MAGMA-sparse Interface Design Whitepaper

ECP PEEKS: Production-ready, Exascale-Enabled, Krylov Solvers for Exascale Computing

Hartwig Anzt
Erik Boman
Jack Dongarra
Goran Flegar
Mark Gates
Mike Heroux
Mark Hoemmen
Jakub Kurzak
Piotr Luszczek
Sivasankaran Rajamanickam
Stanimire Tomov
Stephen Wood
Ichitaro Yamazaki

Innovative Computing Laboratory, University of Tennessee
Sandia National Laboratory

June 29, 2017

ICL INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, in support of the nation's exascale computing imperative.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energys National Nuclear Security Administration under contract DE-NA-0003525.

Summary

In this report we describe the logic and interface we develop for the MAGMA-sparse library to allow for easy integration as third-party library into a top-level software ecosystem. The design choices are based on extensive consultation with other software library developers, in particular the Trilinos software development team. The interface documentation is at this point not exhaustive, but a first proposal for setting a standard. Although the interface description targets the MAGMA-sparse software module, we hope that the design choices carry beyond this specific library, and are attractive for adoption in other packages.

This report is not intended as static document, but will be updated over time to reflect the agile software development in the ECP 1.3.3.11 STMS11-PEEKS project.

Contents

CHAPTER 1

Introduction

What is MAGMA-sparse?

MAGMA-sparse is a high performance sparse linear algebra library targeting a single node consisting of multicore and manycore processors. The library focuses on solving sparse linear systems and provides a large variety of matrix formats, state-of-the-art iterative (Krylov) solvers and preconditioners, which make the library suitable for a variety of different scientific applications. The routines incorporated in the library often consist of bleeding-edge research from both, the core developers, as well as from other contributors from the HPC community.

Currently, MAGMA-sparse supports execution on a single CPU (running a single thread) accompanied by an NVIDIA GPU (Kepler or newer architecture) accelerator. The library design is driven by supporting both, multi-threaded execution, as well as supporting different types of accelerators (AMD GPUs, Intel Xeon Phi). Full support for these architectures will be added in the near future. The programming language of choice is C++ (C++11 standard) as it enables high performance while also providing various language features (data abstraction, generic programming and automatic memory management) which facilitate the use and the maintenance of the library.

An exception to the C++11 standard are the low-level computational kernels, which usually do not use language features that could affect performance. The kernels may even be implemented in a different language if C++11 results in suboptimal performance, or is not even supported on the target device (e.g. computational kernels for NVIDIA GPUs are implemented using CUDA C++).

CHAPTER 2

Design choices

Users First

A common problem for HPC libraries is their complexity and generality, which often goes hand-in-hand with a steep learning curve. For example, the BLAS GEMV routine for matrix-vector multiplication has 11 input parameters, while the simple use case for matrix-vector product involves only three entities: the matrix, the input vector, and the result vector. If the same approach was used for the MAGMA-sparse interface, the parameter list would grow even further as: 1) the data structure used to store a sparse matrix is more complex than the equivalent in the dense case, 2) sparse solvers often have several (optional) parameters which modify their behaviour (shadow space dimension for IDR, preconditioner for any Krylov solver), and 3) it is usually required to provide metadata describing where and in which format the data entities live.

Since one of the goals of this library is to offer state-of-the-art methods for spare computations to a broader community, the main goal of the interface design is to hide complexity, while also enabling a high level of flexibility. In summary: even though complex algorithms should be possible, simple steps should be easy to realize. For example, a simple matrix-vector product should not take 11, but only 3 parameters. On the other hand, a complex use case, like GEMV-type $y = \alpha Ax + \beta y$ should still be possible to realize.

C vs. C++

Library logic and interface

In the tradition of BLAS, LAPACK, the programming language C (or classical FORTRAN) might be considered as the program language for HPC libraries. In the past, and in particular for dense libraries with a moderate complexity level, this choice may have been quite reasonable. The primary reason is that in this case, function calls and interfaces could be designed independent of the data and the hardware architecture. Furthermore, the BLAS and LAPACK standard allows only for a limited scope of combining multiple functions to generate new functionality. At the same time, more sophisticated libraries like Trilinos or deal.II make heavy use of C++ features as these allow for a much higher level of flexibility in terms of algorithm design and hardware usage. A primary reason for this choice is that these libraries aim at covering a much larger application spectrum which virtually forbids the implementation of every possible use case. But also on the BLAS/LAPACK side there exist efforts moving more into the C++ direction. In this case, the motivation is not the proliferation of functionality, but the hardware-specific optimization of the code. Fore example, interleaving data or blocking techniques to increase performance suggest to handle matrices as library-owned opaque objects where the user may and should not have access to. Based on this assessment we propose a library logic based on C++, which naturally results in a C++ interface. Acknowledging the wide-spread use of C-based code in the HPC community, we additionally provide C-bindings.

Low-level kernels

At the other end of the library ecosystem are the low-level kernels, typically implemented in parallelism-supporting languages such as OpenCL, OpenMP, CUDA, or the Kokkos programming model form Sandia National Laboratories, among others. While some of these programming models allow for better cross-platform portability, the price is typically lower performance compared to hardware-specific implementations making heavy use of the low-level optimization features. Also, it is necessary to mention that some hardware architectures provide only limited support for C++ features. Ultimately, the kernel language choice primarily boils down to the available manpower and the cross-platform portability the software aims for whether a more generic kernel implementation or a higher degree of hardware-specific optimization should be preferred. Another point of consideration is to encourage community contributions. According to our assessment, none of the generic programming models have become the de-facto standard at this point, and it may pose a burden to potential contributors to expect contributed kernels to be implemented in a non-standard library. Acknowledging the higher library maintenance effort but hoping for more community contributions and higher performance, we design the low-level kernels with a C-interface and using the distinct vendor-provided programming models as these are expected to give the best performance.

C++ classes

Using a library logic based on C++ naturally suggests the cascaded use of classes and subclasses to encapsulate objects that compose of several parts into a single instance. A simple example is a sparse matrix stored in CSR format and hence composed of three arrays (row pointer, column indices, values) and secondary information such as size, number of nonzero elements, memory location, etc.

The Linear Operator

On an abstract level, any matrix, solver, or preconditioner can be interpreted as a linear operator. And any of these linear operators can be applied to a vector. Examples are the (sparse) matrix vector product being the application of a (sparse) matrix operator; the iterative solution of a linear system being the application of a solver operator; or the preconditioner operator application. Aside from the application, there are two additional ”phases” in the lifetime of a linear operator: the generation and the destruction. The generation can be, e.g., reading a matrix from disk, generating a sparse approximate inverse matrix, or preparing an iterative solver. This motivates to pool all matrices, solvers, preconditioners in a ”super-class” we call ”Linear Operator” (LinOp). All members of this class provide the basic functionalities ”generate”, ”apply”, and ”destroy”. The realization of these general instructions is specific to the distinct subclasses and objects.

Smart Pointers

In order to automate the memory management, MAGMA-sparse uses standard smart pointer classes: `std::unique_ptr` and `std::shared_ptr`. This means that the user is never required to explicitly allocate or free the memory when working with (the C++ interface of) MAGMA-sparse. Instead, this is handled automatically by the library.

Smart Pointers as a Form of Documentation

The use of different kinds of pointers, in combination with the `const` qualifier, helps to document the effect of the routine on that parameter, as well as the visibility of the parameter from within the library after the routine returns:

- If the input parameter is `const Obj* obj` the routine will not modify `obj`. The library will keep a reference to `obj` only until the routine returns.
- If the input parameter is `Obj* obj` the routine may modify `obj`. The library will keep a reference to `obj` only until the routine returns.
- If the input parameter is `std::shared_ptr<const Obj> obj` the routine will not modify `obj`. The library may keep a reference to `obj` even after the routine returns (but it will never modify it). Thus, any changes on `obj` may change the effects of future library calls.
- If the input parameter is `std::shared_ptr<Obj> obj` the routine may modify `obj`. The library may keep a reference to `obj`, and modify it, even after the routine returns. Thus, any changes on the `obj` may change the effects of future library calls, and future library calls can modify `obj`.
- If the input parameter is `std::unique_ptr<Obj> obj` the ownership of `obj` is transferred to the routine (and the library). Thus, the original `obj` **must not be used** after the call to such routine, as at that point the library may have already deleted the object.

In addition, the output parameters of the routines can either be an `std::shared_ptr` or a `std::unique_ptr` to an object. In case of the former, the library will also keep a reference to the object (and may modify it after the routine returns), while in the latter case, the caller is given exclusive access to the returned object.

Conversion Between Smart Pointer Types

It is often required to convert between different types of pointers. The three pointer types: plain pointer, `std::shared_ptr` and `std::unique_ptr` form an ordered set in terms of ownership requirement, where the plain pointer does not imply any ownership requirement, the `std::shared_ptr` implies shared ownership of the object, and the `std::unique_ptr` implies unique ownership of the object. As a result, arbitrary conversions are not possible, but only in the direction of decreasing ownership requirements. Concretely:

- `std::unique_ptr` can be converted to a plain pointer by calling the `get()` method of the pointer object, and to a `std::shared_ptr` by using `std::move` to move the ownership to a `std::shared_ptr` (note that the latter conversion consumes the original `std::unique_ptr` and makes it unusable in the future).
- `std::shared_ptr` can be converted to a plain pointer by calling the `get()` method of the pointer object, but cannot be converted to an `std::unique_ptr`, as there might exist other references to the object.
- A plain pointer can neither be converted to a `std::shared_ptr` nor to an `std::unique_ptr`, as a plain pointer does not have any ownership of the object.

Templating and Polymorphism

Precision generation

To reduce the programming effort, we use templating for generating the precision formats. The standard formats being generated are double complex (z) single complex (c), double real (d), and single real (s). The precision generation is invoked when compiling the code. At this point the code generated for the distinct precisions cannot be combined, each precision should be considered as a stand-alone code.

Polymorphism for handling Linear Operators

A main difference between Templates and Polymorphism is that templated code fills in the specific types at compile time, while polymorphic code decides at runtime during algorithm execution. As a result, templated code typically gives better performance, while polymorphic code gives more flexibility and the possibility to pre-compile code.

The library design is guided by providing a high level of flexibility while still providing also a C-interface. In terms of flexibility, we want to allow for choosing any linear operator as preconditioner (Matrix, Factorization, or, again, a Solver). This allows to generate cascaded solvers like Flexible-GMRES or Iterative Refinement with an ILU-preconditioned BiCGSTAB as inner solver.

Using templates, we cannot compile using generic code that allows for cascading linear operators, as this would result in recursive initialization process. The only option is to implement the different combinations. The large number of possible combinations, however, makes this approach unattractive.

Polymorphism also has advantages also with respect to supporting a C interface.

These advantages outweigh the performance advantage of Templates, and we decided to use Polymorphism for the Linear Operator class and all subclasses.

Error handling via Exceptions

Each library call can throw a C++ exception containing the error information. A major advantage of C++ exceptions over other error models is robustness and usability: ”forgetting” to check for errors will in no case cause a hard-to-track bug in the code, as the application/library call terminates immediately after an exception is encountered. Additionally to a error-specific return code, we design ”exception analysis” routines that translate error codes into human-readable text and provide more information about possible causes for the error.

Library use and naming

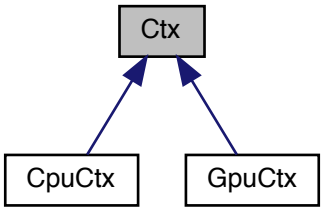
We aim at a generic interface that is attractive for other libraries to adopt while keeping self-explaining but short naming conventions for usability. In vision of several libraries using the same interface a logic design being used simultaneously in an application, we use a library-specific prefix ”msparse” in the c-interface, and the ”msparse” namespace in the C++ interface.

CHAPTER 3

Context

3.1 Ctx

Inheritance diagram for Ctx:



3.1.1 Detailed Description

The first step in using the MAGMA-sparse library consists of creating a context. Contexts are used to specify the location for the data of linear algebra objects, and to determine where the operations will be executed. MAGMA-sparse currently supports two different context types:

- CpuCtx specifies that the data should be stored and the associated operations executed on the host CPU,
- GpuCtx specifies that the data should be stored and the operations operations executed on the NVIDIA GPU accelerator.

The following code snippet demonstrates the simplest possible use of the MAGMA-sparse library:

```
auto cpu = msparse::create<msparse::CpuCtx>();
auto A = msparse::read_from_mtx<msparse::CsrMatrix<float>>("A.mtx", cpu);
```

First, we create a CPU context, 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 a matrix market file 'A.mtx', and store the data on the CPU in CSR format (msparse::CsrMatrix is a MAGMA-sparse 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 gpu = msparse::create<msparse::GpuCtx>(0, cpu);
auto dA = msparse::copy_to<msparse::CsrMatrix<float>>(A.get(), gpu);
```

The first line of the snippet creates a new GPU context. Since there may be multiple 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 CpuCtx which will be used to schedule the requested GPU 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 MAGMA-sparse aims to provide automatic memory management of its objects, the result of calling msparse::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 msparse::copy_to is a plain pointer. Thus, the get() routine is used to convert from a std::unique_ptr to a plain pointer expected by the routine.

As a side note, the msparse::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 msparse::EllMatrix as the template parameter, dA would be stored on the GPU, in ELLPACK format.

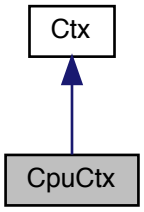
Finally, if all the processing of the matrix is supposed to be done on the GPU, and a CPU copy of the matrix is not required, we could have read the matrix to the GPU directly:

```
auto cpu = msparse::create<msparse::CpuCtx>();
auto gpu = msparse::create<msparse::GpuCtx>(0, cpu);
auto dA = msparse::read_from_mtx<msparse::CsrMatrix<float>>("A.mtx", gpu);
```

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.

3.2 CpuCtx

Inheritance diagram for CpuCtx:



Static Public Member Functions

- static std::shared_ptr< CpuCtx > create ()

3.2.1 Detailed Description

This is the Ctx subclass which represents the CPU device.

3.2.2 Member Function Documentation

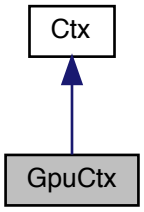
create()

```
static std::shared_ptr<CpuCtx> create ( )  [static]
```

Creates a new CpuCtx.

3.3 GpuCtx

Inheritance diagram for GpuCtx:



Static Public Member Functions

- static std::shared_ptr< GpuCtx > create (int device, std::shared_ptr< CpuCtx > cpu_ctx)

3.3.1 Detailed Description

This is the Ctx subclass which represents the GPU device.

3.3.2 Member Function Documentation

create()

```
static std::shared_ptr<GpuCtx> create (
              int device,
              std::shared_ptr< CpuCtx > cpu_ctx )  [static]
```

Creates a new GpuCtx.

Parameters

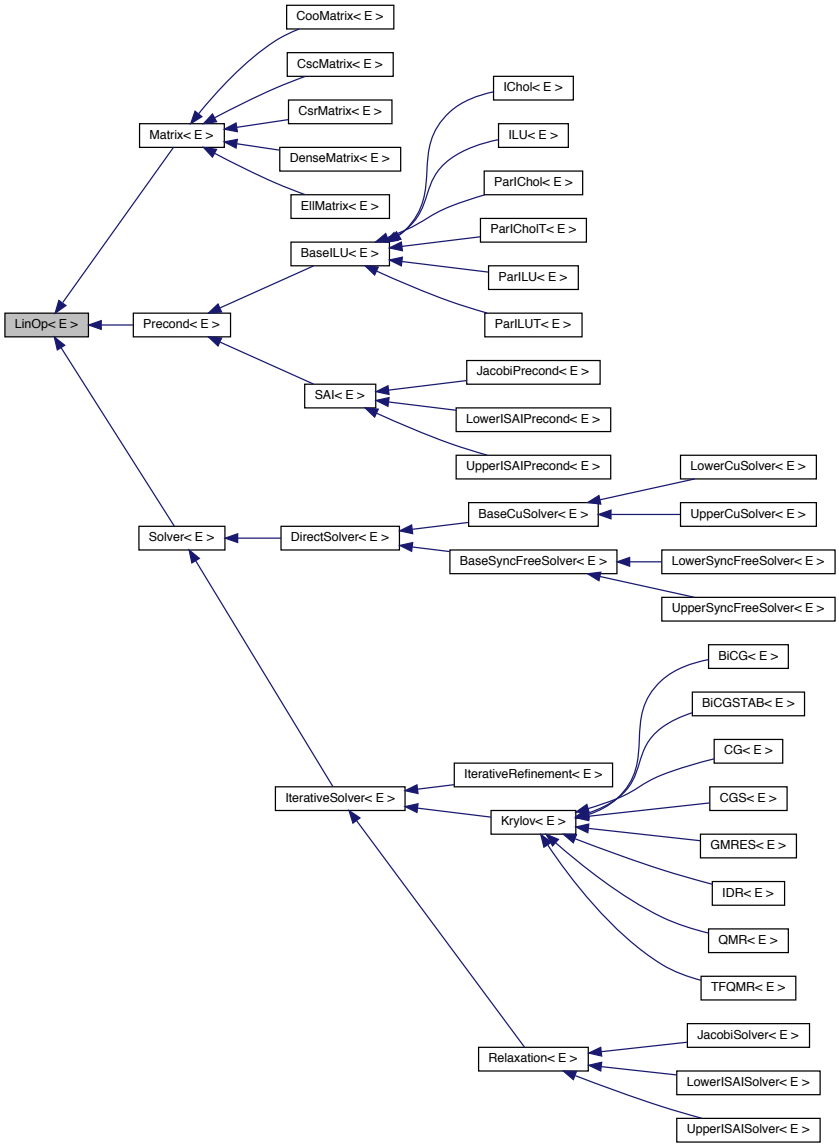
device	the CUDA device number of this device
cpu_ctx	a CPU context used to invoke the device kernels

CHAPTER 4

Linear Operator

4.1 LinOp< E >

Inheritance diagram for LinOp< E >:



Classes

- class Info

Public Member Functions

- virtual void copy_from (const LinOp< E > *other)=0
- virtual void copy_from (std::unique_ptr< LinOp< E >> other)=0
- virtual void make_trans_of (const LinOp< E > *other)=0
- virtual void make_conj_trans_of (const LinOp< E > *other)=0
- virtual void generate (std::shared_ptr< const LinOp< E >> op)=0
- virtual void apply (const DenseMatrix< E > *b, DenseMatrix< E > *x) const
- virtual void apply (E alpha, const DenseMatrix< E > *b, E beta, DenseMatrix< E > *x) const
- virtual std::shared_ptr< Info > get_info () const =0
- virtual std::unique_ptr< LinOp > clone_type () const =0
- std::shared_ptr< const Ctx > get_ctx () const
- size_type get_num_rows () const
- size_type get_num_cols () const
- size_type get_num_nonzeros () const
- virtual void clear ()

4.1.1 Detailed Description

template<typename E>
class msparse::LinOp< E >

The linear operator (LinOp) is a base class for all linear algebra objects in MAGMA-sparse. 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, MAGMA-sparse 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 MAGMA-sparse 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 liner 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}.

An accompanying routine to the linear operator application is its generation from another linear operator A: a solver operator A^{-1} is generated by applying the inverse function, a preconditioner operator M^{-1} by applying the approximate inverse function, and a matrix by just applying the identity operator $id : A \to A$. Thus, every LinOp subclass has a meaningful implementation of the LinOp::generate() routine, which performs the above mentioned operation.

Formally speaking, a MAGMA-sparse LinOp subclass does not represent a linear operator, but in fact a (non-linear) operator op on the space of all linear operators (on a certain vector space). In case of matrices op is the identity operator, in case of solvers the inverse operator, and in case of preconditioners a type of approximate inverse operator (with different preconditioner classes representing different operators). The LinOp::generate() routine can then be viewed as an application of this operator to a linear operator A, yielding the result $op(A)$ and the LinOp::apply() routine as an application $y = op(A)x$ of the resulting linear operator to a vector x.

Finally, direct manipulation of LinOp objects is rarely required in simple scenarios. As an ilustrative example, one could construct a fixed-point iteration routine $x_{k+1} = Lx_k + b$ as follows:

