HAL
open science

LinBox: A Generic Library for Exact Linear Algebra

Jean-Guillaume Dumas, T. Gautier, M. Giesbrecht, Pascal Giorgi, B. Hovinen, Erich Kaltofen, B.D. Saunders, W.J. Turner, Gilles Villard

To cite this version:

Jean-Guillaume Dumas, T. Gautier, M. Giesbrecht, Pascal Giorgi, B. Hovinen, et al.. LinBox: A Generic Library for Exact Linear Algebra. [Research Report] LIp RR-2002-15, Laboratoire de l'informatique du parallélisme. 2002, 2+12p. hal-02102080

HAL Id: hal-02102080

https://hal-lara.archives-ouvertes.fr/hal-02102080

Submitted on 17 Apr 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON
n° 5668

LinBox: A Generic Library for Exact Linear Algebra

J.G.Dumas, T.Gautier, M.Giesbrecht, P.Giorgi, B.Hovinen, E.Kaltofen, B.D.Saunders, W.J.Turner and G.Villard

Mars 2002

Research Report N° 2002-15

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : lip@ens-lyon.fr

LinBox: A Generic Library for Exact Linear Algebra

J.G.Dumas, T.Gautier, M.Giesbrecht, P.Giorgi, B.Hovinen,
E.Kaltofen, B.D.Saunders, W.J.Turner and G.Villard

Mars 2002

Abstract

LinBox is a high-performance generic software library for black box linear algebra over symbolic (exact) entry domains. The generic software methodology enables the user to instantiate the procedures in the library with a multitude of coefficient domains and black box matrices without sacrificing performance. At the top level, LinBox provides algorithms for many standard problems in linear algebra, such as equation solving and matrix normal forms, and includes toolboxes for Lanczos-Krylov approaches and for algebraic preconditioning.

Keywords: Linear algebra, randomized algorithms, black box matrix, matrix rank, seminumeric computation.

Résumé

LinBox est une bibliothèque logicielle générique efficace pour l'algèbre linéaire, à travers des matrices boîte noire définies sur des domaines d'entrées symboliques (exactes). Le principe de la généricité logicielle permet d'instancier les procédures sur une multitude de domaines de coefficients et de matrices boîte noire, sans sacrifier les performances. Au plus haut niveau, LinBox fournit des algorithmes pour plusieurs problèmes standards de l'algèbre linéaire, tels que la résolution d'équation et les formes normales de matrices. Linbox inclut aussi des boîtes à outils pour une approche Lanczos-Krylov et pour du préconditionnement algébrique.

Mots-clés: Algèbre linéaire, algorithmes probabilistes, matrice boîte noire, matrice creuse, rang.

LinBox : A Generic Library for Exact Linear Algebra*

J.-G. Dumas[1], T. Gautier[2], M. Giesbrecht[3], P. Giorgi[7], B. Hovinen[4], E. Kaltofen[5], B.D. Saunders[4], W.J. Turner[5] and G. Villard[6,7]

[1]Lab. Lmc Imag, B.P. 53 F38041 Grenoble Cedex 9; [2]Inria, Lab. Id Imag, Ensimag, Zirst 51, av. J. Kuntzmann, F38330 Montbonnot Saint Martin; [3]Dept. Computer Sci., Univ. Waterloo, Waterloo, N2L3G1, Canada; [4]Dept. Computer & Inf. Sci., Univ. Delaware, Newark, DE 19716; [5]Dept. Math., North Carolina State Univ., Raleigh, NC 27695-8205; [6]Cnrs, [7]Lab. Lip, École Normale Supérieure de Lyon, 46, Allée d'Italie, F69364 Lyon Cedex 07

Abstract

LinBox is a high-performance generic software library for black box linear algebra over symbolic (exact) entry domains. The generic software methodology enables the user to instantiate the procedures in the library with a multitude of coefficient domains and black box matrices without sacrificing performance. At the top level, LinBox provides algorithms for many standard problems in linear algebra, such as equation solving and matrix normal forms, and includes toolboxes for Lanczos-Krylov approaches and for algebraic preconditioning.

