P1673R3: A free function linear algebra interface based on the BLAS

Authors

- Mark Hoemmen (mhoemmen@stellarscience.com) (Stellar Science)
- Daisy 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)
- Li-Ta Lo (ollie@lanl.gov) (Los Alamos National Laboratory)
- Damien Lebrun-Grandie (lebrungrandt@ornl.gov) (Oak Ridge National Laboratories)
- Graham Lopez (glopez@nvidia.com) (NVIDIA)
- 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: 2021-04-15

Revision history

- Revision 0 (pre-Cologne) submitted 2019-06-17
    - Received feedback in Cologne from SG6, LEWGI, and (???).
- Revision 1 (pre-Belfast) to be submitted 2019-10-07
    - Account for Cologne 2019 feedback
        - Make interface more consistent with existing Standard algorithms
            - Change `dot`, `dotc`, `vector_norm2`, and `vector_abs_sum` to imitate `reduce`, so that they return their result, instead of taking an output parameter. Users may set the result type via optional `init` parameter.
    - Minor changes to "expression template" classes, based on implementation experience
    - Briefly address LEWGI request of exploring concepts for input arguments.
    - Lazy ranges style API was NOT explored.

- Revision 2 (pre-Cologne) to be submitted 2020-01-13
  - Add "Future work" section.
  - Remove "Options and votes" section (which were addressed in SG6, SG14, and LEWGI).
  - Remove `basic_mdarray` overloads.
  - Remove batched linear algebra operations.
  - Remove over- and underflow requirement for `vector_norm2`.
  - *Mandate* any extent compatibility checks that can be done at compile time.
  - Add missing functions `{symmetric,hermitian}_matrix_rank_k_update` and `triangular_matrix_{left,right}_product`.
  - Remove `packed_view` function.
  - Fix wording for `{conjugate,transpose,conjugate_transpose}_view`, so that implementations may optimize the return type. Make sure that `transpose_view` of a `layout_blas_packed` matrix returns a `layout_blas_packed` matrix with opposite `Triangle` and `StorageOrder`.
  - Remove second template parameter `T` from `accessor_conjugate`.
  - Make `scaled_scalar` and `conjugated_scalar` exposition only.
  - Add in-place overloads of `triangular_matrix_matrix_{left,right}_solve`, `triangular_matrix_{left,right}_product`, and `triangular_matrix_vector_solve`.
  - Add `alpha` overloads to `{symmetric,hermitian}_matrix_rank_{1,k}_update`.
  - Add Cholesky factorization and solve examples.
- Revision 3 (electronic) to be submitted 2021-04-15
  - Per LEWG request, add a section on our investigation of constraining template parameters with concepts, in the manner of P1813R0 with the numeric algorithms. We concluded that we disagree with the approach of P1813R0, and that the Standard's current **GENERALIZED_SUM** approach better expresses numeric algorithms' behavior.
  - Update references to the current revision of P0009 (`mdspan`).
  - Per LEWG request, introduce `std::linalg` namespace and put everything in there.
  - Per LEWG request, replace the `linalg_` prefix with the aforementioned namespace. We renamed `linalg_add` to `add`, `linalg_copy` to `copy`, and `linalg_swap` to `swap_elements`.
  - Per LEWG request, do not use `_view` as a suffix, to avoid confusion with "views" in the sense of Ranges. We renamed `conjugate_view` to `conjugated`, `conjugate_transpose_view` to `conjugate_transposed`, `scaled_view` to `scaled`, and `transpose_view` to `transposed`.

- Change wording from "then implementations will use `T`'s precision or greater for intermediate terms in the sum," to "then intermediate terms in the sum use `T`'s precision or greater." Thanks to Jens Maurer for this suggestion (and many others!).

- Before, a Note on `vector_norm2` said, "We recommend that implementers document their guarantees regarding overflow and underflow of `vector_norm2` for floating-point return types." Implementations always document "implementation-defined behavior" per **[defs.impl.defined]**. (Thanks to Jens Maurer for pointing out that "We recommend..." does not belong in the Standard.) Thus, we changed this from a Note to normative wording in Remarks: "If either `in_vector_t::element_type` or `T` are floating-point types or complex versions thereof, then any guarantees regarding overflow and underflow of `vector_norm2` are implementation-defined."

- Define return types of the `dot`, `dotc`, `vector_norm2`, and `vector_abs_sum` overloads with `auto` return type.

- Remove the explicitly stated constraint on `add` and `copy` that the rank of the array arguments be no more than 2. This is redundant, because we already impose this via the existing constraints on template parameters named `in_object*_t`, `inout_object*_t`, or `out_object*_t`. If we later wish to relax this restriction, then we only have to do so in one place.

- Add `vector_sum_of_squares`. First, this gives implementers a path to implementing `vector_norm2` in a way that achieves the over/underflow guarantees intended by the BLAS Standard. Second, this is a useful algorithm in itself for parallelizing vector 2-norm computation.

- Add `matrix_frob_norm`, `matrix_one_norm`, and `matrix_inf_norm` (thanks to coauthor Piotr Luszczek).

- Address LEWG request for us to investigate support for GPU memory. See section "Explicit support for asynchronous return of scalar values."

- Add `ExecutionPolicy` overloads of the in-place versions of `triangular_matrix_vector_solve`, `triangular_matrix_left_product`, `triangular_matrix_right_product`, `triangular_matrix_matrix_left_solve`, and `triangular_matrix_matrix_right_solve`.

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. Our proposal implements the following classes of algorithms on arrays that represent matrices and vectors:

- Elementwise vector sums
- Multiplying all elements of a vector or matrix by a scalar
- 2-norms and 1-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 of 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 `basic_mdspan` (P0009R10), a multidimensional array view, 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).

- 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).

- Unlike the BLAS, our proposal can be expanded to support "batched" operations (see 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 P1385, 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 P1385 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. A straightforward way to do that would be for P1385's matrix and vector objects to make views of their data available as `basic_mdspan`.

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

1. C++ applications in "important application areas" (see 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 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) 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. Some of them even implement specific linear algebra operations directly in hardware. Examples include NVIDIA's Tensor Cores and Cerebras' Wafer Scale Engine. 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.") 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 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 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 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).

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 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 a well-understood problem whose solutions could be parameterized for a variety of computer architectures. See, for example, Goto and van de Geijn 2008. There are optimized third-party BLAS implementations for common architectures, like ATLAS and GotoBLAS. A (slow but correct) reference implementation of the BLAS 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" (P1674). 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.

Criteria for including algorithms

We include algorithms in our proposal based on the following criteria, ordered by decreasing importance. Many of our algorithms satisfy multiple criteria.

1. Getting the desired asymptotic run time is nontrivial

2. Opportunity for vendors to provide hardware-specific optimizations

3. Opportunity for vendors to provide quality-of-implementation improvements, especially relating to accuracy or reproducibility with respect to floating-point rounding error

4. User convenience (familiar name, or tedious to implement)

Regarding (1), "nontrivial" means "at least for novices to the field." Dense matrix-matrix multiply is a good example. Getting close to the asymptotic lower bound on the number of memory reads and writes matters a lot for performance, and calls for a nonintuitive loop reordering. An analogy to the current C++ Standard Library is `sort`, where intuitive algorithms that many humans use are not asymptotically optimal.

