >::executor_deleter (
    std::shared_ptr< const Executor > exec ) [inline], [explicit]
```

Creates a new deleter.

Parameters

<code>exec</code>	the executor used to free the data
-------------------	------------------------------------

```

638         : exec_{exec}
639     {}

```

24.41.3 Member Function Documentation

24.41.3.1 operator()

```

template<typename T >
void gko::executor_deleter< T >::operator() (
    pointer ptr ) const [inline]

```

Deletes the object.

Parameters

<code>ptr</code>	pointer to the object being deleted
------------------	-------------------------------------

```

647     {
648         if (exec_) {
649             exec_>free(ptr);
650         }
651     }

```

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

- ginkgo/core/base/executor.hpp (f1a4eb68)

24.42 gko::solver::Fcg< ValueType > Class Template Reference

FCG or the flexible conjugate gradient method is an iterative type Krylov subspace method which is suitable for symmetric positive definite methods.

```
#include <ginkgo/core/solver/fcg.hpp>
```

Public Member Functions

- `std::shared_ptr< const LinOp > get_system_matrix () const`
Gets the system operator (matrix) of the linear system.
- `std::shared_ptr< const LinOp > get_preconditioner () const override`
Returns the preconditioner operator used by the solver.

24.42.1 Detailed Description

```
template<typename ValueType = default_precision>
class gko::solver::Fcg< ValueType >
```

FCG or the flexible conjugate gradient method is an iterative type Krylov subspace method which is suitable for symmetric positive definite methods.

Though this method performs very well for symmetric positive definite matrices, it is in general not suitable for general matrices.

In contrast to the standard CG based on the Polack-Ribiere formula, the flexible CG uses the Fletcher-Reeves formula for creating the orthonormal vectors spanning the Krylov subspace. This increases the computational cost of every Krylov solver iteration but allows for non-constant preconditioners.

The implementation in Ginkgo makes use of the merged kernel to make the best use of data locality. The inner operations in one iteration of FCG are merged into 2 separate steps.

Template Parameters

<i>ValueType</i>	precision of matrix elements
------------------	------------------------------

24.42.2 Member Function Documentation

24.42.2.1 get_preconditioner()

```
template<typename ValueType = default_precision>
std::shared_ptr<const LinOp> gko::solver::Fcg< ValueType >::get_preconditioner ( ) const
[inline], [override], [virtual]
```

Returns the preconditioner operator used by the solver.

Returns

the preconditioner operator used by the solver

Implements [gko::Preconditionable](#).

References [gko::stop::combine\(\)](#), [gko::PolymorphicObject::get_executor\(\)](#), [GKO_CREATE_FACTORY_PARAMETERS](#), [GKO_ENABLE_BUILD_METHOD](#), [GKO_ENABLE_LIN_OP_FACTORY](#), [GKO_FACTORY_PARAMETER](#), and [gko::transpose\(\)](#).

```
99     {
100         return preconditioner_;
101     }
```

24.42.2.2 `get_system_matrix()`

```
template<typename ValueType = default_precision>
std::shared_ptr<const LinOp> gko::solver::Fcg< ValueType >::get_system_matrix ( ) const [inline]
```

Gets the system operator (matrix) of the linear system.

Returns

the system operator (matrix)

```
89     {
90         return system_matrix_;
91     }
```

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

- ginkgo/core/solver/fcg.hpp (f1a4eb68)

24.43 `gko::solver::Gmres< ValueType >` Class Template Reference

GMRES or the generalized minimal residual method is an iterative type Krylov subspace method which is suitable for nonsymmetric linear systems.

```
#include <ginkgo/core/solver/gmres.hpp>
```

Public Member Functions

- `std::shared_ptr< const LinOp > get_system_matrix () const`
Gets the system operator (matrix) of the linear system.
- `std::shared_ptr< const LinOp > get_preconditioner () const override`
Returns the preconditioner operator used by the solver.
- `size_type get_krylov_dim () const`
Returns the krylov dimension.

24.43.1 Detailed Description

```
template<typename ValueType = default_precision>
class gko::solver::Gmres< ValueType >
```

GMRES or the generalized minimal residual method is an iterative type Krylov subspace method which is suitable for nonsymmetric linear systems.

The implementation in Ginkgo makes use of the merged kernel to make the best use of data locality. The inner operations in one iteration of GMRES are merged into 2 separate steps.

Template Parameters

<i>ValueType</i>	precision of matrix elements
------------------	------------------------------

24.43.2 Member Function Documentation

24.43.2.1 get_krylov_dim()

```
template<typename ValueType = default_precision>
size_type gko::solver::Gmres< ValueType >::get_krylov_dim ( ) const [inline]
```

Returns the krylov dimension.

Returns

the krylov dimension

References [gko::stop::combine\(\)](#), [gko::PolymorphicObject::get_executor\(\)](#), [GKO_CREATE_FACTORY_PARAMETERS](#), [GKO_ENABLE_BUILD_METHOD](#), [GKO_ENABLE_LIN_OP_FACTORY](#), [GKO_FACTORY_PARAMETER](#), and [gko::transpose\(\)](#).

```
103 { return krylov_dim_; }
```

24.43.2.2 get_preconditioner()

```
template<typename ValueType = default_precision>
std::shared_ptr<const LinOp> gko::solver::Gmres< ValueType >::get_preconditioner ( ) const
[inline], [override], [virtual]
```

Returns the preconditioner operator used by the solver.

Returns

the preconditioner operator used by the solver

Implements [gko::Preconditionable](#).

```
94 {
95     return preconditioner_;
96 }
```

24.43.2.3 get_system_matrix()

```
template<typename ValueType = default_precision>
std::shared_ptr<const LinOp> gko::solver::Gmres< ValueType >::get_system_matrix ( ) const
[inline]
```

Gets the system operator (matrix) of the linear system.

Returns

the system operator (matrix)

```
84     {
85         return system_matrix_;
86     }
```

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

- ginkgo/core/solver/gmres.hpp (f1a4eb68)

24.44 gko::matrix::Hybrid< ValueType, IndexType > Class Template Reference

HYBRID is a matrix format which splits the matrix into ELLPACK and COO format.

```
#include <ginkgo/core/matrix/hybrid.hpp>
```

Classes

- class [automatic](#)
automatic is a stratgy_type which decides the number of stored elements per row of the ell part automatically.
- class [column_limit](#)
column_limit is a strategy_type which decides the number of stored elements per row of the ell part by specifying the number of columns.
- class [imbalance_bounded_limit](#)
imbalance_bounded_limit is a stratgy_type which decides the number of stored elements per row of the ell part.
- class [imbalance_limit](#)
imbalance_limit is a strategy_type which decides the number of stored elements per row of the ell part according to the percent.
- class [minimal_storage_limit](#)
minimal_storage_limit is a stratgy_type which decides the number of stored elements per row of the ell part.
- class [strategy_type](#)
strategy_type is to decide how to set the hybrid config.

Public Member Functions

- void [read](#) (const [mat_data](#) &data) override
Reads a matrix from a [matrix_data](#) structure.
- void [write](#) ([mat_data](#) &data) const override
Writes a matrix to a [matrix_data](#) structure.
- value_type * [get_ell_values](#) () noexcept
Returns the values of the ell part.
- const value_type * [get_const_ell_values](#) () const noexcept
Returns the values of the ell part.
- index_type * [get_ell_col_idxs](#) () noexcept
Returns the column indexes of the ell part.
- const index_type * [get_const_ell_col_idxs](#) () const noexcept
Returns the column indexes of the ell part.
- size_type [get_ell_num_stored_elements_per_row](#) () const noexcept
Returns the number of stored elements per row of ell part.
- size_type [get_ell_stride](#) () const noexcept
Returns the stride of the ell part.
- size_type [get_ell_num_stored_elements](#) () const noexcept
Returns the number of elements explicitly stored in the ell part.
- value_type & [ell_val_at](#) (size_type row, size_type idx) noexcept
*Returns the *idx*-th non-zero element of the *row*-th row in the ell part.*
- value_type [ell_val_at](#) (size_type row, size_type idx) const noexcept
*Returns the *idx*-th non-zero element of the *row*-th row in the ell part.*
- index_type & [ell_col_at](#) (size_type row, size_type idx) noexcept
*Returns the *idx*-th column index of the *row*-th row in the ell part.*
- index_type [ell_col_at](#) (size_type row, size_type idx) const noexcept
*Returns the *idx*-th column index of the *row*-th row in the ell part.*
- const ell_type * [get_ell](#) () const noexcept
Returns the matrix of the ell part.
- value_type * [get_coo_values](#) () noexcept
Returns the values of the coo part.
- const value_type * [get_const_coo_values](#) () const noexcept
Returns the values of the coo part.
- index_type * [get_coo_col_idxs](#) () noexcept
Returns the column indexes of the coo part.
- const index_type * [get_const_coo_col_idxs](#) () const noexcept
Returns the column indexes of the coo part.
- index_type * [get_coo_row_idxs](#) () noexcept
Returns the row indexes of the coo part.
- const index_type * [get_const_coo_row_idxs](#) () const noexcept
Returns the row indexes of the coo part.
- size_type [get_coo_num_stored_elements](#) () const noexcept
Returns the number of elements explicitly stored in the coo part.
- const coo_type * [get_coo](#) () const noexcept
Returns the matrix of the coo part.
- size_type [get_num_stored_elements](#) () const noexcept
Returns the number of elements explicitly stored in the matrix.
- std::shared_ptr< [strategy_type](#) > [get_strategy](#) () const noexcept
Returns the strategy.
- [Hybrid](#) & [operator=](#) (const [Hybrid](#) &other)
Copies data from another [Hybrid](#).

24.44.1 Detailed Description

```
template<typename ValueType = default_precision, typename IndexType = int32>
class gko::matrix::Hybrid< ValueType, IndexType >
```

HYBRID is a matrix format which splits the matrix into ELLPACK and COO format.

Achieve the excellent performance with a proper partition of ELLPACK and COO.

Template Parameters

<i>ValueType</i>	precision of matrix elements
<i>IndexType</i>	precision of matrix indexes

24.44.2 Member Function Documentation

24.44.2.1 `ell_col_at()` [1/2]

```
template<typename ValueType = default_precision, typename IndexType = int32>
index_type& gko::matrix::Hybrid< ValueType, IndexType >::ell_col_at (
    size_type row,
    size_type idx ) [inline], [noexcept]
```

Returns the `idx`-th column index of the `row`-th row in the ell part.

Parameters

<i>row</i>	the row of the requested element
<i>idx</i>	the <code>idx</code> -th stored element of the row

Note

the method has to be called on the same [Executor](#) the matrix is stored at (e.g. trying to call this method on a GPU matrix from the OMP results in a runtime error)

```
445 {
446     return ell->col_at(row, idx);
447 }
```

24.44.2.2 `ell_col_at()` [2/2]

```
template<typename ValueType = default_precision, typename IndexType = int32>
index_type gko::matrix::Hybrid< ValueType, IndexType >::ell_col_at (
    size_type row,
    size_type idx ) const [inline], [noexcept]
```

Returns the `idx`-th column index of the `row`-th row in the ell part.

Parameters

<i>row</i>	the row of the requested element
<i>idx</i>	the idx-th stored element of the row

Note

the method has to be called on the same [Executor](#) the matrix is stored at (e.g. trying to call this method on a GPU matrix from the OMP results in a runtime error)

```

453     {
454         return ell->col_at(row, idx);
455     }

```

24.44.2.3 ell_val_at() [1/2]

```

template<typename ValueType = default_precision, typename IndexType = int32>
value_type& gko::matrix::Hybrid< ValueType, IndexType >::ell_val_at (
    size_type row,
    size_type idx ) [inline], [noexcept]

```

Returns the *idx*-th non-zero element of the *row*-th row in the ell part.

Parameters

<i>row</i>	the row of the requested element
<i>idx</i>	the idx-th stored element of the row

Note

the method has to be called on the same [Executor](#) the matrix is stored at (e.g. trying to call this method on a GPU matrix from the OMP results in a runtime error)

```

422     {
423         return ell->val_at(row, idx);
424     }

```

24.44.2.4 ell_val_at() [2/2]

```

template<typename ValueType = default_precision, typename IndexType = int32>
value_type gko::matrix::Hybrid< ValueType, IndexType >::ell_val_at (
    size_type row,
    size_type idx ) const [inline], [noexcept]

```

Returns the *idx*-th non-zero element of the *row*-th row in the ell part.

Parameters

<i>row</i>	the row of the requested element
<i>idx</i>	the idx-th stored element of the row

Note

the method has to be called on the same [Executor](#) the matrix is stored at (e.g. trying to call this method on a GPU matrix from the OMP results in a runtime error)

```

430     {
431         return ell->val_at(row, idx);
432     }
```

24.44.2.5 get_const_coo_col_idxs()

```

template<typename ValueType = default_precision, typename IndexType = int32>
const index_type* gko::matrix::Hybrid< ValueType, IndexType >::get_const_coo_col_idxs ( )
const [inline], [noexcept]
```

Returns the column indexes of the coo part.

Returns

the column indexes of the coo part.

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

```

498     {
499         return coo->get_const_col_idxs();
500     }
```

24.44.2.6 get_const_coo_row_idxs()

```

template<typename ValueType = default_precision, typename IndexType = int32>
const index_type* gko::matrix::Hybrid< ValueType, IndexType >::get_const_coo_row_idxs ( )
const [inline], [noexcept]
```

Returns the row indexes of the coo part.

Returns

the row indexes of the coo part.

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

```

517     {
518         return coo->get_const_row_idxs();
519     }
```


24.44.2.7 get_const_coo_values()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const value_type* gko::matrix::Hybrid< ValueType, IndexType >::get_const_coo_values ( ) const
[inline], [noexcept]
```

Returns the values of the coo part.

Returns

the values of the coo part.

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

```
479     {
480         return coo->get_const_values();
481     }
```

24.44.2.8 get_const_ell_col_idxs()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const index_type* gko::matrix::Hybrid< ValueType, IndexType >::get_const_ell_col_idxs ( )
const [inline], [noexcept]
```

Returns the column indexes of the ell part.

Returns

the column indexes of the ell part

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

```
379     {
380         return ell->get_const_col_idxs();
381     }
```

24.44.2.9 get_const_ell_values()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const value_type* gko::matrix::Hybrid< ValueType, IndexType >::get_const_ell_values ( ) const
[inline], [noexcept]
```

Returns the values of the ell part.

Returns

the values of the ell part

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

```
360 {
361     return ell_>get_const_values();
362 }
```

24.44.2.10 get_coo()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const coo_type* gko::matrix::Hybrid< ValueType, IndexType >::get_coo ( ) const [inline],
[noexcept]
```

Returns the matrix of the coo part.

Returns

the matrix of the coo part

```
536 { return coo_.get(); }
```

24.44.2.11 get_coo_col_idxs()

```
template<typename ValueType = default_precision, typename IndexType = int32>
index_type* gko::matrix::Hybrid< ValueType, IndexType >::get_coo_col_idxs ( ) [inline], [noexcept]
```

Returns the column indexes of the coo part.

Returns

the column indexes of the coo part.

```
488 { return coo_>get_col_idxs(); }
```

24.44.2.12 get_coo_num_stored_elements()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Hybrid< ValueType, IndexType >::get_coo_num_stored_elements ( ) const
[inline], [noexcept]
```

Returns the number of elements explicitly stored in the coo part.

Returns

the number of elements explicitly stored in the coo part

```
527     {
528         return coo_->get_num_stored_elements();
529     }
```

24.44.2.13 get_coo_row_idxs()

```
template<typename ValueType = default_precision, typename IndexType = int32>
index_type* gko::matrix::Hybrid< ValueType, IndexType >::get_coo_row_idxs ( ) [inline], [noexcept]
```

Returns the row indexes of the coo part.

Returns

the row indexes of the coo part.

```
507 { return coo_->get_row_idxs(); }
```

24.44.2.14 get_coo_values()

```
template<typename ValueType = default_precision, typename IndexType = int32>
value_type* gko::matrix::Hybrid< ValueType, IndexType >::get_coo_values ( ) [inline], [noexcept]
```

Returns the values of the coo part.

Returns

the values of the coo part.

```
469 { return coo_->get_values(); }
```

24.44.2.15 get_ell()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const ell_type* gko::matrix::Hybrid< ValueType, IndexType >::get_ell ( ) const [inline],
[noexcept]
```

Returns the matrix of the ell part.

Returns

the matrix of the ell part

```
462 { return ell_.get(); }
```

24.44.2.16 get_ell_col_idxs()

```
template<typename ValueType = default_precision, typename IndexType = int32>
index_type* gko::matrix::Hybrid< ValueType, IndexType >::get_ell_col_idxs ( ) [inline], [noexcept]
```

Returns the column indexes of the ell part.

Returns

the column indexes of the ell part

```
369 { return ell_->get_col_idxs(); }
```

24.44.2.17 get_ell_num_stored_elements()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Hybrid< ValueType, IndexType >::get_ell_num_stored_elements ( ) const
[inline], [noexcept]
```

Returns the number of elements explicitly stored in the ell part.

Returns

the number of elements explicitly stored in the ell part

```
406 {
407     return ell_->get_num_stored_elements();
408 }
```

24.44.2.18 get_ell_num_stored_elements_per_row()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Hybrid< ValueType, IndexType >::get_ell_num_stored_elements_per_row ( )
const [inline], [noexcept]
```

Returns the number of stored elements per row of ell part.

Returns

the number of stored elements per row of ell part

```
389     {
390         return ell_->get_num_stored_elements_per_row();
391     }
```

24.44.2.19 get_ell_stride()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Hybrid< ValueType, IndexType >::get_ell_stride ( ) const [inline],
[noexcept]
```

Returns the stride of the ell part.

Returns

the stride of the ell part

```
398 { return ell_->get_stride(); }
```

24.44.2.20 get_ell_values()

```
template<typename ValueType = default_precision, typename IndexType = int32>
value_type* gko::matrix::Hybrid< ValueType, IndexType >::get_ell_values ( ) [inline], [noexcept]
```

Returns the values of the ell part.

Returns

the values of the ell part

```
350 { return ell_->get_values(); }
```

24.44.2.21 get_num_stored_elements()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Hybrid< ValueType, IndexType >::get_num_stored_elements ( ) const
[inline], [noexcept]
```

Returns the number of elements explicitly stored in the matrix.

Returns

the number of elements explicitly stored in the matrix

```
544     {
545         return coo->get_num_stored_elements() +
546            ell->get_num_stored_elements();
547     }
```

24.44.2.22 get_strategy()

```
template<typename ValueType = default_precision, typename IndexType = int32>
std::shared_ptr<strategy_type> gko::matrix::Hybrid< ValueType, IndexType >::get_strategy ( )
const [inline], [noexcept]
```

Returns the strategy.

Returns

the strategy

```
555     {
556         return strategy_;
557     }
```

24.44.2.23 operator=()

```
template<typename ValueType = default_precision, typename IndexType = int32>
Hybrid& gko::matrix::Hybrid< ValueType, IndexType >::operator= (
    const Hybrid< ValueType, IndexType > & other ) [inline]
```

Copies data from another [Hybrid](#).

Parameters

<i>other</i>	the Hybrid to copy from
--------------	---

Returns

this

```

567     {
568         if (&other == this) {
569             return *this;
570         }
571         EnableLinOp<Hybrid<ValueType, IndexType>>::operator=(other);
572         this->coo_->copy_from(other.get_coo());
573         this->ell_->copy_from(other.get_ell());
574         return *this;
575     }

```

24.44.2.24 read()

```

template<typename ValueType = default_precision, typename IndexType = int32>
void gko::matrix::Hybrid< ValueType, IndexType >::read (
    const mat_data & data ) [override], [virtual]

```

Reads a matrix from a [matrix_data](#) structure.

Parameters

<i>data</i>	the matrix_data structure
-------------	---

Implements [gko::ReadableFromMatrixData< ValueType, IndexType >](#).

Referenced by [gko::matrix::Hybrid< ValueType, IndexType >::automatic::compute_ell_num_stored_elements_per_row\(\)](#).

24.44.2.25 write()

```

template<typename ValueType = default_precision, typename IndexType = int32>
void gko::matrix::Hybrid< ValueType, IndexType >::write (
    mat_data & data ) const [override], [virtual]

```

Writes a matrix to a [matrix_data](#) structure.

Parameters

<i>data</i>	the matrix_data structure
-------------	---

Implements [gko::WritableToMatrixData< ValueType, IndexType >](#).

Referenced by [gko::matrix::Hybrid< ValueType, IndexType >::automatic::compute_ell_num_stored_elements_per_row\(\)](#).

The documentation for this class was generated from the following files:

- ginkgo/core/matrix/csr.hpp (ec6dbb34)
- ginkgo/core/matrix/hybrid.hpp (3e51a52b)

24.45 gko::matrix::Identity< ValueType > Class Template Reference

This class is a utility which efficiently implements the identity matrix (a linear operator which maps each vector to itself).

```
#include <ginkgo/core/matrix/identity.hpp>
```

Additional Inherited Members

24.45.1 Detailed Description

```
template<typename ValueType = default_precision>
class gko::matrix::Identity< ValueType >
```

This class is a utility which efficiently implements the identity matrix (a linear operator which maps each vector to itself).

Thus, objects of the [Identity](#) class always represent a square matrix, and don't require any storage for their values. The apply method is implemented as a simple copy (or a linear combination).

Note

This class is useful when composing it with other operators. For example, it can be used instead of a preconditioner in Krylov solvers, if one wants to run a "plain" solver, without using a preconditioner.

Template Parameters

<i>ValueType</i>	precision of matrix elements
------------------	------------------------------

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

- ginkgo/core/matrix/identity.hpp (8045ac75)

24.46 gko::matrix::IdentityFactory< ValueType > Class Template Reference

This factory is a utility which can be used to generate [Identity](#) operators.

```
#include <ginkgo/core/matrix/identity.hpp>
```

Static Public Member Functions

- static std::unique_ptr< [IdentityFactory](#) > [create](#) (std::shared_ptr< const [Executor](#) > exec)
Creates a new [Identity](#) factory.

Additional Inherited Members

24.46.1 Detailed Description

```
template<typename ValueType = default_precision>
class gko::matrix::IdentityFactory< ValueType >
```

This factory is a utility which can be used to generate [Identity](#) operators.

The factory will generate the [Identity](#) matrix with the same dimension as the passed in operator. It will throw an exception if the operator is not square.

Template Parameters

<i>ValueType</i>	precision of matrix elements
------------------	------------------------------

24.46.2 Member Function Documentation

24.46.2.1 create()

```
template<typename ValueType = default_precision>
static std::unique_ptr<IdentityFactory> gko::matrix::IdentityFactory< ValueType >::create (
    std::shared_ptr< const Executor > exec ) [inline], [static]
```

Creates a new [Identity](#) factory.

Parameters

<i>exec</i>	the executor where the Identity operator will be stored
-------------	---

Returns

a unique pointer to the newly created factory

```
129     {
130         return std::unique_ptr<IdentityFactory>{
131             new IdentityFactory(std::move(exec));
132     }
```

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

- ginkgo/core/matrix/identity.hpp (8045ac75)

24.47 gko::matrix::Hybrid< ValueType, IndexType >::imbalance_bounded_limit Class Reference

[imbalance_bounded_limit](#) is a `stratgy_type` which decides the number of stored elements per row of the ell part.

