LinBox: A Generic Library for Exact Linear Algebra*

J.-G. Dumas[1], T. Gautier[2], M. Giesbrecht[3], P. Giorgi[6], B. Hovinen[4], E. Kaltofen[5], B.D. Saunders[4], W.J. Turner[5] and G. Villard[6]

[1]www-lmc.imag.fr/lmc-mosaic/Jean-Guillaume.Dumas [2]www-id.imag.fr/~gautier [3]www.uwaterloo.ca/~mwg [4]www.cis.udel.edu/{~hovinen,~saunders} [5]www.math.ncsu.edu/{~kaltofen,~wjturner} [6]www.ens-lyon.fr/{~pgiorgi,~gvillard}

1 Introduction

Black box techniques [12] are enabling exact linear algebra computations of a scale well beyond anything previously possible. The development of new and interesting algorithms has proceeded apace for the past two decades. It is time for the dissemination of these algorithms in an easily used software library so that the mathematical community may readily take advantage of their power. LinBox is that library. In this paper, we describe the design of this generic library, sketch its current range of capabilities, and give several examples of its use. The examples include a solution of Trefethen's "Hundred Digit Challenge" problem #7 [15] and the computation of all the homology groups of simplicial complexes using the Smith normal form [14].

Exact black box methods are currently successful on sparse matrices with hundreds of thousands of rows and columns and having several million nonzero entries. The main reason large problems can be solved by black box methods is that they require much less memory in general than traditional elimination-based methods do. This fact is widely used in the numerical computation area. We refer for instance to the templates for linear system solution and eigenvalue problems [2, 1]. This has also led the computer algebra community to a considerable interest in black box methods. Since Wiedemann's seminal paper [17], many developments have been proposed especially to adapt Krylov or Lanczos methods to fast exact algorithms. We refer to [5] and references therein for a review of problems and solutions.

LinBox supplies efficient black box solutions for a variety of problems including linear equations and matrix normal forms with the guiding design principle of re-usability. The most essential and driving design criterion for LinBox is that it is generic with respect to the domain of computation. This is because there are many and various representations of finite fields each of which is advantageous to use for some algorithm under some circumstance. The integral and rational number capabilities depend heavily on modular techniques and hence on the

*This material is based on work supported in part by NSF (USA) under grants Nrs. DMS-9977392, CCR-9988177, and ITR/ASC-0113121 (Kaltofen) Nrs. CCR-0098284 and ITR/ASC-0112807 (Saunders) and by CNRS (France) Actions Incitatives No 5929 et Stic LinBox 2001 (Villard) and by NSERC (Canada, Giesbrecht).

capabilities over finite fields. In this regard, generic software methodology is a powerful tool.

Partly modeled on the STL, LinBox uses the C++ template mechanism as the primary tool to achieve the genericity. The library is inspired by the FoxBox black box and plug-and-play design objectives [6]. Projects with some similar goals include MTL [`http://www.osl.iu.edu/research/mtl`] in numerical linear algebra and SYNAPS [`http://www-sop.inria.fr/galaad/logiciels/synaps`] in symbolic computation. A forerunner of LinBox is described in [10].

The following section presents design decisions and their motivation using the field and black box representations as examples. In Section 3 we discuss the current capabilities provided in LinBox and guiding principles for the implementation of their underlying algorithms. In Section 4 we illustrate the power of LinBox with some example solutions.

2 LinBox design

The overarching goal of our design is to create a software library that supports reuse and reconfiguration at a number of levels without sacrificing performance. At the top level, we provide algorithms for many standard problems in linear algebra. As input, these algorithms accept black box matrices, a notion that uniformly captures sparse and structured (and dense) matrices alike. Any object conformant with the specification for a black box matrix can be plugged into these algorithms. At a lower level, we want our code to operate over a multitude of coefficient domains and, for a given domain, a variety of implementations. For instance, into our algorithms one might plug any of several implementations of the integers modulo a prime number, e.g., when the field operations may be performed via a Zech logarithm table or via integer remaindering. We can capture any future improvements on field arithmetic without rewriting our programs. One might also plug in a field of rational functions, or the floating point numbers, although the resulting methods may not be numerically stable. At every stage we have applied the principle commonly called *generic programming*. We realize this through C++ templates and virtual member functions.

