SAND2019-6877C

P1673R0: A free function linear algebra interface based on the BLAS

Authors
* Mark Hoemmen (mhoemme@sandia.gov) (Sandia National Laboratories)
* David Hollman (dshollm@sandia.gov) (Sandia National Laboratories)
* Christian Trott (crtrott@sandia.gov) (Sandia National Laboratories)
* Daniel Sunderland (dsunder@sandia.gov) (Sandia National Laboratories)
* Nevin Liber (nliber@anl.gov) (Argonne National Laboratory)
* Siva Rajamanickam (srajama@sandia.gov) (Sandia National Laboratories)
* Li-Ta Lo (ollie@lanl.gov) (Los Alamos National Laboratory)
* Graham Lopez (lopezmg@ornl.gov) (Oak Ridge National Laboratories)
* Peter Caday (peter.caday@intel.com) (Intel)
* Sarah Knepper (sarah.knepper@intel.com) (Intel)
* Piotr Luszczek (luszczek@icl.utk.edu) (University of Tennessee)
* Timothy Costa (tcosta@nvidia.com) (NVIDIA)
Contributors
* Chip Freitag (chip.freitag@amd.com) (AMD)
* Bryce Lelbach (blelbach@nvidia.com) (NVIDIA)
* Srinath Vadlamani (Srinath.Vadlamani@arm.com) (ARM)
* Rene Vanoostrum (Rene.Vanoostrum@amd.com) (AMD)

Date: 2019-06-17

Purpose of this paper

This paper proposes a C++ Standard Library dense linear algebra interface based on the dense Basic Linear Algebra Subroutines (BLAS). This corresponds to a subset of the [BLAS Standard](http://www.netlib.org/blas/blast-forum/blas-report.pdf). Our proposal implements the following classes of algorithms on matrices and vectors:

* Elementwise vector sums
* Multiplying all elements of a vector or matrix by a scalar
* 2-norms, 1-norms, and infinity-norms of vectors
* Vector-vector, matrix-vector, and matrix-matrix products (contractions)
* Low-rank updates of a matrix
* Triangular solves with one or more "right-hand side" vectors
* Generating and applying plane (Givens) rotations

Our algorithms work with most the matrix storage formats that the BLAS Standard supports:

* "General" dense matrices, in column-major or row-major format
* Symmetric or Hermitian (for complex numbers only) dense matrices, stored either as general dense matrices, or in a packed format
* Dense triangular matrices, stored either as general dense matrices or in a packed format

Our proposal also has the following distinctive characteristics:

* It uses free functions, not arithmetic operator overloading.
* The interface is designed in the spirit of the C++ Standard Library's algorithms.
* It uses the multidimensional array data structures [`basic_mdspan` (P0009R9)](http://wg21.link/p0009) and [`basic_mdarray` (P1684R0)](https://isocpp.org/files/papers/P1684R0.pdf) to represent matrices and vectors. In the future, it could support other proposals' matrix and vector data structures.
* The interface permits optimizations for matrices and vectors with small compile-time dimensions; the standard BLAS interface does not.
* Each of our proposed operations supports all element types for which that operation makes sense, unlike the BLAS, which only supports four element types.
* Our operations permit "mixed-precision" computation with matrices and vectors that have different element types. This subsumes most functionality of the Mixed-Precision BLAS specification (Chapter 4 of the [BLAS Standard](http://www.netlib.org/blas/blast-forum/blas-report.pdf)).
* Like the C++ Standard Library's algorithms, our operations take an optional execution policy argument. This is a hook to support parallel execution and hierarchical parallelism (through the proposed executor extensions to execution policies, see [P1019R2](http://wg21.link/p1019r2)).
* Unlike the BLAS, our proposal can be expanded to support "batched" operations (see [P1417R0](http://wg21.link/p1417r0)) with almost no interface differences. This will support machine learning and other applications that need to do many small matrix or vector operations at once.

Interoperable with other linear algebra proposals

We believe this proposal is complementary to [P1385R1](http://wg21.link/p1385r1), a proposal for a C++ Standard linear algebra library that introduces matrix and vector classes and overloaded arithmetic operators. In fact, we think that our proposal would make a natural foundation for a library like what P1385R1 proposes. However, a free function interface -- which clearly separates algorithms from data structures -- more naturally allows for a richer set of operations such as what the BLAS provides. A natural extension of the present proposal would include accepting P1385's

matrix and vector objects as input for the algorithms proposed here.

Why include dense linear algebra in the C++ Standard Library?

1. C++ applications in "important application areas" (see [P0939R0](http://wg21.link/p0939r0)) have depended on linear algebra for a long time.
2. Linear algebra is like `sort`: obvious algorithms are slow, and the fastest implementations call for hardware-specific tuning.
3. Dense linear algebra is core functionality for most of linear algebra, and can also serve as a building block for tensor operations.
4. The C++ Standard Library includes plenty of "mathematical functions." Linear algebra operations like matrix-matrix multiply are at least as broadly useful.
5. The set of linear algebra operations in this proposal are derived from a well-established, standard set of algorithms that has changed very little in decades. It is one of the strongest possible examples of standardizing existing practice that anyone could bring to C++.
6. This proposal follows in the footsteps of many recent successful incorporations of existing standards into C++, including the UTC and TAI standard definitions from the International Telecommunications Union, the time zone database standard from the International Assigned Numbers Authority, and the ongoing effort to integrate the ISO unicode standard.

Linear algebra has had wide use in C++ applications for nearly three decades (see [P1417R0](http://wg21.link/p1417r0) for a historical survey). For much of that time, many third-party C++ libraries for linear algebra have been available. Many different subject areas depend on linear algebra, including machine learning, data mining, web search, statistics, computer graphics, medical imaging, geolocation and mapping, engineering, and physics-based simulations.

["Directions for ISO C++" (P0939R0)](http://wg21.link/p0939r0) offers the following in support of adding linear algebra to the C++ Standard Library:

* P0939R0 calls out "Support for demanding applications in important application areas, such as medical, finance, automotive, and games (e.g., key libraries...)" as an area of general concern that "we should not ignore." All of these areas depend on linear algebra.
* "Is my proposal essential for some important application domain?" Many large and small private companies, science and engineering laboratories, and academics in many different fields all depend on linear algebra.
* "We need better support for modern hardware": Modern hardware spends many of its cycles in linear algebra. For decades, hardware vendors, some represented at WG21 meetings, have provided and continue to provide features specifically to accelerate linear algebra operations. For example, SIMD (single instruction multiple data) is a feature added to processors to speed up matrix and vector operations. [P0214R9](http://wg21.link/p0214r9), a C++ SIMD library, was voted into the C++20 draft. Several large computer system vendors offer optimized linear algebra libraries based on or closely resembling the BLAS; these include AMD's BLIS, ARM's Performance Libraries, Cray's LibSci, Intel's Math Kernel Library (MKL), IBM's Engineering and Scientific Subroutine Library (ESSL), and NVIDIA's cuBLAS.

Obvious algorithms for some linear algebra operations like dense matrix-matrix multiply are asymptotically slower than less-obvious algorithms. (Please refer to a survey one of us coauthored, ["Communication lower bounds and optimal algorithms for numerical linear algebra."](https://doi.org/10.1017/S0962492914000038)) Furthermore, writing the fastest dense matrix-matrix multiply depends on details of a specific computer architecture. This makes such operations comparable to `sort` in the C++ Standard Library: worth standardizing, so that Standard Library implementers can get them right and hardware vendors can optimize them. In fact, almost all C++ linear algebra libraries end up calling non-C++ implementations of these algorithms, especially the implementations in optimized BLAS libraries (see below). In this respect, linear algebra is also analogous to standard library features like `random_device`: often implemented directly in assembly or even with special hardware, and thus an essential component of allowing no room for another language "below" C++ (see notes on this philosophy in [P0939R0](http://wg21.link/p0939r0) and Stroustrup's seminal work "The Design and Evolution of C++").

Dense linear algebra is the core component of most algorithms and applications that use linear algebra, and the component that is most widely shared over different application areas. For example, tensor computations end up spending most of their time in optimized dense linear algebra functions. Sparse matrix computations get best performance when they spend as much time as possible in dense linear algebra.

The C++ Standard Library includes many "mathematical special functions" (**[sf.cmath]**), like incomplete elliptic integrals, Bessel functions, and other polynomials and functions named after various mathematicians. Any of them comes with its own theory and set of applications for which robust and accurate implementations are indispensible. We think that linear algebra operations are at least as broadly useful, and in many cases significantly more so.

Why base a C++ linear algebra library on the BLAS?

1. The BLAS is a standard that codifies decades of existing practice.
2. The BLAS separates out "performance primitives" for hardware experts to tune, from mathematical operations that rely on those primitives for good performance.
3. Benchmarks reward hardware and system vendors for providing an optimized BLAS implementations.
4. Writing a fast BLAS implementation for common element types is nontrivial, but well understood.
5. Optimized third-party BLAS implementations with liberal software licenses exist.
6. Building a C++ interface on top of the BLAS is a straightforward exercise, but has pitfalls for unaware developers.

Linear algebra has had a cross-language standard, the Basic Linear Algebra Subroutines (BLAS), since 2002. The Standard came out of a [standardization process](http://www.netlib.org/blas/blast-forum/) that started in 1995 and held meetings three times a year until 1999. Participants in the process came from industry, academia, and government research laboratories. The dense linear algebra subset of the BLAS codifies forty years of evolving practice, and has existed in recognizable form since 1990 (see [P1417R0](http://wg21.link/p1417r0)). The BLAS interface was specifically designed as the distillation of the "computer science" / performance-oriented parts of linear algebra algorithms. It cleanly separates operations most critical for performance, from operations whose implementation takes expertise in mathematics and rounding-error analysis. This gives vendors opportunities to add value, without asking for expertise outside the typical required skill set of a Standard Library implementer.

Well-established benchmarks such as the [LINPACK benchmark](https://www.top500.org/project/linpack/) reward computer hardware vendors for optimizing their BLAS implementations. Thus, many vendors provide an optimized BLAS library for their computer architectures. Writing fast BLAS-like operations is not trivial, and depends on computer architecture. However, it is not black magic; it is a well-understood problem whose solutions could be parameterized for a variety of computer architectures. See, for example, [Goto and van de Geijn 2008](https://doi.org/10.1145/1356052.1356053). There are optimized third-party BLAS implementations for common architectures, like [ATLAS](http://math-atlas.sourceforge.net/) and [GotoBLAS](https://www.tacc.utexas.edu/research-development/tacc-software/gotoblas2).

A (slow but correct) [reference implementation of the BLAS](http://www.netlib.org/blas/#_reference_blas_version_3_8_0) exists and it has a liberal software license for easy reuse.

We have experience in the exercise of wrapping a C or Fortran BLAS implementation for use in portable C++ libraries. We describe this exercise in detail in our paper "Evolving a Standard C++ Linear Algebra Library from the BLAS" (P1674R0). It is straightforward for vendors, but has pitfalls for developers. For example, Fortran's application binary interface (ABI) differs across platforms in ways that can cause run-time errors (even incorrect results, not just crashing). Historical examples of vendors' C BLAS implementations have also had ABI issues that required work-arounds. This dependence on ABI details makes availability in a standard C++ library valuable.

Notation and conventions

The BLAS uses Fortran terms

The BLAS' "native" language is Fortran. It has a C binding as well, but the BLAS Standard and documentation use Fortran terms. Where applicable, we will call out relevant Fortran terms and highlight possibly confusing differences with corresponding C++ ideas. Our paper P1674R0 ("Evolving a Standard C++ Linear Algebra Library from the BLAS") goes into more detail on these issues.

We call "subroutines" functions

Like Fortran, the BLAS distinguishes between functions that return a value, and subroutines that do not return a value. In what follows, we will refer to both as "BLAS functions" or "functions."

Element types and BLAS function name prefix

The BLAS implements functionality for four different matrix, vector, or scalar element types:

* `REAL` (`float` in C++ terms)
* `DOUBLE PRECISION` (`double` in C++ terms)
* `COMPLEX` (`complex<float>` in C++ terms)
* `DOUBLE COMPLEX` (`complex<double>` in C++ terms)

The BLAS' Fortran 77 binding uses a function name prefix to

distinguish functions based on element type:
* `S` for `REAL` ("single")
* `D` for `DOUBLE PRECISION`
* `C` for `COMPLEX`
* `Z` for `DOUBLE COMPLEX`
For example, the four BLAS functions `SAXPY`, `DAXPY`, `CAXPY`, and `ZAXPY` all perform the vector update `Y = Y + ALPHA*X` for vectors `X` and `Y` and scalar `ALPHA`, but for different vector and scalar element types.
The convention is to refer to all of these functions together as `xAXPY`. In general, a lower-case `x` is a placeholder for all data type prefixes that the BLAS provides. For most functions, the `x` is a prefix, but for a few functions like `IxAMAX`, the data type "prefix" is not the first letter of the function name. (`IxAMAX` is a Fortran function that returns `INTEGER`, and therefore follows the old Fortran implicit naming rule that integers start with `I`, `J`, etc.)
Not all BLAS functions exist for all four data types. These come in three categories:
1. The BLAS provides only real-arithmetic (`S` and `D`) versions of the function, since the function only makes mathematical sense in real arithmetic.
2. The complex-arithmetic versions perform a slightly different mathematical operation than the real-arithmetic versions, so they have a different base name.
3. The complex-arithmetic versions offer a choice between non-conjugated or conjugated operations.
As an example of the second category, the BLAS functions `SASUM` and `DASUM` compute the sums of absolute values of a vector's elements. Their complex counterparts `CSASUM` and `DZASUM` compute the sums of absolute values of real and imaginary components of a vector `v`, that is, the sum of `abs(real(v(i))) + abs(imag(v(i)))` for all `i` in the domain of `v`. The latter operation is still useful as a vector norm, but it requires fewer arithmetic operations.
Examples of the third category include the following:
* non-conjugated dot product `xDOTU` and conjugated dot product `xDOTC`; and
* rank-1 symmetric (`xGERU`) vs. Hermitian (`xGERC`) matrix update.
The conjugate transpose and the (non-conjugated) transpose are the same operation in real arithmetic (if one considers real arithmetic embedded in complex arithmetic), but differ in complex arithmetic. Different applications have different reasons to want either. The C++ Standard includes complex numbers, so a Standard linear algebra library needs to respect the mathematical structures that go along with complex numbers.
What we exclude from the design
Functions not in the Reference BLAS
The BLAS Standard includes functionality that appears neither in the [Reference BLAS](http://www.netlib.org/lapack/explore-html/d1/df9/group__blas.html) library, nor in the classic BLAS "level" 1, 2, and 3 papers. (For history of the BLAS "levels" and a bibliography, see [P1417R0](http://wg21.link/p1417r0). For a paper describing functions not in the Reference BLAS, see "An updated set of basic linear algebra subprograms (BLAS)," listed in "Other references" below.) For example, the BLAS Standard has
* several new dense functions, like a fused vector update and dot product;
* sparse linear algebra functions, like sparse matrix-vector multiply and an interface for constructing sparse matrices; and
* extended- and mixed-precision dense functions (though we subsume some of their functionality; see below).
Our proposal only includes core Reference BLAS functionality, for the following reasons:
1. Vendors who implement a new component of the C++ Standard Library will want to see and test against an existing reference implementation.
2. Many applications that use sparse linear algebra also use dense, but not vice versa.
3. The Sparse BLAS interface is a stateful interface that is not consistent with the dense BLAS, and would need more extensive redesign to translate into a modern C++ idiom. See discussion in [P1417R0](http://wg21.link/p1417r0).
4. Our proposal subsumes some dense mixed-precision functionality (see below).
LAPACK or related functionality
The [LAPACK](http://www.netlib.org/lapack/) Fortran library implements solvers for the following classes of mathematical problems:
* linear systems,
* linear least-squares problems, and
* eigenvalue and singular value problems.
It also provides matrix factorizations and related linear algebra