1 Introduction

Black box techniques [14] 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 [17] and the computation of all the homology groups of simplicial complexes using the Smith normal form [16].

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

*`http://www.linalg.org`. E-mail addresses: `Jean-Guillaume.Dumas@imag.fr`, `Thierry.Gautier@imag.fr`, `mwg@uwaterloo.ca` (Giesbrecht), `Pascal.Giorgi@ens-lyon.fr`, `hovinen@udel.edu`, `kaltofen@math.ncsu.edu`, `saunders@cis.udel.edu`, `wjturner@math.ncsu.edu`, `Gilles.Villard@ens-lyon.fr`. This material is based on work supported in part by the National Science Foundation under grants Nrs. DMS-9977392, CCR-9988177, and ITR/ASC-0113121 (Kaltofen) Nr. CCR-0098284 and ITR/ASC-0112807 (Saunders) and by the Centre National de la Recherche Scientifique, Actions Incitatives No 5929 et Stic LinBox 2001 (Villard) and by the Natural Sciences and Engineering Research Council of Canada (Giesbrecht).

problems [3, 2]. This has also led the computer algebra community to a considerable interest in black box methods. Since Wiedemann's seminal paper [19], 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 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[1] in numerical linear algebra and SYNAPS[2] in symbolic computation. A forerunner of LinBox is described in [12].

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. In the conclusion we summarize the current status and further development plans.

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 which 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. 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

[1] `http://www.osl.iu.edu/research/mtl`
[2] `http://www-sop.inria.fr/galaad/logiciels/synaps`

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.

2.1 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 in some fashion 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 – this is the approach taken 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 also support assignment and equality checking of whole fields. These methods are required by the field interface, but specific implementations may contain more methods.

For each field type, there exists a class that uniformly generates random elements of that field or an unknown subset thereof of 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 the same interface can be hooked into a generic algorithm. There also exists an abstract class that has virtual methods with the same signatures as those in the archetype. The archetype itself contains a pointer to an instance of this abstract base class. Thus, 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.

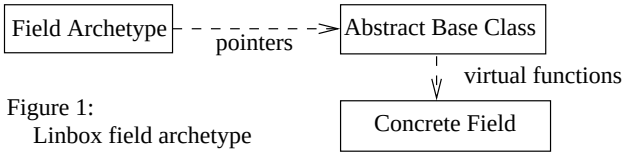
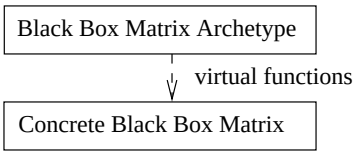


Figure 1: Linbox field archetype

2.2 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. Beyond that, 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 actual black box matrices can be derived directly from it (see Figure 2. We note that the overhead involved with this inheritance mechanism is negligible in comparison with the execution time of the methods. For our field element types, this was not the case.

Figure 2: LinBox black box matrix archetype



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. We saw no necessity for the latter. The field of entries is bound directly to the black box matrix class and is available to be an argument to our black box algorithms, which may perform additional coefficient field operations. Black box matrix classes can have the field type as a template parameter, but such genericity is not required. 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.