```
std::unique_ptr<msparse::DenseMatrix<double>> calculate_fixed_point(
        int iters,
        const msparse::LinOp<double> *L,
        const msparse::DenseMatrix<double> *x0
        const msparse::DenseMatrix<double> *b)
{
    auto x = msparse::clone(x0.get());
    auto tmp = msparse::clone(x0.get());
    for (int i = 0; i < iters; ++i) {
        L->apply(tmp.get(), x.get());
        x->axpy(1.0, b.get());
```

```
        tmp->copy_from(x.get());
    }
    return x;
}
```

Here, as L is a matrix, LinOp::apply() refers to the matrix vector product, and `L->apply(a, b)` computes $b = L \cdot a$. `x->axpy(1.0, 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.

Template Parameters

E	the precision the data is stored in

4.1.2 Member Function Documentation

copy_from() [1/2]

```
virtual void copy_from (
              const LinOp< E > * other )  [pure virtual]
```

Create a copy of another LinOp.

Parameters

other	the LinOp to copy

Exceptions

NotSupported	other is of incompatible type

copy_from() [2/2]

```
virtual void copy_from (
              std::unique_ptr< LinOp< E >> other )  [pure virtual]
```

Move the data from another LinOp.

Parameters

other	the LinOp from which the data will be moved

Exceptions

NotSupported	other is of incompatible type

make_trans_of()

```
virtual void make_trans_of (
              const LinOp< E > * other )  [pure virtual]
```

Create a transpose of another LinOp.

Parameters

other	the LinOp to transpose

Exceptions

NotSupported	other is of incompatible type

make_conj_trans_of()

```
virtual void make_conj_trans_of (
              const LinOp< E > * other )  [pure virtual]
```

Create a conjugate transpose of another LinOp.

Parameters

other	the LinOp to conjugate transpose

Exceptions

NotSupported	other is of incompatible type

generate()

```
virtual void generate (
              std::shared_ptr< const LinOp< E >> op )  [pure virtual]
```

Generate a new LinOp from another operator.

While the LinOp::copy_from routine creates an exact copy of the operator, this routine creates a representation of the operator f(op), where f depends on the category (Matrix, Solver or Precond) of this LinOp.

Parameters

op	the source operator used to generate this LinOp

Exceptions

NotSupported	other is of incompatible type

See also

Matrix::generate(), Precond::generate(), Solver::generate()

apply() [1/2]

```
void apply (
              const DenseMatrix< E > * b,
              DenseMatrix< E > * x ) const  [virtual]
```

Apply a linear operator to a vector (or a sequence of vectors).

Performs the operation x = op(b), where op is this linear operator.

Parameters

b	the input vector on which the operator is applied

Parameters

x	the output vector where the result is stored

Exceptions

DimensionMismatch	the LinOp and the vectors are of incompatible sizes

apply() [2/2]

```
virtual void apply (
                E alpha,
                const DenseMatrix< E > * b,
                E beta,
                DenseMatrix< E > * x ) const  [virtual]
```

Perform the operation x = alpha ∗ op(b) + beta ∗ x.

Parameters

alpha	scaling of the result of op(b)
b	vector on which the operator is applied
beta	scaling of the input x
x	output vector

Exceptions

DimensionMismatch	the LinOp and the vectors are of incompatible sizes

get_info()

```
virtual std::shared_ptr<Info> get_info ( ) const  [pure virtual]
```

Get a reference to the LinOp::Info object containing information about the execution of this object's methods.

clone_type()

```
virtual std::unique_ptr<LinOp> clone_type ( ) const  [pure virtual]
```

Create a new 0x0 LinOp of the same type.

Returns

a LinOp object of the same type as this

get_ctx()

```
std::shared_ptr<const Ctx> get_ctx ( ) const
```

Get the Ctx of this object.

get_num_rows()

```
size_type get_num_rows ( ) const
```

Get the dimension of the codomain of this LinOp.

In other words, the number of rows of the coefficient matrix.

Returns

the dimension of the codomain

get_num_cols()

```
size_type get_num_cols ( ) const
```

Get the dimension of the domain of this LinOp.

In other words, the number of columns of the coefficient matrix.

Returns

the dimension of the codomain

get_num_nonzeros()

```
size_type get_num_nonzeros ( ) const
```

Get an upper bound on the number of nonzero values in the coefficient matrix of the operator.

This routine will get the number of nonzeros as seen by the library, and as used in computations, while the real number of nonzeros might be significantly lower than this value.

For example, for a DenseMatrix `A` it will always hold

```
A->get_num_nonzeros() == A->get_num_rows() * A->get_num_cols()
```

Returns

the number of nonzeros as seen by the library

clear()

```
void clear ( )  [virtual]
```

Transform the object into an empty LinOp.

4.2 LinOp< E >::Info

Inherited by Matrix< E >::Info, Precond< E >::Info, and Solver< E >::Info.

Public Member Functions

- virtual void clear ()
- const std::string & get_name () const
- double get_generation_runtime () const
- double get_application_runtime () const

4.2.1 Detailed Description

template<typename E>
class msparse::LinOp< E >::Info

The Info class encapsulates various information about the execution of the LinOp (generation time, application time, convergence data, etc.).

4.2.2 Member Function Documentation

clear()

```
void clear ( )  [virtual]
```

Reset all the fields to default values.

get_name()

```
const std::string& get_name ( ) const
```

Get the name of the associated LinOp object.

get_generation_runtime()

```
double get_generation_runtime ( ) const
```

Get the accumulated runtime for all calls to LinOp::generate() since the last call to Info::clear().

Returns

the accumulated generation time

get_application_runtime()

```
double get_application_runtime ( ) const
```

Get the accumulated runtime for all calls to LinOp::apply() since the last call to Info::clear().

Returns

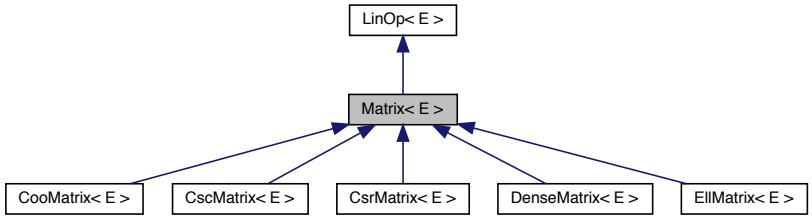
the accumulated application time

CHAPTER 5

Matrices

5.1 Matrix< E >

Inheritance diagram for Matrix< E >:



Public Member Functions

- virtual void generate_identity (size_type m, size_type n)
- virtual void fill (size_type m, size_type n, E val)
- virtual void rand_fill (size_type m, size_type n, E lo, E up)
- virtual void randn_fill (size_type m, size_type n, E mu, E sigma)
- virtual void read_from_mtx (const std::string &filename)
- virtual void write_to_mtx (const std::string &filename) const
- virtual convert< E >::to_real get_norm () const
- virtual void generate (std::shared_ptr< const LinOp< E >> op) override

5.1.1 Detailed Description

template<typename E>
class msparse::Matrix< E >

A Matrix is a linear operator which stores the coefficients of the operator explicitly.

MAGMA-sparse supports several matrix storage format, focusing primarily on sparse matrices (CsrMatrix, CscMatrix, CooMatrix, EllMatrix), which compress the data by explicitly storing only the non-zero coefficients of the matrix (depending on the format, a moderate amount of zero coefficients can also be stored to achieve better memory access patterns). There is also a DenseMatrix format, which explicitly stores all elements (in column major storage), and is primarily used to represent dense right-hand-side vectors and vectors of unknowns.

For constructing a matrix, a context (Ctx) has to be present. For the following examples, we assume a CpuCtx being stored in variable cpu and a GpuCtx being stored in gpu.

First, we show examples for the matrix construction.

```
// create a 0-by-0 "empty" matrix in CSR format
auto E = msparse::create<CsrMatrix<double>>(gpu);

// create a 10-by-10 identity matrix in CSR format
auto I = msparse::create_identity<CooMatrix<float>>(10, 10, cpu);

// create a 10-by-3 matrix in CSC format, with all coefficients set to 5.0
// this, obviously, should rather be considered as dense matrix, but it is
// possible one prefers the CSC storage format.
auto A = msparse::fill<CscMatrix<complex<float>>>(10, 3, 5.0f, gpu);

// create a 10-by-1 dense matrix (i.e. vector) with values randomly chosen
// from the range [-1.0,1.0)
auto B = msparse::rand_fill<DenseMatrix<double>>(10, 1, -1.0, 1.0, cpu);

// create a 4-by-5 dense matrix with values randomly chosen from a normal
// distribution with mean 0.0, and standard deviation 1.0.
auto C = msparse::randn_fill<DenseMatrix<double>>(4, 5, 0.0, 1.0, cpu);

// read a matrix from a matrix market file "D.mtx" and store it in CSR
auto D = msparse::read_from_mtx_to<CsrMatrix<double>>("D.mtx", gpu);

// create a GPU copy of C, stored in ELLPACK format
auto dC = msparse::copy_to<EllMatrix<double>>(C.get(), gpu);

// create a transpose of B
auto tB = msparse::trans_to<DenseMatrix<double>>(B.get(), cpu);

// create a conjugate transpose of A
auto cA = msparse::conj_trans_to<CscMatrix<complex<float>>>(A.get(), gpu);
```

In addition, MAGMA-sparse has several advanced matrix construction routines which could be useful in some situations:

```
std::shared_ptr<CooMatrix<float>> sI = std::move(I);
auto cI = msparse::generate<CsrMatrix<float>>(sI, cpu);

auto H = msparse::adapt<CsrMatrix<double>>(
            cpu, 10, 7, 15, row_ptr, col_idx, val);

auto x = msparse::adapt<DenseMatrix<double>>(
            cpu, 7, 1, val, 7);
```

The first part of the example constructs a copy of the identity matrix in CSR format using the generate() routine, which is part of the LinOp interface. First, the ownership of the original matrix `I` (from the previous example) must be converted from unique ownership to shared ownership, as the LinOp created via generate is allowed to keep a reference to the original object (in case of matrices, this will not be the case though). Next, the newly created shared version of the identity matrix `sI` is used to construct a copy in CSR format by calling msparse::generate(). Even though this example is not very useful for matrices, as the

same can be achieved much easier by calling msparse::copy_to(), it is important for solvers and preconditioners, as they represent more complex operators (inverse and approximate inverse), where the effect of this routine is differs from msparse::copy_to().

The next two examples demonstrate how existing sparse matrix structures from top-level applications can be adapted to use the MAGMA-sparse library. Concretely, a 10-by-7 matrix `H` with 15 nonzeros will be constructed using already existing `row pointer`, `column index` and `value` arrays. Then, a dense vector `x` with 7 elements is constructed from data already in memory. When constructing matrices from already existing data, there are several restrictions one has to be aware of:

- The msparse::adapt() routine only works for CsrMatrix and DenseMatrix.
- The arrays passed in will not be duplicated by MAGMA-sparse, so there is no additional data being allocated. As a consequence, the arrays have to be allocated on the device who's context was passed in to the routine. MAGMA-sparse will not deallocate the memory for the arrays when the matrix is destroyed, as memory management for those arrays is the responsibility of the user. Finally, the pointers to the arrays are expected to be valid during the lifetime of the constructed object.
- Even though matrices constructed using this routine provide the same interface as CsrMatrix or DenseMatrix, they cannot be used in some situations (and attempts to use them will throw an exeption). For example, it is not possible to modify the structure of such matrix by trying to use LinOp::copy_from() routine on the matrix. However, the matrix can be used as an argument to this routine in order to create an exclusive copy of the matrix for MAGMA-sparse.

Finally, once a matrix `A` is constructed, it can be used in matrix-vector product computations as in the following example:

```
auto ctx = A->get_ctx();
auto x = msparse::rand_fill<DenseMatrix<double>>(
            A->get_num_cols(), 1, -1.0, 1.0, ctx);
auto y = msparse::fill<DenseMatrix<double>>(A->get_num_rows(), 1, 0.0, ctx);
A->apply(x.get(), y.get());    // y = Ax
A->apply(2.0, x.get(), 1.0, y.get());  // y = 2.0 * Ax + 1.0 * y
```

Notice that there are two versions of the apply routine. A simpler version performs the standard linear operator application $y = Ax$, while the more complex one performs the BLAS GEMV operation $y = \alpha Ax + \beta y$.

Template Parameters

E	the type representing the precision

5.1.2 Member Function Documentation

generate_identity()

```
virtual void generate_identity (
              size_type m,
              size_type n )  [virtual]
```

Sets the Matrix to an m-by-n identity matrix.

Parameters

m	the number of rows of the matrix
n	the number of columns of the matrix

fill()

```
void fill (
              size_type m,
              size_type n,
              E val )  [virtual]
```

Set the Matrix to an m-by-n matrix with all coefficients set to val. With all elements set to val, this function creates a completely filled matrix (dense for the user), but it is still possible to choose a sparse data structure where the matrix is stored in, e.g. CSR, CSC, etc.

Parameters

m	the number of rows of the Matrix
n	the number of columns of the Matrix
val	coefficient value

rand_fill()

```
void rand_fill (
              size_type m,
              size_type n,
```

```
            E lo,
            E up )  [virtual]
```

Set the Matrix to an m-by-n matrix with the coefficients set to random numbers chosen uniformly from range [lo, up). With all elements set to a random val, this function creates a completely filled matrix (dense for the user), but it is still possible to choose a sparse data structure where the matrix is stored in, e.g. CSR, CSC, etc.

Parameters

m	the number of rows of the Matrix
n	the number of columns of the Matrix
lo	lower limit for the coefficients of the matrix
up	upper limit for the coefficients of the matrix

randn_fill()

```
void randn_fill (
            size_type m,
            size_type n,
            E mu,
            E sigma )  [virtual]
```

Set the Matrix to an m-by-n matrix with the coefficients set to random numbers chosen from a normal distribution with parameters mu and sigma. With all elements set to a random val, this function creates a completely filled matrix (dense for the user), but it is still possible to choose a sparse data structure where the matrix is stored in, e.g. CSR, CSC, etc.

Parameters

m	the number of rows of the Matrix
n	the number of columns of the Matrix
mu	the mean of the distribution
sigma	the standard deviation of the distribution

read_from_mtx()

```
virtual void read_from_mtx (
            const std::string & filename )  [virtual]
```

Read the Matrix coefficients from a file in matrix market format.

Parameters

filename	the file to read the matrix from

write_to_mtx()

```
virtual void write_to_mtx (
              const std::string & filename ) const  [virtual]
```

Write the Matrix to a file in matrix market format.

Parameters

filename	the file to write the matrix into

get_norm()

```
convert< E >::real get_norm ( ) const  [virtual]
```

Calculate the Frobenius norm of the Matrix.

Returns

the Frobenius norm of the matrix

generate()

```
virtual void generate (
              std::shared_ptr< const LinOp< E >> op )  [override], [virtual]
```

Generate a matrix from another LinOp.

This routine will try to extract the coefficients from another LinOp and store them into the Matrix.

Parameters

op	the LinOp from which the matrix will be generated

Exceptions

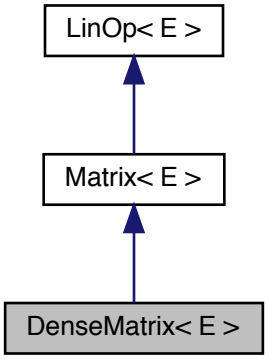
NotSupported	it is not possible to extract the coefficients

Remarks

The Matrix will not keep a reference to the LinOp after this method returns.

5.2 DenseMatrix< E >

Inheritance diagram for DenseMatrix< E >:



Public Member Functions

- E * get_val ()
- const E * get_val () const
- size_type get_ld () const
- void scal (E alpha)
- void axpy (E alpha, const DenseMatrix< E > *X)

- void gemm (E beta, E alpha, const DenseMatrix< E > ∗A, const DenseMatrix< E > ∗B)

Static Public Member Functions

- static std::unique_ptr< DenseMatrix > create (std::shared_ptr< const Ctx > ctx)
- static std::unique_ptr< DenseMatrix > adapt (std::shared_ptr< const Ctx > ctx, size_type nrows, size_type ncols, size_type ld, const E ∗val)

5.2.1 Detailed Description

template<typename E>
class msparse::DenseMatrix< E >

A DenseMatrix stores all of the coefficients explicitly.

The coefficients are stored in column-major storage, i.e. each column is stored consecutively in the memory. This format is suitable for matrices where the majority of the coefficients is nonzero, and in this case, it can have lower storage requirements than one of the compressed (sparse) formats.

Remarks

The format is also used throughout the library to store the dense vectors of unknowns, or the right hand side vectors.

Template Parameters

E	the type representing the precision

5.2.2 Member Function Documentation

create()

```
static std::unique_ptr<DenseMatrix> create (
              std::shared_ptr< const Ctx > ctx )  [static]
```

Create a new 0-by-0 DenseMatrix.

Parameters

ctx	the Ctx where the matrix will be stored

Returns

a unique pointer to the newly created matrix

adapt()