operations. LAPACK deliberately relies on the BLAS for good
performance; in fact, LAPACK and the BLAS were designed together. See
history presented in [P1417R0](http://wg21.link/p1417r0).
Several C++ libraries provide slices of LAPACK functionality. Here is
a brief, noninclusive list, in alphabetical order, of some libraries
actively being maintained:
* [Armadillo](http://arma.sourceforge.net/),
* [Boost.uBLAS](https://github.com/boostorg/ublas),
* [Eigen](http://eigen.tuxfamily.org/index.php?title=Main_Page),
* [Matrix Template Library](http://www.simunova.com/de/mtl4/), and
* [Trilinos](https://github.com/trilinos/Trilinos/).
[P1417R0](http://wg21.link/p1417r0) gives some history of C++ linear
algebra libraries. The authors of this proposal have
[designed](https://www.icl.utk.edu/files/publications/2017/icl-utk-1031-2017.p
df),
[written](https://github.com/kokkos/kokkos-kernels), and
[maintained](https://github.com/trilinos/Trilinos/tree/master/packages/teuchos
/numerics/src)
LAPACK wrappers in C++. Some authors have LAPACK founders as PhD
advisors. Nevertheless, we have excluded LAPACK-like functionality
from this proposal, for the following reasons:
1. LAPACK is a Fortran library, unlike the BLAS, which is a
   multilanguage standard.
2. We intend to support more general element types, beyond the four
   that LAPACK supports. It's much more straightforward to make a C++
   BLAS work for general element types, than to make LAPACK algorithms
   work generically.
First, unlike the BLAS, LAPACK is a Fortran library, not a standard.
LAPACK was developed concurrently with the "level 3" BLAS functions,
and the two projects share contributors. Nevertheless, only the BLAS
and not LAPACK got standardized. Some vendors supply LAPACK
implementations with some optimized functions, but most
implementations likely depend heavily on "reference" LAPACK. There
have been a few efforts by LAPACK contributors to develop C++ LAPACK
bindings, from [Lapack++](https://math.nist.gov/lapack++/) in
pre-templates C++ circa 1993, to the recent ["C++ API for BLAS and
LAPACK"](https://www.icl.utk.edu/files/publications/2017/icl-utk-1031-2017.pdf
).
(The latter shares coauthors with this proposal.) However, these are
still just C++ bindings to a Fortran library. This means that if
vendors had to supply C++ functionality equivalent to LAPACK, they
would either need to start with a Fortran compiler, or would need to
invest a lot of effort in a C++ reimplementation. Mechanical
translation from Fortran to C++ introduces risk, because many LAPACK
functions depend critically on details of floating-point arithmetic
behavior.
Second, we intend to permit use of matrix or vector element types
other than just the four types that the BLAS and LAPACK support. This
includes "short" floating-point types, fixed-point types, integers,
and user-defined arithmetic types. Doing this is easier for BLAS-like
operations than for the much more complicated numerical algorithms in
LAPACK. LAPACK strives for a "generic" design (see Jack Dongarra
interview summary in [P1417R0](http://wg21.link/p1417r0)), but only supports
two real floating-point types and two complex floating-point types.
Directly translating LAPACK source code into a "generic" version could
lead to pitfalls. Many LAPACK algorithms only make sense for number
systems that aim to approximate real numbers (or their complex
extentions). Some LAPACK functions output error bounds that rely on
properties of floating-point arithmetic.
For these reasons, we have left LAPACK-like functionality for future
work. It would be natural for a future LAPACK-like C++ library to
build on our proposal.
Extended-precision BLAS
Our interface subsumes some functionality of the Mixed-Precision BLAS
specification (Chapter 4 of the BLAS Standard). For example, users
may multiply two 16-bit floating-point matrices (assuming that a
16-bit floating-point type exists) and accumulate into a 32-bit
floating-point matrix, just by providing a 32-bit floating-point
matrix as output. Users may specify the precision of a dot product
result. If it is greater than the input vectors' element type
precisions (e.g., `double` vs. `float`), then this effectively
performs accumulation in higher precision. Our proposal imposes
semantic requirements on some functions, like `vector_norm2`, to
behave in this way.
However, we do not include the "Extended-Precision BLAS" in this
proposal. The BLAS Standard lets callers decide at run time whether
to use extended precision floating-point arithmetic for internal
evaluations. We could support this feature at a later time.
Implementations of our interface also have the freedom to use more
accurate evaluation methods than typical BLAS implementations. For
example, it is possible to make floating-point sums completely

[independent of parallel evaluation
order](https://bebop.cs.berkeley.edu/reproblas/).
Arithmetic operators and associated expression templates
Our proposal omits arithmetic operators on matrices and vectors.
We do so for the following reasons:
1. We propose a low-level, minimal interface.
2. `operator*` could have multiple meanings for matrices and vectors.
   Should it mean elementwise product (like `valarray`) or matrix
   product? Should libraries reinterpret "vector times vector" as a
   dot product (row vector times column vector)? We prefer to let a
   higher-level library decide this, and make everything explicit at
   our lower level.
3. Arithmetic operators require defining the element type of the
   vector or matrix returned by an expression. Functions let users
   specify this explicitly, and even let users use different output
   types for the same input types in different expressions.
4. Arithmetic operators may require allocation of temporary matrix or
   vector storage. This prevents use of nonowning data structures.
5. Arithmetic operators strongly suggest expression templates. These
   introduce problems such as dangling references and aliasing.

Our goal is to propose a low-level interface. Other libraries, such
as that proposed by [P1385R1](http://wg21.link/p1385r1), could use our
interface to implement overloaded arithmetic for matrices and vectors.
[P0939R0](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0939r0.pdf)
advocates using "an incremental approach to design to benefit from
actual experience." A constrained, function-based, BLAS-like
interface builds incrementally on the many years of BLAS experience.
Arithmetic operators on matrices and vectors would require the
library, not necessarily the user, to specify the element type of an
expression's result. This gets tricky if the terms have mixed element
types. For example, what should the element type of the result of the
vector sum `x + y` be, if `x` has element type `complex<float>` and
`y` has element type `double`? It's tempting to use `common_type_t`,
but `common_type_t<complex<float>, double>` is `complex<float>`. This
loses precision. Some users may want `complex<double>`; others may
want `complex<long double>` or something else, and others may want to
choose different types in the same program.
[P1385R1](http://wg21.link/p1385r1) lets users customize the return type of
such arithmetic expressions. However, different algorithms may call
for the same expression with the same inputs to have different output
types. For example, iterative refinement of linear systems `Ax=b` can
work either with an extended-precision intermediate residual vector `r
= b - A*x`, or with a residual vector that has the same precision as
the input linear system. Each choice produces a different algorithm
with different convergence characteristics. Thus, our library lets
users specify the result element type of linear algebra operations
explicitly, by calling a named function that takes an output argument
explicitly, rather than an arithmetic operator.
Arithmetic operators on matrices or vectors may also need to allocate
temporary storage. Users may not want that. When LAPACK's developers
switched from Fortran 77 to a subset of Fortran 90, their users
rejected the option of letting LAPACK functions allocate temporary
storage on their own. Users wanted to control memory allocation.
Also, allocating storage precludes use of nonowning input data
structures like `basic_mdspan`, that do not know how to allocate.
Arithmetic expressions on matrices or vectors strongly suggest
expression templates, as a way to avoid allocation of temporaries and
to fuse computational kernels. They do not *require* expression
templates. For example, `valarray` offers overloaded operators for
vector arithmetic, but the Standard lets implementers decide whether
to use expression templates. However, all of the current C++ linear
algebra libraries that we mentioned above have some form of expression
templates for overloaded arithmetic operators, so users will expect
this and rely on it for good performance. This was, indeed, one of
the major complaints about initial implementations of `valarray`: its
lack of mandate for expression templates meant that initial
implementations were slow, and thus users did not want to rely on it.
(See Josuttis 1999, p. 547, and Vandevoorde and Josuttis 2003, p. 342,
for a summary of the history. Fortran has an analogous issue, in
which (under certain conditions) it is implementation defined whether
the run-time environment needs to copy noncontiguous slices of an
array into contiguous temporary storage.)
Expression templates work well, but have issues. Our papers
[P1417R0](http://wg21.link/p1417r0) and "Evolving a Standard C++ Linear
Algebra Library from the BLAS" (P1674R0) give more detail on these
issues. A particularly troublesome one is that modern C++ `auto`
makes it easy for users to capture expressions before their evaluation
and writing into an output array. For matrices and vectors with
container semantics, this makes it easy to create dangling references.
Users might not realize that they need to assign expressions to named
types before actual work and storage happen. [Eigen's

documentation](https://eigen.tuxfamily.org/dox/TopicPitfalls.html) describes this common problem.

Our `scaled_view`, `conjugate_view`, and `transpose_view` functions make use of one aspect of expression templates, namely modifying the array access operator. However, we intend these functions for use only as in-place modifications of arguments of a function call. Also, when modifying `basic_mdspan`, these functions merely view the same data that their input `basic_mdspan` views. They introduce no more potential for dangling references than `basic_mdspan` itself. The use of views like `basic_mdspan` is self-documenting; it tells users that they need to take responsibility for scope of the viewed data. We permit applying these functions to the container `basic_mdarray` (see P1684R0), but this has no more risk of dangling references than `vector::data` does.

Banded matrix layouts

This proposal omits banded matrix types. It would be easy to add the required layouts and specializations of algorithms later. The packed and unpacked symmetric and triangular layouts in this proposal cover the major concerns that would arise in the banded case, like nonstrided and nonunique layouts, and matrix types that forbid access to some multi-indices in the Cartesian product of extents.

Tensors

We exclude tensors from this proposal, for the following reasons. First, tensor libraries naturally build on optimized dense linear algebra libraries like the BLAS, so a linear algebra library is a good first step. Second, `mdspan` and `mdarray` have natural use as a low-level representation of dense tensors, so we are already partway there. Third, even simple tensor operations that naturally generalize the BLAS have infintely many more cases than linear algebra. It's not clear to us which to optimize. Fourth, even though linear algebra is a special case of tensor algebra, users of linear algebra have different interface expectations than users of tensor algebra. Thus, it makes sense to have two separate interfaces.

Design justification

We take a step-wise approach. We begin with core BLAS dense linear algebra functionality. We then deviate from that only as much as necessary to get algorithms that behave as much as reasonable like the existing C++ Standard Library algorithms. Future work or collaboration with other proposals could implement a higher-level interface. We also offer an option for an extension to "batched BLAS" in order to support machine learning and other use cases.

We propose to build the interface on top of `basic_mdspan`, as well as a new `basic_mdarray` variant of `basic_mdspan` with container semantics. We explain the value of these two classes below.

Please refer to our papers "Evolving a Standard C++ Linear Algebra Library from the BLAS" (P1674R0) and "Historical lessons for C++ linear algebra library standardization" [(P1417R0)](http://wg21.link/p1417r0). They will give details and references for many of the points that we summarize here.

We do not require using the BLAS library

Our proposal is based on the BLAS interface, and it would be natural for implementers to use an existing C or Fortran BLAS library. However, we do not require an underlying BLAS C interface. Vendors should have the freedom to decide whether they want to rely on an existing BLAS library. They may also want to write a "pure" C++ implementation that does not depend on an external library. They will, in any case, need a "generic" C++ implementation for matrix and vector element types other than the four that the BLAS supports.

Why use `basic_mdspan` and `basic_mdarray`?

* C++ does not currently have a data structure for representing multidimensional arrays.
* The BLAS' C interface takes a large number of pointer and integer arguments that represent matrices and vectors. Using multidimensional array data structures in the C++ interface reduces the number of arguments and avoids common errors.
* `basic_mdspan` and `basic_mdarray` support row-major, column-major, and strided layouts out of the box, and have `Layout` as an extension point. This lets our interface support layouts beyond what the BLAS Standard permits.
* They can exploit any dimensions or strides known at compile time.
* They have built-in "slicing" capabilities via `subspan`.
* Their layout and accessor policies will let us simplify our interfaces even further, by encapsulating transpose, conjugate, and scalar arguments. See below for details.
* `basic_mdspan` and `basic_mdarray` are low level; they impose no mathematical meaning on multidimensional arrays. This gives users the freedom to develop mathematical libraries with the semantics they want. (Some users object to calling something a "matrix" or "tensor" if it doesn't have the right mathematical properties. The Standard has already taken the word `vector`.)
* They offer a hook for future expansion to support heterogenous

  memory spaces. (This is a key feature of `Kokkos::View`, the data
  structure that inspired `basic_mdspan`.)
* Their encapsulation of matrix indexing makes C++ implementations of
  BLAS-like operations much less error prone and easier to read.
* They make it easier to support an efficient "batched" interface.

Why optionally include batched linear algebra?

* Batched interfaces expose more parallelism for many small linear
  algebra operations.
* Batched linear algebra operations are useful for many different
  fields, including machine learning.
* Hardware vendors offer both hardware features and optimized
  software libraries to support batched linear algebra.
* There is an ongoing [interface standardization
  effort](http://icl.utk.edu/bblas/), in which we participate.

Function argument aliasing and zero scalar multipliers

Summary:

1. The BLAS Standard forbids aliasing any input (read-only) argument
   with any output (write-only or read-and-write) argument.
2. The BLAS uses `INTENT(INOUT)` (read-and-write) arguments to express
   "updates" to a vector or matrix. By contrast, C++ Standard
   algorithms like `transform` take input and output iterator ranges
   as different parameters, but may let input and output ranges be the
   same.
3. The BLAS uses the values of scalar multiplier arguments ("alpha" or
   "beta") of vectors or matrices at run time, to decide whether to
   treat the vectors or matrices as write only. This matters both for
   performance and semantically, assuming IEEE floating-point
   arithmetic.
4. We decide separately, based on the category of BLAS function, how
   to translate `INTENT(INOUT)` arguments into a C++ idiom:
   a. For in-place triangular solve or triangular multiply, we
      translate the function to take separate input and output
      arguments that shall not alias each other.
   b. Else, if the BLAS function unconditionally updates (like
      `xGER`), we retain read-and-write behavior for that argument.
   c. Else, if the BLAS function uses a scalar `beta` argument to
      decide whether to read the output argument as well as write to
      it (like `xGEMM`), we provide two versions: a write-only version
      (as if `beta` is zero), and a read-and-write version (as if
      `beta` is nonzero).

For a detailed analysis, see "Evolving a Standard C++ Linear Algebra
Library from the BLAS" (P1674R0).

Support for different matrix layouts

Summary:

1. The dense BLAS supports several different dense matrix "types."
   Type is a mixture of "storage format" (e.g., packed, banded) and
   "mathematical property" (e.g., symmetric, Hermitian, triangular).
2. Some "types" can be expressed as custom `basic_mdspan` layouts;
   others do not.
3. Thus, a C++ BLAS wrapper cannot overload on matrix "type" simply by
   overloading on `basic_mdspan` specialization. The wrapper must use
   different function names, tags, or some other way to decide what
   the matrix type is.

For more details, including a list and description of the matrix
"types" that the dense BLAS supports, see our paper "Evolving a
Standard C++ Linear Algebra Library from the BLAS" (P1674R0) lists the
different matrix types.

A C++ linear algebra library has a few possibilities for
distinguishing the matrix "type":

1. It could imitate the BLAS, by introducing different function names,
   if the layouts and accessors do not sufficiently describe the
   arguments.
2. It could introduce a hierarchy of higher-level classes for
   representing linear algebra objects, use `basic_mdspan` (or
   something like it) underneath, and write algorithms to those
   higher-level classes.
3. It could use the layout and accessor types in `basic_mdspan` simply
   as tags to indicate the matrix "type." Algorithms could specialize
   on those tags.

We have chosen Approach 1. Our view is that a BLAS-like interface
should be as low-level as possible. Approach 2 is more like a "Matlab
in C++"; a library that implements this could build on our proposal's
lower-level library. Approach 3 _sounds_ attractive. However, most
BLAS matrix "types" do not have a natural representation as layouts.
Trying to hack them in would pollute `basic_mdspan` -- a simple class
meant to be easy for the compiler to optimize -- with extra baggage
for representing what amounts to sparse matrices. We think that BLAS
matrix "type" is better represented with a higher-level library that
builds on our proposal.

Caveats

This proposal does not yet have full wording. We have filled in

enough wording for meaningful design discussions, such as those presented in "Options and votes" below.

Data structures and utilities borrowed from other proposals

`basic_mdspan`

[P0009R9](http://wg21.link/p0009r9) is a proposal for adding multidimensional arrays to the C++ Standard Library. `basic_mdspan` is the main class in this proposal. It is a "view" (in the sense of `span`) of a multidimensional array. The rank (number of dimensions) is fixed at compile time. Users may specify some dimensions at run time and others at compile time; the type of the `basic_mdspan` expresses this. `basic_mdspan` also has two customization points:

* `Layout` expresses the array's memory layout: e.g., row-major (C++ style), column-major (Fortran style), or strided. We use a custom `Layout` later in this paper to implement a "transpose view" of an existing `basic_mdspan`.
* `Accessor` defines the storage handle (i.e., `pointer`) stored in the `mdspan`, as well as the reference type returned by its access operator. This is an extension point for modifying how access happens, for example by using `atomic_ref` to get atomic access to every element. We use custom `Accessor`s later in this paper to implement "scaled views" and "conjugated views" of an existing `basic_mdspan`.

