P1417R0: Historical lessons for C++ linear algebra library standardization

Mark Hoemmen <mhoemme@sandia.gov>,
Jayesh Badwaik <badwaik.jayesh@gmail.com>,
Matthieu Brucher <matthieu.brucher@gmail.com>,
Athanasios (Nasos) Iliopoulos <apiliopou@gmail.com>, and
John Michopoulos <ymichopoulos@gmail.com>

January 21, 2019

Contents

1 Introduction

When regular WG21 participants began discussing standardization of a linear algebra library, many features, issues, and terms came up that had appeared in linear algebra libraries many years before. We were encouraged to see new proposals "rediscovering" good ideas, but we also wanted to make sure that participants had the chance to learn from history. To this end, we wrote this survey of lessons learned from linear algebra libraries and standardization efforts, in C++ and other languages (including Fortran, MATLAB, and Python). Fortran discussions focus on the BLAS [17], LINPACK [13], and LAPACK [2] libraries, with their long and successful history and frequent use in C++ linear algebra libraries and applications. This survey also puts early C++ linear algebra libraries into a larger historical context, to help expose motivations for design decisions.

This document does *not* attempt a comprehensive survey of linear algebra libraries in any language. Omission does not imply judgment. Furthermore, the authors freely admit that their background in scientific and engineering computation will introduce bias. The point is not to claim objectivity or completeness, but rather to tell some stories that could help guide the development of a standard C++ linear algebra library. To this end, one of the authors, Mark Hoemmen, includes anecdotes from his experiences working on the Trilinos C++ project [20] since 2010, and oral histories that he collected circa 2018 from a POOMA project developer, Chris Luchini [23]. Two other authors, Nasos Iliopoulos and John Michopoulos, list lessons learned from their experiences as `boost.uBlas` developers. Another author, Matthieu Brucher, contributes lessons learned from Python. Not all authors necessarily agree completely with each other about all the opinions they express here, but we think readers would prefer a diversity of views at this point. This document also refers to an interview of Jack Dongarra taken for the Society of Industrial and Applied Mathematics in 2005 [14], and Cleve Moler's 2004 article on the history of his creation, MATLAB [41]. We also incorporate the fruits of discussion from many

other contributors, not all of which we were able to credit here. Oral histories carry subjectivity, but nonetheless can help readers understand design motivations.

We begin with Section 2, an overview of 1990s trends in computer hardware and applications that shaped early C++ linear algebra libraries. Section 3 highlights features of C++ linear algebra libraries, both historical and current, that we think should guide development of a standard library. In Section 4, two authors give lessons they learned from their experiences as the primary `boost.uBlas` developers. Section 5 surveys lessons learned from the BLAS, LINPACK, and LAPACK. We describe some other standardization efforts that extend the BLAS in Section 6. Section 7 talks about MATLAB, and Section 8 outlines experiences from the Python community. We conclude in Section 9 with our opinions on what some goals of a C++ linear algebra standardization effort should be.

2 1990s trends leading to C++ linear algebra libraries

Two key features of modern C++ – templates, and the Standard Template Library – both evolved in the 1990s. That time also spawned other trends concurrent to and possibly shaping the development of early C++ linear algebra libraries. Surveying these trends can help us see why we might want such libraries in the first place, and what motivated the libraries' design decisions.

We begin with object-oriented numerics in Section 2.1. Section 2.2 talks about the democratization of high-performance computing, and Section 2.3 about increasing complexity in numerical simulations. We finish with the democratization of computer graphics in Section 2.4.

2.1 Object-Oriented Numerics

Summary: C++ library developers eagerly applied object-oriented programming to numerical computation, including numerical linear algebra. This made libraries easier to use and more reusable. Developers learned to use expression templates to avoid performance issues of naïve encapsulation. They also introduced a new interface design antipattern by failing to factor classes by responsibility.

Linear algebra libraries in C++ at first evolved from the general category of "object-oriented programming" (OOP). One can see the explosion of interest in so-called "object-oriented numerics" by the large variety of conferences that sprang up in the mid-1990s.[1] For example, the First annual Object-Oriented Numerics Conference (OON-SKI) took place in 1993.[2] Rogue Wave, a C++ compiler company, sponsored OON-SKI, so the conference logically focuses on work in C++. "Numerics" here means "numerical computation" or "scientific computing": computations on large arrays of floating-point numbers, doing things like simulating the physical world or analyzing large quantities of data statistically.

Numerical library developers of the time saw OOP as a way to "keep the CPU-time constant while reducing the human time drastically" [5]. The technique encourages modularity and encapsulation, and thus simplifies testing and reuse of software components. It lets developers create "flexible applications that can be easily extended. . . . In short, OOP encourages computer implementations of mathematical abstractions" [5].

[1] See e.g., http://www.math.unipd.it/~michela/OP.htm#conferences.

[2] See https://www.hindawi.com/journals/sp/si/250702/.

2.1.1 Early lessons learned

Summary: in the early 1990s, C++ developers working on object-oriented numerics had already learned the following lessons:

- Libraries want users to write linear algebra expressions in code that look as much like mathematical notation as possible.
- C++ linear algebra libraries can encapsulate some of the details of getting good performance, including parallel computation.
- Naïve abstraction can hurt performance.
- C++ expression templates can help improve performance while enabling interfaces that look like mathematical notation.

The introduction to the journal special issue for OON-SKI 1993 described the phenomenon of "object-oriented numerics" as follows:

> ... [W]e are observing the emergence of a subdiscipline: the use of object-oriented techniques in numerics. But what we are really seeing is something even more profound: finally rejoining of scientific computing with the science of computers. Traditionally, programming has been done by engineers, physicists and mathematicians with little or no training in computer science. Now, however, we are seeing an infusion of ideas coming from [the] computer science world into the scientific computing world, bringing along modern ideas on how to structure complex numerical code. Object-oriented techniques is [sic] merely one of many such ideas [53].

The introduction talks about issues like needing to teach compilers how to fuse loops and avoid temporaries when doing overloaded-operator arithmetic on arrays in C++. It also shows the existence of C++ libraries for a variety of applications, including a library of multidimensional arrays, and the integration of C++ with distributed-memory parallel computation. This shows that C++ developers working on object-oriented numerics had already learned the lessons mentioned in the summary above.

2.1.2 Antipattern: Classes with too many responsibilities

Summary: Some early object-oriented numerics libraries have objects that can be in many different states, and many mutating instance methods with state-dependent preconditions. This is an antipattern that adds complexity for both users and developers. Application of the One Responsibility Rule and idiomatic use of the modern C++ type system can correct this without sacrificing performance.

It's easy to take the phrase "object-oriented programming" for granted, as equivalent to "modern" or "not old fashioned." Proponents of object-oriented numerics contrasted their interfaces with old-fashioned Fortran code: routines with cryptically short names that take dozens of arguments and have thousands of lines of code, and single objects with state scattered over many separate variables [5]. Object-oriented interfaces grouped together state into nouns (e.g., "matrix," "vector") and verbs (e.g., "dot product," "matrix-vector multiply") that users found self-documenting.

This is overall a good thing. For example, contrast ScaLAPACK's parallel dense LU factorization interface, with its long list of mysterious arguments and complicated parallel distribution setup boilerplate, with the concise interfaces to the same functionality in the Elemental [45] or Global Arrays

[31] libraries. However, a common design antipattern in early object-oriented numerics libraries, is to let an object of a single type have many possible states, with many mutating instance methods whose preconditions depend on the object's state.[3] This approach seems natural for many scientific and engineering applications, that have a small number of "objects" representing large memory allocations that the application modifies in place. However, we consider this an "antipattern" for three reasons:

1. The resulting classes violate the One Responsibility Rule.[4]

2. Users and library developers both find it hard to keep in mind what operations are valid on an object at what times.

3. This approach fails to use C++'s type system and language features to help make the current state obvious to users.

What we mean by the third point above, is that a library could represent state transitions as *type* transitions, that consume an object of the old type to produce an object of the new type. "Consuming" an object could happen via move construction. C++ has "smart pointers" that the library could use to let ownership of large memory allocations pass from the old object to the new one, without need for reallocation and copying.

