Elements of Design for Containers and Solutions in the LinBox Library
Brice Boyer, Jean-Guillaume Dumas, Pascal Giorgi, Clément Pernet, B. David Saunders

To cite this version:

HAL Id: hal-01015138
https://hal.archives-ouvertes.fr/hal-01015138
Submitted on 25 Jun 2014

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.
Elements of Design for Containers and Solutions in the LinBox library
Extended abstract

Brice Boyer
Department of Mathematics, North Carolina State University, USA
bbboyer@ncsu.edu

Jean-Guillaume Dumas
Laboratoire J. Kuntzmann, Université de Grenoble, France
Jean-Guillaume.Dumas@imag.fr

Pascal Giorgi
LIRMM, CNRS, Université Montpellier 2, France
pascal.giorgi@lirmm.fr

Clément Pernet
Laboratoire LIG, Université de Grenoble et INRIA, France
clement.pernet@imag.fr

B. David Saunders
University of Delaware, Computer and Information Science Department, USA
saunders@udel.edu

Abstract

We describe in this paper new 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 controllers for subtasks. 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; Creating a framework for automatically tuning the library and supporting regression testing.

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

4This material is based on work supported in part by the National Science Foundation under Grant CCF-1115772 (Kaltofen) and Grant CCF-1018063 (Saunders)
5This 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 concerning LinBox\(^1\) (cf. [2,7,8,13,19]) and builds upon them. LinBox is a C++ template library for fast and exact linear algebra, designed with generality 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. The evolution is also motivated by developing a high-performance mathematical library available for researchers and engineers that is easy to use and help produce quality reliable results and quality research papers.

Let us start from a basic consideration: we show in the Table 1 the increase in the “lines of code” size\(^2\) of LinBox and its coevolved dependencies Givaro and Fflas–Ffpack\(^3\). 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 (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, and setting up buildbots to check the code frequently.

This size increase also requires more efforts to make the library user friendly. For instance, we have: Developed scripts that install automatically the latest stable/development versions of the trio, resolving version dependencies; Eased the discovery of Blas/Lapack libraries; Simplified and sped up the checking process, 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 draw from many internet sources and experience acquired

<table>
<thead>
<tr>
<th>LinBox</th>
<th>1.0.0(^†)</th>
<th>1.1.0(†)</th>
<th>1.1.6(†)</th>
<th>1.1.7(†)</th>
<th>1.2.0</th>
<th>1.2.2</th>
<th>1.3.0</th>
<th>1.4.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>loc (×1000)</td>
<td>77.3</td>
<td>85.8</td>
<td>93.5</td>
<td>103</td>
<td>108</td>
<td>109</td>
<td>112</td>
<td>135</td>
</tr>
<tr>
<td>Fflas–Ffpack</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>1.33</td>
<td>1.40</td>
<td>1.43</td>
<td>1.50</td>
<td>1.80</td>
</tr>
<tr>
<td>loc</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>11.6</td>
<td>23.9</td>
<td>25.2</td>
<td>25.5</td>
<td>32.1</td>
</tr>
<tr>
<td>Givaro</td>
<td>n/a</td>
<td>n/a</td>
<td>3.216</td>
<td>3.33</td>
<td>3.43</td>
<td>3.50</td>
<td>3.60</td>
<td>3.80</td>
</tr>
<tr>
<td>loc</td>
<td>—</td>
<td>—</td>
<td>30.8</td>
<td>33.6</td>
<td>39.4</td>
<td>41.1</td>
<td>41.4</td>
<td>42.8</td>
</tr>
<tr>
<td>total</td>
<td>77.3</td>
<td>85.8</td>
<td>124</td>
<td>137</td>
<td>171</td>
<td>175</td>
<td>179</td>
<td>210</td>
</tr>
</tbody>
</table>

Table 1: Evolution of the number of lines of code in LinBox.

