Elements of Design for Containers and Solutions in the LinBox library

Extended abstract

Brice Boyer[1], Jean-Guillaume Dumas[2], Pascal Giorgi[3], Clément Pernet[4], and B. David Saunders[5]

[1] Department of Mathematics, North Carolina State University, USA[†]
bbboyer@ncsu.edu.
[2] Laboratoire J. Kuntzmann, Université de Grenoble. France[‡]
Jean-Guillaume.Dumas@imag.fr.
[3] LIRMM, CNRS, Université Montpellier 2, France[‡]
pascal.giorgi@lirmm.fr.
[4] Laboratoire LIG, Université de Grenoble et INRIA, France[‡]
clement.pernet@imag.fr.
[5] University of Delaware, Computer and Information Science Department, USA
saunders@udel.edu.

Abstract. We develop in this paper design techniques used in the C++ exact linear algebra library LinBox, intended to make the library safer and easier to use, while keeping it generic and efficient. First, we review the new simplified structure for containers, based on our *founding scope allocation* model. We explain design choices and their impact on coding: unification of our matrix classes, clearer model for matrices and submatrices, *etc.* Then we present a variation of the *strategy* design pattern that is comprised of a controller–plugin system: the controller (solution) chooses among plug-ins (algorithms) that always call back the controller. We give examples using the solution `mul`. Finally we present a benchmark architecture that serves two purposes: Providing the user with easier ways to produce graphs using C++; Creating a framework for automatically tuning the library and provide regression testing schemes.

Keywords: LinBox; design pattern; algorithms and containers; benchmarking; matrix multiplication algorithms; exact linear algebra.

1 Introduction

This article follows several papers and memoirs on the LinBox[6] (*cf.* [2,7,8,13,19]) and builds upon them. LinBox is a C++ template library for fast and exact linear

[†] This material is based on work supported in part by the National Science Foundation under Grant CCF-1115772 (Kaltofen)
[‡] This material is based on work supported in part by the Agence Nationale pour la Recherche under Grant ANR-11-BS02-013 HPAC (Dumas, Giorgi, Pernet).
[6] See http://www.linalg.org.

algebra, designed with genericity and efficiency in mind. The LinBox library is under constant evolution, driven by new problems and algorithms, by new computing paradigms, new compilers and architectures. This poses many challenges: we are incrementally updating the *design* of the library towards a 2.0 release.

Let us start from a basic consideration: we show in the Table 1 the increase in the "lines of code" size [7] of LinBox and its co-evolved dependencies Givaro and Fflas–Ffpack. This increase affects the library in several ways. First, it demands

LinBox	1.0.0†‡	1.1.0†‡	1.1.6‡	1.1.7‡	1.2.0	1.2.2	1.3.0	1.4.0
loc (×1 000)	77.3	85.8	93.5	103	108	109	112	135
Fflas–Ffpack	n/a	n/a	n/a	1.3.3	1.4.0	1.4.3	1.5.0	1.8.0
loc	—	—	—	11.6	23.9	25.2	25.5	32.1
Givaro	n/a	n/a	3.2.16	3.3.3	3.4.3	3.5.0	3.6.0	3.8.0
loc	—	—	30.8	33.6	39.4	41.1	41.4	42.8
total	77.3	85.8	124	137	171	175	179	210

Table 1: Evolution of the number of lines of code (loc, in thousands) in LinBox, Fflas–Ffpack and Givaro (†contains Givaro, ‡contains Fflas–Ffpack).

a stricter development model, and we are going to list some techniques we used. For instance, we have transformed Fflas–Ffpack[8] (*cf.* [10]) into a new standalone header library, resulting in more visibility for the Fflas–Ffpack project and also in better structure and maintainability of the library. A larger template library is harder to manage. There is more difficulty to trace, debug and write new code. Techniques employed for easier development include reducing compile times, enforcing stricter warnings and checks, supporting more compilers and architectures, simplifying and automating version number changes, automating memory leak checks, setting up build-bots to check the code frequently,...

This size increase also requires more efforts to make the library user friendly. For instance, we have: Developed an `auto-install.sh` script that installs automatically the latest stable or development versions of the trio, resolving the version dependencies; Facilitated the discovery of the Blas/Lapack libraries; Simplified and sped up the checking process while covering more of the library; Updated the documentation and distinguished user and developer oriented docs; Added comprehensive benchmarking tools,...