As an example of this antipattern, consider sparse matrices (`Epetra_CrsMatrix`) in the Trilinos project's Epetra linear algebra library [20]. One creates an Epetra sparse matrix by giving its constructor an object representing the parallel distribution of its rows. At that point, the matrix is literally empty. One then calls mutating methods on the matrix to insert sparse matrix entries, one or more at a time. After one is done inserting, one must call the matrix's `FillComplete` method in order to prepare the matrix for linear solves. The matrix's type stays the same through all of these mutations. Its type tells one nothing about what operations are currently legal on the object. For example, one can only change the matrix's sparsity structure *before* calling `FillComplete`, and one can only solve linear systems with the matrix *after* `FillComplete`. The matrix class has several different constructors besides the one mentioned here, and the set of valid pre-`FillComplete` operations on the matrix (e.g., whether one is allowed to change its sparsity structure, or whether one needs to use local or global column indices when changing entries) depends entirely on which constructor one uses. Changing anything inside this class is risky, since most unit tests and code exercise only common cases that do not cover all states. It would have been better for Epetra to have had separate classes to represent various kinds of "half-baked matrix" vs. a "fully baked" matrix.

Note that even though the Sparse BLAS interface standard gives matrices to users by integer handle rather than by instance of a C++ class, it shares the same property that the handle does not change after the equivalent of `FillComplete` [17, pp. 129–30].

2.2 Democratization of high-performance computing

A phenomenon we call the "democratization of high-performance computing" may have also contributed to the development of C++ linear algebra libraries. This phenomenon has three parts. First, high-performance, lower-cost workstations let users solve more problems without resorting to expensive "big

[3] See e.g., [5, p. 6]. It's a bit unfair to pick on this one source, as the authors can think of many examples, including codes on which the authors work.

[4] See e.g., http://wiki.c2.com/?OneResponsibilityRule.

iron."[5] Workstation hardware tracked the 18-month Moore's Law performance curve, so users could just wait for the next processor generation, instead of expensively optimizing their code. This gave developers more freedom to write more complicated codes. In turn, this moved them towards programming languages like C++, that aid encapsulation and larger-scale software architecture.[6]

Second, distributed-memory parallel computers emerged. These new systems did not rely on custom vectorizing Fortran compilers, as did the older generation of expensive vector supercomputers. Some of these systems did not even have a Fortran compiler, or strongly favored other programming languages.[7] New systems had new programming models anyway, which encouraged writing new libraries in languages like C++.

Third, distributed-memory parallel computation and standard programming models made it cheaper and easier to build high-performance computers with large memory capacities. The rapid increase in workstation performance meant that users often only reached for "supercomputers" when they need to solve problems too big to fit on a single workstation. Techniques like Network of Workstations let users assemble a parallel computer with a large memory, out of the large number of inexpensive workstations they already had [4]. Standard parallel programming models like MPI[8] and OpenMP[9] made Fortran less important and simplified writing standard libraries. The emerging World Wide Web in turn accelerated such libraries' promulgation.

2.3 Increasing complexity in numerical simulations

Concurrent with the above trends was an increasing demand for more complex numerical simulations. The United States Department of Energy's Accelerated Strategic Computing Initiative (ASCI) program, founded in 1996 [38], is one example of a program that helped spur this demand. An important part of ASCI was "[s]oftware design and development" for complicated "multi-physics scientific applications." Before ASCI, experience with such software development was "limited to a few isolated projects." ASCI aimed to "carry out multiple substantial simulation projects that will provide proofs of principle, at scale, of software development methodologies and frameworks that provide portability, extensibility, modularity, and maintainability while delivering acceptable performance" [38].

[5] This is part of the "killer micros" revolution. Killer micros were consumer-grade, low-cost microprocessors whose performance threatened and eventually overtook that of expensive vector supercomputers [34].

[6] Fortran 90 has modules, but no "instances of a class." Fortran 2003 has features more like C++ instance methods and polymorphism. Note, however, that Fortran compilers tend to lag behind the latest Fortran standard, much more than C++ compilers lag behind the latest C++ standard. The authors' general experience with relying on Fortran 2003 features has been poor. For example, the use of these features in the ForTrilinos project proved unsustainable, and drove a shift back to assuming a modest subset of Fortran 90 on the Fortran side of this interface to the C++ linear algebra library Trilinos.

[7] For example, the technical report describing Connection Machine's CM-1 only mentions one programming language: *Lisp, an ANSI Common Lisp derivative [27]. Applications running on CM-2 also used C*, a parallel superset of ANSI C. CM-2 only later added CM Fortran, an extension of Fortran, in 1991 [29, p. 7]. Other distributed-memory parallel computers of the era favored C and offered C++ compilers. Library-oriented approaches to parallel computing, like PVM and MPI, came with C as well as Fortran bindings.

[8] The Message Passing Interface, whose Version 1.0 came out in 1994.

[9] A standard for directives-based shared-memory parallel programming, whose C and C++ interface came out in 1997.

2.4 Democratization of computer graphics

The 1990s also saw a standardization of programming models for computer graphics. For example, OpenGL 1.0 came out in 1992 [19], and DirectX not long after.[10] Along with this came new hardware and instruction sets, like MMX [40] and 3DNow! [43], for accelerating graphics operations on consumer processors. Computer graphics depends heavily on performing many small dense linear algebra operations. This is a use case that historically did not cross over much with non-graphics scientific computation. However, the scientific computing community is starting to show more interest in so-called "batched" linear algebra interfaces for performing many small dense operations in parallel [15].

2.5 Summary of 1990s trends

A massive change in high-performance computer architectures and programming models led software developers to start using C++ for linear algebra libraries. Concurrently, increasing interest in computer graphics, and the availability of dedicated instruction sets for accelerating primitive operations needed for graphics, stirred a growing interest in linear algebra outside of traditional scientific and engineering computing circles. C++ linear algebra library developers discovered that naïve encapsulation could have a performance cost, and that C++ techniques like expression templates could help reduce or eliminate this cost.

3 Features of C++ linear algebra libraries

In this section, we highlight common features of early C++ linear algebra libraries, that we think express good lessons learned for any C++ linear algebra library standardization effort.

3.1 Templates to improve performance

In summary:

- C++ code had a poor reputation for performance, compared with optimized C or Fortran code.
- C++ templates, including expression templates, helped close the "performance gap" with optimized Fortran code.
- C++ compilers of the 1990s did not necessarily have complete or correct implementations of templates. Some linear algebra libraries excluded templates as a result.
- Compilers are better at compiling templates now, so we don't have to be afraid to use them. However, we need to respect concerns about compilation time.
- With expression templates, users must be careful using `auto` for the left-hand side of an expression assignment.

[10] See, e.g., the following overview of the history of OpenGL and Direct3D: https://news.ycombinator.com/item?id=2711231.

C++ had a reputation for poor performance compared with Fortran. Even developers willing to write C++ in "numerical" codes considered it better to use C++ as a high-level coordination language, and reserve lower-level languages like C or Fortran for tight loops.[11] Developers saw C++ templates, in particular expression templates, as an optimization technique that could close the performance gap. Expression templates would let developers write compact, abstract code that "looks like math," yet optimizes by fusing loops and avoiding temporaries. For example, the Dr. Dobbs article [51] on the Blitz++ library [50], written by the library's author, focuses on expression templates for vector operations.

Developers also recognized the cost of virtual method calls in C++, especially in inner tight loops, and used templates to reduce the cost of run-time polymorphism. For example, the Bernoulli Generic Matrix Library uses the "Barton-Nackman trick" [8], a special case of the "Curiously Recurring Template Pattern," to turn run-time polymorphism into compile-time polymorphism.[12] This pattern shows up in other C++ linear algebra libraries; for example, Eigen uses it for expression base classes.[13]

Early libraries that relied on templates suffered due to incomplete compiler implementations. For example, Blitz++'s installation process exercises the compiler to test language feature compliance. Its User's Guide recommends that if the compiler "doesn't have member templates and enum computations, just give up."[50, Section 1.4.3] A comparable library, POOMA (Parallel Object-Oriented Methods and Applications),[14] pushed the boundaries of what the available C++ compilers could handle. Chris Luchini, a POOMA developer, recalls that the project exposed many compiler bugs [23]. Many compilers lagged behind the C++ standard, only implemented a subset of features, and generated slow code [35].

Software for scientific computing may need to build with several different compilers and run on different kinds of hardware. Lack of consistently complete implementations of templates challenged portability requirements and restricted adoption. For example, in the Trilinos software project, a requirement to support a C++ compiler with incomplete template support drove the project to forbid templates in its foundational linear algebra library, Epetra [23].

One of the authors, Mark Hoemmen, has had experience reducing compile times and compiler memory usage of the Trilinos [20] and Kokkos[15] projects. He has put months and months of developer time into efforts like the following:

- explicit template instantiation;
- splitting explicit instantiations into separate files, sometimes automatically, through build system scripts; and
- reducing the number of separate instantiations of a templated class that add no value, because they generate identical code.