The `basic_mdspan` class has an alias `mdspan` that uses the default `Layout` and `Accessor`. In this paper, when we refer to `mdspan` without other qualifiers, we mean the most general `basic_mdspan`.

`basic_mdarray`

`basic_mdspan` views an existing memory allocation. It does not give users a way to allocate a new array, even if the array has all compile-time dimensions. Furthermore, `basic_mdspan` always stores a pointer. For very small matrices or vectors, this is not a zero-overhead abstraction. Also, it's often more natural to pass around very small objects by value. For these reasons, our paper (P1684R0) proposes a new class `basic_mdarray`.

`basic_mdarray` is a new kind of container, with the same deep copy behavior as `vector`. It has the same extension points as `basic_mdspan`, and also has the ability to use any *contiguous container* (see **[container.requirements.general]**) for storage. Contiguity matters because `basic_mdspan` views a subset of a contiguous pointer range, and we want to be able to get a `basic_mdspan` that views the `basic_mdarray`. `basic_mdarray` will come with support for two different underlying containers: `array` and `vector`. A `subspan` (see [P0009R9](http://wg21.link/p0009r9)) of a `basic_mdarray` will return a `basic_mdspan` with the appropriate layout and corresponding accessor. Users must guard against dangling pointers, just as they currently must do when using `span` to view a subset of a `vector`.

The `basic_mdarray` class has an alias `mdarray` that uses default policies. In this paper, when we refer to `mdarray` without other qualifiers, we mean `basic_mdarray`.

Data structures and utilities

Layouts

Our proposal uses the layout policy of `basic_mdspan` and `basic_mdarray` in order to represent different matrix and vector data layouts. Layouts as described by P0009R9 come in three different categories:

* Unique
* Contiguous
* Strided

P0009R9 includes three different layouts -- `layout_left`, `layout_right`, and `layout_stride` -- all of which are unique, contiguous, and strided.

This proposal includes the following additional layouts:

* `layout_blas_general`: Generalization of `layout_left` and `layout_right`; describes layout used by General matrix "type"
* `layout_blas_packed`: Describes layout used by the BLAS' Symmetric Packed (SP), Hermitian Packed (HP), and Triangular Packed (TP) "types"

These layouts have "tag" template parameters that control their properties; see below.

We do not include layouts for unpacked "types," such as Symmetric (SY), Hermitian (HE), and triangular (TR). P1674R0 explains our reasoning. In summary: Their actual layout -- the arrangement of matrix elements in memory -- is the same as General. The only differences are constraints on what entries of the matrix algorithms may access, and assumptions about the matrix's mathematical properties. Trying to express those constraints or assumptions as "layouts" or "accessors" violates the spirit (and sometimes the law) of `basic_mdspan`.

The packed matrix "types" do describe actual arrangements of matrix elements in memory that are not the same as in General. This is why we provide `layout_blas_packed`. Note that these layouts would thus

be the first additions to the layouts in P0009R9 that are not unique, contiguous, and strided.
Algorithms cannot be written generically if they permit arguments with nonunique layouts, especially output arguments. Nonunique output arguments require specialization of the algorithm to the layout, since there's no way to know generically at compile time what indices map to the same matrix element. Thus, we impose the following rule: Any `basic_mdspan` or `basic_mdarray` argument to our functions must always have unique layout (`is_always_unique()` is `true`), unless otherwise specified.
Some of our functions explicitly require outputs with specific nonunique layouts. This includes low-rank updates to symmetric or Hermitian matrices, and matrix-matrix multiplication with symmetric or Hermitian matrices.
Tag classes for layouts
We use tag classes to parameterize a small number of layout names. Layouts take tag types as template arguments, and function callers use the corresponding `constexpr` instances of tag types for compile-time control of function behavior.
Storage order tags
```c++
struct column_major_t {
  constexpr explicit column_major_t() noexcept = default;
};
inline constexpr column_major_t column_major = { };
struct row_major_t {
  constexpr explicit row_major_t() noexcept = default;
};
inline constexpr row_major_t row_major = { };
```
`column_major_t` indicates a column-major order, and `row_major_t` indicates a row-major order. The interpretation of each depends on the specific layout that uses the tag. See `layout_blas_general` and `layout_blas_packed`.
Triangle tags
Linear algebra algorithms find it convenient to distinguish between the "upper triangle," "lower triangle," and "diagonal" of a matrix.
* The *upper triangle* of a matrix `A` is the set of all elements of `A` accessed by `A(i,j)` with `i >= j`.
* The *lower triangle* of `A` is the set of all elements of `A` accessed by `A(i,j)` with `i <= j`.
* The *diagonal* is the set of all elements of `A` accessed by `A(i,i)`. It is included in both the upper triangle and the lower triangle.
```c++
struct upper_triangle_t {
  constexpr explicit upper_triangle_t() noexcept = default;
};
inline constexpr upper_triangle_t upper_triangle = { };
struct lower_triangle_t {
  constexpr explicit lower_triangle_t() noexcept = default;
};
inline constexpr lower_triangle_t lower_triangle = { };
```
These tag classes specify whether algorithms and other users of a matrix (represented as a `basic_mdspan` or `basic_mdarray`) should access the upper triangle (`upper_triangular_t`) or lower triangle (`lower_triangular_t`) of the matrix. This is also subject to the restrictions of `implicit_unit_diagonal_t` if that tag is also applied; see below.
Diagonal tags
```c++
struct implicit_unit_diagonal_t {
  constexpr explicit implicit_unit_diagonal_t() noexcept = default;
};
inline constexpr implicit_unit_diagonal_t implicit_unit_diagonal = { };
struct explicit_diagonal_t {
  constexpr explicit explicit_diagonal_t() noexcept = default;
};
inline constexpr explicit_diagonal_t explicit_diagonal = { };
```
These tag classes specify what algorithms and other users of a matrix (represented as a `basic_mdspan` or `basic_mdarray`) should assume about the diagonal entries of the matrix, and whether algorithms and users of the matrix should access those diagonal entries explicitly.
The `implicit_unit_diagonal_t` tag indicates two things:
  * the function will never access the `i,i` element of the matrix, and
  * the matrix has a diagonal of ones (a unit diagonal).
[Note: Typical BLAS practice is that the algorithm never actually needs to form an explicit `1.0`, so there is no need to impose a

constraint that `1` or `1.0` is convertible to the matrix's
`element_type`. --*end note]*
The tag `explicit_diagonal_t` indicates that algorithms and other
users of the viewer may access the matrix's diagonal entries directly.
Side tags
Linear algebra algorithms find it convenient to distinguish between
applying some operator to the left side of an object, or the right
side of an object. *[Note:* Matrix-matrix product and triangular
solve with a matrix generally do not commute. --*end note]*
```c++
struct left_side_t {
  constexpr explicit left_side_t() noexcept = default;
};
constexpr left_side_t left_side = { };
struct right_side_t {
  constexpr explicit right_side_t() noexcept = default;
};
constexpr right_side_t right_side = { };
```
These tag classes specify whether algorithms should apply some
operator to the left side (`left_side_t`) or right side
(`right_side_t`) of an object.
New "General" layouts
```c++
template<class StorageOrder>
class layout_blas_general;
```
* *Constraints:* `StorageOrder` is either `column_major_t` or
  `row_major_t`.
These new layouts represent exactly the layout assumed by the General
(GE) matrix type in the BLAS' C binding.
* `layout_blas_general<column_major_t>` represents a column-major
  matrix layout, where the stride between columns (in BLAS terms,
  "leading dimension of the matrix A" or `LDA`) may be greater than or
  equal to the number of rows.
* `layout_blas_general<row_major_t>` represents a row-major matrix
  layout, where the stride (again, `LDA`) between rows may be greater
  than or equal to the number of columns.
These layouts are both always unique and always strided. They are
contiguous if and only if the "leading dimension" equals the number of
rows resp. columns. Both layouts are more general than `layout_left`
and `layout_right`, because they permit a stride between columns
resp. rows that is greater than the corresponding extent. This is why
BLAS functions take an `LDA` (leading dimension of the matrix A)
argument separate from the dimensions (extents, in `mdspan` terms) of
A. However, these layouts are slightly *less* general than
`layout_stride`, because they assume contiguous storage of columns
resp. rows. See P1674R0 for further discussion.
These new layouts have natural generalizations to ranks higher than 2.
The definition of each of these layouts would look and work like
`layout_stride`, except for the following differences:
* `layout_blas_general::mapping` would be templated on two `extent`
  types. The first would express the `mdspan`'s dimensions, just like
  with `layout_left`, `layout_right`, or `layout_stride`. The second
  would express the `mdspan`'s strides. The second `extent` would
  have rank `rank()-1`.
* `layout_blas_general::mapping`'s constructor would take an instance
  of each of these two `extent` specializations.
These new layouts differ intentionally from `layout_stride`, which (as
of P0009R9) takes strides all as run-time elements in an `array`. (We
favor changing this in the next revision of P0009, to make
`layout_stride` take the strides as a second `extents` object.) We
want users to be able to express strides as an arbitrary mix of
compile-time and run-time values, just as they can express dimensions.
Packed layouts
```c++
template<class Triangle,
         class StorageOrder>
class layout_blas_packed;
```
Requirements
Throughout this Clause, where the template parameters are not
constrained, the names of template parameters are used to express type
requirements.
  * `Triangle` is either `upper_triangle_t` or `lower_triangle_t`.
  * `StorageOrder` is either `column_major_t` or `row_major_t`.
Packed layout mapping
The BLAS' packed matrix "types" all store the represented entries of
the matrix contiguously. They start at the top left side of the
matrix.
A `StorageOrder` of `column_major_t` indicates column-major ordering.

This packs matrix elements starting with the leftmost (least column index) column, and proceeding column by column, from the top entry (least row index). A `StorageOrder` of `row_major_t` indicates row-major ordering. This packs matrix elements starting with the topmost (least row index) row, and proceeding row by row, from the leftmost (least column index) entry.
Whether the "type" stores the upper or lower triangle of the matrix matters for the layout, not just for the matrix's mathematical properties. Thus, the choice of upper or lower triangle must be part of the layout. `Triangle=upper_triangle_t` means that the layout represents the upper triangle; `Triangle=lower_triangle_t` means that the layout represents the lower triangle. We will describe the mapping as a function of `StorageOrder` and `Triangle` below.
Packed layouts require that the matrix/matrices are square. That is, the rightmost two extents (`extents(extents().rank()-2)` and `extents(extents().rank()-1)`) are equal.
Packed layouts generalize just like unpacked layouts to "batches" of matrices. The last two (rightmost) indices index within a matrix, and the remaining index/indices identify which matrix.
Let N be `extents(extents().rank()-1)`. (That is, each matrix in the batch has N rows and N columns.) Let `i,j` be the last two (rightmost) indices in the `is` parameter pack given to the packed layout's `mapping::operator()`.
* For the upper triangular, column-major format, index pair i,j maps to i + (1 + 2 + ... + j).
* For the lower triangular, column-major format, index pair i,j maps to i + (1 + 2 + ... + N-j-1).
* For the upper triangular, row-major format, index pair i,j maps to j + (1 + 2 + ... + i).
* For the lower triangular, row-major format, index pair i,j maps to j + (1 + 2 + ... + N-i-1).

[Note: Whether or not the storage format has an implicit unit diagonal (see the `implicit_unit_diagonal_t` tag above) does not change the mapping. This means that packed matrix storage "wastes" the unit diagonal, if present. This follows BLAS convention; see Section 2.2.4 of the BLAS Standard. It also has the advantage that every index pair `i,j` in the Cartesian product of the extents maps to a valid (though wrong) codomain index. This is why we declare the packed layout mappings as "nonunique." --*end note]*

Packed layout views
The idea behind packed matrix types is that users take an existing 1-D array, and view it as a matrix data structure. We adapt this approach to our library by including functions that create a "packed view" of an existing `basic_mdspan` or `basic_mdarray`. The resulting packed object has one higher rank.
Requirements
Throughout this Clause, where the template parameters are not constrained, the names of template parameters are used to express type requirements.
  * `Extents::rank()` is at least 1.
  * `Layout` is a unique, contiguous, and strided layout.
  * `Triangle` is either `upper_triangle_t` or `lower_triangle_t`.
  * `StorageOrder` is either `column_major_t` or `row_major_t`.

Create a packed triangular view of an existing object
```c++
template<class EltType,
         class Extents,
         class Layout,
         class Accessor,
         class Triangle,
         class StorageOrder>
constexpr basic_mdspan<EltType,
  <i>extents-see-returns-below</i>,
  layout_blas_packed<
    Triangle,
    StorageOrder>,
  Accessor>
packed_view(
  const basic_mdspan<EltType, Extents, Layout, Accessor>& m,
  typename basic_mdspan<EltType, Extents, Layout, Accessor>::index_type
num_rows,
  Triangle,
  StorageOrder);
template<class EltType,
         class Extents,
         class Layout,
         class Accessor,
         class Triangle,
         class DiagonalStorage,
         class StorageOrder>
constexpr basic_mdspan<const EltType,
```

```
  <i>extents-see-returns-below</i>,
  layout_blas_packed<
    Triangle,
    StorageOrder>,
  Accessor>
packed_view(
  const basic_mdarray<EltType, Extents, Layout, Accessor>& m,
  typename basic_mdarray<EltType, Extents, Layout, Accessor>::index_type
num_rows,
  Triangle,
  StorageOrder);
template<class EltType,
         class Extents,
         class Layout,
         class Accessor,
         class Triangle,
         class DiagonalStorage,
         class StorageOrder>
constexpr basic_mdspan<EltType,
  <i>extents-see-returns-below</i>,
  layout_blas_triangular_packed<
    Triangle,
    StorageOrder>,
  Accessor>
packed_view(
  basic_mdarray<EltType, Extents, Layout, Accessor>& m,
  typename basic_mdarray<EltType, Extents, Layout, Accessor>::index_type
num_rows,
  Triangle,
  StorageOrder);
```
* *Requires:* If `num_rows` is nonzero, then `m.extent(0)` is at least (`num_rows` + 1) * `num_rows` / 2.
* *Effects:* Views the given `basic_mdspan` or `basic_mdarray` in packed layout, with the given `Triangle` and `StorageOrder`, where each matrix (corresponding to the rightmost two extents of the result) has `num_rows` rows and columns.
* *Returns:* A `basic_mdspan` `r` with packed layout and the following properties:
  * `r.extent(r.rank()-2)` equals `num_rows`.
  * `r.extent(r.rank()-1)` equals `num_rows`.
  * Let `E_r` be the type of `r.extents()`. Then,
    * `E_r::rank()` is one plus `Extents::rank()`, and
    * `E_r::rank() - E_r::dynamic_rank()` (the number of static extents) is no less than `Extents::rank() - Extents::dynamic_rank()`.

Scaled view of an object

Most BLAS functions that take scalar arguments use those arguments as a transient scaling of another vector or matrix argument. For example, `xAXPY` computes `y = alpha*x + y`, where `x` and `y` are vectors, and `alpha` is a scalar. Scalar arguments help make the BLAS more efficient by combining related operations and avoiding temporary vectors or matrices. In this `xAXPY` example, users would otherwise need a temporary vector `z = alpha*x` (`xSCAL`), and would need to make two passes over the input vector `x` (once for the scale, and another for the vector add). However, scalar arguments complicate the interface.

We can solve all these issues in C++ by introducing a "scaled view" of an existing vector or matrix, via a changed `basic_mdspan` `Accessor`. For example, users could imitate what `xAXPY` does by using our `linalg_add` function (see below) as follows:

```c++
mdspan<double, extents<dynamic_extent>> y = ...;
mdspan<double, extents<dynamic_extent>> x = ...;
double alpha = ...;
linalg_add(scaled_view(alpha, x), y, y);
```

The resulting operation would only need to iterate over the entries of the input vector `x` and the input / output vector `y` once. An implementation could dispatch to the BLAS by noticing that the first argument has an `accessor_scaled` (see below) `Accessor` type, extracting the scalar value `alpha`, and calling the corresponding `xAXPY` function (assuming that `alpha` is nonzero; see discussion above).

The same `linalg_add` interface would then support the operation `w := alpha*x + beta*y`:

```c++
linalg_add(scaled_view(alpha, x), scaled_view(beta, y), w);
```

Note that this operation could not dispatch to an existing BLAS library, unless the library implements the `xWAXPBY` function

specified in the BLAS Standard. However, implementations could specialize on the result of a `scaled_view`, in order to transform the user's arguments into something suitable for a BLAS library call. For example, if the user calls `matrix_product` (see below) with `A` and `B` both results of `scaled_view`, then the implementation could combine both scalars into a single "alpha" and call `xGEMM` with it.