Regarding (2), a good example is copying multidimensional arrays. The Kokkos library spends about 2500 lines of code on multidimensional array copy, yet still relies on system libraries for low-level optimizations. An analogy to the current C++ Standard Library is `copy` or even `memcpy`.

Regarding (3), accurate floating-point summation is nontrivial. Well-meaning compiler optimizations might defeat even simple technqiues, like compensated summation. The most obvious way to compute a vector's Euclidean norm (square root of sum of squares) can cause overflow or underflow, even when the exact answer is much smaller than the overflow threshold, or larger than the underflow threshold. Some users care deeply about sums, even parallel sums, that always get the same answer, despite rounding error. This can help debugging, for example. It is possible to make floating-point sums completely independent of parallel evaluation order. See e.g., the ReproBLAS effort. Naming these algorithms and providing `ExecutionPolicy` customization hooks gives vendors a chance to provide these improvements. An analogy to the current C++ Standard Library is `hypot`, whose language in the C++ Standard alludes to the tighter POSIX requirements.

Regarding (4), the C++ Standard Library is not entirely minimalist. One example is `std::string::contains`. Existing Standard Library algorithms already offered this functionality, but a member `contains` function is easy for novices to find and use, and avoids the tedium of comparing the result of `find` to `npos`.

The BLAS exists mainly for the first two reasons. It includes functions that were nontrivial for compilers to optimize in its time, like scaled elementwise vector sums, as well as functions that generally require human effort to optimize, like matrix-matrix multiply.

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 nonconjugated 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:

- nonconjugated dot product `xDOTU` and conjugated dot product `xDOTC`; and
- rank-1 symmetric (`xGERU`) vs. Hermitian (`xGERC`) matrix update.

The conjugate transpose and the (nonconjugated) 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 library, nor in the classic BLAS "level" 1, 2, and 3 papers. (For history of the BLAS "levels" and a bibliography, see 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.

4. Our proposal subsumes some dense mixed-precision functionality (see below).

LAPACK or related functionality

The 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.

Several C++ libraries provide slices of LAPACK functionality. Here is a brief, noninclusive list, in alphabetical order, of some libraries actively being maintained:

- Armadillo,
- Boost.uBLAS,
- Eigen,
- Matrix Template Library, and
- Trilinos.

P1417R0 gives some history of C++ linear algebra libraries. The authors of this proposal have designed, written, and maintained 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++ in pre-templates C++ circa 1993, to the recent "C++ API for BLAS and LAPACK". (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), 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.

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 P1385, could use our interface to implement overloaded arithmetic for matrices and vectors. P0939R0 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.

P1385 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, per-iteration run time, and memory requirements. 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 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 describes this common problem.

Our `scaled`, `conjugated`, `transposed`, and `conjugate_transposed` functions make use of one aspect of expression templates, namely modifying the `basic_mdspan` 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.

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` has 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.

Explicit support for asynchronous return of scalar values

After we presented revision 2 of this paper, LEWG asked us to consider support for discrete graphics processing units (GPUs). GPUs have two features of interest here. First, they might have memory that is not accessible from ordinary C++ code, but could be accessed in a standard algorithm (or one of our proposed algorithms) with the right implementation-specific `ExecutionPolicy`. (For instance, a policy could say "run this algorithm on the GPU.") Second, they might execute those algorithms asynchronously. That is, they might write to output arguments at some later time after the algorithm invocation returns. This would imply different interfaces in some cases. For instance, a hypothetical asynchronous vector 2-norm might write its scalar result via a pointer to GPU memory, instead of returning the result "on the CPU."

Nothing in principle prevents `basic_mdspan` from viewing memory that is inaccessible from ordinary C++ code. This is a major feature of the `Kokkos::View` class from the Kokkos library, and `Kokkos::View` directly inspired `basic_mdspan`. The C++ Standard does not currently define how such memory behaves, but implementations could define its behavior and make it work with `basic_mdspan`. This would, in turn, let implementations define our algorithms to operate on such memory efficiently, if given the right implementation-specific `ExecutionPolicy`.

Our proposal excludes algorithms that might write to their output arguments at some time after after the algorithm returns. First, LEWG insisted that our proposed algorithms that compute a scalar result, like `vector_norm2`, return that result in the manner of `reduce`, rather than writing the result to an output reference or pointer. (Previous revisions of our proposal used the latter interface pattern.) Second, it's not clear whether writing a scalar result to a pointer is the right interface for asynchronous algorithms. Follow-on proposals to Executors (P0443R14) include asynchronous algorithms, but none of these suggest returning results asynchronously by pointer. Our proposal deliberately imitates the existing standard algorithms. Right now, we have no standard asynchronous algorithms to imitate.

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 propose to build the initial interface on top of `basic_mdspan`, and plan to extend that later with overloads for a new `basic_mdarray` variant of `basic_mdspan` with container semantics as well as any type implementing a `get_mdspan` customization point. We explain the value of these choices 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). 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`?

- 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` supports row-major, column-major, and strided layouts out of the box, and it has `Layout` as an extension point. This lets our interface support layouts beyond what the BLAS Standard permits.

- Using `basic_mdspan` lets our algorithms exploit any dimensions or strides known at compile time.

- `basic_mdspan` has built-in "slicing" capabilities via `subspan`.

- `basic_mdspan`'s layout and accessor policies let us simplify our interfaces, by encapsulating transpose, conjugate, and scalar arguments. See below for details.

- `basic_mdspan` is low level; it imposes 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 C++ Standard has already taken the word `vector`.)

- Using `basic_mdspan` offers us a hook for future expansion to support heterogeneous memory spaces. (This is a key feature of `Kokkos::View`, the data structure that inspired `basic_mdspan`.)

- `basic_mdspan`'s encapsulation of matrix indexing makes C++ implementations of BLAS-like operations much less error prone and easier to read.

- Using `basic_mdspan` will make it easier for us to add an efficient "batched" interface in future proposals.

Defining a concept for the data structures instead

LEWGI requested in the 2019 Cologne meeting that we explore using a concept instead of `basic_mdspan` to define the arguments for the linear algebra functions. We investigated this option, and rejected it, for the following reasons.

1. Our proposal uses enough features of `basic_mdspan` that any concept generally applicable to all functions we propose would largely replicate the definition of `basic_mdspan`.

2. This proposal could support most multidimensional array types, if the array types just made themselves convertible to `basic_mdspan`.

3. We could always generalize our algorithms later.

4. Any multidimensional array concept would need revision in the light of P2128R3.

This proposal refers to almost all of `basic_mdspan`'s features, including `extents`, `layout`, and `accessor_policy`. We expect implementations to use all of them for optimizations, for example to extract the scaling factor from the return value of `scaled` in order to call an optimized BLAS library directly.

Suppose that a general customization point `get_mdspan` existed, that takes a reference to a multidimensional array type and returns a `basic_mdspan` that views the array. Then, our proposal could support most multidimensional array types. "Most" includes all such types that refer to a subset of a contiguous span of memory.

Requiring that a multidimensional array refer to a subset of a contiguous span of memory would exclude multidimensional array types that have a noncontiguous backing store, such as a `map`. If we later wanted to support such types, we could always generalize our algorithms later.