Indiscriminate use of templates can increase compile times, both by preventing implementation hiding, and by proliferating instantiations. Compilers sometimes generate slower code when a compilation unit is too complicated or takes too long to build, since compilers want to bound the time and memory they use. However, neither of these discourage most developers these days from using templated functions

[11] See [6]. The Epetra linear algebra library in the Trilinos project [20] has optional Fortran implementations of sparse matrix times multiple dense vectors at a time, since it found them faster than C++. In 2010–11, one of the authors found Fortran versions of dense QR factorization kernels faster than equivalent C++ versions [22].

[12] In [35], authors cite [52].

[13] See http://eigen.tuxfamily.org/dox/TopicClassHierarchy.html.

[14] See http://www.nongnu.org/freepooma/tutorial/introduction.html.

[15] See https://github.com/kokkos/kokkos.

and classes in the C++ Standard Template Library. We think it is possible and good to use templates, but developers need to think about compilation time. Users must also take responsibility for this, by designing interfaces to hide implementations as much as possible, and breaking up long compilation units.

Expression templates may hinder use of `auto` for the left-hand side of expressions, and users need to be careful returning expressions from a function without first assigning them to some concrete linear algebra type. This is because the type to which the expression evaluates, is not necessarily a concrete matrix or vector or other linear algebra object. It may be some expression type, that may hold references to concrete linear algebra objects.[16] Returning the expression may result in dangling references, if the concrete linear algebra objects to which the expression refers have fallen out of scope. C++ copy elision[17] makes it impossible for a library to detect and report user code that creates expressions on the left-hand side, such as `auto z = x + y`.

3.2 Generic iteration over multidimensional sequences

Summary: Developers discovered that C++ let them express generic iteration over sequences in a high-level way, and generate optimized code from the high-level specification. A linear algebra or tensor library may want to expose iteration over multidimensional sequences or index spaces. There are different ways of expressing this.

As experience with C++ templates increased, some developers applied them to more radical code optimizations. For example, the Bernoulli Generic Matrix Library used templates to generate optimized sparse matrix codes from a high-level specification [1]. Bernoulli used a kind of relational algebra (described in detail in the PhD dissertation) that gives users a generic way to describe operations over sequences, while optimizing by avoiding storage of temporary intermediate sequences. One can think of this as a generalization of C++ iterators, and as a precursor to the C++ Ranges proposal [42].

By the time one of the authors (Mark Hoemmen) encountered the Bernoulli project in the early 2000s, it had abandoned C++ templates in favor of a code generation framework based on OCaml (or some other ML derivative). Our guess is that using OCaml to generate C code, instead of using C++ templates, avoided C++ compiler correctness and performance issues that were more common in the early 2000s. Nevertheless, this suggests a lesson learned: the more ambitiously complex a C++ library, the more programmer effort and expertise it requires, and the bigger the risk for good performance on many platforms.

The Kokkos C++ library[18] uses a different approach for parallel iteration over multidimensional index ranges. Kokkos' `MDRangePolicy` has users specify the index ranges. They may also specify nondefault features as template parameters, like the computer architecture, the iteration order, and whether to use tiling with compile-time sizes. Kokkos uses this specification to generate parallel code. An important lesson learned from Kokkos is that users need control over multidimensional iteration order in order to get best performance.

Other approaches to multidimensional sequence iteration include Einstein notation for tensor summation. Authors Nasos Iliopoulos and John Michopoulos have experience optimizing this in a C++ library. See also Section 4.

[16] See e.g., http://eigen.tuxfamily.org/dox/TopicPitfalls.html.

[17] See https://en.cppreference.com/w/cpp/language/copy_elision.

[18] See https://github.com/kokkos/kokkos.

3.3 Engines: Implementation polymorphism

Summary: C++ template partial specialization lets library developers write linear algebra interfaces that are polymorphic on their implementations. This is an extension point, like the `Allocator` template parameter in C++ Standard Library objects. Library authors can and have used this for polymorphism on things like the parallelization mechanism, the allocator, and the storage layout.

The POOMA (Parallel Object-Oriented Methods and Applications) project was most active 1998-2000. POOMA's goal was to support structured grid and dense array computations. As per oral history [23] and POOMA's documentation,[19] the team had a particular interest in SGI Origin shared-memory parallel computers. POOMA shares features with more recent linear algebra libraries, such as polymorphism on storage layout and parallel programming model, so it is worth studying for historical lessons.

POOMA's main data structure is Array, a multidimensional array. Array has three template parameters: the rank (the number of dimensions), the entry type (e.g., `double`), and the "Engine." Engines express where and how data are stored. They implement access to entries of the Array. For example, depending on the POOMA Engine used, Array entries could either actually exist in some storage somewhere, or they could be computed on the fly at access time from their indices. POOMA Engines (in particular, the MultiPatch engine) could also describe distributed-memory parallel data distribution.

This feature of implementation polymorphism by template specialization shows up in many other libraries. For example, the `mdspan` multidimensional array proposal [16] has an Accessor policy, an optional template parameter that implements access to the entries of the `mdspan`. The Kokkos[20] library's `View` multidimensional array class has "execution space" and "memory space" template parameters, that control where data live (allocation and deallocation) and how operations execute in parallel over those data.

3.4 Deducing return type of linear algebra expressions

Summary: Figuring out the right return type of a linear algebra expression is a nonobvious design decision, that early C++ libraries have already encountered.

One recent discussion point that has proven controversial, is how to deduce the return type of an arithmetic expression involving linear algebra objects with mixed element types. For example, suppose that I have a matrix `A` with elements of type `complex<float>`, and I write the expression `4.2 * A`. Should I get a matrix with elements of type `complex<double>`, since that avoids loss of accuracy when multiplying the `double` 4.2 by `complex<float>`? What if I really want `complex<float>`, for its lower storage requirements or for compatibility with other interfaces? It's easy to write mixed-type expressions like `4.2 * A`, but not necessarily easy to decide the type to which they should convert.

This is a simple example of a general problem: allowing arithmetic expressions of linear algebra objects of mixed types means that a library must decide the return type of such expressions. The aforementioned POOMA project encountered this general issue. For example, they learned the hard way that C++ does not permit "templating on return type."[21]

[19] See http://www.nongnu.org/freepooma/tutorial.

[20] See https://github.com/kokkos/kokkos.

[21] See http://www.nongnu.org/freepooma/tutorial/tut-03.html.

3.5 Lazy evaluation may not always pay

Summary: Implementations of expression templates for linear algebra expressions may lose performance and even get the wrong answer, if they evaluate all expressions lazily. Fixing this in the library turns library developers into compiler authors and introduces a complexity / performance trade-off.

C++ expression templates are one way to implement *lazy evaluation* of arithmetic expressions involving linear algebra objects. Natural implementations of expression templates lead to fully lazy evaluation: No actual computation happens until assignment. However, multiple C++ linear algebra libraries have learned the lesson that fully lazy evaluation does not always pay. For example, for algorithms with reuse, like dense matrix-matrix multiply, lazy evaluation can give incorrect results for expressions where the output alias the input (like `A = A * A` where `A` is a matrix). This is why the Eigen library's expressions have an option to turn off lazy evaluation, and do so by default for some kinds of expressions. Furthermore, allocating a temporary result and/or eager evaluation of subexpressions may be faster in some cases.[22] This is true not necessarily just for expressions whose computations have significant reuse, like matrix-matrix multiply, but also for some expressions that "stream" through the entries of vectors or matrices. For example, fusing too many loops may thrash the cache or cause register spilling, so deciding whether to evaluate eagerly or lazily may require hardware-specific information [49]. It's possible to encode many such compilation decisions in a pure C++ library with architecture-specific parameters.[23] However, this burdens the library with higher implementation complexity and increased compilation time. Library designers may prefer a simpler interface that excludes expressions with reuse (that have correctness issues with lazy evaluation) and lets users decide where temporaries get allocated.

3.6 Vector spaces and parallel data distributions

Summary: A "vector space" is a useful generalization of the number of rows or columns of a matrix. It can pay to expose this generalization to library users, though a tensor library may find this abstraction less useful.

Linear algebra libraries all must figure out what to do if users attempt to perform operations on objects with incompatible dimensions. For example, it does not make sense to add together two vectors x and y with different lengths, or to multiply two dense matrices A and B if A's number of columns does not equal B's number of rows. The only complication is whether the library has this information at compile time, or must check at run time.