LinBox provides what we call *archetype* classes for fields and black box matrices. An archetype serves three purposes: to define the common object interface, to supply one instance of the library as distributable compiled code, and to control code bloat. An archetype is an abstract class whose use is similar to a Java interface. It specifies exactly what methods and members an explicitly designed class must have to be a pluggable template parameter type. Through the use of pointers and virtual member functions, the field archetype, for instance, can hook into different LinBox fields. Thus the precompiled library code can be executed on a user supplied field.

Field design. The algorithms in LinBox are designed to operate with a variety of domains, particularly finite prime fields. To perform the required arithmetic, additional parameters, such as the modulus, must be available to the algorithm. One can store a pointer to the required parameters in each field element, but that would require too much memory. One can also use a global variable to store these parameters—as is done in NTL, for instance—but it is then impossible to operate over more than one field concurrently. Our approach is to have a separate field object whose methods include field arithmetic operations. For example, the call `F.add(x,y,z)` adds the elements `y` and `z` in the field `F` and stores the result in the element `x`. The field object stores the required parameters, and it is passed to each of the generic algorithms. Because of this design, we do not support traditional infix notation for arithmetic.

Given a field class `Field`, elements of this field are of the type `Field::element`. This may be a C++ `long`, for integers modulo a word size prime, or a more complicated data structure declared elsewhere in the library. The field interface requires only that the data type support C++ assignment and a copy constructor. Because elements by themselves do not have access to the field parameters, they are initialized by the field, as in `F.init(x,5)`. The field type contains methods to initialize and perform arithmetic on field elements and check their equality. In addition to standard arithmetic, in which the result is stored in a separate field element, we support "in-place" arithmetic, similar to C++ `+=`, `-=`, `*=`, and `/=`. Field types are also required to support assignment and equality checking of whole fields. For each field type, there exists a class that uniformly generates random elements of that field or an unspecified subset of a given cardinality. Many of the algorithms in LinBox depend on the availability of such random elements (see Section 3).

Whether or not a field requires parameters, such as a modulus, to perform arithmetic, its interface is the same. We provide a template *wrapper* class for the creation of an unparameterized field meeting the LinBox interface from a C++ data type that supports standard arithmetic and equality operations. For example, `unparam_field<ZZ_p> F` is a field of NTL modular integers. If a user-defined field implements a required method in a different manner, one can resort to partial template specialization in order to define the corresponding operation. The following example adjusts Victor Shoup's `inv` function of his `ZZ_p` class to the signature of LinBox's `inv` method.

```
template <> NTL::ZZ_p& unparam_field<NTL::ZZ_p>::inv (
                 NTL::ZZ_p& x, const NTL::ZZ_p& y) const {
                              return x = NTL::inv(y); }
```

Thus we can easily adapt fields from other libraries to LinBox.

The field archetype defines the interface that all field implementations must satisfy. Any class that meets that interface can be hooked into a generic algorithm. A virtual copy method ("clone") is introduced via an abstract base class pointed to by the archetype thus yielding an STL-compliant copy constructor in the archetype. Generic algorithms can be instantiated with the archetype and compiled separately. Code making use of these algorithms can supply a field inheriting the abstract field type and link against this code, with a modest performance loss resulting from the inability to inline field operations and from additional memory indirection. Finally, LinBox provides a template class called `Field_envelope` that hooks any archetype-compliant field type onto this abstract class, so any field type may be used in this manner, then without any performance loss.

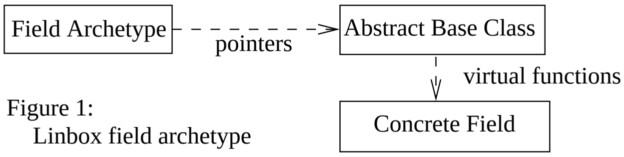

Figure 1:
Linbox field archetype

Black box design. The LinBox black box matrix archetype is simpler than the field archetype because the design constraints are less stringent. As with the field type, we need a common object interface to describe how algorithms are to access black box matrices, but it only requires functions to access the matrix's dimensions and to apply the matrix or its transpose to a vector. Thus our black box matrix archetype is simply an abstract class, and all actual black box matrices are subclasses of the archetype class. We note that the overhead involved with this inheritance mechanism is negligible in comparison with the execution time of the methods, unlike for our field element types.