Finally, any multidimensional array concept would need revision in the light of P2128R3, which finished LEWG review in March 2021. P2128 proposes letting `operator[]` take multiple parameters. Its authors intend to let `basic_mdspan` use `operator[]` instead of `operator()`.

After further discussion at the 2019 Belfast meeting, LEWGI accepted our position that having our algorithms take `basic_mdspan` instead of template parameters constrained by a multidimensional array concept would be fine for now.

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 triangular solve and triangular multiply, in-place behavior is essential for computing matrix factorizations in place, without requiring extra storage proportional to the input matrix's dimensions. However, in-place functions cannot be parallelized for arbitrary execution policies. Thus, we have both not-in-place and in-place overloads, and only the not-in-place overloads take an optional `ExecutionPolicy&&`.

   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. Other types actually represent algorithmic constraints: for instance, what entries of the matrix the algorithm is allowed to access.

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.

Over- and underflow wording for vector 2-norm

SG6 recommended to us at Belfast 2019 to change the special overflow / underflow wording for `vector_norm2` to imitate the BLAS Standard more closely. The BLAS Standard does say something about overflow and underflow for vector 2-norms. We reviewed this wording and conclude that it is either a nonbinding quality of implementation (QoI) recommendation, or too vaguely stated to translate directly into C++ Standard wording. Thus, we removed our special overflow / underflow wording. However, the BLAS Standard clearly expresses the intent that implementations document their underflow and overflow guarantees for certain functions, like vector 2-norms. The C++ Standard requires documentation of "implementation-defined behavior." Therefore, we added language to our proposal that makes "any guarantees regarding overflow and underflow" of those certain functions "implementation-defined."

Previous versions of this paper asked implementations to compute vector 2-norms "without undue overflow or underflow at intermediate stages of the computation." "Undue" imitates existing C++ Standard wording for `hypot`. This wording hints at the stricter requirements in F.9 (normative, but optional) of the C Standard for math library functions like `hypot`, without mandating those requirements. In particular, paragraph 9 of F.9 says:

> Whether or when library functions raise an undeserved "underflow" floating-point exception is unspecified. Otherwise, as implied by F.7.6, the **<math.h>** functions do not raise spurious floating-point exceptions (detectable by the user) [including the "overflow" exception discussed in paragraph 6], other than the "inexact" floating-point exception.

However, these requirements are for math library functions like `hypot`, not for general algorithms that return floating-point values. SG6 did not raise a concern that we should treat `vector_norm2` like a math library function; their concern was that we imitate the BLAS Standard's wording.

The BLAS Standard says of several operations, including vector 2-norm: "Here are the exceptional routines where we ask for particularly careful implementations to avoid unnecessary over/underflows, that could make the output unnecessarily inaccurate or unreliable" (p. 35).

The BLAS Standard does not define phrases like "unnecessary over/underflows." The likely intent is to avoid naïve implementations that simply add up the squares of the vector elements. These would overflow even if the norm in exact arithmetic is significantly less than the overflow threshold. The POSIX Standard (IEEE Std 1003.1-2017) analogously says that `hypot` must "take precautions against overflow during intermediate steps of the computation."

The phrase "precautions against overflow" is too vague for us to translate into a requirement. The authors likely meant to exclude naïve implementations, but not require implementations to know whether a result computed in exact arithmetic would overflow or underflow. The latter is a special case of computing floating-point sums exactly, which is costly for vectors of arbitrary length. While it would be a useful feature, it is difficult enough that we do not want to require it, especially since the BLAS Standard itself does not. The Reference BLAS implementation of vector 2-norms `DNRM2` maintains the current maximum absolute value of all the vector entries seen thus far, and scales each vector entry by that maximum, in the same way as the LAPACK routine `DLASSQ`. Implementations could also first compute the sum of squares in a straightforward loop. They could then recompute if needed, for example by testing if the result is `Inf` or `NaN`.

For all of the functions listed on p. 35 of the BLAS Standard as needing "particularly careful implementations," *except* vector norm, the BLAS Standard has an "Advice to implementors" section with extra accuracy requirements. The BLAS Standard does have an "Advice to implementors" section for matrix norms (see Section 2.8.7, p. 69), which have similar over- and underflow concerns as vector norms. However, the Standard merely states that "[h]igh-quality implementations of these routines should be accurate" and should document their accuracy, and gives examples of "accurate implementations" in LAPACK.

The BLAS Standard never defines what "Advice to implementors" means. However, the BLAS Standard shares coauthors and audience with the Message Passing Interface (MPI) Standard, which defines "Advice to implementors" as "primarily commentary to implementors" and permissible to skip (see e.g., MPI 3.0, Section

2.1, p. 9). We thus interpret "Advice to implementors" in the BLAS Standard as a nonbinding quality of implementation (QoI) recommendation.

Why no concepts for template parameters?

We need adverbs, not adjectives

LEWG's 2020 review of P1673R2 asked us to investigate conceptification of its algorithms. "Conceptification" here refers to an effort like that of P1813R0 ("A Concept Design for the Numeric Algorithms"), to come up with concepts that could be used to constrain the template parameters of numeric algorithms like `reduce` or `transform`. (We are not referring to LEWGI's request for us to consider generalizing our algorithm's parameters from `basic_mdspan` to a hypothetical multidimensional array concept. We discuss that above; see "Defining a concept for the data structures instead.") The numeric algorithms are relevant to P1673 because many of the algorithms proposed in P1673 look like generalizations of `reduce` or `transform`. We intend for our algorithms to be generic on their matrix and vector element types, so these questions matter a lot to us.

We agree that it is useful to set constraints that make it possible to reason about correctness of algorithms. However, our concern is that P1813R0 imposes requirements that are too strict to be useful for practical types, like associativity. Concepts give us *adjectives*, that describe the element types of input and output arrays. What we actually want are *adverbs*, that describe the algorithms we apply to those arrays. The Standard already has machinery like **GENERALIZED_SUM** that we can (and do) use to describe our algorithms in an adverbial way.

Associativity is too strict

P1813R0 requires associative addition for many algorithms, such as `reduce`. However, many practical arithmetic systems that users might like to use with algorithms like `reduce` have non-associative addition. These include

- systems with rounding;
- systems with an "infinity": e.g., if 10 is Inf, 3 + 8 - 7 could be either Inf or 4; and
- saturating arithmetic: e.g., if 10 saturates, 3 + 8 - 7 could be either 3 or 4.

Note that the latter two arithmetic systems have nothing to do with rounding error. With saturating integer arithmetic, parenthesizing a sum in different ways might give results that differ by as much as the saturation threshold. It's true that many non-associative arithmetic systems behave "associatively enough" that users don't fear parallelizing sums. However, a concept with an exact property (like "commutative semigroup") isn't the right match for "close enough," just like `operator==` isn't the right match for describing "nearly the same." For some number systems, a rounding error bound might be more appropriate, or guarantees on when underflow or overflow may occur (as in POSIX's `hypot`).