2.3 Input iterators

A major idea of the STL design is the separation of containers and their iterators. The container types provide various ways to store data, while the iterator types provide different strategies to access the data. In LinBox, we have extended this notion in order to obtain more genericity at the level of algorithm inputs. We distinguish between the way an input is computed and the way it is accessed. Indeed, algorithms are independent of the way their inputs are produced, even if for the sake of simplicity or of considerations about speed or memory management, both are intimately tied. We propose the use of *input iterators* as input data. These may themselves be an interface to a primary container-like data type (e.g., black boxes) and involve some computations on this data. This indirection allows the algorithm to be oblivious, for instance, to some pre-computations, to memory management or to the order of evaluation, and more generally to serialization or parallelization issues. An example is Wiedemann's algorithm for the computation of the minimal polynomial of a matrix (see Section 3.2). 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 [12]). We have implemented these solvers with the STL iterator interface and show here an example with asynchronous calls:

```
BlackBox_Container::const_iterator&
                  BlackBox_Container::operator++()
                  { _c._launch(); return *this; }
const BlackBox_Container::value_type&
                  BlackBox_Container::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 interface. The sequence itself may be computed by an input iterator with any of the previously mentioned methods. Thus the outer algorithm is generic with respect to the choice of those methods. We now have the advantage to easily switch

among parallel, pipelined, or other strategies with the purpose of very effective adaptation to available resources. For example, one can avoid redundant computation by making a first pass on-the-fly; a verification run then can reuse the same algorithm but with the sequence already known. (Cf. *streams* in the Scheme language [1].)

3 Black box algorithms

As seen in the introduction, Wiedemann's paper [19] has led to many developments in the application of Krylov or Lanczos methods to fast exact problem solving. In this section we present the main directions followed by LinBox regarding these methods in the first version of the library. The reader may refer to [5] for a detailed presentation of theoretical aspects. Linear algebra done over finite fields is the core of the library. Computations over the integers or rationals build upon this core.

3.1 Randomized algorithms, heuristics and checking

In black box linear algebra very often the fastest algorithms use random bits. Our library includes *Monte Carlo*, *Las Vegas*, and deterministic algorithms. Monte Carlo algorithms produce correct results with controllably high probability, but the results are not certified. With Las Vegas algorithms, the answer is always correct and, with high probability, is produced quickly. Properties of these algorithms are theoretically proven correct under conditions on the source of random bits. For example, random elements must be chosen uniformly from a sufficiently large subset of the field. This condition may be prohibitively costly in practical situations. For example, if the field is small, use of an algebraic extension may be required, greatly slowing the computation. This has led us to a new implementation strategy. We relax conditions on our algorithms but introduce subsequent checks for correctness. We exploit randomized algorithms as heuristics when the conditions for provably high probability of success are not satisfied. To complement this, specific checks are developed to certify the outputs. These checks may themselves be randomized (see Section 3.3), in which case we certify only the probability of success. This strategy has been powerful in obtaining solutions for huge problems in Sections 4.1 and 4.3. Following [11] we also use cross checks between algorithms. For instance when finite field computations are used within Chinese remaindering for integer computations, additional information at the integer level may give further checking of the finite field level outputs.

3.2 Minimal polynomial and nonsingular 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 the matrix A only via matrix-vector products $v = A \cdot u$ and $w = A^{\mathrm{T}} \cdot u$. Therefore the library need only supply a black box for A. The black box may efficiently 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 $v^{\mathrm{T}}u, v^{\mathrm{T}}Au, v^{\mathrm{T}}A^2u, \ldots$ for a random vector v. We let d be the degree

of the minimum linear generating polynomial $f_v^{A,u}(x) = x^d - f_{d-1}x^{d-1} - \ldots - f_1 x - f_0$ of the latter sequence, meaning that $v^{\mathrm{T}} A^{d+i} u = f_{d-1}(v^{\mathrm{T}} A^{d+i-1} u) + \ldots + f_{i+1}(v^{\mathrm{T}} A u) + f_i(v^{\mathrm{T}} u), i \geq 0$. Then $f_v^{A,u}(x)$ always divides $f^{A,u}$ and, furthermore, with high probability, $f_v^{A,u}(x) = f^{A,u}$ [19]. Berlekamp-Massey will compute $f_v^{A,u}(x)$ after processing the first $2d$ elements of the sequence. The generation of the sequence is distinguished from the computation of the linear recurrence. In the Lanczos approach, these tasks are intermixed in a single iteration. As the vectors in the Krylov subspace are generated and orthogonalized, the coefficients of the recurrence are computed on the fly as dot products. For a unifying study of both approaches over finite fields we refer to [15].