```
static std::unique_ptr<DenseMatrix> adapt (
              std::shared_ptr< const Ctx > ctx,
              size_type nrows,
              size_type ncols,
              size_type ld,
              const E * val )  [static]
```

Create a new DenseMatrix using the data already present on the ctx.

This routine is useful in case MAGMA-sparse is intended to be used from within the code that already uses its own matrix structures, as it allows to adapt these structures to the MAGMA-sparse interface without duplicating the data.

Nevertheless, the resulting Matrix object is not as general as MAGMA's native DenseMatrix and does not support some of the functionalities (like moves and copies to the object).

Parameters

ctx	the Ctx where the data is located, and where the resulting matrix will be stored
nrows	number of rows
ncols	number of columns
ld	the leading dimension of val
val	the array storing the coefficients of the matrix

get_val() [1/2]

```
E* get_val ( )
```

Get a raw pointer to the block of memory containing the coefficients of the matrix.

Returns

a pointer to the array storing the coefficients of the matrix

Remarks

the pointer becomes invalid if the owning DenseMatrix is destroyed.

get_val() [2/2]

```
const E* get_val ( ) const
```

Get a raw pointer to the block of memory containing the coefficients of the matrix.

Returns

a pointer to the array storing the coefficients of the matrix

Remarks

the pointer becomes invalid if the owning DenseMatrix is destroyed.

get_ld()

```
size_type get_ld ( ) const
```

Get the leading dimension of the matrix.

This is used to determine the starting position of each column of the matrix in memory, i.e. column i starts at position `i*get_ld()` of the array returned by `get_val()`.

Returns

the leading dimension of the matrix

scal()

```
void scal (
            E alpha )
```

Scale the matrix with the specified value.

Parameters

alpha	scaling factor

Remarks

This is the equivalent of BLAS xSCAL routine.

axpy()

```
void axpy (
              E alpha,
              const DenseMatrix< E > * X )
```

Add another (scaled) matrix to this matrix.

Parameters

alpha	scaling factor
X	the matrix to add

Remarks

This is the equivalent of BLAS xAXPY routine.

gemm()

```
void gemm (
              E beta,
              E alpha,
              const DenseMatrix< E > * A,
              const DenseMatrix< E > * B )
```

Create a liner combination of this matrix and a product of two other matrices.

Parameters

beta	scaling factor of the original matrix
alpha	scaling factor of the product

Parameters

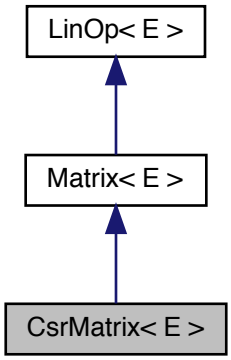
A	left factor of the product
B	right factor of the product

Remarks

This is the equivalent of the BLAS xGEMM routine.

5.3 CsrMatrix< E >

Inheritance diagram for CsrMatrix< E >:



Public Member Functions

- E * get_val ()
- const E * get_val () const
- idx_type * get_col_idx ()
- const idx_type * get_col_idx () const
- idx_type * get_row_ptr ()
- const idx_type * get_row_ptr () const

Static Public Member Functions

- static std::unique_ptr< CsrMatrix > create (std::shared_ptr< const Ctx > ctx)
- static std::unique_ptr< CsrMatrix > adapt (std::shared_ptr< const Ctx > ctx, size_type nrows, size_type ncols, size_type nnz, const idx_type *row_ptr, const idx_type *col_idx, const E *val)

5.3.1 Detailed Description

template<typename E>
class msparse::CsrMatrix< E >

A CsrMatrix is a sparse Matrix 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

E	the type representing the precision

5.3.2 Member Function Documentation

create()

```
static std::unique_ptr<CsrMatrix> create (
              std::shared_ptr< const Ctx > ctx )  [static]
```

Create a new 0-by-0 CsrMatrix.

Parameters

ctx	the Ctx where the matrix will be stored

Returns

a unique pointer to the newly created matrix

adapt()

```
static std::unique_ptr<CsrMatrix> adapt (
              std::shared_ptr< const Ctx > ctx,
              size_type nrows,
              size_type ncols,
              size_type nnz,
              const idx_type * row_ptr,
              const idx_type * col_idx,
              const E * val )  [static]
```

Create a new CsrMatrix using the data already present on the ctx.

This routine is useful in case MAGMA-sparse is intended to be used from within the code that already uses its own matrix structures, as it allows to adapt these structures to the MAGMA-sparse interface without duplicating the data.

Nevertheless, the resulting Matrix object is not as general as MAGMA-sparse's native CsrMatrix and does not support some of the functionalities (like moves and copies to the object).

Parameters

ctx	the Ctx where the data is located, and where the resulting matrix will be stored
nrows	number of rows
ncols	number of columns
nnz	number of non-zeors
row_ptr	the array storing the row pointers
col_idx	the array storing the column indices
val	the array storing the coefficients of the matrix

get_val() [1/2]

```
E* get_val ( )
```

Get a raw pointer to the block of memory containing the nonzero coefficients of the matrix.

The position of each nonzero can be determined by inspecting the arrays returned by methods get_row_ptr() and get_col_idx().

Returns

a pointer to the array storing the nonzero coefficients

Remarks

the pointer becomes invalid if the owning CsrMatrix is destroyed.

get_val() [2/2]

```
const E* get_val ( ) const
```

Get a raw pointer to the block of memory containing the nonzero coefficients of the matrix.

The position of each nonzero can be determined by inspecting the arrays returned by methods get_row_ptr() and get_col_idx().

Returns

a pointer to the array storing the nonzero coefficients

Remarks

the pointer becomes invalid if the owning CsrMatrix is destroyed.

get_col_idx() [1/2]

```
idx_type* get_col_idx ( )
```

Get a raw pointer to the block of memory containing the column indices of the matrix.

The i-th value of the array corresponds to the coefficient stored at position i in the coefficients array (obtained with get_val()).

Returns

a pointer to the array of column indexes.

Remarks

the pointer becomes invalid if the owning CsrMatrix is destroyed.

get_col_idx() [2/2]

```
const idx_type* get_col_idx ( ) const
```

Get a raw pointer to the block of memory containing the column indices of the matrix.

The i-th value of the array corresponds to the coefficient stored at position i in the coefficients array (obtained with get_val()).

Returns

a pointer to the array of column indexes.

Remarks

the pointer becomes invalid if the owning CsrMatrix is destroyed.

get_row_ptr() [1/2]

```
idx_type* get_row_ptr ( )
```

Get a raw pointer to the block of memory containing the row pointers.

The value at position i indicates the starting location of the i-th row of the matrix in coefficient and column index arrays (obtained from get_val() and get_col_idx()).

Returns

a pointer to the row pointer array

Remarks

the pointer becomes invalid if the owning CsrMatrix is destroyed.

get_row_ptr() [2/2]

```
const idx_type* get_row_ptr ( ) const
```

Get a raw pointer to the block of memory containing the row pointers.

The value at position i indicates the starting location of the i-th row of the matrix in coefficient and column index arrays (obtained from get_val() and get_col_idx()).

Returns

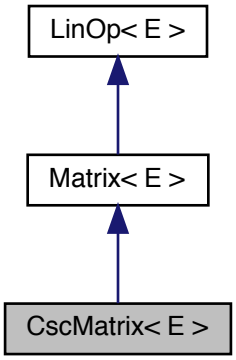
a pointer to the row pointer array

Remarks

the pointer becomes invalid if the owning CsrMatrix is destroyed.

5.4 CscMatrix< E >

Inheritance diagram for CscMatrix< E >:



Public Member Functions

- E * get_val ()
- const E * get_val () const
- idx_type * get_row_idx ()
- const idx_type * get_row_idx () const
- idx_type * get_col_ptr ()
- const idx_type * get_col_ptr () const

Static Public Member Functions

- static std::unique_ptr< CscMatrix > create (std::shared_ptr< const Ctx > ctx)

5.4.1 Detailed Description

template<typename E>
class msparse::CscMatrix< E >

A CscMatrix is a sparse Matrix which stores only the nonzero coefficients by compressing each column of the matrix (compressed sparse column format).

The nonzero elements are stored in a 1D array column-wise, and accompanied with a *column pointer* array which stores the starting index of each column. An additional *row index* array is used to identify the row of each nonzero element.

Template Parameters

E	the type representing the precision

5.4.2 Member Function Documentation

create()

```
static std::unique_ptr<CscMatrix> create (
              std::shared_ptr< const Ctx > ctx )  [static]
```

Create a new 0-by-0 CscMatrix.

Parameters

ctx	the Ctx where the matrix will be stored

Returns

a unique pointer to the newly created matrix

get_val() [1/2]

```
E* get_val ( )
```

Get a raw pointer to the block of memory containing the nonzero coefficients of the matrix.

The position of each nonzero can be determined by inspecting the arrays returned by methods get_col_ptr() and get_row_idx().

Returns

a pointer to the array storing the nonzero coefficients

Remarks

the pointer becomes invalid if the owning CscMatrix is destroyed.

get_val() [2/2]

```
const E* get_val ( ) const
```

Get a raw pointer to the block of memory containing the nonzero coefficients of the matrix.

The position of each nonzero can be determined by inspecting the arrays returned by methods get_col_ptr() and get_row_idx().

Returns

a pointer to the array storing the nonzero coefficients

Remarks

the pointer becomes invalid if the owning CscMatrix is destroyed.

get_row_idx() [1/2]

```
idx_type* get_row_idx ( )
```

Get a raw pointer to the block of memory containing the row indices of the matrix.

The i-th value of the array corresponds to the coefficient stored at position i in the coefficients array (obtained with get_val()).

Returns

a pointer to the array of row indexes.

Remarks

the pointer becomes invalid if the owning CscMatrix is destroyed.

get_row_idx() [2/2]

```
const idx_type* get_row_idx ( ) const
```

Get a raw pointer to the block of memory containing the row indices of the matrix.

The i-th value of the array corresponds to the coefficient stored at position i in the coefficients array (obtained with get_val()).

Returns

a pointer to the array of row indexes.

Remarks

the pointer becomes invalid if the owning CscMatrix is destroyed.

get_col_ptr() [1/2]

```
idx_type* get_col_ptr ( )
```

Get a raw pointer to the block of memory containing the column pointers.

The value at position i indicates the starting location of the i-th column of the matrix in coefficient and row index arrays (obtained from get_val() and get_row_idx()).

Returns

a pointer to the column pointer array

Remarks

the pointer becomes invalid if the owning CscMatrix is destroyed.

get_col_ptr() [2/2]

```
const idx_type* get_col_ptr ( ) const
```

Get a raw pointer to the block of memory containing the column pointers.

The value at position i indicates the starting location of the i-th column of the matrix in coefficient and row index arrays (obtained from get_val() and get_row_idx()).

Returns

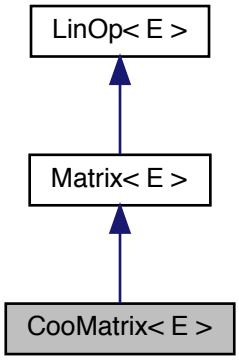
a pointer to the column pointer array

Remarks

the pointer becomes invalid if the owning CscMatrix is destroyed.

5.5 CooMatrix< E >

Inheritance diagram for CooMatrix< E >:



Public Member Functions

- E * get_val ()
- const E * get_val () const
- idx_type * get_row_idx ()
- const idx_type * get_row_idx () const
- idx_type * get_col_idx ()
- const idx_type * get_col_idx () const

Static Public Member Functions

- static std::unique_ptr< CooMatrix > create (std::shared_ptr< const Ctx > ctx)

5.5.1 Detailed Description

template<typename E>
class msparse::CooMatrix< E >

A CooMatrix is a sparse Matrix which stores only the nonzero coefficients by compressing the entire 2D coefficient table (coordinate matrix format).

The nonzero elements are stored in a 1D array row-wise. Additional *row index* and *column index* arrays are used to identify both the row and the column of each nonzero element.

Template Parameters

E	the type representing the precision

5.5.2 Member Function Documentation

create()

```
static std::unique_ptr<CooMatrix> create (
              std::shared_ptr< const Ctx > ctx )  [static]
```

Create a new 0-by-0 CsrMatrix.

Parameters

ctx	the Ctx where the matrix will be stored

Returns

a unique pointer to the newly created matrix

get_val() [1/2]

```
E* get_val ( )
```

Get a raw pointer to the block of memory containing the nonzero coefficients of the matrix.

The position of each nonzero can be determined by inspecting the arrays returned by methods get_row_idx() and get_col_idx().

Returns

a pointer to the array storing the nonzero coefficients

Remarks

the pointer becomes invalid if the owning CooMatrix is destroyed.

get_val() [2/2]

```
const E* get_val ( ) const
```

Get a raw pointer to the block of memory containing the nonzero coefficients of the matrix.

The position of each nonzero can be determined by inspecting the arrays returned by methods get_row_idx() and get_col_idx().

Returns

a pointer to the array storing the nonzero coefficients

Remarks

the pointer becomes invalid if the owning CooMatrix is destroyed.

get_row_idx() [1/2]

```
idx_type* get_row_idx ( )
```

Get a raw pointer to the block of memory containing the row indices of the matrix.

The i-th value of the array corresponds to the coefficient stored at position i in the coefficients array (obtained with get_val()).

Returns

a pointer to the array of row indexes.

Remarks

the pointer becomes invalid if the owning CooMatrix is destroyed.

get_row_idx() [2/2]

```
const idx_type* get_row_idx ( ) const
```

Get a raw pointer to the block of memory containing the row indices of the matrix.

The i-th value of the array corresponds to the coefficient stored at position i in the coefficients array (obtained with get_val()).

Returns

a pointer to the array of row indexes.

Remarks

the pointer becomes invalid if the owning CooMatrix is destroyed.

get_col_idx() [1/2]

```
idx_type* get_col_idx ( )
```

Get a raw pointer to the block of memory containing the column indices of the matrix.

The i-th value of the array corresponds to the coefficient stored at position i in the coefficients array (obtained with get_val()).

Returns

a pointer to the array of column indexes.

Remarks

the pointer becomes invalid if the owning CooMatrix is destroyed.

get_col_idx() [2/2]

```
const idx_type* get_col_idx ( ) const
```

Get a raw pointer to the block of memory containing the column indices of the matrix.

The i-th value of the array corresponds to the coefficient stored at position i in the coefficients array (obtained with get_val()).

Returns

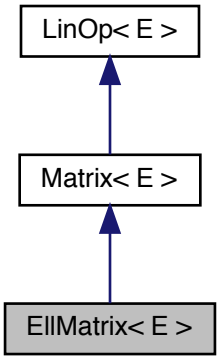
a pointer to the array of column indexes.

Remarks

the pointer becomes invalid if the owning CooMatrix is destroyed.

5.6 EllMatrix< E >

Inheritance diagram for EllMatrix< E >:



Public Member Functions

- E * get_val ()
- const E * get_val () const
- idx_type * get_col_idx ()
- const idx_type * get_col_idx () const
- size_type get_ld () const

Static Public Member Functions

- static std::unique_ptr< EllMatrix > create (std::shared_ptr< const Ctx > ctx)

5.6.1 Detailed Description

template<typename E>
class msparse::EllMatrix< E >

An EllMatrix is a sparse Matrix which stores only the nonzero coefficients by compressing all rows of the matrix to the same length (ELLPACK matrix format).

The original m-by-n matrix is compressed to an m-by-mnr matrix (where mnr is the maximal number of nonzeros among all rows), by storing only the nonzero elements of each row, and optionally padding with zeros, in order for all rows to be of length mnr.

An additional m-by-mnr *column index* matrix is used to identify the column of each nonzero element.

Both the coefficient, as well as the column index matrices are stored in dense format, in column-major storage.

Remarks

> This format is suitable for matrices which have roughly the same amount of nonzeros per row (with maybe a few rows having less, but not more than the other rows). Otherwise, the overhead of padding the rows with zeros can be significant both in terms of storage, as well as in terms of computational cost.
> The value of mnr for an EllMatrix can be obtained by dividing the result of LinOp::get_num_nonzeros() with LinOp::get_num_rows().

Template Parameters

E	the type representing the precision

5.6.2 Member Function Documentation

create()

```
static std::unique_ptr<EllMatrix> create (
              std::shared_ptr< const Ctx > ctx )  [static]
```

Create a new 0-by-0 EllMatrix.

Parameters

ctx	the Ctx where the matrix will be stored

Returns

a unique pointer to the newly created matrix

get_val() [1/2]

```
E* get_val ( )
```

Get a raw pointer to the block of memory containing the nonzero coefficients of the matrix.

The position of each nonzero can be determined by inspecting the value of get_ld() and the array returned by get_col_idx().

Returns

a pointer to the array storing the nonzero coefficients

Remarks

the pointer becomes invalid if the owning EllMatrix is destroyed.

get_val() [2/2]

```
const E* get_val ( ) const
```

Get a raw pointer to the block of memory containing the nonzero coefficients of the matrix.

The position of each nonzero can be determined by inspecting the value of get_ld() and the array returned by get_col_idx().

Returns

a pointer to the array storing the nonzero coefficients

Remarks

the pointer becomes invalid if the owning EllMatrix is destroyed.

get_col_idx() [1/2]

```
idx_type* get_col_idx ( )
```

Get a raw pointer to the block of memory containing the column indices of the matrix.

The i-th value of the array corresponds to the coefficient stored at position i in the coefficients array (obtained with get_val()).

Returns

a pointer to the array of column indexes.

Remarks

the pointer becomes invalid if the owning EllMatrix is destroyed.

get_col_idx() [2/2]

```
const idx_type* get_col_idx ( ) const
```

Get a raw pointer to the block of memory containing the column indices of the matrix.

The i-th value of the array corresponds to the coefficient stored at position i in the coefficients array (obtained with get_val()).

Returns

a pointer to the array of column indexes.

Remarks

the pointer becomes invalid if the owning EllMatrix is destroyed.

get_ld()

