Iterative Solver Template Library*

Peter Bastian, Markus Blatt
Institut für parallele und verteilte Systeme (IPVS),
Universität Stuttgart, Universitätsstr. 38, D-70569 Stuttgart,
email: Peter.Bastian@ipvs.uni-stuttgart.de, Markus.Blatt@ipvs.uni-stuttgart.de

July 28, 2009

Abstract

This document describes the rationale behind and use of the Iterative Solver Template Library (ISTL) which provides a set of C++ templates to represent vectors, (sparse) matrices and some generic algorithms based on these. The most prominent features of the matrix/vector classes is that they support a recursive block structure in a bottom up way. The classes can be used, e. g., to efficiently implement block preconditioners for *hp*-finite elements.

Contents

1 Introduction

The numerical solution of partial differential equations (PDEs) frequently requires the solution of large and sparse linear systems. Naturally, there are many libraries available on the internet for doing sparse matrix/vector computations. A comprehensive overview is given in [7].

The widely availably Basic Linear Algebra Subprograms (BLAS) standard has been extended to cover als sparse matrices [5]. BLAS divides the available functions into level 1 (vector operations), level 2 (vector/matrix operations) and level 3 (matrix/matrix

*Part of the Distributed and Unified Numerics Environment (DUNE) which is available from the site http://www.dune-project.org/

operations). BLAS for sparse matrices contains only level 1 and 2 functionality and is quite different to the standard for dense matrices. The standard uses procedural programming style and offers only a FORTRAN and C interface. As a consequence, the interface is "coarse grained", meaning that "small" functions such as access to individual matrix elements is relatively slow.

Generic programming techniqes in C++ offer the possibility to combine flexibility and reuse ("efficiency of the programmer") with fast execution ("efficieny of the program") as has been demonstrated with the Standard Template Library (STL), [15] or the Blitz++ library for multidimensional arrays [6]. A variety of template programming techniques such as traits, template metaprograms, expression templates or the Barton-Nackman trick are used in the implementations, see [2, 16] for an introduction. Application of these ideas to matrix/vector operations is available with the Matrix Template Library (MTL), [12, 14]. The Iterative Template Library (ITL), [11], implements iterative solvers for linear systems (mostly Krylov subspace methods) in a generic way based on MTL. The Distributed and Unified Numerics Environment (DUNE), [3, 8], applies the STL ideas to finite element computations.

Why bother with yet another OO-implementation of (sparse) linear algebra when libraries, most notably the MTL, are available? The most important reason is that the functionality in existing libraries has not been designed specifically with advanced finite element methods in mind. Sparse matrices in finite element computations have a lot of structure. Here are some examples:

- Certain discretizations for systems of PDEs or higher order methods result in matrices where individual entries are replaced by small blocks, say of size 2×2 or 4×4, see Fig. 1(a). Dense blocks of different sizes e. g. arise in *hp* Discontinuous Galerkin discretization methods, see Fig. 1(b). Straightforward iterative methods solve these small blocks exactly, see e. g. [4].

- Equation-wise ordering for systems results in matrices having an $n \times n$ block structure where n corresponds to the number of variables in the PDE and the blocks themselves are large, see Fig. 1(d). As an example we mention the Stokes system. Iterative solvers such as the SIMPLE or Uzawa algorithm use this structure.

- Other discretizations, e. g. those of reaction/diffusion systems, produce sparse matrices whose blocks are sparse matrices of small dense blocks, see fig. 1(c).

- Other structures that can be exploited are the level structure arising from hierarchic meshes, a p-hierarchic structure (e. g. decomposition in linear and quadratic part), geometric structure from decomposition in subdomains or topological structure where unknowns are associated with nodes, edges, faces or elements of a mesh.

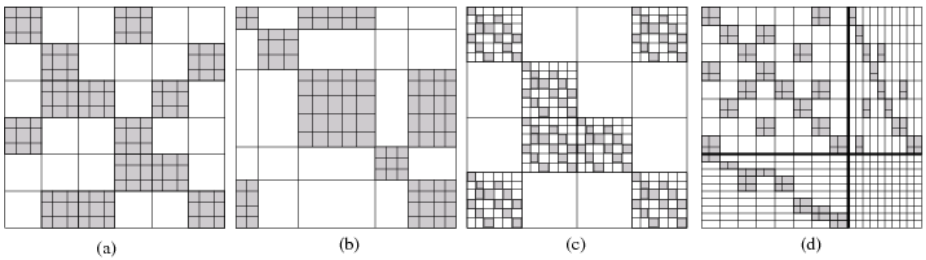

Figure 1: Block structure of matrices arising in the finite element method

It is very important to note that this structure is typically known at compile-time and this knowledge should be exploited to produce efficient code. Moreover, block structuredness is recursive, i. e. matrices are build from blocks which can themselves be build from blocks.