The problem is a mismatch between the constraint we want to express -- that "the algorithm may reparenthesize addition" -- and the constraint that "addition is associative." The former is an adverb, describing what the algorithm (a verb) does. The latter is an adjective, describing the type (a noun) used with an algorithm. Given the huge variety of possible arithmetic systems, an approach like the Standard's use of **GENERALIZED_SUM** to describe `reduce` and its kin seems more helpful. If the Standard describes an algorithm in terms of **GENERALIZED_SUM**, then that tells the caller what the algorithm might do. The caller then takes responsibility for interpreting the algorithm's results.

We think this is important both for adding new algorithms (like those in this proposal) and for defining behavior of an algorithm with respect to different `ExecutionPolicy` arguments. (For instance, `par_unseq` could imply that the algorithm might change the order of terms in a sum, while `par` need not. Compare to `MPI_Op_create`'s `commute` parameter, that affects the behavior of algorithms like `MPI_Reduce` when used with the resulting user-defined reduction operator.)

Generalizing associativity does not help

Suppose we accept that associativity and related properties are not useful for describing our proposed algorithms. Could there be a generalization of associativity that *would* be useful? P1813R0's most general concept is a `magma`. Mathematically, a *magma* is a set M with a binary operation ×, such that if a and b are in M, then a × b is in M. The operation need not be associative or commutative. While this seems almost too general to be useful, there are two reasons why even a magma is too specific for our proposal.

- It only assumes one set, that is, one type. This does not accurately describe what the algorithms do, and it excludes useful features like mixed precision and types that use expression templates.
- Magma is too specific, because algorithms are useful even if the binary operation is not closed.

First, even for simple linear algebra operations that "only" use plus and times, there is no one "set M" over which plus and times operate. There are actually three operations: plus, times, and assignment. Each operation may have completely heterogeneous input(s) and output. The sets (types) that may occur vary from algorithm to algorithm, depending on the input type(s), and the algebraic expression(s) that the algorithm is allowed to use. We might need several different concepts to cover all the expressions that algorithms use, and the concepts would end up being less useful to users than the expressions themselves.

For instance, consider the Level 1 BLAS "AXPY" function. This computes `y(i) = alpha * x(i) + y(i)` elementwise. What type does the expression `alpha * x(i) + y(i)` have? It doesn't need to have the same type as `y(i)`; it just needs to be assignable to `y(i)`. The types of `alpha`, `x(i)`, and `y(i)` could all differ. As a simple example, `alpha` might be `int`, `x(i)` might be `float`, and `y(i)` might be `double`. The types of `x(i)` and `y(i)` might be more complicated; e.g., `x(i)` might be a polynomial with `double` coefficients, and `y(i)` a polynomial with `float` coefficients. If those polynomials use expression templates, then the expression `x(i) + x(i)` might have a completely different type than `decltype(x(i))` (possibly with references removed), and might also have a completely different type than `alpha * x(i) + y(i)`.

We could try to describe this with a concept that expresses a sum type. The sum type would include all the types that might show up in the expression. However, we do not think this would improve clarity over just the expression. Furthermore, different algorithms may need different expressions, so we would need multiple concepts, one for each expression. Why not just use the expressions to describe what the algorithms can do?

Second, the magma concept is not helpful even if we only had one set M, because our algorithms would still be useful even if binary operations were not closed over that set. For example, consider a hypothetical user-defined rational number type, where plus and times throw if representing the result of the operation would take more than a given fixed amount of memory. Programmers might handle this exception by falling back to different algorithms. Neither plus or times on this type would satisfy the magma requirement, but the algorithms would still be useful for such a type. One could consider the magma requirement satisfied in a purely syntactic sense, because of the return type of plus and times. However, saying that would not accurately express the type's behavior.

This point returns us to the concerns we expressed earlier about assuming associativity. "Approximately associative" or "usually associative" are not useful concepts without further refinement. The way to refine these concepts usefully is to describe the behavior of a type fully, e.g., the way that IEEE 754 describes the behavior of floating-point numbers. However, algorithms rarely depend on all the properties in a specification like IEEE 754. The problem, again, is that we need adverbs, not adjectives. We want to describe what the algorithms do -- e.g., that they can rearrange terms in a sum -- not how the types that go into the algorithms behave.

Summary

- Many useful types have nonassociative or even non-closed arithmetic.
- Lack of (e.g.,) associativity is not just a rounding error issue.
- It can be useful to let algorithms do things like reparenthesize sums or products, even for types that are not associative.
- Permission for an algorithm to reparenthesize sums is not the same as a concept constraining the terms in the sum.
- We can and do use existing Standard language, like **GENERALIZED_SUM**, for expressing permissions that algorithms have.

Future work

Summary:

1. Generalize function parameters to take any type that implements the `get_mdspan` customization point, including `basic_mdarray`.

2. Add batched linear algebra overloads.

Generalize function parameters

Our functions differ from the C++ Standard algorithms, in that they take a concrete type `basic_mdspan` with template parameters, rather than any type that satisfies a concept. We think that the template parameters of `basic_mdspan` fully describe the multidimensional equivalent of a multipass iterator, and that "conceptification" of multidimensional arrays would unnecessarily delay both this proposal. and P0009 (the `basic_mdspan` proposal).

In a future proposal, we plan to generalize our function's template parameters, to permit any type besides `basic_mdspan` that implements the `get_mdspan` customization point, as long as the return value of `get_mdspan` satisfies the current requirements. `get_mdspan` will return a `basic_mdspan` that views its argument's data.

`basic_mdarray`, proposed in P1684, is the container analog of `basic_mdspan`. It is a new kind of container, with the same copy behavior as containers like `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 P0009) 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`.

Previous versions of this proposal included function overloads that took `basic_mdarray` directly. The goals were user convenience, and to avoid any potential overhead of conversion to `basic_mdspan`, especially for very small matrices and vectors. In a future revision of P1684, `basic_mdarray` will implement `get_mdspan`. This will let users use `basic_mdarray` directly in our functions. This customization point approach would also simplify using our functions with other matrix and vector types, such as those proposed by P1385. Implementations may optionally add direct overloads of our functions for `basic_mdarray` or other types. This would address any concerns about overhead of converting from `basic_mdarray` to `basic_mdspan`.

Batched linear algebra

We plan to write a separate proposal that will add "batched" versions of linear algebra functions to this proposal. "Batched" linear algebra functions solve many independent problems all at once, in a single function call. For discussion, see Section 6.2 of our background paper P1417R0. Batched interfaces have the following advantages:

- They expose more parallelism and vectorization opportunities for many small linear algebra operations.
- They are useful for many different fields, including machine learning.
- Hardware vendors currently offer both hardware features and optimized software libraries to support batched linear algebra.
- There is an ongoing interface standardization effort, in which we participate.

The `basic_mdspan` data structure makes it easy to represent a batch of linear algebra objects, and to optimize their data layout.

With few exceptions, the extension of this proposal to support batched operations will not require new functions or interface changes. Only the requirements on functions will change. Output arguments can have an additional rank; if so, then the leftmost extent will refer to the batch dimension. Input arguments may also have an additional rank to match; if they do not, the function will use ("broadcast") the same input argument for all the output arguments in the batch.

Data structures and utilities borrowed from other proposals

`basic_mdspan`

This proposal depends on P0009R10, which is a proposal for adding multidimensional arrays to the C++ Standard Library. `basic_mdspan` is the main class in P0009. 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`.