This gets more complicated, though, if we generalize "compatible dimensions" to the mathematical idea of a "vector space." Two vectors might have the same dimensions, but it still might not make sense to add them together. For example, I can't add a coordinate in 3-D Euclidean space to a quadratic polynomial with real coefficients, just like I can't add meters to seconds. Two vectors from different vector spaces may not be addable together, even though the two spaces have the same dimension or are otherwise isomorphic. Like a physical unit, a vector space is a kind of "metadata." Equating all isomorphic vector spaces strips off their metadata.

The "vector space as metadata" idea also can apply to parallel data distributions. We use the term *parallel data distribution* to refer either to a particular distribution of data over a distributed-memory parallel computer, or to the memory affinity of data on shared-memory parallel computers

[22] See http://eigen.tuxfamily.org/dox/TopicLazyEvaluation.html.

[23] See, e.g., the following 2013 Eigen presentation: http://downloads.tuxfamily.org/eigen/eigen_CGLibs_Giugno_Pisa_2013.pdf. For comparable `Boost.uBlas` optimizations, see Section 4.

with nonuniform memory access (NUMA). It may even be mathematically correct to add two vectors from the same (mathematical) vector space but with different parallel distributions. However, doing so would require communication. "Communication" here may mean different things on different parallel computers, but generally implies some combination of moving data between processors (either through a memory hierarchy, or across a network), and synchronization between processors. Communication is expensive relative to floating-point arithmetic [54, 9]. It also affects correctness, because it may introduce deadlock, depending on what surrounding code does. Thus, programmers like to see communication made explicit, even if it is hidden behind a convenient interface.

A natural way for a linear algebra library to expose parallel data distributions to users is by folding the distribution into a vector space. For example, the library's vector space would not only give the "global" dimension of a vector, but would express which vector entries "live" on which parallel processes / in which NUMA memory affinity regions. Linear algebra libraries in the Trilinos C++ software project [20] do this (e.g., in Epetra and Tpetra, a vector space / distribution is called a "Map"), and they give users a way to query vector spaces for "sameness."

One issue with this approach is how to generalize it in a usable way to tensors. Tensors have more spaces than matrices. Users may find it hard to keep track of which matter for the operations that they want to perform.

4 Lessons learned from Boost::uBlas development

Coauthors Nasos Iliopoulos and John Michopoulos had been the primary `boost.uBlas` developers for a number of years. They contribute the following lessons in this section, based on their personal experience and algebra system design reflections. Author Mark Hoemmen contributes footnotes with his views on a few of their lessons.

1. The separation between matrix and vector classes provide no real benefit. At the same time it necessitates redundant overloads and hinders the ability to develop certain generic algorithms. Other libraries like Eigen use the matrix class to represent both matrices and vectors. Vectors then can be just a typedef away.[24]

2. With the current compile-time utilities in C++, there is little reason not to base an algebra system on a tensor type (or multidimensional algebraic container) interface.

3. The distinction between vector and matrix can be easily performed using a static / dynamic dimensions mechanism.

4. Full container aliasing issues are resolved using move semantics. This can be achieved using the copy and swap idiom, or a modified version of it (evaluate to temporary and swap) in case the right-hand side is an expression and not a concrete object. This technique is limited to full-container alias resolution.[25]

[24] Hoemmen: At the lowest level of data access containers, it helps performance to know the number of dimensions at compile time. This is based on my experience with `Kokkos::View`, ancestor of `mdspan`. The `mdspan` class can represent vectors, matrices, and general tensors, by changing its `Extents` template argument. See also the remarks below, e.g., the `tensor` example.

[25] Hoemmen: "Aliasing issues" refers to expressions where the output aliases the input in a way that makes computation much harder without creating a temporary. See the dense matrix-matrix multiply example `A = A * A` in Section 3.5. For a discussion of the copy and swap idiom and an optimization that gives up the strong exception guarantee, see [48].

5. Staying away from an object-based interface has many benefits. For example, prefer `inverse(A)` to `A.inverse()`.[26]

6. Users need out-of-the-box algorithms, even though there are strong arguments to not provide them by default. In any case, access to algorithms is very important and should be a simple task.

7. Expecting users to install LAPACK to get a general matrix inverse is unfortunately too much to ask and in practice turns users away. For fixed-size containers, providing some default algorithms (e.g., matrix inverse) is an acceptable compromise.[27]

8. Performance of certain algorithms (like matrix-matrix multiply or transpose) can be increased dramatically when the implementation accounts for cache locality. That is to say, that even simple algorithms are particularly hard to implement optimally.

9. There will always be the need for customizing even the most basic of algorithms.

10. Deeply nested expression templates (like `matmult(matmult(A, B), C);`) result in redundant operations. Pattern matching and sub-expression temporaries can easily and optimally deal with this issue though.[28]

11. Sub-indexing syntax is very important for the clarity of written code. `uBlas`' sub-indexing is not as clean as it should be.

12. A descriptive multi-index traversal interface (like for example exists in Matlab, Python, Mathematica and others) is VERY powerful and is missing from C++. Index notation and / or Kokkos' `MDRangePolicy` are possible solutions.[29]

13. The number of iterator concepts you can have for matrices is very big. A possible solution is to define the simplest iterator possible (i.e., the one that optimally traverses the memory and returns coordinates and a value).

14. Mixed static / dynamic dimension specifications are very important both in terms of semantics and in terms of performance. For example a tensor with its first dimension size known at compile time, while the other not, can be defined as: `tensor<double, fixed<3>, dynamic> mat;`.[30]

15. Fixed-size containers are essential for an algebra library.

[26] Hoemmen: See Section 2.1.2, and Scott Meyer's article on how non-member functions improve encapsulation [39].

[27] Hoemmen: A library like `Boost.uBlas`, that users must download and install, differs from a component of the C++ Standard Library. Getting a project's build system to interface portably with the BLAS and LAPACK takes a perhaps surprising amount of work. However, system vendors can and do provide pre-installed optimized BLAS and LAPACK libraries. Some vendors, like Cray, make their compilers link with these libraries by default. Note that the BLAS and LAPACK were designed for large problems, and may not perform as well for very small, fixed-size problems. See Section 6.2.

[28] Hoemmen: See Section 3.5.

[29] Hoemmen: See Section 3.2.

[30] Hoemmen: The "extents" feature in the `mdspan` proposal [16] lets users specify some dimensions at compile time, and others at run time.

16. Development of generic algorithms that operate on different matrix types (for example sparse with dense) may work, but cannot be trivially optimal. Pattern matching can be leveraged in this case and can provide acceptable solutions.[31]

17. Deciding what the matrix-matrix, matrix-vector, and vector-vector `operator*` should behave like is not an easy choice. Probably the best choice is to use the simplest form as for example adopted by Mathematica, i.e., to define it as the element-wise product.

18. `C++11, 14, 17, 20` and beyond provide tremendous opportunities for reimagining the algebra library design, making it more generic, with zero abstraction overhead and at the same time satisfying all users (by supporting scalars, vectors, matrices and tensors through a single interface).

19. Expression templates are a good mechanism for supporting various architectures. See [25], with the erratum that "MFlops" should be replaced with "GFlops."

20. Awareness of how lazy evaluation works, particularly with the `auto` keyword, is of paramount importance.[32]

21. Type traits for resolving type issues and performing pattern matching can become very complex and does not work nicely because of the "curse of overload dimensionallity." A system where stakeholders provide optimal solutions though can work in this case. Prefer Concepts when they become available.

22. Separation of algorithms from data structures is desirable but probably very hard to achieve in an optimal sense, particularly when addressing multiple architectures.

23. Higher-dimensional array (tensor) algebra enables particularly far reaching programming capabilities. The design of systems from the get-go to account for them is too much of a golden opportunity to ignore. All newer systems (for example like Eigen) are designed with a higher-dimensional container type that lends its qualities to other abstractions if needed.

24. `Auto`, Concepts, and `constexpr if` need to be factored in on how users will be using a contemporary C++ algebra system.

5 Lessons learned from Fortran libraries

Here, we talk about experiences from Fortran. We include both historical and current Fortran linear algebra libraries, as well as lessons learned from High Performance Fortran, a parallel extension of Fortran that influenced later revisions of the Fortran standard. Section 5.1 uses Jack Dongarra's oral history of the LINPACK project and examples from other libraries to show that one of the most important features of a linear algebra library, is that it is generic on the matrix element type. In Section 5.2, we illustrate with the example of the BLAS that it's helpful to separate libraries or levels of abstraction, by whether their developers are more likely to be numerical linear algebra experts or performance tuning experts. Finally, the example of High Performance Fortran in Section 5.3 teaches that an interface should make expensive operations explicit, whenever possible.