```
#include <ginkgo/core/matrix/hybrid.hpp>
```

Public Member Functions

- [imbalance_bounded_limit](#) (float percent=0.8, float ratio=0.0001)
Creates a [imbalance_bounded_limit](#) strategy.
- [size_type compute_ell_num_stored_elements_per_row](#) (Array< [size_type](#) > *row_nnz) const override
Computes the number of stored elements per row of the ell part.

24.47.1 Detailed Description

```
template<typename ValueType = default_precision, typename IndexType = int32>
class gko::matrix::Hybrid< ValueType, IndexType >::imbalance_bounded_limit
```

[imbalance_bounded_limit](#) is a [stratgy_type](#) which decides the number of stored elements per row of the ell part.

It uses the [imbalance_limit](#) and adds the upper bound of the number of ell's cols by the number of rows.

24.47.2 Member Function Documentation

24.47.2.1 compute_ell_num_stored_elements_per_row()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size\_type gko::matrix::Hybrid< ValueType, IndexType >::imbalance_bounded_limit::compute_ell_
num_stored_elements_per_row (
    Array< size\_type > * row_nnz ) const [inline], [override], [virtual]
```

Computes the number of stored elements per row of the ell part.

Parameters

row_nnz	the number of nonzeros of each row
-------------------------	------------------------------------

Returns

the number of stored elements per row of the ell part

Implements [gko::matrix::Hybrid< ValueType, IndexType >::strategy_type](#).

References [gko::Array< ValueType >::get_num_elems\(\)](#).

```
271         {
272             auto num_rows = row_nnz->get_num_elems();
273             auto ell_cols =
274                 strategy_.compute_ell_num_stored_elements_per_row(
row_nnz);
275             return std::min(ell_cols,
276                             static_cast<size_type>(num_rows * ratio_));
277         }
```

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

- [ginkgo/core/matrix/hybrid.hpp](#) (3e51a52b)

24.48 gko::matrix::Hybrid< ValueType, IndexType >::imbalance_limit Class Reference

`imbalance_limit` is a [strategy_type](#) which decides the number of stored elements per row of the ell part according to the percent.

```
#include <ginkgo/core/matrix/hybrid.hpp>
```

Public Member Functions

- [imbalance_limit](#) (float percent=0.8)
Creates a [imbalance_limit](#) strategy.
- [size_type compute_ell_num_stored_elements_per_row](#) (Array< [size_type](#) > *row_nnz) const override
Computes the number of stored elements per row of the ell part.

24.48.1 Detailed Description

```
template<typename ValueType = default_precision, typename IndexType = int32>
class gko::matrix::Hybrid< ValueType, IndexType >::imbalance_limit
```

`imbalance_limit` is a [strategy_type](#) which decides the number of stored elements per row of the ell part according to the percent.

It sorts the number of nonzeros of each row and takes the value at the position `floor(percent * num_row)` as the number of stored elements per row of the ell part. Thus, at least `percent` rows of all are in the ell part.

24.48.2 Constructor & Destructor Documentation

24.48.2.1 imbalance_limit()

```
template<typename ValueType = default_precision, typename IndexType = int32>
gko::matrix::Hybrid< ValueType, IndexType >::imbalance_limit (
    float percent = 0.8 ) [inline], [explicit]
```

Creates a [imbalance_limit](#) strategy.

Parameters

<i>percent</i>	the <code>row_nnz[floor(num_rows*percent)]</code> is the number of stored elements per row of the ell part
----------------	--

```
231                                     : percent_(percent)
232     {
233         percent_ = std::min(percent_, 1.0f);
234         percent_ = std::max(percent_, 0.0f);
235     }
```

24.48.3 Member Function Documentation

24.48.3.1 compute_ell_num_stored_elements_per_row()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Hybrid< ValueType, IndexType >::imbalance_limit::compute_ell_num_↵
stored_elements_per_row (
    Array< size_type > * row_nnz ) const [inline], [override], [virtual]
```

Computes the number of stored elements per row of the ell part.

Parameters

<code>row_nnz</code>	the number of nonzeros of each row
----------------------	------------------------------------

Returns

the number of stored elements per row of the ell part

Implements [gko::matrix::Hybrid< ValueType, IndexType >::strategy_type](#).

References [gko::Array< ValueType >::get_data\(\)](#), and [gko::Array< ValueType >::get_num_elems\(\)](#).

```
239     {
240         auto row_nnz_val = row_nnz->get_data();
241         auto num_rows = row_nnz->get_num_elems();
242         std::sort(row_nnz_val, row_nnz_val + num_rows);
243         if (percent_ < 1) {
244             auto percent_pos = static_cast<size_type>(num_rows * percent_);
245             return row_nnz_val[percent_pos];
246         } else {
247             return row_nnz_val[num_rows - 1];
248         }
249     }
```

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

- [ginkgo/core/matrix/hybrid.hpp \(3e51a52b\)](#)

24.49 gko::solver::lr< ValueType > Class Template Reference

Iterative refinement (IR) is an iterative method that uses another coarse method to approximate the error of the current solution via the current residual.

```
#include <ginkgo/core/solver/ir.hpp>
```

Public Member Functions

- `std::shared_ptr< const LinOp > get_system_matrix () const`
Returns the system operator (matrix) of the linear system.
- `std::shared_ptr< const LinOp > get_solver () const`
Returns the solver operator used as the inner solver.

24.49.1 Detailed Description

```
template<typename ValueType = default_precision>
class gko::solver::lr< ValueType >
```

Iterative refinement (IR) is an iterative method that uses another coarse method to approximate the error of the current solution via the current residual.

For any approximation of the solution `solution` to the system $Ax = b$, the residual is defined as: `residual = b - A solution`. The error in solution, $e = x - \text{solution}$ (with x being the exact solution) can be obtained as the solution to the residual equation $Ae = \text{residual}$, since $Ae = Ax - A \text{ solution} = b - A \text{ solution} = \text{residual}$. Then, the real solution is computed as $x = \text{solution} + e$. Instead of accurately solving the residual equation $Ae = \text{residual}$, the solution of the system e can be approximated to obtain the approximation error using a coarse method `solver`, which is used to update `solution`, and the entire process is repeated with the updated `solution`. This yields the iterative refinement method:

```
solution = initial_guess
while not converged:
    residual = b - A solution
    error = solver(A, residual)
    solution = solution + error
```

Assuming that `solver` has accuracy `c`, i.e., $|e - \text{error}| \leq c |e|$, iterative refinement will converge with a convergence rate of `c`. Indeed, from $e - \text{error} = x - \text{solution} - \text{error} = x - \text{solution} * \text{inv}(A)b - \text{inv}(A) A \text{ solution} = x - \text{solution}$ it follows that $|x - \text{solution}| \leq c |x - \text{solution}|$.

Unless otherwise specified via the `solver` factory parameter, this implementation uses the identity operator (i.e. the solver that approximates the solution of a system $Ax = b$ by setting $x := b$) as the default inner solver. Such a setting results in a relaxation method known as the Richardson iteration with parameter 1, which is guaranteed to converge for matrices whose spectrum is strictly contained within the unit disc around 1 (i.e., all its eigenvalues `lambda` have to satisfy the equation $|\lambda - 1| < 1$).

Template Parameters

<i>ValueType</i>	precision of matrix elements
------------------	------------------------------

24.49.2 Member Function Documentation

24.49.2.1 get_solver()

```
template<typename ValueType = default_precision>
std::shared_ptr<const LinOp> gko::solver::lr< ValueType >::get_solver ( ) const [inline]
```

Returns the solver operator used as the inner solver.

Returns

the solver operator used as the inner solver

References `gko::stop::combine()`, `gko::PolymorphicObject::get_executor()`, `GKO_CREATE_FACTORY_PARAMETERS`, `GKO_ENABLE_BUILD_METHOD`, `GKO_ENABLE_LIN_OP_FACTORY`, `GKO_FACTORY_PARAMETER`, and `gko::transpose()`.

```
116 { return solver_; }
```

24.49.2.2 `get_system_matrix()`

```
template<typename ValueType = default_precision>
std::shared_ptr<const LinOp> gko::solver::Ir< ValueType >::get_system_matrix ( ) const [inline]
```

Returns the system operator (matrix) of the linear system.

Returns

the system operator (matrix)

```
106 {
107     return system_matrix_;
108 }
```

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

- `ginkgo/core/solver/ir.hpp` (4bde4271)

24.50 `gko::stop::Iteration` Class Reference

The `Iteration` class is a stopping criterion which stops the iteration process after a preset number of iterations.

```
#include <ginkgo/core/stop/iteration.hpp>
```

24.50.1 Detailed Description

The `Iteration` class is a stopping criterion which stops the iteration process after a preset number of iterations.

Note

to use this stopping criterion, it is required to update the iteration count for the `::check()` method.

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

- `ginkgo/core/stop/iteration.hpp` (f1a4eb68)

24.51 gko::log::iteration_complete_data Struct Reference

Struct representing iteration complete related data.

```
#include <ginkgo/core/log/record.hpp>
```

24.51.1 Detailed Description

Struct representing iteration complete related data.

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

- ginkgo/core/log/record.hpp (f0a50f96)

24.52 gko::preconditioner::Jacobi< ValueType, IndexType > Class Template Reference

A block-Jacobi preconditioner is a block-diagonal linear operator, obtained by inverting the diagonal blocks of the source operator.

```
#include <ginkgo/core/preconditioner/jacobi.hpp>
```

Public Member Functions

- [size_type get_num_blocks](#) () const noexcept
Returns the number of blocks of the operator.
- const [block_interleaved_storage_scheme](#)< index_type > & [get_storage_scheme](#) () const noexcept
Returns the storage scheme used for storing [Jacobi](#) blocks.
- const value_type * [get_blocks](#) () const noexcept
Returns the pointer to the memory used for storing the block data.
- const [remove_complex](#)< value_type > * [get_conditioning](#) () const noexcept
Returns an array of 1-norm condition numbers of the blocks.
- [size_type get_num_stored_elements](#) () const noexcept
Returns the number of elements explicitly stored in the matrix.
- void [convert_to](#) ([matrix::Dense](#)< value_type > *result) const override
Converts the implementer to an object of type result_type.
- void [move_to](#) ([matrix::Dense](#)< value_type > *result) override
Converts the implementer to an object of type result_type by moving data from this object.
- void [write](#) ([mat_data](#) &data) const override
Writes a matrix to a [matrix_data](#) structure.

24.52.1 Detailed Description

```
template<typename ValueType = default_precision, typename IndexType = int32>
class gko::preconditioner::Jacobi< ValueType, IndexType >
```

A block-Jacobi preconditioner is a block-diagonal linear operator, obtained by inverting the diagonal blocks of the source operator.

The [Jacobi](#) class implements the inversion of the diagonal blocks using Gauss-Jordan elimination with column pivoting, and stores the inverse explicitly in a customized format.

If the diagonal blocks of the matrix are not explicitly set by the user, the implementation will try to automatically detect the blocks by first finding the natural blocks of the matrix, and then applying the supervariable agglomeration procedure on them. However, if problem-specific knowledge regarding the block diagonal structure is available, it is usually beneficial to explicitly pass the starting rows of the diagonal blocks, as the block detection is merely a heuristic and cannot perfectly detect the diagonal block structure. The current implementation supports blocks of up to 32 rows / columns.

The implementation also includes an improved, adaptive version of the block-Jacobi preconditioner, which can store some of the blocks in lower precision and thus improve the performance of preconditioner application by reducing the amount of memory transfers. This variant can be enabled by setting the `Jacobi::Factory's` `storage←_optimization` parameter. Refer to the documentation of the parameter for more details.

Template Parameters

<i>ValueType</i>	precision of matrix elements
<i>IndexType</i>	integral type used to store pointers to the start of each block

Note

The current implementation supports blocks of up to 32 rows / columns.

When using the adaptive variant, there may be a trade-off in terms of slightly longer preconditioner generation due to extra work required to detect the optimal precision of the blocks.

24.52.2 Member Function Documentation

24.52.2.1 `convert_to()`

```
template<typename ValueType = default_precision, typename IndexType = int32>
void gko::preconditioner::Jacobi< ValueType, IndexType >::convert_to (
    matrix::Dense< value_type > * result ) const [override], [virtual]
```

Converts the implementer to an object of type `result_type`.

Parameters

<i>result</i>	the object used to store the result of the conversion
---------------	---

Implements [gko::ConvertibleTo< matrix::Dense< ValueType > >](#).

24.52.2.2 get_blocks()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const value_type* gko::preconditioner::Jacobi< ValueType, IndexType >::get_blocks ( ) const
[inline], [noexcept]
```

Returns the pointer to the memory used for storing the block data.

Element (i, j) of block b is stored in position (get_block_pointers()[b] + i) * stride + j of the array.

Returns

the pointer to the memory used for storing the block data

```
249     {
250         return blocks_.get_const_data();
251     }
```

24.52.2.3 get_conditioning()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const remove_complex<value_type>* gko::preconditioner::Jacobi< ValueType, IndexType >::get_↵
conditioning ( ) const [inline], [noexcept]
```

Returns an array of 1-norm condition numbers of the blocks.

Returns

an array of 1-norm condition numbers of the blocks

Note

This value is valid only if adaptive precision variant is used, and implementations of the standard non-adaptive variant are allowed to omit the calculation of condition numbers.

```
263     {
264         return conditioning_.get_const_data();
265     }
```

24.52.2.4 get_num_blocks()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::preconditioner::Jacobi< ValueType, IndexType >::get_num_blocks ( ) const [inline],
[noexcept]
```

Returns the number of blocks of the operator.

Returns

the number of blocks of the operator

```
221 { return num_blocks_; }
```

24.52.2.5 get_num_stored_elements()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::preconditioner::Jacobi< ValueType, IndexType >::get_num_stored_elements ( )
const [inline], [noexcept]
```

Returns the number of elements explicitly stored in the matrix.

Returns

the number of elements explicitly stored in the matrix

References `gko::preconditioner::block_interleaved_storage_scheme< IndexType >::block_offset`, `gko::Array< ValueType >::get_num_elems()`, `gko::get_significant_bit()`, `gko::get_superior_power()`, `GKO_CREATE_FACTORY_PARAMETERS`, `GKO_ENABLE_BUILD_METHOD`, `GKO_ENABLE_LIN_OP_FACTORY`, `GKO_FACTORY_PARAMETER`, `gko::preconditioner::block_interleaved_storage_scheme< IndexType >::group_offset`, `gko::lend()`, `gko::transpose()`, and `gko::write()`.

```
273 {
274     return blocks_.get_num_elems();
275 }
```

24.52.2.6 get_storage_scheme()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const block_interleaved_storage_scheme<index_type>& gko::preconditioner::Jacobi< ValueType,
IndexType >::get_storage_scheme ( ) const [inline], [noexcept]
```

Returns the storage scheme used for storing [Jacobi](#) blocks.

Returns

the storage scheme used for storing [Jacobi](#) blocks

```
233 {
234     return storage_scheme_;
235 }
```

24.52.2.7 move_to()

```
template<typename ValueType = default_precision, typename IndexType = int32>
void gko::preconditioner::Jacobi< ValueType, IndexType >::move_to (
    matrix::Dense< value_type > * result ) [override], [virtual]
```

Converts the implementer to an object of type `result_type` by moving data from this object.

This method is used when the implementer is a temporary object, and move semantics can be used.

Parameters

<i>result</i>	the object used to emplace the result of the conversion
---------------	---

Note

[ConvertibleTo::move_to](#) can be implemented by simply calling [ConvertibleTo::convert_to](#). However, this operation can often be optimized by exploiting the fact that implementer's data can be moved to the result.

Implements [gko::ConvertibleTo< matrix::Dense< ValueType > >](#).

24.52.2.8 write()

```
template<typename ValueType = default_precision, typename IndexType = int32>
void gko::preconditioner::Jacobi< ValueType, IndexType >::write (
    mat_data & data ) const [override], [virtual]
```

Writes a matrix to a [matrix_data](#) structure.

Parameters

<i>data</i>	the matrix_data structure
-------------	---

Implements [gko::WritableToMatrixData< ValueType, IndexType >](#).

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

- [ginkgo/core/preconditioner/jacobi.hpp](#) (8045ac75)

24.53 gko::KernelNotFound Class Reference

[KernelNotFound](#) is thrown if Ginkgo cannot find a kernel which satisfies the criteria imposed by the input arguments.

```
#include <ginkgo/core/base/exception.hpp>
```

Public Member Functions

- [KernelNotFound](#) (const std::string &file, int line, const std::string &func)
Initializes a [KernelNotFound](#) error.

24.53.1 Detailed Description

[KernelNotFound](#) is thrown if Ginkgo cannot find a kernel which satisfies the criteria imposed by the input arguments.

24.53.2 Constructor & Destructor Documentation

24.53.2.1 KernelNotFound()

```
gko::KernelNotFound::KernelNotFound (
    const std::string & file,
    int line,
    const std::string & func ) [inline]
```

Initializes a [KernelNotFound](#) error.

Parameters

<i>file</i>	The name of the offending source file
<i>line</i>	The source code line number where the error occurred
<i>func</i>	The name of the function where the error occurred

```
348         : Error(file, line, func + ": unable to find an eligible kernel")
349     {}
```

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

- ginkgo/core/base/exception.hpp (8045ac75)

24.54 gko::log::linop_data Struct Reference

Struct representing LinOp related data.

```
#include <ginkgo/core/log/record.hpp>
```

24.54.1 Detailed Description

Struct representing LinOp related data.

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

- ginkgo/core/log/record.hpp (f0a50f96)

24.55 gko::log::linop_factory_data Struct Reference

Struct representing LinOp factory related data.

```
#include <ginkgo/core/log/record.hpp>
```

24.55.1 Detailed Description

Struct representing LinOp factory related data.

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

- `ginkgo/core/log/record.hpp` (f0a50f96)

24.56 gko::LinOpFactory Class Reference

A [LinOpFactory](#) represents a higher order mapping which transforms one linear operator into another.

```
#include <ginkgo/core/base/lin_op.hpp>
```

Additional Inherited Members

24.56.1 Detailed Description

A [LinOpFactory](#) represents a higher order mapping which transforms one linear operator into another.

In Ginkgo, every linear solver is viewed as a mapping. For example, given an s.p.d linear system $Ax = b$, the solution $x = A^{-1}b$ can be computed using the CG method. This algorithm can be represented in terms of linear operators and mappings between them as follows:

- A `Cg::Factory` is a higher order mapping which, given an input operator A , returns a new linear operator A^{-1} stored in "CG format"
- Storing the operator A^{-1} in "CG format" means that the data structure used to store the operator is just a simple pointer to the original matrix A . The application $x = A^{-1}b$ of such an operator can then be implemented by solving the linear system $Ax = b$ using the CG method. This is achieved in code by having a special class for each of those "formats" (e.g. the "Cg" class defines such a format for the CG solver).

Another example of a [LinOpFactory](#) is a preconditioner. A preconditioner for a linear operator A is a linear operator M^{-1} , which approximates A^{-1} . In addition, it is stored in a way such that both the data of M^{-1} is cheap to compute from A , and the operation $x = M^{-1}b$ can be computed quickly. These operators are useful to accelerate the convergence of Krylov solvers. Thus, a preconditioner also fits into the [LinOpFactory](#) framework:

- The factory maps a linear operator A into a preconditioner M^{-1} which is stored in suitable format (e.g. as a product of two factors in case of ILU preconditioners).
- The resulting linear operator implements the application operation $x = M^{-1}b$ depending on the format the preconditioner is stored in (e.g. as two triangular solves in case of ILU)

Example: using CG in Ginkgo

```
{c++}
// Suppose A is a matrix, b a rhs vector, and x an initial guess
// Create a CG which runs for at most 1000 iterations, and stops after
// reducing the residual norm by 6 orders of magnitude
auto cg_factory = solver::Cg<>::build()
    .with_max_iters(1000)
    .with_rel_residual_goal(1e-6)
    .on(cuda);
// create a linear operator which represents the solver
auto cg = cg_factory->generate(A);
// solve the system
cg->apply(gko::lend(b), gko::lend(x));
```

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

- ginkgo/core/base/lin_op.hpp (fb72cdf1)

24.57 gko::log::Loggable Class Reference

[Loggable](#) class is an interface which should be implemented by classes wanting to support logging.

```
#include <ginkgo/core/log/logger.hpp>
```

Public Member Functions

- virtual void [add_logger](#) (std::shared_ptr< const Logger > logger)=0
Adds a new logger to the list of subscribed loggers.
- virtual void [remove_logger](#) (const Logger *logger)=0
Removes a logger from the list of subscribed loggers.

24.57.1 Detailed Description

[Loggable](#) class is an interface which should be implemented by classes wanting to support logging.

For most cases, one can rely on the [EnableLogging](#) mixin which provides a default implementation of this interface.

24.57.2 Member Function Documentation

24.57.2.1 add_logger()

```
virtual void gko::log::Loggable::add_logger (
    std::shared_ptr< const Logger > logger ) [pure virtual]
```

Adds a new logger to the list of subscribed loggers.

Parameters

<i>logger</i>	the logger to add
---------------	-------------------

Implemented in [gko::log::EnableLogging< ConcreteLoggable, PolymorphicBase >](#), [gko::log::EnableLogging< PolymorphicObject >](#), and [gko::log::EnableLogging< Executor >](#).

24.57.2.2 remove_logger()

```
virtual void gko::log::Loggable::remove_logger (
    const Logger * logger ) [pure virtual]
```

Removes a logger from the list of subscribed loggers.

Parameters

<i>logger</i>	the logger to remove
---------------	----------------------

Note

The comparison is done using the logger's object unique identity. Thus, two loggers constructed in the same way are not considered equal.

Implemented in [gko::log::EnableLogging< ConcreteLoggable, PolymorphicBase >](#), [gko::log::EnableLogging< PolymorphicObject >](#), and [gko::log::EnableLogging< Executor >](#).

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

- [ginkgo/core/log/logger.hpp \(0d7578c9\)](#)

24.58 gko::log::Record::logged_data Struct Reference

Struct storing the actually logged data.

```
#include <ginkgo/core/log/record.hpp>
```

24.58.1 Detailed Description

Struct storing the actually logged data.