The black box matrix archetype is template-parameterized by the vector type upon which it acts, but not by the field in which the arithmetic is done as we saw no necessity for this. The field of entries is bound directly to the black box matrix class and is available as an argument to our black box algorithms, which may perform additional coefficient field operations. Optionally, black box matrix classes can have the field type as a template parameter. In addition, variants of the `apply` method are provided through which one could pass additional information, including the field over which to operate.

LinBox currently has three types of vectors, *dense*, *sparse sequence*, and *sparse map*. The dense vectors store every entry of the vector and are generally implemented as `std::vector<element>`. Sparse vectors only store nonzero entries. Sparse sequence vectors have the archetype `std::list<std::pair<integer, element>>`, and sparse map vectors have the archetype `std::map<integer,element>`. The C++ `operator[]` is disallowed in the latter to avoid fill-in with zero values. By its data structure, a map entry access is logarithmic time.

We do not parameterize our algorithms with a black box type. We use the black box archetype directly in the algorithms. The caller provides a specific black box subclass. For convenience, some methods have default implementations in the archetype. For example, `apply` and `applyTranspose` each have three variants, which handle allocation of the input and output vectors differently. Only one variant is necessary:

```
Vector& apply (Vector& y, const Vector& x) const;
Vector& applyTranspose (Vector& y, const Vector& x) const;
```

The archetype base class provides default implementations of the other variants, but a derived class can override them.

Input iterators. The STL separates container and iterator types, thus allowing algorithms on iterators, which can correspond to different container types. In LinBox, we have adopted and extended this idea. We distinguish between the way an input is computed and the way it is accessed by using *input iterators* as input data. These may themselves be an interface to a primary container-like data type (e.g., black boxes). An example is Wiedemann's algorithm for the computation of the minimal polynomial of a matrix (see Section 3). It uses scalar iterates of matrix-vector products and dot products $v^{\mathrm{T}}(A^i u)$ to feed a linear recurring sequence solver such as the Berlekamp-Massey algorithm. The sequence solver is completely independent of the way the scalars are produced. For example, (1) they may all be precomputed and stored in an array; (2) they may be produced "on-the-fly" when the number of iterations is not known in advance; (3) the required matrix-vector products can be parallelized or pipelined (see [10]). We have implemented these solvers with the STL iterator interface and show here an example with asynchronous calls:

```
BlackBox_Container::const_iterator&
        BlackBox_Container::const_iterator::operator++() {
        _c._launch(); return *this; }
const BlackBox_Container::value_type&
        BlackBox_Container::const_iterator::operator*() {
        _c._wait(); return _c.getvalue(); }
void launch() { for(long i=0; i<_size;++i) launch_one(i); }
void wait() { Wait<value_type>() (_dotproducts[_current]); }
```

The Berlekamp-Massey algorithm accesses the sequence via this interator. Thus the outer algorithm is generic with respect to the choice of the methods mentioned above. We now have the advantage to easily switch among parallel, pipelined, or other strategies with the purpose of very effective adaptation to available resources. As with *streams* of the Scheme language, our iterators can fill-in the sequence on demand for later reuse.

3 Black box algorithms

Wiedemann's paper [17] has led to many developments in the application of Krylov and Lanczos methods to fast exact problem solving. In this section we present the main directions followed in LinBox regarding these methods. Linear algebra done over finite fields is the core of the library. Computations over the integers or rationals build upon this core.