---

\(^1\)See [http://www.linalg.org](http://www.linalg.org).


\(^3\)symbol \(^†\) when Givaro is included and \(^‡\) when contains Fflas–Ffpack
by/from free software projects. 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, in Section 3, the improved matrix multiplication algorithms made by contributing special purpose matrix multiplication plugins, and, finally, we present the new benchmark/optimization architecture (Section 4).

2 Containers architecture

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

LinBox uses a variety of container types (representations) for matrix and vectors over fields and rings. The fragmentation of the containers into various matrix and blackbox types has been addressed and simplified. The many different matrix and vector types with different interfaces has been reduced into only two containers: Matrix and Vector.

2.1 General Interface for Matrices

First, in order to allow operations on its elements, a container is parameterized by a field object (Listing 1), not the field’s element type. This is simpler and more general. Indeed, the field element type can be inferred from a value 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.

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

Listing 1: Matrix or Vector classes in LinBox.

In the founding scope allocation model, we must distinguish containers that own (responsible for dynamically allocated memory) and containers that share memory of another. SubMatrix and SubVector types share the memory; Matrix and Vector own it. All matrix containers share the common BlackBox interface described in the next paragraphs, it accommodates both owner and sharer container types, and defines the minimal methods required for a template BlackBox matrix type:

**Input/Output.** Our matrix containers all read and write from Matrix Market format which is well established in the numerical linear algebra community and facilitates sharing matrices with other software tools. The MatrixMarket header comment provides space for metadata about the provenance of a matrix and our interest in it. However, because of our many entry domains and matrix representations, extensions are necessary to the MatrixMarket format. For instance, the header comment records the modulus and irreducible polynomial defining the representation of a matrix over GF($p^e$). We can further adapt the

---

header to suit our needs, for instance create new file formats that save space (e.g.
CSR fashion saves roughly a third space over COO, cf. Harwell-Boeing format).
Structured matrices (Toeplitz, Vandermonde, etc.) can have file representations
specified.

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

Rebind/Conversions. In addition to the rebind mechanism (convert from
one field to the other), we add conversion mechanisms between formats, 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 some central formats are well tuned for some task.

This is a common minimal interface to all our matrix containers that can
be used by all algorithms. This interface provides the basic external function-
ality of a matrix as a “linear mapping” (black box). This interface is shared by:
dense containers (Blas-like,...); permutation containers (compressed Lapack
or cycle representation); sparse containers (based on common formats or on
STL containers such as map, deque,...); structured containers (Diagonal, Hankel,
Butterfly,...); compound containers (Compose, Submatrix,...). Additional func-
tions of a container can be added, and flagged with a trait, for example those
that support internal changes as for Gaussian elimination.

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 2), where _In and _Out are vector or
matrices, and Side is Tag::Right or Tag::Left, whether the operation
y ← Aᵀx
or y ← Ax
is performed. We also generalize to the operation
y ← αAx + βy.

```cpp
template< class _In, class _Out >
_Out& apply(_Out &y, const _In& x, enum Side);
```

Listing 2: Apply methods.

This method is fundamental as it is the building block of the BlackBox
algorithms (for instance block-Wiedemann) and as the matrix multiplication,
main operation in linear algebra, needs to be extremely efficient (Section 3).
The implementation of the apply method can be left to a mul solution, which
can include a helper/method argument if the apply parameters are specialized
enough.

3 Improving LinBox matrix multiplication

We propose a design pattern (the closest pattern to our knowledge is the strategy
one, see [6, Fig 2.]) in Section 3.1 and we show a variety of new algorithms where
it is used in the mul solution (Section 3.2).
3.1 Plugin structure

We propose in Figure 1 a generalization of the strategy design pattern of [6, Fig 2.], where distinct algorithms (modules) can solve the same problem and are combined, recursively, by a controller. The main advantage of our pattern is that the modules always call the controller of a function so that the best version will be chosen at each level. 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 (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 that of any of the algorithms individually. 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 module must be fast. On top of this design, we have Method objects that allow caller selection of preferred algorithms, shortcutting the strategy selection.