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

- [ginkgo/core/log/record.hpp \(f0a50f96\)](#)

24.59 gko::matrix_data< ValueType, IndexType > Struct Template Reference

This structure is used as an intermediate data type to store a sparse matrix.

```
#include <ginkgo/core/base/matrix_data.hpp>
```

Classes

- struct [nonzero_type](#)
Type used to store nonzeros.

Public Member Functions

- [matrix_data](#) (dim< 2 > size__{=dim< 2 >{}}, ValueType value=[zero](#)< ValueType >())
Initializes a matrix filled with the specified value.
- template<typename RandomDistribution, typename RandomEngine >
[matrix_data](#) (dim< 2 > size__, RandomDistribution &&dist, RandomEngine &&engine)
Initializes a matrix with random values from the specified distribution.
- [matrix_data](#) (std::initializer_list< std::initializer_list< ValueType >> values)
List-initializes the structure from a matrix of values.
- [matrix_data](#) (dim< 2 > size__, std::initializer_list< detail::input_triple< ValueType, IndexType >> nonzeros__↵
_)
Initializes the structure from a list of nonzeros.
- [matrix_data](#) (dim< 2 > size__, const [matrix_data](#) &block)
Initializes a matrix out of a matrix block via duplication.
- template<typename Accessor >
[matrix_data](#) (const [range](#)< Accessor > &data)
Initializes a matrix from a range.
- void [ensure_row_major_order](#) ()
Sorts the nonzero vector so the values follow row-major order.

Static Public Member Functions

- static [matrix_data diag](#) (dim< 2 > size__, ValueType value)
Initializes a diagonal matrix.
- static [matrix_data diag](#) (dim< 2 > size__, std::initializer_list< ValueType > nonzeros__↵
_)
Initializes a diagonal matrix using a list of diagonal elements.
- static [matrix_data diag](#) (dim< 2 > size__, const [matrix_data](#) &block)
Initializes a block-diagonal matrix.
- template<typename ForwardIterator >
static [matrix_data diag](#) (ForwardIterator begin, ForwardIterator end)
Initializes a block-diagonal matrix from a list of diagonal blocks.
- static [matrix_data diag](#) (std::initializer_list< [matrix_data](#) > blocks)
Initializes a block-diagonal matrix from a list of diagonal blocks.
- template<typename RandomDistribution, typename RandomEngine >
static [matrix_data cond](#) (size_type size, [remove_complex](#)< ValueType > condition_number, Random↵
Distribution &&dist, RandomEngine &&engine, size_type num_reflectors)
Initializes a random dense matrix with a specific condition number.
- template<typename RandomDistribution, typename RandomEngine >
static [matrix_data cond](#) (size_type size, [remove_complex](#)< ValueType > condition_number, Random↵
Distribution &&dist, RandomEngine &&engine)
Initializes a random dense matrix with a specific condition number.

Public Attributes

- `dim< 2 > size`
Size of the matrix.
- `std::vector< nonzero_type > nonzeros`
A vector of tuples storing the non-zeros of the matrix.

24.59.1 Detailed Description

```
template<typename ValueType = default_precision, typename IndexType = int32>
struct gko::matrix_data< ValueType, IndexType >
```

This structure is used as an intermediate data type to store a sparse matrix.

The matrix is stored as a sequence of nonzero elements, where each element is a triple of the form (row_index, column_index, value).

Note

All Ginkgo functions returning such a structure will return the nonzeros sorted in row-major order.
 All Ginkgo functions that take this structure as input expect that the nonzeros are sorted in row-major order.
 This structure is not optimized for usual access patterns and it can only exist on the CPU. Thus, it should only be used for utility functions which do not have to be optimized for performance.

Template Parameters

<i>ValueType</i>	type of matrix values stored in the structure
<i>IndexType</i>	type of matrix indexes stored in the structure

24.59.2 Constructor & Destructor Documentation

24.59.2.1 `matrix_data()` [1/6]

```
template<typename ValueType = default_precision, typename IndexType = int32>
gko::matrix_data< ValueType, IndexType >::matrix_data (
    dim< 2 > size_ = dim<2>{},
    ValueType value = zero<ValueType>() ) [inline]
```

Initializes a matrix filled with the specified value.

Parameters

<i>size_</i>	dimensions of the matrix
<i>value</i>	value used to fill the elements of the matrix

```

143                                     {}, ValueType value = zero<ValueType>())
144     : size{size_}
145     {
146         if (value == zero<ValueType>()) {
147             return;
148         }
149         for (size_type row = 0; row < size[0]; ++row) {
150             for (size_type col = 0; col < size[1]; ++col) {
151                 nonzeros.emplace_back(row, col, value);
152             }
153         }
154     }

```

24.59.2.2 matrix_data() [2/6]

```

template<typename ValueType = default_precision, typename IndexType = int32>
template<typename RandomDistribution, typename RandomEngine >
gko::matrix_data< ValueType, IndexType >::matrix_data (
    dim< 2 > size_,
    RandomDistribution && dist,
    RandomEngine && engine ) [inline]

```

Initializes a matrix with random values from the specified distribution.

Template Parameters

<i>RandomDistribution</i>	random distribution type
<i>RandomEngine</i>	random engine type

Parameters

<i>size</i> ↔ —	dimensions of the matrix
<i>dist</i>	random distribution of the elements of the matrix
<i>engine</i>	random engine used to generate random values

```

168     : size{size_}
169     {
170         for (size_type row = 0; row < size[0]; ++row) {
171             for (size_type col = 0; col < size[1]; ++col) {
172                 const auto value =
173                     detail::get_rand_value<ValueType>(dist, engine);
174                 if (value != zero<ValueType>()) {
175                     nonzeros.emplace_back(row, col, value);
176                 }
177             }
178         }
179     }

```

24.59.2.3 matrix_data() [3/6]

```

template<typename ValueType = default_precision, typename IndexType = int32>
gko::matrix_data< ValueType, IndexType >::matrix_data (
    std::initializer_list< std::initializer_list< ValueType >> values ) [inline]

```

List-initializes the structure from a matrix of values.

Parameters

<i>values</i>	a 2D braced-init-list of matrix values.
---------------	---

```

187         : size{values.size(), 0}
188     {
189         for (size_type row = 0; row < values.size(); ++row) {
190             const auto row_data = begin(values)[row];
191             size[1] = std::max(size[1], row_data.size());
192             for (size_type col = 0; col < row_data.size(); ++col) {
193                 const auto &val = begin(row_data)[col];
194                 if (val != zero<ValueType>()) {
195                     nonzeros.emplace_back(row, col, val);
196                 }
197             }
198         }
199     }

```

24.59.2.4 `matrix_data()` [4/6]

```

template<typename ValueType = default_precision, typename IndexType = int32>
gko::matrix_data< ValueType, IndexType >::matrix_data (
    dim< 2 > size_,
    std::initializer_list< detail::input_triple< ValueType, IndexType >> nonzeros_ )
[inline]

```

Initializes the structure from a list of nonzeros.

Parameters

<i>size_</i>	dimensions of the matrix
<i>nonzeros_</i>	list of nonzero elements
—	

```

211         : size{size_, nonzeros()}
212     {
213         nonzeros.reserve(nonzeros_.size());
214         for (const auto &elem : nonzeros_) {
215             nonzeros.emplace_back(elem.row, elem.col, elem.val);
216         }
217     }

```

24.59.2.5 `matrix_data()` [5/6]

```

template<typename ValueType = default_precision, typename IndexType = int32>
gko::matrix_data< ValueType, IndexType >::matrix_data (
    dim< 2 > size_,
    const matrix_data< ValueType, IndexType > & block ) [inline]

```

Initializes a matrix out of a matrix block via duplication.

Parameters

<i>size</i>	size of the block-matrix (in blocks)
<i>diag_block</i>	matrix block used to fill the complete matrix

References gko::matrix_data< ValueType, IndexType >::size.

```

226         : size{size_ * block.size}
227     {
228         nonzeros.reserve(size_[0] * size_[1] * block.nonzeros.size());
229         for (size_type row = 0; row < size_[0]; ++row) {
230             for (size_type col = 0; col < size_[1]; ++col) {
231                 for (const auto &elem : block.nonzeros) {
232                     nonzeros.emplace_back(row * block.size[0] + elem.row,
233                                           col * block.size[1] + elem.column,
234                                           elem.value);
235                 }
236             }
237         }
238         this->ensure_row_major_order();
239     }

```

24.59.2.6 matrix_data() [6/6]

```

template<typename ValueType = default_precision, typename IndexType = int32>
template<typename Accessor >
gko::matrix_data< ValueType, IndexType >::matrix_data (
    const range< Accessor > & data ) [inline]

```

Initializes a matrix from a range.

Template Parameters

<i>Accessor</i>	accessor type of the input range
-----------------	----------------------------------

Parameters

<i>data</i>	range used to initialize the matrix
-------------	-------------------------------------

References gko::range< Accessor >::length().

```

250         : size{data.length(0), data.length(1)}
251     {
252         for (gko::size_type row = 0; row < size[0]; ++row) {
253             for (gko::size_type col = 0; col < size[1]; ++col) {
254                 if (data(row, col) != zero<ValueType>()) {
255                     nonzeros.emplace_back(row, col, data(row, col));
256                 }
257             }
258         }
259     }

```

24.59.3 Member Function Documentation

24.59.3.1 cond() [1/2]

```
template<typename ValueType = default_precision, typename IndexType = int32>
template<typename RandomDistribution , typename RandomEngine >
static matrix_data gko::matrix_data< ValueType, IndexType >::cond (
    size_type size,
    remove_complex< ValueType > condition_number,
    RandomDistribution && dist,
    RandomEngine && engine,
    size_type num_reflectors ) [inline], [static]
```

Initializes a random dense matrix with a specific condition number.

The matrix is generated by applying a series of random Hausholder reflectors to a diagonal matrix with diagonal entries uniformly distributed between `sqrt(condition_number)` and `1/sqrt(condition_number)`.

Template Parameters

<i>RandomDistribution</i>	the type of the random distribution
<i>RandomEngine</i>	the type of the random engine

Parameters

<i>size</i>	number of rows and columns of the matrix
<i>condition_number</i>	condition number of the matrix
<i>dist</i>	random distribution used to generate reflectors
<i>engine</i>	random engine used to generate reflectors
<i>num_reflectors</i>	number of reflectors to apply from each side

Returns

the dense matrix with the specified condition number

```
396 {
397     using range = range<accessor::row_major<ValueType, 2>>;
398     std::vector<ValueType> mtx_data(size * size, zero<ValueType>());
399     std::vector<ValueType> ref_data(size);
400     std::vector<ValueType> work(size);
401     range matrix(mtx_data.data(), size, size, size);
402     range reflector(ref_data.data(), size, lu, lu);
403
404     initialize_diag_with_cond(condition_number, matrix);
405     for (size_type i = 0; i < num_reflectors; ++i) {
406         generate_random_reflector(dist, engine, reflector);
407         reflect_domain(reflector, matrix, work.data());
408         generate_random_reflector(dist, engine, reflector);
409         reflect_range(reflector, matrix, work.data());
410     }
411     return matrix;
412 }
```

24.59.3.2 cond() [2/2]

```
template<typename ValueType = default_precision, typename IndexType = int32>
template<typename RandomDistribution , typename RandomEngine >
```

```
static matrix_data gko::matrix_data< ValueType, IndexType >::cond (
    size_type size,
    remove_complex< ValueType > condition_number,
    RandomDistribution && dist,
    RandomEngine && engine ) [inline], [static]
```

Initializes a random dense matrix with a specific condition number.

The matrix is generated by applying a series of random Hausholder reflectors to a diagonal matrix with diagonal entries uniformly distributed between `sqrt(condition_number)` and `1/sqrt(condition_number)`.

This version of the function applies `size - 1` reflectors to each side of the diagonal matrix.

Template Parameters

<i>RandomDistribution</i>	the type of the random distribution
<i>RandomEngine</i>	the type of the random engine

Parameters

<i>size</i>	number of rows and columns of the matrix
<i>condition_number</i>	condition number of the matrix
<i>dist</i>	random distribution used to generate reflectors
<i>engine</i>	random engine used to generate reflectors

Returns

the dense matrix with the specified condition number

```
439 {
440     return cond(size, condition_number,
441                std::forward<RandomDistribution>(dist),
442                std::forward<RandomEngine>(engine), size - 1);
443 }
```

24.59.3.3 diag() [1/5]

```
template<typename ValueType = default_precision, typename IndexType = int32>
static matrix_data gko::matrix_data< ValueType, IndexType >::diag (
    dim< 2 > size_,
    ValueType value ) [inline], [static]
```

Initializes a diagonal matrix.

Parameters

<i>size</i> ↔	dimensions of the matrix
<i>value</i>	value used to fill the elements of the matrix

Returns

the diagonal matrix

References `gko::matrix_data< ValueType, IndexType >::nonzeros`.

```

270     {
271         matrix_data res(size_);
272         if (value != zero<ValueType>()) {
273             const auto num_nnz = std::min(size_[0], size_[1]);
274             res.nonzeros.reserve(num_nnz);
275             for (size_type i = 0; i < num_nnz; ++i) {
276                 res.nonzeros.emplace_back(i, i, value);
277             }
278         }
279         return res;
280     }

```

24.59.3.4 diag() [2/5]

```

template<typename ValueType = default_precision, typename IndexType = int32>
static matrix_data gko::matrix_data< ValueType, IndexType >::diag (
    dim< 2 > size_,
    std::initializer_list< ValueType > nonzeros_ ) [inline], [static]

```

Initializes a diagonal matrix using a list of diagonal elements.

Parameters

<i>size_</i>	dimensions of the matrix
<i>nonzeros_↔</i>	list of diagonal elements
—	

Returns

the diagonal matrix

References `gko::matrix_data< ValueType, IndexType >::nonzeros`.

```

292     {
293         matrix_data res(size_);
294         res.nonzeros.reserve(nonzeros_.size());
295         int pos = 0;
296         for (auto value : nonzeros_) {
297             res.nonzeros.emplace_back(pos, pos, value);
298             ++pos;
299         }
300         return res;
301     }

```


24.59.3.5 diag() [3/5]

```
template<typename ValueType = default_precision, typename IndexType = int32>
static matrix_data gko::matrix_data< ValueType, IndexType >::diag (
    dim< 2 > size_,
    const matrix_data< ValueType, IndexType > & block ) [inline], [static]
```

Initializes a block-diagonal matrix.

Parameters

<i>size_</i>	the size of the matrix
<i>diag_block</i>	matrix used to fill diagonal blocks

Returns

the block-diagonal matrix

References `gko::matrix_data< ValueType, IndexType >::nonzeros`, and `gko::matrix_data< ValueType, IndexType >::size`.

```

312     {
313         matrix_data res(size_ * block.size);
314         const auto num_blocks = std::min(size_[0], size_[1]);
315         res.nonzeros.reserve(num_blocks * block.nonzeros.size());
316         for (size_type b = 0; b < num_blocks; ++b) {
317             for (const auto &elem : block.nonzeros) {
318                 res.nonzeros.emplace_back(b * block.size[0] + elem.row,
319                                           b * block.size[1] + elem.column,
320                                           elem.value);
321             }
322         }
323         return res;
324     }

```

24.59.3.6 `diag()` [4/5]

```

template<typename ValueType = default_precision, typename IndexType = int32>
template<typename ForwardIterator >
static matrix_data gko::matrix_data< ValueType, IndexType >::diag (
    ForwardIterator begin,
    ForwardIterator end ) [inline], [static]

```

Initializes a block-diagonal matrix from a list of diagonal blocks.

Template Parameters

<i>ForwardIterator</i>	type of list iterator
------------------------	-----------------------

Parameters

<i>begin</i>	the first iterator of the list
<i>end</i>	the last iterator of the list

Returns

the block-diagonal matrix with diagonal blocks set to the blocks between `begin` (inclusive) and `end` (exclusive)

References `gko::matrix_data< ValueType, IndexType >::nonzeros`.

```

339     {
340         matrix_data res(std::accumulate(
341             begin, end, dim<2>{}, [](dim<2> s, const matrix_data &d) {
342                 return dim<2>{s[0] + d.size[0], s[1] + d.size[1]};
343             }));
344
345         size_type row_offset{};
346         size_type col_offset{};
347         for (auto it = begin; it != end; ++it) {
348             for (const auto &elem : it->nonzeros) {
349                 res.nonzeros.emplace_back(row_offset + elem.row,
350                                         col_offset + elem.column, elem.value);
351             }
352             row_offset += it->size[0];
353             col_offset += it->size[1];
354         }
355
356         return res;
357     }

```

24.59.3.7 diag() [5/5]

```

template<typename ValueType = default_precision, typename IndexType = int32>
static matrix_data gko::matrix_data< ValueType, IndexType >::diag (
    std::initializer_list< matrix_data< ValueType, IndexType > > blocks ) [inline],
[static]

```

Initializes a block-diagonal matrix from a list of diagonal blocks.

Parameters

<i>blocks</i>	a list of blocks to initialize from
---------------	-------------------------------------

Returns

the block-diagonal matrix with diagonal blocks set to the blocks passed in blocks

```

368     {
369         return diag(begin(blocks), end(blocks));
370     }

```

24.59.4 Member Data Documentation

24.59.4.1 nonzeros

```

template<typename ValueType = default_precision, typename IndexType = int32>
std::vector<nonzero_type> gko::matrix_data< ValueType, IndexType >::nonzeros

```

A vector of tuples storing the non-zeros of the matrix.

The first two elements of the tuple are the row index and the column index of a matrix element, and its third element is the value at that position.

Referenced by gko::matrix_data< ValueType, IndexType >::diag().

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

- ginkgo/core/base/matrix_data.hpp (4bde4271)

24.60 gko::matrix::Hybrid< ValueType, IndexType >::minimal_storage_limit Class Reference

[minimal_storage_limit](#) is a `stratgy_type` which decides the number of stored elements per row of the ell part.

```
#include <ginkgo/core/matrix/hybrid.hpp>
```

Public Member Functions

- [minimal_storage_limit](#) ()
Creates a [minimal_storage_limit](#) strategy.
- [size_type compute_ell_num_stored_elements_per_row](#) (Array< [size_type](#) > *row_nnz) const override
Computes the number of stored elements per row of the ell part.

24.60.1 Detailed Description

```
template<typename ValueType = default_precision, typename IndexType = int32>
class gko::matrix::Hybrid< ValueType, IndexType >::minimal_storage_limit
```

[minimal_storage_limit](#) is a `stratgy_type` which decides the number of stored elements per row of the ell part.

It is determined by the size of `ValueType` and `IndexType`, the storage is the minimum among all partition.

24.60.2 Member Function Documentation

24.60.2.1 compute_ell_num_stored_elements_per_row()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size\_type gko::matrix::Hybrid< ValueType, IndexType >::minimal_storage_limit::compute_ell_num_stored_elements_per_row (
    Array< size\_type > * row_nnz ) const [inline], [override], [virtual]
```

Computes the number of stored elements per row of the ell part.

Parameters

<code>row_nnz</code>	the number of nonzeros of each row
----------------------	------------------------------------

Returns

the number of stored elements per row of the ell part

Implements [gko::matrix::Hybrid< ValueType, IndexType >::strategy_type](#).

```

303         {
304             return strategy_.compute_ell_num_stored_elements_per_row
305             (row_nnz);
306         }

```

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

- ginkgo/core/matrix/hybrid.hpp (3e51a52b)

24.61 gko::matrix_data< ValueType, IndexType >::nonzero_type Struct Reference

Type used to store nonzeros.

```
#include <ginkgo/core/base/matrix_data.hpp>
```

24.61.1 Detailed Description

```
template<typename ValueType = default_precision, typename IndexType = int32>
struct gko::matrix_data< ValueType, IndexType >::nonzero_type
```

Type used to store nonzeros.

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

- ginkgo/core/base/matrix_data.hpp (4bde4271)

24.62 gko::NotCompiled Class Reference

[NotCompiled](#) is thrown when attempting to call an operation which is a part of a module that was not compiled on the system.

```
#include <ginkgo/core/base/exception.hpp>
```

Public Member Functions

- [NotCompiled](#) (const std::string &file, int line, const std::string &func, const std::string &module)
Initializes a [NotCompiled](#) error.

24.62.1 Detailed Description

[NotCompiled](#) is thrown when attempting to call an operation which is a part of a module that was not compiled on the system.

24.62.2 Constructor & Destructor Documentation

24.62.2.1 NotCompiled()

```
gko::NotCompiled::NotCompiled (
    const std::string & file,
    int line,
    const std::string & func,
    const std::string & module ) [inline]
```

Initializes a [NotCompiled](#) error.

Parameters

<i>file</i>	The name of the offending source file
<i>line</i>	The source code line number where the error occurred
<i>func</i>	The name of the function that has not been compiled
<i>module</i>	The name of the module which contains the function

```

142         : Error(file, line,
143                 "feature " + func + " is part of the " + module +
144                 " module, which is not compiled on this system")
145     {}

```

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

- ginkgo/core/base/exception.hpp (8045ac75)

24.63 gko::NotImplemented Class Reference

NotImplemented is thrown in case an operation has not yet been implemented (but will be implemented in the future).

```
#include <ginkgo/core/base/exception.hpp>
```

Public Member Functions

- **NotImplemented** (const std::string &file, int line, const std::string &func)
*Initializes a **NotImplemented** error.*

24.63.1 Detailed Description

NotImplemented is thrown in case an operation has not yet been implemented (but will be implemented in the future).

24.63.2 Constructor & Destructor Documentation

24.63.2.1 NotImplemented()

```

gko::NotImplemented::NotImplemented (
    const std::string & file,
    int line,
    const std::string & func ) [inline]

```

Initializes a **NotImplemented** error.

Parameters

<i>file</i>	The name of the offending source file
<i>line</i>	The source code line number where the error occurred
<i>func</i>	The name of the not-yet implemented function

```

122         : Error(file, line, func + " is not implemented")
123     {}

```

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

- ginkgo/core/base/exception.hpp (8045ac75)

24.64 gko::NotSupported Class Reference

[NotSupported](#) is thrown in case it is not possible to perform the requested operation on the given object type.

```
#include <ginkgo/core/base/exception.hpp>
```

Public Member Functions

- [NotSupported](#) (const std::string &file, int line, const std::string &func, const std::string &obj_type)
Initializes a [NotSupported](#) error.

24.64.1 Detailed Description

[NotSupported](#) is thrown in case it is not possible to perform the requested operation on the given object type.

24.64.2 Constructor & Destructor Documentation

24.64.2.1 NotSupported()

```

gko::NotSupported::NotSupported (
    const std::string & file,
    int line,
    const std::string & func,
    const std::string & obj_type ) [inline]

```

Initializes a [NotSupported](#) error.

Parameters

<i>file</i>	The name of the offending source file
<i>line</i>	The source code line number where the error occurred
<i>func</i>	The name of the function where the error occurred
<i>obj_type</i>	The object type on which the requested operation cannot be performed.