[31] Hoemmen: See Section 3.4.

[32] Hoemmen: See Sections 3.1 and 3.5.

5.1 LINPACK: Can write algorithms generic on the matrix element type

Summary: The LINPACK project shows that it's possible to write linear algebra algorithms that are generic on the matrix element type, even in languages (like Fortran 77) that lack polymorphism. Genericity is one of the most important features that a C++ linear algebra library can offer.

Dongarra gives an oral history [14] of standardization of popular Fortran linear algebra libraries, including EISPACK and LINPACK. LINPACK [13] is a library for solving dense linear systems, and EISPACK [18] is for solving dense eigenvalue problems. The two libraries together are precursors to LAPACK [3], that combines functionality from both. Dongarra explains that LINPACK wrote one version of all the algorithms (in so far as possible) for four different matrix element types. In C++ terms, the types are `float`, `double`, `complex<float>`, and `complex<double>`. LINPACK developers then used string processing scripts to generate Fortran code for each of the four data types from this "abstract" representation. LINPACK and its successor LAPACK use functionality comparable to C++'s `numeric_limits` in order to implement linear algebra algorithms that are correct and accurate with respect to floating-point arithmetic, without hard-coding floating-point traits directly into the algorithm [24].

One of the authors, Mark Hoemmen, has been a Trilinos [20] developer since 2010. One of the most heavily used parts of Trilinos are the Teuchos BLAS and LAPACK wrappers. These are very thin C++ wrappers over the BLAS and LAPACK, that are templated on the matrix element type (what Trilinos calls the `Scalar` type). The only thing "C++" about them, and their main value, is that they let users write generic C++ code that uses BLAS and LAPACK functionality. The wrappers have existed in more or less their current form for several years longer than the author's tenure as a Trilinos developer, yet few Trilinos developers complained and none actually offered a replacement. This shows that a generic C++ BLAS and LAPACK binding might satisfy many users, even if it has no more features than the BLAS and LAPACK themselves.

5.2 BLAS: Draw the line between libraries based on developer expertise

Summary: The BLAS (Basic Linear Algebra Subprograms) set a good precedent, by drawing the line between libraries based on whether developers are more likely to be numerical linear algebra experts or performance tuning experts.

The BLAS (Basic Linear Algebra Subprograms) Technical Forum defines a standard Fortran 77, Fortran 95, and C interface for linear algebra operations [17]. The Standard was published in 2002 and includes dense, banded, and sparse linear algebra operations. However, the heart of the BLAS is dense and banded linear algebra operations. The BLAS in some form has been around since the 1970s [32, 14], and its dense core has been in recognizably modern form since 1990 [11].

One benefit of the BLAS is that it draws the line between libraries based on likely developer expertise. Efficient BLAS implementation requires expertise in performance tuning and computer architecture, but not so much in linear algebra. Accurate and efficient implementation of LINPACK and LAPACK calls for understanding numerical linear algebra and floating-point arithmetic. The intent was that computer vendors would optimize the BLAS, and LAPACK would spent almost all of its time running inside the optimized BLAS [2, "The BLAS as the Key to Portability"]. That way, numerical linear algebra experts could get good performance by thinking about algorithms at a high level, e.g., by blocking to favor operations with more reuse ("BLAS 3").

This boundary may be a bit fuzzy. For example, BLAS implementations can contribute a lot to

accuracy in floating-point arithmetic. Thus justifies projects like mixed-precision BLAS [33] and Reproducible BLAS[33], and proofs that numerical linear algebra algorithms are numerically stable even if they use Strassen-like implementations of matrix-matrix multiply [10]. Furthermore, the set of core linear algebra operations may evolve over time, as computer architectures and algorithms evolve. The version of the BLAS that LINPACK used only provided vector-vector operations (what later became known as "BLAS 1") [32]; this made sense at the time, because those were the right operations to optimize on vector computers. As floating-point operations became much faster than memory operations [54] and computer architectures became cache based, algorithm developers realized that they needed to redesign matrix algorithms to reuse data better. (One fruit of that redesign is the LAPACK project [3].) This motivated them to extend the original BLAS to support matrix-vector [12] ("BLAS 2") and matrix-matrix [11] ("BLAS 3") operations, that have more reuse than the original vector-vector operations.

Even though the boundary between "what belongs to BLAS" and "what belongs to LAPACK" has shifted over time, both library developers and users appreciate that there is a boundary.

5.3 High Performance Fortran: Make expensive operations explicit

Summary: If a library (or language) makes time-consuming operations look no different than fast operations, then users might have trouble debugging performance issues.

The High Performance Fortran (HPF) programming language [29, 28] inherited Fortran's native support for multidimensional arrays. It added support for parallel arithmetic operations on these multidimensional arrays, that might be distributed in parallel over multiple processors. Compare also to the 2-D block cyclic data distributions that ScaLAPACK [9] supports for dense matrices. 2-D block cyclic distributions include block and cyclic layouts, both 1-D and 2-D, as special cases.

HPF had a number of issues that hindered its adoption. One issue relating to language design, is that the "the relationship between what the developer wrote and what the parallel machine executed [was] somewhat murky" [29, p. 13]. The language did not make clear what operations could result in expensive parallel communication. This made it hard to diagnose suboptimal performance without dropping down to the parallel equivalent of assembly language (in this case, generated Fortran + MPI communication calls).

An important lesson learned is that expensive operations should be explicit. For example, distributed-memory parallel linear algebra libraries like Trilinos [20] forbid arithmetic operations between vectors that do not have the same distribution, unless the user first explicitly invokes a data redistribution operation on one of the vectors. Authors of a C++ standard linear algebra library will need to think about this. For example, C++ operator overloading is convenient, but it risks hiding computational expense behind the mask of simple arithmetic.

One of HPF's motivations was to hide the details of distributed-memory parallel communication. However, many distributed-memory parallel libraries and applications wrote to a lower-level programming model (e.g., MPI), but built higher-level library abstractions around it. For example, discretizations of partial differential equations that use finite-element or finite-volume method have a "boundary exchange" or "ghost region exchange" primitive, that communicates whatever data a parallel process needs in order to compute the next time step or nonlinear solve step. Good library design hides almost all parallel communication behind a small number of library primitives, at least for applications that have regular communication patterns.

[33] See publications list here: https://bebop.cs.berkeley.edu/reproblas/.

6 Other standardization efforts

Here, we briefly mention related linear algebra standardization efforts, the GraphBLAS (Section 6.1) and Batched BLAS (Section 6.2). Should we consider those in scope for a standard C++ linear algebra library? We should at least know about them and make a conscious decision to include or exclude them.

6.1 GraphBLAS: Rewards of genericity

Summary: The GraphBLAS offers to make a generic sparse linear algebra library useful for implementing graph algorithms. It requires custom matrix element types and custom arithmetic, that could impose interesting requirements on a linear algebra library.

The GraphBLAS Forum is a recent, ongoing effort to standardize "building blocks for graph algorithms in the language of linear algebra."[34] The point of the GraphBLAS is that it's both possible and performant to express many graph algorithms using a small number of linear algebra primitives, such as sparse matrix-matrix multiply. One can think of the GraphBLAS as a subset of traditional sparse BLAS functionality, but with a few "wrinkles" that differ from what a BLAS implementer might expect. For example:

- Vectors are "sparse by default"; for example, one can express one step of breadth-first search by a sparse matrix-vector multiply, but this is only efficient if the vector is stored sparsely.
- Users need to be able to define custom arithmetic types, and custom arithmetic operations to replace plus, times, and zero (the arithmetic identity and multiplicative annihilator). Desired arithmetic types may be real numbers (with different operations), Boolean, or subsets of a set of integers [30]. The latter is interesting because naïve implementations might store subsets in a way that requires dynamic allocation.

Many C++ linear algebra libraries permit custom arithmetic types and operations. We argue in Section 5.1 that genericity on the matrix and vector element types is one of the most important features that a C++ linear algebra library can offer. However, how "generic" is "generic"? The Eigen library assumes that the usual arithmetic operations (including division) work, and imposes some other interface requirements.[35] Other libraries that claim to be generic have similar requirements.