The Matrix Template Library also offers the possibility to partition a matrix into blocks. However, their concept is top-down, i. e. an already existing matrix is enriched by additional information to implement the block structure. This is done at run-time and might thus be less efficient and requires additional memory. In contrast the bottom-up composition of block matrices from blocks can save memory. We would like to stress that the library to be presented in this paper is not nearly as broad in scope as the MTL.

2 Vectors

The interface of our vector classes is designed according to what they represent from a mathematical point of view. The vector classes are representations of vector spaces.

2.1 Vector spaces

In mathemetics vectors are elements of a vector space. A vector space $V(\mathbb{K})$, defined over a field $\mathbb{K}$, is a set of elements with two operations: (i) vector space addition $+ : V \times V \to V$ and (ii) scalar multiplication $* : \mathbb{K} \times V \to V$. These operations obey certain formal rules, see your favourite textbook on linear algebra, e. g. [10]. In addition a vector space may be normed, i. e. there is a function (obeying certain rules) $\|.\| : V \to \mathbb{R}$ which measures distance in the vector space. Even more specialized vector spaces have a scalar product which is a function $\cdot : V \times V \to \mathbb{K}$.

How do you construct a vector space? The easiest way is to take a field, such as $\mathbb{K} = \mathbb{R}$ or $\mathbb{K} = \mathbb{C}$ and take a tensor product:

$$V = \mathbb{K}^n = \underbrace{\mathbb{K} \times \mathbb{K} \times \ldots \times \mathbb{K}}_{n \text{ times}}.$$

$n \in \mathbb{N}$ is called the dimension of the vector space. There are also infinite-dimensional vector spaceswhich are, however, not of interest in the context here. The idea of tensor products can be generalized. If we have vector spaces $V_1(\mathbb{K}), \ldots, V_n(\mathbb{K})$ we can construct a new vector space by setting

$$V(\mathbb{K}) = V_1 \times V_2 \times \ldots \times V_n.$$

The V_i can be *any* vector space over the field $\mathbb{K}$. The dimension of V is the sum of the dimensions of the V_i. For a a mathematician every finite-dimensional vector space is isomorphic to the $\mathbb{R}^k$ for an appropriate k but in our applications it is important to know the difference between $(\mathbb{R}^2)^7$ and $\mathbb{R}^{14}$. Having these remarks about vector spaces in mind we can now turn to the class design.

2.2 Vector classes

ISTL provides the following classes to make up vector spaces:

FieldVector. The `template<class K, int n> FieldVector<K,n>` class template is used to represent a vector space $V = \mathbb{K}^n$ where the field is given by the type `K`. `K` may be `double`, `float`, `complex<double>` or any other numeric type. The dimension given by the

template parameter `n` is assumed to be small. Members of this class are implemented with template metaprograms to avoid tiny loops. Example: Use `FieldVector<double,2>` to define vectors with a fixed dimension 2.

BlockVector. The `template<class B> BlockVector<B>` class template builds a vector space $V = B^n$ where the "block type" B is given by the template parameter `B`. `B` may be any other class implementing the vector interface. The number of blocks n is given at run-time. Example:

```
BlockVector<FieldVector<double,2> >
```

can be used to define vectors of variable size where each block in turn consists of two `double` values.

VariableBlockVector. The `template<class B> VariableBlockVector<B>` class can be used to construct a vector space having a two-level block structure of the form $V = B^{n_1} \times B^{n_2} \times \ldots \times B^{n_m}$, i.e. it consists of m blocks $i = 1, \ldots, m$ and each block in turn consists of n_i blocks given by the type `B`. In principle this structure could be built also with the previous classes but the implementation here is more efficient. It allocates memory in one big array for all components and for certain operations it is more efficient to interpret the vector space as $V = B^{\sum_{i=1}^{m} n_i}$.