```

165         : Error(file, line,
166                 "Operation " + func + " does not support parameters of type " +
167                 obj_type)
168     {}

```

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

- ginkgo/core/base/exception.hpp (8045ac75)

24.65 gko::null_deleter< T > Class Template Reference

This is a deleter that does not delete the object.

```
#include <ginkgo/core/base/utils.hpp>
```

Public Member Functions

- void [operator\(\)](#) (pointer) const noexcept
Deletes the object.

24.65.1 Detailed Description

```
template<typename T>
class gko::null_deleter< T >
```

This is a deleter that does not delete the object.

It is useful where the object has been allocated elsewhere and will be deleted manually.

24.65.2 Member Function Documentation

24.65.2.1 operator()

```
template<typename T >
void gko::null_deleter< T >::operator() (
    pointer ) const [inline], [noexcept]
```

Deletes the object.

Parameters

<i>ptr</i>	pointer to the object being deleted
------------	-------------------------------------

```
334 {}
```

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

- ginkgo/core/base/utils.hpp (4bde4271)

24.66 gko::OmpExecutor Class Reference

This is the [Executor](#) subclass which represents the OpenMP device (typically CPU).

```
#include <ginkgo/core/base/executor.hpp>
```

Public Member Functions

- `std::shared_ptr< Executor > get_master ()` noexcept override
Returns the master [OmpExecutor](#) of this [Executor](#).
- `std::shared_ptr< const Executor > get_master ()` const noexcept override
Returns the master [OmpExecutor](#) of this [Executor](#).
- `void synchronize ()` const override
Synchronize the operations launched on the executor with its master.

Static Public Member Functions

- `static std::shared_ptr< OmpExecutor > create ()`
Creates a new [OmpExecutor](#).

24.66.1 Detailed Description

This is the [Executor](#) subclass which represents the OpenMP device (typically CPU).

24.66.2 Member Function Documentation

24.66.2.1 `get_master()` [1/2]

```
std::shared_ptr<Executor> gko::OmpExecutor::get_master ( ) [override], [virtual], [noexcept]
```

Returns the master [OmpExecutor](#) of this [Executor](#).

Returns

the master [OmpExecutor](#) of this [Executor](#).

Implements [gko::Executor](#).

24.66.2.2 `get_master()` [2/2]

```
std::shared_ptr<const Executor> gko::OmpExecutor::get_master ( ) const [override], [virtual], [noexcept]
```

Returns the master [OmpExecutor](#) of this [Executor](#).

Returns

the master [OmpExecutor](#) of this [Executor](#).

Implements [gko::Executor](#).

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

- `ginkgo/core/base/executor.hpp` (f1a4eb68)

24.67 `gko::Operation` Class Reference

Operations can be used to define functionalities whose implementations differ among devices.

```
#include <ginkgo/core/base/executor.hpp>
```

Public Member Functions

- virtual const char * [get_name](#) () const noexcept
Returns the operation's name.

24.67.1 Detailed Description

Operations can be used to define functionalities whose implementations differ among devices.

This is done by extending the [Operation](#) class and implementing the overloads of the `Operation::run()` method for all [Executor](#) types. When invoking the `Executor::run()` method with the [Operation](#) as input, the library will select the `Operation::run()` overload corresponding to the dynamic type of the [Executor](#) instance.

Consider an overload of `operator<<` for Executors, which prints some basic device information (e.g. device type and id) of the [Executor](#) to a C++ stream:

```
std::ostream& operator<<(std::ostream &os, const gko::Executor &exec);
```

One possible implementation would be to use RTTI to find the dynamic type of the [Executor](#). However, using the [Operation](#) feature of Ginkgo, there is a more elegant approach which utilizes polymorphism. The first step is to define an [Operation](#) that will print the desired information for each [Executor](#) type.

```
class DeviceInfoPrinter : public gko::Operation {
public:
    explicit DeviceInfoPrinter(std::ostream &os) : os_(os) {}

    void run(const gko::OmpExecutor *) const override { os_ << "OMP"; }

    void run(const gko::CudaExecutor *exec) const override
    { os_ << "CUDA(" << exec->get_device_id() << ")"; }

    // This is optional, if not overloaded, defaults to OmpExecutor overload
    void run(const gko::ReferenceExecutor *) const override
    { os_ << "Reference CPU"; }

private:
    std::ostream &os_;
};
```

Using `DeviceInfoPrinter`, the implementation of `operator<<` is as simple as calling the `run()` method of the executor.

```
std::ostream& operator<<(std::ostream &os, const gko::Executor &exec)
{
    DeviceInfoPrinter printer(os);
    exec.run(printer);
    return os;
}
```

Now it is possible to write the following code:

```
auto omp = gko::OmpExecutor::create();
std::cout << *omp << std::endl
          << *gko::CudaExecutor::create(0, omp) << std::endl
          << *gko::ReferenceExecutor::create() << std::endl;
```

which produces the expected output:

```
OMP
CUDA (0)
Reference CPU
```

One might feel that this code is too complicated for such a simple task. Luckily, there is an overload of the [Executor::run\(\) method, which is designed to facilitate writing simple operations like this one. The method takes two closures as input: one which is run for OMP, and the other one for CUDA executors. Using this method, there is no need to implement an \[Operation\]\(#\) subclass:](#)

```
std::ostream& operator<<(std::ostream &os, const gko::Executor &exec)
{
    exec.run(
        [&]() { os << "OMP"; }, // OMP closure
        [&]() { os << "CUDA(" // CUDA closure
            << static_cast<gko::CudaExecutor&>(exec)
                .get_device_id()
            << ")"; });
    return os;
}
```

Using this approach, however, it is impossible to distinguish between a [OmpExecutor](#) and [ReferenceExecutor](#), as both of them call the OMP closure.

24.67.2 Member Function Documentation

24.67.2.1 `get_name()`

```
virtual const char* gko::Operation::get_name ( ) const [virtual], [noexcept]
```

Returns the operation's name.

Returns

the operation's name

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

- `ginkgo/core/base/executor.hpp` (f1a4eb68)

24.68 `gko::log::operation_data` Struct Reference

Struct representing Operator related data.

```
#include <ginkgo/core/log/record.hpp>
```

24.68.1 Detailed Description

Struct representing Operator related data.

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

- `ginkgo/core/log/record.hpp` (f0a50f96)

24.69 `gko::OutOfBoundsError` Class Reference

[OutOfBoundsError](#) is thrown if a memory access is detected to be out-of-bounds.

```
#include <ginkgo/core/base/exception.hpp>
```

Public Member Functions

- [OutOfBoundsError](#) (const std::string &file, int line, [size_type](#) index, [size_type](#) bound)
Initializes an [OutOfBoundsError](#).

24.69.1 Detailed Description

[OutOfBoundsError](#) is thrown if a memory access is detected to be out-of-bounds.

24.69.2 Constructor & Destructor Documentation

24.69.2.1 OutOfBoundsError()

```
gko::OutOfBoundsError::OutOfBoundsError (
    const std::string & file,
    int line,
    size_type index,
    size_type bound ) [inline]
```

Initializes an [OutOfBoundsError](#).

Parameters

<i>file</i>	The name of the offending source file
<i>line</i>	The source code line number where the error occurred
<i>index</i>	The position that was accessed
<i>bound</i>	The first out-of-bound index

```
308         : Error(file, line,
309               "trying to access index " + std::to_string(index) +
310               " in a memory block of " + std::to_string(bound) +
311               " elements")
312     {}
```

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

- ginkgo/core/base/exception.hpp (8045ac75)

24.70 gko::factorization::Parllu< ValueType, IndexType > Class Template Reference

ParILU is an incomplete LU factorization which is computed in parallel.

```
#include <ginkgo/core/factorization/par_ilu.hpp>
```

Additional Inherited Members

24.70.1 Detailed Description

```
template<typename ValueType = default_precision, typename IndexType = int32>
class gko::factorization::Parllu< ValueType, IndexType >
```

ParLLU is an incomplete LU factorization which is computed in parallel.

L is a lower unitriangular, while U is an upper triangular matrix, which approximate a given matrix A with $A \approx LU$. Here, L and U have the same sparsity pattern as A , which is also called ILU(0).

The ParLLU algorithm generates the incomplete factors iteratively, using a fixed-point iteration of the form

$$F(L, U) = \begin{cases} \frac{1}{u_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right), & i > j \\ a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}, & i \leq j \end{cases}$$

In general, the entries of L and U can be iterated in parallel and in asynchronous fashion, the algorithm asymptotically converges to the incomplete factors L and U fulfilling $(R = A - L \cdot U)|_S = 0|_S$ where S is the pre-defined sparsity pattern (in case of ILU(0) the sparsity pattern of the system matrix A). The number of ParLLU sweeps needed for convergence depends on the parallelism level: For sequential execution, a single sweep is sufficient, for fine-grained parallelism, 3 sweeps are typically generating a good approximation.

The ParLLU algorithm in Ginkgo follows the design of E. Chow and A. Patel, Fine-grained Parallel Incomplete LU Factorization, SIAM Journal on Scientific Computing, 37, C169-C193 (2015).

Template Parameters

<i>ValueType</i>	Type of the values of all matrices used in this class
<i>IndexType</i>	Type of the indices of all matrices used in this class

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

- ginkgo/core/factorization/par_ilu.hpp (e9e876bc)

24.71 gko::log::polymorphic_object_data Struct Reference

Struct representing [PolymorphicObject](#) related data.

```
#include <ginkgo/core/log/record.hpp>
```

24.71.1 Detailed Description

Struct representing [PolymorphicObject](#) related data.

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

- ginkgo/core/log/record.hpp (f0a50f96)

24.72 gko::PolymorphicObject Class Reference

A [PolymorphicObject](#) is the abstract base for all "heavy" objects in Ginkgo that behave polymorphically.

```
#include <ginkgo/core/base/polymorphic_object.hpp>
```

Public Member Functions

- `std::unique_ptr< PolymorphicObject > create_default (std::shared_ptr< const Executor > exec) const`
Creates a new "default" object of the same dynamic type as this object.
- `std::unique_ptr< PolymorphicObject > create_default () const`
Creates a new "default" object of the same dynamic type as this object.
- `std::unique_ptr< PolymorphicObject > clone (std::shared_ptr< const Executor > exec) const`
Creates a clone of the object.
- `std::unique_ptr< PolymorphicObject > clone () const`
Creates a clone of the object.
- `PolymorphicObject * copy_from (const PolymorphicObject *other)`
Copies another object into this object.
- `PolymorphicObject * copy_from (std::unique_ptr< PolymorphicObject > other)`
Moves another object into this object.
- `PolymorphicObject * clear ()`
Transforms the object into its default state.
- `std::shared_ptr< const Executor > get_executor () const noexcept`
Returns the [Executor](#) of the object.

24.72.1 Detailed Description

A [PolymorphicObject](#) is the abstract base for all "heavy" objects in Ginkgo that behave polymorphically.

It defines the basic utilities (copying moving, cloning, clearing the objects) for all such objects. It takes into account that these objects are dynamically allocated, managed by smart pointers, and used polymorphically. Additionally, it assumes their data can be allocated on different executors, and that they can be copied between those executors.

Note

Most of the public methods of this class should not be overridden directly, and are thus not virtual. Instead, there are equivalent protected methods (ending in `<method_name>_impl`) that should be overridden instead. This allows polymorphic objects to implement default behavior around virtual methods (parameter checking, type casting).

See also

[EnablePolymorphicObject](#) if you wish to implement a concrete polymorphic object and have sensible defaults generated automatically. [EnableAbstractPolymorphicObject](#) if you wish to implement a new abstract polymorphic object, and have the return types of the methods updated to your type (instead of having them return [PolymorphicObject](#)).

24.72.2 Member Function Documentation

24.72.2.1 clear()

```
PolymorphicObject* gko::PolymorphicObject::clear ( ) [inline]
```

Transforms the object into its default state.

Equivalent to `this->copy_from(this->create_default())`.

See also

`clear_impl()` when implementing this method

Returns

this

```
194 { return this->clear_impl(); }
```

24.72.2.2 clone() [1/2]

```
std::unique_ptr<PolymorphicObject> gko::PolymorphicObject::clone (
    std::shared_ptr< const Executor > exec ) const [inline]
```

Creates a clone of the object.

This is the polymorphic equivalent of the *executor copy constructor* `decltype(*this) (exec, this)`.

Parameters

<i>exec</i>	the executor where the clone will be created
-------------	--

Returns

A clone of the LinOp.

References `create_default()`.

```
124 {
125     auto new_op = this->create_default(exec);
126     new_op->copy_from(this);
127     return new_op;
128 }
```


24.72.2.3 clone() [2/2]

```
std::unique_ptr<PolymorphicObject> gko::PolymorphicObject::clone ( ) const [inline]
```

Creates a clone of the object.

This is a shorthand for `clone(std::shared_ptr<const Executor>)` that uses the executor of this object to construct the new object.

Returns

A clone of the LinOp.

```
139     {
140         return this->clone(exec_);
141     }
```

24.72.2.4 copy_from() [1/2]

```
PolymorphicObject* gko::PolymorphicObject::copy_from (
    const PolymorphicObject * other ) [inline]
```

Copies another object into this object.

This is the polymorphic equivalent of the copy assignment operator.

See also

`copy_from_impl(const PolymorphicObject *)`

Parameters

<i>other</i>	the object to copy
--------------	--------------------

Returns

this

```
155     {
156         this->template log<log::Logger::polymorphic_object_copy_started>(
157             exec_.get(), other, this);
158         auto copied = this->copy_from_impl(other);
159         this->template log<log::Logger::polymorphic_object_copy_completed>(
160             exec_.get(), other, this);
161         return copied;
162     }
```

24.72.2.5 `copy_from()` [2/2]

```
PolymorphicObject* gko::PolymorphicObject::copy_from (
    std::unique_ptr< PolymorphicObject > other ) [inline]
```

Moves another object into this object.

This is the polymorphic equivalent of the move assignment operator.

See also

`copy_from_impl(std::unique_ptr<PolymorphicObject>)`

Parameters

<i>other</i>	the object to move from
--------------	-------------------------

Returns

this

```
176     {
177         this->template log<log::Logger::polymorphic_object_copy_started>(
178             exec_.get(), other.get(), this);
179         auto copied = this->copy_from_impl(std::move(other));
180         this->template log<log::Logger::polymorphic_object_copy_completed>(
181             exec_.get(), other.get(), this);
182         return copied;
183     }
```

24.72.2.6 `create_default()` [1/2]

```
std::unique_ptr<PolymorphicObject> gko::PolymorphicObject::create_default (
    std::shared_ptr< const Executor > exec ) const [inline]
```

Creates a new "default" object of the same dynamic type as this object.

This is the polymorphic equivalent of the *executor default constructor* `decltype(*this) (exec) ;`.

Parameters

<i>exec</i>	the executor where the object will be created
-------------	---

Returns

a polymorphic object of the same type as this

```
90     {
91         this->template log<log::Logger::polymorphic_object_create_started>(
92             exec_.get(), this);
93         auto created = this->create_default_impl(std::move(exec));
```

```

94         this->template log<log::Logger::polymorphic_object_create_completed>(
95             exec_.get(), this, created.get());
96         return created;
97     }

```

24.72.2.7 create_default() [2/2]

```
std::unique_ptr<PolymorphicObject> gko::PolymorphicObject::create_default ( ) const [inline]
```

Creates a new "default" object of the same dynamic type as this object.

This is a shorthand for `create_default(std::shared_ptr<const Executor>)` that uses the executor of this object to construct the new object.

Returns

a polymorphic object of the same type as this

Referenced by `clone()`.

```

108     {
109         return this->create_default(exec_);
110     }

```

24.72.2.8 get_executor()

```
std::shared_ptr<const Executor> gko::PolymorphicObject::get_executor ( ) const [inline],
[noexcept]
```

Returns the [Executor](#) of the object.

Returns

[Executor](#) of the object

Referenced by `gko::matrix::Dense< ValueType >::add_scaled()`, `gko::matrix::Coo< ValueType, IndexType >::apply2()`, `gko::matrix::Dense< ValueType >::compute_dot()`, `gko::matrix::Dense< ValueType >::compute_norm2()`, `gko::matrix::Dense< ValueType >::create_submatrix()`, `gko::solver::Gmres< ValueType >::get_krylov_dim()`, `gko::solver::Cgs< ValueType >::get_preconditioner()`, `gko::solver::Cg< ValueType >::get_preconditioner()`, `gko::solver::Fcg< ValueType >::get_preconditioner()`, `gko::solver::Bicgstab< ValueType >::get_preconditioner()`, `gko::solver::Irr< ValueType >::get_solver()`, and `gko::matrix::Dense< ValueType >::scale()`.

```

202     {
203         return exec_;
204     }

```

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

- `ginkgo/core/base/polymorphic_object.hpp` (4bde4271)

24.73 gko::precision_reduction Class Reference

This class is used to encode storage precisions of low precision algorithms.

```
#include <ginkgo/core/base/types.hpp>
```

Public Types

- using [storage_type](#) = [uint8](#)
The underlying datatype used to store the encoding.

Public Member Functions

- constexpr [precision_reduction](#) () noexcept
Creates a default [precision_reduction](#) encoding.
- constexpr [precision_reduction](#) ([storage_type](#) preserving, [storage_type](#) nonpreserving) noexcept
Creates a [precision_reduction](#) encoding with the specified number of conversions.
- constexpr [operator storage_type](#) () const noexcept
Extracts the raw data of the encoding.
- constexpr [storage_type](#) [get_preserving](#) () const noexcept
Returns the number of preserving conversions in the encoding.
- constexpr [storage_type](#) [get_nonpreserving](#) () const noexcept
Returns the number of non-preserving conversions in the encoding.

Static Public Member Functions

- static constexpr [precision_reduction autodetect](#) () noexcept
Returns a special encoding which instructs the algorithm to automatically detect the best precision.
- static constexpr [precision_reduction common](#) ([precision_reduction](#) x, [precision_reduction](#) y) noexcept
Returns the common encoding of input encodings.

24.73.1 Detailed Description

This class is used to encode storage precisions of low precision algorithms.

Some algorithms in Ginkgo can improve their performance by storing parts of the data in lower precision, while doing computation in full precision. This class is used to encode the precisions used to store the data. From the user's perspective, some algorithms can provide a parameter for fine-tuning the storage precision. Commonly, the special value returned by [precision_reduction::autodetect\(\)](#) should be used to allow the algorithm to automatically choose an appropriate value, though manually selected values can be used for fine-tuning.

In general, a lower precision floating point value can be obtained by either dropping some of the insignificant bits of the significand (keeping the same number of exponent bits, and thus preserving the range of representable values) or using one of the hardware or software supported conversions between IEEE formats, such as double to float or float to half (reducing both the number of exponent, as well as significand bits, and thus decreasing the range of representable values).

The [precision_reduction](#) class encodes the lower precision format relative to the base precision used and the algorithm in question. The encoding is done by specifying the amount of range non-preserving conversions and

the amount of range preserving conversions that should be done on the base precision to obtain the lower precision format. For example, starting with a double precision value (11 exp, 52 sig. bits), the encoding specifying 1 non-preserving conversion and 1 preserving conversion would first use a hardware-supported non-preserving conversion to obtain a single precision value (8 exp, 23 sig. bits), followed by a preserving bit truncation to obtain a value with 8 exponent and 7 significand bits. Note that non-preserving conversion are always done first, as preserving conversions usually result in datatypes that are not supported by builtin conversions (thus, it is generally not possible to apply a non-preserving conversion to the result of a preserving conversion).

If the specified conversion is not supported by the algorithm, it will most likely fall back to using full precision for storing the data. Refer to the documentation of specific algorithms using this class for details about such special cases.

24.73.2 Constructor & Destructor Documentation

24.73.2.1 precision_reduction() [1/2]

```
constexpr gko::precision_reduction::precision_reduction ( ) [inline], [noexcept]
```

Creates a default [precision_reduction](#) encoding.

This encoding represents the case where no conversions are performed.

Referenced by `common()`.

```
250 : data_{0x0} {}
```

24.73.2.2 precision_reduction() [2/2]

```
constexpr gko::precision_reduction::precision_reduction (
    storage_type preserving,
    storage_type nonpreserving ) [inline], [noexcept]
```

Creates a [precision_reduction](#) encoding with the specified number of conversions.

Parameters

<i>preserving</i>	the number of range preserving conversion
<i>nonpreserving</i>	the number of range non-preserving conversions

```
261 : data_((GKO_ASSERT(preserving < (0x1 << preserving_bits) - 1),
262         GKO_ASSERT(nonpreserving < (0x1 << nonpreserving_bits) - 1),
263         (preserving << nonpreserving_bits) | nonpreserving))
264 {}
```

24.73.3 Member Function Documentation

24.73.3.1 autodetect()

```
static constexpr precision_reduction gko::precision_reduction::autodetect ( ) [inline], [static],
[noexcept]
```

Returns a special encoding which instructs the algorithm to automatically detect the best precision.

Returns

a special encoding instructing the algorithm to automatically detect the best precision.

```
304     {
305         return precision_reduction{preserving_mask | nonpreserving_mask};
306     }
```

24.73.3.2 common()

```
static constexpr precision_reduction gko::precision_reduction::common (
    precision_reduction x,
    precision_reduction y ) [inline], [static], [noexcept]
```

Returns the common encoding of input encodings.

The common encoding is defined as the encoding that does not have more preserving, nor non-preserving conversions than the input encodings.

Parameters

<i>x</i>	an encoding
<i>y</i>	an encoding

Returns

the common encoding of *x* and *y*

References `precision_reduction()`.

```
321     {
322         return precision_reduction(
323             min(x.data_ & preserving_mask, y.data_ & preserving_mask) |
324             min(x.data_ & nonpreserving_mask, y.data_ & nonpreserving_mask));
325     }
```

24.73.3.3 get_nonpreserving()

```
constexpr storage_type gko::precision_reduction::get_nonpreserving ( ) const [inline], [noexcept]
```

Returns the number of non-preserving conversions in the encoding.

Returns

the number of non-preserving conversions in the encoding.

```
292     {  
293         return data_ & nonpreserving_mask;  
294     }
```

24.73.3.4 get_preserving()

```
constexpr storage_type gko::precision_reduction::get_preserving ( ) const [inline], [noexcept]
```

Returns the number of preserving conversions in the encoding.

Returns

the number of preserving conversions in the encoding.

```
282     {  
283         return (data_ & preserving_mask) >> nonpreserving_bits;  
284     }
```

24.73.3.5 operator storage_type()

```
constexpr gko::precision_reduction::operator storage_type ( ) const [inline], [noexcept]
```

Extracts the raw data of the encoding.

Returns

the raw data of the encoding

```
272     {  
273         return data_;  
274     }
```

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

- ginkgo/core/base/types.hpp (4bde4271)

24.74 gko::Preconditionable Class Reference

A LinOp implementing this interface can be preconditioned.