New `basic_mdspan` layouts in this proposal

Our proposal uses the layout mapping policy of `basic_mdspan` in order to represent different matrix and vector data layouts. Layout mapping policies as described by P0009R10 have three basic properties:

- Unique
- Contiguous
- Strided

P0009R10 includes three different layouts -- `layout_left`, `layout_right`, and `layout_stride` -- all of which are unique and strided. Only `layout_left` and `layout_right` are contiguous.

This proposal includes the following additional layouts:

- `layout_blas_general`: Generalization of `layout_left` and `layout_right`; describes layout used by General (GE) 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). P1674 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`. We address these different matrix types with different function names.

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 `layout_blas_packed` is the first addition to the layouts in P0009R10 that is neither always unique, nor always strided.

Algorithms cannot be written generically if they permit output arguments with nonunique layouts. 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 will impose the following rule: Any `basic_mdspan` output 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.

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 Energy'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

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,", *Acta Numerica*, Vol. 23, May 2014, pp. 1-155.

- C. Trott, D. S. Hollman, D. Lebrun-Grande, M. Hoemmen, D. Sunderland, H. C. Edwards, B. A. Lelbach, M. Bianco, B. Sander, A. Iliopoulos, and J. Michopoulos, "`mdspan`: a Non-Owning Multidimensional Array Reference," P0009R10, Feb. 2020.

- M. Hoemmen, D. S. 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), Jan. 2019.

- M. Hoemmen, D. S. Hollman, C. Jabot, I. Muerte, and C. Trott, "Multidimensional subscript operator," P2128R3, Feb. 2021.

- D. S. Hollman, C. Trott, M. Hoemmen, and D. Sunderland, "`mdarray`: An Owning Multidimensional Array Analog of `mdspan`", P1684R0, Jun. 2019.

- D. S. Hollman, C. Kohlhoff, B. A. Lelbach, J. Hoberock, G. Brown, and M. Dominiak, "A General Property Customization Mechanism," P1393R0, Jan. 2019.

Other references

- Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard, 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)," *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," P1385R4, Nov. 2019.

- B. Dawes, H. Hinnant, B. Stroustrup, D. Vandevoorde, and M. Wong, "Direction for ISO C++," 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," 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, Jan. 2019.
- N. A. Josuttis, "The C++ Standard Library: A Tutorial and Reference," Addison-Wesley, 1999.
- M. Kretz, "Data-Parallel Vector Types & Operations," P0214r9, Mar. 2018.
- D. Vandevoorde and N. A. Josuttis, "C++ Templates: The Complete Guide," Addison-Wesley Professional, 2003.

Wording

> Text in blockquotes is not proposed wording, but rather instructions for generating proposed wording. The � character is used to denote a placeholder section number which the editor shall determine. First, apply all wording from P0009R10 (this proposal is a "rebase" atop the changes proposed by P0009R10). At the end of Table � ("Numerics library summary") in *[numerics.general]*, add the following: [linalg], Linear algebra, `<linalg>`. At the end of *[numerics]*, add all the material that follows.