2.3 Vectors are containers

Vectors are containers over the base type `K` or `B` in the sense of the Standard Template Library. Random access is provided via `operator[](int i)` where the indices are in the range $0, \ldots, n-1$ with the number of blocks n given by the `N` method. Here is a code fragment for illustration:

```
typedef Dune::FieldVector<std::complex<double>,2> BType;
Dune::BlockVector<BType> v(20);
v[1] = 3.14;
v[3][0] = 2.56;
v[3][1] = std::complex<double>(1,-1);
```

Note how one `operator[]()` is used for each level of block recursion.

Sequential access to container elements is provided via iterators. Here is a generic function accessing all the elements of a vector:

```
template<class V> void f (V& v)
{
  typedef typename V::Iterator iterator;
  for (iterator i=v.begin(); i!=v.end(); ++i)
    *i = i.index();

  typedef typename V::ConstIterator const_iterator;
  for (const_iterator i=v.begin(); i!=v.end(); ++i)
    std::cout << (*i).two_norm() << std::endl;
}
```

The `Iterator` class provides read/write access while the `ConstIterator` provides read-only access. The type names are accessed via the `::`-operator from the scope of the vector class.

A uniform naming scheme enables writing of generic algorithms. The following types are provided in the scope of any vector class:

2.4 Operations

A full list of all members of a vector class is given in Table 1. The norms are the same as defined for the sparse BLAS standard [5]. The "real" variants avoid the evaluation of a square root for each component in case of complex vectors. The `allocator_type` member type is explained below in the section on memory management.

expression	**return type**	**note**
`X::field_type`	T	T is assignable
`X::block_type`	T	T is assignable
`X::allocator_type`	T	see mem. mgt.
`X::blocklevel`	`int`	block levels inside
`X::Iterator`	T	read/write access
`X::ConstIterator`	T	read-only access
`X::X()`		empty vector
`X::X(X&)`		deep copy
`X::~X()`		free memory
`X::operator=(X&)`	`X&`	
`X::operator=(field_type&)`	`X&`	from scalar
`X::operator[](int)`	`field_type&`	
`X::operator[](int)`	`const field_type&`	
`X::begin()`	`Iterator`	
`X::end()`	`Iterator`	
`X::rbegin()`	`Iterator`	for reverse iteration
`X::rend()`	`Iterator`	
`X::find(int)`	`Iterator`	
`X::operator+=(X&)`	`X&`	$x = x + y$
`X::operator-=(X&)`	`X&`	$x = x - y$
`X::operator*=(field_type&)`	`X&`	$x = \alpha x$
`X::operator/=(field_type&)`	`X&`	$x = \alpha^{-1} x$
`X::axpy(field_type&,X&)`	`X&`	$x = x + \alpha y$
`X::operator*(X&)`	`field_type`	$x \cdot y$
`X::one_norm()`	`double`	$\sum_i \sqrt{Re(x_i)^2 + Im(x_i)^2}$
`X::one_norm_real()`	`double`	$\sum_i (\lvert Re(x_i)\rvert + \lvert Im(x_i)\rvert)$
`X::two_norm()`	`double`	$\sqrt{\sum_i (Re(x_i)^2 + Im(x_i)^2)}$
`X::two_norm2()`	`double`	$\sum_i (Re(x_i)^2 + Im(x_i)^2)$
`X::infinity_norm()`	`double`	$\max_i \sqrt{Re(x_i)^2 + Im(x_i)^2}$
`X::infinity_norm_real()`	`double`	$\max_i (\lvert Re(x_i)\rvert + \lvert Im(x_i)\rvert)$
`X::N()`	`int`	number of blocks
`X::dim()`	`int`	dimension of space

Table 1: Members of a class `X` conforming to the vector interface.

2.5 Object memory model and memory management

The memory model for all ISTL objects is deep copy as in the Standard Template Library and in contrast to the Matrix Template Library. Therefore, references must be used to avoid excessive copying of objects. On the other hand temporary vectors with appropriate structure can be generated simply with the copy constructor.