Randomized algorithms, heuristics and checking. In black box linear algebra, the fastest known algorithms generally use random bits. Our library includes *Monte Carlo* algorithms (which produce correct results with controllably high probability), *Las Vegas* algorithms (which always produce correct results and are fast with high probability), and deterministic algorithms. Properties of these algorithms are proven under conditions on the source of random bits, e.g., that we are able to choose random elements uniformly from a sufficiently large subset of the ground field. This condition may be prohibitively costly in practice, e.g., if the field is small the use of an algebraic extension may be required. This has led us to a new implementation strategy. We relax theoretical conditions on our algorithms but introduce subsequent checks for correctness. We exploit randomized algorithms as heuristics even when the provable conditions for success are not satisfied. To complement this, specific checks are developed to certify the output. These checks may themselves be randomized (see below), in which case we certify only the probability of success. This strategy has been powerful in obtaining solutions for huge determinant and Smith form problems in Section 4.

Minimal polynomial and linear system solution over finite fields. For a matrix $A \in \mathbb{F}^{n\times n}$ over a field $\mathbb{F}$, Lanczos and Krylov subspace methods essentially compute the minimal polynomial of a vector with respect to A. These methods access A only via matrix-vector products $v = A \cdot u$ and $w = A^{\mathrm{T}} \cdot u$, i.e., they treat A as a black box function. Thus the library will only employ a black box for A, which can exploit the structure or sparsity of A.

The minimal polynomial $f^{A,u}$ of a vector u is computed as a linear dependency between the iterates $u, Au, A^2u, \ldots$. In the Krylov/Wiedemann approach, the dependency is found by applying the Berlekamp-Massey algorithm to the sequence $v^{\mathrm{T}}u, v^{\mathrm{T}}Au, v^{\mathrm{T}}A^2u, \ldots \in \mathbb{F}$, for a random vector v. This identifies the *minimum generating polynomial* $f_v^{A,u}(x) = x^d - f_{d-1}x^{d-1} - \ldots - f_1x - f_0$, i.e., $v^{\mathrm{T}}A^{d+i}u = f_{d-1} \cdot v^{\mathrm{T}}A^{d+i-1}u + \ldots + f_1 \cdot v^{\mathrm{T}}A^{i+1}u + f_0 \cdot v^{\mathrm{T}}A^iu$, for all $i \geq 0$. Then $f_v^{A,u}(x)$ always divides $f^{A,u}$, and they are equal with high probability [17]. Berlekamp-Massey computes $f_v^{A,u}(x)$ after processing the first $2d$ elements of the sequence. Wiedemann's algorithm distinguishes the sequence generation from the computation of the linear recurrence, while these tasks are intermixed in the Lanczos approach. Lambert [13] gives a unifying study of both approaches.

`LinBox::minpoly`: With high probability, the minimal polynomial of the sequence $\{v^{\mathrm{T}}A^iu\}_{0\leq i\leq 2n-1}$ is the minimum polynomial f^A of A [17, 11], for randomly chosen vectors u, v. This algorithm is randomized of the Monte Carlo type. As first observed by Lobo, the cost can be reduced by early termination: as soon as the linear generator computed by the Berlekamp-Massey process remains the same for a few steps, it is likely the minimal polynomial. This argument is heuristic in general but provable for the Lanczos algorithm on preconditioned matrices over suitably large fields [9, also Eberly (private Communication 2000)]. A Monte Carlo check of the early termination is implemented by applying the computed polynomial to a random vector.

Dominant costs in these algorithms are given in Table 1 (from [13], [7, Chap. 6]). Terms between brackets give memory requirements (in field elements). Early termination and randomized Monte Carlo algorithms correspond to bi-orthogonal Lanczos algorithms with or without lookahead. In both approaches, the number of matrix-vector products may be cut in half if the matrix is symmetric. Since the update of the linear generator is computed by dot products instead of elementary polynomial operations, a Lanczos strategy has a slightly higher cost for computing the minimal polynomial.

Table 1: **Costs of Wiedemann and Lanczos algorithms** for f^A of degree d and for $Az = b$. A or A^{T} can be applied to a vector using at most ω operations.