```
size_type get_ld ( ) const
```

Get the leading dimension of the compressed matrix.

This is used to determine the starting position of each column of the compressed matrix in memory, i.e. column j starts at position `j*get_ld()` of the array returned by `get_val()`.

Thus, the j-th nonzero element of row i is located at position `i + j*get_ld()` of the coefficients array.

Returns

the leading dimension of the matrix

CHAPTER 6

Top-level routines

6.1 Initialization routines

Functions

- template<typename ObjType >
  std::unique_ptr< ObjType > clone (const ObjType *obj)
- template<typename ObjType , typename... CreateArgs>
  auto create (CreateArgs... args) -> decltype(ObjType::create(args...))
- template<typename ObjType , typename E , typename... CreateArgs>
  auto copy_to (const LinOp< E > *op, CreateArgs... args) -> decltype(ObjType::create(args...))
- template<typename ObjType , typename E , typename... CreateArgs>
  auto copy_to (std::unique_ptr< const LinOp< E >> op, CreateArgs... args) -> decltype(ObjType::create(args...))
- template<typename ObjType , typename E , typename... CreateArgs>
  auto trans_to (const LinOp< E > *op, CreateArgs... args) -> decltype(ObjType::create(args...))
- template<typename ObjType , typename E , typename... CreateArgs>
  auto conj_trans_to (const LinOp< E > *op, CreateArgs... args) -> decltype(ObjType::create(args...))
- template<typename ObjType , typename OpType , typename... CreateArgs>
  auto generate (std::shared_ptr< OpType > op, CreateArgs... args) -> decltype(ObjType::create(args...))
- template<typename E >
  std::unique_ptr< DenseMatrix< E > > apply (const LinOp< E > *op, const DenseMatrix< e > *x)
- template<typename ObjType , typename E , typename... CreateArgs>
  auto create_identity (size_type nrows, size_type ncols, CreateArgs... args) -> decltype(ObjType::create(args...))
- template<typename ObjType , typename E , typename... CreateArgs>
  auto fill (size_type nrows, size_type ncols, E val, CreateArgs... args) -> decltype(ObjType::create(args...))
- template<typename ObjType , typename E , typename... CreateArgs>
  auto rand_fill (size_type nrows, size_type ncols, E lo, E up, CreateArgs... args) -> decltype(ObjType::create(args...))
- template<typename ObjType , typename E , typename... CreateArgs>
  auto randn_fill (size_type nrows, size_type ncols, E mu, E sigma, CreateArgs... args) -> decltype(ObjType::create(args...))
- template<typename ObjType , typename... CreateArgs>
  auto read_from_mtx_to (const char *fname, CreateArgs... args) -> decltype(ObjType::create(args...))

6.1.1 Detailed Description

Top-level routines described in this group provide a ”facade” to simplify common tasks when creating MAGMA-sparse objects.

The process of creation of any LinOp object in MAGMA-sparse consists of two steps:

- Determining the type of the object (e.g. matrix format, solver type) and passing the parameters for the type (e.g. shadow space dimension for the IDR solver).
- Initializing the object with useful data (e.g. reading the matrix coefficients from a file, generating a preconditioner from a system matrix, setting the system matrix for a solver).

To provide a high degree of flexibility, MAGMA-sparse separates these steps into two separate routines. For example, it might be useful to determine the type and parameters of a LinOp in one place in the code, and then call a routine which will fill the object's data independent of the actual type of the object. Thus, a complete creation of a LinOp (in this example a CsrMatrix) looks like this:

```
auto A = msparse::CsrMatrix<double>::create(gpu); // empty 0-by-0 CSR
       matrix
A->read_from_mtx("A.mtx"); // fill the data from file
```

In this example, the second line could be easily moved to a separate routine which could determine from which file to read, without concerning itself with the actual format of the matrix.

However, most simple scenarios do not need such a high level of flexibility and the previous method of creating a matrix can reduce code readability by being unnecessarily verbose, as well as cause subtle bugs if the object is used while it is still in its empty, 0-by-0 state. To avoid these problems MAGMA-sparse provides a set of initializer routines which combine these two steps by first calling create on the specified object type, and immediately initializing it with data.

Using the initializer routines, the above example could be written as a single library call:

```
auto A = msparse::read_from_mtx_to<CsrMatrix<double>>("A.mtx", gpu);
```

There are many more initializer routines listed below and most of them work in the same manner:

- they initialize the object provided by the template parameter using a static create() method on the template parameter;
- then fill the returned object with data using a initalizer-specific method on that object;
- and finally return the created object.

The parameters for the initializer routines always commence with the parameters which will be passed to the initializer-specific method, and end with the parameters for the object's create method. The documentation for each initializer routine contains information about

which method will be called. Thus, to find out the effect of such routine on a specific object, one should read the object's documentation for this method.

The exceptions to the above statements are the clone() and the apply() routines, which, even though they can be categorized as initializers, do not follow these rules.

6.1.2 Function Documentation

clone()

```
std::unique_ptr<ObjType> msparse::clone (
              const ObjType * obj )
```

Create an exact clone of the passed object.

Template Parameters

ObjType	the type of object to clone

Parameters

obj	the object to clone

Returns

the clone of the object

create()

```
auto msparse::create (
              CreateArgs...  args ) -> decltype(ObjType::create(args...))
```

Create a new ObjType object, by calling ObjType::create().

This is a convenience function which allows to call the default create method using the same syntax as with the other initializer routines

Template Parameters

ObjType	the type of the object to create

Template Parameters

CreateArgs	the argument pack passed to ObjType::create()

Parameters

args	the arguments passed to ObjType::create()

Returns

a new object returned by ObjType::create()

copy_to() [1/2]

```
auto msparse::copy_to (
              const LinOp< E > * op,
              CreateArgs...  args ) -> decltype(ObjType::create(args...))
```

Create a new LinOp object by making a copy of another LinOp.

The routine first creates a new operator by calling ObjType::create(), and then uses ObjType::copy_from() to create a copy of op.

Template Parameters

ObjType	the type of operator to create
E	the precision of the LinOp
CreateArgs	the argument pack passed to ObjType::create()

Parameters

op	the LinOp op to copy
args	the arguments passed to ObjType::create()

Returns

a new ObjType operator

copy_to() [2/2]

```
auto msparse::copy_to (
              std::unique_ptr< const LinOp< E >> op,
              CreateArgs...  args ) -> decltype(ObjType::create(args...))
```

Create a new LinOp object by moving another LinOp.

This is the move version of the copy_to() initializer. The routine first creates a new operator by calling ObjType::create(), and then uses ObjType::copy_from() to move op.

Template Parameters

ObjType	the type of operator to create
E	the precision of the LinOp
CreateArgs	the argument pack passed to ObjType::create()

Parameters

op	the LinOp op to copy
args	the arguments passed to ObjType::create()

Returns

a new ObjType operator

trans_to()

```
auto msparse::trans_to (
              const LinOp< E > * op,
              CreateArgs...  args ) -> decltype(ObjType::create(args...))
```

Create a new LinOp object by transposing another LinOp.

The routine first creates a new operator by calling ObjType::create(), and then uses ObjType::make_trans_of() to create a transpose of op.

Template Parameters

ObjType	the type of operator to create
E	the precision of the LinOp
CreateArgs	the argument pack passed to ObjType::create()

Parameters

op	the LinOp to transpose
args	the arguments passed to ObjType::create()

Returns

the transpose of op, of type ObjType

conj_trans_to()

```
auto msparse::conj_trans_to (
              const LinOp< E > * op,
              CreateArgs...  args ) -> decltype(ObjType::create(args...))
```

Create a new LinOp object by conjugate transposing another LinOp.

The routine first creates a new operator by calling ObjType::create(), and then uses ObjType::make_conj_trans_of() to create a conjugate transpose of op.

Template Parameters

ObjType	the type of operator to create
E	the precision of the LinOp
CreateArgs	the argument pack passed to ObjType::create()

Parameters

op	the LinOp to transpose
args	the arguments passed to ObjType::create()

Returns

the transpose of op, of type ObjType

generate()

```
auto msparse::generate (
```

```
std::shared_ptr< OpType > op,
CreateArgs...  args ) -> decltype(ObjType::create(args...))
```

Generate a new LinOp from another LinOp.

This routine combines both the LinOp creation and the call to LinOp::generate() into a single routine in order to avoid handling of non-generated (empty) LinOp objects.

Template Parameters

ObjType	the type of LinOp to create
CreateArgs	the argument pack passed to ObjType::create()

Parameters

op	the LinOp to generate the object from
args	the arguments passed to ObjType::create()

Returns

a new ObjType operator

apply()

```
std::unique_ptr<DenseMatrix<E> > msparse::apply (
              const LinOp< E > * op,
              const DenseMatrix< e > * x )
```

Create a new DenseMatrix by applying a LinOp to another DenseMatrix.

The resulting DenseMatrix will be allocated on the same Ctx as the LinOp.

Template Parameters

E	the precision of the objects

Parameters

op	the operator to apply
x	the vector on which op is applied

Returns

the resulting vector $y = op(x)$

create_identity()

```
auto msparse::create_identity (
              size_type nrows,
              size_type ncols,
              CreateArgs...  args ) -> decltype(ObjType::create(args...))
```

Create an nrows-by-ncols identity Matrix.

Template Parameters

ObjType	the type of Matrix to create
CreateArgs	the argument pack passed to ObjType::create()

Parameters

nrows	the number of rows of the Matrix
ncols	the number of columns of the Matrix
args	the arguments passed to ObjType::create()

Returns

a new ObjType matrix

fill()

```
auto msparse::fill (
              size_type nrows,
              size_type ncols,
              E val,
              CreateArgs...  args ) -> decltype(ObjType::create(args...))
```

Create an nrows-by-ncols Matrix with all coefficients set to val.

Template Parameters

ObjType	the type of Matrix to create
E	the precision of val
CreateArgs	the argument pack passed to ObjType::create()

Parameters

nrows	the number of rows of the Matrix
ncols	the number of columns of the Matrix
val	coefficient value
args	the arguments passed to ObjType::create()

Returns

a new ObjType matrix

rand_fill()

```
auto msparse::rand_fill (
              size_type nrows,
              size_type ncols,
              E lo,
              E up,
              CreateArgs...  args ) -> decltype(ObjType::create(args...))
```

Create an nrows-by-ncols Matrix with the coefficients set to random numbers chosen uniformly from range [lo, up].

Template Parameters

ObjType	the type of Matrix to create
E	the precision of lo and up
CreateArgs	the argument pack passed to ObjType::create()

Parameters

nrows	the number of rows of the Matrix
ncols	the number of columns of the Matrix

Parameters

lo	lower limit for the coefficients of the matrix
up	upper limit for the coefficients of the matrix
args	the arguments passed to ObjType::create()

Returns

a new ObjType matrix

randn_fill()

```
auto msparse::randn_fill (
              size_type nrows,
              size_type ncols,
              E mu,
              E sigma,
              CreateArgs...  args ) -> decltype(ObjType::create(args...))
```

Create an nrows-by-ncols Matrix with the coefficients set to random numbers chosen from a normal distribution with parameters mu and sigma.

Template Parameters

ObjType	the type of Matrix to create
E	the precision of mu and sigma
CreateArgs	the argument pack passed to ObjType::create()

Parameters

nrows	the number of rows of the Matrix
ncols	the number of columns of the Matrix
mu	the mean of the distribution
sigma	the standard deviation of the distribution
args	the arguments passed to ObjType::create()

Returns

a new ObjType matrix

read_from_mtx_to()

```
auto msparse::read_from_mtx_to (
              const char * fname,
              CreateArgs...  args ) -> decltype(ObjType::create(args...))
```

Create a new Matrix object by reading it from a matrix market file.

This routine both creates, and fills the matrix with data, by calling Matrix::read_from_mtx(). This routine should be preferred over creating an empty matrix and the reading the data from file manually, as it avoids handling of empty objects.

Template Parameters

ObjType	the type of Matrix to create
CreateArgs	the argument pack passed to ObjType::create()

Parameters

fname	the file to read the matrix from
args	the arguments passed to ObjType::create()

Returns

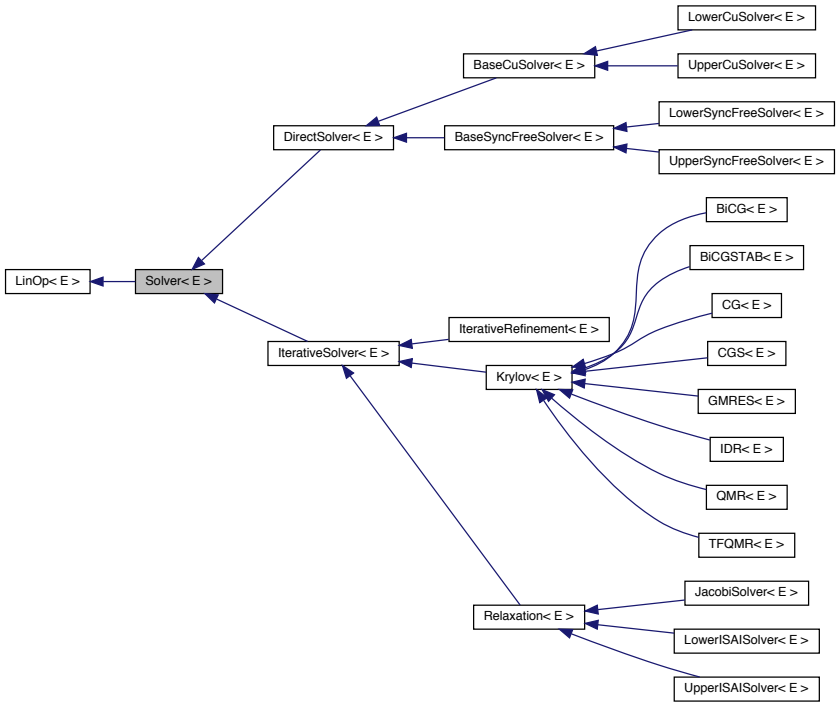
a new ObjType matrix

CHAPTER 7

Solvers

7.1 Solver< E >

Inheritance diagram for Solver< E >:



7.1.1 Detailed Description

template<typename E>
class msparse::Solver< E >

A Solver is a class which encapsulates the core functionality of MAGMA-sparse: the solution of linear systems. Viewed as a linear operator (LinOp), this class represents a linear operator A^{-1}, as its application $x = A^{-1}b$ consists of solving a linear system $Ax = b$. Various Solver subclasses implement different methods of solving linear systems, some of them direct (triangular) solvers (LowerCuSolver, UpperCuSolver, LowerSyncFreeSolver, UpperSyncFreeSolver), while others iterative, relaxation-based (JacobiSolver, LowerISAISolver, UpperISAISolver) or Krylov subspace-based (CG, CGS, BiCG, BiCGSTAB, GMRES, IDR, QMR, TFQMR) solvers.

Typically, a solver is constructed using the msparse::generate() initializer routine and spec-

ifying the solver type and the system matrix. To construct a conjugate gradient solver for example, one would write the following code (assuming a GpuCtx is already stored in variable gpu).

```
std::shared_ptr<CsrMatrix<double>> A =
        msparse::read_from_mtx_to<CsrMatrix<double>>("A.mtx", gpu);
auto cg = msparse::generate<CG<double>>(A, gpu);
```

The above code reads a matrix from a matrix market file 'A.mtx' and stores the matrix in CSR format on the GPU. Then, it constructs a CG solver with this matrix, and specifies that the solver should also run on the GPU. Notice that the generate method requires shared ownership of the matrix. Thus, the solver will try to avoid copying the content of the matrix and use the original CsrMatrix object (nevertheless a copy will be created if the matrix is not present in the same context as the solver). This implies that any changes to the matrix could (but are not required to!) be reflected on the solver. Thus, changing the matrix from outside the solver, and then using the solver to solve a linear system might result in undefined behaviour. In addition to avoiding all data-modifying operations on the matrix, there are two more ways of ensuring this problem does not occur. If the matrix is not needed elsewhere, instead of constructing a shared pointer to the matrix, a unique pointer returned by msparse::read_from_mtx_to() can be directly moved to the solver (of course, this renders the original matrix pointer unusable):

```
auto A = msparse::read_from_mtx_to<CsrMatrix<double>>("A.mtx", gpu);
auto cg = msparse::generate<CG<double>>(std::move(A), gpu);
// A is moved to the solver, the variable cannot be used anymore
```

Alternatively, one could explicitly provide the solver with its own dedicated copy of the matrix:

```
auto A = msparse::read_from_mtx_to<CsrMatrix<double>>("A.mtx", gpu);
auto cg = msparse::generate<CG<double>>(msparse::clone(A.get()), gpu);
// cg received a copy, so A can be modified freely
```

Once a solver has been constructed it can be used to solve linear systems using the LinOp::apply() method. The following example combines several MAGMA-sparse routines to read a matrix from file, save it in CSR format, use the conjugate gradient method to solve a linear system with this matrix and a randomly generated right-hand-side vector on the GPU, and finally store the result into a matrix market file.

```
#include <utility>
#include <msparse.h>


int main()
{
    // create a GPU context
    auto cpu = msparse::create<CpuCtx>();
```

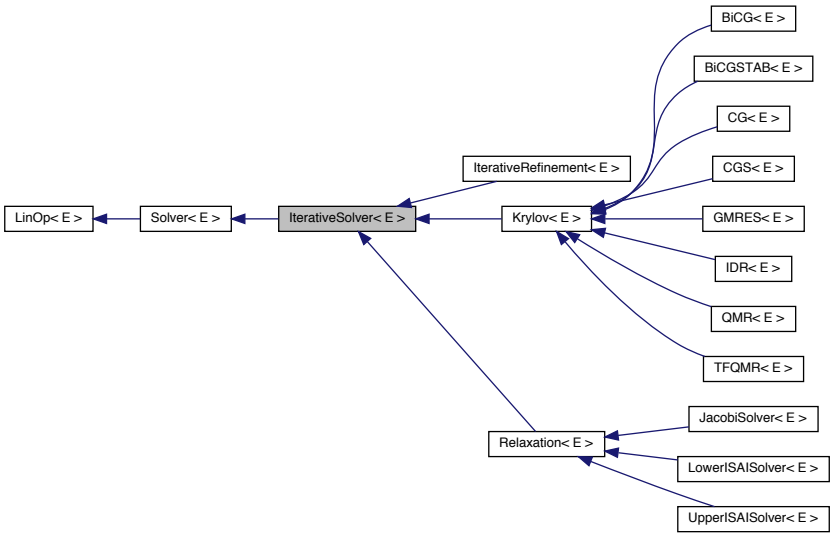
```
    auto gpu = msparse::create<GpuCtx>(0, cpu);  // use GPU with ID 0

    // read the matrix from file
    auto A = msparse::read_from_mtx_to<CsrMatrix<double>>("A.mtx", gpu);
    int n = A->get_num_rows();

    // initialize the solver
    auto cg = msparse::generate<CG<double>>(std::move(A), gpu);

    // generate a random rhs vector
    auto b = msparse::rand_fill<DenseMatrix<double>>(
                n, 1, -1.0, 1.0, gpu);

    // set starting guess to 0
    auto x = msparse::apply(cg.get(), b.get());

    // save the result
    x->write_to_mtx('x.mtx');
    return 0;
}
```

Template Parameters

E	the precision the data is stored in

7.2 IterativeSolver< E >

Inheritance diagram for IterativeSolver< E >:



Public Member Functions

- void set_iteration_limit (size_type iteration_limit)
- size_type get_iteration_limit () const
- void set_runtime_limit (double runtime_limit)
- double get_runtime_limit () const
- void set_absolute_stop_threshold (real abs_stop)
- convert< E >::to_real get_absolute_stop_threshold ()
- void set_relative_stop_threshold (real rel_stop)
- convert< e >::to_real get_relative_stop_threshold ()
- void set_verbose_flags (int64 verbose_flags)
- int64 get_verbose_flags () const
- void set_verbose_freq (int32 verbose_freq)
- int64 get_verbose_flags () const

7.2.1 Detailed Description

template<typename E>
class msparse::IterativeSolver< E >

This class represents iterative solvers. This can be Krylov methods, relaxation methods, iterative refinement.

Template Parameters

E	the type representing the precision E

7.2.2 Member Function Documentation

set_iteration_limit()