`scaled_scalar`

`scaled_scalar` expresses a scaled version of an existing scalar. This must be read only. *[Note:* This avoids likely confusion with the definition of "assigning to a scaled scalar." --*end note]*

[Note:
`scaled_scalar` and `conjugated_scalar` (see below) behave a bit like `atomic_ref`, in that they are special `reference` types for the Accessors `accessor_scaled` resp. `accessor_conjugate` (see below), and that they provide overloaded arithmetic operators that implement a limited form of expression templates. The arithmetic operators are templated on their input type, so that they only need to compile if an algorithm actually uses them. The conversion to `T` allows simple assignment to `T`, or invocation in functions that take `T`. There are other surprising results outside the scope of this proposal to fix, like the fact that `operator*` does not work for `complex<float>` times `double`. This means `scaled_view(x, 93.0)` for `x` with `element_type` `complex<float>` will not compile. Neither does `complex<float>(5.0, 6.0) * y`. Our experience with generic numerical algorithms is that floating-point literals need type adornment.
-*end note]*

```c++
template<class T, class S>
class scaled_scalar {
public:
  scaled_scalar(const T& v, const S& s) :
    val(v), scale(s) {}
  operator T() const { return val * scale; }
  T operator- () const { return -(val * scale); }
  template<class T2>
  decltype(auto) operator+ (const T2& upd) const {
    return val*scale + upd;
  }
  template<class T2>
  decltype(auto) operator* (const T2 upd) const {
    return val*scale * upd;
  }
  // ... add only those operators needed for the functions
  // in this proposal ...
private:
  const T& val;
  const S scale;
};
```

`accessor_scaled`

Accessor to make `basic_mdspan` return a `scaled_scalar`.

```c++
template<class Accessor, class S>
class accessor_scaled {
public:
  using element_type  = Accessor::element_type;
  using pointer       = Accessor::pointer;
  using reference     = scaled_scalar<Accessor::reference,S>;
  using offset_policy = accessor_scaled<Accessor::offset_policy,S>;
  accessor_scaled(Accessor a, S sval) :
    acc(a), scale_factor(sval) {}
  reference access(pointer& p, ptrdiff_t i) const noexcept {
    return reference(acc.access(p,i), scale_factor);
  }
  offset_policy::pointer offset(pointer p, ptrdiff_t i) const noexcept {
    return a.offset(p,i);
  }
  element_type* decay(pointer p) const noexcept {
    return a.decay(p);
  }
private:
  Accessor acc;
  S scale_factor;
};
```

`scaled_view`

Return a scaled view using a new accessor.

```c++
template<class T, class Extents, class Layout,
         class Accessor, class S>
```

```
basic_mdspan<T, Extents, Layout, accessor_scaled<Accessor, S>>
scaled_view(S s, const basic_mdspan<T, Extents, Layout, Accessor>& a);
template<class T, class Extents, class Layout,
         class Accessor, class S>
basic_mdspan<const T, Extents, Layout, <i>see-below</i> >
scaled_view(S s, const basic_mdarray<T, Extents, Layout, Accessor>& a);
```
The Accessor type of the `basic_mdspan` returned by the overload that takes `basic_mdarray` is `accessor_scaled<ConstAccessor, S>`, where `ConstAccessor` is an implementation-defined type. See P1684R0 for details.
Example:
```c++
void test_scaled_view(basic_mdspan<double, extents<10>> a)
{
  auto a_scaled = scaled_view(5.0, a);
  for(int i = 0; i < a.extent(0); ++i)
    assert(a_scaled(i) == 5.0 * a(i));
}
```
Conjugated view of an object
Some BLAS functions of matrices also take an argument that specifies whether to view the transpose or conjugate transpose of the matrix. The BLAS uses this argument to modify a read-only input transiently. This means that users can let the BLAS work with the data in place, without needing to compute the transpose or conjugate transpose explicitly. However, it complicates the BLAS interface.
Just as we did above with "scaled views" of an object, we can apply the complex conjugate operation to each element of an object using a special accessor.
What does the complex conjugate mean for non-complex numbers? We use the convention that the "complex conjugate" of a non-complex number is just the number. This makes sense mathematically, if we embed a field (of real numbers) in the corresponding set of complex numbers over that field, as all complex numbers with zero imaginary part. It's also the convention that the [Trilinos](https://github.com/trilinos/Trilinos) library (among others) uses. However, as we will show below, this does not work with the C++ Standard Library's definition of `conj`. We deal with this by defining `conjugate_view` so that it does not use `conj` for real element types.
`conjugated_scalar`
`conjugated_scalar` expresses a conjugated version of an existing scalar. This must be read only. *[Note:* This avoids likely confusion with the definition of "assigning to the conjugate of a scalar." --*end note]*
The C++ Standard imposes the following requirements on `complex<T>` numbers:
1. `T` may only be `float`, `double`, or `long double`.
2. Overloads of `conj(const T&)` exist for `T=float`, `double`, `long double`, or any built-in integer type, but all these overloads have return type `complex<U>` with `U` either `float`, `double`, or `long double`. (See **[cmplx.over]**.)

We need the return type of `conjugated_scalar`'s arithmetic operators to be the same as the type of the scalar that it wraps. This means that `conjugated_scalar` only works for `complex<T>` scalar types. Users cannot define custom types that are complex numbers. (The alternative would be to permit users to specialize `conjugated_scalar`, but we didn't want to add a *customization point* in the sense of **[namespace.std]**. Our definition of `conjugated_scalar` is compatible with any future expansion of the C++ Standard to permit `complex<T>` for other `T`.)
```c++
template<class T>
class conjugated_scalar {
public:
  using value_type = T;
  conjugated_scalar(const T& v) : val(v) {}
  operator T() const { return conj(val); }
  template<class T2>
  T operator* (const T2 upd) const {
    return conj(val) * upd;
  }
  template<class T2>
  T operator+ (const T2 upd) const {
    return conj(val) + upd;
  }
  // ... add only those operators needed for the functions in this
  // proposal ...
private:
  const T& val;
```

```
};
```
`accessor_conjugate`
The `accessor_conjugate` Accessor makes `basic_mdspan` access return a
`conjugated_scalar` if the scalar type is `std::complex<R>` for some
`R`. Otherwise, it makes `basic_mdspan` access return the original
`basic_mdspan`'s reference type.
```c++
template<class Accessor, class T>
class accessor_conjugate {
public:
  using element_type  = Accessor::element_type;
  using pointer       = Accessor::pointer;
  using reference     = Accessor::reference;
  using offset_policy = Accessor::offset_policy;
  accessor_conjugate(Accessor a) : acc(a) {}
  reference access(pointer p, ptrdiff_t i) const noexcept {
    return reference(acc.access(p,i),scale_factor);
  }
  offset_policy::pointer offset(pointer p, ptrdiff_t i) const noexcept {
    return a.offset(p,i);
  }
  element_type* decay(pointer p) const noexcept {
    return a.decay(p);
  }
private:
  Accessor acc;
};
template<class Accessor, class T>
class accessor_conjugate<Accessor, std::complex<T>> {
public:
  using element_type  = Accessor::element_type;
  using pointer       = Accessor::pointer;
  using reference     =
    conjugated_scalar<Accessor::reference, std::complex<T>>;
  using offset_policy =
    accessor_conjugate<Accessor::offset_policy, std::complex<T>>;
  accessor_conjugate(Accessor a) : acc(a) {}
  reference access(pointer p, ptrdiff_t i) const noexcept {
    return reference(acc.access(p,i),scale_factor);
  }
  offset_policy::pointer offset(pointer p, ptrdiff_t i) const noexcept {
    return a.offset(p,i);
  }
  element_type* decay(pointer p) const noexcept {
    return a.decay(p);
  }
private:
  Accessor acc;
};
```
`conjugate_view`
The `conjugate_view` function returns a conjugated view using a new
accessor.
```c++
template<class EltType, class Extents, class Layout, class Accessor>
basic_mdspan<EltType, Extents, Layout,
             accessor_conjugate<Accessor, EltType>>
conjugate_view(basic_mdspan<EltType, Extents, Layout, Accessor> a);
template<class EltType, class Extents, class Layout, class Accessor>
basic_mdspan<const EltType, Extents, Layout, <i>see-below</i> >
conjugate_view(const basic_mdarray<EltType, Extents, Layout, Accessor>& a);
```
The Accessor type of the `basic_mdspan` returned by the overload that
takes `basic_mdarray` is `accessor_conjugate<ConstAccessor, S>`, where
`ConstAccessor` is an implementation-defined type. See P1684R0 for
details.
Example:
```c++
void test_conjugate_view(basic_mdspan<complex<double>, extents<10>>)
{
  auto a_conj = conjugate_view(a);
  for(int i = 0; i < a.extent(0); ++i)
    assert(a_conj(i) == conj(a(i));
}
```
[Note: Instead of a partial specialization of `accessor_conjugate`,
one could have different overloads of `conjugate_view` that return fo
non-complex scalar types the same accessor as the input
argument. --*end note]*
Transpose view of an object

Many BLAS functions of matrices take an argument that specifies
whether to view the transpose or conjugate transpose of the matrix.
The BLAS uses this argument to modify a read-only input transiently.
This means that users can let the BLAS work with the data in place,
without needing to compute the transpose or conjugate transpose
explicitly. However, it complicates the BLAS interface.
Just as we did above with a "scaled view" of an object, we can
construct a "transposed view" or "conjugate transpose" view of an
object. This lets us simplify the interface.
An implementation could dispatch to the BLAS by noticing that the
first argument has a `layout_transpose` (see below) `Layout` type
(in both transposed and conjugate transposed cases), and/or an
`accessor_conjugate` (see below) `Accessor` type (in the conjugate
transposed case). It could use this information to extract the
appropriate run-time BLAS parameters.
`layout_transpose`
This layout wraps an existing layout, and swaps its rightmost two
indices.
```c++
template<class Layout>
class layout_transpose {
  struct mapping {
    Layout::mapping nested_mapping;
    mapping(Layout::mapping map):nested_mapping(map) {}
    // ... insert other standard mapping things ...
    // for non-batched layouts
    ptrdiff_t operator() (ptrdiff_t i, ptrdiff_t j) const {
      return nested_mapping(j, i);
    }
    // for batched layouts
    ptrdiff_t operator() (ptrdiff_t... rest, ptrdiff_t i, ptrdiff_t j) const {
      return nested_mapping(rest..., j, i);
    }
  };
};
```
* *Constraints:*
  * `Layout` is a unique layout.
  * `Layout::mapping::rank()` is at least 2.
`transpose_view`
The `transpose_view` function returns a transposed view of an object.
For rank-2 objects, the transposed view swaps the row and column
indices. In the batched (higher rank) case, the transposed view swaps
the rightmost two indices.
Note that `transpose_view` always returns a `basic_mdspan` with the
`layout_transpose` argument. This gives a type-based indication of
the transpose operation. However, functions' implementations may
convert the `layout_transpose` object to an object with a different
but equivalent layout. For example, functions can view the transpose
of a `layout_blas<column_major_t>` matrix as a
`layout_blas<row_major_t>` matrix. (This is a classic technique for
supporting row-major matrices using the Fortran BLAS interface.)
```c++
template<class EltType, class Extents, class Layout, class Accessor>
basic_mdspan<EltType, Extents, layout_transpose<Layout>, Accessor>>
transpose_view(basic_mdspan<EltType, Extents, Layout, Accessor> a);
template<class EltType, class Extents, class Layout, class Accessor>
basic_mdspan<EltType, Extents, layout_transpose<Layout>, <i>see-below</i> >
transpose_view(const basic_mdarray<EltType, Extents, Layout, Accessor>& a);
```
The Accessor type of the `basic_mdspan` returned by the overload that
takes `basic_mdarray` is an implementation-defined type. See P1684R0
for details.
Conjugate transpose view
The `conjugate_transpose_view` function returns a conjugate transpose
view of an object. This combines the effects of `transpose_view` and
`conjugate_view`.
```c++
template<class EltType, class Extents, class Layout, class Accessor>
basic_mdspan<EltType, Extents, layout_transpose<Layout>,
             accessor_conjugate<Accessor>>>
conjugate_transpose_view(
  basic_mdspan<EltType, Extents, Layout, Accessor> a);
template<class EltType, class Extents, class Layout, class Accessor>
basic_mdspan<EltType, Extents, layout_transpose<Layout>,
             <i>see-below</i> >
conjugate_transpose_view(
  const basic_mdarray<EltType, Extents, Layout, Accessor>& a)
```
The Accessor type of the `basic_mdspan` returned by the overload that
takes `basic_mdarray` is `accessor_conjugate<ConstAccessor, S>`, where

`ConstAccessor` is an implementation-defined type. See P1684R0 for
details.
Algorithms
Requirements
Throughout this Clause, where the template parameters are not
constrained, the names of template parameters are used to express type
requirements. In the requirements below, we use `*` in a typename to
denote a "wildcard," that matches zero characters, `_1`, `_2`, `_3`,
or other things as appropriate.
* Algorithms that have a template parameter named `ExecutionPolicy`
  are parallel algorithms **[algorithms.parallel.defns]**.
* `Scalar` meets the requirements of `SemiRegular<Scalar>`. (Some
  algorithms below impose further requirements.)
* `Real` is any of the following types: `float`, `double`, or `long
  double`.
* `in_vector*_t` is a rank-1 `basic_mdarray` or `basic_mdspan` with a
  potentially `const` element type and a unique layout. If the algorithm
  accesses the object, it will do so in read-only fashion.
* `inout_vector*_t` is a rank-1 `basic_mdarray` or `basic_mdspan`
  with a non-`const` element type and a unique layout.
* `out_vector*_t` is a rank-1 `basic_mdarray` or `basic_mdspan` with
  a non-`const` element type and a unique layout. If the algorithm
  accesses the object, it will do so in write-only fashion.
* `in_matrix*_t` is a rank-2 `basic_mdarray` or `basic_mdspan` with a
  `const` element type. If the algorithm accesses the object, it will
  do so in read-only fashion.
* `inout_matrix*_t` is a rank-2 `basic_mdarray` or `basic_mdspan`
  with a non-`const` element type.
* `out_matrix*_t` is a rank-2 `basic_mdarray` or `basic_mdspan` with
  a non-`const` element type. If the algorithm accesses the object,
  it will do so in write-only fashion.
* `in_object*_t` is a rank-1 or rank-2 `basic_mdarray` or
  `basic_mdspan` with a potentially `const` element type and a unique
  layout. If the algorithm accesses the object, it will do so in read-only
  fashion.
* `inout_object*_t` is a rank-1 or rank-2 `basic_mdarray` or
  `basic_mdspan` with a non-`const` element type and a unique layout.
* `out_object*_t` is a rank-1 or rank-2 `basic_mdarray` or
  `basic_mdspan` with a non-`const` element type and a unique layout.
* `Triangle` is either `upper_triangle_t` or `lower_triangle_t`.
* `DiagonalStorage` is either `implicit_unit_diagonal_t` or
  `explicit_diagonal_t`.
* `Side` is either `left_side_t` or `right_side_t`.
* `in_*_t` template parameters may deduce a `const` lvalue reference
   or a (non-`const`) rvalue reference to a `basic_mdarray` or a
   `basic_mdspan`.
* `inout_*_t` and `out_*_t` template parameters may deduce a `const` lvalue
  reference to a `basic_mdspan`, a (non-`const`) rvalue reference to a
  `basic_mdspan`, or a non-`const` lvalue reference to a `basic_mdarray`.
BLAS 1 functions
[Note:
The BLAS developed in three "levels": 1, 2, and 3. BLAS 1 includes
vector-vector operations, BLAS 2 matrix-vector operations, and BLAS 3
matrix-matrix operations. The level coincides with the number of
nested loops in a naÃ¯ve sequential implementation of the operation.
Increasing level also comes with increasing potential for data reuse.
The BLAS traditionally lists computing a Givens rotation among the
BLAS 1 operations, even though it only operates on scalars.
--*end note]*
Givens rotations
Compute Givens rotations
```c++
template<class Real>
void givens_rotation_setup(const Real a,
                           const Real b,
                           Real& c,
                           Real& s);
template<class Real>
void givens_rotation_setup(const complex<Real>& a,
                           const complex<Real>& b,
                           Real& c,
                           complex<Real>& s);
```
This function computes the plane (Givens) rotation represented by the
two values `c` and `s` such that the mathematical expression
```
[c        s]    [a]    [r]
[          ] *  [ ] =  [ ]
[-conj(s) c]    [b]    [0]
```
holds, where `conj` indicates the mathematical conjugate of `s`, `c`