```
#include <ginkgo/core/base/lin_op.hpp>
```

Public Member Functions

- virtual std::shared_ptr< const LinOp > [get_preconditioner](#) () const =0
Returns the preconditioner operator used by the [Preconditionable](#).

24.74.1 Detailed Description

A LinOp implementing this interface can be preconditioned.

24.74.2 Member Function Documentation

24.74.2.1 get_preconditioner()

```
virtual std::shared_ptr<const LinOp> gko::Preconditionable::get_preconditioner ( ) const
[pure virtual]
```

Returns the preconditioner operator used by the [Preconditionable](#).

Returns

the preconditioner operator used by the [Preconditionable](#)

Implemented in [gko::solver::Bicgstab< ValueType >](#), [gko::solver::Fcg< ValueType >](#), [gko::solver::Cg< ValueType >](#), [gko::solver::Gmres< ValueType >](#), and [gko::solver::Cgs< ValueType >](#).

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

- ginkgo/core/base/lin_op.hpp (fb72cdf1)

24.75 gko::range< Accessor > Class Template Reference

A range is a multidimensional view of the memory.

```
#include <ginkgo/core/base/range.hpp>
```


Public Types

- using `accessor` = Accessor
The type of the underlying accessor.

Public Member Functions

- template<typename... AccessorParams>
constexpr `range` (AccessorParams &&... params)
Creates a new range.
- template<typename... DimensionTypes>
constexpr auto `operator()` (DimensionTypes &&... dimensions) const -> decltype(std::declval< `accessor` >())(std::forward< DimensionTypes >(dimensions)...))
Returns a value (or a sub-range) with the specified indexes.
- template<typename OtherAccessor >
const `range` & `operator=` (const `range`< OtherAccessor > &other) const
- const `range` & `operator=` (const `range` &other) const
Assigns another range to this range.
- constexpr `size_type length` (`size_type` dimension) const
Returns the length of the specified dimension of the range.
- constexpr const `accessor` * `operator->` () const noexcept
Returns a pointer to the accessor.
- constexpr const `accessor` & `get_accessor` () const noexcept
Returns a reference to the accessor.

Static Public Attributes

- static constexpr `size_type dimensionality` = accessor::dimensionality
The number of dimensions of the range.

24.75.1 Detailed Description

```
template<typename Accessor>
class gko::range< Accessor >
```

A range is a multidimensional view of the memory.

The range does not store any of its values by itself. Instead, it obtains the values through an accessor (e.g. `accessor::row_major`) which describes how the indexes of the range map to physical locations in memory.

There are several advantages of using ranges instead of plain memory pointers:

1. Code using ranges is easier to read and write, as there is no need for index linearizations.
2. Code using ranges is safer, as it is impossible to accidentally miscalculate an index or step out of bounds, since range accessors perform bounds checking in debug builds. For performance, this can be disabled in release builds by defining the `NDEBUG` flag.
3. Ranges enable generalized code, as algorithms can be written independent of the memory layout. This does not impede various optimizations based on memory layout, as it is always possible to specialize algorithms for ranges with specific memory layouts.
4. Ranges have various pointwise operations predefined, which reduces the amount of loops that need to be written.

Range operations

Ranges define a complete set of pointwise unary and binary operators which extend the basic arithmetic operators in C++, as well as a few pointwise operations and mathematical functions useful in ginkgo, and a couple of non-pointwise operations. Compound assignment (`+=`, `*=`, etc.) is not yet supported at this moment. Here is a complete list of operations:

- standard unary operations: `+`, `-`, `!`, `~`
- standard binary operations: `+`, `*` (this is pointwise, not matrix multiplication), `/`, `%`, `<`, `>`, `<=`, `>=`, `==`, `!=`, `||`, `&&`, `|`, `&`, `^`, `<<`, `>>`
- useful unary functions: `zero`, `one`, `abs`, `real`, `imag`, `conj`, `squared_norm`
- useful binary functions: `min`, `max`

All binary pointwise operations also work as expected if one of the operands is a scalar and the other is a range. The scalar operand will have the effect as if it was a range of the same size as the other operand, filled with the specified scalar.

Two "global" functions `transpose` and `mmul` are also supported. `transpose` transposes the first two dimensions of the range (i.e. `transpose(r)(i, j, ...) == r(j, i, ...)`). `mmul` performs a (batched) matrix multiply of the ranges - the first two dimensions represent the matrices, while the rest represent the batch. For example, given the ranges `r1` and `r2` of dimensions `(3, 2, 3)` and `(2, 4, 3)`, respectively, `mmul(r1, r2)` will return a range of dimensions `(3, 4, 3)`, obtained by multiplying the 3 frontal slices of the range, and stacking the result back vertically.

Compound operations

Multiple range operations can be combined into a single expression. For example, an "axpy" operation can be obtained using `y = alpha * x + y`, where `x` and `y` are ranges, and `alpha` is a scalar. Range operations are optimized for memory access, and the above code does not allocate additional storage for intermediate ranges `alpha * x` or `alpha * x + y`. In fact, the entire computation is done during the assignment, and the results of operations `+` and `*` only register the data, and the types of operations that will be computed once the results are needed.

It is possible to store and reuse these intermediate expressions. The following example will overwrite the range `x` with its 4th power:

```
{c++}
auto square = x * x; // this is range constructor, not range assignment!
x = square; // overwrites x with x*x (this is range assignment)
x = square; // overwrites new x (x*x) with (x*x)*(x*x) (as is this)
```

Caveats

`__mmul` is not a highly-optimized BLAS-3 version of the matrix multiplication. The current design of ranges and accessors prevents that, so if you need a high-performance matrix multiplication, you should use one of the libraries that provide that, or implement your own (you can use pointwise range operations to help simplify that). However, range design might get improved in the future to allow efficient implementations of BLAS-3 kernels.

Aliasing the result range in `mmul` and `transpose` is not allowed. Constructs like `A = transpose(A)`, `A = mmul(A, A)`, or `A = mmul(A, A) + C` lead to undefined behavior. However, aliasing input arguments is allowed: `C = mmul(A, A)`, and even `C = mmul(A, A) + C` is valid code (in the last example, only pointwise operations are aliased). `C = mmul(A, A + C)` is not valid though.

Examples

The range unit tests in `core/test/base/range.cpp` contain lots of simple 1-line examples of range operations. The accessor unit tests in `core/test/base/range.cpp` show how to use ranges with concrete accessors, and how to use range slices using `spans` as arguments to range function call operator. Finally, `examples/range` contains a complete example where ranges are used to implement a simple version of the right-looking LU factorization.

Template Parameters

<i>Accessor</i>	underlying accessor of the range
-----------------	----------------------------------

24.75.2 Constructor & Destructor Documentation

24.75.2.1 range()

```
template<typename Accessor>
template<typename... AccessorParams>
constexpr gko::range< Accessor >::range (
    AccessorParams &&... params ) [inline], [explicit]
```

Creates a new range.

Template Parameters

<i>AccessorParam</i>	types of parameters forwarded to the accessor constructor
----------------------	---

Parameters

<i>params</i>	parameters forwarded to Accessor constructor.
---------------	---

```
318         : accessor_{std::forward<AccessorParams>(params)...}
319     {}
```

24.75.3 Member Function Documentation

24.75.3.1 get_accessor()

```
template<typename Accessor>
constexpr const accessor& gko::range< Accessor >::get_accessor ( ) const [inline], [noexcept]
```

‘Returns a reference to the accessor.

Returns

reference to the accessor

Referenced by gko::range< Accessor >::operator=().

```
410     {
411         return accessor_;
412     }
```

24.75.3.2 length()

```
template<typename Accessor>
constexpr size_type gko::range< Accessor >::length (
    size_type dimension ) const [inline]
```

Returns the length of the specified dimension of the range.

Parameters

<i>dimension</i>	the dimensions whose length is returned
------------------	---

Returns

the length of the *dimension*-th dimension of the range

Referenced by `gko::matrix_data< ValueType, IndexType >::ensure_row_major_order()`, and `gko::matrix_data< ValueType, IndexType >::matrix_data()`.

```
388     {
389         return accessor_.length(dimension);
390     }
```

24.75.3.3 operator>()

```
template<typename Accessor>
template<typename... DimensionTypes>
constexpr auto gko::range< Accessor >::operator() (
    DimensionTypes &&... dimensions ) const -> decltype(std::declval<accessor>()) (
    std::forward<DimensionTypes>(dimensions)...) [inline]
```

Returns a value (or a sub-range) with the specified indexes.

Template Parameters

<i>DimensionTypes</i>	The types of indexes. Supported types depend on the underlying accessor, but are usually either integer types or spans. If at least one index is a span, the returned value will be a sub-range.
-----------------------	--

Parameters

<i>dimensions</i>	the indexes of the values.
-------------------	----------------------------

Returns

a value on position `(dimensions...)`.

```
337     {
```

```

338         static_assert(sizeof...(dimensions) <= dimensionality,
339                        "Too many dimensions in range call");
340         return accessor_(std::forward<DimensionTypes>(dimensions)...);
341     }

```

24.75.3.4 operator->()

```

template<typename Accessor>
constexpr const accessor* gko::range< Accessor >::operator-> ( ) const [inline], [noexcept]

```

Returns a pointer to the accessor.

Can be used to access data and functions of a specific accessor.

Returns

pointer to the accessor

```

400     {
401         return &accessor_;
402     }

```

24.75.3.5 operator=() [1/2]

```

template<typename Accessor>
template<typename OtherAccessor >
const range& gko::range< Accessor >::operator= (
    const range< OtherAccessor > & other ) const [inline]

```

This is a version of the function which allows to copy between ranges of different accessors.

Template Parameters

<i>OtherAccessor</i>	accessor of the other range
----------------------	-----------------------------

```

354     {
355         GKO_ASSERT(detail::equal_dimensions(*this, other));
356         accessor_.copy_from(other);
357         return *this;
358     }

```

24.75.3.6 operator=() [2/2]

```

template<typename Accessor>
const range& gko::range< Accessor >::operator= (
    const range< Accessor > & other ) const [inline]

```

Assigns another range to this range.

The order of assignment is defined by the accessor of this range, thus the memory access will be optimized for the resulting range, and not for the other range. If the sizes of two ranges do not match, the result is undefined. Sizes of the ranges are checked at runtime in debug builds.

Note

Temporary accessors are allowed to define the implementation of the assignment as deleted, so do not expect `r1 * r2 = r2` to work.

Parameters

<i>other</i>	the range to copy the data from
--------------	---------------------------------

References `gko::range< Accessor >::get_accessor()`.

```

374     {
375         GKO_ASSERT(detail::equal_dimensions(*this, other));
376         accessor_.copy_from(other.get_accessor());
377         return *this;
378     }

```

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

- `ginkgo/core/base/range.hpp` (f1a4eb68)

24.76 gko::ReadableFromMatrixData< ValueType, IndexType > Class Template Reference

A LinOp implementing this interface can read its data from a [matrix_data](#) structure.

```
#include <ginkgo/core/base/lin_op.hpp>
```

Public Member Functions

- virtual void [read](#) (const [matrix_data](#)< ValueType, IndexType > &data)=0
Reads a matrix from a [matrix_data](#) structure.

24.76.1 Detailed Description

```

template<typename ValueType, typename IndexType>
class gko::ReadableFromMatrixData< ValueType, IndexType >

```

A LinOp implementing this interface can read its data from a [matrix_data](#) structure.

24.76.2 Member Function Documentation

24.76.2.1 read()

```
template<typename ValueType, typename IndexType>
virtual void gko::ReadableFromMatrixData< ValueType, IndexType >::read (
    const matrix_data< ValueType, IndexType > & data ) [pure virtual]
```

Reads a matrix from a [matrix_data](#) structure.

Parameters

<i>data</i>	the matrix_data structure
-------------	---

Implemented in [gko::matrix::Hybrid< ValueType, IndexType >](#), [gko::matrix::Csr< ValueType, IndexType >](#), [gko::matrix::Coo< ValueType, IndexType >](#), [gko::matrix::Ell< ValueType, IndexType >](#), and [gko::matrix::Sellp< ValueType, IndexType >](#).

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

- [ginkgo/core/base/lin_op.hpp](#) (fb72cdf1)

24.77 gko::log::Record Class Reference

[Record](#) is a Logger which logs every event to an object.

```
#include <ginkgo/core/log/record.hpp>
```

Classes

- struct [logged_data](#)
Struct storing the actually logged data.

Public Member Functions

- const [logged_data](#) & [get](#) () const noexcept
Returns the logged data.
- [logged_data](#) & [get](#) () noexcept

Static Public Member Functions

- static std::unique_ptr< [Record](#) > [create](#) (std::shared_ptr< const [Executor](#) > exec, const mask_type &enabled_events=Logger::all_events_mask, [size_type](#) max_storage=1)
Creates a [Record](#) logger.

24.77.1 Detailed Description

[Record](#) is a Logger which logs every event to an object.

The object can then be accessed at any time by asking the logger to return it.

Note

Please note that this logger can have significant memory and performance overhead. In particular, when logging events such as the `check` events, all parameters are cloned. If it is sufficient to clone one parameter, consider implementing a specific logger for this. In addition, it is advised to tune the history size in order to control memory overhead.

24.77.2 Member Function Documentation

24.77.2.1 create()

```
static std::unique_ptr<Record> gko::log::Record::create (
    std::shared_ptr< const Executor > exec,
    const mask_type & enabled_events = Logger::all_events_mask,
    size_type max_storage = 1 ) [inline], [static]
```

Creates a [Record](#) logger.

This dynamically allocates the memory, constructs the object and returns an `std::unique_ptr` to this object.

Parameters

<i>exec</i>	the executor
<i>enabled_events</i>	the events enabled for this logger. By default all events.
<i>max_storage</i>	the size of storage (i.e. history) wanted by the user. By default 0 is used, which means unlimited storage. It is advised to control this to reduce memory overhead of this logger.

Returns

an `std::unique_ptr` to the the constructed object

```
397 {
398     return std::unique_ptr<Record>(
399         new Record(exec, enabled_events, max_storage));
400 }
```

24.77.2.2 get() [1/2]

```
const logged_data& gko::log::Record::get ( ) const [inline], [noexcept]
```

Returns the logged data.

Returns

the logged data

```
407 { return data_; }
```

24.77.2.3 `get()` [2/2]

```
logged_data& gko::log::Record::get ( ) [inline], [noexcept]
```

```
412 { return data_; }
```

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

- ginkgo/core/log/record.hpp (f0a50f96)

24.78 gko::ReferenceExecutor Class Reference

This is a specialization of the [OmpExecutor](#), which runs the reference implementations of the kernels used for debugging purposes.

```
#include <ginkgo/core/base/executor.hpp>
```

Public Member Functions

- void [run](#) (const [Operation](#) &op) const override
Runs the specified [Operation](#) using this [Executor](#).

Additional Inherited Members

24.78.1 Detailed Description

This is a specialization of the [OmpExecutor](#), which runs the reference implementations of the kernels used for debugging purposes.

24.78.2 Member Function Documentation

24.78.2.1 `run()`

```
void gko::ReferenceExecutor::run (
    const Operation & op ) const [inline], [override], [virtual]
```

Runs the specified [Operation](#) using this [Executor](#).

Parameters

<i>op</i>	the operation to run
-----------	----------------------

Implements [gko::Executor](#).

```

779     {
780         this->template log<log::Logger::operation_launched>(this, &op);
781         op.run(std::static_pointer_cast<const ReferenceExecutor>(
782             this->shared_from_this()));
783         this->template log<log::Logger::operation_completed>(this, &op);
784     }

```

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

- ginkgo/core/base/executor.hpp (f1a4eb68)

24.79 gko::stop::ResidualNormReduction< ValueType > Class Template Reference

The [ResidualNormReduction](#) class is a stopping criterion which stops the iteration process when the relative residual norm is below a certain threshold.

```
#include <ginkgo/core/stop/residual_norm_reduction.hpp>
```

24.79.1 Detailed Description

```
template<typename ValueType = default_precision>
class gko::stop::ResidualNormReduction< ValueType >
```

The [ResidualNormReduction](#) class is a stopping criterion which stops the iteration process when the relative residual norm is below a certain threshold.

For better performance, the checks are run thanks to kernels on the executor where the algorithm is executed.

Note

To use this stopping criterion there are some dependencies. The constructor depends on `initial_residual` in order to compute the first relative residual norm. The check method depends on either the `residual_norm` or the `residual` being set. When any of those is not correctly provided, an exception `::gko::NotSupported()` is thrown.

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

- ginkgo/core/stop/residual_norm_reduction.hpp (8045ac75)

24.80 gko::accessor::row_major< ValueType, Dimensionality > Class Template Reference

A [row_major](#) accessor is a bridge between a range and the row-major memory layout.

```
#include <ginkgo/core/base/range_accessors.hpp>
```

Public Types

- using `value_type` = `ValueType`
Type of values returned by the accessor.
- using `data_type` = `value_type` *
Type of underlying data storage.

Public Member Functions

- constexpr `value_type` & `operator()` (`size_type` row, `size_type` col) const
Returns the data element at position (row, col)
- constexpr `range`< `row_major` > `operator()` (const `span` &rows, const `span` &cols) const
Returns the sub-range spanning the range (rows, cols)
- constexpr `size_type` `length` (`size_type` dimension) const
Returns the length in dimension `dimension`.
- template<typename OtherAccessor >
void `copy_from` (const OtherAccessor &other) const
Copies data from another accessor.

Public Attributes

- const `data_type` `data`
Reference to the underlying data.
- const `std::array`< const `size_type`, `dimensionality` > `lengths`
An array of dimension sizes.
- const `size_type` `stride`
Distance between consecutive rows.

Static Public Attributes

- static constexpr `size_type` `dimensionality` = 2
Number of dimensions of the accessor.

24.80.1 Detailed Description

```
template<typename ValueType, size_type Dimensionality>
class gko::accessor::row_major< ValueType, Dimensionality >
```

A `row_major` accessor is a bridge between a range and the row-major memory layout.

You should never try to explicitly create an instance of this accessor. Instead, supply it as a template parameter to a range, and pass the constructor parameters for this class to the range (it will forward it to this class).

Warning

The current implementation is incomplete, and only allows for 2-dimensional ranges.

Template Parameters

<i>ValueType</i>	type of values this accessor returns
<i>Dimensionality</i>	number of dimensions of this accessor (has to be 2)

24.80.2 Member Function Documentation

24.80.2.1 `copy_from()`

```
template<typename ValueType , size_type Dimensionality>
template<typename OtherAccessor >
void gko::accessor::row_major< ValueType, Dimensionality >::copy_from (
    const OtherAccessor & other ) const [inline]
```

Copies data from another accessor.

Template Parameters

<i>OtherAccessor</i>	type of the other accessor
----------------------	----------------------------

Parameters

<i>other</i>	other accessor
--------------	----------------

References `gko::accessor::row_major< ValueType, Dimensionality >::lengths`.

```
164     {
165         for (size_type i = 0; i < lengths[0]; ++i) {
166             for (size_type j = 0; j < lengths[1]; ++j) {
167                 (*this)(i, j) = other(i, j);
168             }
169         }
170     }
```

24.80.2.2 `length()`

```
template<typename ValueType , size_type Dimensionality>
constexpr size_type gko::accessor::row_major< ValueType, Dimensionality >::length (
    size_type dimension ) const [inline]
```

Returns the length in dimension dimension.

Parameters

<i>dimension</i>	a dimension index
------------------	-------------------

Returns

length in dimension `dimension`

References `gko::accessor::row_major< ValueType, Dimensionality >::lengths`.

```

151     {
152         return dimension < 2 ? lengths[dimension] : 1;
153     }

```

24.80.2.3 operator() [1/2]

```

template<typename ValueType , size_type Dimensionality>
constexpr value_type& gko::accessor::row_major< ValueType, Dimensionality >::operator() (
    size_type row,
    size_type col ) const [inline]

```

Returns the data element at position (row, col)

Parameters

<i>row</i>	row index
<i>col</i>	column index

Returns

data element at (row, col)

References `gko::accessor::row_major< ValueType, Dimensionality >::data`, `gko::accessor::row_major< ValueType, Dimensionality >::lengths`, and `gko::accessor::row_major< ValueType, Dimensionality >::stride`.

```

119     {
120         return GKO_ASSERT(row < lengths[0]), GKO_ASSERT(col < lengths[1]),
121             data[row * stride + col];
122     }

```

24.80.2.4 operator() [2/2]

```

template<typename ValueType , size_type Dimensionality>
constexpr range<row_major> gko::accessor::row_major< ValueType, Dimensionality >::operator()
(
    const span & rows,
    const span & cols ) const [inline]

```

Returns the sub-range spanning the range (rows, cols)

Parameters

<i>rows</i>	row span
<i>cols</i>	column span

Returns

sub-range spanning the range (rows, cols)

References `gko::span::begin`, `gko::accessor::row_major< ValueType, Dimensionality >::data`, `gko::span::end`, `gko::span::is_valid()`, `gko::accessor::row_major< ValueType, Dimensionality >::lengths`, and `gko::accessor::row↵_major< ValueType, Dimensionality >::stride`.

```

134     {
135         return GKO_ASSERT(rows.is_valid()), GKO_ASSERT(cols.is_valid()),
136                GKO_ASSERT(rows <= span{lengths[0]}),
137                GKO_ASSERT(cols <= span{lengths[1]}),
138                range<row_major>(data + rows.begin * stride + cols.begin,
139                                rows.end - rows.begin, cols.end - cols.begin,
140                                stride);
141     }

```

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

- `ginkgo/core/base/range_accessors.hpp` (f1a4eb68)

24.81 `gko::matrix::Sellp< ValueType, IndexType >` Class Template Reference

SELL-P is a matrix format similar to ELL format.