Developing generic high-performance libraries is difficult. We can find a large literature on coding standards and software design references in (*cf.* [1,11,15,17,18]), and many internet sources and experience acquired by/from free software projects. Another motivation for developing a high-performance mathematical library is to make it available and easy for researchers and engineers that will use it for producing quality reliable results and quality research papers.

We describe advances in the design of LinBox in the next three sections. We will first describe the new *container* framework in Section 2, then the im-

[7] Using sloccount, available at http://sourceforge.net/projects/sloccount/.
[8] See http://www.linalg.org/projects/fflas-ffpack/.

proved *matrix multiplication* algorithms in Section 3 made by contributing special purpose matrix multiplication plug-ins, and, finally, we present the new *benchmark/optimisation* architecture (Section 4).

2 Containers architecture

LinBox is mainly conceived around the RAII (Resource Acquisition Is Initialisation, see [17]) concept with re-entrant function. We also follow the founding scope allocation model (or *mother model*) from [8] which ensures that the memory used by objects is allocated in the constructor and freed only at its destruction. The gestion of the memory allocated by an object is then exlusively reserved to it.

LinBox essentially uses matrix and vectors over fields and rings as data objects (containers). The fragmentation of the containers into various matrix and blackbox types needed to be addressed and simplified. The many different matrix and vector types with different interfaces needed to be reduced into only two (possibly essentially one in the future) containers: `Matrix` and `Vector`.

2.1 General Interface for Matrices

First, in order to allow operations on its elements, a container is parametrized by a field object (Listing 1.1), not the field's element type. This is simpler and more general. Indeed, the field element type can be infered from a `value_type` type definition within the field type. Then, the storage type is given by a second template parameter that can use defaults, *e.g.* dense BLAS matrices (stride and leading dimension or increment), or some sparse format.

```
template< class _Field, class _Storage = denseDefault >
class Vector ;
```

Listing 1.1: Matrix or Vector classes in LinBox.

In the mother model, we must distinguish containers that own (are responsible for allocated memory) and containers that share memory. The `SubMatrix` and `SubVector` types share the memory while `Matrix` and `Vector` own it. The common interface shared by all container matrices is the BlackBox interface described in the following paragraphs, it accomodates both owner and sharer container types, and defines the minimal methods required to be a template BlackBox type:

Input/Output. Our matrix all read and write from MatrixMarket format[9]) which is well established in the numerical linear algebra community. This allows for easy exchange and interoperability, but LinBox containers can also add some comments to the file, for instance the field characteristics (modulo, irreducible polynomial, `init` function, rank,...) We can adapt the header to suit our needs, and even create new formats that save space (CSR fashion saving roughly a third space over COO), or dedicated to structured matrices.

Accessing Elements. The function `setEntry` can be used to populate the matrix (from some `init()` until a `finish()` is emitted). The function `setEntry`

[9] See http://math.nist.gov/MatrixMarket/.

can be (very) costly (for some sparse formats for instance) Other functions such as `refEntry` (that retrieves a writable reference to an entry) may be difficult to implement or inefficient (compressed fields, sparse matrices); `getEntry` may be specialized, in all cases, there is a solution for this operation falling back to the `apply` method. Accessing (read or write) elements, rows or columns through iterators, although handy, is not required because it can be tricky to implement (compressed fields, hybrid or recursive formats). Traits on matrices where efficient iterators are available allow for their selections in some algorithms.

Apply method. This is essential in the BlackBox interface (Sections 2.2 and 3).

Rebind/Conversions. Rebind from one field to the other (if possible, using some default homeomorphism). Conversion mechanisms between formats are added to the interface when convenient, for instance all sparse matrix formats can convert to/from CSR format. This 'star' mechanism can simplify the code (to the expense of memory usage) and may speed it up when the mother format (CSR for instance) is well tuned for some task.

This is a common minimal interface to all our matrices that can be used by all of our algorithms. Additional elements to a container can be added, and flagged with a trait. This model also ensures basic "linear application" use of a matrix (blackbox) which is the least one can request from a matrix. For instance, this interface is shared by: Our *dense* containers (BLAS type or vector of rows); Our *permutations* containers (compressed LAPACK or cycle representation); Our *sparse* formats (based on common formats such as COO, CSR,... or based on STL containers such as `map`, `deque`, `pair`,...); Our *structured* containers (Diagonal, Hankel, Butterfly,...); Our *compound* containers (Sum, Compose, Submatrix,...).

2.2 The `apply` method

The `apply` method (left or right) is arguably the most important feature in the matrix interface and the LinBox library. It performs what a linear application is defined for: apply to a vector (and by extension a block of vectors, *i.e.* a matrix).