Algorithm 1: Algo: controller

| Input: | A and B, dense, with resp. dimensions n × k and k × n. |
| Input: | H Helper |
| Output: | C = A × 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 × B Cut A,B,C in S_i, T_i |
| ... |
| P_i = Algo(S_i, T_i, H) |
| ... |

This infrastructure supports modular code. For instance, Fflas–Ffpack has seen major modularization (addition, scaling, reduction,...) Not only does it enable code to be hardly longer than the corresponding pseudocode listings, [5], (compared to ≈ 2.5× on some routines before) but it also automatically brings performance, because we can separately improve individual modules and immediately have the benefit throughout the whole library.

3.2 New algorithms for the mul solution

New algorithms and techniques improve on matrix multiplication in several ways: reducing memory consumption, reducing runtime, using graphics capa-
bilities, generalizing the Blas to integer routines.

Reduced memory. The routine \texttt{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 contained in a submatrix of the original matrix, that one cannot free or reallocate.

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 \texttt{fgemm} by \(\approx 5 - 10\%\) on sizes 1500–3000. This is a cascade of Bini’s algorithm and Strassen–Winograd algorithm and/or the naïve algorithm (using Blas). The idea is to analyze 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 multimodular approach through the Chinese remainder theorem. Our approach is to reduce as much as possible to \texttt{fgemm}. Despite, the existence of fast multimodular reduction (resp. reconstruction) algorithm [12], the naïve quadratic approach can be reduced to \texttt{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 \texttt{fgemm} for pointwise matrix multiplications. One can find some benchmark of our code in [14].

Sparse Matrix–Vector Multiplication. For sparse matrices a main issue is that 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 SpMV operation 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 independent and using specific routines. For instance, on \(\mathbb{Z}/p\mathbb{Z}\) 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 tradeoff 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 storage of the transpose in an optimized fashion, usually yielding huge speedups. This can be decided by \textit{ad hoc} optimizers.

Work on parallelizations using OpenCL, OpenMP or XKaapi for dense or sparse matrix multiplication include [4,9,20].
4 Benchmarking for automated tuning and regression testing

Benchmarking was introduced in LinBox for several reasons. First, it gives the user a convenient way to produce quality graphs with the help of a graphing library like gnuplot\(^5\) and provides the LinBox website with automatically updated tables and graphs. Second, it can be used for regression testing. Finally, it will be used for selecting default methods and setting thresholds in installation time autotuning.

4.1 Performance evaluation and Automated regression testing

Our plotting mechanism is based on two structures: \texttt{PlotStyle} and \texttt{PlotData}. The \texttt{PlotGraph} structure uses the style and data to manage the output. We allow plotting in standard image formats, html and \LaTeX{}tables, but also in raw csv or xml for file exchange, data comparisons and extrapolation. This mechanism can also automatically create benchmarks in LinBox feature matrix (this is a table that describes what solutions we support, on which the fields).

Saving graphs in raw format can also enable automatic regression testing on the buildbots that already checked our code. For some specifically determined matrices (of various shapes and sizes and over several fields), we can accumulate the timings for key solutions such as (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 Automated tuning and method selection

Some of the code in LinBox is already automatically tuned (such as thresholds in \texttt{fgemm}), but we improve on it. 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 may even tolerate outliers (but this is time consuming).

Automatically tuning a library is not only about thresholds, it may also involve method/algorithm selection. Our strategy is the following: a given algorithm is tuned for each \texttt{Helper} (method) it has. Then the solution (that uses these algorithms) is tuned for selecting the best methods. At each stage, defaults are given, but can be overridden by the optimizer. The areas where a method is better are extrapolated from the benchmark curves.

References


\(^5\)http://www.gnuplot.info/