is always a real scalar, and `c*c + abs(s)*abs(s)` equals one. That is, `c` and `s` represent a 2 x 2 matrix, that when multiplied by the right by the input vector whose components are `a` and `b`, produces a result vector whose first component `r` is the Euclidean norm of the input vector, and whose second component as zero. *[Note:* The C++ Standard Library `conj` function always returns `complex<T>` for some `T`, even though overloads exist for non-complex input. The above exprssion uses `conj` as mathematical notation, not as code. --*end note]*
[Note: This function corresponds to the BLAS function `xROTG`. It has an overload for complex numbers, because the output argument `c` (cosine) is a signed magnitude. --*end note]*
* *Constraints:* `Real` is `float`, `double`, or `long double`.
* *Effects:* Assigns to `c` and `s` the plane (Givens) rotation corresponding to the input `a` and `b`.
* *Throws:* Nothing.
Apply a computed Givens rotation to vectors
```c++
template<class ExecutionPolicy,
         class inout_vector_1_t,
         class inout_vector_2_t,
         class Real>
void givens_rotation_apply(
  ExecutionPolicy&& exec,
  inout_vector_1_t v1,
  inout_vector_2_t v2,
  const Real c,
  const Real s);
template<class inout_vector_1_t,
         class inout_vector_2_t,
         class Real>
void givens_rotation_apply(
  inout_vector_1_t v1,
  inout_vector_2_t v2,
  const Real c,
  const Real s);
template<class ExecutionPolicy,
         class inout_vector_1_t,
         class inout_vector_2_t,
         class Real>
void givens_rotation_apply(
  ExecutionPolicy&& exec,
  inout_vector_1_t v1,
  inout_vector_2_t v2,
  const Real c,
  const complex<Real> s);
template<class inout_vector_1_t,
         class inout_vector_2_t,
         class Real>
void givens_rotation_apply(
  inout_vector_1_t v1,
  inout_vector_2_t v2,
  const Real c,
  const complex<Real> s);
```
[Note:
These functions correspond to the BLAS function `xROT`. `c` and `s` form a plane (Givens) rotation. Users normally would compute `c` and `s` using `givens_rotation_setup`, but they are not required to do this.
--*end note]*
* *Requires:*
  * `v1` and `v2` have the same domain.
* *Constraints:*
  * `Real` is `float`, `double`, or `long double`.
  * `v1.rank()` and `v2.rank()` are both one.
  * For the overloads that take the last argument `s` as `Real`, for `i` in the domain of `v1` and `j` in the domain of `v2`, the expressions `v1(i) = c*v1(i) + s*v2(j)` and `v2(j) = -s*v1(i) + c*v2(j)` are well formed.
  * For the overloads that take the last argument `s` as `const complex<Real>`, for `i` in the domain of `v1` and `j` in the domain of `v2`, the expressions `v1(i) = c*v1(i) + s*v2(j)` and `v2(j) = -conj(s)*v1(i) + c*v2(j)` are well formed.
* *Effects:* Applies the plane (Givens) rotation specified by `c` and `s` to the input vectors `v1` and `v2`, as if the rotation were a 2 x 2 matrix and the input vectors were successive rows of a matrix with two rows.
Swap matrix or vector elements
```c++
template<class inout_object_1_t,
```

```
         class inout_object_2_t>
void linalg_swap(inout_object_1_t v1,
                 inout_object_2_t v2);
template<class ExecutionPolicy,
         class inout_object_1_t,
         class inout_object_2_t>
void linalg_swap(ExecutionPolicy&& exec,
                 inout_object_1_t v1,
                 inout_object_2_t v2);
```
[Note: These functions correspond to the BLAS function `xSWAP`.
--*end note]*
* *Requires:* `v1` and `v2` have the same domain.
* *Constraints:*
  * `v1.rank()` equals `v2.rank()`.
  * `v1.rank()` is no more than 3.
  * For `i...` in the domain of `v2` and `v1`, the
    expression `v2(i...) = v1(i...)` is well formed.
* *Effects:* Swap all corresponding elements of the objects
  `v1` and `v2`.
Multiply the elements of an object in place by a scalar
```c++
template<class Scalar,
         class inout_object_t>
void scale(const Scalar alpha,
           inout_object_t obj);
template<class ExecutionPolicy,
         class Scalar,
         class inout_object_t>
void scale(ExecutionPolicy&& exec,
           const Scalar alpha,
           inout_object_t obj);
```
[Note: These functions correspond to the BLAS function `xSCAL`.
--*end note]*
* *Constraints:*
  * `obj.rank()` is no more than 3.
  * For `i...` in the domain of `obj`, the expression
    `obj(i...) *= alpha` is well formed.
* *Effects*: Multiply each element of `obj` in place by `alpha`.
Copy elements of one matrix or vector into another
```c++
template<class in_object_t,
         class out_object_t>
void linalg_copy(in_object_t x,
                 out_object_t y);
template<class ExecutionPolicy,
         class in_object_t,
         class out_object_t>
void linalg_copy(ExecutionPolicy&& exec,
                 in_object_t x,
                 out_object_t y);
```
[Note: These functions correspond to the BLAS function `xCOPY`.
--*end note]*
* *Constraints:*
  * `x.rank()` equals `y.rank()`.
  * `x.rank()` is no more than 3.
  * For all `i...` in the domain of `x` and `y`, the expression
    `y(i...) = x(i...)` is well formed.
* *Requires:* The domain of `y` equals the domain of `x`.
* *Effects:* Overwrite each element of `y` with the corresponding
  element of `x`.
Add vectors or matrices elementwise
```c++
template<class in_object_1_t,
         class in_object_2_t,
         class out_object_t>
void linalg_add(in_object_1_t x,
                in_object_2_t y,
                out_object_t z);
template<class ExecutionPolicy,
         class in_object_1_t,
         class in_object_2_t,
         class out_object_t>
void linalg_add(ExecutionPolicy&& exec,
                in_object_1_t x,
                in_object_2_t y,
                out_object_t z);
```
[Note: These functions correspond to the BLAS function `xAXPY`.

```
--*end note]*
* *Requires:* The domain of `z` equals the domains of `x` and `y`.
* *Constraints:*
  * `x.rank()`, `y.rank()`, and `z.rank()` are all equal.
  * `x.rank()` is no more than 3.
  * For `i...` in the domain of `x`, `y`, and `z`, the expression
    `z(i...) = x(i...) + y(i...)` is well formed.
* *Effects*: Compute the elementwise sum z = x + y.
#### Inner (dot) product of two vectors
##### Non-conjugated inner (dot) product
```c++
template<class in_vector_1_t,
         class in_vector_2_t,
         class Scalar>
void dot(in_vector_1_t v1,
         in_vector_2_t v2,
         Scalar& result);
template<class ExecutionPolicy,
         class in_vector_1_t,
         class in_vector_2_t,
         class Scalar>
void dot(ExecutionPolicy&& exec,
         in_vector_1_t v1,
         in_vector_2_t v2,
         Scalar& result);
```
[Note: These functions correspond to the BLAS functions `xDOT` (for
real element types), `xDOTC`, and `xDOTU` (for complex element types).
--*end note]*
* *Requires:* `v1` and `v2` have the same domain.
* *Constraints:* For all `i` in the domain of `v1` and `v2`,
  the expression `result += v1(i)*v2(i)` is well formed.
* *Effects:* Assigns to `result` the sum of the products of
  corresponding entries of `v1` and `v2`.
* *Remarks:* If `in_vector_t::element_type` and `Scalar` are both
  floating-point types or complex versions thereof, and if `Scalar`
  has higher precision than `in_vector_type::element_type`, then
  implementations will use `Scalar`'s precision or greater for
  intermediate terms in the sum.
[Note: Users can get `xDOTC` behavior by giving the second argument
as a `conjugate_view`. Alternately, they can use the shortcut `dotc`
below. --*end note]*
Conjugated inner (dot) product
```c++
template<class in_vector_1_t,
         class in_vector_2_t,
         class Scalar>
void dotc(in_vector_1_t v1,
          in_vector_2_t v2,
          Scalar& result);
template<class ExecutionPolicy,
         class in_vector_1_t,
         class in_vector_2_t,
         class Scalar>
void dotc(ExecutionPolicy&& exec,
          in_vector_1_t v1,
          in_vector_2_t v2,
          Scalar& result);
```
* *Effects:* Equivalent to `dot(v1, conjugate_view(v2), result);`.
[Note: `dotc` exists to give users reasonable default inner product
behavior for both real and complex element types. --*end note]*
Euclidean (2) norm of a vector
```c++
template<class in_vector_t,
         class Scalar>
void vector_norm2(in_vector_t v,
                  Scalar& result);
template<class ExecutionPolicy,
         class in_vector_t,
         class Scalar>
void vector_norm2(ExecutionPolicy&& exec,
                  in_vector_t v,
                  Scalar& result);
```
[Note: These functions correspond to the BLAS function `xNRM2`.
--*end note]*
* *Constraints:* For all `i` in the domain of `v1` and `v2`, the
  expressions `result += abs(v(i))*abs(v(i))` and `sqrt(result)` are
  well formed. *[Note:* This does not imply a recommended
  implementation for floating-point types. See *Remarks*
```

```
  below. --*end note]*
* *Effects:* Assigns to `result` the Euclidean (2) norm of the
  vector `v`.
* *Remarks:*
  1. If `in_vector_t::element_type` and `Scalar` are both
     floating-point types or complex versions thereof, and if `Scalar`
     has higher precision than `in_vector_type::element_type`, then
     implementations will use `Scalar`'s precision or greater for
     intermediate terms in the sum.
  2. Let `E` be `in_vector_t::element_type`.  If
     * `E` is `float`, `double`, `long double`, `complex<float>`,
       `complex<double>`, or `complex<long double>`;
     * `Scalar` is `E` or larger in the above list of types; and
     * `numeric_limits<E>::is_iec559` is `true`;
     then implementations compute without undue overflow or
     underflow at intermediate stages of the computation.
*[Note:* The intent of the second point of *Remarks* is that
implementations generalize the guarantees of `hypot` regarding
overflow and underflow.  This excludes naÃ¯ve implementations for
floating-point types. --*end note]*
#### Sum of absolute values
```c++
template<class in_vector_t,
         class Scalar>
void vector_abs_sum(in_vector_t v,
                    Scalar& result);
template<class ExecutionPolicy,
         class in_vector_t,
         class Scalar>
void vector_abs_sum(ExecutionPolicy&& exec,
                    in_vector_t v,
                    Scalar& result);
```
[Note: This function corresponds to the BLAS functions `SASUM`,
`DASUM`, `CSASUM`, and `DZASUM`. --*end note]*
* *Constraints:*
  * If `in_vector_t::element_type` is `complex<T>` for some `T`, then
    for all `i` in the domain of `v`, the expression `result +=
    real(v(i)) + imag(v(i))` is well formed.
  * Else, for all `i` in the domain of `v`, the expression
    `result += abs(v(i))` is well formed.
* *Effects:*
  * If `in_vector_t::element_type` is `complex<T>` for some `T`, then
    assigns to `result` the sum of absolute values of the real and
    imaginary components of the elements of the vector `v`.
  * Else, assigns to `result` the sum of absolute values of the
    elements of the vector `v`.
* *Remarks:*
  * If `in_vector_t::element_type` and `Scalar` are both
    floating-point types or complex versions thereof, and if `Scalar`
    has higher precision than `in_vector_type::element_type`, then
    implementations will use `Scalar`'s precision or greater for
    intermediate terms in the sum.
Index of maximum absolute value of vector elements
```c++
template<class in_vector_t>
ptrdiff_t vector_idx_abs_max(in_vector_t v);
template<class ExecutionPolicy,
         class in_vector_t>
ptrdiff_t vector_idx_abs_max(ExecutionPolicy&& exec,
                             in_vector_t v);
```
[Note: These functions correspond to the BLAS function `IxAMAX`.
--*end note]*
* *Constraints:* For `i` and `j` in the domain of `v`, the expression
  `abs(v(i)) < abs(v(j))` is well formed.
* *Returns:* The index (in the domain of `v`) of the first element of
  `v` having largest absolute value. If `v` has zero elements, then
  returns `-1`.
BLAS 2 functions
General matrix-vector product
[Note: These functions correspond to the BLAS function
`xGEMV`. --*end note]*
The following requirements apply to all functions in this section.
* *Requires:*
  * If `i,j` is in the domain of `A`, then `i` is in the domain of `y`
    and `j` is in the domain of `x`.
* *Constraints:* For all functions in this section:
  * `in_matrix_t` has unique layout; and
  * `A.rank()` equals 2, `x.rank()` equals 1, `y.rank()` equals 1, and
    `z.rank()` equals 1.
```

Overwriting matrix-vector product
```c++
template<class ExecutionPolicy,
         class in_vector_t,
         class in_matrix_t,
         class out_vector_t>
void matrix_vector_product(in_matrix_t A,
                           in_vector_t x,
                           out_vector_t y);
template<class ExecutionPolicy, class in_vector_t,
         class in_matrix_t, class out_vector_t>
void matrix_vector_product(ExecutionPolicy&& exec,
                           in_matrix_t A,
                           in_vector_t x,
                           out_vector_t y);
```
* *Constraints:* For `i,j` in the domain of `A`, the expression
  `y(i) += A(i,j)*x(j)` is well formed.
* *Effects:* Assigns to the elements of `y` the product of the matrix
  `A` with the vector `x`.
Updating matrix-vector product
```c++
template<class ExecutionPolicy, class in_vector_1_t,
         class in_matrix_t, class in_vector_2_t,
         class out_vector_t>
void matrix_vector_product(in_matrix_t A,
                           in_vector_1_t x,
                           in_vector_2_t y,
                           out_vector_t z);
template<class ExecutionPolicy, class in_vector_1_t,
         class in_matrix_t, class in_vector_2_t,
         class out_vector_t>
void matrix_vector_product(ExecutionPolicy&& exec,
                           in_matrix_t A,
                           in_vector_1_t x,
                           in_vector_2_t y,
                           out_vector_t z);
```
* *Requires:*
  * `y` and `z` have the same domain.
* *Constraints:*
  * For `i,j` in the domain of `A`, the expression
    `z(i) = y(i) + A(i,j)*x(j)` is well formed.
* *Effects:* Assigns to the elements of `z` the elementwise sum of
  `y`, and the product of the matrix `A` with the vector `x`.
Symmetric matrix-vector product
[Note: These functions correspond to the BLAS functions `xSYMV` and
`xSPMV`. --*end note]*
The following requirements apply to all functions in this section.
* *Requires:*
  * If `i,j` is in the domain of `A`, then `i` is in the domain of `y`
    and `j` is in the domain of `x`.
* *Constraints:*
  * `in_matrix_t` either has unique layout, or `layout_blas_packed`
    layout.
  * If `in_matrix_t` has `layout_blas_packed` layout, then the
    layout's `Triangle` template argument has the same type as
    the function's `Triangle` template argument.
  * `A.rank()` equals 2, `x.rank()` equals 1, `y.rank()` equals 1, and
    `z.rank()` equals 1.
* *Remarks:* The functions will only access the triangle of `A`
  specified by the `Triangle` argument `t`, and will assume for
  indices `i,j` outside that triangle, that `A(j,i)` equals `A(i,j)`.
Overwriting symmetric matrix-vector product
```c++
template<class in_matrix_t,
         class Triangle,
         class in_vector_t,
         class out_vector_t>
void symmetric_matrix_vector_product(in_matrix_t A,
                                     Triangle t,
                                     in_vector_t x,
                                     out_vector_t y);
template<class ExecutionPolicy,
         class in_matrix_t,
         class Triangle,
         class in_vector_t,
         class out_vector_t>
void symmetric_matrix_vector_product(ExecutionPolicy&& exec,
                                     in_matrix_t A,
                                     Triangle t,
```

```
                                      in_vector_t x,
                                      out_vector_t y);
```
* *Constraints:* For `i,j` in the domain of `A`, the expression
  `y(i) += A(i,j)*x(j)` is well formed.
* *Effects:* Assigns to the elements of `y` the product of the matrix
  `A` with the vector `x`.

Updating symmetric matrix-vector product

```c++
template<class in_matrix_t,
         class Triangle,
         class in_vector_1_t,
         class in_vector_2_t,
         class out_vector_t>
void symmetric_matrix_vector_product(
  in_matrix_t A,
  Triangle t,
  in_vector_1_t x,
  in_vector_2_t y,
  out_vector_t z);
template<class ExecutionPolicy,
         class in_matrix_t,
         class Triangle,
         class in_vector_1_t,
         class in_vector_2_t,
         class out_vector_t>
void symmetric_matrix_vector_product(
  ExecutionPolicy&& exec,
  in_matrix_t A,
  Triangle t,
  in_vector_1_t x,
  in_vector_2_t y,
  out_vector_t z);
```
* *Requires:* `y` and `z` have the same domain.
* *Constraints:* For `i,j` in the domain of `A`, the expression
  `z(i) = y(i) + A(i,j)*x(j)` is well formed.
* *Effects:* Assigns to the elements of `z` the elementwise sum of
  `y`, with the product of the matrix `A` with the vector `x`.