	Early termin. f^A	Monte Carlo f^A	Sys. soln. $Az = b$
Wiedem. $[6n]$	$2d\omega + 4d(n+d)$	$2n\omega + 4n^2 + 2d(n+d)$	$+d\omega + 2dn$
Wiedem. $[O(dn)]$	$2d\omega + 4d(n+d)$	$2n\omega + 4n^2 + 2d(n+d)$	$+2dn$
Lanczos $[3n]$	$2d\omega + 8dn$	$2n\omega + 4n^2 + 4dn$	$+2dn$

`LinBox::linsolve`: For nonsingular A, a linear system $Az = b$ is solved by computing the minimal polynomial $f^{A,b}(x) = x^d + f_{d-1}x^{d-1} + \ldots + f_1x + f_0$ of b; its coefficients directly give $z = -(1/f_0)(A^{d-1}b + f_{d-1}A^{d-2}b + \ldots + f_1b)$.

Checking $Az = b$ makes the system solution Las Vegas. The Lanczos approach allows one to compute z within the iterations for the minimal polynomial, thus the arithmetic and memory costs are only slightly greater than for basic Lanczos. The main drawback of the Wiedemann approach is that it needs to either store or recompute the sequence $\{A^i b\}_{0\leq i\leq d-1}$.

For both minimal polynomial and system solution, we are developing block versions of the Wiedemann and Lanczos algorithms (see [16]). The choice between the strategies will rely on the same criteria as for the non-blocked versions [7]. We also generalizing the techniques of this section to singular systems.

Preconditioning for the rank and the determinant over finite fields. The computation of the rank and determinant of a black box matrix A reduce to the computation of the minimal polynomial of a matrix A', called a *preconditioning* of A, obtained by composing A with one or more other black boxes called *preconditioners* [17, 5]. A first type of preconditioning involves multiplication of A by Toeplitz, sparse, or butterfly matrices, and typically incurs a cost of $O(n^2 \log n)$ or $O(n^2 \log^2 n)$ operations. For large n, this cost can be prohibitive. We have thus focused on diagonal preconditioners, i.e. *scalings*, which involve only n field elements and incur $O(n)$ cost. They are proven effective for large fields [9, 5] and may be used as heuristics combined with Monte Carlo checking.

`LinBox::rank`: The minimal polynomial $f_v^{A,u}(x)$ of $\{v^{\mathrm{T}} A^i u\}_{i\geq 0}$ always reveals a lower bound for rank A. Whether this bound coincides with the rank depends on the spectral structure of A [17]. In a given application, if that structure is favorable then `LinBox::minpoly` is sufficient (for instance, full rank is certified by checking if $\deg f_v^{A,u}(x)$ is maximal). For large fields the preconditioning $A' = D_1 A^{\mathrm{T}} D_2 A D_1$, with D_1 and D_2 two random diagonal matrices, probabilistically ensures a good structure [9]. For smaller fields, one might use a field extension, but we can also use this preconditioning in the ground field as a heuristic. Moreover, the Lanczos approach allows an easy check. It can produce an orthogonal basis for the range of A', and with $O(n)$ dot products one may orthogonalize a random vector with respect to this basis. The rank is quickly certified by checking that the result is in the null space.

`LinBox::determinant`: Using [5, Theorem 4.2], the determinant is easily obtained from the constant term of the minimal polynomial of the preconditioning $A' = DA$, where D is a random scaling. This has been used for some of the experiments in Sections 4.

Integer computations. Most LinBox algorithms for integer and rational number computations are based on the finite field functionality. Minimal polynomial, determinant and system solution may be computed using Chinese remaindering. We include early termination strategies based on the fact that the solution is known with high probability once the reconstructed values remain stable for a small number of consecutive primes [4, 12]. Integer linear systems, especially sparse ones, may also be solved using the p-adic lifting approach combined with the use of a black box for the inverse matrix modulo p [17, 11]. An efficient Monte Carlo rank determination is based on rank computations modulo random primes (see above). Specialized algorithms for Smith normal form computations (see Section 4 and [8]) and diophantine problems served during the early development of LinBox to validate the design of the library.

4 Computational experiences

What do we know in practice about the prospects for high performance exact linear algebra computations? In this section we cite some examples to illustrate the range of possibilities. The problems described here have been difficult to solve by other means. The alternative to LinBox's exact linear algebra is numeric approximation in some cases and combinatorial methods in others. These alternative approaches suffer fatal numerical instability and/or exponential memory demand, which are avoided here. The first example is not a real application, but rather a posed Challenge problem. Its solution is illustrative of LinBox's capability. A noteworthy fact is that a very long computation succeeded without exhausting memory. Computations with high time to memory ratio are rare in computer algebra.

