

Elements of Design for Containers and Solutions in the LinBox library

Brice Boyer¹, Jean-Guillaume Dumas², Pascal Giorgi³, Clément Pernet⁴, and
B. David Saunders⁵

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. They are intended to make the library safer and easier to use, while keeping it generic and efficient.

First, we review the new simplified structure of the containers, based on our *founding scope allocation* model. Namely, vectors and matrix containers are all templated by a field and a storage type. Matrix interfaces all agree with the same minimal blackbox interface. This allows e.g. for a unification of our dense and sparse matrices, as well as a clearer model for matrices and submatrices. We explain the design choices and their impact on coding. We will describe several of the new containers, especially our sparse and dense matrices storages as well as their **apply** (*blackbox*) method and compare to previous implementations.

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) and the plug-ins always call back the solution so a new choice can be made by the controller. We give examples using the solution `mul`, and generalise this design pattern to the library. We also show performance comparisons with former LinBox versions.

Finally we present a benchmark architecture that serves two purposes. The first one consists in providing the user with an easy way to produce graphs using C++. The second goal is to create a framework for automatically tuning the library (determine thresholds, choose algorithms) and provide a regression testing scheme.

Keywords: LinBox, design pattern, solutions and containers, benchmarking

[†] 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).

1 Introduction

This article follows several papers and memoirs on the LinBox⁶ (*cf.* [11,15,2,7,8]) and builds upon them.

LinBox is a C++ template library for fast and exact linear algebra. It is 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 new challenges. To address this changes, we are incrementally updating the *design* of the library towards a 2.0 release.

Let’s start from a basic consideration: we show in the Table 1 the increase in the size⁷ of LinBox and its dependancies in terms of “lines of code”. This

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).

increase affects the library in several ways. First, it demands a stricter developpement model, and we are going to list some techniques we used. For instance, we have transformed FFLAS–FFPACK⁸ (*cf.* [9]) into a new stand-alone header library, resulting in more visibility for the FFLAS–FFPACK project (Singular ?) but also in a better structuration and maintainability of the library, focusing the developpement areas more precisely. Also, a larger template library is harder to manage, there is more difficulty to trace, debug and write new code: techniques employed for easier developpement include reducing compile times, enforcing stricter warnings and checks, supporting for more compilers and more architectures, simplifying and automatising version number changes, automatising memory leak checks, setting up buildbots to check the code frequently,...

But this increase also forces the library to be more user friendly. For instance, we have: Developed an `auto-install.sh` script that installs automatically the lastest stable or developpement versions of the trio, resolving the version dependancies; Facilitated the discovery of the BLAS/LAPACK libraries; Simplified and sped up the checking process while covering more of the library (△dave ?); Added comprehensive benchmarking tools,...

⁶ See <http://www.linalg.org>.

⁷ Using `sloccount`, available at <http://sourceforge.net/projects/sloccount/>.

⁸ See <http://www.linalg.org/projects/fflas-ffpack/>.

Developping generic and high-performance libraries is difficult. We can find a large litterature on coding standards and software design references in (*cf.* [1,10,14,13,12]), and many internet sources and a lot of experience acquired by/from free software projects.

We are going to describe the advancement in the design of LinBox in the next three sections. We will first describe the new *container* framework in Section 2, then improve the *matrix multiplication* algorithms in Section 3 by contributing special purpose matrix multiplication plugings, and finally present the new *benchmark/optimisation* architecture (Section 4). \triangle develop this § more later

2 Containers architecture

LinBox is mainly conceived around the RAII concept with re-entrant function (Resource Acquisition Is Initialisation), introduced by [13]. 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 as data objects (containers). The fragmentation of the containers into various matrices and blackboxes 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

Firstly, in order to allow operations on its elements, a container is parametrized by a field (*cf.* Listing 1.1), not the element type; this is also more general. The storage type is given by another template parameter that can default to *e.g.* dense BLAS type matrices (a stride and a leading dimension or an increment).

```

5  template< class _Field, class _Storage = denseDefault >
    class Matrix ;

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

```

Listing 1.1: Matrix and vector classes in LinBox.

In the mother model, we need types that own or and types that share some memory. The **SubMatrix** and **SubVector** types share the memory while **Matrix** and **Vector** own it. The common interface shared by all matrices is the **BlackBox** interface described in the following paragraphs.

Input/Output. Our matrix all read and write from MatrixMarket format (ref, link). Adding extra comments ? (for instance the `init` field function in GF(q)

needs a polynomial...) We can adapt the header to suit our needs. In particular write matrix in CSR fashion (saving roughly 1/3 space over COO)

Accessing Elements. The function `setEntry(...)` can be used to populate/-grow the matrix (from some `init()` until a `finish()` is emitted). The function `setEntry` can be (very) costly (for some sparse formats for instance) (Dave ?)

- `refEntry` that retrieves a 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 (can always be implemented from `imply`, *cf.* later.
- `clearEntry` can be used to zero out an entry, especially for a sparse matrix, if this is allowed (possibly not for structured matrices).
- iterators may be difficult to implement (but a lot of code relies on them...). Do we want only `const` iterators ?

Apply method. Described later

Rebind Rebind from one field to the other

Other Conversion mechanism Added to the interface is a `convert` method to/from CSR (default) for sparse matrix.