Hermitian matrix-vector product

[Note: These functions correspond to the BLAS functions `xHEMV` and
`xHPMV`. --*end note]*

The following requirements apply to all functions in this section.

* *Requires:"*
  * If `i,j` is in the domain of `A`, then `i` is in the domain of `y`
    and `j` is in the domain of `x`.
* *Constraints:*
  * `in_matrix_t` either has unique layout, or `layout_blas_packed`
    layout.
  * If `in_matrix_t` has `layout_blas_packed` layout, then the
    layout's `Triangle` template argument has the same type as
    the function's `Triangle` template argument.
  * `A.rank()` equals 2, `x.rank()` equals 1, `y.rank()` equals 1, and
    `z.rank()` equals 1.
* *Remarks:* The functions will only access the triangle of `A`
  specified by the `Triangle` argument `t`, and will assume for
  indices `i,j` outside that triangle, that `A(j,i)` equals
  `conj(A(i,j))`.

Overwriting Hermitian matrix-vector product

```c++
template<class in_matrix_t,
         class Triangle,
         class in_vector_t,
         class out_vector_t>
void hermitian_matrix_vector_product(in_matrix_t A,
                                     Triangle t,
                                     in_vector_t x,
                                     out_vector_t y);
template<class ExecutionPolicy,
         class in_matrix_t,
         class Triangle,
         class in_vector_t,
         class out_vector_t>
void hermitian_matrix_vector_product(ExecutionPolicy&& exec,
                                     in_matrix_t A,
                                     Triangle t,
                                     in_vector_t x,
                                     out_vector_t y);
```
* *Constraints:* For `i,j` in the domain of `A`, the expressions
  `y(i) += A(i,j)*x(j)` and `y(i) += conj(A(i,j))*x(j)` are well

  formed.
* *Effects:* Assigns to the elements of `y` the product of the matrix
  `A` with the vector `x`.
Updating Hermitian matrix-vector product
```c++
template<class in_matrix_t,
         class Triangle,
         class in_vector_1_t,
         class in_vector_2_t,
         class out_vector_t>
void hermitian_matrix_vector_product(in_matrix_t A,
                                     Triangle t,
                                     in_vector_1_t x,
                                     in_vector_2_t y,
                                     out_vector_t z);
template<class ExecutionPolicy,
         class in_matrix_t,
         class Triangle,
         class in_vector_1_t,
         class in_vector_2_t,
         class out_vector_t>
void hermitian_matrix_vector_product(ExecutionPolicy&& exec,
                                     in_matrix_t A,
                                     Triangle t,
                                     in_vector_1_t x,
                                     in_vector_2_t y,
                                     out_vector_t z);
```
* *Requires:* `y` and `z` have the same domain.
* *Constraints:* For `i,j` in the domain of `A`, the expressions
  `z(i) = y(i) + A(i,j)*x(j)` and `z(i) = y(i) + conj(A(i,j))*x(j)`
  are well formed.
* *Effects:* Assigns to the elements of `z` the elementwise sum of
  `y`, and the product of the matrix `A` with the vector `x`.
Triangular matrix-vector product
[Note: These functions correspond to the BLAS functions `xTRMV` and
`xTPMV`. --*end note]*
The following requirements apply to all functions in this section.
* *Requires:*
  * If `i,j` is in the domain of `A`, then `i` is in the domain of `y`
    and `j` is in the domain of `x`.
* *Constraints:*
  * `in_matrix_t` either has unique layout, or `layout_blas_packed`
    layout.
  * If `in_matrix_t` has `layout_blas_packed` layout, then the
    layout's `Triangle` template argument has the same type as
    the function's `Triangle` template argument.
  * `A.rank()` equals 2, `x.rank()` equals 1, `y.rank()` equals 1, and
    `z.rank()` equals 1.
* *Remarks:*
  * The functions will only access the triangle of `A` specified by
    the `Triangle` argument `t`.
  * If the `DiagonalStorage` template argument has type
    `implicit_unit_diagonal_t`, then the functions will not access the
    diagonal of `A`, and will assume that that the diagonal elements
    of `A` all equal one. *[Note:* This does not imply that the
    function needs to be able to form an `element_type` value equal to
    one. --*end note]
Overwriting triangular matrix-vector product
```c++
template<class in_matrix_t,
         class Triangle,
         class DiagonalStorage,
         class in_vector_t,
         class out_vector_t>
void triangular_matrix_vector_product(
  in_matrix_t A,
  Triangle t,
  DiagonalStorage d,
  in_vector_t x,
  out_vector_t y);
template<class ExecutionPolicy,
         class in_matrix_t,
         class Triangle,
         class DiagonalStorage,
         class in_vector_t,
         class out_vector_t>
void triangular_matrix_vector_product(
  ExecutionPolicy&& exec,
  in_matrix_t A,
  Triangle t,
```

```
  DiagonalStorage d,
  in_vector_t x,
  out_vector_t y);
```
* *Constraints:* For `i,j` in the domain of `A`, the expression
  `y(i) += A(i,j)*x(j)` is well formed.
* *Effects:* Assigns to the elements of `y` the product of the matrix
  `A` with the vector `x`.

Updating triangular matrix-vector product

```c++
template<class in_matrix_t,
         class Triangle,
         class DiagonalStorage,
         class in_vector_1_t,
         class in_vector_2_t,
         class out_vector_t>
void triangular_matrix_vector_product(in_matrix_t A,
                                      Triangle t,
                                      DiagonalStorage d,
                                      in_vector_1_t x,
                                      in_vector_2_t y,
                                      out_vector_t z);
template<class ExecutionPolicy,
         class in_matrix_t,
         class Triangle,
         class DiagonalStorage,
         class in_vector_1_t,
         class in_vector_2_t,
         class out_vector_t>
void triangular_matrix_vector_product(ExecutionPolicy&& exec,
                                      in_matrix_t A,
                                      Triangle t,
                                      DiagonalStorage d,
                                      in_vector_1_t x,
                                      in_vector_2_t y,
                                      out_vector_t z);
```
* *Requires:* `y` and `z` have the same domain.
* *Constraints:* For `i,j` in the domain of `A`, the expression
  `z(i) = y(i) + A(i,j)*x(j)` is well formed.
* *Effects:* Assigns to the elements of `z` the elementwise sum of
  `y`, with the product of the matrix `A` with the vector `x`.

Solve a triangular linear system

```c++
template<class in_matrix_t,
         class Triangle,
         class DiagonalStorage,
         class in_object_t,
         class out_object_t>
void triangular_matrix_vector_solve(
  in_matrix_t A,
  Triangle t,
  DiagonalStorage d,
  in_object_t b,
  out_object_t x);
template<class ExecutionPolicy,
         class in_matrix_t,
         class Triangle,
         class DiagonalStorage,
         class in_object_t,
         class out_object_t>
void triangular_matrix_vector_solve(
  ExecutionPolicy&& exec,
  in_matrix_t A,
  Triangle t,
  DiagonalStorage d,
  in_object_t b,
  out_object_t x);
```
[Note: These functions correspond to the BLAS functions `xTRSV` and `xTPSV`. --*end note]*
* *Requires:*
  * If `i,j` is in the domain of `A`, then `i` is in the domain of `x`
    and `j` is in the domain of `b`.
* *Constraints:*
  * `A.rank()` equals 2.
  * `b.rank()` equals 1 and `x.rank()` equals 1.
  * `in_matrix_t` either has unique layout, or `layout_blas_packed`
    layout.
  * If `in_matrix_t` has `layout_blas_packed` layout, then the
    layout's `Triangle` template argument has the same type as

```
    the function's `Triangle` template argument.
  * If `r` is in the domain of `x` and `b`, then the expression
    `x(r) = y(r)` is well formed.
  * If `r` is in the domain of `x`, then the expression `x(r) -=
    A(r,c)*x(c)` is well formed.
  * If `r` is in the domain of `x` and `DiagonalStorage` is
    `explicit_diagonal_t`, then the expression `x(r) /= A(r,r)` is
    well formed.
* *Effects:* Assigns to the elements of `x` the result of solving the
  triangular linear system(s) Ax=b.
* *Remarks:*
  * The functions will only access the triangle of `A` specified by
    the `Triangle` argument `t`.
  * If the `DiagonalStorage` template argument has type
    `implicit_unit_diagonal_t`, then the functions will not access the
    diagonal of `A`, and will assume that that the diagonal elements
    of `A` all equal one. *[Note:* This does not imply that the
    function needs to be able to form an `element_type` value equal to
    one. --*end note]
#### Rank-1 (outer product) update of a matrix
##### Nonsymmetric non-conjugated rank-1 update
```c++
template<class in_vector_1_t,
         class in_vector_2_t,
         class inout_matrix_t>
void matrix_rank_1_update(
  in_vector_1_t x,
  in_vector_2_t y,
  inout_matrix_t A);
template<class ExecutionPolicy,
         class in_vector_1_t,
         class in_vector_2_t,
         class inout_matrix_t>
void matrix_rank_1_update(
  ExecutionPolicy&& exec,
  in_vector_1_t x,
  in_vector_2_t y,
  inout_matrix_t A);
```
[Note: This function corresponds to the BLAS functions `xGER` (for
real element types), `xGERC`, and `xGERU` (for complex element
types). --*end note]*
* *Requires:*
  * If `i,j` is in the domain of `A`, then `i` is in the domain of `x`
    and `j` is in the domain of `y`.
* *Constraints:*
  * `A.rank()` equals 2, `x.rank()` equals 1, and `y.rank()` equals 1.
  * For `i,j` in the domain of `A`, the expression
    `A(i,j) += x(i)*y(j)` is well formed.
* *Effects:* Assigns to `A` on output the sum of `A` on input, and the
  outer product of `x` and `y`.
[Note: Users can get `xGERC` behavior by giving the second argument
as a `conjugate_view`. Alternately, they can use the shortcut
`matrix_rank_1_update_c` below. --*end note]*
Nonsymmetric conjugated rank-1 update
```c++
template<class in_vector_1_t,
         class in_vector_2_t,
         class inout_matrix_t>
void matrix_rank_1_update_c(
  in_vector_1_t x,
  in_vector_2_t y,
  inout_matrix_t A);
template<class ExecutionPolicy,
         class in_vector_1_t,
         class in_vector_2_t,
         class inout_matrix_t>
void matrix_rank_1_update_c(
  ExecutionPolicy&& exec,
  in_vector_1_t x,
  in_vector_2_t y,
  inout_matrix_t A);
```
* *Effects:* Equivalent to
  `matrix_rank_1_update(x, conjugate_view(y), A);`.
Rank-1 update of a Symmetric matrix
```c++
template<class in_vector_t,
         class inout_matrix_t,
         class Triangle>
void symmetric_matrix_rank_1_update(
```
```

```
  in_vector_t x,
  inout_matrix_t A,
  Triangle t);
template<class ExecutionPolicy,
         class in_vector_t,
         class inout_matrix_t,
         class Triangle>
void symmetric_matrix_rank_1_update(
  ExecutionPolicy&& exec,
  in_vector_t x,
  inout_matrix_t A,
  Triangle t);
```
[Note: These functions correspond to the BLAS functions `xSYR` and `xSPR`. --*end note]*
* *Requires:*
  * If `i,j` is in the domain of `A`, then `i` and `j` are in the domain of `x`.
* *Constraints:*
  * `A.rank()` equals 2 and `x.rank()` equals 1.
  * `A` either has unique layout, or `layout_blas_packed` layout.
  * If `A` has `layout_blas_packed` layout, then the layout's `Triangle` template argument has the same type as the function's `Triangle` template argument.
  * For `i,j` in the domain of `A`, the expression `A(i,j) += x(i)*x(j)` is well formed.
* *Effects:* Assigns to `A` on output the sum of `A` on input, and the outer product of `x` and `x`.
* *Remarks:* The functions will only access the triangle of `A` specified by the `Triangle` argument `t`, and will assume for indices `i,j` outside that triangle, that `A(j,i)` equals `A(i,j)`.

Rank-1 update of a Hermitian matrix
```c++
template<class in_vector_t,
         class inout_matrix_t,
         class Triangle>
void hermitian_matrix_rank_1_update(
  in_vector_t x,
  inout_matrix_t A,
  Triangle t);
template<class ExecutionPolicy,
         class in_vector_t,
         class inout_matrix_t,
         class Triangle>
void hermitian_matrix_rank_1_update(
  ExecutionPolicy&& exec,
  in_vector_t x,
  inout_matrix_t A,
  Triangle t);
```
[Note: These functions correspond to the BLAS functions `xHER` and `xHPR`. --*end note]*
* *Requires:*
  * If `i,j` is in the domain of `A`, then `i` and `j` are in the domain of `x`.
* *Constraints:*
  * `A.rank()` equals 2 and `x.rank()` equals 1.
  * `A` either has unique layout, or `layout_blas_packed` layout.
  * If `A` has `layout_blas_packed` layout, then the layout's `Triangle` template argument has the same type as the function's `Triangle` template argument.
  * For `i,j` in the domain of `A`, the expression `A(i,j) += x(i)*conj(x(j))` is well formed.
* *Effects:* Assigns to `A` on output the sum of `A` on input, and the outer product of `x` and the conjugate of `x`.
* *Remarks:* The functions will only access the triangle of `A` specified by the `Triangle` argument `t`, and will assume for indices `i,j` outside that triangle, that `A(j,i)` equals `conj(A(i,j))`.

Rank-2 update of a symmetric matrix
```c++
template<class in_vector_1_t,
         class in_vector_2_t,
         class inout_matrix_t,
         class Triangle>
void symmetric_matrix_rank_2_update(
  in_vector_1_t x,
  in_vector_2_t y,
  inout_matrix_t A,
  Triangle t);
template<class ExecutionPolicy,
```

```
          class in_vector_1_t,
          class in_vector_2_t,
          class inout_matrix_t,
          class Triangle>
void symmetric_matrix_rank_2_update(
  ExecutionPolicy&& exec,
  in_vector_1_t x,
  in_vector_2_t y,
  inout_matrix_t A,
  Triangle t);
```
[Note: These functions correspond to the BLAS functions `xSYR2` and
`xSPR2`. --*end note]*
* *Requires:*
  * If `i,j` is in the domain of `A`, then `i` and `j` are in the
    domain of `x` and `y`.
* *Constraints:*
  * `A.rank()` equals 2, `x.rank()` equals 1, and
    `y.rank()` equals 1.
  * `A` either has unique layout, or `layout_blas_packed` layout.
  * If `A` has `layout_blas_packed` layout, then the layout's
    `Triangle` template argument has the same type as the function's
    `Triangle` template argument.
  * For `i,j` in the domain of `A`, the expression
    `A(i,j) += x(i)*y(j) + y(i)*x(j)` is well formed.
* *Effects:* Assigns to `A` on output the sum of `A` on input, the
  outer product of `x` and `y`, and the outer product of `y` and `x`.
* *Remarks:* The functions will only access the triangle of `A`
  specified by the `Triangle` argument `t`, and will assume for
  indices `i,j` outside that triangle, that `A(j,i)` equals `A(i,j)`.
Rank-2 update of a Hermitian matrix
```c++
template<class in_vector_1_t,
          class in_vector_2_t,
          class inout_matrix_t,
          class Triangle>
void hermitian_matrix_rank_2_update(
  in_vector_1_t x,
  in_vector_2_t y,
  inout_matrix_t A,
  Triangle t);
template<class ExecutionPolicy,
          class in_vector_1_t,
          class in_vector_2_t,
          class inout_matrix_t,
          class Triangle>
void hermitian_matrix_rank_2_update(
  ExecutionPolicy&& exec,
  in_vector_1_t x,
  in_vector_2_t y,
  inout_matrix_t A,
  Triangle t);
```
[Note: These functions correspond to the BLAS functions `xHER2` and
`xHPR2`. --*end note]*
* *Requires:*
  * If `i,j` is in the domain of `A`, then `i` and `j` are in the
    domain of `x` and `y`.
* *Constraints:*
  * `A.rank()` equals 2, `x.rank()` equals 1, and
    `y.rank()` equals 1.
  * `A` either has unique layout, or `layout_blas_packed` layout.
  * If `A` has `layout_blas_packed` layout, then the layout's
    `Triangle` template argument has the same type as the function's
    `Triangle` template argument.
  * For `i,j` in the domain of `A`, the expression `A(i,j) +=
    x(i)*conj(y(j)) + y(i)*conj(x(j))` is well formed.
* *Effects:* Assigns to `A` on output the sum of `A` on input, the
  outer product of `x` and the conjugate of `y`, and the outer product
  of `y` and the conjugate of `x`.
* *Remarks:* The functions will only access the triangle of `A`
  specified by the `Triangle` argument `t`, and will assume for
  indices `i,j` outside that triangle, that `A(j,i)` equals
  `conj(A(i,j))`.
BLAS 3 functions
General matrix-matrix product
[Note: These functions correspond to the BLAS function `xGEMM`.
--*end note]*
The following requirements apply to all functions in this section.
* *Requires:* If `i,j` is in the domain of `C`, then there exists `k`
  such that `i,k` is in the domain of `A`, and `k,j` is in the domain