```
#include <ginkgo/core/matrix/sellp.hpp>
```

Public Member Functions

- void `read` (const `mat_data` &data) override
Reads a matrix from a `matrix_data` structure.
- void `write` (`mat_data` &data) const override
Writes a matrix to a `matrix_data` structure.
- value_type * `get_values` () noexcept
Returns the values of the matrix.
- const value_type * `get_const_values` () const noexcept
Returns the values of the matrix.
- index_type * `get_col_idx`s () noexcept
Returns the column indexes of the matrix.
- const index_type * `get_const_col_idx`s () const noexcept
Returns the column indexes of the matrix.
- size_type * `get_slice_lengths` () noexcept
Returns the lengths(columns) of slices.
- const size_type * `get_const_slice_lengths` () const noexcept
Returns the lengths(columns) of slices.
- size_type * `get_slice_sets` () noexcept

- Returns the offsets of slices.*
- `const size_type * get_const_slice_sets ()` const noexcept
Returns the offsets of slices.
- `size_type get_slice_size ()` const noexcept
Returns the size of a slice.
- `size_type get_stride_factor ()` const noexcept
Returns the stride factor(t) of SELL-P.
- `size_type get_total_cols ()` const noexcept
Returns the total column number.
- `size_type get_num_stored_elements ()` const noexcept
Returns the number of elements explicitly stored in the matrix.
- `value_type & val_at (size_type row, size_type slice_set, size_type idx)` noexcept
Returns the idx-th non-zero element of the row-th row with slice_set slice set.
- `value_type val_at (size_type row, size_type slice_set, size_type idx)` const noexcept
Returns the idx-th non-zero element of the row-th row with slice_set slice set.
- `index_type & col_at (size_type row, size_type slice_set, size_type idx)` noexcept
Returns the idx-th column index of the row-th row with slice_set slice set.
- `index_type col_at (size_type row, size_type slice_set, size_type idx)` const noexcept
Returns the idx-th column index of the row-th row with slice_set slice set.

24.81.1 Detailed Description

```
template<typename ValueType = default_precision, typename IndexType = int32>
class gko::matrix::Sellp< ValueType, IndexType >
```

SELL-P is a matrix format similar to ELL format.

The difference is that SELL-P format divides rows into smaller slices and store each slice with ELL format.

Template Parameters

<i>ValueType</i>	precision of matrix elements
<i>IndexType</i>	precision of matrix indexes

24.81.2 Member Function Documentation

24.81.2.1 col_at() [1/2]

```
template<typename ValueType = default_precision, typename IndexType = int32>
index_type& gko::matrix::Sellp< ValueType, IndexType >::col_at (
    size_type row,
    size_type slice_set,
    size_type idx ) [inline], [noexcept]
```

Returns the idx-th column index of the row-th row with slice_set slice set.

Parameters

<i>row</i>	the row of the requested element in the slice
<i>slice_set</i>	the slice set of the slice
<i>idx</i>	the idx-th stored element of the row in the slice

Note

the method has to be called on the same [Executor](#) the matrix is stored at (e.g. trying to call this method on a GPU matrix from the CPU results in a runtime error)

References `gko::matrix::Sellp< ValueType, IndexType >::get_col_idxxs()`.

```

251     {
252         return this->get_col_idxxs() [this->linearize_index(row, slice_set, idx)];
253     }

```

24.81.2.2 `col_at()` [2/2]

```

template<typename ValueType = default_precision, typename IndexType = int32>
index_type gko::matrix::Sellp< ValueType, IndexType >::col_at (
    size_type row,
    size_type slice_set,
    size_type idx ) const [inline], [noexcept]

```

Returns the `idx`-th column index of the `row`-th row with `slice_set` slice set.

Parameters

<i>row</i>	the row of the requested element in the slice
<i>slice_set</i>	the slice set of the slice
<i>idx</i>	the idx-th stored element of the row in the slice

Note

the method has to be called on the same [Executor](#) the matrix is stored at (e.g. trying to call this method on a GPU matrix from the CPU results in a runtime error)

References `gko::ceildiv()`.

```

260     {
261         return this
262             ->get_const_col_idxxs() [this->linearize_index(row, slice_set, idx)];
263     }

```

24.81.2.3 get_col_idxes()

```
template<typename ValueType = default_precision, typename IndexType = int32>
index_type* gko::matrix::Sellp< ValueType, IndexType >::get_col_idxes ( ) [inline], [noexcept]
```

Returns the column indexes of the matrix.

Returns

the column indexes of the matrix.

References gko::Array< ValueType >::get_data().

Referenced by gko::matrix::Sellp< ValueType, IndexType >::col_at().

```
123 { return col_idxes_.get_data(); }
```

24.81.2.4 get_const_col_idxes()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const index_type* gko::matrix::Sellp< ValueType, IndexType >::get_const_col_idxes ( ) const
[inline], [noexcept]
```

Returns the column indexes of the matrix.

Returns

the column indexes of the matrix.

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

References gko::Array< ValueType >::get_const_data().

```
133 {
134     return col_idxes_.get_const_data();
135 }
```

24.81.2.5 get_const_slice_lengths()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const size_type* gko::matrix::Selp< ValueType, IndexType >::get_const_slice_lengths ( ) const
[inline], [noexcept]
```

Returns the lengths(columns) of slices.

Returns

the lengths(columns) of slices.

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

References gko::Array< ValueType >::get_const_data().

```
155     {
156         return slice_lengths_.get_const_data();
157     }
```

24.81.2.6 get_const_slice_sets()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const size_type* gko::matrix::Selp< ValueType, IndexType >::get_const_slice_sets ( ) const
[inline], [noexcept]
```

Returns the offsets of slices.

Returns

the offsets of slices.

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

References gko::Array< ValueType >::get_const_data().

```
174     {
175         return slice_sets_.get_const_data();
176     }
```

24.81.2.7 get_const_values()

```
template<typename ValueType = default_precision, typename IndexType = int32>
const value_type* gko::matrix::Sellp< ValueType, IndexType >::get_const_values ( ) const [inline],
[noexcept]
```

Returns the values of the matrix.

Returns

the values of the matrix.

Note

This is the constant version of the function, which can be significantly more memory efficient than the non-constant version, so always prefer this version.

References gko::Array< ValueType >::get_const_data().

```
114     {
115         return values_.get_const_data();
116     }
```

24.81.2.8 get_num_stored_elements()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Sellp< ValueType, IndexType >::get_num_stored_elements ( ) const [inline],
[noexcept]
```

Returns the number of elements explicitly stored in the matrix.

Returns

the number of elements explicitly stored in the matrix

References gko::Array< ValueType >::get_num_elems().

```
205     {
206         return values_.get_num_elems();
207     }
```

24.81.2.9 get_slice_lengths()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type* gko::matrix::Selp< ValueType, IndexType >::get_slice_lengths ( ) [inline], [noexcept]
```

Returns the lengths(columns) of slices.

Returns

the lengths(columns) of slices.

References gko::Array< ValueType >::get_data().

```
143     {
144         return slice_lengths_.get_data();
145     }
```

24.81.2.10 get_slice_sets()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type* gko::matrix::Selp< ValueType, IndexType >::get_slice_sets ( ) [inline], [noexcept]
```

Returns the offsets of slices.

Returns

the offsets of slices.

References gko::Array< ValueType >::get_data().

```
164 { return slice_sets_.get_data(); }
```

24.81.2.11 get_slice_size()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Selp< ValueType, IndexType >::get_slice_size ( ) const [inline],
[noexcept]
```

Returns the size of a slice.

Returns

the size of a slice.

```
183 { return slice_size_; }
```

24.81.2.12 get_stride_factor()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Sellp< ValueType, IndexType >::get_stride_factor ( ) const [inline],
[noexcept]
```

Returns the stride factor(t) of SELL-P.

Returns

the stride factor(t) of SELL-P.

```
190 { return stride_factor_; }
```

24.81.2.13 get_total_cols()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Sellp< ValueType, IndexType >::get_total_cols ( ) const [inline],
[noexcept]
```

Returns the total column number.

Returns

the total column number.

```
197 { return total_cols_; }
```

24.81.2.14 get_values()

```
template<typename ValueType = default_precision, typename IndexType = int32>
value_type* gko::matrix::Sellp< ValueType, IndexType >::get_values ( ) [inline], [noexcept]
```

Returns the values of the matrix.

Returns

the values of the matrix.

References gko::Array< ValueType >::get_data().

```
104 { return values_.get_data(); }
```

24.81.2.15 read()

```
template<typename ValueType = default_precision, typename IndexType = int32>
void gko::matrix::Sellp< ValueType, IndexType >::read (
    const mat_data & data ) [override], [virtual]
```

Reads a matrix from a [mat_data](#) structure.

Parameters

<i>data</i>	the matrix_data structure
-------------	---

Implements [gko::ReadableFromMatrixData< ValueType, IndexType >](#).

24.81.2.16 `val_at()` [1/2]

```
template<typename ValueType = default_precision, typename IndexType = int32>
value_type& gko::matrix::Sellp< ValueType, IndexType >::val_at (
    size_type row,
    size_type slice_set,
    size_type idx ) [inline], [noexcept]
```

Returns the `idx`-th non-zero element of the `row`-th row with `slice_set` slice set.

Parameters

<i>row</i>	the row of the requested element in the slice
<i>slice_set</i>	the slice set of the slice
<i>idx</i>	the <code>idx</code> -th stored element of the row in the slice

Note

the method has to be called on the same [Executor](#) the matrix is stored at (e.g. trying to call this method on a GPU matrix from the CPU results in a runtime error)

References [gko::Array< ValueType >::get_data\(\)](#).

```
223     {
224         return values_.get\_data() [this->linearize_index(row, slice_set, idx)];
225     }
```

24.81.2.17 `val_at()` [2/2]

```
template<typename ValueType = default_precision, typename IndexType = int32>
value_type gko::matrix::Sellp< ValueType, IndexType >::val_at (
    size_type row,
    size_type slice_set,
    size_type idx ) const [inline], [noexcept]
```

Returns the `idx`-th non-zero element of the `row`-th row with `slice_set` slice set.

Parameters

<i>row</i>	the row of the requested element in the slice
<i>slice_set</i>	the slice set of the slice
<i>idx</i>	the <code>idx</code> -th stored element of the row in the slice

Note

the method has to be called on the same [Executor](#) the matrix is stored at (e.g. trying to call this method on a GPU matrix from the CPU results in a runtime error)

References `gko::Array< ValueType >::get_const_data()`.

```

232     {
233         return values_
234             .get_const_data() [this->linearize_index(row, slice_set, idx)];
235     }

```

24.81.2.18 write()

```

template<typename ValueType = default_precision, typename IndexType = int32>
void gko::matrix::Sellp< ValueType, IndexType >::write (
    mat_data & data ) const [override], [virtual]

```

Writes a matrix to a [matrix_data](#) structure.

Parameters

<i>data</i>	the matrix_data structure
-------------	---

Implements [gko::WritableToMatrixData< ValueType, IndexType >](#).

The documentation for this class was generated from the following files:

- `ginkgo/core/matrix/csr.hpp` (ec6dbb34)
- `ginkgo/core/matrix/sellp.hpp` (8045ac75)

24.82 gko::span Struct Reference

A span is a lightweight structure used to create sub-ranges from other ranges.

```
#include <ginkgo/core/base/range.hpp>
```

Public Member Functions

- constexpr [span](#) ([size_type](#) point) noexcept
Creates a span representing a point `point`.
- constexpr [span](#) ([size_type](#) begin, [size_type](#) end) noexcept
Creates a span.
- constexpr bool [is_valid](#) () const
Checks if a span is valid.

Public Attributes

- const `size_type begin`
Beginning of the span.
- const `size_type end`
End of the span.

24.82.1 Detailed Description

A span is a lightweight structure used to create sub-ranges from other ranges.

A span `s` represents a contiguous set of indexes in one dimension of the range, starting on index `s.begin` (inclusive) and ending at index `s.end` (exclusive). A span is only valid if its starting index is smaller than its ending index.

Spans can be compared using the `==` and `!=` operators. Two spans are identical if both their `begin` and `end` values are identical.

Spans also have two distinct partial orders defined on them:

1. `x < y` (`y > x`) if and only if `x.end < y.begin`
2. `x <= y` (`y >= x`) if and only if `x.end <= y.begin`

Note that the orders are in fact partial - there are spans `x` and `y` for which none of the following inequalities holds: `x < y`, `x > y`, `x == y`, `x <= y`, `x >= y`. An example are spans `span{0, 2}` and `span{1, 3}`.

In addition, `<=` is a distinct order from `<`, and not just an extension of the strict order to its weak equivalent. Thus, `x <= y` is not equivalent to `x < y || x == y`.

24.82.2 Constructor & Destructor Documentation

24.82.2.1 `span()` [1/2]

```
constexpr gko::span::span (
    size_type point ) [inline], [noexcept]
```

Creates a span representing a point `point`.

The `begin` of this span is set to `point`, and the `end` to `point + 1`.

Parameters

<i>point</i>	the point which the span represents
--------------	-------------------------------------

```
82         : span{point, point + 1}
83         {}
```

24.82.2.2 span() [2/2]

```
constexpr gko::span::span (
    size_type begin,
    size_type end ) [inline], [noexcept]
```

Creates a span.

Parameters

<i>begin</i>	the beginning of the span
<i>end</i>	the end of the span

References *begin*, and *end*.

```
92         : begin{begin}, end{end}
93     {}
```

24.82.3 Member Function Documentation**24.82.3.1 is_valid()**

```
constexpr bool gko::span::is_valid ( ) const [inline]
```

Checks if a span is valid.

Returns

true if and only if `this->begin < this->end`

References *begin*, and *end*.

Referenced by `gko::accessor::row_major< ValueType, Dimensionality >::operator>()()`.

```
100 { return begin < end; }
```

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

- `ginkgo/core/base/range.hpp` (f1a4eb68)

24.83 gko::stopping_status Class Reference

This class is used to keep track of the stopping status of one vector.

```
#include <ginkgo/core/stop/stopping_status.hpp>
```

Public Member Functions

- bool [has_stopped](#) () const noexcept
Check if any stopping criteria was fulfilled.
- bool [has_converged](#) () const noexcept
Check if convergence was reached.
- bool [is_finalized](#) () const noexcept
Check if the corresponding vector stores the finalized result.
- [uint8 get_id](#) () const noexcept
Get the id of the stopping criterion which caused the stop.
- void [reset](#) () noexcept
Clear all flags.
- void [stop](#) ([uint8 id](#), bool set_finalized=true) noexcept
Call if a stop occured due to a hard limit (and convergence was not reached).
- void [converge](#) ([uint8 id](#), bool set_finalized=true) noexcept
Call if convergence occurred.
- void [finalize](#) () noexcept
Set the result to be finalized (it needs to be stopped or converged first).

Friends

- bool [operator==](#) (const [stopping_status](#) &x, const [stopping_status](#) &y) noexcept
Checks if two stopping statuses are equivalent.
- bool [operator!=](#) (const [stopping_status](#) &x, const [stopping_status](#) &y) noexcept
Checks if two stopping statuses are different.

24.83.1 Detailed Description

This class is used to keep track of the stopping status of one vector.

24.83.2 Member Function Documentation

24.83.2.1 [converge\(\)](#)

```
void gko::stopping_status::converge (
    uint8 id,
    bool set_finalized = true ) [inline], [noexcept]
```

Call if convergence occurred.

Parameters

<i>id</i>	id of the stopping criteria.
<i>set_finalized</i>	Controls if the current version should count as finalized (set to true) or not (set to false).

References `has_stopped()`.

```
124     {
125         if (!this->has_stopped()) {
126             data_ |= converged_mask_ | (id & id_mask_);
127             if (set_finalized) {
128                 data_ |= finalized_mask_;
129             }
130         }
131     }
```

24.83.2.2 `get_id()`

```
uint8 gko::stopping_status::get_id ( ) const [inline], [noexcept]
```

Get the id of the stopping criterion which caused the stop.

Returns

Returns the id of the stopping criterion which caused the stop.

Referenced by `has_stopped()`.

```
89     {
90         return data_ & id_mask_;
91     }
```

24.83.2.3 `has_converged()`

```
bool gko::stopping_status::has_converged ( ) const [inline], [noexcept]
```

Check if convergence was reached.

Returns

Returns true if convergence was reached.

```
70     {
71         return data_ & converged_mask_;
72     }
```

24.83.2.4 has_stopped()

```
bool gko::stopping_status::has_stopped ( ) const [inline], [noexcept]
```

Check if any stopping criteria was fulfilled.

Returns

Returns true if any stopping criteria was fulfilled.

References `get_id()`.

Referenced by `converge()`, `finalize()`, and `stop()`.

```
61     {
62         return get_id();
63     }
```

24.83.2.5 is_finalized()

```
bool gko::stopping_status::is_finalized ( ) const [inline], [noexcept]
```

Check if the corresponding vector stores the finalized result.

Returns

Returns true if the corresponding vector stores the finalized result.

```
80     {
81         return data_ & finalized_mask_;
82     }
```

24.83.2.6 stop()

```
void gko::stopping_status::stop (
    uint8 id,
    bool set_finalized = true ) [inline], [noexcept]
```

Call if a stop occurred due to a hard limit (and convergence was not reached).

Parameters

<i>id</i>	id of the stopping criteria.
<i>set_finalized</i>	Controls if the current version should count as finalized (set to true) or not (set to false).

References `has_stopped()`.

```
107     {
108         if (!this->has_stopped()) {
109             data_ |= (id & id_mask_);
110             if (set_finalized) {
111                 data_ |= finalized_mask_;
112             }
113         }
114     }
```

24.83.3 Friends And Related Function Documentation

24.83.3.1 operator!=

```
bool operator!= (
    const stopping_status & x,
    const stopping_status & y ) [friend]
```

Checks if two stopping statuses are different.

Parameters

<i>x</i>	a stopping status
<i>y</i>	a stopping status

Returns

true if and only if ! (x == y)

```
179 {
180     return x.data_ != y.data_;
181 }
```

24.83.3.2 operator==

```
bool operator== (
    const stopping_status & x,
    const stopping_status & y ) [friend]
```

Checks if two stopping statuses are equivalent.

Parameters

<i>x</i>	a stopping status
<i>y</i>	a stopping status

Returns

true if and only if both `x` and `y` have the same mask and converged and finalized state

```
164 {
165     return x.data_ == y.data_;
166 }
```

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

- ginkgo/core/stop/stopping_status.hpp (f1a4eb68)

24.84 gko::matrix::Hybrid< ValueType, IndexType >::strategy_type Class Reference

`strategy_type` is to decide how to set the hybrid config.

```
#include <ginkgo/core/matrix/hybrid.hpp>
```

Public Member Functions

- `strategy_type ()`
Creates a `strategy_type`.
- `void compute_hybrid_config (const Array< size_type > &row_nnz, size_type *ell_num_stored_elements_↵
per_row, size_type *coo_nnz)`
Computes the config of the `Hybrid` matrix (`ell_num_stored_elements_per_row` and `coo_nnz`).
- `size_type get_ell_num_stored_elements_per_row () const noexcept`
Returns the number of stored elements per row of the `ell` part.
- `size_type get_coo_nnz () const noexcept`
Returns the number of nonzeros of the `coo` part.
- `virtual size_type compute_ell_num_stored_elements_per_row (Array< size_type > *row_nnz) const =0`
Computes the number of stored elements per row of the `ell` part.

24.84.1 Detailed Description

```
template<typename ValueType = default_precision, typename IndexType = int32>
class gko::matrix::Hybrid< ValueType, IndexType >::strategy_type
```

`strategy_type` is to decide how to set the hybrid config.

It computes the number of stored elements per row of the `ell` part and then set the number of residual nonzeros as the number of nonzeros of the `coo` part.

The practical strategy method should inherit `strategy_type` and implement its `compute_ell_num_stored_↵
elements_per_row` function.

24.84.2 Member Function Documentation

24.84.2.1 compute_ell_num_stored_elements_per_row()

```
template<typename ValueType = default_precision, typename IndexType = int32>
virtual size_type gko::matrix::Hybrid< ValueType, IndexType >::strategy_type::compute_ell_↵  
num_stored_elements_per_row (
    Array< size_type > * row_nnz ) const [pure virtual]
```

Computes the number of stored elements per row of the `ell` part.

Parameters

<i>row_nnz</i>	the number of nonzeros of each row
----------------	------------------------------------

Returns

the number of stored elements per row of the ell part

Implemented in [gko::matrix::Hybrid< ValueType, IndexType >::automatic](#), [gko::matrix::Hybrid< ValueType, IndexType >::minimal_storage_limit](#), [gko::matrix::Hybrid< ValueType, IndexType >::imbalance_bounded_limit](#), [gko::matrix::Hybrid< ValueType, IndexType >::imbalance_limit](#), and [gko::matrix::Hybrid< ValueType, IndexType >::column_limit](#).

Referenced by [gko::matrix::Hybrid< ValueType, IndexType >::strategy_type::compute_hybrid_config\(\)](#), and [gko::matrix::Hybrid< ValueType, IndexType >::strategy_type::get_coo_nnz\(\)](#).

24.84.2.2 compute_hybrid_config()

```
template<typename ValueType = default_precision, typename IndexType = int32>
void gko::matrix::Hybrid< ValueType, IndexType >::strategy_type::compute_hybrid_config (
    const Array< size_type > & row_nnz,
    size_type * ell_num_stored_elements_per_row,
    size_type * coo_nnz ) [inline]
```

Computes the config of the [Hybrid](#) matrix ([ell_num_stored_elements_per_row](#) and [coo_nnz](#)).

For now, it copies [row_nnz](#) to the reference executor and performs all operations on the reference executor.

Parameters

<i>row_nnz</i>	the number of nonzeros of each row
<i>ell_num_stored_elements_per_row</i>	the output number of stored elements per row of the ell part
<i>coo_nnz</i>	the output number of nonzeros of the coo part

References [gko::matrix::Hybrid< ValueType, IndexType >::strategy_type::compute_ell_num_stored_elements_per_row\(\)](#), [gko::Array< ValueType >::get_executor\(\)](#), and [gko::Array< ValueType >::get_num_elems\(\)](#).

```
126         {
127             Array<size_type> ref_row_nnz(row_nnz.get_executor()->get_master(),
128                                         row_nnz.get_num_elems());
129             ref_row_nnz = row_nnz;
130             ell_num_stored_elements_per_row_ =
131             this->compute_ell_num_stored_elements_per_row(&
ref_row_nnz);
132             coo_nnz_ = this->compute_coo_nnz(ref_row_nnz);
133             *ell_num_stored_elements_per_row = ell_num_stored_elements_per_row_;
134             *coo_nnz = coo_nnz_;
135         }
```

24.84.2.3 get_coo_nnz()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Hybrid< ValueType, IndexType >::strategy_type::get_coo_nnz ( ) const
[inline], [noexcept]
```

Returns the number of nonzeros of the coo part.

Returns

the number of nonzeros of the coo part

References `gko::matrix::Hybrid< ValueType, IndexType >::strategy_type::compute_ell_num_stored_elements_per_row()`, `gko::Array< ValueType >::get_const_data()`, and `gko::Array< ValueType >::get_num_elems()`.

```
152 { return coo_nnz_; }
```

24.84.2.4 get_ell_num_stored_elements_per_row()

```
template<typename ValueType = default_precision, typename IndexType = int32>
size_type gko::matrix::Hybrid< ValueType, IndexType >::strategy_type::get_ell_num_stored_
elements_per_row ( ) const [inline], [noexcept]
```

Returns the number of stored elements per row of the ell part.

Returns

the number of stored elements per row of the ell part

```
143     {
144         return ell_num_stored_elements_per_row_;
145     }
```

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

- `ginkgo/core/matrix/hybrid.hpp` (3e51a52b)

24.85 gko::log::Stream< ValueType > Class Template Reference

`Stream` is a Logger which logs every event to a stream.