We propose the new interface (Listing 1.2), where `_In` and `_Out` are vector or matrices, and `Side` is either `Tag::Right` or `Tag::Left`, whether the operation $y \leftarrow A^\top x$ or $y \leftarrow Ax$ is performed. We also generalize to the operation $y \leftarrow \alpha Ax + \beta y$ as it is often handy and efficient to allow for accumulation.

```
// y = A.x
template< class _In, class _Out  >
_Out& apply(_Out &y, const _In& x, enum Side) ;
```

Listing 1.2: Apply methods.

This method is important for two reasons: first it is the building block of the BlackBox algorithms (for instance Wiedemann and block-Wiedemann); second the matrix multiplication is a basic operation in linear algebra that needs to be extremely efficient (this is the matter of Section 3). The implementation of the apply method can be left to the `mul` solution (see Section 3), with possibly a helper/method argument if the apply parameters are specialized enough.

3 Improving LinBox matrix multiplication

3.1 Plug-in structure

We propose the following *design pattern* (the closest design pattern to our knowledge is the *strategy* one, see also [6, Fig 2.]). The main advantage of this design pattern is that the modules always call back the controller so that the best choice is always chosen. Besides modules can be easily added as *plug-ins*. An analogy can be drawn with dynamic systems—once the controller sends a correction to the system, it receives back a new measure that allows for a new correction.

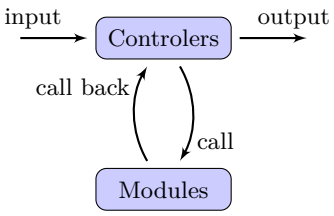

Fig. 1: Controller/Module design pattern

For instance, we can write (Figure 2) the standard cascade algorithms (see [10]) in that model. Cascade algorithms are used to combine several algorithms that are switched using thresholds, ensuring better efficiency than any of the algorithms within used separately. This method allows for the reuse of modules and ensures efficiency. It is then possible to adapt to the architecture, the available modules, the resources. The only limitation is that the choice of the module should be done fast. On

Algorithm 1: `Algo`: controler

Input: A and B, denses, with resp. dimensions $n \times k$ and $k \times n$.
Input: H Helper
Output: $C = A \times B$
if $\min(m, k, n) < H.$`threshold`() **then**
  | `Algo`(C,A,B,`BaseCase`()) ;
else
  | `Algo`(C,A,B,`RecursiveCase`())
end

Algorithm 2: `Algo`: recursive module

Input: A, B, C as in controller.
Input: H, `RecursiveCase` Helper
Output: $C = A \times B$
Cut A,B,C in S_i, T_i
...
$P_i =$ `Algo`(S_i, T_i, H)
...

Fig. 2: Conception of a recursive controlled algorithm

top of this design, we have Methods/Helpers that allow (preferred) selection of algorithms and cut short in the strategy selection.

This infrastructure also invites to modularise the code. For instance, FFLAS–FFPACK has seen major modularization (addition, scaling, reduction,...) Not only it allows to write code with hardly more line than it takes for pseudo-code listings in [5] (compared to $\approx 2.5\times$ on some routines before) but also it automatically brings performance, because we can the separately improve on these modules and can switch easier between them. Also, this reduces the lines of code, hence the probability for bugs, eases their tracing/tracking, and allows for more unit tests. Modularising the code comes at almost no cost because we may add $O(1)$

operations that: Do not cost much compared to $O(n^2)$ or more complexity of the modules; Allow early decisions and terminations (*e.g.* by testing against 0, ± 1, checking leading dimensions or increments); Allow better code (AVX, SSE, copy–cache friendly operation–copy back, representation switching,...)

3.2 New algorithms/infrastructure

We show several new algorithms that improve on matrix multiplication in various ways: reducing memory consumption, introducing new efficient algorithms, using graphics capabilities, generalizing the BLAS to integer routines.

Reduced memory. The routine `fgemm` in FFLAS uses by default the classic schedules for the multiplication and the product with accumulation (*cf.* [5]), but we also implement the low memory routines therein. The new algorithms are competitive and can reach sizes that were limiting. One difficulty consists in using the memory space contained in a sub-matrix of the original matrix, that one cannot free or re-allocate.