```
void set_iteration_limit (
              size_type iteration_limit )
```

Set the maximum number of iterations IterativeSolver's apply() method should perform.

Parameters

iteration_limit	the maximum number of iterations

get_iteration_limit()

```
size_type get_iteration_limit ( ) const
```

Get the IterativeSolver's iteration limit.

Returns

iteration limit

See also

set_iteration_limit().

set_runtime_limit()

```
void set_runtime_limit (
              double runtime_limit )
```

Set the maximum allowed time for IterativeSolver's apply() method. If this limit is reached during computation, the solver will not advance to the next iteration.

Parameters

runtime_limit	maximum allowed time in seconds

get_runtime_limit()

```
double get_runtime_limit ( ) const
```

Get the IterativeSolver's runtime limit.

Returns

runtime limit

See also

set_runtime_limit().

set_absolute_stop_threshold()

```
void set_absolute_stop_threshold (
              real abs_stop )
```

Set the absolute residual stopping criterion. If the norm of the absolute residual norm drops below this threshold, the iterative solver completes successfully.

Parameters

abs_stop	absolute residual stopping criterion

get_absolute_stop_threshold()

```
convert<E>::to_real get_absolute_stop_threshold ( )
```

Get the IterativeSolver's absolute stopping criterion.

Returns

absolute stopping criterion

See also

set_absolute_stop_threshold().

set_relative_stop_threshold()

```
void set_relative_stop_threshold (
            real rel_stop )
```

Set the relative residual stopping criterion. If the norm of the relative residual norm drops below this threshold, the iterative solver completes successfully.

Parameters

rel_stop	relative residual stopping criterion

get_relative_stop_threshold()

```
convert<e>::to_real get_relative_stop_threshold ( )
```

Get the IterativeSolver's relative stopping criterion.

Returns

relative stopping criterion

See also

set_relative_stop_threshold().

set_verbose_flags()

```
void set_verbose_flags (
              int64 verbose_flags )
```

Set the verbose flags. The flags are stored as bits of a 64-bit integer: 0 is off; 1 is on.

Parameters

verbose_flags	verbose flags encoded in bits

get_verbose_flags() [1/2]

```
int64 get_verbose_flags ( ) const
```

Get the IterativeSolver's verbosity flags.

Returns

verbose flags

See also

set_verbose_flags().

set_verbose_freq()

```
void set_verbose_freq (
              int32 verbose_freq )
```

Set the verbose frequency. This is the frequency at which the distinct metrics are recorded. Precisely: verbose_freq = 0 : never verbose_freq = 1 : every iteration verbose_freq = 2 : every second iteration ...

Parameters

verbose_freq	verbose frequency

get_verbose_flags() [2/2]

```
int64 get_verbose_flags ( ) const
```

Get the IterativeSolver's verbosity frequency

Returns

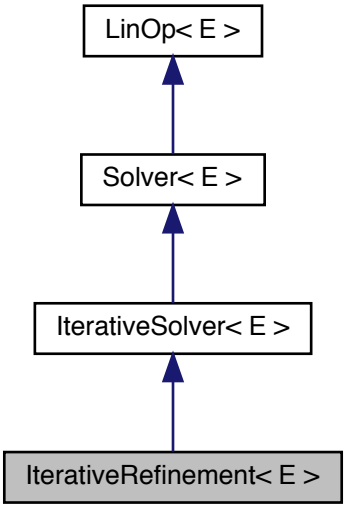
verbose frequency

See also

set_verbose_frequency().

7.3 IterativeRefinement< E >

Inheritance diagram for IterativeRefinement< E >:



7.3.1 Detailed Description

template<typename E>
class msparse::IterativeRefinement< E >

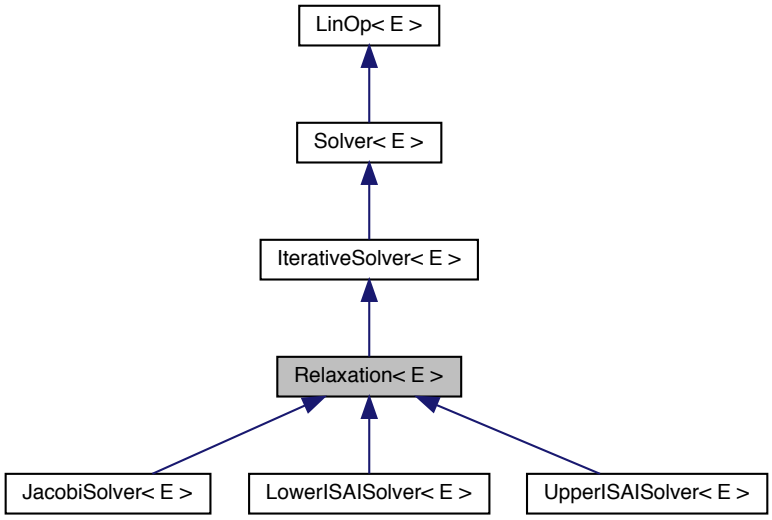
This class represents the Iterative Refinement Solver.

Template Parameters

E	the type representing the precision E

7.4 Relaxation< E >

Inheritance diagram for Relaxation< E >:



7.4.1 Detailed Description

template<typename E>
class msparse::Relaxation< E >

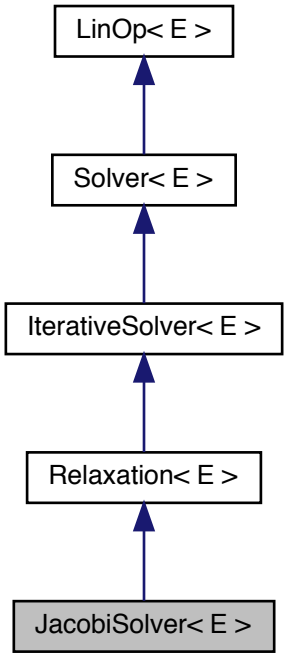
This class represents relaxation-based iterative solvers like, e.g., the Jacobi stationary iterations.

Template Parameters

E	the type representing the precision E

7.5 JacobiSolver< E >

Inheritance diagram for JacobiSolver< E >:



Public Member Functions

- virtual void set_blocksize (int32 block_size)
- int32 get_blocksize ()

Static Public Member Functions

- static std::unique_ptr< JacobiSolver > create (std::shared_ptr< const Ctx > ctx, int32 block_size=1)

7.5.1 Detailed Description

template<typename E>
class msparse::JacobiSolver< E >

This class represents the Jacobi iterative solver, flexible in terms of the block size. The stationary iterations are defined as: x += D^(-1)(b-Ax) where D has diagonal or block-diagonal structure. In the block-diagonal case, the blocks are determined using supervariable agglomeration algorithm, and the block-size parameter is an upper bound. By default, the block size is set to 1 (scalar Jacobi).

Template Parameters

E	the type representing the precision E

7.5.2 Member Function Documentation

create()

```
static std::unique_ptr<JacobiSolver> create (
              std::shared_ptr< const Ctx > ctx,
              int32 block_size = 1 )  [static]
```

Create a JacobiSolver solver for a 0-by-0 matrix.

Parameters

ctx	the Ctx determining location of the new object
block_size	the upper bound for the Jacobi blocks

Returns

a unique pointer to the newly created object

set_blocksize()

```
virtual void set_blocksize (
              int32 block_size )  [virtual]
```

Set the upper bound for the size of the Jacobi blocks. The actual size of the distinct blocks is determined via supervariable agglomeration. Default value is 1 (scalar Jacobi).

Parameters

block_size	upper bound for the size of the Jacobi blocks

get_blocksize()

```
int32 get_blocksize ( )
```

Returns the upper bound for the size of the Jacobi blocks

Returns

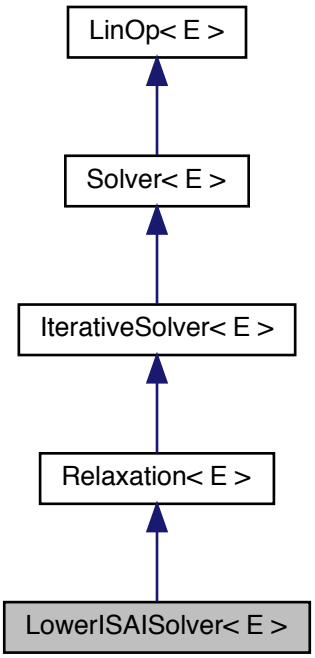
the upper bound for the size of the Jacobi blocks

See also

set_blocksize

7.6 LowerISAISolver< E >

Inheritance diagram for LowerISAISolver< E >:



Public Member Functions

- void set_pattern (std::shared_ptr< const Matrix< E >> pattern)
- std::shared_ptr< const Matrix< E > > get_pattern () const

Static Public Member Functions

- static std::unique_ptr< LowerISAISolver > create (std::shared_ptr< const Ctx > ctx)

7.6.1 Detailed Description

template<typename E>
class msparse::LowerISAISolver< E >

This class represents the Incomplete Sparse Approximate Inverses (ISAI). Specifically, the ISAI for lower triangular matrices. See: H. Anzt, E. Chow, T. Huckle, and J. Dongarra: Batched Generation of Incomplete Sparse Approximate Inverses on GPUs

Template Parameters

E	the type representing the precision E

7.6.2 Member Function Documentation

create()

```
static std::unique_ptr<LowerISAISolver> create (
              std::shared_ptr< const Ctx > ctx )  [static]
```

Create a LowerISAISolver for a 0-by-0 matrix.

Parameters

ctx	the Ctx where the LowerISAISolver will be stored

Returns

a unique pointer to the newly created LowerISAISolver

set_pattern()

```
void set_pattern (
              std::shared_ptr< const Matrix< E >> pattern )
```

Set the pattern for a ISAI solver.

The nonzero pattern of the input matrix A will be used to approximate the inverse of the entitiy the function is called on.

Parameters

pattern	the sparse matrix (pattern) that is used for the ISAI nonzero pattern

get_pattern()

```
std::shared_ptr<const Matrix<E> > get_pattern ( ) const
```

Get the ISAI solver sparsity pattern.

Returns

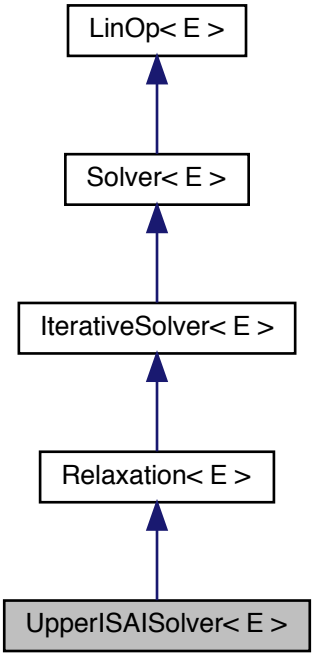
the sparsity pattern used by the solver

See also

set_pattern()

7.7 UpperISAISolver< E >

Inheritance diagram for UpperISAISolver< E >:



Public Member Functions

- void set_pattern (std::shared_ptr< const Matrix< E >> pattern)
- std::shared_ptr< const Matrix< E > > get_pattern () const

Static Public Member Functions

- static std::unique_ptr< UpperISAISolver > create (std::shared_ptr< const Ctx > ctx)

7.7.1 Detailed Description

template<typename E>
class msparse::UpperISAISolver< E >

This class represents the Incomplete Sparse Approximate Inverses (ISAI). Specifically, the ISAI for upper triangular matrices. See: H. Anzt, E. Chow, T. Huckle, and J. Dongarra: Batched Generation of Incomplete Sparse Approximate Inverses on GPUs

Template Parameters

E	the type representing the precision E

7.7.2 Member Function Documentation

create()

```
static std::unique_ptr<UpperISAISolver> create (
              std::shared_ptr< const Ctx > ctx )  [static]
```

Create a UpperISAISolver for a 0-by-0 matrix.

Parameters

ctx	the Ctx where the UpperISAISolver will be stored

Returns

a unique pointer to the newly created UpperISAISolver

set_pattern()

```
void set_pattern (
              std::shared_ptr< const Matrix< E >> pattern )
```

Set the pattern for a ISAI solver.

The nonzero pattern of the input matrix A will be used to approximate the inverse of the entitiy the function is called on.

Parameters

A	the sparse matrix (pattern) that is used for the ISAI nonzero pattern

get_pattern()

```
std::shared_ptr<const Matrix<E> > get_pattern ( ) const
```

Get the ISAI solver sparsity pattern.

Returns

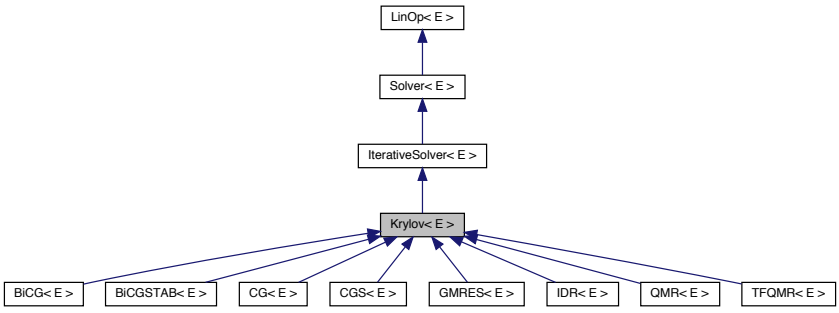
the sparsity pattern used by the solver

See also

set_pattern()

7.8 Krylov< E >

Inheritance diagram for Krylov< E >:



Public Member Functions

- void set_precond (std::shared_ptr< const LinOp< E >> precond)
- std::shared_ptr< const LinOp< E > > get_precond () const

7.8.1 Detailed Description

template<typename E>
class msparse::Krylov< E >

This class represents Krylov solvers, e.g. CG, BiCGSTAB, GMRES...

Template Parameters

E	the type representing the precision E

7.8.2 Member Function Documentation

set_precond()

```
void set_precond (
            std::shared_ptr< const LinOp< E >> precond )
```

Set the preconditioner used by the Krylov solver. The preconditioner is expected to be generated for the linear system which will be solved using the preconditioner.

Parameters

precond	the linear operator used as a preconditioner

get_precond()

```
std::shared_ptr<const LinOp<E> > get_precond ( ) const
```

Get the preconditioner used by the Krylov solver.

Returns

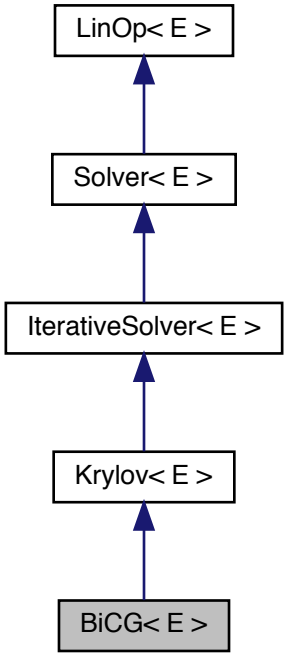
the LinOp used as a preconditioner

See also

set_precond()

7.9 BiCG< E >

Inheritance diagram for BiCG< E >:



Static Public Member Functions

- static std::unique_ptr< BiCG > create (std::shared_ptr< const Ctx > ctx)

7.9.1 Detailed Description

template<typename E>
class msparse::BiCG< E >

This class represents the BiCG Krylov solver that is suitable for iteratively solving general problems. See: Y. Saad, Iterative Methods for Sparse Linear Systems, 2nd edition

Template Parameters

E	the type representing the precision E

7.9.2 Member Function Documentation

create()

```
static std::unique_ptr<BiCG> create (
              std::shared_ptr< const Ctx > ctx )  [static]
```

Create a BiCG Krylov solver for a 0-by-0 matrix.

Parameters

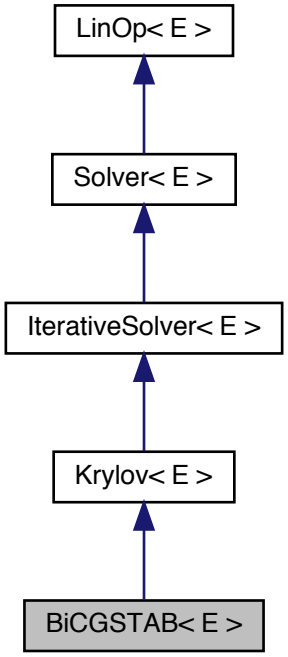
ctx	the Ctx determining location of the new object

Returns

a unique pointer to the newly created object

7.10 BiCGSTAB< E >

Inheritance diagram for BiCGSTAB< E >:



Static Public Member Functions

- static std::unique_ptr< BiCGSTAB > create (std::shared_ptr< const Ctx > ctx)

7.10.1 Detailed Description

template<typename E>
class msparse::BiCGSTAB< E >

This class represents the BiCGSTAB Krylov solver that is suitable for iteratively solving general problems. The stabelized variant of the initial BiCG method has improved numerical

stability and avoids the transposition of the sparse system matrix. See: Y. Saad, Iterative Methods for Sparse Linear Systems, 2nd edition

Template Parameters

E	the type representing the precision E

7.10.2 Member Function Documentation

create()

```
static std::unique_ptr<BiCGSTAB> create (
              std::shared_ptr< const Ctx > ctx )  [static]
```

Create a BiCGSTAB Krylov solver for a 0-by-0 matrix.

Parameters

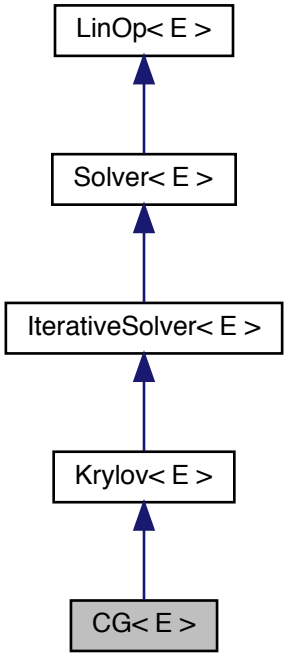
ctx	the Ctx determining location of the new object

Returns

a unique pointer to the newly created object

7.11 CG< E >

Inheritance diagram for CG< E >:



Static Public Member Functions

- static std::unique_ptr< CG > create (std::shared_ptr< const Ctx > ctx)

7.11.1 Detailed Description

template<typename E>
class msparse::CG< E >

This class represents the Conjugate Gradient (CG) Krylov solver that is suitable for iteratively solving symmetric positive definite problems. See: Y. Saad, Iterative Methods for Sparse Linear Systems, 2nd edition

Template Parameters

E	the type representing the precision E

7.11.2 Member Function Documentation

create()