2.6 Vector creation

3 Matrices

3.1 Linear mappings

3.2 Matrix classes

For a matrix representing a linear map (or homomorphism) $A : V \mapsto W$ from vector space V to vector space W the recursive block structure of the matrix rows and columns immediatly follows from the recursive block structure of the vectors representing the domain and range of the mapping, respectively. As a natural consequence we designed the following matrix classes:

FieldMatrix. the `template<class K, int n> FieldMatrix<K,n,m>` class template is used to represent a linear map $M : V_1 \to V_2$ where $V_1 = \mathbb{K}^n$ and $V_2 = \mathbb{K}^m$ are vector spaces over the field given by template parameter `K`. `K` may be `double`, `float`, `complex<double>` or any other numeric type. The dimensions of the two vector spaces given by the template parameters `n` and `m` are assumed to be small. The matrix is stored as a dense matrix. Example: Use `FieldMatrix<double,2,3>` to define a linear map from a vector space over doubles with dimension 2 to one with dimension 3.

BCRSMatrix. The `template<class B> BCRSMatrix` class template represents a sparse matrix where the "block type" B is given by the template parameter `B`. `B` may be any other class implementing the matrix interface. The matrix class uses a compressed row storage scheme.

VariableBCRSMatrix. The `template<class B> VariableBCRSMatrix<B>` class can be used to construct a linear map between two vector spaces having a two-level block structure $V = B^{n_1} \times B^{n_2} \times \ldots \times B^{n_m}$ and $W = B^{m_1} \times B^{m_2} \times \ldots \times B^{m_k}$. Both are represented by the `template<class B> VariableBlockVector<B>` class, see 2.2. This is not implemented yet.

3.3 Matrices are containers of containers

Matrices are containers over the matrix rows. The matrix rows are containers over the type `K` or `B` in the sense of the Standard Template Library. Random access is provided via `operator[](int i)` on the matrix to the matrix rows and on the matrix rows to the matrix columns (if present). Note that except for `FieldMatrix`, which is a dense matrix, `operator[]` on the matrix row triggers a binary search for the column.

For sequential access use `RowIterator` and `ColIterator` for read/write access or `ConstRowIterator` and `ConstColIterator` for readonly access to rows and columns, respectively. Here is a small example that prints the sparsity pattern of a matrix of type `M`:

```
typedef typename M::ConstRowIterator RowI;
typedef typename M::ConstColIterator ColI;
for(RowI row = matrix.begin(); row != matrix.end(); ++row){
  std::cout << "row␣"<<row.index()<<":␣"
  for(ColI col = row->begin(); col != row->end(); ++col)
    std::cout<<col.index()<<"␣";
  std::cout<<std::endl;
}
```

3.4 Precision control

3.5 Operations

As with the vector interface a uniform naming convention enables generic algorithms. See Table 2 for a complete list of names.

3.6 Matrix creation

4 Algorithms

4.1 Input/output

4.2 Block recursion

The basic feature of the concept described by the matrix and vector classes, is their recursive block structure. Let A be a matrix with blocklevel $l > 1$ then each block A_{ij} can be treated as (or actually is) a matrix itself. This recursiveness can be exploited in generic algorithm using the defined `block_level` of the matrix and vector classes.

Most preconditioner can be modified to honor this recursive structure for a specific number of block levels k. They then work as normal on the offdiagonal blocks, treating them as traditional matrix entries. For the diagonal values a special procedure applies: If $k > 1$ the diagonal is treated as a matrix itself and the preconditioner is applied recursively on the matrix representing the diagonal value $D = A_{ii}$ with blocklevel $k - 1$. For the case that $k = 1$ the diagonal is treated as a matrix entry resulting in a linear solve or an identity operation depending on the algorithm.

4.3 Triangular solves

In the formulation of most iterative methods upper and lower triangular and diagonal solves play an important role. ISTL provides block recursive versions of these generic building blocks using template metaprogramming, see Table 3 for a listing of these methods. In the table matrix A is decomposed into $A = L + D + U$, where L is a strictly lower block triangular, D is a block diagonal and U is a strictly upper block triangular matrix. An arbitrary block recursion level can be given by an additional parameter. If this parameter is omitted it defaults to 1.

4.4 Simple iterative solvers

Using the same block recursive template metaprogramming technique, kernels for the defect formulations of simple iterative solvers are available in ISTL. The number of block recursion levels can again be given as an additional argument. See Table 4 for a list of these kernels.