One of the authors, Mark Hoemmen, has some experience with this in the Tpetra linear algebra library [7] in the Trilinos project [20]. Tpetra's vector and matrix data structures are templated on the element type (`Scalar`), but Tpetra is more restrictive about the set of types it permits than other libraries might be. While this is may be a design flaw in Tpetra, it's likely that other existing C++ linear algebra libraries have this restriction as well. Thus, understanding the reasons for the restriction could help expose pitfalls when attempting to use an existing library to implement the GraphBLAS. Here are some reasons why Tpetra does not permit arbitrary matrix entry types:

- Tpetra needs the types to work in CUDA code. This excludes types that do dynamic memory allocation inside, such as the "naïve implementation of subsets of a set of integers" type mentioned above. A small number of Trilinos developers go through a lot of trouble to make Tpetra work for a few types that have run-time sizes, where all instances of the type have the same run-time size.

[34] http://graphblas.org/index.php?title=Graph_BLAS_Forum; see also the brief position paper [36].

[35] See http://eigen.tuxfamily.org/dox/TopicCustomizing_CustomScalar.html.

- Tpetra has some coupling with downstream preconditioners and iterative solvers. These generally assume that the type of matrix and vector entries implements (mathematical) field operations like division. The "custom arithmetic" mentioned above is more like a mathematical semiring; for example, it only requires plus and times. Tpetra does permit specializations of vectors and matrices to have a different "dot product result type" than the matrix / vector entry type; this could be a path forward for refactoring.

- Tpetra needs to be able to communicate instances of the matrix entry type using MPI (the Message Passing Interface for distributed-memory parallel programming). Permitting types that have different run-time sizes would likely require more communication (more volume or more rounds or both). Tpetra's communication layer would become more complicated, since Tpetra would need to optimize for the common case of built-in types like `double`.

The GraphBLAS does have a reference C++ implementation.[36] However, any parallelization would need to think carefully about the implications of custom arithmetic. These might include load balancing (for types with different run-time sizes), thread safety and scalability of the type, and whether the type could work as a reduction result. (The OpenMP standard only recently started allowing custom types for reduction results, for example.)

6.2 Batched BLAS: A profitable special case

Summary: "Batched" basic linear algebra operations, that solve many small dense problems at once, can improve performance of some libraries and applications. Vendors offer implementations with different interface options, and a "Batched BLAS" standardization effort has begun. The C++ library standardization effort should consider whether offering batched operations could help improve performance for primary use cases.

We mentioned "batched" linear algebra interfaces briefly above. These perform many small dense operations in an optimized way, generally with the intention of parallelization and/or vectorization [15]. Many applications that use dense linear algebra actually do many small, often independent dense operations. Scientific and engineering computing have plenty of examples, but the strong and growing interest in machine learning also calls for batched linear algebra. It's possible to use existing libraries, like the BLAS and LAPACK, to solve these problems. However, these libraries were designed to solve one large problem at a time, fast and accurately. For example, they check input arguments for consistency on every call, which could take longer than the actual algorithm for a tiny matrix. A batched library could amortize these checks.

The interface of batched linear algebra operations matters a lot for performance, but may constrain generality. For example, requiring a specific data layout and constraining all matrices to have the same dimensions makes vectorization easier, but applications in sparse multifrontal matrix factorizations may produce dense matrices of different dimensions. Vendors like Intel[37] and NVIDIA[38] have different interfaces with different levels of generality. Relton et al. survey different interface options [47]. A standardization effort began circa 2016,[39] and continues with regular workshops and publications.

[36] See https://github.com/cmu-sei/gbtl.

[37] See https://software.intel.com/en-us/articles/introducing-batch-gemm-operations.

[38] See https://docs.nvidia.com/cuda/cublas/index.html.

[39] See [15] and http://icl.utk.edu/bblas/.

The `mdspan` [16] multidimensional array data structure could give users a way to store a batch of small dense matrices in an optimized layout. It also could give a linear algebra library a way to express interface polymorphism. For example, `mdspan` supports different layouts. An implementation could provide generic algorithms, but specialize for specific layouts. Some layouts can express different ways to pack linear algebra objects that can help with vectorization.

7 Lessons learned from MATLAB

Summary:

- Users appreciate concise syntax and a low barrier to entry.
- Interfaces meant for interactive use may need a different design than interfaces in a non-interactive C++ standard library.

The commercial software product MATLAB[40] provides a programming language for expressing matrices, vectors, and mathematical operations on those objects. It can do much more than that, but this is how MATLAB started and how many users think of it. The product's original author, Cleve Moler, was a coauthor of EISPACK and LINPACK (see Section 5.1). He wrote MATLAB as a teaching tool, so that students could use EISPACK and LINPACK algorithms without needing to write Fortran programs. Moler gives an account of MATLAB's history in [41].

Many students of numerical linear algebra and numerical algorithms in general "grew up" using MATLAB. They appreciate its concise syntax for both data structures and algorithms. Its interactive prompt also lowers the barrier to entry. Students' and practitioners' familarity with MATLAB means that the product has influenced the design of other linear algebra libraries, including C++ libraries.[41] This makes it attractive to use MATLAB as a model for a standard C++ linear algebra library.

Developers who wish to imitate MATLAB, though, should remember that its interface was designed first for interactive use. Its "backslash" operator for solving linear systems is a good example. Users like that they can express "solve the linear system $Ax = b$" concisely as `x = A \ b`. MATLAB then tries its best to solve the problem, but what does "solve" mean? If the matrix is square, MATLAB first tries LU factorization with partial pivoting and condition number estimation. It may find that the linear system is hard to solve accurately that way – for example, because it finds a zero pivot, or because it estimates the matrix is very close to a rank-deficient matrix in finite-precision arithmetic. At that point, MATLAB prints out warning messages and "solves" the linear system in a minimum-norm least-squares sense. That is, it finds the x with minimum 2-norm that minimizes the 2-norm of $b - Ax$. Depending on the right-hand side b, the resulting x may not even be close to satisfying $Ax = b$. Users can read MATLAB's warning messages and use its interactive help system to know whether this is happening, learn what it means, and decide whether the resulting x is useful to them. For many applications, this is exactly the x they want. MATLAB's fall-back to minimum-norm least squares even lets its "solve" $Ax = b$ when A is not square. All these checks and fall-backs make MATLAB slower than the Fortran routines it wraps, but users accept this in exchange for interactivity and other features.

Note that the Fortran routines (e.g., in what currently would be the BLAS [17] and LAPACK [2]) that MATLAB wraps do *not* behave in this way. The routine that solves $Ax = b$ using LU with partial

[40] See https://www.mathworks.com/products/matlab.html.

[41] See e.g., http://arma.sourceforge.net/.

pivoting does nothing more than that. It expects the matrix A to be square. If it detects a zero pivot, it stops and returns an error code, without falling back to a different algorithm or problem. If you want to solve the minimum-norm least-squares problem, you must call a different routine.

The Fortran routines behave in this way because they were not intended to work interactively. Typical BLAS and LAPACK library customers must design their code to give sensible input to the library, and to detect and handle any error conditions on return, without user intervention. Fortran has no interactive prompt like MATLAB, that can offer hints and links to documentation.

C++ is much more like Fortran than MATLAB. Almost no components of the C++ Standard Library work interactively. If a standard C++ linear algebra interface were to imitate MATLAB's backslash, how would it report warnings back to the user? Printing warnings would make no sense if the library were being called in an interactive spreadsheet application. The spreadsheet would need to detect conditions like near rank deficiency *in code*, not by parsing some human-readable terminal output. This condition might mean that the user constructed the problem incorrectly. Thus, the application might not want to "solve" the problem in a least-squares sense, and may want to construct its own error message in terms that spreadsheet users would understand.

MATLAB is a product which the authors themselves have found useful. It has a syntax designed by experts in matrix computations, that makes it easy to get work done quickly. However, MATLAB is designed for interactive use, so not all of its interfaces may fit the C++ model.

8 Lessons learned from Python libraries

Summary: The Python programming language does not come with native linear algebra or multidimensional array support, but Numpy offers a library solution. Users find Numpy's Fortran / MATLAB[42] - style slice notation and its integration with many provided numerical algorithms attractive. Many potential users of a C++ linear algebra library would be familiar with Python and Numpy.

Scientists in the 2000s saw Python as an alternative to MATLAB. Python is a general-purpose programming language that comes with many standard libraries. This can make it easier for writing complete applications that, for example, interact with databases, access the internet, or generate web pages. Python also comes built into many operating systems, without installation or licensing concerns. While the language was not designed for numerical computation, its availability and ease of learning makes it attractive in that domain.

Python does not support multidimensional arrays in the Python language itself. Python 3's C API does have a Buffer protocol[43] that could be used to access multidimensional arrays. Numpy (see Section 8.1) builds on top of the Buffer protocol.