```
static std::unique_ptr<CG> create (
              std::shared_ptr< const Ctx > ctx )  [static]
```

Create a CG Krylov solver for a 0-by-0 matrix.

Parameters

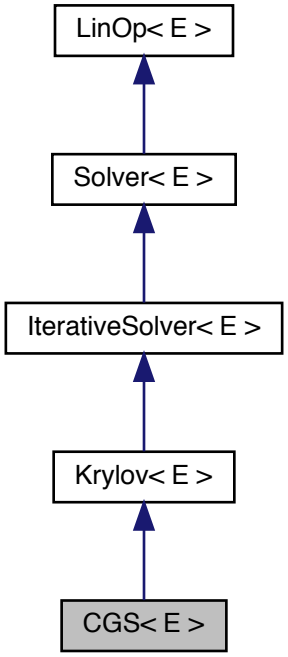
ctx	the Ctx determining location of the new object

Returns

a unique pointer to the newly created object

7.12 CGS< E >

Inheritance diagram for CGS< E >:



Static Public Member Functions

- static std::unique_ptr< CGS > create (std::shared_ptr< const Ctx > ctx)

7.12.1 Detailed Description

template<typename E>
class msparse::CGS< E >

This class represents the Conjugate Gradient Squares (CGS) Krylov solver which is suitable for iteratively solving general problems. See: Y. Saad, Iterative Methods for Sparse Linear Systems, 2nd edition

Template Parameters

E	the type representing the precision E

7.12.2 Member Function Documentation

create()

```
static std::unique_ptr<CGS> create (
              std::shared_ptr< const Ctx > ctx )  [static]
```

Create a CGS Krylov solver for a 0-by-0 matrix.

Parameters

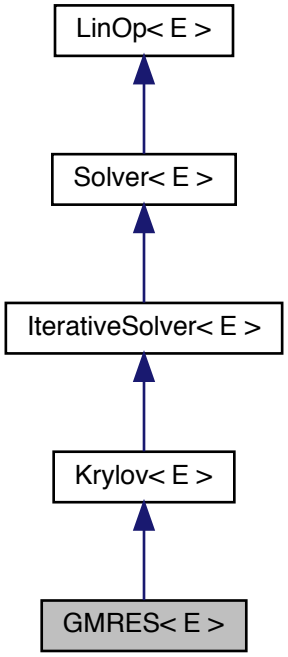
ctx	the Ctx determining location of the new object

Returns

a unique pointer to the newly created object

7.13 GMRES< E >

Inheritance diagram for GMRES< E >:



Public Member Functions

- virtual void set_restart (int32 restart)
- int32 get_restart ()

Static Public Member Functions

- static std::unique_ptr< GMRES > create (std::shared_ptr< const Ctx > ctx)

7.13.1 Detailed Description

template<typename E>
class msparse::GMRES< E >

This class represents the restarted GMRES (Generalized Minimal RESidual) Krylov solver that is suitable for iteratively solving general problems. The restart parameter is by default pre-set to 50. See: Y. Saad, Iterative Methods for Sparse Linear Systems, 2nd edition

Template Parameters

E	the type representing the precision E

7.13.2 Member Function Documentation

create()

```
static std::unique_ptr<GMRES> create (
              std::shared_ptr< const Ctx > ctx )  [static]
```

Create a GMRES Krylov solver for a 0-by-0 matrix.

Parameters

ctx	the Ctx determining location of the new object

Returns

a unique pointer to the newly created object

set_restart()

```
virtual void set_restart (
              int32 restart )  [virtual]
```

Set the restart parameter of the GMRES solver. Default value is 50.

Parameters

restart	parameter determining when the GMRES is restarted

get_restart()

```
int32 get_restart ( )
```

Get the restart parameter of the GMRES solver.

Returns

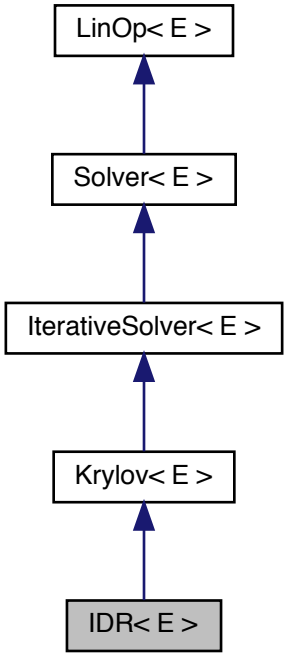
restart parameter

See also

set_restart().

7.14 IDR< E >

Inheritance diagram for IDR< E >:



Public Member Functions

- virtual void set_shadow_space_dim (int32 shadow_space_dim)
- int32 get_shadow_space_dim ()

Static Public Member Functions

- static std::unique_ptr< IDR > create (std::shared_ptr< const Ctx > ctx)

7.14.1 Detailed Description

template<typename E>
class msparse::IDR< E >

This class represents the induced dimension reduction Krylov solver with flexible shadow space dimension (IDR(s)). It is suitable for solving general problems iteratively. The shadow space dimension is by default set to 4. See: P. Sonneveld and M. van Gijzen, IDR(s): A family of simple and fast algorithms for solving large nonsymmetric systems of linear equations.

Template Parameters

E	the type representing the precision E

7.14.2 Member Function Documentation

create()

```
static std::unique_ptr<IDR> create (
              std::shared_ptr< const Ctx > ctx )  [static]
```

Create a IDR(s) Krylov solver for a 0-by-0 matrix.

Parameters

ctx	the Ctx determining location of the new object

Returns

a unique pointer to the newly created object

set_shadow_space_dim()

```
virtual void set_shadow_space_dim (
              int32 shadow_space_dim )  [virtual]
```

Set the shadow space dimension s of an IDR(s) solver.

Parameters

shadow	space dimension

get_shadow_space_dim()

```
int32 get_shadow_space_dim ( )
```

Get the shadow space dimension of the IDR(s) solver.

Returns

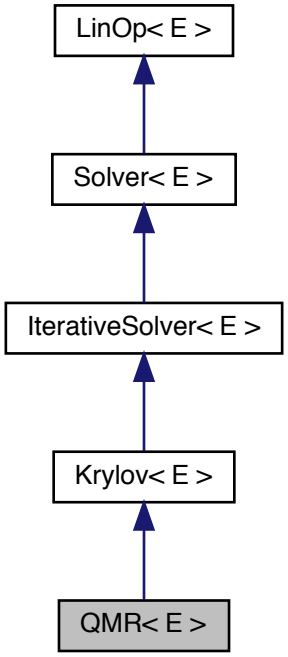
shadow space dimension

See also

set_shadow_space_dim().

7.15 QMR< E >

Inheritance diagram for QMR< E >:



Static Public Member Functions

- static std::unique_ptr< QMR > create (std::shared_ptr< const Ctx > ctx)

7.15.1 Detailed Description

template<typename E>
class msparse::QMR< E >

This class represents the Quasi-Minimal Residual (QMR) Krylov solver that is suitable for iteratively solving general problems. See: Y. Saad, Iterative Methods for Sparse Linear Systems, 2nd edition

Template Parameters

E	the type representing the precision E

7.15.2 Member Function Documentation

create()

```
static std::unique_ptr<QMR> create (
              std::shared_ptr< const Ctx > ctx )  [static]
```

Create a QMR Krylov solver for a 0-by-0 matrix.

Parameters

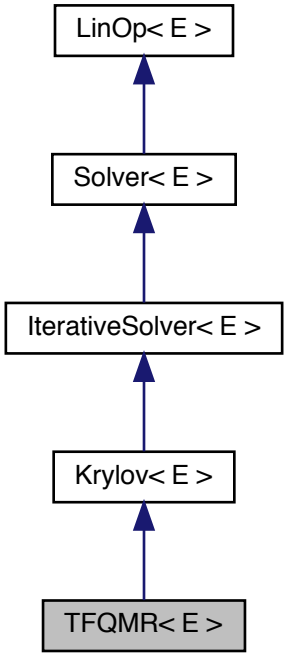
ctx	the Ctx determining location of the new object

Returns

a unique pointer to the newly created object

7.16 TFQMR< E >

Inheritance diagram for TFQMR< E >:



Static Public Member Functions

- static std::unique_ptr< TFQMR > create (std::shared_ptr< const Ctx > ctx)

7.16.1 Detailed Description

template<typename E>
class msparse::TFQMR< E >

This class represents the Transpose-Free Quasi-Minimal Residual (TFQMR) Krylov solver that is suitable for iteratively solving general problems. It avoids the generation of the

transposed system matrix. See: Y. Saad, Iterative Methods for Sparse Linear Systems, 2nd edition

Template Parameters

E	the type representing the precision E

7.16.2 Member Function Documentation

create()

```
static std::unique_ptr<TFQMR> create (
              std::shared_ptr< const Ctx > ctx )  [static]
```

Create a TFQMR Krylov solver for a 0-by-0 matrix.

Parameters

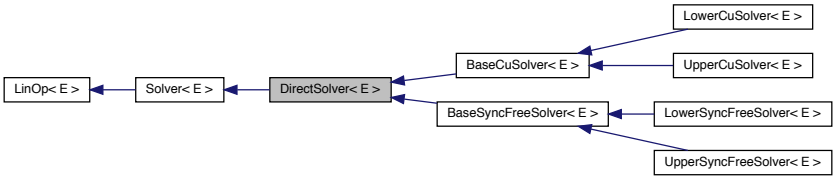
ctx	the Ctx determining location of the new object

Returns

a unique pointer to the newly created object

7.17 DirectSolver< E >

Inheritance diagram for DirectSolver< E >:



7.17.1 Detailed Description

template<typename E>
class msparse::DirectSolver< E >

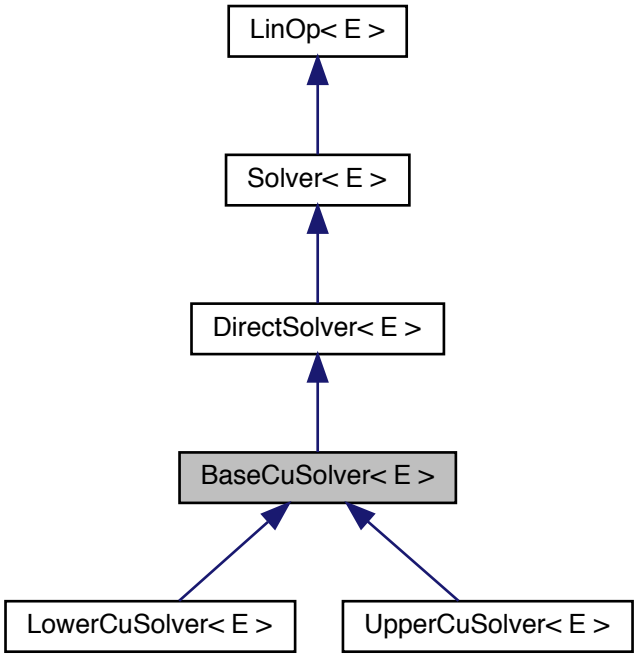
This class represents exact solvers, i.e. direct methods for solving a linear system of equations.

Template Parameters

E	the type representing the precision E

7.18 BaseCuSolver< E >

Inheritance diagram for BaseCuSolver< E >:



7.18.1 Detailed Description

template<typename E>
class msparse::BaseCuSolver< E >

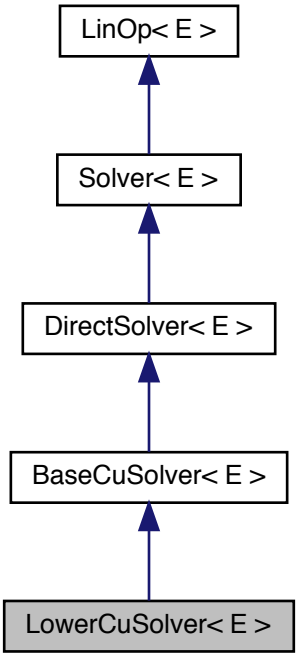
This class represents the exact triangular solver included in NVIDIA's cuSPARSE library. The parallelization of this solver is based on level-scheduling.

Template Parameters

E	the type representing the precision E

7.19 LowerCuSolver< E >

Inheritance diagram for LowerCuSolver< E >:



Static Public Member Functions

- static std::unique_ptr< LowerCuSolver > create (std::shared_ptr< const Ctx > ctx)

7.19.1 Detailed Description

template<typename E>
class msparse::LowerCuSolver< E >

This class represents the exact lower triangular solver included in NVIDIA's cuSPARSE library.

Template Parameters

E	the type representing the precision E

7.19.2 Member Function Documentation

create()

```
static std::unique_ptr<LowerCuSolver> create (
              std::shared_ptr< const Ctx > ctx )  [static]
```

Create a LowerCuSolver for a 0-by-0 matrix.

Parameters

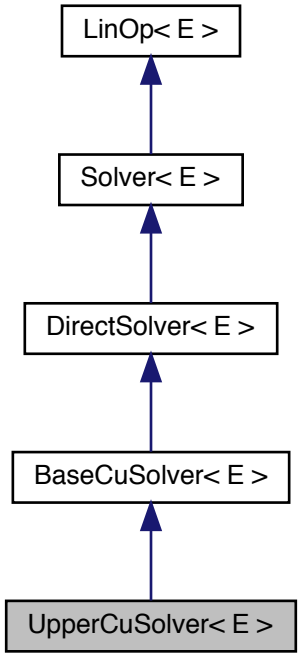
ctx	the Ctx determining location of the new object

Returns

a unique pointer to the newly created object

7.20 UpperCuSolver< E >

Inheritance diagram for UpperCuSolver< E >:



Static Public Member Functions

- static std::unique_ptr< UpperCuSolver > create (std::shared_ptr< const Ctx > ctx)

7.20.1 Detailed Description

template<typename E>
class msparse::UpperCuSolver< E >

This class represents the exact upper triangular solver included in NVIDIA's cuSPARSE library.

Template Parameters

E	the type representing the precision E

7.20.2 Member Function Documentation

create()

```
static std::unique_ptr<UpperCuSolver> create (
              std::shared_ptr< const Ctx > ctx )  [static]
```

Create a UpperCuSolver for a 0-by-0 matrix.

Parameters

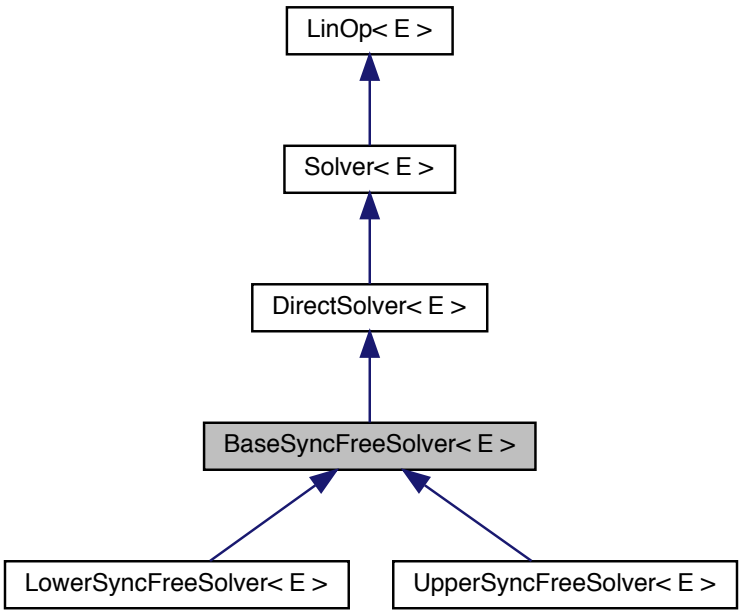
ctx	the Ctx determining location of the new object

Returns

a unique pointer to the newly created object

7.21 BaseSyncFreeSolver< E >

Inheritance diagram for BaseSyncFreeSolver< E >:



7.21.1 Detailed Description

template<typename E>
class msparse::BaseSyncFreeSolver< E >

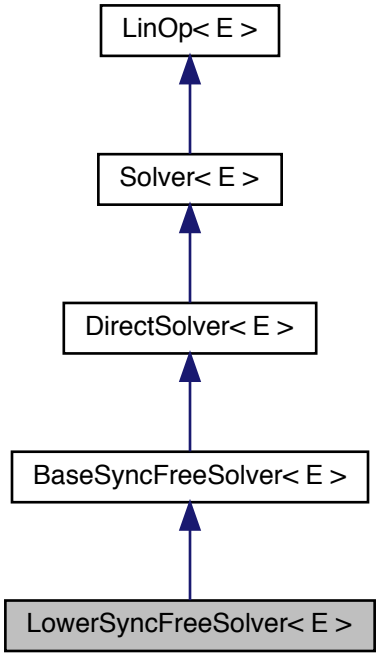
This class represents the Synchronization-free solvers. See: W. Liu et al., A Synchronization-Free Algorithm for Parallel Sparse Triangular Solves

Template Parameters

E	the type representing the precision E

7.22 LowerSyncFreeSolver< E >

Inheritance diagram for LowerSyncFreeSolver< E >:



Static Public Member Functions

- static std::unique_ptr< LowerSyncFreeSolver > create (std::shared_ptr< const Ctx > ctx, block_size_=1)

7.22.1 Detailed Description

template<typename E>
class msparse::LowerSyncFreeSolver< E >

This class represents the synchronization-free solvers for lower triangular matrices.

Template Parameters

E	the type representing the precision E

7.22.2 Member Function Documentation

create()

```
static std::unique_ptr<LowerSyncFreeSolver> create (
              std::shared_ptr< const Ctx > ctx,
              block_size_  = 1 )  [static]
```

Create a LowerSyncFreeSolver solver for a 0-by-0 matrix.

Parameters

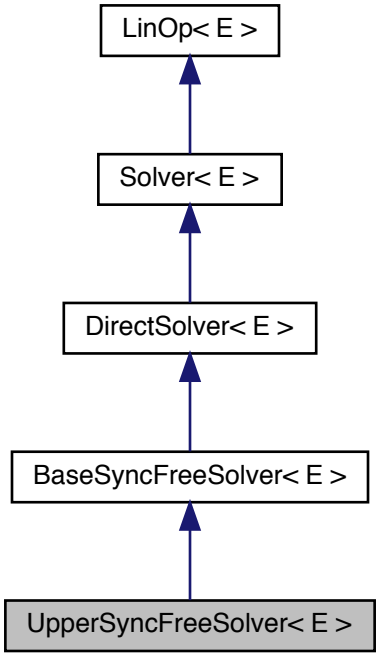
ctx	the Ctx determining location of the new object

Returns

a unique pointer to the newly created object

7.23 UpperSyncFreeSolver< E >

Inheritance diagram for UpperSyncFreeSolver< E >:



Static Public Member Functions

- static std::unique_ptr< UpperSyncFreeSolver > create (std::shared_ptr< const Ctx > ctx)

7.23.1 Detailed Description

template<typename E>
class msparse::UpperSyncFreeSolver< E >

This class represents the synchronization-free solvers for upper triangular matrices.

Template Parameters

E	the type representing the precision E

7.23.2 Member Function Documentation

create()