4.5 Sparse LU decomposition

5 Solver Interface

The solvers in ISTL do not work on matrices directly. Instead we use an abstract Operator concept. Thus we can even model and solve linear maps that are not stored as matrices (e. g. on the fly computed linear operators).

5.1 Operators

The base class `template<class X, class Y> LinearOperator` represents linear maps. The template parameter `X` is the type of the domain and `Y` is the type of the range of the operator. A linear operator provides the methods `apply(const X& x, Y& y)` and apply `applyscaledadd(const X& x, Y& y)` performing the operations $y = A(x)$ and $y = y + \alpha A(x)$, respectively. The subclass `template<class M, class X, class Y> AssembledLinearOperator` represents linear operators that have a matrix representation. Convertion from any matrix into a linear operator is done by the class `template< class M, class X, class Y> MatrixAdapter`.

5.2 Scalarproducts

For convergence tests and the stopping criteria Krylow methods need to compute scalar products and norms on the underlying vector spaces. The base class `template<class X> Scalarproduct` provides methods `field_type dot(const X& x, const X&y)` and `double norm(const X& x)` to calculate these. For sequential programs use `template< class X> SeqScalarProduct` which simply maps this to functions of the vector implementations.

5.3 Preconditioners

The `template<class X, class Y> Preconditioner` provides the abstract base class for all precondioners in ISTL. The method `void pre(X& x, Y& b)` has to be called before applying the preconditioner. Here `x` is the left hand side and `b` is the right hand side of the operator equation. The method may, e. g. scale the system, allocate memory or compute an (I)LU decomposition. The method `void apply(X& v, const Y&)` applies one step of the preconditioner to the system $A(\vec{v}) = \vec{d}$. Here `b` should contain the current defect and `v` should be 0. Upon exit of the method `v` contains the computed update to the current guess, i. e. $\vec{v} = M^{-1}\vec{d}$ where M is the approximate inverse of the operator A characterizing the preconditioner. The method `void post(X& x)` should be called after all computations to give the precondtioner the chance to clean allocated resources.

See Table 5 for a list of available preconditioner. They have the template parameters `M` representing the type of the matrix they work on, `X` representing the type of the domain, `Y` representing the type of the range of the linear system. The block recursive preconditioner are marked with "x" in the last column. For them the recursion depth is specified via an additional template parameter `int l`. The column labeled "s/p" specifies whether they support **s**equential and/or **p**arallel mode.

5.4 Solvers

All solvers are subclasses of the abstract base class `template<class X, class Y> InverseOperator` representing the inverse of an operator from the domain of type `X` to

the range of type `Y`. The actual solve of the system $A(\vec{x}) = \vec{b}$ is done in the method `void apply(X& x, Y& b, InverseOperatorResult& r)`. In the `InverseOperatorResult` some statistics about the solution process, e. g. iteration count, achieved defect reduction, etc., are stored. All solvers only use methods of instances of `LinearOperator`, `ScalarProduct` and `Preconditioner`. These are provided in the constructor.

See Table 6 for a list of available solvers. All solvers are template classes with a template parameter `X` providing them with the vector implementation used.

5.5 Parallel Solvers

Instead of using parallel data structures (matrices and vectors) that (implicitly) know the data distribution and communication patterns like in PETSc [13, 1] we decided to decouple the parallelization from the data structures used. Basically we provide an abstract consistency model on top of our linear algebra. This is hidden in the parallel implementations of the interfaces of `LinearOperator`, `Scalarproduct` and `Preconditioner`, which assure consistency of the data (by communication) for the `InverseOperator` implementation. Therefore the same Krylow method algorithms work in parallel and sequential mode.

Based on the idea proposed in [9] we implemented parallel overlapping Schwarz preconditioners with inexact (sequential) subdomain solvers and a parallel algebraic multigrid preconditioner together with appropriate implementations of `LinearOperator` and `Scalarproduct`. Nonoverlapping versions are currently being worked on.

Note that using this approach it easy two switch form the currently implemented MPI version to new parallel programming paradigms that might be needed on new platforms.

