

The Iterative Solver Template Library

Markus Blatt and Peter Bastian

Institut für Parallele und Verteilte Systeme,
Universität Stuttgart, Universitätstr. 38, 70569 Stuttgart, Germany
{Markus.Blatt, Peter.Bastian}@ipvs.uni-stuttgart.de
<http://www.dune-project.org>

Abstract. The numerical solution of partial differential equations frequently requires the solution of large and sparse linear systems. Using generic programming techniques in C++ one can create solver libraries that allow efficient realization of “fine grained interfaces”, i.e. with functions consisting only of a few lines, like access to individual matrix entries. This prevents code replication and allows programmers to work more efficiently. We present the “Iterative Solver Template Library” (ISTL) which is part of the “Distributed and Unified Numerics Environment” (DUNE). It applies generic programming in C++ to the domain of iterative solvers of linear systems stemming from finite element discretizations. Those discretizations exhibit a lot of structure. Our matrix and vector interface supports a block recursive structure. Each sparse matrix entry can itself be either a sparse or a small dense matrix. Based on this interface we present efficient solvers that use the recursive block structure via template metaprogramming.

1 Introduction

The numerical solution of partial differential equations (PDEs) frequently requires solving large and sparse linear systems. Naturally, there are many libraries available for doing sparse matrix/vector computations, see [7] for a comprehensive list.

The widely available Basic Linear Algebra Subprograms (BLAS) standard has been extended to cover also sparse matrices [5]. The standard uses procedural programming style and offers only a FORTRAN and C interface. “Fine grained” interfaces, i.e. with functions consisting only of a few lines of code, such as access to individual matrix elements, impose an efficiency penalty here, as the relative cost for indirect function calls becomes huge.

Generic programming techniques in C++ or Ada offer the possibility to combine flexibility and reuse (“efficiency of the programmer”) with fast execution (“efficiency of the program”). They allow the compiler to apply optimizations even for “fine grained” interfaces via static function typing. These techniques were pioneered by the Standard Template Library (STL), [16]. Their efficiency advantage for scientific C++ was later demonstrated by the Blitz++ library [6]. For an introduction to generic programming for scientific computing see [2,17].

Application of these ideas to matrix/vector operations is available with the Matrix Template Library (MTL), [13,15] and to iterative solvers for linear systems with the Iterative Template Library (ITL), [12].

In contrast to these libraries the “Iterative Solver Template Library” (ISTL), which is part of the “Distributed and Unified Numerics Environment” (DUNE), [3,8], is designed specifically for linear systems stemming from finite element discretizations. The sparse matrices representing these linear systems exhibit a lot of structure, e.g.:

- 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). It is straightforward and efficient to treat these small dense blocks as fully coupled and solve them with direct methods within the iterative method, 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 and sparse. As an example we mention the Stokes system, see Fig. 1(d). 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 hierarchical meshes, a *p*-hierarchical 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.

Our library takes advantage from this natural block structure at compile time and supports the recursive block structuredness in a natural way.

Other libraries, like MTL, provide the blockings as views to the programmer. As this is done dynamically the block structure cannot be used efficiently in custom generic algorithms. In the Optimized Sparse Kernel Interface (OSKI), see [18], the sparse matrices are stored as scalar matrices, too. Here the user can

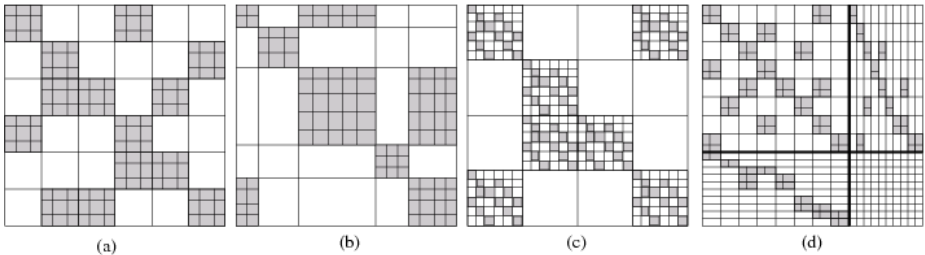


Fig. 1. Block structure of matrices arising in the finite element method

provide hints about the dense block sizes which are used at runtime to tune the solvers.

In the next section we describe the matrix and vector interface that represents this recursive block structure via templates. In Sect. 3 we show how to exploit the block structure using template metaprogramming at compile time. Finally we sketch the high level iterative solver interface in Sect. 4.

2 Matrix and Vector Interface

The interface of our matrices is designed according to what they represent from a mathematical point of view. The vector classes are representations of vector spaces while the matrix classes are representations of linear maps between two vector spaces.

2.1 Vector Spaces

We assume the reader is familiar with the concept of vector spaces. Essentially a vector space over a field \mathbb{K} is a set V of elements (called vectors) along with vector addition $+: V \mapsto V$ and scalar multiplication $\cdot: \mathbb{K} \times V \mapsto V$ with the well known properties. See your favourite textbook for details, e.g. [11].