`LinBox::minpoly`: With high probability, the minimal polynomial $f^{A,u}$ of a random vector u is the minimum polynomial f^A of A [19, 13]. The basic implementation computes the minimal polynomial of A using Wiedemann's algorithm and two random vectors u and v to generate the sequence $\{v^{\mathrm{T}} A^i u\}_{0 \leq i \leq 2n-1}$. The 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. The argument is heuristic in general but probabilistic when applied over large fields to preconditioned matrices with Lanczos' algorithm [10, 9]. A Monte Carlo check of the early termination is implemented by incorporating the application of the computed polynomial to a random vector. From [15] and [8, Chap. 6], we give the dominant terms of the arithmetic costs in Table 1. Terms between brackets give the number of memory locations required for 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	System sol. $Az = b$
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`: If the matrix A is nonsingular, a linear system $Az = b$ is solved by computing the minimal polynomial $f^{A,b}(x) = x^d + f_{d-1}x^{d-1} + \ldots + f_1 x + f_0$ of b, since its coefficients directly give $z = -(1/f_0)(A^{d-1}b + f_{d-1}A^{d-2}b + \ldots + f_1 b)$. 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 the minimal polynomial and system solution, we are developing block versions of the Wiedemann and Lanczos algorithms [18, and the references

therein] The choice between the two strategies will rely on the same criteria as for the non-blocked versions [8].

3.3 Preconditioning for the rank and the determinant over finite fields

The computation of the rank and of the 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* [19, 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 very large n, this overhead can be prohibitively costly. 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 [10, 5] and, as we will see, may be used as heuristics combined with Monte Carlo checking.

`LinBox::rank`: The minimal polynomial of $\{v^{\mathrm{T}} A^i u\}_{i \geq 0}$ always reveals a lower bound for the rank of A. Whether this bound coincides with the rank depends on the spectral structure of A [19]. In a given application, if that structure is favorable then `LinBox::minpoly` is sufficient, for instance, to certify full rank, as this bound reaches the maximum possible value. In the general case 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 [10]. For smaller fields, a first alternative is to use this preconditioning together with field extensions. Nonetheless, we may use this preconditioning without the field extension as a heuristic, even if theoretical conditions are not satisfied. It happens that the Lanczos approach allows an easy check. Indeed, the Lanczos algorithm may be used to produce an orthogonal basis for the range of A'. With $O(n)$ dot products one may orthogonalize a random vector with respect to this basis. The rank is certified by checking that the result is in the null space. This induces an overhead of only $4nr$ operations, where n is the matrix dimension and r is the rank.

`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 intensively used for the experiments in Sections 4.1 and 4.3.

The techniques of this section will be generalized for singular system solution and consistency certification.

3.4 Integer computations

Most of 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 relatively small number of consecutive primes [4]. This idea has been initially proposed for Monte Carlo interpolation algorithms [14, 11]. 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 [19, 13]. A very efficient Monte Carlo rank determination is based on rank computations modulo random primes (see Section 4.2). The specialized algorithms for Smith normal

form computations (see Section 4.3 and [7]) 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.

4.1 Trefethen's Hundred Digits

Nick Trefethen has just posed a "Hundred Dollar, Hundred Digit Challenge," [17]. 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 hard and he suggests, "If anyone gets 50 digits in total, I will be impressed." One of the problems is a linear algebra problem having an exact rational solution in principle. We took it on as a challenge for LinBox to produce this exact solution in practice. Specifically, 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 $|i-j| = 1, 2, 4, 8, \ldots, 16384$. The exact answer is a rational number, the quotient of two integers of approximately 100,000 digits, as is suggested by Hadamard's bound, which turns out to be sharp in this case. To compute this rational 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.

4.2 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 Section 3.3. 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 [8] 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 [8, §5.1]. The timings in column "SuperLU" are those of a generic version[3] of the SuperLU numerical code[4]. Our experiments were performed on an Intel Pentium III, 1 GHz and 1024 Mb of memory. We computed ranks of matrices over finite fields GF(q) where q has half word size. The chosen arithmetic used discrete logarithms with precomputed tables. Let Ω denote the number of nonzero elements, $N \times n$ the shape, and r the integer rank of the matrix under consideration. For several cases, fill-in causes a failure of elimination. This is due to memory thrashing (MT). All the timings presented are in seconds, except as otherwise specified, and reflect the CPU time for sequential computation and the real time for parallel computation.