```
static std::unique_ptr<UpperSyncFreeSolver> create (
              std::shared_ptr< const Ctx > ctx )  [static]
```

Create a UpperSyncFreeSolver solver for a 0-by-0 matrix.

Parameters

ctx	the Ctx determining location of the new object

Returns

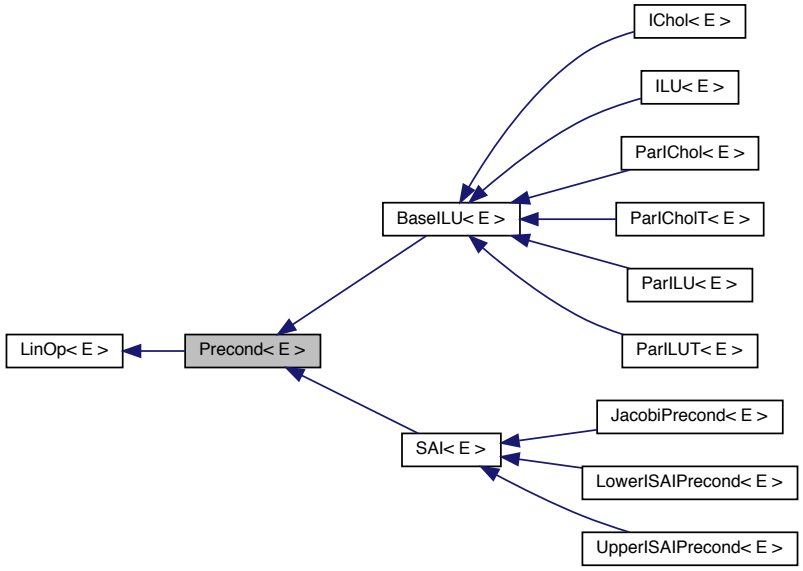
a unique pointer to the newly created object

CHAPTER 8

Preconditioners

8.1 Precond< E >

Inheritance diagram for Precond< E >:



8.1.1 Detailed Description

template<typename E>
class msparse::Precond< E >

The underlying concept of preconditioning is to replace the original problem $Ax = b$ with a preconditioned problem $M^{-1}Ax = M^{-1}b$, where M^{-1} approximates the inverse of A, and $M^{-1}A$ allows for better convergence properties of an iterative solver (e.g., Krylov). Usually, the product $M^{-1}A$ is not formed explicitly, but instead the preconditioner M^{-1} is generated (in some form) prior to the iteration phase, and then applied in every iteration step of the iterative solver. Hence, the use of a preconditioner typically splits into two distinct blocks:

- The preconditioner generation prior to the iteration phase;
- The application of the preconditioner in every step of the iterative solver;

In terms of the LinOp interface, the LinOp::generate() routine applies the "aproximate inverse" operator $op : A \rightarrow M^{-1}$ to the coefficient matrix A, while the LinOp::apply()

routine applies the implicitly stored preconditioner matrix to a vector producing the output $z = op(A)y = M^{-1}y$.

Thus, the process of using a preconditioner is similar to using a solver:

```
auto ilu = msparse::generate<msparse::ILU<double>>(A, gpu);
auto z = msparse::apply(ilu.get(), y.get());
```

Nevertheless, direct use of preconditioners is rare, since they are usually supplied as input parameters for Krylov solvers, like in the following example of the BiCGSTAB solver enhanced with an ILU preconditioner:

```
std::shared_ptr<msparse::CsrMatrix<double>> A
    = msparse::read_from_mtx_to<msparse::CsrMatrix<double>>("A.mtx", gpu);
auto solver = msparse::generate<msparse::BiCGSTAB<double>>(A, gpu);
// create an ILU preconditioner with fill-in levels set to 1
auto precond = msparse::generate<msparse::ILU<double>>(A, gpu, 1);
// use this preconditioner for the solver
solver->set_precond(std::move(precond));

// now the preconditioned solver can be used to solve a system
```

MAGMA-sparse provides a good variety of different types of ILU-based preconditioners (ILU, ParILU, ParILUT, and symmetric variants IChol, ParIChol, ParICholT), as well as sparse approximate inverse-based preconditioners, such as JacobiPrecond and LowerISAIPrecond / UpperISAIPrecond.

In addition, every LinOp can be used as a preconditioner. For example, if a user already knows of a good preconditioner matrix M^{-1} for his linear system, this can be supplied directly to the solver:

```
std::shared_ptr<msparse::CsrMatrix<double>> A
    = msparse::read_from_mtx_to<msparse::CsrMatrix<double>>("A.mtx", gpu);
auto solver = msparse::generate<BiCGSTAB<double>>(A, gpu);
// read a preconditioner matrix from file
auto precond = msparse::read_from_mtx_to<msparse::CsrMatrix<double>>(
            "Minv.mtx", gpu);
// use this preconditioner for the solver
solver->set_precond(std::move(precond));

// now the preconditioned solver can be used to solve a system
```

Finally, it is possible to construct cascaded solvers like flexible GMRES by preconditioning a solver with another solver (which could in turn also be preconditioned by another preconditioner):

```
std::shared_ptr<msparse::CsrMatrix<double>> A
    = msparse::read_from_mtx_to<msparse::CsrMatrix<double>>("A.mtx", gpu);
```

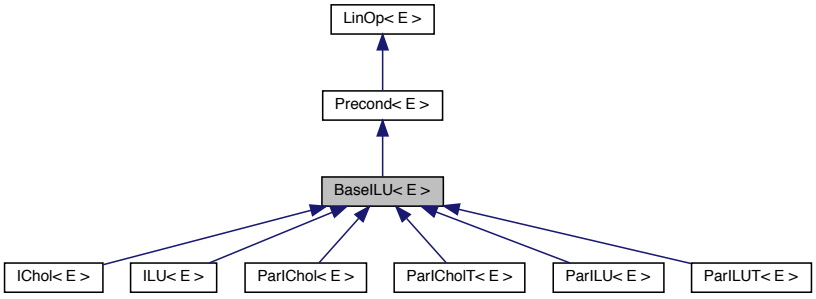
```
auto solver = msparse::generate<msparse::GMRES<double>>(A, gpu);

// create a solver to be used as a preconditioner
auto precond = msparse::generate<msparse::GMRES<double>>(A, gpu);
precond->set_iteration_limit(20);

// it is also possible to precondition the inner solver by constructing
// another preconditioner and uncommenting the following line:
//precond->set_precond(another_precond);

// use this preconditioner for the solver
solver->set_precond(std::move(precond));

// now the preconditioned solver can be used to solve a system
```

Template Parameters

E	the type representing the precision

8.2 BaseILU< E >

Inheritance diagram for BaseILU< E >:



Public Member Functions

- std::shared_ptr< const Matrix< E > > get_lower () const
- std::shared_ptr< const Matrix< E > > get_upper () const
- void set_solvers (std::unique_ptr< LinOp< E >> lsolver, std::unique_ptr< LinOp< E >> usolver)

- std::shared_ptr< const LinOp< E > > get_lower_solver () const
- std::shared_ptr< const LinOp< E > > get_upper_solver () const
- virtual void apply_first (const DenseMatrix< E > *b, DenseMatrix< E > *x) const override
- virtual void apply_second (const DenseMatrix< E > *b, DenseMatrix< E > *x) const override

8.2.1 Detailed Description

template<typename E>
class msparse::BaseILU< E >

This class represents preconditioners based on incomplete factorizations.

Template Parameters

E	the type representing the precision E

8.2.2 Member Function Documentation

get_lower()

```
std::shared_ptr<const Matrix<E> > get_lower ( ) const
```

Get the lower triangular factor.

Returns

lower triangular solver

get_upper()

```
std::shared_ptr<const Matrix<E> > get_upper ( ) const
```

Get the upper triangular factor.

Returns

upper triangular solver

set_solvers()

```
void set_solvers (
              std::unique_ptr< LinOp< E >> lsolver,
              std::unique_ptr< LinOp< E >> usolver )
```

Set the triangular solvers for a incomplete factorization type of preconditioner.

Parameters

lsolver	solver for the lower triangular factor
usolver	solver for the upper triangular factor

get_lower_solver()

```
std::shared_ptr<const LinOp<E> > get_lower_solver ( ) const
```

Get the lower triangular solver.

Returns

lower triangular solver

See also

set_solvers()

get_upper_solver()

```
std::shared_ptr<const LinOp<E> > get_upper_solver ( ) const
```

Get the upper triangular solver.

Returns

upper triangular solver

See also

set_solvers()

apply_first()

```
virtual void apply_first (
              const DenseMatrix< E > * b,
              DenseMatrix< E > * x ) const  [override], [virtual]
```

For factorization type of preconditioners: apply the lower triangular factor.

Parameters

right-hand-side	b
solution	vector x

Reimplemented from LinOp< E >.

apply_second()

```
virtual void apply_second (
              const DenseMatrix< E > * b,
              DenseMatrix< E > * x ) const  [override], [virtual]
```

For factorization type of preconditioners: apply the upper triangular factor.

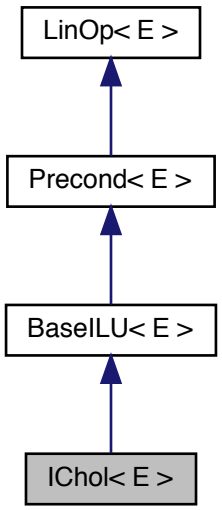
Parameters

right-hand-side	b
solution	vector x

Reimplemented from LinOp< E >.

8.3 IChol< E >

Inheritance diagram for IChol< E >:



Public Member Functions

- void set_levels (size_type n_levels)
- size_type get_levels () const

Static Public Member Functions

- static std::unique_ptr< IChol > create (std::shared_ptr< const Ctx > ctx, size_type n_levels=0)

8.3.1 Detailed Description

template<typename E>
class msparse::IChol< E >

This class represents level-based Incomplete Cholesky factorization preconditioners. Incomplete Cholesky-based preconditioners are suitable for symmetric positive definite linear problems. The number of fill-in levels is controlled via the parameter n_levels, the default value is set to zero.

Template Parameters

E	the type representing the precision E

8.3.2 Member Function Documentation

create()

```
static std::unique_ptr<IChol> create (
              std::shared_ptr< const Ctx > ctx,
              size_type n_levels = 0 )  [static]
```

Create a new 0-by-0 IChol preconditioner.

Parameters

ctx	the Ctx where the ILU will be stored
n_levels	the number of ILU levels

Returns

a unique pointer to the newly created IChol

set_levels()

```
void set_levels (
              size_type n_levels )
```

Set the number of level-ILU levels for the preconditioner. Default is n_levels = 0 : ILU without fill-in.

Parameters

n_levels	is the number of ILU levels

get_levels()

```
size_type get_levels ( ) const
```

Get the number of levels for the level-ILU preconditioner.

Returns

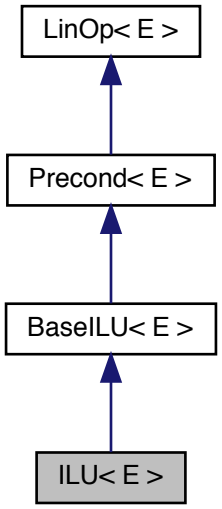
the number of ILU levels of the preconditioner

See also

set_levels()

8.4 ILU< E >

Inheritance diagram for ILU< E >:



Public Member Functions

- void set_levels (size_type n_levels)
- size_type get_levels () const

Static Public Member Functions

- static std::unique_ptr< ILU > create (std::shared_ptr< const Ctx > ctx, size_type n_levels=0)

8.4.1 Detailed Description

template<typename E>
class msparse::ILU< E >

This class represents level-based Incomplete LU factorization preconditioners. ILU-based preconditioners are suitable for general linear problems. The number of fill-in levels is controlled via the parameter n_levels, the default value is set to zero.

Template Parameters

E	the type representing the precision E

8.4.2 Member Function Documentation

create()

```
static std::unique_ptr<ILU> create (
              std::shared_ptr< const Ctx > ctx,
              size_type n_levels = 0 )  [static]
```

Create a new 0-by-0 ILU preconditioner.

Parameters

ctx	the Ctx where the ILU will be stored
n_levels	the number of ILU levels

Returns

a unique pointer to the newly created ILU

set_levels()

```
void set_levels (
              size_type n_levels )
```

Set the number of level-ILU levels for the preconditioner. Default is n_levels = 0 : ILU without fill-in.

Parameters

n_levels	is the number of ILU levels

get_levels()

```
size_type get_levels ( ) const
```

Get the number of levels for the level-ILU preconditioner.

Returns

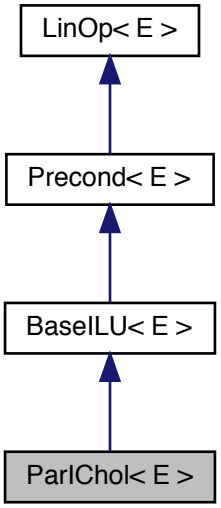
the number of ILU levels of the preconditioner

See also

set_levels()

8.5 ParIChol< E >

Inheritance diagram for ParIChol< E >:



Public Member Functions

- void set_levels (size_type n_levels)
- size_type get_levels () const
- set_sweeps (int32 n_sweeps)
- size_type get_sweeps () const

Static Public Member Functions

- static std::unique_ptr< ParIChol > create (std::shared_ptr< const Ctx > ctx, size_type n_levels=0, size_type n_sweeps=5)

8.5.1 Detailed Description

template<typename E>
class msparse::ParIChol< E >

This class represents Incomplete Cholesky factorization preconditioners that are generated via fixed-point iterations. ParIChol-based preconditioners are suitable for symmetric positive definite linear problems. The sweep count is used to control the number of ParIChol steps in the generation.

Template Parameters

E	the type representing the precision E

8.5.2 Member Function Documentation

create()

```
static std::unique_ptr<ParIChol> create (
              std::shared_ptr< const Ctx > ctx,
              size_type n_levels = 0,
              size_type n_sweeps = 5 )  [static]
```

Create a new 0-by-0 ParIChol preconditioner.

Parameters

ctx	the Ctx determining location of the new object
n_levels	the number of ILU levels
n_sweeps	the number of ParIChol sweeps

Returns

a unique pointer to the newly created object

set_levels()

```
void set_levels (
```

```
            size_type n_levels )
```

Set the number of level-ILU levels for the preconditioner. Default is n_levels = 0 : ILU without fill-in.

Parameters

n_levels	is the number of ILU levels

get_levels()

```
size_type get_levels ( ) const
```

Get the number of levels for the level-ILU preconditioner.

Returns

the number of ILU levels of the preconditioner

See also

set_levels()

set_sweeps()

```
set_sweeps (
            int32 n_sweeps )
```

Set the number of fixed-point sweeps to generate the Incomplete Cholesky preconditioner.

Parameters

n_sweeps	is the number of fixed-point sweeps

get_sweeps()

```
size_type get_sweeps ( ) const
```

Get the number of ParILUT sweeps, each containing 2 fixed-point sweeps.

Returns

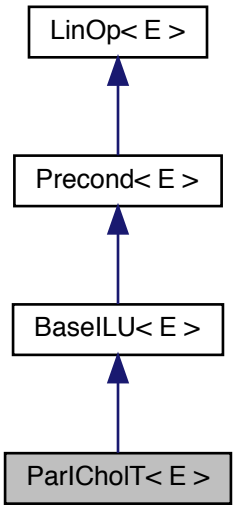
number of ParILUT sweeps.

See also

set_sweeps()

8.6 ParICholT< E >

Inheritance diagram for ParICholT< E >:



Public Member Functions

- void set_levels (size_type n_levels)
- size_type get_levels () const
- set_sweeps (int32 n_sweeps)
- size_type get_sweeps () const

- set_drop_tol (real drop_tol)
- real get_drop_tol () const
- set_fill_tol (real fill_tol)
- real get_fill_tol () const

Static Public Member Functions

- static std::unique_ptr< ParICholT > create (std::shared_ptr< const Ctx > ctx, size_type n_levels=0, size_type n_sweeps=5)

8.6.1 Detailed Description

template<typename E>
class msparse::ParICholT< E >

This class represents Incomplete Cholesky factorization preconditioners using thresholding that are generated via fixed-point iterations. The fill-in is controlled via drop_tol, fill_tol. ParIChol-based preconditioners are suitable for symmetric positive definite linear problems. The sweep count is used to control the number of ParICholT steps in the generation.

Template Parameters

E	the type representing the precision E

8.6.2 Member Function Documentation

create()

```
static std::unique_ptr<ParICholT> create (
              std::shared_ptr< const Ctx > ctx,
              size_type n_levels = 0,
              size_type n_sweeps = 5 )  [static]
```

Create a new 0-by-0 ParICholT preconditioner.

Parameters

ctx	the Ctx determining location of the new object
n_levels	the number of ILU levels
n_sweeps	the number of ParIChol sweeps

Returns

a unique pointer to the newly created object

set_levels()

```
void set_levels (
            size_type n_levels )
```

Set the number of level-ILU levels for the preconditioner. Default is n_levels = 0 : ILU without fill-in.

Parameters

n_levels	is the number of ILU levels

get_levels()

```
size_type get_levels ( ) const
```

Get the number of levels for the level-ILU preconditioner.

Returns

the number of ILU levels of the preconditioner

See also

set_levels()

set_sweeps()

```
set_sweeps (
            int32 n_sweeps )
```

Set the number of fixed-point sweeps to generate the Incomplete Cholesky preconditioner.

Parameters

n_sweeps	is the number of fixed-point sweeps

get_sweeps()

```
size_type get_sweeps ( ) const
```

Get the number of ParILUT sweeps, each containing 2 fixed-point sweeps.

Returns

number of ParILUT sweeps.

See also

set_sweeps()

set_drop_tol()

```
set_drop_tol (
              real drop_tol )
```

Set the parameter controlling the drop tolerance in ParILUT. All elements of magnitude size smaller are discarded in the generation process.

Parameters

drop_tol	the magnitude threshold below elements are ignored

get_drop_tol()

```
real get_drop_tol ( ) const
```

Get the drop tolerance of ParILUT.

Returns

the tolerance working as magnitude threshold which elements are excluded.

See also

set_drop_tol()

set_fill_tol()

```
set_fill_tol (
              real fill_tol )
```

Set the parameter controlling the ratio between the elements in the original factors and the threshold ILU factors.

Parameters

fill_tol	is the number of fixed-point sweeps

get_fill_tol()