For our application the following way of construction plays an important role: Let V_i , $i = 1, 2, \dots, n$, be a normed vector space of dimension n_i with a scalar product, then the n -ary Cartesian product

$$V := V_1 \times V_2 \times \dots \times V_n = \{(\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n) | \mathbf{v}_1 \in V_1, \mathbf{v}_2 \in V_2, \dots, \mathbf{v}_n \in V_n\} \quad (1)$$

is again a normed vector space of dimension $\sum_{i=1}^n n_i$ with the canonical norm and scalar product.

Treating \mathbb{K} as a vector space itself we can apply this construction recursively starting from the field \mathbb{K} .

While for a mathematician every finite dimensional vector space is isomorphic to \mathbb{R}^k for an appropriate k , for our application it is important to know how the vector space was constructed recursively by the procedure described in (1).

Vector Classes. To express the construction of the vector space by n -ary products of other vector spaces ISTL provides the following classes:

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.

Example: Use `FieldVector<double,2>` for vectors with a fixed dimension 2.

BlockVector. The `template<class B> BlockVector` class template builds a vector space $V = B^n$ where the “block type” B is given by the template parameter

Table 1. Types of vector classes

expression	return type
<code>field_type</code>	The type of the field of the represented vector space, e.g. <code>double</code> .
<code>block_type</code>	The type of the vector blocks.
<code>size_type</code>	The type used for the index access and size operations.
<code>block_level</code>	The block level of the vector, e.g. 1 for <code>FieldVector</code> , 2 for <code>BlockVector<FieldVector<K>,n></code> .
<code>Iterator</code>	The type of the iterator.
<code>ConstIterator</code>	The type of the immutable iterator.

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` class can be used to construct a vector space having a two-level block structure of the form $V = B^{n_1} \times B^{n_2} \times \dots \times B^{n_m}$, i.e. it consists of m blocks $i = 1, \dots, m$ 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. For certain operations it is more efficient to interpret the vector space as $V = B^N$, where $N = \sum_{i=1}^m n_i$.

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 $0, \dots, n-1$ 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[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. The `Iterator` class provides read/write access while the `ConstIterator` class 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. See Table 1 for the types provided in the scope of any vector class.

2.2 Linear Maps

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 immediately 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:

Matrix classes. Using the construction in (1) the structure of our vector spaces carries over to linear maps in a natural way.

FieldMatrix. The `template<class K, int n> FieldMatrix<K,n,m>` class template is used to represent a linear map $M : V_1 \rightarrow V_2$ 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` class can be used to construct a linear map between two vector spaces having a two-level block structure $V = B^{n_1} \times B^{n_2} \times \dots \times B^{n_k}$ and $W = B^{m_1} \times B^{m_2} \times \dots \times B^{m_l}$. Both are represented by the `template<class B> VariableBlockVector` class, see Sect. 2.1. This is not implemented yet.

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 read-only access to rows and columns, respectively.

As with the vector interface a uniform naming convention enables generic algorithms. See Table 2 for the most important names.

3 Block Recursive Algorithms

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

Table 2. Type names in the matrix classes

expression	return type
field_type	The type of the field of the vector spaces we map from and to
block_type	The type representing the matrix components
row_type	The container type of the rows.
size_type	The type used for index access and size operations
block_level	The block recursion level, e.g. 1 for <code>FieldMatrix</code> and 2 for <code>BCSRMatrix<FieldMatrix<K,m,n> ></code> .
RowIterator	The type of the mutable iterator over the rows
ConstRowIterator	Ditto, but immutable
ColIterator	The type of the mutable iterator over the columns of a row.
ConstColIterator	Ditto, but immutable

Note that we do not use recursive blocked algorithms on the dense matrix blocks, as described in [9], as the dense blocks resulting from finite element discretizations will generally be small.

Most preconditioners 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 block level $k - 1$. 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.

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. $d = b - Ax$ denotes the current residual used to calculate the update v to the current guess x . An arbitrary block recursion level can be given by an additional parameter. If this parameter is omitted it defaults to 1.

Using the same block recursive template metaprogramming technique, kernels for the residual formulations of simple iterative solvers are available in ISTL. The number of block recursion levels can again be given as an additional argument. See the second part of Table 3 for a list of these kernels.

4 Solver Interface

The solvers in ISTL do not work on matrices directly. Instead we use an abstract operator concept. This allows for using matrix-free operators, i.e. operators that are not stored as matrices in any form. Thus our solver algorithms can easily be

Table 3. Iterative Solver Kernels

function	computation
block triangular and block diagonal solves	
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)	$v = \omega D^{-1}d$
iterative solves	
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)	$x_i^{k+1} = x_i^k + \omega A_{ii}^{-1} \left[b_i - \sum_{j \leq i} A_{ij} x_j^k - \sum_{j > i} A_{ij} x_j^{k+1} \right]$

turned into matrix-free solvers just by plugging in matrix-free representations of linear operators and preconditioners.

4.1 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 `appliescaledadd(field_type alpha, 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. Conversion from any matrix into a linear operator is done by the class `template<class M, class X, class Y> MatrixAdapter`.