Header `<linalg>` synopsis [linalg.syn]

```
namespace std::linalg {
// [linalg.tags.order], storage order tags
struct column_major_t;
inline constexpr column_major_t column_major;
struct row_major_t;
inline constexpr row_major_t row_major;

// [linalg.tags.triangle], triangle tags
struct upper_triangle_t;
inline constexpr upper_triangle_t upper_triangle;
struct lower_triangle_t;
inline constexpr lower_triangle_t lower_triangle;

// [linalg.tags.diagonal], diagonal tags
struct implicit_unit_diagonal_t;
inline constexpr implicit_unit_diagonal_t implicit_unit_diagonal;
struct explicit_diagonal_t;
inline constexpr explicit_diagonal_t explicit_diagonal;

// [linalg.layouts.general], class template layout_blas_general
```

```
template<class StorageOrder>
class layout_blas_general;

// [linalg.layouts.packed], class template layout_blas_packed
template<class Triangle,
         class StorageOrder>
class layout_blas_packed;

// [linalg.scaled.accessor_scaled], class template accessor_scaled
template<class ScalingFactor,
         class Accessor>
class accessor_scaled;

// [linalg.scaled.scaled], scaled in-place transformation
template<class ScalingFactor,
         class ElementType,
         class Extents,
         class Layout,
         class Accessor>
/* see-below */
scaled(
  const ScalingFactor& s,
  const basic_mdspan<ElementType, Extents, Layout, Accessor>& a);

// [linalg.conj.accessor_conjugate], class template accessor_conjugate
template<class Accessor>
class accessor_conjugate;

// [linalg.conj.conjugated], conjugated in-place transformation
template<class ElementType,
         class Extents,
         class Layout,
         class Accessor>
/* see-below */
conjugated(
  basic_mdspan<ElementType, Extents, Layout, Accessor> a);

// [linalg.transp.layout_transpose], class template layout_transpose
template<class Layout>
class layout_transpose;

// [linalg.transp.transposed], transposed in-place transformation
template<class ElementType,
         class Extents,
         class Layout,
         class Accessor>
/* see-below */
transposed(
  basic_mdspan<ElementType, Extents, Layout, Accessor> a);

// [linalg.conj_transp],
// conjugated transposed in-place transformation
template<class ElementType,
         class Extents,
```

```
         class Layout,
         class Accessor>
/* see-below */
conjugate_transposed(
  basic_mdspan<ElementType, Extents, Layout, Accessor> a);

// [linalg.algs.blas1.givens.lartg], compute Givens rotation
template<class Real>
void givens_rotation_setup(const Real a,
                           const Real b,
                           Real& c,
                           Real& s,
                           Real& r);
template<class Real>
void givens_rotation_setup(const complex<Real>& a,
                           const complex<Real>& a,
                           Real& c,
                           complex<Real>& s,
                           complex<Real>& r);

// [linalg.algs.blas1.givens.rot], apply computed Givens rotation
template<class inout_vector_1_t,
         class inout_vector_2_t,
         class Real>
void givens_rotation_apply(
  inout_vector_1_t x,
  inout_vector_2_t y,
  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 x,
  inout_vector_2_t y,
  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 x,
  inout_vector_2_t y,
  const Real c,
  const complex<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 x,
```

```
  inout_vector_2_t y,
  const Real c,
  const complex<Real> s);
}

// [linalg.algs.blas1.swap], swap elements
template<class inout_object_1_t,
         class inout_object_2_t>
void swap_elements(inout_object_1_t x,
                   inout_object_2_t y);
template<class ExecutionPolicy,
         class inout_object_1_t,
         class inout_object_2_t>
void swap_elements(ExecutionPolicy&& exec,
                   inout_object_1_t x,
                   inout_object_2_t y);

// [linalg.algs.blas1.scal], multiply elements by scalar
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);

// [linalg.algs.blas1.copy], copy elements
template<class in_object_t,
         class out_object_t>
void copy(in_object_t x,
          out_object_t y);
template<class ExecutionPolicy,
         class in_object_t,
         class out_object_t>
void copy(ExecutionPolicy&& exec,
          in_object_t x,
          out_object_t y);

// [linalg.algs.blas1.add], add elementwise
template<class in_object_1_t,
         class in_object_2_t,
         class out_object_t>
void 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 add(ExecutionPolicy&& exec,
         in_object_1_t x,
```

```
         in_object_2_t y,
         out_object_t z);

// [linalg.algs.blas1.dot],
// dot product of two vectors

// [linalg.algs.blas1.dot.dotu],
// nonconjugated dot product of two vectors
template<class in_vector_1_t,
         class in_vector_2_t,
         class T>
T dot(in_vector_1_t v1,
      in_vector_2_t v2,
      T init);
template<class ExecutionPolicy,
         class in_vector_1_t,
         class in_vector_2_t,
         class T>
T dot(ExecutionPolicy&& exec,
      in_vector_1_t v1,
      in_vector_2_t v2,
      T init);
template<class in_vector_1_t,
         class in_vector_2_t>
auto dot(in_vector_1_t v1,
         in_vector_2_t v2) -> /* see-below */;
template<class ExecutionPolicy,
         class in_vector_1_t,
         class in_vector_2_t>
auto dot(ExecutionPolicy&& exec,
         in_vector_1_t v1,
         in_vector_2_t v2) -> /* see-below */;

// [linalg.algs.blas1.dot.dotc],
// conjugated dot product of two vectors
template<class in_vector_1_t,
         class in_vector_2_t,
         class T>
T dotc(in_vector_1_t v1,
       in_vector_2_t v2,
       T init);
template<class ExecutionPolicy,
         class in_vector_1_t,
         class in_vector_2_t,
         class T>
T dotc(ExecutionPolicy&& exec,
       in_vector_1_t v1,
       in_vector_2_t v2,
       T init);
template<class in_vector_1_t,
         class in_vector_2_t>
auto dotc(in_vector_1_t v1,
          in_vector_2_t v2) -> /* see-below */;
template<class ExecutionPolicy,
```

```
         class in_vector_1_t,
         class in_vector_2_t>
auto dotc(ExecutionPolicy&& exec,
          in_vector_1_t v1,
          in_vector_2_t v2) -> /* see-below */;

// [linalg.algs.blas1.ssq],
// Scaled sum of squares of a vector's elements
template<class T>
struct sum_of_squares_result {
  T scaling_factor;
  T scaled_sum_of_squares;
};
template<class in_vector_t,
         class T>
sum_of_squares_result<T> vector_sum_of_squares(
  in_vector_t v,
  sum_of_squares_result init);
sum_of_squares_result<T> vector_sum_of_squares(
  ExecutionPolicy&& exec,
  in_vector_t v,
  sum_of_squares_result init);

// [linalg.algs.blas1.nrm2],
// Euclidean norm of a vector
template<class in_vector_t,
         class T>
T vector_norm2(in_vector_t v,
               T init);
template<class ExecutionPolicy,
         class in_vector_t,
         class T>
T vector_norm2(ExecutionPolicy&& exec,
               in_vector_t v,
               T init);
template<class in_vector_t>
auto vector_norm2(in_vector_t v) -> /* see-below */;
template<class ExecutionPolicy,
         class in_vector_t>
auto vector_norm2(ExecutionPolicy&& exec,
                  in_vector_t v) -> /* see-below */;

// [linalg.algs.blas1.asum],
// sum of absolute values of vector elements
template<class in_vector_t,
         class T>
T vector_abs_sum(in_vector_t v,
                 T init);
template<class ExecutionPolicy,
         class in_vector_t,
         class T>
T vector_abs_sum(ExecutionPolicy&& exec,
                 in_vector_t v,
                 T init);
```

```
template<class in_vector_t>
auto vector_abs_sum(in_vector_t v) -> /* see-below */;
template<class ExecutionPolicy,
         class in_vector_t>
auto vector_abs_sum(ExecutionPolicy&& exec,
                    in_vector_t v) -> /* see-below */;

// [linalg.algs.blas1.iamax],
// index of maximum absolute value of vector elements
template<class in_vector_t>
ptrdiff_t idx_abs_max(in_vector_t v);
template<class ExecutionPolicy,
         class in_vector_t>
ptrdiff_t idx_abs_max(ExecutionPolicy&& exec,
                      in_vector_t v);

// [linalg.algs.blas1.matfrobnorm],
// Frobenius norm of a matrix
template<class in_matrix_t,
         class T>
T matrix_frob_norm(
  in_matrix_t A,
  T init);
template<class ExecutionPolicy,
         class in_matrix_t,
         class T>