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 (n*ick*; 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.**

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	938.81	594.29
n4c6.b12	1721226, 25605x69235, 20165	188.34	1312.27	2158.86
mk13.b5	810810, 135135x270270, 134211	MT		44907.10
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		34h×4[a]
ch8-8.b5	3386880, 564480x376320, 279237	MT		55h×4[a]
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

[a] In hours on 4 processors

4.3 Smith Form, Simplicial Homology

As a way to recognize patterns, algebraic topologists use the intrinsic structure of associated simplical complexes. This structure is the sequence of homology groups of the complex and can be computed by integer diagonalizations, called the Smith normal forms, of the boundary matrices of the complex [16]. These matrices can be very large, with several hundreds of thousands of rows and columns and having several millions of nonzero entries. Therefore, despite their sparsity, the diagonalizations were often infeasible by elimination because of significant fill-in and coefficient growth.

For the complexes we have studied, the minimal polynomials of the symmetrizations ($A^T A$) of the boundary matrices A, usually have a very low degree, on the order of 25 for even the largest cases. In such cases Wiedemann's

[3] `http://www-sop.inria.fr/galaad/logiciels/synaps`
[4] `http://www.nersc.gov/~xiaoye/SuperLU`

algorithm with early termination is extremely fast, and the integer coefficients of the minimal polynomial are very small. We are able to compute these coefficients and directly determine the prime numbers involved in the Smith form. Then rank computations, done either via elimination or using the rank heuristic discussed in Sections 3.3 and 4.2 over the associated prime fields, complete the algorithm (see [7]). We have computed Smith forms that were not previously known. In particular, we have disproved a combinatorial conjecture that a certain simplical complex has only 3-torsion. We proved the existence of 2-torsion in the 7×8 chess board complex using about a month of CPU time. This was done with LinBox code incorporated into a share package for the computer algebra software GAP. We also provide free web-based access to these capabilities via a math server (http://linalg.org). For instance, some users have succeeded in completing more than 36 hour homology computations using the server.

5 Conclusion

LinBox is a generic C++ source code library for exact and semi-numerical linear algebra computation. It is designed for genericity and for use as a component. The domain of computation may be any field implementation, and for some algorithms, any commutative ring. A plug-in of part of LinBox to the GAP package exists and component use as a plug-in from such systems as Maple and Mathematica is envisioned for the future.

The black box matrix representation is central to LinBox. Any idea whatsoever of how to implement the application of a given linear operator (the matrix-vector product) may be easily set up as a black box object and used with the LinBox algorithms.

Algorithm implementations exist for minimal polynomial, rank, system solution, and Smith normal form. Currently the library is undergoing polishing for release of a first public version. Web access to some of the functionality already exists at linbox.org. Future work will extend the capabilities, for instance for characteristic polynomial and Frobenius form computation. Performance improvements are also in the works, especially the use of block-Wiedemann and block-Lanczos algorithms and the use of parallelism.

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

References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Program.* MIT Press, Cambridge, MA, 1985.

[2] 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.

[3] 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, Philadelphia, PA, 1994.

[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. 1998 Internat. Symp. Symbolic Algebraic Comput. (ISSAC'98)*, pages 30–37, New York, N. Y., 1998. ACM Press.

[7] 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.

[8] 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.

[9] W. Eberly. Avoidance of look-ahead in Lanczos by random projections. Manuscript in preparation.

[10] 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.

[11] E. Kaltofen, W.-s. Lee, and A.A. Lobo. Early termination in Ben-Or/Tiwari sparse interpolation and a hybrid of Zippel's algorithm. In C. Traverso, editor, *Proc. 2000 Internat. Symp. Symbolic Algebraic Comput. (ISSAC'00)*, pages 192–201, New York, N. Y., 2000. ACM Press.

[12] 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".

[13] 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.

[14] 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.

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

[16] J. R. Munkres. *Elements of algebraic topology*, chapter The computability of homology groups, pages 53–61. Advanced Book Program. The Benjamin/Cummings Publishing Company, Inc., 1994.

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

[18] G. Villard. A study of Coppersmith's block Wiedemann algorithm using matrix polynomials. Rapport de Recherche 975 IM, Institut d'Informatique et de Mathématiques Appliquées de Grenoble, `www.imag.fr`, April 1997. Extended abstract in Proc. ISSAC'97.

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