Using Bini's approximate formula. In [3], we use Bini's approximate matrix multiplication formula to derive a new algorithms that is more efficient that the Strassen–Winograd implementation in `fgemm` by $\approx 5 - 10\%$ on sizes 1 500–3 000. This is a cascade of Bini's algorithm and Strassen–Winograd algorithm and/or the naïve algorithm (using BLAS). The idea is to analyse precisely the error term in the approximate formula and make it vanish.

Integer BLAS. In order to provide fast matrix multiplication with multiprecision integers, we rely on multi-modular approach through the chinese remainder theorem. Our approach is to reduce as much as possible to `fgemm`. Despite, the existence of fast multimodular reduction (resp. reconstruction) algorithm [12], the naive quadratic approach can be reduced to `fgemm` which makes it more efficient into practice. Note that providing optimized fast multimodular reduction remains challenging. This code is directly integrated into FFLAS.

Polynomial Matrix Multiplication over small prime fields. The situation is similar to integer matrices since one can use evaluation/interpolation techniques through DFT transforms. However, the optimized Fast Fourier Transform of [16] makes fast evaluation (resp. interpolation) competitive into practice. We thus rely on this scheme together with `fgemm` for pointwise matrix multiplications. One can find some benchmark of our code in [14].

Sparse Matrix–Vector Multiplication Sparse matrices are usually problematic because the notion of *sparsity* is too general *vs.* the specificity of real world sparse matrices: the algorithms have to adapt to the shape of the sparse matrices. There is a huge literature from numerical linear algebra on SpMV(Sparse Matrix Vector multiplication) and on sparse matrix formats, some of which are becoming standard (COO, CSR, BCSR, SKY,...). In [4] we developed some techniques to improve the SpMVoperation in LinBox. Ideas include the separation of the ± 1 for removing multiplications, splitting in a sum (HYB for hybrid format) of sparse matrix whose formats are independant and using specific routines. For instance, on $\mathbf{Z}/p\mathbf{Z}$ with word size p, one can split the matrix ensuring no reduction is needed in the dot product and call Sparse BLAS (from Intel MKL or

Nvidia cuBLAS for instance) on each matrix. One trade-off is as usual between available memory, time spent on optimizing *vs.* time spent on `apply`, and all the more so because we allow the concurrent storarge of the transpose in an optimised fashion, usually yielding huge speed-ups. This can be decided by *ad hoc.* optimizers.

Parallelisation Work on parallelizations using OpenCL, OpenMP or XKaapi for dense or sparse matrix multiplication include [4, 9, 20].

These state-of-the-art algorithms, often inter-dependant, need to be selected by the `mul` solution, possibly using a Method/Helper parameter. The Controller/-Module model works particularly well here: cascading, switch methods, algorithms selection is made easy. Besides, the choosing of a particular algorithms or base case is never settled in advance: the library has to be tuned wrt. available libraries, local performance of these libraries.

4 Benchmarking for automatized tuning and regression testing

Benchmarking was introduced in LinBox for several reasons. First, It would give the user a user-friendly way for producing quality graph with no necessary knowledge of a graphing library like gnuplot[10] or provide the LinBox website with automatically updated tables and graphs. Second, it would be used for regression testing. Finally, it would be used for selecting default method, threshold. A lot of libraries do some automatic tuning at installation (fftw, ATLAS, NTL,...).

4.1 Performance evaluation and Automatized regression testing

Our plotting mechanism is based on two structures: `PlotStyle` and `PlotData`. The `PlotGraph` structure uses the style and data to manage the output. We allow plotting in standard image formats, html and LaTeXtables, but also in raw csv or xml for file exchange, data comparisons and extrapolation. This mechanism can also automatically update LinBox feature matrix that describes how our solutions behave on the fields we support. (vague)

Saving graphs in raw format can enable automatic regression testing on the build-bots that already checked our code. For some determined matrices (of different shape and size) over several fields, we can accumulate the timings for some of our solutions (`rank`, `det`, `mul`,...) over time. At each new release, when the documentation is updated, we can check any regression on these base cases and automatically update the regression plots.

4.2 Automatized tuning and method selection

Some of the code in LinBox is already automatically tuned (such as thresholds in `fgemm`), but we improve on it. It is well known that CPU throttling while

[10] http://www.gnuplot.info/

building ATLAS causes bad optimisations; the FFLAS–FFPACK library could also suffer from not-to-reliable threshold detection (up to 50% relative difference for some thresholds between runs). Instead of searching for a threshold using fast dichotomous techniques for instance, we propose to interpolate curves and find the intersection. Using least squares fitting, we can even tolerate outliers. This is however more time consuming.