```
real get_fill_tol ( ) const
```

Get the parameter controlling the ratio between the elements in the original factors and the threshold ILU factors.

Returns

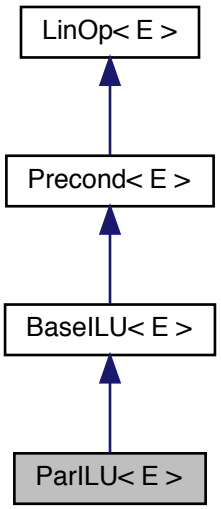
the fill tolerance

See also

set_fill_tol()

8.7 ParILU< E >

Inheritance diagram for ParILU< E >:



Public Member Functions

- void set_levels (size_type n_levels)
- size_type get_levels () const
- set_sweeps (int32 n_sweeps)
- size_type get_sweeps () const

Static Public Member Functions

- static std::unique_ptr< ParILU > create (std::shared_ptr< const Ctx > ctx, size_type n_levels=0, size_type n_sweeps=5)

8.7.1 Detailed Description

template<typename E>
class msparse::ParILU< E >

This class represents Incomplete LU factorization preconditioners that are generated via fixed-point iterations. See: E. Chow, H. Anzt, and J. Dongarra, Asynchronous iterative algorithm for computing incomplete factorizations on GPUs. ILU-based preconditioners are suitable for general linear problems. The sweep count is used to control the number of fixed-point sweeps in the generation.

Template Parameters

E	the type representing the precision E

8.7.2 Member Function Documentation

create()

```
static std::unique_ptr<ParILU> create (
              std::shared_ptr< const Ctx > ctx,
              size_type n_levels = 0,
              size_type n_sweeps = 5 )  [static]
```

Create a new 0-by-0 ParILU preconditioner.

Parameters

ctx	the Ctx determining location of the new object
n_levels	the number of ILU levels
n_sweeps	the number of ParIChol sweeps

Returns

a unique pointer to the newly created object

set_levels()

```
void set_levels (
            size_type n_levels )
```

Set the number of level-ILU levels for the preconditioner. Default is n_levels = 0 : ILU without fill-in.

Parameters

n_levels	is the number of ILU levels

get_levels()

```
size_type get_levels ( ) const
```

Get the number of levels for the level-ILU preconditioner.

Returns

the number of ILU levels of the preconditioner

See also

set_levels()

set_sweeps()

```
set_sweeps (
            int32 n_sweeps )
```

Set the number of fixed-point sweeps to generate the Incomplete Cholesky preconditioner.

Parameters

n_sweeps	is the number of fixed-point sweeps

get_sweeps()

```
size_type get_sweeps ( ) const
```

Get the number of ParILUT sweeps, each containing 2 fixed-point sweeps.

Returns

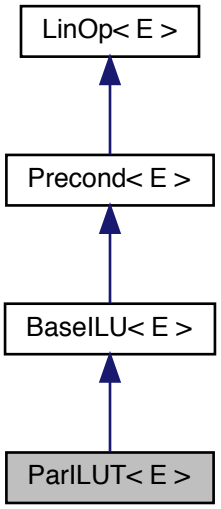
number of ParILUT sweeps.

See also

set_sweeps()

8.8 ParILUT< E >

Inheritance diagram for ParILUT< E >:



Public Member Functions

- void set_levels (size_type n_levels)
- size_type get_levels () const
- set_sweeps (int32 n_sweeps)
- size_type get_sweeps () const
- set_drop_tol (real drop_tol)
- real get_drop_tol () const
- set_fill_tol (real fill_tol)
- real get_fill_tol () const

Static Public Member Functions

- static std::unique_ptr< ParILUT > create (std::shared_ptr< const Ctx > ctx, size_type n_levels=0, size_type n_sweeps=5)

8.8.1 Detailed Description

template<typename E>
class msparse::ParILUT< E >

This class represents Incomplete LU factorization preconditioners using thresholding that are generated via fixed-point iterations. See: H. Anzt, E. Chow, and J. Dongarra, ParILUT - a new parallel threshold ILU. The fill-in is controlled via drop_tol, fill_tol. ILU-based preconditioners are suitable for general linear problems. The sweep count is used to control the number of ParILUT steps in the generation.

Template Parameters

E	the type representing the precision E

8.8.2 Member Function Documentation

create()

```
static std::unique_ptr<ParILUT> create (
              std::shared_ptr< const Ctx > ctx,
              size_type n_levels = 0,
              size_type n_sweeps = 5 )  [static]
```

Create a new 0-by-0 ParILU preconditioner.

Parameters

ctx	the Ctx determining location of the new object
n_levels	the number of ILU levels
n_sweeps	the number of ParIChol sweeps

Returns

a unique pointer to the newly created object

set_levels()

```
void set_levels (
            size_type n_levels )
```

Set the number of level-ILU levels for the preconditioner. Default is n_levels = 0 : ILU without fill-in.

Parameters

n_levels	is the number of ILU levels

get_levels()

```
size_type get_levels ( ) const
```

Get the number of levels for the level-ILU preconditioner.

Returns

the number of ILU levels of the preconditioner

See also

set_levels()

set_sweeps()

```
set_sweeps (
            int32 n_sweeps )
```

Set the number of fixed-point sweeps to generate the Incomplete Cholesky preconditioner.

Parameters

n_sweeps	is the number of fixed-point sweeps

get_sweeps()

```
size_type get_sweeps ( ) const
```

Get the number of ParILUT sweeps, each containing 2 fixed-point sweeps.

Returns

number of ParILUT sweeps.

See also

set_sweeps()

set_drop_tol()

```
set_drop_tol (
              real drop_tol )
```

Set the parameter controlling the drop tolerance in ParILUT. All elements of magnitude size smaller are discarded in the generation process.

Parameters

drop_tol	the magnitude threshold below elements are ignored

get_drop_tol()

```
real get_drop_tol ( ) const
```

Get the drop tolerance of ParILUT.

Returns

the tolerance working as magnitude threshold which elements are excluded.

See also

set_drop_tol()

set_fill_tol()

```
set_fill_tol (
              real fill_tol )
```

Set the parameter controlling the ratio between the elements in the original factors and the threshold ILU factors.

Parameters

fill_tol	is the number of fixed-point sweeps

get_fill_tol()

```
real get_fill_tol ( ) const
```

Get the parameter controlling the ratio between the elements in the original factors and the threshold ILU factors.

Returns

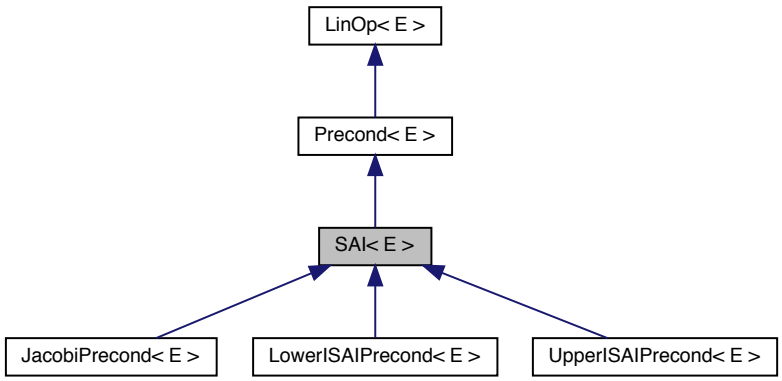
the fill tolerance

See also

set_fill_tol()

8.9 SAI< E >

Inheritance diagram for SAI< E >:



8.9.1 Detailed Description

template<typename E>
class msparse::SAI< E >

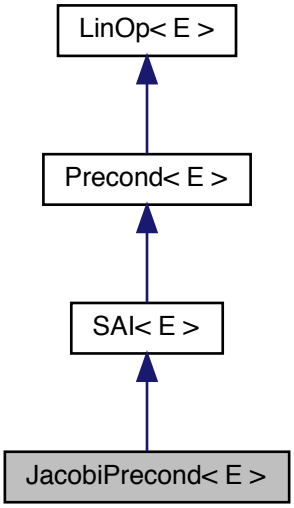
This class represents Sparse Approximate Inverse (SAI) preconditioners. The strategy is to approximate the inverse of the linear operator on a pre- defined nonzero pattern.

Template Parameters

E	the type representing the precision E

8.10 JacobiPrecond< E >

Inheritance diagram for JacobiPrecond< E >:



Public Member Functions

- virtual void set_blocksize (int32 block_size)
- int32 get_blocksize ()

Static Public Member Functions

- static std::unique_ptr< JacobiPrecond > create (std::shared_ptr< const Ctx > ctx, int32 block_size=1)

8.10.1 Detailed Description

template<typename E>
class msparse::JacobiPrecond< E >

This class represents the Jacobi preconditioner (diagonal scaling), flexible in terms of the size of the diagonal blocks. The blocks are determined using supervariable agglomeration algorithm, and the block-size parameter is an upper bound. By default, the block size is set to 1 (scalar Jacobi).

Template Parameters

E	the type representing the precision E

8.10.2 Member Function Documentation

create()

```
static std::unique_ptr<JacobiPrecond> create (
              std::shared_ptr< const Ctx > ctx,
              int32 block_size = 1 )  [static]
```

Create a new 0-by-0 (block-) Jacobi preconditioner.

Parameters

ctx	the Ctx determining location of the new object
block_size	the upper bound for the Jacobi blocks

Returns

a unique pointer to the newly created object

set_blocksize()

```
virtual void set_blocksize (
              int32 block_size )  [virtual]
```

Set the upper bound for the size of the Jacobi blocks. The actual size of the distinct blocks is determined via supervariable agglomeration. The default value is 1 (scalar Jacobi).

Parameters

block_size	upper bound for the size of the Jacobi blocks

get_blocksize()

```
int32 get_blocksize ( )
```

Returns the upper bound for the size of the Jacobi blocks

Returns

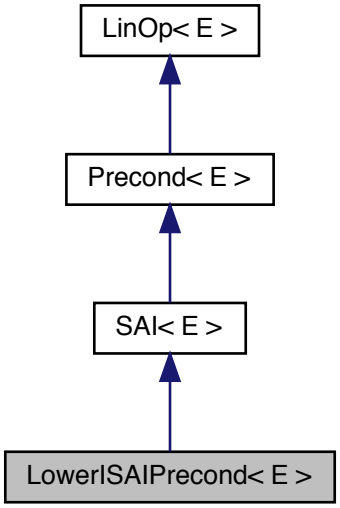
the upper bound for the size of the Jacobi blocks

See also

set_blocksize()

8.11 LowerISAIPrecond< E >

Inheritance diagram for LowerISAIPrecond< E >:



Public Member Functions

- virtual void set_pattern (Matrix *A)

Static Public Member Functions

- static std::unique_ptr< LowerISAIPrecond > create (std::shared_ptr< const Ctx > ctx)

8.11.1 Detailed Description

template<typename E>
class msparse::LowerISAIPrecond< E >

This class represents the Incomplete Sparse Approximate Inverses (ISAI). Specifically, the ISAI for lower triangular matrices. See: H. Anzt, E. Chow, T. Huckle, and J. Dongarra: Batched Generation of Incomplete Sparse Approximate Inverses on GPUs

Template Parameters

E	the type representing the precision E

8.11.2 Member Function Documentation

create()

```
static std::unique_ptr<LowerISAIPrecond> create (
              std::shared_ptr< const Ctx > ctx )  [static]
```

Create a new 0-by-0 LowerISAIPrecond.

Parameters

ctx	the Ctx where the LowerISAIPrecond will be stored

Returns

a unique pointer to the newly created LowerISAIPrecond

set_pattern()

```
virtual void set_pattern (
              Matrix * A )  [virtual]
```

Set the pattern for a ISAI preconditioner.

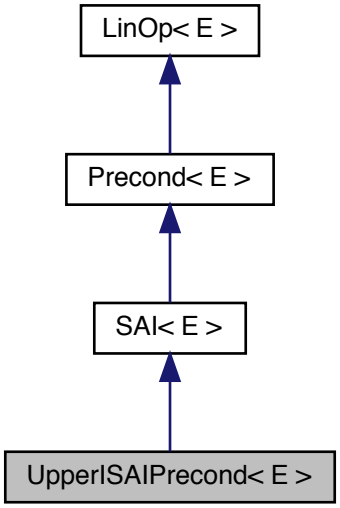
The nonzero pattern of the input matrix A will be used to approximate the inverse of the entitiy the function is called on.

Parameters

A	the sparse matrix (pattern) for that is used for the ISAI nonzero pattern

8.12 UpperISAIPrecond< E >

Inheritance diagram for UpperISAIPrecond< E >:



Public Member Functions

- virtual void set_pattern (Matrix *A)

Static Public Member Functions

- static std::unique_ptr< UpperISAIPrecond > create (std::shared_ptr< const Ctx > ctx)

8.12.1 Detailed Description

template<typename E>
class msparse::UpperISAIPrecond< E >

This class represents the Incomplete Sparse Approximate Inverses (ISAI). Specifically, the ISAI for upper triangular matrices. See: H. Anzt, E. Chow, T. Huckle, and J. Dongarra: Batched Generation of Incomplete Sparse Approximate Inverses on GPUs

Template Parameters

E	the type representing the precision E

8.12.2 Member Function Documentation

create()

```
static std::unique_ptr<UpperISAIPrecond> create (
              std::shared_ptr< const Ctx > ctx )  [static]
```

Create a new 0-by-0 UpperISAIPrecond.

Parameters

ctx	the Ctx where the UpperISAIPrecond will be stored

Returns

a unique pointer to the newly created UpperISAIPrecond

set_pattern()

```
virtual void set_pattern (
              Matrix * A )  [virtual]
```

Set the pattern for a ISAI preconditioner.

The nonzero pattern of the input matrix A will be used to approximate the inverse of the entitiy the function is called on.

Parameters

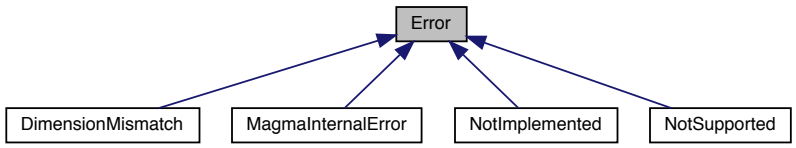
A	the sparse matrix (pattern) for that is used for the ISAI nonzero pattern

CHAPTER 9

Errors

9.1 Error

Inheritance diagram for Error:



Public Member Functions

- virtual const char * what () const noexcept override

9.1.1 Detailed Description

The Error class is used to report exceptional behaviour in library functions. MAGMA-sparse 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 <msparse.h>
#include <iostream>

using namespace msparse;

int main()
{
    auto cpu = create<CpuCtx>();
    auto A = randn_fill<CsrMatrix<float>>(5, 5, 0f, 1f, cpu);
    auto x = fill<DenseMatrix<float>>(6, 1, 1f, cpu);
    try {
        auto y = apply(A.get(), x.get());
    } catch(Error e) {
        // an error occured, write the message to screen and exit
```

```
        std::cout << e.what() << std::endl;
        return -1;
    }
    return 0;
}
```

9.1.2 Member Function Documentation

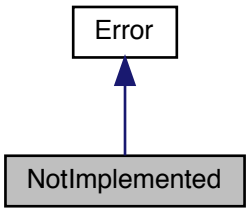
what()

```
virtual const char* what ( ) const  [override], [virtual], [noexcept]
```

Return a human-readable string with a more detailed description of the error.

9.2 NotImplemented

Inheritance diagram for NotImplemented:

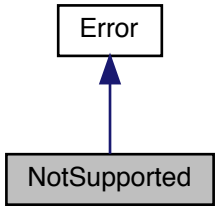


9.2.1 Detailed Description

This type of Error is thrown in case an operation has not yet been implemented (but will be implemented in the future).

9.3 NotSupported

Inheritance diagram for NotSupported:

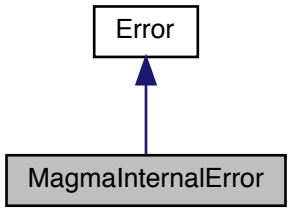


9.3.1 Detailed Description

This type of Error is thrown in case it is not possible to perform the requested operation on the given object type.

9.4 MagmaInternalError

Inheritance diagram for MagmaInternalError:

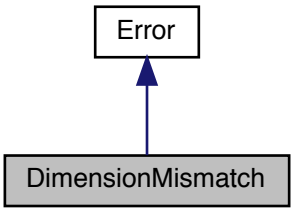


9.4.1 Detailed Description

This type of Error is thrown in case one of the low-level MAGMA(-sparse) routines exits with a nonzero error code.

9.5 DimensionMismatch

Inheritance diagram for DimensionMismatch:



9.5.1 Detailed Description

This type of Error is thrown if a LinOp is being applied to a DenseMatrix of incompatible size.

CHAPTER 10

Abbreviations

Abbreviation	Name	Description
5pt	Five-Point	Finite Difference Stencil
BCSR	Block-CSR	Matrix Storage Format
BiCGStab	Biconjugate Gradient Method	Krylov Solver
CG	Conjugate Gradient Method	Krylov Solver
CGS	Conjugate Gradient Squares	Krylov Solver
col_idx	Column-Index	Array
col_ptr	Column-Pointer	Array
conj	Conjugate	Complex value is conjugated
Coo	Coordinate Format	Matrix Storage Format
CSR	Compressed Sparse Row Format	Matrix Storage Format
Ctx	Context	Represents device
Ell	Ellpack Format	Matrix Storage Format
GMRES	Generalized Minimal Residuals	Krylov Solver
Hyb	Hybrid Format	Matrix Storage Format
IChol	Incomplete Cholesky	Preconditioner
IDR	Induced Dimension Reduction	Krylov Solver
idx	Index	Index
ILU	Incomplete LU factorization	Preconditioner

Abbreviation	Name	Description
ISAI	Incomplete Sparse Approximate Inverse	Matrix
IterRef	Iterative Refinement	Iterative Solver
LinOp	Linear Operator	Matrix, Solver, ...
Mtx	Matrix Market Format	Matrix File Storage Standard
ncols	Number of Columns	Matrix Information
nelems	Number of Elements	Array Information
nnz	Number of Non-Zeros	Matrix Information
nrows	Number of Rows	Matrix Information
ParIChol	Parallel Incomplete Cholesky	Preconditioner
ParICholT	Parallel threshold Incomplete Cholesky	Preconditioner
ParILU	Parallel Incomplete LU factorization	Preconditioner
ParILUT	Parallel threshold ILU	Preconditioner
precond	Preconditioner	Approximate solver
ptr	Pointer	points to memory location
QMR	Quasi-Minimal Residual	Krylov Solver
rand	Random	uniformly distributed
row_idx	Row-Index	Array
row_ptr	Row-Pointer	Array
SAI	Sparse Approximate Inverse	Matrix
Sellp	Sliced Ellpack Format	Matrix Storage Format
Spmv	Sparse Matrix Vector Product	Operation
TFQMR	Transpose-free Quasi-Minimal Residual	Krylov Solver
trans	Transpose	transposed linear operator
TriSolver	Sparse Triangular Solver	Solver
val	Values	Array