4.2 Scalar Products

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.

Table 4. 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 decomposition of order N	s	
Pamg::AMG	algebraic multigrid method	s/p	
BlockPreconditioner	Additive overlapping Schwarz	p	

4.3 Preconditioners

The `template<class X, class Y> Preconditioner` provides the abstract base class for all preconditioners 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& d)` applies one step of the preconditioner to the system $A(\mathbf{v}) = \mathbf{d}$. Here `d` should contain the current residual and `v` should use 0 as the initial guess. Upon exit of the method `v` contains the computed update to the current guess, i.e. $\mathbf{v} = M^{-1}\mathbf{d}$ where M is the approximate of the operator A characterizing the preconditioner. The method `void post(X& x)` should be called after all computations to give the preconditioner the chance to clean up allocated resources.

See Table 4 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 preconditioners are marked with “x” in the last column. For them the recursion depth is specified via an additional template parameter `int` 1. The column labeled “s/p” specifies whether they support sequential and/or parallel mode.

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

See Table 5 for a list of available solvers. All solvers are template classes with a template parameter `X` providing them with the vector implementation used.

Table 5. ISTL Solvers

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

Table 6. Performance Tests

(a) scalar product						(b) daxpy operation $y = y + \alpha x$				
N	500	5000	50000	500000	5000000	500	5000	50000	500000	5000000
MFLOPS	896	775	167	160	164	936	910	108	103	107

(c) Matrix-vector product, 5-point stencil, b : block size						(d) Damped Gauß-Seidel ($N = 10^6, b = 1$)	
N, b	100,1	10000,1	1000000,1	1000000,2	1000000,3	C	ISTL
MFLOPS	388	140	136	230	260	time / it. [s]	0.17 0.18

4.5 Parallel Solvers

Instead of using parallel data structures (matrices and vectors) that (implicitly) know the data distribution and communication patterns like in PETSc [14,1] 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 [10] 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 is easy to switch from the currently implemented MPI version to new parallel programming paradigms that might be needed on new platforms.

4.6 Performance Evaluation

We evaluated the performance of our implementation on a Pentium 4 Mobile 2.4 GHz processor with a measured memory bandwidth of 1084 MB/s for the daxpy operation ($x = y + \alpha z$) in Tables 6. The code was compiled 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 unknowns for the scalar cases

in Tables 6(a), 6(b), 6(d), b is the size of the dense blocks. All matrices are sparse matrices of dense blocks. The performance for the scalar product, see Table 6(a), and the daxpy operation, see Table 6(b) is nearly optimal and for large N the limiting factor is clearly the memory bandwidth. Table 6(c) shows that we take advantage of cache reuse for matrices of dense blocks with block size $b > 1$. In Table 6(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 no significant lack of computational efficiency due to the generic implementation.

References

1. Balay, S., Buschelman, K., Eijkhout, V., Gropp, W.D., Kaushik, D., Knepley, M.G., McInnes, L.C., Smith, B.F., Zhang, H.: PETSc users manual. Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory (2004)
2. Barton, J.J., Nackman, L.R.: Scientific and Engineering C++. Addison-Wesley, London (1994)
3. Bastian, P., Droske, M., Engwer, C., Klöforn, R., Neubauer, T., Ohlberger, M., Rumpf, M.: Towards a unified framework for scientific computing. In: Kornhuber, R., Hoppe, R., Périaux, J., Widlund, O., Pironneau, O., Xu, J. (eds.) Domain Decomposition Methods in Science and Engineering. LNCSE, vol. 40, pp. 167–174. Springer, Heidelberg (2005)
4. Bastian, P., Helmig, R.: 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. Dongarra, J.: 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. Elmroth, E., Gustavson, F., Jonsson, I., Kagström, B.: Recursive blocked algorithms and hybrid data structures for dense matrix library software. SIAM Review 46(1), 3–45 (2004)
10. Haase, G., Langer, U., Meyer, A.: The approximate dirichlet domain decomposition method. part i: An algebraic approach. Computing 47, 137–151 (1991)
11. Hefferson, J.: Linear algebra. in the web (May 2006), <http://joshua.amcvt.edu/>
12. Iterative template library, <http://www.osl.iu.edu/research/itl/>
13. Matrix template library, <http://www.osl.iu.edu/research/mtl/>
14. PETSc, <http://www.mcs.anl.gov/petsc/>
15. Siek, J., Lumsdaine, A.: A modern framework for portable high-performance numerical linear algebra. In: Langtangen, H.P., Bruaset, A.M., Quak, E. (eds.) Advances in Software Tools for Scientific Computing, LNCSE, vol. 10, pp. 1–56. Springer, Heidelberg (1974)
16. Stroustrup, B.: The C++ Programming Language. Addison-Wesley, London (1997)
17. Veldhuizen, T.: Techniques for scientific C++. Technical report, Indiana University, Computer Science Department (1999)
18. Vuduc, R., Demmel, J.W., Yelick, K.A.: Oski: A library of automatically tuned sparse matrix kernels. Journal of Physics Conference Series 16, 521–530 (2005)