Automatically tuning a library is not only about defining thresholds, it is also about method/algorithm selection. Our strategy is the following: a given algorithm is tuned for each `Helper` (method) it has. Then the solution (that uses these algorithms) is tuned for selecting the best methods. At each stages, defaults are given, but can be overridden by the optimiser. The areas where a method is better are extrapolated from the benchmarking curves.

References

1. A. Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. C++ in-depth series. Addison-Wesley, 2001.
2. B. Boyer. *Multiplication matricielle efficace et conception logicielle pour la bibliothèque de calcul exact* LINBOX. PhD thesis, Université de Grenoble, June 2012.
3. B. Boyer and J.-G. Dumas. Matrix multiplication over word-size prime fields using Bini's approximate formula. Submitted. http://hal.archives-ouvertes.fr/hal-00987812, May 2014.
4. B. Boyer, J.-G. Dumas, and P. Giorgi. Exact sparse matrix-vector multiplication on GPU's and multicore architectures. In *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, PASCO '10, pages 80–88, New York, NY, USA, 2010. ACM.
5. B. Boyer, J.-G. Dumas, C. Pernet, and W. Zhou. Memory efficient scheduling of Strassen-Winograd's matrix multiplication algorithm. In *Proceedings of the 2009 international symposium on Symbolic and algebraic computation*, ISSAC '09, pages 55–62, New York, NY, USA, 2009. ACM.
6. V.-D. Cung, V. Danjean, J.-G. Dumas, T. Gautier, G. Huard, B. Raffin, C. Rapine, J.-L. Roch, and D. Trystram. Adaptive and hybrid algorithms: classification and illustration on triangular system solving. In J.-G. Dumas, editor, *Proceedings of Transgressive Computing 2006, Granada, España*, Apr. 2006.
7. J.-G. Dumas, T. Gautier, M. Giesbrecht, P. Giorgi, B. Hovinen, E. Kaltofen, B. D. Saunders, W. J. Turner, and G. Villard. LINBOX: A generic library for exact linear algebra. In *Proceedings of the 2002 International Congress of Mathematical Software, Beijing, China*. World Scientific Pub, Aug. 2002.
8. J.-G. Dumas, T. Gautier, C. Pernet, and B. D. Saunders. LinBox founding scope allocation, parallel building blocks, and separate compilation. In K. Fukuda, J. Van Der Hoeven, and M. Joswig, editors, *Proceedings of the Third International Congress Conference on Mathematical Software*, volume 6327 of *ICMS'10*, pages 77–83, Berlin, Heidelberg, Sept. 2010. Springer-Verlag.
9. J.-G. Dumas, T. Gautier, C. Pernet, and Z. Sultan. Parallel computation of echelon forms. Technical report, Feb. 2014.
10. J.-G. Dumas, P. Giorgi, and C. Pernet. Dense linear algebra over word-size prime fields: the FFLAS and FFPACK packages. *ACM Trans. Math. Softw.*, 35(3):1–42, 2008.

11. E. Gamma. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.
12. J. v. Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 1999.
13. P. Giorgi. *Arithmétique et algorithmique en algèbre linéaire exacte pour la bibliothèque* LinBox. PhD thesis, École normale supérieure de Lyon, Dec. 2004.
14. P. Giorgi and R. Lebreton. Online order basis and its impact on block Wiedemann algorithm. In *Proceedings of the 2014 international symposium on symbolic and algebraic computation*, ISSAC '14. ACM, 2014. (to appear).
15. D. Gregor, J. Järvi, M. Kulkarni, A. Lumsdaine, D. Musser, and S. Schupp. Generic programming and high-performance libraries. *International Journal of Parallel Programming*, 33:145–164, 2005. 10.1007/s10766-005-3580-8.
16. D. Harvey. Faster arithmetic for number-theoretic transforms. *Journal of Symbolic Compututations.*, 60:113–119, Jan. 2014.
17. B. Stroustrup. *The design and evolution of C++*. Programming languages/C++. Addison-Wesley, 1994.
18. H. Sutter and A. Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, And Best Practices*. The C++ In-Depth Series. Addison-Wesley, 2005.
19. W. J. Turner. *Blackbox linear algebra with the* LinBox *library*. PhD thesis, North Carolina State University, May 2002.
20. M. Wezowicz, B. D. Saunder, and M. Taufer. Dealing with performance/portability and performance/accuracy trade-offs in heterogeneous computing systems: a case study with matrix multiplication modulo primes. volume 8403, pages 08–08–10, 2012.