  of `B`.
* *Constraints:*
  * `in_matrix_1_t`, `in_matrix_2_t`, `in_matrix_3_t` (if applicable),
    and `out_matrix_t` have unique layout.
  * `A.rank()` equals 2, `B.rank()` equals 2, `C.rank()` equals 2, and
    `E.rank()` (if applicable) equals 2.
Overwriting general matrix-matrix product
```c++
template<class in_matrix_1_t,
         class in_matrix_2_t,
         class out_matrix_t>
void matrix_product(in_matrix_1_t A,
                    in_matrix_2_t B,
                    out_matrix_t C);
template<class ExecutionPolicy,
         class in_matrix_1_t,
         class in_matrix_2_t,
         class out_matrix_t>
void matrix_product(ExecutionPolicy&& exec,
                    in_matrix_1_t A,
                    in_matrix_2_t B,
                    out_matrix_t C);
```
* *Constraints:*
  * For `i,j` in the domain of `C`, `i,k` in the domain of `A`, and
    `k,j` in the domain of `B`, the expression `C(i,j) +=
    A(i,k)*B(k,j)` is well formed.
* *Effects:* Assigns to the elements of the matrix `C` the product of
  the matrices `A` and `B`.
Updating general matrix-matrix product
```c++
template<class in_matrix_1_t,
         class in_matrix_2_t,
         class in_matrix_3_t,
         class out_matrix_t>
void matrix_product(in_matrix_1_t A,
                    in_matrix_2_t B,
                    in_matrix_3_t E,
                    out_matrix_t C);
template<class ExecutionPolicy,
         class in_matrix_1_t,
         class in_matrix_2_t,
         class in_matrix_3_t,
         class out_matrix_t>
void matrix_product(ExecutionPolicy&& exec,
                    in_matrix_1_t A,
                    in_matrix_2_t B,
                    in_matrix_3_t E,
                    out_matrix_t C);
```
* *Requires:*
  * `C` and `E` have the same domain.
* *Constraints:* For `i,j` in the domain of `C`, `i,k` in the domain
  of `A`, and `k,j` in the domain of `B`, the expression `C(i,j) +=
  E(i,j) + A(i,k)*B(k,j)` is well formed.
* *Effects:* Assigns to the elements of the matrix `C` on output, the
  elementwise sum of `E` and the product of the matrices `A` and `B`.
* *Remarks:* `C` and `E` may refer to the same matrix. If so, then
  they must have the same layout.
Symmetric matrix-matrix product
[Note: These functions correspond to the BLAS function `xSYMM`.
Unlike the symmetric rank-1 update functions, these functions assume
that the input matrix -- not the output matrix -- is symmetric. --*end
note]*
The following requirements apply to all functions in this section.
* *Requires:* If `i,j` is in the domain of `C`, then there exists `k`
  such that `i,k` is in the domain of `A`, and `k,j` is in the domain
  of `B`.
* *Constraints:*
  * `in_matrix_1_t` either has unique layout, or `layout_blas_packed`
    layout.
  * `in_matrix_2_t`, `in_matrix_3_t` (if applicable), and
    `out_matrix_t` have unique layout.
  * If `in_matrix_t` has `layout_blas_packed` layout, then the
    layout's `Triangle` template argument has the same type as
    the function's `Triangle` template argument.
  * `A.rank()` equals 2, `B.rank()` equals 2, `C.rank()` equals 2, and
    `E.rank()` (if applicable) equals 2.
* *Remarks:* The functions will only access the triangle of `A`
  specified by the `Triangle` argument `t`, and will assume for
  indices `i,j` outside that triangle, that `A(j,i)` equals `A(i,j)`.

Overwriting symmetric matrix-matrix product

```c++
template<class in_matrix_1_t,
         class Triangle,
         class Side,
         class in_matrix_2_t,
         class out_matrix_t>
void symmetric_matrix_product(
  in_matrix_1_t A,
  Triangle t,
  Side s,
  in_matrix_2_t B,
  out_matrix_t C);
template<class ExecutionPolicy,
         class in_matrix_1_t,
         class Triangle,
         class Side,
         class in_matrix_2_t,
         class out_matrix_t>
void symmetric_matrix_product(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
  Side s,
  in_matrix_2_t B,
  out_matrix_t C);
```

* *Constraints:*
  * If `Side` is `left_side_t`, then for `i,j` in the domain of `C`, `i,k` in the domain of `A`, and `k,j` in the domain of `B`, the expression `C(i,j) += A(i,k)*B(k,j)` is well formed.
  * If `Side` is `right_side_t`, then for `i,j` in the domain of `C`, `i,k` in the domain of `B`, and `k,j` in the domain of `A`, the expression `C(i,j) += B(i,k)*A(k,j)` is well formed.
* *Effects:*
  * If `Side` is `left_side_t`, then assigns to the elements of the matrix `C` the product of the matrices `A` and `B`.
  * If `Side` is `right_side_t`, then assigns to the elements of the matrix `C` the product of the matrices `B` and `A`.

Updating symmetric matrix-matrix product

```c++
template<class in_matrix_1_t,
         class Triangle,
         class Side,
         class in_matrix_2_t,
         class in_matrix_3_t,
         class out_matrix_t>
void symmetric_matrix_product(
  in_matrix_1_t A,
  Triangle t,
  Side s,
  in_matrix_2_t B,
  in_matrix_3_t E,
  out_matrix_t C);
template<class ExecutionPolicy,
         class in_matrix_1_t,
         class Triangle,
         class Side,
         class in_matrix_2_t,
         class in_matrix_3_t,
         class out_matrix_t>
void symmetric_matrix_product(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
  Side s,
  in_matrix_2_t B,
  in_matrix_3_t E,
  out_matrix_t C);
```

* *Requires:*
  * `C` and `E` have the same domain.
* *Constraints:*
  * If `Side` is `left_side_t`, then for `i,j` in the domain of `C`, `i,k` in the domain of `A`, and `k,j` in the domain of `B`, the expression `C(i,j) += E(i,j) + A(i,k)*B(k,j)` is well formed.
  * If `Side` is `right_side_t`, then for `i,j` in the domain of `C`, `i,k` in the domain of `B`, and `k,j` in the domain of `A`, the expression `C(i,j) += E(i,j) + B(i,k)*A(k,j)` is well formed.
* *Effects:*
  * If `Side` is `left_side_t`, then assigns to the elements of the

```
    matrix `C` on output, the elementwise sum of `E` and the product of
    the matrices `A` and `B`.
  * If `Side` is `right_side_t`, then assigns to the elements of the
    matrix `C` on output, the elementwise sum of `E` and the product of
    the matrices `B` and `A`.
* *Remarks:* `C` and `E` may refer to the same matrix.  If so, then
  they must have the same layout.
#### Hermitian matrix-matrix product
*[Note:* These functions correspond to the BLAS function `xHEMM`.
Unlike the Hermitian rank-1 update functions, these functions assume
that the input matrix -- not the output matrix -- is Hermitian. --*end
note]*
The following requirements apply to all functions in this section.
* *Requires:* If `i,j` is in the domain of `C`, then there exists `k`
  such that `i,k` is in the domain of `A`, and `k,j` is in the domain
  of `B`.
* *Constraints:*
  * `in_matrix_1_t` either has unique layout, or `layout_blas_packed`
    layout.
  * `in_matrix_2_t`, `in_matrix_3_t` (if applicable), and
    `out_matrix_t` have unique layout.
  * If `in_matrix_t` has `layout_blas_packed` layout, then the
    layout's `Triangle` template argument has the same type as
    the function's `Triangle` template argument.
  * `A.rank()` equals 2, `B.rank()` equals 2, `C.rank()` equals 2, and
    `E.rank()` (if applicable) equals 2.
* *Remarks:* The functions will only access the triangle of `A`
  specified by the `Triangle` argument `t`, and will assume for
  indices `i,j` outside that triangle, that `A(j,i)` equals
  `conj(A(i,j))`.
##### Overwriting Hermitian matrix-matrix product
```c++
template<class in_matrix_1_t,
         class Triangle,
         class Side,
         class in_matrix_2_t,
         class out_matrix_t>
void hermitian_matrix_product(
  in_matrix_1_t A,
  Triangle t,
  Side s,
  in_matrix_2_t B,
  out_matrix_t C);
template<class ExecutionPolicy,
         class in_matrix_1_t,
         class Triangle,
         class Side,
         class in_matrix_2_t,
         class out_matrix_t>
void hermitian_matrix_product(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
  Side s,
  in_matrix_2_t B,
  out_matrix_t C);
```
* *Constraints:*
  * If `Side` is `left_side_t`, then for `i,j` in the domain of `C`,
    `i,k` in the domain of `A`, and `k,j` in the domain of `B`, the
    expression `C(i,j) += A(i,k)*B(k,j)` is well formed.
  * If `Side` is `right_side_t`, then for `i,j` in the domain of `C`,
    `i,k` in the domain of `B`, and `k,j` in the domain of `A`, the
    expression `C(i,j) += B(i,k)*A(k,j)` is well formed.
* *Effects:*
  * If `Side` is `left_side_t`, then assigns to the elements of the
    matrix `C` the product of the matrices `A` and `B`.
  * If `Side` is `right_side_t`, then assigns to the elements of the
    matrix `C` the product of the matrices `B` and `A`.
Updating Hermitian matrix-matrix product
```c++
template<class in_matrix_1_t,
         class Triangle,
         class Side,
         class in_matrix_2_t,
         class in_matrix_3_t,
         class out_matrix_t>
void hermitian_matrix_product(
  in_matrix_1_t A,
  Triangle t,
  Side s,
```
```

```
  in_matrix_2_t B,
  in_matrix_3_t E,
  out_matrix_t C);
template<class ExecutionPolicy,
         class in_matrix_1_t,
         class Triangle,
         class Side,
         class in_matrix_2_t,
         class in_matrix_3_t,
         class out_matrix_t>
void hermitian_matrix_product(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
  Side s,
  in_matrix_2_t B,
  in_matrix_3_t E,
  out_matrix_t C);
```
* *Requires:*
  * `C` and `E` have the same domain.
* *Constraints:*
  * If `Side` is `left_side_t`, then for `i,j` in the domain of `C`, `i,k` in the domain of `A`, and `k,j` in the domain of `B`, the expression `C(i,j) += E(i,j) + A(i,k)*B(k,j)` is well formed.
  * If `Side` is `right_side_t`, then for `i,j` in the domain of `C`, `i,k` in the domain of `B`, and `k,j` in the domain of `A`, the expression `C(i,j) += E(i,j) + B(i,k)*A(k,j)` is well formed.
* *Effects:*
  * If `Side` is `left_side_t`, then assigns to the elements of the matrix `C` on output, the elementwise sum of `E` and the product of the matrices `A` and `B`.
  * If `Side` is `right_side_t`, then assigns to the elements of the matrix `C` on output, the elementwise sum of `E` and the product of the matrices `B` and `A`.
* *Remarks:* `C` and `E` may refer to the same matrix. If so, then they must have the same layout.

Rank-2k update of a symmetric or Hermitian matrix

[Note: Users can achieve the effect of the `TRANS` argument of these BLAS functions, by making `C` a `transpose_view` or `conjugate_transpose_view`. --*end note]*

Rank-2k update of a symmetric matrix

```c++
template<class in_matrix_1_t,
         class in_matrix_2_t,
         class inout_matrix_t,
         class Triangle>
void symmetric_matrix_rank_2k_update(
  in_matrix_1_t A,
  in_matrix_2_t B,
  inout_matrix_t C,
  Triangle t);
template<class ExecutionPolicy,
         class in_matrix_1_t,
         class in_matrix_2_t,
         class inout_matrix_t,
         class Triangle>
void symmetric_matrix_rank_2k_update(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  in_matrix_2_t B,
  inout_matrix_t C,
  Triangle t);
```

[Note: These functions correspond to the BLAS function `xSYR2K`. The BLAS "quick reference" has a typo; the "ALPHA" argument of `CSYR2K` and `ZSYR2K` should not be conjugated. --*end note]*

* *Requires:*
  * If `i,j` is in the domain of `C`, then there exists `k` such that `i,k` and `j,k` are in the domain of `A`, and `j,k` and `i,k` are in the domain of `B`.
* *Constraints:*
  * `A.rank()` equals 2, `B.rank()` equals 2, and `C.rank()` equals 2.
  * `C` either has unique layout, or `layout_blas_packed` layout.
  * If `C` has `layout_blas_packed` layout, then the layout's `Triangle` template argument has the same type as the function's `Triangle` template argument.
  * For `i,j` in the domain of `C`, `i,k` and `k,i` in the domain of `A`, and `j,k` and `k,j` in the domain of `B`, the expression `C(i,j) += A(i,k)*B(j,k) + B(i,k)*A(j,k)` is well formed.

```
* *Effects:* Assigns to `C` on output, the elementwise sum of `C` on
  input with (the matrix product of `A` and the non-conjugated
  transpose of `B`) and (the matrix product of `B` and the
  non-conjugated transpose of `A`.)
* *Remarks:* The functions will only access the triangle of `C`
  specified by the `Triangle` argument `t`, and will assume for
  indices `i,j` outside that triangle, that `C(j,i)` equals `C(i,j)`.
##### Rank-2k update of a Hermitian matrix
```c++
template<class in_matrix_1_t,
         class in_matrix_2_t,
         class inout_matrix_t,
         class Triangle>
void hermitian_matrix_rank_2k_update(
  in_matrix_1_t A,
  in_matrix_2_t B,
  inout_matrix_t C,
  Triangle t);
template<class ExecutionPolicy,
         class in_matrix_1_t,
         class in_matrix_2_t,
         class inout_matrix_t,
         class Triangle>
void hermitian_matrix_rank_2k_update(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  in_matrix_2_t B,
  inout_matrix_t C,
  Triangle t);
```
[Note: These functions correspond to the BLAS function `xHER2K`.
--*end note]*
* *Requires:*
  * If `i,j` is in the domain of `C`, then there exists `k` such that
    `i,k` and `j,k` are in the domain of `A`, and `j,k` and `i,k` are
    in the domain of `B`.
* *Constraints:*
  * `A.rank()` equals 2, `B.rank()` equals 2, and
    `C.rank()` equals 2.
  * `C` either has unique layout, or `layout_blas_packed` layout.
  * If `C` has `layout_blas_packed` layout, then the layout's
    `Triangle` template argument has the same type as the function's
    `Triangle` template argument.
  * For `i,j` in the domain of `C`, `i,k` and `k,i` in the domain of
    `A`, and `j,k` and `k,j` in the domain of `B`, the expression
    `C(i,j) += A(i,k)*conj(B(j,k)) + B(i,k)*conj(A(j,k))` is well
    formed.
* *Effects:* Assigns to `C` on output, the elementwise sum of `C` on
  input with (the matrix product of `A` and the conjugate transpose of
  `B`) and (the matrix product of `B` and the conjugate transpose of
  `A`.)
* *Remarks:* The functions will only access the triangle of `C`
  specified by the `Triangle` argument `t`, and will assume for
  indices `i,j` outside that triangle, that `C(j,i)` equals
  `conj(C(i,j))`.
Solve multiple triangular linear systems with the same matrix
```c++
template<class in_matrix_t,
         class Triangle,
         class DiagonalStorage,
         class Side,
         class in_matrix_t,
         class out_matrix_t>
void triangular_matrix_matrix_solve(
  in_matrix_t A,
  Triangle t,
  DiagonalStorage d,
  Side s,
  in_object_t B,
  out_object_t X);
template<class ExecutionPolicy,
         class in_matrix_1_t,
         class Triangle,
         class DiagonalStorage,
         class Side,
         class in_matrix_2_t,
         class out_matrix_t>
void triangular_matrix_matrix_solve(
  ExecutionPolicy&& exec,
  in_matrix_t A,
  Triangle t,
```
```

```
  DiagonalStorage d,
  Side s,
  in_object_t B,
  out_object_t X);
```
[Note: These functions correspond to the BLAS function `xTRSM`. The Reference BLAS does not have a `xTPSM` function. --*end note]*
* *Requires:*
  * `X.extent(1)` equals `B.extent(1)`.
  * If `X.extent(1) != 0` and `i,j` is in the domain of `A`, then there exists `k` such that `i,k` is in the domain of `X` and `j,k` is in the domain of `B`.
* *Constraints:*
  * `A.rank()` equals 2, `B.rank()` equals 2, and `X.rank()` equals 2.
  * `in_matrix_1_t` either has unique layout, or `layout_blas_packed` layout.
  * `in_matrix_2_t` and `out_matrix_t` have unique layout.
  * If `r,j` is in the domain of `X` and `B`, then the expression `X(r,j) = B(r,j)` is well formed.
  * If `r,j` and `c,j` are in the domain of `X`, then the expression `X(r,j) -= A(r,c)*X(c,j)` is well formed.
  * If `r,j` is in the domain of `X` and `DiagonalStorage` is `explicit_diagonal_t`, then the expression `X(r,j) /= A(r,r)` is well formed.
* *Effects:*
  * If `Side` is `left_side_t`, then assigns to the elements of `X` the result of solving the triangular linear system(s) AX=B for X.
  * If `Side` is `right_side_t`, then assigns to the elements of `X` the result of solving the triangular linear system(s) XA=B for X.
* *Remarks:*
  * The functions will only access the triangle of `A` specified by the `Triangle` argument `t`.
  * If the `DiagonalStorage` template argument has type `implicit_unit_diagonal_t`, then the functions will not access the diagonal of `A`, and will assume that that the diagonal elements of `A` all equal one. *[Note:* This does not imply that the function needs to be able to form an `element_type` value equal to one. --*end note]