6 Performance

We evaluated the performance of our implementation on a Petium 4 Mobile 2.4 GHz with a measured memory bandwith of 1084 MB/s for the daypy operation ($x = y + \alpha z$) in Tables 7. The code was comiled with the GNU C++ compiler version 4.0 with -O3 optimization. In the tables N is the number of unknown blocks (equals the number of unknows for the scalar cases in Tables 7(a), 7(b), 7(d)). The performance for the scalarproduct, see Table 7(a), and the daxpy operation, see Table 7(b) is nearly optimal and for large N the limiting factor is clearly the memory bandwith. Table 7(c) shows that we take advantage of cache reusage for matrices of dense blocks with block size $b > 1$. In Table 7(d) we compared the generic implementation of the Gauss Seidel solver in ISTL with a specialized C implementation. The measured times per iteration show that there is now lack of computational efficiency due to the generic implementation.

References

[1] S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.

[2] J. J. Barton and L. R. Nackman. *Scientific and Engineering C++*. Addison-Wesley, 1994.

[3] P. Bastian, M. Droske, C. Engwer, R. Klöfkorn, T. Neubauer, M. Ohlberger, and M. Rumpf. Towards a unified framework for scientific computing. In R. Kornhuber, R.H.W. Hoppe, D.E. Keyes, J. Périaux, O. Pironneau, and J. Xu, editors,

Proceedings of the 15th Conference on Domain Decomposition Methods, LNCSE. Springer-Verlag, 2004. accepted for publication.

[4] P. Bastian and R. Helmig. Efficient fully-coupled solution techniques for two-phase flow in porous media. Parallel multigrid solution and large scale computations. *Adv. Water Res.*, 23:199–216, 1999.

[5] BLAST Forum. Basic linear algebra subprograms technical (BLAST) forum standard, 2001. `http://www.netlib.org/blas/blast-forum/`.

[6] Blitz++. `http://www.oonumerics.org/blitz/`.

[7] Jack Dongarra. List of freely available software for linear algebra on the web, 2006. `http://netlib.org/utk/people/JackDongarra/la-sw.html`.

[8] DUNE. `http://www.dune-project.org/`.

[9] G. Haase, U. Langer, and A. Meyer. The approximate dirichlet domain decomposition method. part i: An algebraic approach. *Computing*, 47:137–151, 1991.

[10] J. Hefferson. Linear algebra, May 2006. `http://joshua.amcvt.edu/`.

[11] Iterative template library. `http://www.osl.iu.edu/research/itl/`.

[12] Matrix template library. `http://www.osl.iu.edu/research/mtl/`.

[13] PETSc. `http://www.mcs.anl.gov/petsc/`.

[14] J. Siek and A. Lumsdaine. A modern framework for portable high-performance numerical linear algebra. In H. P. Langtangen, A. M. Bruaset, and E. Quak, editors, *Advances in Software Tools for Scientific Computing*, volume 10 of *LNCSE*, pages 1–56. Springer-Verlag, 2000.

[15] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1997.

[16] T. Veldhuizen. Techniques for scientific C++. Technical report, Indiana University, 1999. Computer Science Department.

expression	return type	note
M::field_type	T	T is assignable
M::block_type	T	T is assignable
M::row_type	T	a T is assignable
M::allocator_type	T	see mem. mgt.
M::blocklevel	int	block levels inside
M::RowIterator	T	over rows
M::ColIterator	T	over columns
M::ConstRowIterator	T	over rows
M::ConstColIterator	T	over columns
M::M()		empty matrix
M::M(M&)		deep copy
M::∼M()		free memory
M::operator=(M&)	M&	
M::operator=(field_type&)	M&	from scalar
M::operator[](int)	row_type&	
M::operator[](int)	const row_type&	
M::begin()	RowIterator	
M::end()	RowIterator	
M::rbegin()	RowIterator	reverse iteration
M::rend()	RowIterator	
M::operator*=(field_type&)	M&	$A = \alpha A$
M::operator/=(field_type&)	M&	$A = \alpha^{-1} A$
M::umv(X& x,Y& y)		$y = y + Ax$
M::mmv(X& x,Y& y)		$y = y - Ax$
M::usmv(field_type&,X& x,Y& y)		$y = y + \alpha Ax$
M::umtv(X& x,Y& y)		$y = y + A^T x$
M::mmtv(X& x,Y& y)		$y = y - A^T x$
M::usmtv(field_type&,X& x,Y& y)		$y = y + \alpha A^T x$
M::umhv(X& x,Y& y)		$y = y + A^H x$
M::mmhv(X& x,Y& y)		$y = y - A^H x$
M::usmhv(field_type&,X& x,Y& y)		$y = y + \alpha A^H x$
M::solve(X& x,Y& b)		$x = A^{-1} b$
M::inverse(M& B)		$B = A^{-1}$
M::leftmultiply(M& B)	M&	$A = BA$
M::frobenius_norm()	double	see text
M::frobenius_norm2()	double	see text
X::infinity_norm()	double	see text
X::infinity_norm_real()	double	see text
M::N()	int	row blocks
M::M()	int	col blocks
M::rowdim(int)	int	dim. of row block
M::rowdim()	int	dim. of row space
M::coldim(int)	int	dim. of col block
M::coldim()	int	dim. of col space
M::exists(int i, int j)	bool	