8.1 Numpy

Python's lack of in-language support for multidimensional arrays and associated mathematical operations led to development of library solutions. Travis Oliphant merged two main projects, `Numarray` and `Numeric`, to create `Numpy` [44]. The communities behind the two other projects soon merged.

From the beginning of `Numpy`, the sister project `Scipy` [26] held most of the non-core features. If Numpy contained the array itself as well as the basic mathematical operations, it still contained a few

[42] See Section 7.

[43] See https://docs.python.org/3/c-api/buffer.html.

outside features (like support for fast Fourier transforms) that were considered as being part of Scipy.

Numpy provides several key features:

- a true multidimensional container that was partially integrated inside Python 3,
- a "slice" (subarray) notation with `:`, and
- broadcasting capabilities,[44] meaning that all dimension sizes don't have to match for an operation to be performed.

Numpy executes all component-wise operations in C with so-called "universal functions." This has helped Numpy achieve most of the efficiency of compiled C code. Projects like Numba[45] go further, by using just-in-time compilation to translate a subset of Python and Numpy directly to machine code.

8.2 Array vs. matrix

One of the most known issues of Numpy is that it has two different fundamental data containers:

- `array`, a true multidimensional array; and
- `matrix`, purely a two-dimensional object.

The two containers interpreted the multiplication operator in two mathematically different ways: `array` used the Hadamard product, and `matrix` used matrix-matrix multiplication.

The consensus now is to avoid confusion by using only `array`. Python 3 introduced a new operator for matrix-matrix multiplication with `array`. This lets users perform all typical array operations with operators.

8.3 The current language of choice for data science and machine learning

Python is currently the `de facto` language for data science and machine learning. The features that are provided by the core language (batteries included but also the readable notation) have made it unavoidable in the numerical landscape.

The other languages have not, as of today, been able to have a community as big as Python's, which will make it a reference in the domain for the years to come.

8.4 xtensor, a C++ equivalent to Numpy

Quantstack has developed a C++ equivalent to Numpy named `xtensor` [46]. It provides 3 different containers that add the static aspects of an array (fixed number of dimensions and fixed size in a dimension), instead of just one.

The basic operations are always componentwise (using the Hadamard product for the multiplication operator), leaving the matrix product in another sister project.

Despite being close to Numpy, the notation is far more cumbersome than the Python equivalent (missing matrix operator and slice operations).

[44] See https://docs.scipy.org/doc/numpy-1.13.0/user/basics.broadcasting.html#module-numpy.doc.broadcasting and https://scipy.github.io/old-wiki/pages/EricsBroadcastingDoc.

[45] See http://numba.pydata.org/.

9 Conclusions

The authors offer here some subjective views on the goals of a C++ linear algebra standardization effort.

A standard C++ linear algebra library will likely have many users who are not practitioners of traditional "scientific and engineering computing." They might be interested in computer graphics, gaming, computer vision, machine learning, or graph algorithms. Some users care about matrices and vectors, others about tensors, and still others about quaternions and octernions. Nevertheless, we think all these fields have enough in common that a unified effort can pay off.

A linear algebra library will want to expose iteration over multidimensional sequences or index spaces, much like the C++ Standard Template Library or Ranges do with single-dimensional sequences. Einstein index notation for tensors is a general way of expressing this, but is not the only way. Users want to access the entries of vectors, matrices, and tensors, so we will need to come up with good abstractions for this.

Users want code to look as much like mathematical notation as possible. This can have a performance cost, but libraries can reduce or eliminate that cost using techniques like expression templates. Nevertheless, developers must take care to avoid pitfalls like incorrect use of `auto` with temporary expressions, and long build times. Users also need to take responsibility for compilation time and memory issues by breaking up long compilation units.

One of the most important lessons learned from the BLAS, LINPACK, and LAPACK, is that it pays to draw the boundaries between libraries based on the likely expertise of the developers who would work on each library. For example, implementing "inverse of a matrix" accurately requires much more numerical linear algebra knowledge than implementing "dot product of two vectors." Writing a fast dense matrix-matrix multiply takes much more knowledge about computer architecture than writing a reasonably efficient dense LU factorization. C++ does put domain-specific knowledge into its Standard Library; the calendar and time zone library [21] that was voted into C++20 is a good example. Nevertheless, C++ Standard Library developers, despite their fantastic talents, generally are not experts in numerical linear algebra. This suggests that a C++ linear algebra library should carefully separate the operations whose implementation calls for numerical linear algebra expertise, from the more "computer science-y" operations.

Examining the history of the BLAS Standard, and related standardization efforts like the GraphBLAS and Batched BLAS, can help us more consciously define the scope of a standard C++ linear algebra library. Goals like "generic on matrix element type" or "zero-overhead abstraction" have concrete implications on interface and implementation complexity, for example.

We find it encouraging that C++ is seriously thinking about standardizing a linear algebra library, and hope that this essay contributes to that effort.

10 Acknowledgements

Our thanks to the community of developers who came together for this discussion. We had our first phone meeting on October 10, 2018,[46] with subsequent teleconferences as well as a face-to-face session at the San Diego WG21 meeting. Michael Wong gave of his time to help organize and lead meetings, and to offer the SG14 e-mail list as a discussion forum. Thanks to Guy Davidson and Bob Steagall for boldly

[46] See minutes with participants list here: https://github.com/cpp-linear-algebra/cpp-linear-algebra-brainstorming/blob/master/minutes/2018_10_10__telecon.txt.

offering their proposal, "Towards Standardization of Linear Algebra," as a first draft. It's heartening to note that these two authors approached the topic from different angles: Bob, from his experiences writing software to solve linear least-squares problems arising in functional magnetic resonance imaging, and Guy, from his work on the 2-D graphics proposal [37]. Nasos Iliopoulos and John Michopoulos contributed their experiences as uBlas developers[47] and with C++ techniques for implementing Einstein (tensor index) notation. Cem Bassoy also contributed to the discussion of the value of accepting tensors as an initial part of this library. Many others offered useful insights and suggestions that enriched this paper and will certainly enrich the library. This paper will undergo further revision, so if we omitted your name, please let us know.

The following authors wish to list their affiliations:

- Mark Hoemmen, Sandia National Laboratories; and
- Jayesh Badwaik, Department of Mathematics, University of Würzburg.

Other authors represent only themselves, not their institutions.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE–NA0003525.

References

[1] Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. "A framework for sparse matrix code synthesis from high-level specifications". In: *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society. 2000, p. 58.

[2] E. Anderson et al. *LAPACK Users' Guide*. 3rd. SIAM, 1999.

[3] E. Anderson et al. *LAPACK: A portable linear algebra library for high-performance computers*. Tech. rep. CS-90-105. Computer Science Dept., University of Tennessee, Knoxville, TN, May 1990.

[4] Thomas E. Anderson, David E. Culler, David A. Patterson, et al. *A Case for NOW (Networks of Workstations)*. Feb. 1995.

[5] Erlend Arge, Are Magnus Bruaset, and Hans Petter Langtangen. "Object-Oriented Numerics". In: *Numerical Methods and Software Tools in Industrial Mathematics*. Nov. 1996.

[6] Erlend Arge et al. "On the numerical efficiency of C++ in scientific computing". In: *Numerical Methods and Software Tools in Industrial Mathematics*. Springer, 1997, pp. 91–118.

[7] C. G. Baker and M. A. Heroux. "Tpetra, and the Use of Generic Programming in Scientific Computing". In: *Sci. Program.* 20.2 (Apr. 2012), pp. 115–128. ISSN: 1058-9244. DOI: 10.1155/2012/693861. URL: http://dx.doi.org/10.1155/2012/693861.

[8] John J. Barton and Lee R. Nackman. *Scientific and Engineering C++: an introduction with advanced techniques and examples*. Addison-Wesley Longman Publishing Co., Inc., 1994.

[9] L Susan Blackford et al. *ScaLAPACK Users' Guide*. Philadelphia, PA, USA: SIAM, 1997.

[47] See e.g., [25].

[10] James Demmel, Ioana Dumitriu, and Olga Holtz. "Fast linear algebra is stable". In: *Numerische Mathematik* 108.1 (Oct. 2007), pp. 59–91.

[11] J. J. Dongarra et al. "A set of Level 3 Basic Linear Algebra Subprograms". In: *ACM Trans. Math. Softw.* (1990).

[12] J. J. Dongarra et al. "An extended set of FORTRAN Basic Linear Algebra Subprograms". In: *ACM Trans. Math. Softw.* (1988).