```
#include <ginkgo/core/log/stream.hpp>
```

Static Public Member Functions

- static `std::unique_ptr< Stream > create` (`std::shared_ptr< const Executor > exec`, `const Logger::mask_type &enabled_events=Logger::all_events_mask`, `std::ostream &os=std::cout`, `bool verbose=false`)

Creates a `Stream` logger.

24.85.1 Detailed Description

```
template<typename ValueType = default_precision>
class gko::log::Stream< ValueType >
```

`Stream` is a Logger which logs every event to a stream.

This can typically be used to log to a file or to the console.

Template Parameters

<i>ValueType</i>	the type of values stored in the class (i.e. <i>ValueType</i> template parameter of the concrete Loggable this class will log)
------------------	--

24.85.2 Member Function Documentation

24.85.2.1 create()

```
template<typename ValueType = default_precision>
static std::unique_ptr<Stream> gko::log::Stream< ValueType >::create (
    std::shared_ptr< const Executor > exec,
    const Logger::mask_type & enabled_events = Logger::all_events_mask,
    std::ostream & os = std::cout,
    bool verbose = false ) [inline], [static]
```

Creates a [Stream](#) logger.

This dynamically allocates the memory, constructs the object and returns an `std::unique_ptr` to this object.

Parameters

<i>exec</i>	the executor
<i>enabled_events</i>	the events enabled for this logger. By default all events.
<i>os</i>	the stream used for this logger
<i>verbose</i>	whether we want detailed information or not. This includes always printing residuals and other information which can give a large output.

Returns

an `std::unique_ptr` to the the constructed object

```
178     {
179         return std::unique_ptr<Stream>(
180             new Stream(exec, enabled_events, os, verbose));
181     }
```

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

- `ginkgo/core/log/stream.hpp` (f1a4eb68)

24.86 gko::StreamError Class Reference

[StreamError](#) is thrown if accessing a stream failed.

```
#include <ginkgo/core/base/exception.hpp>
```

Public Member Functions

- [StreamError](#) (const std::string &file, int line, const std::string &func, const std::string &message)
Initializes a file access error.

24.86.1 Detailed Description

[StreamError](#) is thrown if accessing a stream failed.

24.86.2 Constructor & Destructor Documentation

24.86.2.1 StreamError()

```
gko::StreamError::StreamError (
    const std::string & file,
    int line,
    const std::string & func,
    const std::string & message ) [inline]
```

Initializes a file access error.

Parameters

<i>file</i>	The name of the offending source file
<i>line</i>	The source code line number where the error occurred
<i>func</i>	The name of the function that tried to access the file
<i>message</i>	The error message

```
330         : Error(file, line, func + ": " + message)
331     {}
```

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

- ginkgo/core/base/exception.hpp (8045ac75)

24.87 gko::temporary_clone< T > Class Template Reference

A [temporary_clone](#) is a special smart pointer-like object that is designed to hold an object temporarily copied to another executor.

```
#include <ginkgo/core/base/utils.hpp>
```

Public Member Functions

- [temporary_clone](#) (std::shared_ptr< const [Executor](#) > exec, pointer ptr)
Creates a [temporary_clone](#).
- T * [get](#) () const
Returns the object held by [temporary_clone](#).
- T * [operator->](#) () const
Calls a method on the underlying object.

24.87.1 Detailed Description

```
template<typename T>
class gko::temporary_clone< T >
```

A [temporary_clone](#) is a special smart pointer-like object that is designed to hold an object temporarily copied to another executor.

After the [temporary_clone](#) goes out of scope, the stored object will be copied back to its original location. This class is optimized to avoid copies if the object is already on the correct executor, in which case it will just hold a reference to that object, without performing the copy.

Template Parameters

T	the type of object held in the temporary_clone
-------------------	--

24.87.2 Constructor & Destructor Documentation

24.87.2.1 temporary_clone()

```
template<typename T >
gko::temporary_clone< T >::temporary_clone (
    std::shared_ptr< const Executor > exec,
    pointer ptr ) [inline], [explicit]
```

Creates a [temporary_clone](#).

Parameters

<i>exec</i>	the executor where the clone will be created
<i>ptr</i>	a pointer to the object of which the clone will be created

References gko::clone().

```
429     {
430         if (ptr->get_executor() == exec) {
```

```

431         // just use the object we already have
432         handle_ = handle_type(ptr, [] (pointer) {});
433     } else {
434         // clone the object to the new executor and make sure it's copied
435         // back before we delete it
436         handle_ = handle_type(gko::clone(std::move(exec), ptr).release(),
437                               copy_back_deleter<T>(ptr));
438     }
439 }

```

24.87.3 Member Function Documentation

24.87.3.1 get()

```

template<typename T >
T* gko::temporary_clone< T >::get ( ) const [inline]

```

Returns the object held by [temporary_clone](#).

Returns

the object held by [temporary_clone](#)

```

446 { return handle_.get(); }

```

24.87.3.2 operator->()

```

template<typename T >
T* gko::temporary_clone< T >::operator-> ( ) const [inline]

```

Calls a method on the underlying object.

Returns

the underlying object

```

453 { return handle_.get(); }

```

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

- ginkgo/core/base/utils.hpp (4bde4271)

24.88 gko::stop::Time Class Reference

The [Time](#) class is a stopping criterion which stops the iteration process after a certain amount of time has passed.

```
#include <ginkgo/core/stop/time.hpp>
```

24.88.1 Detailed Description

The [Time](#) class is a stopping criterion which stops the iteration process after a certain amount of time has passed.

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

- ginkgo/core/stop/time.hpp (f1a4eb68)

24.89 gko::Transposable Class Reference

Linear operators which support transposition should implement the [Transposable](#) interface.

```
#include <ginkgo/core/base/lin_op.hpp>
```

Public Member Functions

- virtual std::unique_ptr< LinOp > [transpose](#) () const =0
Returns a LinOp representing the transpose of the [Transposable](#) object.
- virtual std::unique_ptr< LinOp > [conj_transpose](#) () const =0
Returns a LinOp representing the conjugate transpose of the [Transposable](#) object.

24.89.1 Detailed Description

Linear operators which support transposition should implement the [Transposable](#) interface.

It provides two functionalities, the normal transpose and the conjugate transpose.

The normal transpose returns the transpose of the linear operator without changing any of its elements representing the operation, $B = A^T$.

The conjugate transpose returns the conjugate of each of the elements and additionally transposes the linear operator representing the operation, $B = A^H$.

Example: Transposing a Csr matrix:

```
{c++}
//Transposing an object of LinOp type.
//The object you want to transpose.
auto op = matrix::Csr::create(exec);
//Transpose the object by first converting it to a transposable type.
auto trans = op->transpose();
```

24.89.2 Member Function Documentation

24.89.2.1 conj_transpose()

```
virtual std::unique_ptr<LinOp> gko::Transposable::conj_transpose ( ) const [pure virtual]
```

Returns a LinOp representing the conjugate transpose of the [Transposable](#) object.

Returns

a pointer to the new conjugate transposed object

Implemented in [gko::matrix::Csr< ValueType, IndexType >](#), and [gko::matrix::Dense< ValueType >](#).

24.89.2.2 transpose()

```
virtual std::unique_ptr<LinOp> gko::Transposable::transpose ( ) const [pure virtual]
```

Returns a LinOp representing the transpose of the [Transposable](#) object.

Returns

a pointer to the new transposed object

Implemented in [gko::matrix::Csr< ValueType, IndexType >](#), and [gko::matrix::Dense< ValueType >](#).

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

- [ginkgo/core/base/lin_op.hpp](#) (fb72cdf1)

24.90 gko::stop::Criterion::Updater Class Reference

The [Updater](#) class serves for convenient argument passing to the [Criterion](#)'s check function.

```
#include <ginkgo/core/stop/criterion.hpp>
```

Public Member Functions

- [Updater](#) (const [Updater](#) &)=delete
Prevent copying and moving the object This is to enforce the use of argument passing and calling check at the same time.
- bool [check](#) (uint8 stoppingId, bool setFinalized, [Array](#)< [stopping_status](#) > *stop_status, bool *one_changed) const
Calls the parent [Criterion](#) object's check method.

24.90.1 Detailed Description

The [Updater](#) class serves for convenient argument passing to the [Criterion](#)'s check function.

The pattern used is a Builder, except [Updater](#) builds a function's arguments before calling the function itself, and does not build an object. This allows calling a [Criterion](#)'s check in the form of: `stop_criterion->update().num_↵ iterations(num_iterations).residual_norm(residual_norm).residual(residual).solution(solution).check(converged);`

If there is a need for a new form of data to pass to the [Criterion](#), it should be added here.

24.90.2 Member Function Documentation

24.90.2.1 check()

```
bool gko::stop::Criterion::Updater::check (
    uint8 stoppingId,
    bool setFinalized,
    Array< stopping_status > * stop_status,
    bool * one_changed ) const [inline]
```

Calls the parent [Criterion](#) object's check method.

References `gko::stop::Criterion::check()`, and `Updater()`.

```
99         {
100             auto converged = parent_->check(stoppingId, setFinalized,
101                                             stop_status, one_changed, *this);
102             return converged;
103         }
```

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

- `ginkgo/core/stop/criterion.hpp` (f0a50f96)

24.91 gko::version Struct Reference

This structure is used to represent versions of various Ginkgo modules.

```
#include <ginkgo/core/base/version.hpp>
```

Public Attributes

- const [uint64](#) `major`
The major version number.
- const [uint64](#) `minor`
The minor version number.
- const [uint64](#) `patch`
The patch version number.
- const char *const `tag`
Addition tag string that describes the version in more detail.

24.91.1 Detailed Description

This structure is used to represent versions of various Ginkgo modules.

Version structures can be compared using the usual relational operators.

24.91.2 Member Data Documentation

24.91.2.1 tag

```
const char* const gko::version::tag
```

Addition tag string that describes the version in more detail.

It does not participate in comparisons.

Referenced by `gko::operator<<()`.

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

- `ginkgo/core/base/version.hpp` (8045ac75)

24.92 gko::version_info Class Reference

Ginkgo uses version numbers to label new features and to communicate backward compatibility guarantees:

```
#include <ginkgo/core/base/version.hpp>
```

Static Public Member Functions

- static const [version_info](#) & `get` ()
Returns an instance of [version_info](#).

Public Attributes

- [version_header_version](#)
Contains version information of the header files.
- [version_core_version](#)
Contains version information of the core library.
- [version_reference_version](#)
Contains version information of the reference module.
- [version_omp_version](#)
Contains version information of the OMP module.
- [version_cuda_version](#)
Contains version information of the CUDA module.

24.92.1 Detailed Description

Ginkgo uses version numbers to label new features and to communicate backward compatibility guarantees:

1. Versions with different major version number have incompatible interfaces (parts of the earlier interface may not be present anymore, and new interfaces can appear).
2. Versions with the same major number X, but different minor numbers Y1 and Y2 numbers keep the same interface as version X.0.0, but additions to the interface in X.0.0 present in X.Y1.0 may not be present in X.Y2.0 and vice versa.
3. Versions with the same major and minor version numbers, but different patch numbers have exactly the same interface, but the functionality may be different (something that is not implemented or has a bug in an earlier version may have this implemented or fixed in a later version).

This structure provides versions of different parts of Ginkgo: the headers, the core and the kernel modules (reference, OpenMP, CUDA). To obtain an instance of [version_info](#) filled with information about the current version of Ginkgo, call the [version_info::get\(\)](#) static method.

24.92.2 Member Function Documentation

24.92.2.1 get()

```
static const version\_info& gko::version_info::get ( ) [inline], [static]
```

Returns an instance of [version_info](#).

Returns

an instance of version info

```
148     {
149         static version\_info info{};
150         return info;
151     }
```

24.92.3 Member Data Documentation

24.92.3.1 core_version

```
version gko::version_info::core_version
```

Contains version information of the core library.

This is the version of the static/shared library called "ginkgo".

24.92.3.2 cuda_version

`version` `gko::version_info::cuda_version`

Contains version information of the CUDA module.

This is the version of the static/shared library called "ginkgo_cuda".

24.92.3.3 omp_version

`version` `gko::version_info::omp_version`

Contains version information of the OMP module.

This is the version of the static/shared library called "ginkgo_omp".

24.92.3.4 reference_version

`version` `gko::version_info::reference_version`

Contains version information of the reference module.

This is the version of the static/shared library called "ginkgo_reference".

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

- ginkgo/core/base/version.hpp (8045ac75)

24.93 gko::WritableToMatrixData< ValueType, IndexType > Class Template Reference

A LinOp implementing this interface can write its data to a `matrix_data` structure.

```
#include <ginkgo/core/base/lin_op.hpp>
```

Public Member Functions

- virtual void `write` (`matrix_data`< ValueType, IndexType > &data) const =0
Writes a matrix to a `matrix_data` structure.

24.93.1 Detailed Description

```
template<typename ValueType, typename IndexType>
class gko::WritableToMatrixData< ValueType, IndexType >
```

A LinOp implementing this interface can write its data to a `matrix_data` structure.

24.93.2 Member Function Documentation

24.93.2.1 write()