Examples

```c++
using vector_t = basic_mdspan<double, extents<dynamic_extent>>;
using dy_ext2_t = extents<dynamic_extent, dynamic_extent>;
using matrix_t = basic_mdspan<double, dy_ext2_t>;
// Create vectors
vector_t x = ...;
vector_t y = ...;
vector_t z = ...;
// Create matrices
matrix_t A = ...;
matrix_t B = ...;
matrix_t C = ...;
// z = 2.0 * x + y;
linalg_add(par, scaled_view(2.0, x), y, z);
// y = 2.0 * y + z;
linalg_add(par, z, scaled_view(2.0, y), y);
// y = 3.0 * A * x;
matrix_vector_product(par, scaled_view(3.0, A), x, y);
// y = 3.0 * A * x + 2.0 * y;
matrix_vector_product(par, scaled_view(3.0, A), x,
                      scaled_view(2.0, y), y);
// y = transpose(A) * x;
matrix_vector_product(par, transpose_view(A), x, y);
```

Batched BLAS

This proposal has an optional extension to support batched operations. Functions that take matrices and/or vectors would simply be overloaded to take arguments with one higher rank. The leftmost dimension of each `basic_mdspan` or `basic_mdarray` would refer to a specific matrix or vector in the "batch." A nonunique "broadcast" layout could also be used to use the same lower-rank object in the operation for each of the batched operations. Otherwise, the `extent(0)` of each `basic_mdspan` or `basic_mdarray` argument must be equal.

Options and votes

This is a preliminary proposal. Besides the usual bikeshedding, we also want to present more broad options for voting. Here is a list; we will explain each option below.

1. Omit vector-vector operations in favor of existing C++ Standard algorithms?
2. Retain "view" functions (modest expression templates)?
3. Combine functions that differ only by rank of arguments?
4. Prefer overloads to different function names?

5. Retain existing BLAS behavior for scalar multipliers?
Omit vector-vector operations in favor of existing C++ Standard algorithms?
Annex C of the BLAS Standard offers a "Thin BLAS" option for Fortran 95, where the language itself could replace many BLAS operations. Fortran 95 comes with dot products and other vector operations built in, so the "Thin BLAS" only retains four "BLAS 1" functions: `SWAP`, `ROT`, `NRM2`, and `ROTG`. By analogy with the "Thin BLAS," we could reduce the number of new functions, by relying on functionality either already in C++, or likely to enter C++ soon. For example, if we defined iterators for rank-1 `basic_mdspan` and `basic_mdarray`, we could rely on `transform` and `transform_reduce` for most of the vector-vector operations.
Matrix-vector ("BLAS 2") and matrix-matrix ("BLAS 3") operations require iteration over two or three dimensions, and are thus less natural to implement using `transform` or `transform_reduce`. They are also more likely to get performance benefits from specialized implementations.
Here are arguments _for_ this approach:
1. It reduces the number of new functions.
2. It focuses on "performance primitives" most likely to benefit from vendor optimization.
3. If a hypothetical "parallel Ranges" enters the Standard, it could cover many of the use cases for parallel vector-vector operations.

Here are arguments _against_ this approach:
1. It takes some effort to implement correct and accurate vector norms. Compare to [POSIX requirements for

`hypot`](http://pubs.opengroup.org/onlinepubs/9699919799/functions/hypot.html)
.
   If `hypot` is in the Standard, then perhaps norms should also be.
2. In general, a linear algebra library can make more specific statements about the precision at which output arguments are computed.
3. Some of our "vector-vector" operations are actually "object-object" operations that work for matrices too. Replacing those with existing Standard algorithms would call for iterators on matrices.
4. It's easier to apply hardware-specific optimizations to vector-vector operations if they are exposed as such.
5. Exposing a full linear algebra interface would give implementers the option to use extended-precision or even reproducible floating-point arithmetic for all linear algebra operations. This can be useful for debugging complicated algorithms. Compare to "checked iterator" debug options for the C++ Standard Library.
6. It helps to have linear algebra names for linear algebra operations. For example, `string` still exists, even though much of its functionality is covered by `vector<char>`.

Our preference:
* We would prefer a complete BLAS-like library, but if we had to give up some BLAS 1 functions, we would prefer to keep at least the vector norms.
* We think that iterators are not always the right way to access multidimensional objects.

Retain "view" functions (modest expression templates)?
The four functions `scaled_view`, `conjugate_view`, `transpose_view`, and `conjugate_transpose_view` use `mdspan` accessors to implement a modest form of expression templates. We say "modest" because they mitigate several known issues with expression templates:
1. They do not introduce ADL-accessible arithmetic operators on matrices or vectors.
2. If used with `mdspan`, then they would not introduce any more dangling references than `span` (**[views.span]**) would introduce.
3. Their intended use case, as temporary "decorators" for function arguments, discourages capture as `auto` (which may result in unexpectedly dangling references).

The functions have the following other _advantages_:
1. They reduce the number of linear algebra function arguments.
2. They simplify native C++ implementations, especially for BLAS 1 and BLAS 2 functions that do not need complicated optimizations in order to get reasonable performance.

However, the functions have the following _disadvantages_:
1. They would be the first instance of required expression templates in the C++ Standard Library. (`valarray` in **[valarray.syn]** permits but does not require expression templates.)
2. When applied to a `basic_mdarray`, the functions could introduce dangling references. Compare to `gslice_array` for `valarray`.
3. If users can "tag" a matrix with a scaling factor or the transpose property, why can't they "tag" the matrix with other properties, like symmetry? That suggests a design in which function parameters are generic "things," convertible to `basic_mdspan` or `basic_mdarray`, with properties (in the sense of

[P0939R0](http://wg21.link/p0939r0)) that the function can query.

Here are the options:

1. Keep existing "view" functions.
2. Add a general property tagging mechanism, so users can tag a matrix with mathematical properties like "symmetric," "Hermitian," or "triangular." Use this mechanism to pass assumptions into functions, and eliminate `symmetric_*`, `hermitian_*`, and (in some cases) `triangular_*` versions of functions.
3. Drop "view" functions. Specify scaling, transpose, and conjugation as function parameters.

Our preference: Option 1 (retain existing "view" functions).

Option 2 is interesting but would add a lot of complication. Would we let users customize properties? Algorithms could never be made generic on arbitrary mathematical properties of matrices. This is also closer to the high-level interface -- "Matlab in C++" -- that is _not_ our target for this proposal. In addition, Option 2 would generalize well beyond what the BLAS does. For example, the BLAS' `xSYMM` (symmetric matrix-matrix multiply) only specifies that one of the input matrices is symmetric.

We would prefer Option 3 over Option 2.

Combine functions that differ only by rank of arguments?

This relates to the "thin BLAS" proposal mentioned above. Another part of that proposal was the elimination of separate function names, when (the Fortran 95 equivalent of) overloads could express the same idea. There are two parts to this:

1. Combine functions that differ only by rank of arguments.
2. Combine functions that differ only by matrix "type."

The second part especially has pitfalls that we will describe below.

As an example of the first part, the BLAS functions `xSYRK` and `xSYR1` differ only by rank of their input arguments. Both perform a symmetric outer-product update of their input/output matrix argument `C`. `xSYRK` could implement `xSYR1` by taking an input "matrix" `A` with a single column. This is not necessarily the fastest way to implement `xSYR1`. However, since the rank of an `mdspan` or `mdarray` is known at compile time, implementations could dispatch to the appropriate low-level computational kernel with no run-time overhead. (Existing BLAS implementations do not always optimize for the case where a "matrix" argument to a BLAS 3 function like `xGEMM` has only one column, and thus the function could dispatch to a BLAS 2 function like `xGEMV`.)

Here are arguments _for_ this approach:

1. It reduces the number of new functions.
2. Implementations could identify all special cases at compile time.

Here are arguments _against_ this approach:

1. It adds special cases to implementations.
2. It's easy to make mistakes: for example, `xTRMV` and `xTRMM` differ by `SIDE` argument. Combining them while ignoring `SIDE` would lose use cases.
3. It calls for a "tagging matrices with properties" mechanism that we rejected above.

For instance, the BLAS functions `xGEMM` and `xSYRK` appear to differ just by assumptions on their output argument. `xGEMM` computes the matrix-matrix product update `C := alpha * A * B + beta * C`, and assumes that `C` has the General BLAS matrix "type." `xSYRK` computes the symmetric matrix-matrix product update `C := alpha * A * A^T + beta * C`, where `C` is assumed to be symmetric and the algorithm only accesses either the upper or lower triangle of `C`. If users could "tag" `C` as symmetric, then it seems like we could express both algorithms as a single function `gemm`. However, this approach easily leads to unexpected behavior. What if `C` has a symmetric layout and `A * B` is nonsymmetric, but users request to compute `C := A * B`? The result `C` would be mathematically incorrect, even though it would retain symmetry.

Our preference: Do not combine functions in this way.

Retain existing BLAS behavior for scalar multipliers?

The BLAS Standard treats zero values of `alpha` or `beta` scalar multipliers as special short-circuiting cases. For example, the matrix-matrix multiply update `C := alpha * A * B + beta * C` does not compute `A*B` if `alpha` is zero, and treats `C` as write only if `beta` is zero. We propose to change this behavior by always performing the requested operation, regardless of the values of any scalar multipliers.

This has the following _advantages_:

1. It removes special cases.
2. It avoids branches, which could affect performance for small problems.
3. It does not privilege floating-point element types.

However, it has the following _disadvantages_:

1. Implementations based on an existing BLAS library must "double-check" scaling factors. If any is zero, the implementation cannot call the BLAS and must perform the scaling manually. This

   will likely reduce performance for a case that users intend to be
   fast, unless the implementation has access to internal BLAS details
   that can skip the special cases.
2. Users may expect BLAS semantics in a library that imitates BLAS
   functionality. These users will get unpleasantly surprising
   results (like `Inf` or `NaN` instead of zero, if they set `alpha=0`
   and assume short circuiting).

Our preference: Remove the special short-circuiting cases.
We mitigate the disadvantages by offering both write-only and
read-and-write versions of algorithms like matrix-matrix multiply,
whose BLAS versions take a `beta` argument. In our experience using
the BLAS, users are more likely to expect that setting `beta=0` causes
write-only behavior. Thus, if the interface suggests write-only
behavior, users are less likely to be unpleasantly surprised.

Acknowledgments

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 Energyâ€™s National Nuclear Security
Administration under contract DE-NA0003525.

Special thanks to Bob Steagall and Guy Davidson for boldly leading the
charge to add linear algebra to the C++ Standard Library, and for many
fruitful discussions. Thanks also to Andrew Lumsdaine for his
pioneering efforts and history lessons.

References by coathors

* G. Ballard, E. Carson, J. Demmel, M. Hoemmen, N. Knight, and
  O. Schwartz, ["Communication lower bounds and optimal algorithms for
  numerical linear
  algebra,"](https://doi.org/10.1017/S0962492914000038), *Acta
  Numerica*, Vol. 23, May 2014, pp. 1-155.
* H. C. Edwards, B. A. Lelbach, D. Sunderland, D. Hollman, C. Trott,
  M. Bianco, B. Sander, A. Iliopoulos, J. Michopoulos, and M. Hoemmen,
  "`mdspan`: a Non-Owning Multidimensional Array Reference,"
  [P0009R0](http://wg21.link/p0009r9), Jan. 2019.
* M. Hoemmen, D. Hollman, and C. Trott, "Evolving a Standard C++
  Linear Algebra Library from the BLAS," P1674R0, Jun. 2019.
* M. Hoemmen, J. Badwaik, M. Brucher, A. Iliopoulos, and
  J. Michopoulos, "Historical lessons for C++ linear algebra library
  standardization," [(P1417R0)](http://wg21.link/p1417r0), Jan. 2019.
* D. Hollman, C. Trott, M. Hoemmen, and D. Sunderland, "`mdarray`: An
  Owning Multidimensional Array Analog of `mdspan`",
  [P1684R0](https://isocpp.org/files/papers/P1684R0.pdf), Jun. 2019.
* D. Hollman, C. Kohlhoff, B. Lelbach, J. Hoberock, G. Brown, and
  M. Dominiak, "A General Property Customization Mechanism,"
  [P1393R0](http://wg21.link/p1393r0), Jan. 2019.

Other references

* [Basic Linear Algebra Subprograms Technical (BLAST) Forum
  Standard](http://netlib.org/blas/blast-forum/blas-report.pdf),
  International Journal of High Performance Applications and
  Supercomputing, Vol. 16. No. 1, Spring 2002.
* L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling,
  G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo,
  K. Remington, and R. C. Whaley, ["An updated set of basic linear
  algebra subprograms (BLAS),"](https://doi.org/10.1145/567806.567807)
  ACM Transactions on Mathematical Software (TOMS), Vol. 28, No. 2,
  Jun. 2002, pp. 135-151.
* G. Davidson and B. Steagall, "A proposal to add linear algebra
  support to the C++ standard library," [P1385R1](http://wg21.link/p1385r1),
  Mar. 2019.
* B. Dawes, H. Hinnant, B. Stroustrup, D. Vandevoorde, and M. Wong,
  "Direction for ISO C++," [P0939R0](http://wg21.link/p0939r0), Feb. 2018.
* J. Dongarra, R. Pozo, and D. Walker, "LAPACK++: A Design Overview of
  Object-Oriented Extensions for High Performance Linear Algebra," in
  Proceedings of Supercomputing '93, IEEE Computer Society Press,
  1993, pp. 162-171.
* M. Gates, P. Luszczek, A. Abdelfattah, J. Kurzak, J. Dongarra,
  K. Arturov, C. Cecka, and C. Freitag, ["C++ API for BLAS and

LAPACK,"](https://www.icl.utk.edu/files/publications/2017/icl-utk-1031-2017.pdf)
  SLATE Working Notes, Innovative Computing Laboratory, University of
  Tennessee Knoxville, Feb. 2018.
* K. Goto and R. A. van de Geijn, "Anatomy of high-performance matrix
  multiplication,"](https://doi.org/10.1145/1356052.1356053), *ACM
  Transactions on Mathematical Software* (TOMS), Vol. 34, No. 3, May
  2008.
* J. Hoberock, "Integrating Executors with Parallel Algorithms,"
  [P1019R2](http://wg21.link/p1019r2), Jan. 2019.
* N. A. Josuttis, "The C++ Standard Library: A Tutorial and Reference,"
  Addison-Wesley, 1999.
* M. Kretz, "Data-Parallel Vector Types & Operations,"

[P0214r9](http://wg21.link/p0214r9), Mar. 2018.
* D. Vandevoorde and N. A. Josuttis, "C++ Templates: The Complete Guide," Addison-Wesley Professional, 2003.