[13] J. J. Dongarra et al. *LINPACK User's Guide*. Philadelphia, PA, USA: Society of Industrial and Applied Mathematics, 1979.

[14] Jack Dongarra. *Oral history interview by Thomas Haigh, 26 April, 2005, University of Tennessee, Knoxville, TN, USA*. 2005. URL: http://history.siam.org/oralhistories/dongarra.htm.

[15] Jack Dongarra et al. *A Proposed API for Batched Basic Linear Algebra Subprograms*. Tech. rep. School of Mathematics, The University of Manchester, 2016. URL: http://eprints.maths.manchester.ac.uk/2464/1/batched_api.pdf.

[16] H. Carter Edwards et al. *mdspan: A Non-Owning Multidimensional Array Reference*. Tech. rep. P0009r8. Oct. 2018. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0009r8.html.

[17] Basic Linear Algebra Subprograms Technical Forum. "Basic Linear Alegbra Subprograms Technical Forum Standard". In: *International Journal of High Performance Applications and Supercomputing* 16.1 (2002). URL: http://www.netlib.org/blas/blast-forum/blas-report.pdf.

[18] Burton S. Garbow. "EISPACK – A package of matrix eigensystem routines". In: *Computer Physics Communications* 7.4 (Apr. 1974), pp. 179–184.

[19] Khronos Group. *History of OpenGL*. Sept. 2017. URL: https://www.khronos.org/opengl/wiki/History_of_OpenGL.

[20] Michael A. Heroux et al. "An Overview of the Trilinos Project". In: *ACM Trans. Math. Softw.* 31.3 (Sept. 2005), pp. 397–423. URL: http://doi.acm.org/10.1145/1089014.1089021.

[21] Howard E. Hinnant and Tomasz Kamiński. *Extending <chrono> to Calendars and Time Zones*. Tech. rep. P0355R7. Mar. 2018. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0355r7.html.

[22] Mark Hoemmen. "A communication-avoiding, hybrid-parallel, rank-revealing orthogonalization method". In: *International Symposium on Parallel and Distributed Processing 2011 (IPDPS 2011)*. IEEE, May 2011.

[23] Mark Hoemmen. *Oral history, collected by Trilinos developer Mark Hoemmen from Chris Luchini (POOMA developer) and various Trilinos developers*. 2018.

[24] Mark Hoemmen and Damien Lebrun-Grandie. *Generic numerical algorithm development with(out) numeric_limits*. Tech. rep. P1370r0. Nov. 2018. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1370r0.txt.

[25] Athanasios Iliopoulos and John G. Michopoulos. "uBlasCL: Architecture Agnostic Massively Parallel Linear Algebra System". In: *Proceedings of the ASME 2011 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference IDETC/CIE, August 29-31, 2011, Washington, DC, USA*. Aug. 2011. DOI: 10.1115/DETC2011-48228.

[26] Eric Jones, Travis Oliphant, Pearu Peterson, et al. *SciPy: Open source scientific tools for Python*. 2001–. URL: http://www.scipy.org/.

[27] Brewster Kahle and W. Daniel Hillis. *The Connection Machine Model CM-1 Architecture*. Tech. rep. Thinking Machines Corporation, Cambridge, MA, USA, 1989.

[28] Ken Kennedy, Charles Koelbel, and Hans Zima. "The rise and fall of High Performance Fortran". In: *Communications of the ACM* 54.11 (2011), pp. 74–82.

[29] Ken Kennedy, Charles Koelbel, and Hans Zima. "The rise and fall of High Performance Fortran: an historical object lesson". In: *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. ACM. 2007, pp. 7-1–7-21.

[30] Jeremy Kepner et al. "Mathematical Foundations of the GraphBLAS". In: *arXiv e-prints*, arXiv:1606.057 (June 2016), arXiv:1606.05790. arXiv: 1606.05790 [cs.MS].

[31] Manojkumar Krishnan et al. *The Global Arrays User Manual*. Feb. 2012. URL: http://hpc.pnl.gov/globalarrays/papers/GA-UserManual-Main.pdf.

[32] C. L. Lawson et al. "Basic Linear Algebra Subprograms for FORTRAN usage". In: *ACM Trans. Math. Softw.* 5.3 (Sept. 1979), pp. 308–323.

[33] Xiaoye S. Li et al. *Design, Implementation and Testing of Extended and Mixed Precision BLAS*. Tech. rep. 149. LAPACK Working Note, Oct. 2000. URL: http://www.netlib.org/lapack/lawnspdf/lawn149.pdf.

[34] John Markoff. "The Attack of the 'Killer Micros'". In: *The New York Times* (May 1991).

[35] Nikolay Mateev, Keshav Pingali, and Paul Stodghill. *The Bernoulli Generic Matrix Library*. Tech. rep. Cornell University, 2000.

[36] T. Mattson et al. "Standards for graph algorithm primitives". In: *2013 IEEE High Performance Extreme Computing Conference (HPEC)*. Sept. 2013, pp. 1–2. DOI: 10.1109/HPEC.2013.6670338.

[37] Michael B. McLaughlin et al. *A Proposal to Add 2D Graphics Rendering and Display to C++*. Tech. rep. P0267r8. June 2018. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0267r8.pdf.

[38] Paul Messina. "The Accelerated Strategic Computing Initiative". In: *Impact of Advances in Computing and Communications Technologies on Chemical Science and Technology: Report of a Workshop*. Washington, D.C., USA: National Academy Press, 1999.

[39] Scott Meyers. "How Non-Member Functions Improve Encapsulation". In: *Dr. Dobb's* (Feb. 2000). URL: http://www.drdobbs.com/cpp/how-non-member-functions-improve-encapsu/184401197.

[40] Millind Mittal, Alex Peleg, and Uri Weiser. *MMX Technology Architecture Overview*. 1997. URL: https://www.intel.com/content/dam/www/public/us/en/documents/research/1997-vol01-iss-3-intel-technology-journal.pdf.

[41] Cleve Moler. *The Origins of MATLAB*. 2004. URL: https://www.mathworks.com/company/newsletters/articles/the-origins-of-matlab.html.

[42] Eric Niebler, Casey Carter, and Christopher Di Bella. *The One Ranges Proposal*. Tech. rep. P0896r4. Nov. 2018. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0896r4.pdf.

[43] S. Oberman, G. Favor, and F. Weber. *AMD 3DNow! technology: architecture and implementations*. Mar. 1999.

[44] Travis Oliphant. *A guide to NumPy*. Trelgol Publishing, 2006.

[45] Jack Poulson et al. *Elemental: A New Framework for Distributed Memory Dense Matrix Computations*. Feb. 2013.

[46] QuantStack. *xtensor: C++ tensors with broadcasting and lazy computing*. 2018–. URL: https://github.com/QuantStack/xtensor.

[47] Samuel D. Relton, Pedro Valero-Lara, and Mawussi Zounon. *A Comparison of Potential Interfaces for Batched BLAS Computations*. Tech. rep. 2016.42. Manchester Institute for Mathematical Sciences, School of Mathematics, The University of Manchester, 2016. URL: http://eprints.maths.manchester.ac.uk/2493/.

[48] Mathieu Ropert. *Copy and Swap, 20 years later*. Jan. 2019. URL: https://mropert.github.io/2019/01/07/copy_swap_20_years/.

[49] Jeremy G. Siek, Ian Karlin, and E. R. Jessup. "Build to order linear algebra kernels". In: *International Symposium on Parallel and Distributed Processing 2008 (IPDPS 2008)*. 2008, pp. 1–8.

[50] Todd Veldhuizen. *Blitz++ User's Guide: A C++ class library for scientific computing for version 0.9*. Oct. 2005. URL: http://physik.uni-graz.at/~crg/Programmierkurs1112/pdfs/blitz.pdf.

[51] Todd Veldhuizen. "Scientific Computing: C++ Versus Fortran". In: *Dr. Dobb's* (Nov. 1997). URL: http://www.drdobbs.com/cpp/scientific-computing-c-versus-fortran/184410315.

[52] Todd Veldhuizen. *Techniques for Scientific C++*. Tech. rep. TR542. School of Informatics, Computing, and Engineering, Indiana University Bloomington, 2000. URL: https://www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR542.

[53] Al Vermeulen and Margaret Chapman. "OON-SKI: an introduction". In: *Scientific Programming* 2.4 (1993), pp. 109–110.

[54] William A. Wulf and Sally A. McKee. "Hitting the memory wall: implications of the obvious". In: *ACM SIGARCH Computer Architecture News* 23.1 (1995).