T matrix_frob_norm(
  ExecutionPolicy&& exec,
  in_matrix_t A,
  T init);
template<class in_matrix_t>
auto matrix_frob_norm(
  in_matrix_t A) -> /* see-below */;
template<class ExecutionPolicy,
         class in_matrix_t>
auto matrix_frob_norm(
  ExecutionPolicy&& exec,
  in_matrix_t A) -> /* see-below */;

// [linalg.algs.blas1.matonenorm],
// One norm of a matrix
template<class in_matrix_t,
         class T>
T matrix_one_norm(
  in_matrix_t A,
  T init);
template<class ExecutionPolicy,
         class in_matrix_t,
         class T>
T matrix_one_norm(
  ExecutionPolicy&& exec,
  in_matrix_t A,
  T init);
template<class in_matrix_t>
```

```
auto matrix_one_norm(
  in_matrix_t A) -> /* see-below */;
template<class ExecutionPolicy,
         class in_matrix_t>
auto matrix_one_norm(
  ExecutionPolicy&& exec,
  in_matrix_t A) -> /* see-below */;

// [linalg.algs.blas1.matinfnorm],
// Infinity norm of a matrix
template<class in_matrix_t,
         class T>
T matrix_inf_norm(
  in_matrix_t A,
  T init);
template<class ExecutionPolicy,
         class in_matrix_t,
         class T>
T matrix_inf_norm(
  ExecutionPolicy&& exec,
  in_matrix_t A,
  T init);
template<class in_matrix_t>
auto matrix_inf_norm(
  in_matrix_t A) -> /* see-below */;
template<class ExecutionPolicy,
         class in_matrix_t>
auto matrix_inf_norm(
  ExecutionPolicy&& exec,
  in_matrix_t A) -> /* see-below */;

// [linalg.algs.blas2.gemv],
// general matrix-vector product
template<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);
template<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);

// [linalg.algs.blas2.symv],
// symmetric matrix-vector product
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);
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);

// [linalg.algs.blas2.hemv],
// Hermitian matrix-vector product
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);
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);

// [linalg.algs.blas2.trmv],
// Triangular matrix-vector product

// [linalg.algs.blas2.trmv.ov],
// Overwriting triangular matrix-vector product
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);

// [linalg.algs.blas2.trmv.in-place],
// In-place triangular matrix-vector product
template<class in_matrix_t,
          class Triangle,
          class DiagonalStorage,
          class inout_vector_t>
void triangular_matrix_vector_product(
  in_matrix_t A,
  Triangle t,
  DiagonalStorage d,
  inout_vector_t y);

// [linalg.algs.blas2.trmv.up],
// Updating triangular matrix-vector product
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);

// [linalg.algs.blas2.trsv],
// Solve a triangular linear system

// [linalg.algs.blas2.trsv.not-in-place],
// Solve a triangular linear system, not in place
template<class in_matrix_t,
         class Triangle,
         class DiagonalStorage,
         class in_vector_t,
         class out_vector_t>
void triangular_matrix_vector_solve(
  in_matrix_t A,
  Triangle t,
  DiagonalStorage d,
  in_vector_t b,
  out_vector_t x);
template<class ExecutionPolicy,
         class in_matrix_t,
         class Triangle,
         class DiagonalStorage,
         class in_vector_t,
         class out_vector_t>
void triangular_matrix_vector_solve(
  ExecutionPolicy&& exec,
  in_matrix_t A,
  Triangle t,
  DiagonalStorage d,
  in_vector_t b,
  out_vector_t x);

// [linalg.algs.blas2.trsv.in-place],
// Solve a triangular linear system, in place
template<class in_matrix_t,
         class Triangle,
         class DiagonalStorage,
         class inout_vector_t>
void triangular_matrix_vector_solve(
  in_matrix_t A,
  Triangle t,
  DiagonalStorage d,
  inout_vector_t b);
template<class ExecutionPolicy,
         class in_matrix_t,
```

```
         class Triangle,
         class DiagonalStorage,
         class inout_vector_t>
void triangular_matrix_vector_solve(
  ExecutionPolicy&& exec,
  in_matrix_t A,
  Triangle t,
  DiagonalStorage d,
  inout_vector_t b);

// [linalg.algs.blas2.rank1.geru],
// nonconjugated rank-1 matrix update
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);

// [linalg.algs.blas2.rank1.gerc],
// conjugated rank-1 matrix update
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);

// [linalg.algs.blas2.rank1.syr],
// symmetric rank-1 matrix update
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);
  template<class T,
           class in_vector_t,
           class inout_matrix_t,
           class Triangle>
  void symmetric_matrix_rank_1_update(
    T alpha,
    in_vector_t x,
    inout_matrix_t A,
    Triangle t);
  template<class ExecutionPolicy,
           class T,
           class in_vector_t,
           class inout_matrix_t,
           class Triangle>
  void symmetric_matrix_rank_1_update(
    ExecutionPolicy&& exec,
    T alpha,
    in_vector_t x,
    inout_matrix_t A,
    Triangle t);

  // [linalg.algs.blas2.rank1.her],
  // Hermitian rank-1 matrix update
  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);
  template<class T,
           class in_vector_t,
           class inout_matrix_t,
```

```
          class Triangle>
void hermitian_matrix_rank_1_update(
  T alpha,
  in_vector_t x,
  inout_matrix_t A,
  Triangle t);
template<class ExecutionPolicy,
          class T,
          class in_vector_t,
          class inout_matrix_t,
          class Triangle>
void hermitian_matrix_rank_1_update(
  ExecutionPolicy&& exec,
  T alpha,
  in_vector_t x,
  inout_matrix_t A,
  Triangle t);

// [linalg.algs.blas2.rank2.syr2],
// symmetric rank-2 matrix update
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);

// [linalg.algs.blas2.rank2.her2],
// Hermitian rank-2 matrix update
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);

// [linalg.algs.blas3.gemm],
// general matrix-matrix product
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);
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);

// [linalg.algs.blas3.symm],
// symmetric matrix-matrix product

// [linalg.algs.blas3.symm.ov.left],
// overwriting symmetric matrix-matrix left product
template<class in_matrix_1_t,
         class Triangle,
         class in_matrix_2_t,
         class out_matrix_t>
```

```
void symmetric_matrix_left_product(
  in_matrix_1_t A,
  Triangle t,
  in_matrix_2_t B,
  out_matrix_t C);
template<class ExecutionPolicy,
         class in_matrix_1_t,
         class Triangle,
         class in_matrix_2_t,
         class out_matrix_t>
void symmetric_matrix_left_product(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
  in_matrix_2_t B,
  out_matrix_t C);

// [linalg.algs.blas3.symm.ov.right],
// overwriting symmetric matrix-matrix right product
template<class in_matrix_1_t,
         class Triangle,
         class in_matrix_2_t,
         class out_matrix_t>
void symmetric_matrix_right_product(
  in_matrix_1_t A,
  Triangle t,
  in_matrix_2_t B,
  out_matrix_t C);
template<class ExecutionPolicy,
         class in_matrix_1_t,
         class Triangle,
         class in_matrix_2_t,
         class out_matrix_t>
void symmetric_matrix_right_product(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
  in_matrix_2_t B,
  out_matrix_t C);

// [linalg.algs.blas3.symm.up.left],
// updating symmetric matrix-matrix left product
template<class in_matrix_1_t,
         class Triangle,
         class in_matrix_2_t,
         class in_matrix_3_t,
         class out_matrix_t>
void symmetric_matrix_left_product(
  in_matrix_1_t A,
  Triangle t,
  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 in_matrix_2_t,
          class in_matrix_3_t,
          class out_matrix_t>
void symmetric_matrix_left_product(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
  in_matrix_2_t B,
  in_matrix_3_t E,
  out_matrix_t C);

// [linalg.algs.blas3.symm.up.right],
// updating symmetric matrix-matrix right product
template<class in_matrix_1_t,
         class Triangle,
         class in_matrix_2_t,
         class in_matrix_3_t,
         class out_matrix_t>
void symmetric_matrix_right_product(
  in_matrix_1_t A,
  Triangle t,
  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 in_matrix_2_t,
         class in_matrix_3_t,
         class out_matrix_t>
void symmetric_matrix_right_product(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
  in_matrix_2_t B,
  in_matrix_3_t E,
  out_matrix_t C);

// [linalg.algs.blas3.hemm],
// Hermitian matrix-matrix product

// [linalg.algs.blas3.hemm.ov.left],