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 $y \leftarrow A^T x$ or $y \leftarrow Ax$ is performed.

```

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

5 // y = alpha y + beta A.x
template< class _In, class _Out >
_Out& applyAcc(_Out &y, const Element& alpha, const _In& x, const
      Element& beta, 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).

Matrix Formats Dense matrix for instance `BLAS`, inherits from the iterators of `std::vector`. All other matrices (including the special case of `Permutations`) : the same `Structured` matrices, `add`, `sub`, `stacked`,...Now special case of `Sparse Matrix`.

2.3 The Sparse Matrix Case

Sparse matrices are usually problematic because the notion of *sparsity* is too general and the matrices we use are usually very specific: the algorithms have to adapt to the shape of the sparse matrices. Getting the best performance for an sparse matrix as an **BlackBox** is not an easy task. In [4] we developped some techniques to improve the SpMV(Sparse Matrix Vector multiplication). There is a huge literature on sparse matrix formats and SpMV, some of which are becoming standard. Numerical analysis brings various shapes for sparse matrices and numerous algorithms and routines. Just like the BLAS numerical routines, we would like to take advantage of existing high performance libraries. They are however not very widespread, for instance sparse blas in intel mkl (only). Metis for graph partitionning. zero-one matrices are special (no numerical routines)

Legacy LinBox sparse matrix formats: STL structures (map,...) Helper structure to store a matrix as a sum of structures (HYB), possibly the transpose. Memory. Reduce inits on $\mathbb{Z}/p\mathbb{Z}$.

XXX timing new matrices, example rank.

3 Improving LinBox matrix multiplication

Efficient matrix multiplication is key to LinBox library.

3.1 Plugin

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.

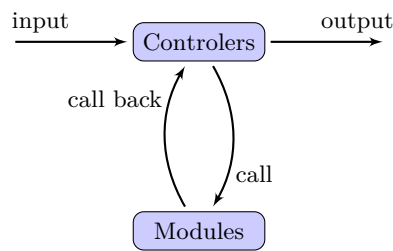


Fig. 1: Controller–Module design pattern

For instance, we can write the standard cascade algorithms in that model:

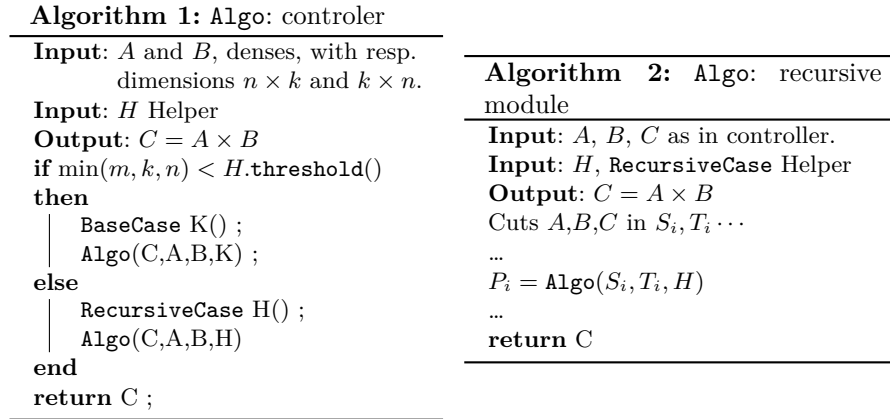


Fig. 2: Conception of a recursive controlled algorithm

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 top of this design, we have Methods/Helpers that...

△ timing old fgemmm/plugin fgemmm with no noticeable change ?

3.2 New algorithms/infrastructure

We introduce now 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.

New algorithms: low memory ffgem in FFLAS uses the classic schedules for the multiplication and the product with accumulation (*cf.* [5]), but we also implement the lower memory routines therein.

The difficultly consists in using the part of the memory contained in a sub-matrix of the original matrix. It is two-fold. – First we use some part of a memory that has already been allocated to the input matrices, therefore we cannot free and reallocate part of it. – Second, several of these algorithms are meant for square matrices and rectangular sub-matrices will just not be enough. For instance,

△ table comparing speeds

New algorithms: Bini [3]

integer blas △ pascal

Polynomial Matrix Multiplication [△](#)Pascal**OpenCL** [△](#)dave**Sparse Matrix–Vector Multiplication** [△](#)brice**Using conversions**

- - double->float
- - using flint for integer matmul is faster, even with conversion. Need better CRA implementation.
- - implementation of Toom-Cook for GF(q)
- - when does spmv choose to optimise ?
- - transition to benchmarking

4 Benchmarking

Benchmarking was introduced in LinBox for several reasons. It would give the user a user-friendly way for producing nice graph with no necessary knowledge of the graphing library `gnuplot`⁹ or provide LinBox website with automatic new tables/graphs. It would be used for regression testing. It can be used for selecting default method, threshold. (cite atlas and?)

XXX needs introductory bench references
 XXX BTL¹⁰/eigen [△](#)dave benchmark formats

4.1 Graph/Table creation

XXX data/plot structure of abstraction (update code 6.1 with new plot structure)

XXX Time spent on each data is limited (will not start execution if fit (linear least squares) forecasts 'too long').
 XXX Adapts to the environment.

4.2 Regression Testing

XXX Tests are supposed to be comparable: regression testing.
 XXX howto.

4.3 Method Selecting

XXX Default are provided, method can be selected via a benchmark (cf `wino_threshold`)

XXX howto

⁹ <http://www.gnuplot.info/>

¹⁰ <http://projects.opencascade.org/btl/>

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, 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.
10. E. Gamma. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.
11. P. Giorgi. *Arithmétique et algorithmique en algèbre linéaire exacte pour la bibliothèque LINBOX*. PhD thesis, École normale supérieure de Lyon, Dec. 2004.
12. 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.
13. B. Stroustrup. *The design and evolution of C++*. Programming languages/C++. Addison-Wesley, 1994.
14. H. Sutter and A. Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, And Best Practices*. The C++ In-Depth Series. Addison-Wesley, 2005.
15. W. J. Turner. *Blackbox linear algebra with the LINBOX library*. PhD thesis, North Carolina State University, May 2002.