Trefethen's Hundred Digits. Nick Trefethen has just posed a "Hundred Dollar, Hundred Digit Challenge," [15]. Aimed at numerical analysts, the Challenge consists of 10 problems which have real number solutions. Ten digits of accuracy are asked in the answer to each. All of them are numerically difficult. Problem #7 is the computation of the $(1, 1)$ entry of A^{-1}, where A is the 20000×20000 matrix whose entries are zero everywhere except for the primes $2, 3, 5, 7, \ldots, 224737$ along the main diagonal and the number 1 in all the positions a_{ij} with $|i - j| = 1, 2, 4, 8, \ldots, 16384$. We took this on as a challenge for LinBox to produce the *exact* solution. It is a rational number whose numerator and denominator each has approximately 100,000 digits. To compute this number is well beyond the capability of present day general purpose systems such as Maple and Mathematica on current processors and memories. Indeed, it is beyond the capabilities of any software we know of, save LinBox. We have computed this exact answer using approximately 2 years of CPU time (about 180 CPUs running for 4 days). Trefethen has asked that we not reveal the solution or the details of our method before the deadline of his Challenge, May 20, 2002. However, we intend to include a more complete description of this computation in the final version of this paper. Also, we are experimenting with several approaches to this problem and will expand on their relative performances.

Rank, competition with Gaussian elimination techniques. In table 2 we report some comparisons between Wiedemann's algorithm and elimination with reordering for computing the rank. For the first method, we compute the minimal polynomial of $D_1 A^T D_2 A D_1$, as seen in Subsection Preconditioning of Section 3. In order to show the usefulness of this approach, we compare it to elimination-based methods. As the matrices are sparse, some pre-elimination and on-the-fly reordering may be used to reduce fill-in and augment efficiency.

We have chosen not to focus on this, and the reader may refer to [7] and references therein for a review of some reordering heuristics and their application to the matrices. The timings in column "Gauß" of Table 2 have been obtained using the algorithm of [7, §5.1]. The timings in column "SuperLU" are those of a generic version [`www-sop.inria.fr/galaad/logiciels/synaps`] of the SuperLU numerical code [`www.nersc.gov/~xiaoye/SuperLU`]. For several cases, fill-in causes a failure of elimination. This is due to memory thrashing (MT).

Our experiments show that *as long as enough memory is available*, elimination is very often more efficient when the matrix is already nearly triangular (cyclic8_m11, from Gröbner basis computation), when the matrix has a very small number of elements per row (nick; ch$i$-j, from algebraic topology), or when the matrix is very unbalanced (bibd – Balanced Incomplete Block Design, from combinatorics). Nonetheless, when the mean number of elements per row is significant (greater than 5, say), Wiedemann's algorithm is superior, even on small matrices, and is sometimes the only practical solution.

Table 2: **Rank modulo 65521, Elimination vs. Wiedemann** on an intel PIII, 1GHz, 1Gb. $n \times m$ is the shape, Ω the number of non-zero elements, r the integer rank of the matrix, timings are in seconds.

Matrix	Ω, $n \times m$, r	Gauß	SuperLU	Wiedem.
cyclic8_m11	2462970, 4562x5761, 3903	257.33	448.38	2215.36
bibd_22_8	8953560, 231x319770, 231	100.24	375.23	594.29
n4c6.b12	1721226, 25605x69235, 20165	188.34	1312.27	2158.86
mk13.b5	810810, 135135x270270, 134211	MT		44907.1
ch7-7.b5	211680, 35280x52920, 29448	2179.62	MT	2404.5
ch7-8.b5	846720, 141120x141120, 92959	5375.76	MT	29109.8
ch7-9.b5	2540160, 423360x317520, 227870	MT		210698
ch8-8.b5	3386880, 564480x376320, 279237	MT		363754
TF14	29862, 2644x3160, 2644	50.58	50.34	27.21
TF15	80057, 6334x7742, 6334	734.39	776.68	165.67
TF16	216173, 15437x19321, 15437	18559.40	15625.79	1040.36
TF17	586218, 38132x48630, 38132	MT	MT	7094.97

Smith Form, Simplicial Homology. Integer Smith Normal form, especially as applied to Simplicial Homology computation, was the earliest application of LinBox methods. A GAP share package implements this and the web site http://linalg.org offers both downloads and access to these Smith form computation using a web math server. Combinatorial conjectures have been disproven and others affirmed or inspired by computations using these tools. Matrices with hundreds of thousands of rows and columns and several million nonzero entries have proven tractable for computing Smith forms. See [8] for details, and [3] for some of the homology applications.