```
template<typename ValueType, typename IndexType>
virtual void gko::WritableToMatrixData< ValueType, IndexType >::write (
    matrix_data< ValueType, IndexType > & data ) const [pure virtual]
```

Writes a matrix to a `matrix_data` structure.

Parameters

<i>data</i>	the matrix_data structure
-------------	---

Implemented in [gko::matrix::Hybrid< ValueType, IndexType >](#), [gko::matrix::Csr< ValueType, IndexType >](#), [gko::matrix::preconditioner::Jacobi< ValueType, IndexType >](#), [gko::matrix::Coo< ValueType, IndexType >](#), [gko::matrix::Ell< ValueType, IndexType >](#), and [gko::matrix::Sellp< ValueType, IndexType >](#).

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

- [ginkgo/core/base/lin_op.hpp](#) (fb72cdf1)

Index

- abs
 - gko, [183](#)
- add_logger
 - gko::log::EnableLogging, [296](#)
 - gko::log::Loggable, [339](#)
- add_scaled
 - gko::matrix::Dense, [272](#)
- alloc
 - gko::Executor, [303](#)
- AllocationError
 - gko::AllocationError, [215](#)
- apply2
 - gko::matrix::Coo, [248](#), [249](#)
- Array
 - gko::Array, [217–222](#)
- array
 - gko, [182](#)
- as
 - gko, [183](#), [184](#)
- at
 - gko::matrix::Dense, [272–274](#)
- autodetect
 - gko::precision_reduction, [370](#)
- CUDA Executor, [149](#)
- ceildiv
 - gko, [184](#)
- check
 - gko::stop::Criterion, [255](#)
 - gko::stop::Criterion::Updater, [413](#)
- clear
 - gko::Array, [222](#)
 - gko::PolymorphicObject, [363](#)
- clone
 - gko, [185](#)
 - gko::PolymorphicObject, [364](#)
- col_at
 - gko::matrix::Ell, [286](#)
 - gko::matrix::Sellp, [389](#), [390](#)
- column_limit
 - gko::matrix::Hybrid::column_limit, [238](#)
- combine
 - Stopping criteria, [173](#)
- common
 - gko::precision_reduction, [370](#)
- compute_dot
 - gko::matrix::Dense, [275](#)
- compute_ell_num_stored_elements_per_row
 - gko::matrix::Hybrid::automatic, [228](#)
 - gko::matrix::Hybrid::column_limit, [238](#)
 - gko::matrix::Hybrid::imbalance_bounded_limit, [326](#)
 - gko::matrix::Hybrid::imbalance_limit, [328](#)
 - gko::matrix::Hybrid::minimal_storage_limit, [352](#)
 - gko::matrix::Hybrid::strategy_type, [404](#)
- compute_hybrid_config
 - gko::matrix::Hybrid::strategy_type, [405](#)
- compute_norm2
 - gko::matrix::Dense, [275](#)
- compute_storage_space
 - gko::preconditioner::block_interleaved_storage_scheme, [231](#)
- cond
 - gko::matrix_data, [345](#), [346](#)
- conj
 - gko, [186](#)
- conj_transpose
 - gko::Transposable, [411](#)
 - gko::matrix::Csr, [259](#)
 - gko::matrix::Dense, [275](#)
- converge
 - gko::stopping_status, [400](#)
- convert_to
 - gko::ConvertibleTo, [245](#)
 - gko::EnablePolymorphicAssignment, [298](#)
 - gko::preconditioner::Jacobi, [332](#)
- coordinate
 - gko, [182](#)
- copy_and_convert_to
 - gko, [187](#)
- copy_back_deleter
 - gko::copy_back_deleter, [254](#)
- copy_from
 - gko::Executor, [304](#)
 - gko::PolymorphicObject, [365](#)
 - gko::accessor::row_major, [386](#)
- core_version
 - gko::version_info, [415](#)
- create
 - gko::CudaExecutor, [267](#)
 - gko::EnableDefaultFactory, [294](#)
 - gko::log::Convergence, [243](#)
 - gko::log::Record, [382](#)
 - gko::log::Stream, [407](#)
 - gko::matrix::IdentityFactory, [325](#)
- create_default
 - gko::PolymorphicObject, [366](#), [367](#)
- create_submatrix
 - gko::matrix::Dense, [276](#)
- create_with_config_of

- gko::matrix::Dense, 276
- CublasError
 - gko::CublasError, 264
- cuda_version
 - gko::version_info, 415
- CudaError
 - gko::CudaError, 265
- CusparsError
 - gko::CusparsError, 269
- diag
 - gko::matrix_data, 347, 348, 350, 351
- dim
 - gko::dim, 280
- DimensionMismatch
 - gko::DimensionMismatch, 283
- ell_col_at
 - gko::matrix::Hybrid, 314
- ell_val_at
 - gko::matrix::Hybrid, 315
- EnableDefaultCriterionFactory
 - gko::stop, 211
- EnableDefaultLinOpFactory
 - Linear Operators, 160
- Error
 - gko::Error, 301
- executor_deleter
 - gko::executor_deleter, 307
- Executors, 150
 - GKO_REGISTER_OPERATION, 151
- Factorizations, 153
- free
 - gko::Executor, 305
- GKO_CREATE_FACTORY_PARAMETERS
 - Linear Operators, 157
- GKO_ENABLE_BUILD_METHOD
 - Linear Operators, 157
- GKO_ENABLE_CRITERION_FACTORY
 - Stopping criteria, 172
- GKO_ENABLE_LIN_OP_FACTORY
 - Linear Operators, 158
- GKO_FACTORY_PARAMETER
 - Linear Operators, 160
- GKO_REGISTER_OPERATION
 - Executors, 151
- generate
 - gko::AbstractFactory, 214
- get
 - gko::log::Record, 382
 - gko::temporary_clone, 410
 - gko::version_info, 415
- get_accessor
 - gko::range, 376
- get_block_offset
 - gko::preconditioner::block_interleaved_storage_↔
scheme, 232
- get_blocks
 - gko::preconditioner::Jacobi, 333
- get_coefficients
 - gko::Combination, 240
- get_col_idxs
 - gko::matrix::Coo, 250
 - gko::matrix::Csr, 259
 - gko::matrix::Ell, 287
 - gko::matrix::Sellp, 390
- get_conditioning
 - gko::preconditioner::Jacobi, 333
- get_const_col_idxs
 - gko::matrix::Coo, 250
 - gko::matrix::Csr, 259
 - gko::matrix::Ell, 287
 - gko::matrix::Sellp, 391
- get_const_coo_col_idxs
 - gko::matrix::Hybrid, 316
- get_const_coo_row_idxs
 - gko::matrix::Hybrid, 316
- get_const_coo_values
 - gko::matrix::Hybrid, 316
- get_const_data
 - gko::Array, 222
- get_const_ell_col_idxs
 - gko::matrix::Hybrid, 317
- get_const_ell_values
 - gko::matrix::Hybrid, 317
- get_const_row_idxs
 - gko::matrix::Coo, 250
- get_const_row_ptrs
 - gko::matrix::Csr, 259
- get_const_slice_lengths
 - gko::matrix::Sellp, 391
- get_const_slice_sets
 - gko::matrix::Sellp, 392
- get_const_srow
 - gko::matrix::Csr, 260
- get_const_values
 - gko::matrix::Coo, 251
 - gko::matrix::Csr, 260
 - gko::matrix::Dense, 277
 - gko::matrix::Ell, 287
 - gko::matrix::Sellp, 392
- get_coo
 - gko::matrix::Hybrid, 318
- get_coo_col_idxs
 - gko::matrix::Hybrid, 318
- get_coo_nnz
 - gko::matrix::Hybrid::strategy_type, 405
- get_coo_num_stored_elements
 - gko::matrix::Hybrid, 318
- get_coo_row_idxs
 - gko::matrix::Hybrid, 319
- get_coo_values
 - gko::matrix::Hybrid, 319
- get_cublas_handle
 - gko::CudaExecutor, 267

- get_cuspars_handle
 - gko::CudaExecutor, [267](#)
- get_data
 - gko::Array, [223](#)
- get_dynamic_type
 - gko::name_demangling, [208](#)
- get_ell
 - gko::matrix::Hybrid, [319](#)
- get_ell_col_idxs
 - gko::matrix::Hybrid, [320](#)
- get_ell_num_stored_elements
 - gko::matrix::Hybrid, [320](#)
- get_ell_num_stored_elements_per_row
 - gko::matrix::Hybrid, [320](#)
 - gko::matrix::Hybrid::strategy_type, [406](#)
- get_ell_stride
 - gko::matrix::Hybrid, [321](#)
- get_ell_values
 - gko::matrix::Hybrid, [321](#)
- get_executor
 - gko::Array, [223](#)
 - gko::PolymorphicObject, [367](#)
- get_global_block_offset
 - gko::preconditioner::block_interleaved_storage_↔
scheme, [232](#)
- get_group_offset
 - gko::preconditioner::block_interleaved_storage_↔
scheme, [233](#)
- get_group_size
 - gko::preconditioner::block_interleaved_storage_↔
scheme, [233](#)
- get_id
 - gko::stopping_status, [401](#)
- get_krylov_dim
 - gko::solver::Gmres, [311](#)
- get_master
 - gko::CudaExecutor, [267](#), [268](#)
 - gko::Executor, [305](#)
 - gko::OmpExecutor, [357](#), [358](#)
- get_name
 - gko::Operation, [360](#)
- get_nonpreserving
 - gko::precision_reduction, [370](#)
- get_num_blocks
 - gko::preconditioner::Jacobi, [333](#)
- get_num_elems
 - gko::Array, [224](#)
- get_num_iterations
 - gko::log::Convergence, [243](#)
- get_num_srow_elements
 - gko::matrix::Csr, [261](#)
- get_num_stored_elements
 - gko::matrix::Coo, [251](#)
 - gko::matrix::Csr, [261](#)
 - gko::matrix::Dense, [277](#)
 - gko::matrix::Ell, [288](#)
 - gko::matrix::Hybrid, [321](#)
 - gko::matrix::Sellp, [393](#)
 - gko::preconditioner::Jacobi, [334](#)
- get_num_stored_elements_per_row
 - gko::matrix::Ell, [288](#)
- get_operators
 - gko::Combination, [240](#)
 - gko::Composition, [242](#)
- get_parameters
 - gko::EnableDefaultFactory, [295](#)
- get_preconditioner
 - gko::Preconditionable, [372](#)
 - gko::solver::Bicgstab, [229](#)
 - gko::solver::Cg, [235](#)
 - gko::solver::Cgs, [237](#)
 - gko::solver::Fcgs, [309](#)
 - gko::solver::Gmres, [311](#)
- get_preserving
 - gko::precision_reduction, [371](#)
- get_residual
 - gko::log::Convergence, [244](#)
- get_residual_norm
 - gko::log::Convergence, [244](#)
- get_row_idxs
 - gko::matrix::Coo, [252](#)
- get_row_ptrs
 - gko::matrix::Csr, [261](#)
- get_significant_bit
 - gko, [188](#)
- get_slice_lengths
 - gko::matrix::Sellp, [393](#)
- get_slice_sets
 - gko::matrix::Sellp, [394](#)
- get_slice_size
 - gko::matrix::Sellp, [394](#)
- get_solver
 - gko::solver::lr, [329](#)
- get_srow
 - gko::matrix::Csr, [262](#)
- get_static_type
 - gko::name_demangling, [208](#)
- get_storage_scheme
 - gko::preconditioner::Jacobi, [334](#)
- get_strategy
 - gko::matrix::Csr, [262](#)
 - gko::matrix::Hybrid, [322](#)
- get_stride
 - gko::matrix::Dense, [278](#)
 - gko::matrix::Ell, [289](#)
 - gko::preconditioner::block_interleaved_storage_↔
scheme, [233](#)
- get_stride_factor
 - gko::matrix::Sellp, [394](#)
- get_superior_power
 - gko, [189](#)
- get_system_matrix
 - gko::solver::Bicgstab, [230](#)
 - gko::solver::Cg, [235](#)
 - gko::solver::Cgs, [237](#)
 - gko::solver::Fcgs, [309](#)

- gko::solver::Gmres, 311
- gko::solver::Irr, 330
- get_total_cols
 - gko::matrix::Sellp, 395
- get_values
 - gko::matrix::Coo, 252
 - gko::matrix::Csr, 262
 - gko::matrix::Dense, 278
 - gko::matrix::Ell, 289
 - gko::matrix::Sellp, 395
- give
 - gko, 189
- gko, 175
 - abs, 183
 - array, 182
 - as, 183, 184
 - ceildiv, 184
 - clone, 185
 - conj, 186
 - coordinate, 182
 - copy_and_convert_to, 187
 - get_significant_bit, 188
 - get_superior_power, 189
 - give, 189
 - imag, 190
 - is_complex, 190
 - is_complex_s, 182
 - isfinite, 191
 - layout_type, 182
 - lend, 191, 192
 - make_temporary_clone, 192
 - max, 193
 - min, 194
 - one, 194, 195
 - operator!=, 195, 196
 - operator<<, 197
 - operator==, 198
 - read, 198
 - read_raw, 199
 - real, 200
 - round_down, 200
 - round_up, 201
 - share, 201
 - squared_norm, 202
 - transpose, 202
 - write, 203
 - write_raw, 204
 - zero, 204, 205
- gko::AbstractFactory
 - generate, 214
- gko::AbstractFactory< AbstractProductType, ComponentsType >, 213
- gko::AllocationError, 214
 - AllocationError, 215
- gko::Array
 - Array, 217–222
 - clear, 222
 - get_const_data, 222
 - get_data, 223
 - get_executor, 223
 - get_num_elems, 224
 - operator=, 224, 225
 - resize_and_reset, 226
 - set_executor, 226
 - view, 227
- gko::Array< ValueType >, 215
- gko::Combination
 - get_coefficients, 240
 - get_operators, 240
- gko::Combination< ValueType >, 239
- gko::Composition
 - get_operators, 242
- gko::Composition< ValueType >, 241
- gko::ConvertibleTo
 - convert_to, 245
 - move_to, 246
- gko::ConvertibleTo< ResultType >, 244
- gko::CublasError, 264
 - CublasError, 264
- gko::CudaError, 265
 - CudaError, 265
- gko::CudaExecutor, 266
 - create, 267
 - get_cublas_handle, 267
 - get_cuspars_handle, 267
 - get_master, 267, 268
 - run, 268
- gko::CusparsError, 268
 - CusparsError, 269
- gko::DimensionMismatch, 283
 - DimensionMismatch, 283
- gko::EnableAbstractPolymorphicObject< AbstractObject, PolymorphicBase >, 292
- gko::EnableCreateMethod< ConcreteType >, 293
- gko::EnableDefaultFactory
 - create, 294
 - get_parameters, 295
- gko::EnableDefaultFactory< ConcreteFactory, ProductType, ParametersType, PolymorphicBase >, 293
- gko::EnableLinOp< ConcreteLinOp, PolymorphicBase >, 295
- gko::EnablePolymorphicAssignment
 - convert_to, 298
 - move_to, 299
- gko::EnablePolymorphicAssignment< ConcreteType, ResultType >, 298
- gko::EnablePolymorphicObject< ConcreteObject, PolymorphicBase >, 299
- gko::Error, 300
 - Error, 301
- gko::Executor, 302
 - alloc, 303
 - copy_from, 304
 - free, 305
 - get_master, 305

- run, [305](#), [306](#)
- gko::KernelNotFound, [336](#)
 - KernelNotFound, [337](#)
- gko::LinOpFactory, [338](#)
- gko::NotCompiled, [353](#)
 - NotCompiled, [353](#)
- gko::NotImplemented, [354](#)
 - NotImplemented, [354](#)
- gko::NotSupported, [355](#)
 - NotSupported, [355](#)
- gko::OmpExecutor, [357](#)
 - get_master, [357](#), [358](#)
- gko::Operation, [358](#)
 - get_name, [360](#)
- gko::OutOfBoundsError, [360](#)
 - OutOfBoundsError, [361](#)
- gko::PolymorphicObject, [363](#)
 - clear, [363](#)
 - clone, [364](#)
 - copy_from, [365](#)
 - create_default, [366](#), [367](#)
 - get_executor, [367](#)
- gko::Preconditionable, [372](#)
 - get_preconditioner, [372](#)
- gko::ReadableFromMatrixData
 - read, [380](#)
- gko::ReadableFromMatrixData< ValueType, IndexType >, [379](#)
- gko::ReferenceExecutor, [383](#)
 - run, [383](#)
- gko::StreamError, [407](#)
 - StreamError, [408](#)
- gko::Transposable, [411](#)
 - conj_transpose, [411](#)
 - transpose, [412](#)
- gko::WritableToMatrixData
 - write, [416](#)
- gko::WritableToMatrixData< ValueType, IndexType >, [416](#)
- gko::accessor, [205](#)
- gko::accessor::row_major
 - copy_from, [386](#)
 - length, [386](#)
 - operator(), [387](#)
- gko::accessor::row_major< ValueType, Dimensionality >, [384](#)
- gko::copy_back_deleter
 - copy_back_deleter, [254](#)
 - operator(), [254](#)
- gko::copy_back_deleter< T >, [253](#)
- gko::default_converter
 - operator(), [270](#)
- gko::default_converter< S, R >, [269](#)
- gko::dim
 - dim, [280](#)
 - operator bool, [281](#)
 - operator*, [282](#)
 - operator==, [282](#)
 - operator[], [281](#), [282](#)
- gko::dim< Dimensionality, DimensionType >, [279](#)
- gko::enable_parameters_type
 - on, [292](#)
- gko::enable_parameters_type< ConcreteParameters, Type, Factory >, [291](#)
- gko::executor_deleter
 - executor_deleter, [307](#)
 - operator(), [308](#)
- gko::executor_deleter< T >, [307](#)
- gko::factorization, [206](#)
- gko::factorization::Parllu< ValueType, IndexType >, [361](#)
- gko::log, [206](#)
- gko::log::Convergence
 - create, [243](#)
 - get_num_iterations, [243](#)
 - get_residual, [244](#)
 - get_residual_norm, [244](#)
- gko::log::Convergence< ValueType >, [242](#)
- gko::log::EnableLogging
 - add_logger, [296](#)
 - remove_logger, [297](#)
- gko::log::EnableLogging< ConcreteLoggable, PolymorphicBase >, [296](#)
- gko::log::Loggable, [339](#)
 - add_logger, [339](#)
 - remove_logger, [340](#)
- gko::log::Record, [381](#)
 - create, [382](#)
 - get, [382](#)
- gko::log::Record::logged_data, [340](#)
- gko::log::Stream
 - create, [407](#)
- gko::log::Stream< ValueType >, [406](#)
- gko::log::criterion_data, [256](#)
- gko::log::executor_data, [306](#)
- gko::log::iteration_complete_data, [331](#)
- gko::log::linop_data, [337](#)
- gko::log::linop_factory_data, [337](#)
- gko::log::operation_data, [360](#)
- gko::log::polymorphic_object_data, [362](#)
- gko::matrix, [207](#)
- gko::matrix::Coo
 - apply2, [248](#), [249](#)
 - get_col_idxxs, [250](#)
 - get_const_col_idxxs, [250](#)
 - get_const_row_idxxs, [250](#)
 - get_const_values, [251](#)
 - get_num_stored_elements, [251](#)
 - get_row_idxxs, [252](#)
 - get_values, [252](#)
 - read, [252](#)
 - write, [253](#)
- gko::matrix::Coo< ValueType, IndexType >, [247](#)
- gko::matrix::Csr
 - conj_transpose, [259](#)
 - get_col_idxxs, [259](#)
 - get_const_col_idxxs, [259](#)

- get_const_row_ptrs, 259
- get_const_srow, 260
- get_const_values, 260
- get_num_srow_elements, 261
- get_num_stored_elements, 261
- get_row_ptrs, 261
- get_srow, 262
- get_strategy, 262
- get_values, 262
- read, 263
- transpose, 263
- write, 263
- gko::matrix::Csr< ValueType, IndexType >, 257
- gko::matrix::Dense
 - add_scaled, 272
 - at, 272–274
 - compute_dot, 275
 - compute_norm2, 275
 - conj_transpose, 275
 - create_submatrix, 276
 - create_with_config_of, 276
 - get_const_values, 277
 - get_num_stored_elements, 277
 - get_stride, 278
 - get_values, 278
 - scale, 278
 - transpose, 279
- gko::matrix::Dense< ValueType >, 270
- gko::matrix::Ell
 - col_at, 286
 - get_col_idxxs, 287
 - get_const_col_idxxs, 287
 - get_const_values, 287
 - get_num_stored_elements, 288
 - get_num_stored_elements_per_row, 288
 - get_stride, 289
 - get_values, 289
 - read, 289
 - val_at, 290
 - write, 291
- gko::matrix::Ell< ValueType, IndexType >, 284
- gko::matrix::Hybrid
 - ell_col_at, 314
 - ell_val_at, 315
 - get_const_coo_col_idxxs, 316
 - get_const_coo_row_idxxs, 316
 - get_const_coo_values, 316
 - get_const_ell_col_idxxs, 317
 - get_const_ell_values, 317
 - get_coo, 318
 - get_coo_col_idxxs, 318
 - get_coo_num_stored_elements, 318
 - get_coo_row_idxxs, 319
 - get_coo_values, 319
 - get_ell, 319
 - get_ell_col_idxxs, 320
 - get_ell_num_stored_elements, 320
 - get_ell_num_stored_elements_per_row, 320
 - get_ell_stride, 321
 - get_ell_values, 321
 - get_num_stored_elements, 321
 - get_strategy, 322
 - operator=, 322
 - read, 323
 - write, 323
- gko::matrix::Hybrid< ValueType, IndexType >, 312
- gko::matrix::Hybrid< ValueType, IndexType >↔
 - ::automatic, 227
- gko::matrix::Hybrid< ValueType, IndexType >↔
 - ::column_limit, 238
- gko::matrix::Hybrid< ValueType, IndexType >↔
 - ::imbalance_bounded_limit, 325
- gko::matrix::Hybrid< ValueType, IndexType >↔
 - ::imbalance_limit, 327
- gko::matrix::Hybrid< ValueType, IndexType >↔
 - ::minimal_storage_limit, 352
- gko::matrix::Hybrid< ValueType, IndexType >↔
 - ::strategy_type, 404
- gko::matrix::Hybrid::automatic
 - compute_ell_num_stored_elements_per_row, 228
- gko::matrix::Hybrid::column_limit
 - column_limit, 238
 - compute_ell_num_stored_elements_per_row, 238
- gko::matrix::Hybrid::imbalance_bounded_limit
 - compute_ell_num_stored_elements_per_row, 326
- gko::matrix::Hybrid::imbalance_limit
 - compute_ell_num_stored_elements_per_row, 328
 - imbalance_limit, 327
- gko::matrix::Hybrid::minimal_storage_limit
 - compute_ell_num_stored_elements_per_row, 352
- gko::matrix::Hybrid::strategy_type
 - compute_ell_num_stored_elements_per_row, 404
 - compute_hybrid_config, 405
 - get_coo_nnz, 405
 - get_ell_num_stored_elements_per_row, 406
- gko::matrix::Identity< ValueType >, 324
- gko::matrix::IdentityFactory
 - create, 325
- gko::matrix::IdentityFactory< ValueType >, 324
- gko::matrix::Selp
 - col_at, 389, 390
 - get_col_idxxs, 390
 - get_const_col_idxxs, 391
 - get_const_slice_lengths, 391
 - get_const_slice_sets, 392
 - get_const_values, 392
 - get_num_stored_elements, 393
 - get_slice_lengths, 393
 - get_slice_sets, 394
 - get_slice_size, 394
 - get_stride_factor, 394
 - get_total_cols, 395
 - get_values, 395
 - read, 395
 - val_at, 396
 - write, 397

gko::matrix::Sellp< ValueType, IndexType >, 388
 gko::matrix_data
 cond, 345, 346
 diag, 347, 348, 350, 351
 matrix_data, 342–345
 nonzeros, 351
 gko::matrix_data< ValueType, IndexType >, 341
 gko::matrix_data< ValueType, IndexType >::nonzero↔
 _type, 353
 gko::name_demangling, 207
 get_dynamic_type, 208
 get_static_type, 208
 gko::null_deleter
 operator(), 356
 gko::null_deleter< T >, 356
 gko::precision_reduction, 368
 autodetect, 370
 common, 370
 get_nonpreserving, 370
 get_preserving, 371
 operator storage_type, 371
 precision_reduction, 369
 gko::preconditioner, 209
 gko::preconditioner::Jacobi
 convert_to, 332
 get_blocks, 333
 get_conditioning, 333
 get_num_blocks, 333
 get_num_stored_elements, 334
 get_storage_scheme, 334
 move_to, 334
 write, 336
 gko::preconditioner::Jacobi< ValueType, IndexType >, 331
 gko::preconditioner::block_interleaved_storage_scheme
 compute_storage_space, 231
 get_block_offset, 232
 get_global_block_offset, 232
 get_group_offset, 233
 get_group_size, 233
 get_stride, 233
 group_power, 234
 gko::preconditioner::block_interleaved_storage_↔
 scheme< IndexType >, 230
 gko::range
 get_accessor, 376
 length, 376
 operator(), 377
 operator->, 378
 operator=, 378
 range, 376
 gko::range< Accessor >, 372
 gko::solver, 209
 gko::solver::Bicgstab
 get_preconditioner, 229
 get_system_matrix, 230
 gko::solver::Bicgstab< ValueType >, 229
 gko::solver::Cg
 get_preconditioner, 235
 get_system_matrix, 235
 gko::solver::Cg< ValueType >, 234
 gko::solver::Cgs
 get_preconditioner, 237
 get_system_matrix, 237
 gko::solver::Cgs< ValueType >, 236
 gko::solver::Fcg
 get_preconditioner, 309
 get_system_matrix, 309
 gko::solver::Fcg< ValueType >, 308
 gko::solver::Gmres
 get_krylov_dim, 311
 get_preconditioner, 311
 get_system_matrix, 311
 gko::solver::Gmres< ValueType >, 310
 gko::solver::lr
 get_solver, 329
 get_system_matrix, 330
 gko::solver::lr< ValueType >, 328
 gko::span, 397
 is_valid, 399
 span, 398, 399
 gko::stop, 210
 EnableDefaultCriterionFactory, 211
 gko::stop::Combined, 241
 gko::stop::Criterion, 255
 check, 255
 update, 256
 gko::stop::Criterion::Updater, 412
 check, 413
 gko::stop::CriterionArgs, 257
 gko::stop::Iteration, 330
 gko::stop::ResidualNormReduction< ValueType >, 384
 gko::stop::Time, 410
 gko::stopping_status, 399
 converge, 400
 get_id, 401
 has_converged, 401
 has_stopped, 401
 is_finalized, 402
 operator!=, 403
 operator==, 403
 stop, 402
 gko::syn, 211
 gko::temporary_clone
 get, 410
 operator->, 410
 temporary_clone, 409
 gko::temporary_clone< T >, 408
 gko::version, 413
 tag, 414
 gko::version_info, 414
 core_version, 415
 cuda_version, 415
 get, 415
 omp_version, 416
 reference_version, 416

- gko::xstd, 211
- group_power
 - gko::preconditioner::block_interleaved_storage_scheme, 234
- has_converged
 - gko::stopping_status, 401
- has_stopped
 - gko::stopping_status, 401
- imag
 - gko, 190
- imbalance_limit
 - gko::matrix::Hybrid::imbalance_limit, 327
- initialize
 - SpMV employing different Matrix formats, 164–166
- is_complex
 - gko, 190
- is_complex_s
 - gko, 182
- is_finalized
 - gko::stopping_status, 402
- is_valid
 - gko::span, 399
- isfinite
 - gko, 191
- KernelNotFound
 - gko::KernelNotFound, 337
- layout_type
 - gko, 182
- lend
 - gko, 191, 192
- length
 - gko::accessor::row_major, 386
 - gko::range, 376
- Linear Operators, 154
 - EnableDefaultLinOpFactory, 160
 - GKO_CREATE_FACTORY_PARAMETERS, 157
 - GKO_ENABLE_BUILD_METHOD, 157
 - GKO_ENABLE_LIN_OP_FACTORY, 158
 - GKO_FACTORY_PARAMETER, 160
- Logging, 162
- make_temporary_clone
 - gko, 192
- matrix_data
 - gko::matrix_data, 342–345
- max
 - gko, 193
- min
 - gko, 194
- move_to
 - gko::ConvertibleTo, 246
 - gko::EnablePolymorphicAssignment, 299
 - gko::preconditioner::Jacobi, 334
- nonzeros
 - gko::matrix_data, 351
- NotCompiled
 - gko::NotCompiled, 353
- NotImplemented
 - gko::NotImplemented, 354
- NotSupported
 - gko::NotSupported, 355
- omp_version
 - gko::version_info, 416
- on
 - gko::enable_parameters_type, 292
- one
 - gko, 194, 195
- OpenMP Executor, 168
- operator bool
 - gko::dim, 281
- operator storage_type
 - gko::precision_reduction, 371
- operator!=
 - gko, 195, 196
 - gko::stopping_status, 403
- operator<<
 - gko, 197
- operator*
 - gko::dim, 282
- operator()
 - gko::accessor::row_major, 387
 - gko::copy_back_deleter, 254
 - gko::default_converter, 270
 - gko::executor_deleter, 308
 - gko::null_deleter, 356
 - gko::range, 377
- operator->
 - gko::range, 378
 - gko::temporary_clone, 410
- operator=
 - gko::Array, 224, 225
 - gko::matrix::Hybrid, 322
 - gko::range, 378
- operator==
 - gko, 198
 - gko::dim, 282
 - gko::stopping_status, 403
- operator[]
 - gko::dim, 281, 282
- OutOfBoundsError
 - gko::OutOfBoundsError, 361
- precision_reduction
 - gko::precision_reduction, 369
- Preconditioners, 169
- range
 - gko::range, 376
- read
 - gko, 198
 - gko::ReadableFromMatrixData, 380
 - gko::matrix::Coo, 252
 - gko::matrix::Csr, 263

- gko::matrix::Ell, [289](#)
- gko::matrix::Hybrid, [323](#)
- gko::matrix::Sellp, [395](#)
- read_raw
 - gko, [199](#)
- real
 - gko, [200](#)
- Reference Executor, [170](#)
- reference_version
 - gko::version_info, [416](#)
- remove_logger
 - gko::log::EnableLogging, [297](#)
 - gko::log::Loggable, [340](#)
- resize_and_reset
 - gko::Array, [226](#)
- round_down
 - gko, [200](#)
- round_up
 - gko, [201](#)
- run
 - gko::CudaExecutor, [268](#)
 - gko::Executor, [305](#), [306](#)
 - gko::ReferenceExecutor, [383](#)
- scale
 - gko::matrix::Dense, [278](#)
- set_executor
 - gko::Array, [226](#)
- share
 - gko, [201](#)
- Solvers, [171](#)
- SpMV employing different Matrix formats, [163](#)
 - initialize, [164–166](#)
- span
 - gko::span, [398](#), [399](#)
- squared_norm
 - gko, [202](#)
- stop
 - gko::stopping_status, [402](#)
- Stopping criteria, [172](#)
 - combine, [173](#)
 - GKO_ENABLE_CRITERION_FACTORY, [172](#)
- StreamError
 - gko::StreamError, [408](#)
- tag
 - gko::version, [414](#)
- temporary_clone
 - gko::temporary_clone, [409](#)
- transpose
 - gko, [202](#)
 - gko::Transposable, [412](#)
 - gko::matrix::Csr, [263](#)
 - gko::matrix::Dense, [279](#)
- update
 - gko::stop::Criterion, [256](#)
- val_at
 - gko::matrix::Ell, [290](#)
 - gko::matrix::Sellp, [396](#)
- view
 - gko::Array, [227](#)
- write
 - gko, [203](#)
 - gko::WritableToMatrixData, [416](#)
 - gko::matrix::Coo, [253](#)
 - gko::matrix::Csr, [263](#)
 - gko::matrix::Ell, [291](#)
 - gko::matrix::Hybrid, [323](#)
 - gko::matrix::Sellp, [397](#)
 - gko::preconditioner::Jacobi, [336](#)
- write_raw
 - gko, [204](#)
- zero
 - gko, [204](#), [205](#)