Elements of Design for Containers and Solutions in the LinBox library

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. 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 serveral 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 produces 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 LinBox[6] (*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 size[7] of LinBox and its dependencies in terms of "lines of code". This increase affects the library in several ways. First, it demands a stricter development model, and we are going to list some techniques we used. For instance, we have transformed FFLAS–FFPACK[8] (*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 an maintainability of the library, focusing the development 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 development 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 build-bots to check the code frequently,...

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).

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 latest stable or development versions of the trio, resolving the version dependencies; Facilitated the discovery of the Blas/Lapack libraries; Simplified and

[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/.

sped up the checking process while covering more of the library (⚠dave ?); Updated the documentation and created user and developer oriented docs; Added comprehensive benchmarking tools,...

Developing generic and high-performance libraries is difficult. We can find a large literature 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. 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 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 plug-ins, and finally present the new *benchmark/optimisation* architecture (Section 4). ⚠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).

```
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 hearder 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, wether the operation $y \leftarrow A^\top x$ or $y \leftarrow Ax$ is performed.

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

// 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 Z/pZ.

XXX timing new matrices, example rank.

3 Improving LinBox matrix multiplication

Efficient matrix multiplication is key to LinBox library.

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.

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

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.

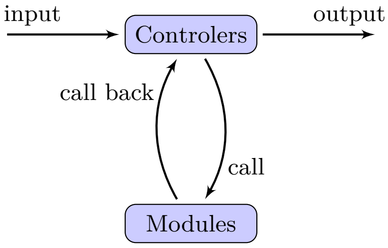

Fig. 1: Controller–Module design pattern

```
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 $\min(m,k,n) < H.\texttt{threshold}()$
then
    BaseCase K() ;
    Algo(C,A,B,K) ;
else
    RecursiveCase H() ;
    Algo(C,A,B,H)
end
return C ;
```

```
Algorithm 2: Algo: recursive
module
Input: $A$, $B$, $C$ as in controller.
Input: $H$, RecursiveCase Helper
Output: $C = A \times B$
Cuts $A$,$B$,$C$ in $S_i, T_i \cdots$
...
$P_i = \texttt{Algo}(S_i, T_i, H)$
...
return C
```

Fig. 2: Conception of a recursive controlled algorithm

On top of this design, we have Methods/Helpers that allow (preferred) selection of algorithms and cut short in the strategy selection of Figure 1.

⚠timing old fgemm/plugin fgemm with no noticeable change ?

This infrastructure also forces to modularise the code. For instance, a lot of work has been done in FFLAS–FFPACK to factor code in modules (addition, scaling, initialisation,...). Not only this permits to write code with hardly more line than it takes for pseudo-code listings in [5] (compared to ≈ 2.5× on some routines before) but also it automatically brings performance, because we can the separately improve on these modules. Also, this reduces the lines of code, hence the probability for bugs, and eases the tracing/traking of bugs, allows for more unit tests. Modularising the code comes at almost no cost because we may add $O(1)$ operations that 1) don't cost much compared to $O(n^2)$ or more complexity of the modules; 2) allow early decisions and terminations by testing against 0, ± 1 or checking the leading dimensions and increments; 3) allow better code (AVX, SSE, copy–cache friendly operation–copy back, representation switching,...)

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 The routine `fgemm` 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 difficulty consists in using the part of the memory contained in a submatrix 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 (but with the plugins, we can do without our faulty code and just use flint).
- – - 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. First, It would give the user a user-friendly way for producing quality graph with no necessary knowledge of a graphing library like gnuplot[9] 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,...).

What do we do differently ? Selection between ”larger” algorithms, takes more time. Interpolation.

4.1 Graph/Table creation

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 ploting in standard image formats, html and LaTeXtables, but also in raw csv or xml. The last raw formats allow for file exchange, data comparisons and extrapolation. ⚠dave benchmark formats discussion ?

4.2 Regression Testing

Saving graphs in raw format can enable automatic regression testing on the buildbots. For some determined matrices (of different shape and size) over a few fields, we can accumulate over time the timings for some of our solutions (rank, det, mul,...). At each new release, when we update the documentation, we can check any regression on these base cases and automatically update the regression plots. ⚠We need to implement this framework (not difficult; anybody?).

4.3 Method Selecting

CPU throttling for ATLAS, FFLAS–FFPACK not reliable. XXX Default are provided, method can be selected via a benchmark (cf wino_threshold)
XXX howto

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

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.