Acknowledgements. We are grateful to B. Mourrain and P. Trébuchet for their help in benchmarking their SuperLU code, to A. Lobo for his comments, and to W. Eberly for his many algorithmic improvements published elsewhere.

References

[1] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, editors. *Templates for the solution of Algebraic Eigenvalue Problems: A Practical Guide.* SIAM, Philadelphia, PA, 2000.

[2] R. Barrett, M. Berry, T.F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition.* SIAM, 1994.

[3] Anders Björner and Volkmar Welker. Complexes of directed graphs. *SIAM Journal on Discrete Mathematics*, 12(4):413–424, November 1999.

[4] H. Brönnimann, I.Z. Emiris, V.Y. Pan, and S. Pion. Sign determination in residue number systems. *Theoret. Comput. Sci.*, 210(1):173–197, 1999. Special Issue on Real Numbers and Computers.

[5] L. Chen, W. Eberly, E. Kaltofen, B.D. Saunders, W.J. Turner, and G. Villard. Efficient matrix preconditioners for black box linear algebra. *Linear Algebra and its Applications*, 343-344:119–146, 2002.

[6] A. Díaz and E. Kaltofen. FoxBox a system for manipulating symbolic objects in black box representation. In O. Gloor, editor, *Proc. ISSAC'98*, pages 30–37, New York, N. Y., 1998. ACM Press.

[7] J.-G. Dumas. Algorithmes parallèles efficaces pour le calcul formel: algèbre linéaire creuse et extensions algébriques. Thèse de Doctorat, Institut National Polytechnique de Grenoble, France, décembre 2000.

[8] J-G. Dumas, B. D. Saunders, and G. Villard. On efficient sparse integer matrix Smith normal form computations. *Journal of Symbolic Computations*, 32(1/2):71–99, July–August 2001.

[9] W. Eberly and E. Kaltofen. On randomized Lanczos algorithms. In *International Symposium on Symbolic and Algebraic Computation, Maui, Hawaii, USA*, pages 176–183. ACM Press, July 1997.

[10] E. Kaltofen and A. Lobo. Distributed matrix-free solution of large sparse linear systems over finite fields. *Algorithmica*, 24(3–4):331–348, July–Aug. 1999. Special Issue on "Coarse Grained Parallel Algorithms".

[11] E. Kaltofen and B.D. Saunders. On Wiedemann's method of solving sparse linear systems. In *Proc. AAECC-9*, LNCS 539, Springer Verlag, pages 29–38, 1991.

[12] E. Kaltofen and B. Trager. Computing with polynomials given by black boxes for their evaluations: Greatest common divisors, factorization, separation of numerators and denominators. *J. Symb. Comp.*, 9(3):301–320, 1990.

[13] R. Lambert. *Computational aspects of discrete logarithms.* PhD thesis, University of Waterloo, Ontario, Canada, 1996.

[14] J. R. Munkres. *Elements of algebraic topology*, chapter The computability of homology groups, pages 53–61. Benjamin/Cummings, Inc., 1994.

[15] N. Trefethen. A Hundred-dollar, Hundred-digit Challenge. *SIAM News*, 35(1), 2002.

[16] G. Villard. A study of Coppersmith's block Wiedemann algorithm using matrix polynomials. Rapport de Recherche 975 IM, `www.imag.fr` (Grenoble), April 1997. Extended abstract in Proc. ISSAC'97.

[17] D. Wiedemann. Solving sparse linear equations over finite fields. *IEEE Transf. Inform. Theory*, IT-32:54–62, 1986.