Table 2: Members of a class M conforming to the matrix interface. X and Y are any vector classes.

function	computation
`bltsolve(A,v,d)`	$v = (L+D)^{-1}d$
`bltsolve(A,v,d,`ω`)`	$v = \omega(L+D)^{-1}d$
`ubltsolve(A,v,d)`	$v = L^{-1}d$
`ubltsolve(A,v,d,`ω`)`	$v = \omega L^{-1}d$
`butsolve(A,v,d)`	$v = (D+U)^{-1}d$
`butsolve(A,v,d,`ω`)`	$v = \omega(D+U)^{-1}d$
`ubutsolve(A,v,d)`	$v = U^{-1}d$
`ubutsolve(A,v,d,`ω`)`	$v = \omega U^{-1}d$
`bdsolve(A,v,d)`	$v = D^{-1}d$
`bdsolve(A,v,d,`ω`)`	$v = \omega D^{-1}d$

Table 3: Functions available for block triangular and block diagonal solves. The matrix A is decomposed into $A = L + D + U$ where L is strictly lower block triangular, D is block diagonal and U is strictly upper block triangular. Standard is one level of block recursion, arbitrary level can be given by additional parameter.

function	computation
`dbjac(A,x,b,`ω`)`	$x = x + \omega D^{-1}(b - Ax)$
`dbgs(A,x,b,`ω`)`	$x = x + \omega(L+D)^{-1}(b - Ax)$
`bsorf(A,x,b,`ω`)`	$x_i^{k+1} = x_i^k + \omega A_{ii}^{-1}\left[b_i - \sum_{j<i} A_{ij}x_j^{k+1} - \sum_{j\geq i} A_{ij}x_j^k\right]$
`bsorb(A,x,b,`ω`)`	$x_i^{k+1} = x_i^k + \omega A_{ii}^{-1}\left[b_i - \sum_{j\leq i} A_{ij}x_j^k - \sum_{j>i} A_{ij}x_j^{k+1}\right]$

Table 4: Kernels for iterative solvers. The matrix A is decomposed into $A = L + D + U$ where L is strictly lower block triangular, D is block diagonal and U is strictly upper block triangular. Standard is one level of block recursion, arbitrary level can be given by additional parameter.

Table 5: Preconditioners

class	implements	s/p	recursive
`SeqJac`	Jacobi method	s	x
`SeqSOR`	successive overrelaxation (SOR)	s	x
`SeqSSOR`	symmetric SSOR	s	x
`SeqILU`	incomplete LU decomposition (ILU)	s	
`SeqILUN`	ILU decpmposition of order N	s	
`Pamg::AMG`	algebraic multigrid method	s/p	
`BlockPreconditioner`	Additive overlapping Schwarz	p	

Table 6: ISTL Solvers

class	implements
`LoopSolver`	only apply precoditioner multiple time
`GradientSolver`	preconditioned radient method
`CGSolver`	preconditioned conjugate gradient method
`BiCGStab`	preconditioned biconjugate gradient stabilized method

Table 7: Performance Tests

(a) scalar product

N	500	5000	50000	500000	5000000
MFLOPS	896	775	167	160	164

(b) daxpy operation $y = y + \alpha x$

	500	5000	50000	500000	5000000
MFLOPS	936	910	108	103	107

(c) Matrix-vector product, 5-point stencil, b: block size

N, b	100,1	10000,1	1000000,1	1000000,2	1000000,3
MFLOPS	388	140	136	230	260

(d) Damped Gaußß-Seidel

	C	ISTL
time / it. [s]	0.17	0.18