// overwriting Hermitian matrix-matrix left product
template<class in_matrix_1_t,
         class Triangle,
         class in_matrix_2_t,
         class out_matrix_t>
void hermitian_matrix_left_product(
  in_matrix_1_t A,
  Triangle t,
  in_matrix_2_t B,
  out_matrix_t C);
```

```
template<class ExecutionPolicy,
         class in_matrix_1_t,
         class Triangle,
         class in_matrix_2_t,
         class out_matrix_t>
void hermitian_matrix_left_product(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
  in_matrix_2_t B,
  out_matrix_t C);

// [linalg.algs.blas3.hemm.ov.right],
// overwriting Hermitian matrix-matrix right product
template<class in_matrix_1_t,
         class Triangle,
         class in_matrix_2_t,
         class out_matrix_t>
void hermitian_matrix_right_product(
  in_matrix_1_t A,
  Triangle t,
  in_matrix_2_t B,
  out_matrix_t C);
template<class ExecutionPolicy,
         class in_matrix_1_t,
         class Triangle,
         class in_matrix_2_t,
         class out_matrix_t>
void hermitian_matrix_right_product(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
  in_matrix_2_t B,
  out_matrix_t C);

// [linalg.algs.blas3.hemm.up.left],
// updating Hermitian matrix-matrix left product
template<class in_matrix_1_t,
         class Triangle,
         class in_matrix_2_t,
         class in_matrix_3_t,
         class out_matrix_t>
void hermitian_matrix_left_product(
  in_matrix_1_t A,
  Triangle t,
  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 in_matrix_2_t,
         class in_matrix_3_t,
         class out_matrix_t>
```

```
void hermitian_matrix_left_product(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
  in_matrix_2_t B,
  in_matrix_3_t E,
  out_matrix_t C);

// [linalg.algs.blas3.hemm.up.right],
// updating Hermitian matrix-matrix right product
template<class in_matrix_1_t,
         class Triangle,
         class in_matrix_2_t,
         class in_matrix_3_t,
         class out_matrix_t>
void hermitian_matrix_right_product(
  in_matrix_1_t A,
  Triangle t,
  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 in_matrix_2_t,
         class in_matrix_3_t,
         class out_matrix_t>
void hermitian_matrix_right_product(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
  in_matrix_2_t B,
  in_matrix_3_t E,
  out_matrix_t C);

// [linalg.algs.blas3.trmm],
// triangular matrix-matrix product

// [linalg.algs.blas3.trmm.ov.left],
// overwriting triangular matrix-matrix left product
template<class in_matrix_1_t,
         class Triangle,
         class DiagonalStorage,
         class in_matrix_2_t,
         class out_matrix_t>
void triangular_matrix_left_product(
  in_matrix_1_t A,
  Triangle t,
  DiagonalStorage d,
  in_matrix_2_t B,
  out_matrix_t C);
template<class ExecutionPolicy,
         class in_matrix_1_t,
         class Triangle,
```

```
         class DiagonalStorage,
         class in_matrix_2_t,
         class out_matrix_t>
void triangular_matrix_left_product(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
  DiagonalStorage d,
  in_matrix_2_t B,
  out_matrix_t C);
template<class in_matrix_1_t,
         class Triangle,
         class DiagonalStorage,
         class inout_matrix_t>
void triangular_matrix_left_product(
  in_matrix_1_t A,
  Triangle t,
  DiagonalStorage d,
  inout_matrix_t C);
template<class ExecutionPolicy,
         class in_matrix_1_t,
         class Triangle,
         class DiagonalStorage,
         class inout_matrix_t>
void triangular_matrix_left_product(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
  DiagonalStorage d,
  inout_matrix_t C);

// [linalg.algs.blas3.trmm.ov.right],
// overwriting triangular matrix-matrix right product
template<class in_matrix_1_t,
         class Triangle,
         class DiagonalStorage,
         class in_matrix_2_t,
         class out_matrix_t>
void triangular_matrix_right_product(
  in_matrix_1_t A,
  Triangle t,
  DiagonalStorage d,
  in_matrix_2_t B,
  out_matrix_t C);
template<class ExecutionPolicy,
         class in_matrix_1_t,
         class Triangle,
         class DiagonalStorage,
         class in_matrix_2_t,
         class out_matrix_t>
void triangular_matrix_right_product(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
```

```
  DiagonalStorage d,
  in_matrix_2_t B,
  out_matrix_t C);
template<class in_matrix_1_t,
         class Triangle,
         class DiagonalStorage,
         class inout_matrix_t>
void triangular_matrix_right_product(
  in_matrix_1_t A,
  Triangle t,
  DiagonalStorage d,
  inout_matrix_t C);
template<class ExecutionPolicy,
         class in_matrix_1_t,
         class Triangle,
         class DiagonalStorage,
         class inout_matrix_t>
void triangular_matrix_right_product(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
  DiagonalStorage d,
  inout_matrix_t C);

// [linalg.algs.blas3.trmm.up.left],
// updating triangular matrix-matrix left product
template<class in_matrix_1_t,
         class Triangle,
         class DiagonalStorage,
         class in_matrix_2_t,
         class in_matrix_3_t,
         class out_matrix_t>
void triangular_matrix_left_product(
  in_matrix_1_t A,
  Triangle t,
  DiagonalStorage d,
  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 DiagonalStorage,
         class in_matrix_2_t,
         class in_matrix_3_t,
         class out_matrix_t>
void triangular_matrix_left_product(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
  DiagonalStorage d,
  in_matrix_2_t B,
  in_matrix_3_t E,
  out_matrix_t C);
```

```
// [linalg.algs.blas3.trmm.up.right],
// updating triangular matrix-matrix right product
template<class in_matrix_1_t,
         class Triangle,
         class DiagonalStorage,
         class in_matrix_2_t,
         class in_matrix_3_t,
         class out_matrix_t>
void triangular_matrix_right_product(
  in_matrix_1_t A,
  Triangle t,
  DiagonalStorage d,
  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 DiagonalStorage,
         class in_matrix_2_t,
         class in_matrix_3_t,
         class out_matrix_t>
void triangular_matrix_right_product(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
  DiagonalStorage d,
  in_matrix_2_t B,
  in_matrix_3_t E,
  out_matrix_t C);

// [linalg.alg.blas3.rank-k.syrk],
// rank-k symmetric matrix update
template<class in_matrix_1_t,
         class inout_matrix_t,
         class Triangle>
void symmetric_matrix_rank_k_update(
  in_matrix_1_t A,
  inout_matrix_t C,
  Triangle t);
template<class ExecutionPolicy,
         class in_matrix_1_t,
         class inout_matrix_t,
         class Triangle>
void symmetric_matrix_rank_k_update(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  inout_matrix_t C,
  Triangle t);
template<class T,
         class in_matrix_1_t,
         class inout_matrix_t,
         class Triangle>
```

```
void symmetric_matrix_rank_k_update(
  T alpha,
  in_matrix_1_t A,
  inout_matrix_t C,
  Triangle t);
template<class T,
         class ExecutionPolicy,
         class in_matrix_1_t,
         class inout_matrix_t,
         class Triangle>
void symmetric_matrix_rank_k_update(
  ExecutionPolicy&& exec,
  T alpha,
  in_matrix_1_t A,
  inout_matrix_t C,
  Triangle t);

// [linalg.alg.blas3.rank-k.herk],
// rank-k Hermitian matrix update
template<class in_matrix_1_t,
         class inout_matrix_t,
         class Triangle>
void hermitian_matrix_rank_k_update(
  in_matrix_1_t A,
  inout_matrix_t C,
  Triangle t);
template<class ExecutionPolicy,
         class in_matrix_1_t,
         class inout_matrix_t,
         class Triangle>
void hermitian_matrix_rank_k_update(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  inout_matrix_t C,
  Triangle t);
template<class T,
         class in_matrix_1_t,
         class inout_matrix_t,
         class Triangle>
void hermitian_matrix_rank_k_update(
  T alpha,
  in_matrix_1_t A,
  inout_matrix_t C,
  Triangle t);
template<class ExecutionPolicy,
         class T,
         class in_matrix_1_t,
         class inout_matrix_t,
         class Triangle>
void hermitian_matrix_rank_k_update(
  ExecutionPolicy&& exec,
  T alpha,
  in_matrix_1_t A,
  inout_matrix_t C,
```

```
  Triangle t);

// [linalg.alg.blas3.rank2k.syr2k],
// rank-2k symmetric matrix update
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);

// [linalg.alg.blas3.rank2k.her2k],
// rank-2k Hermitian matrix update
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);

// [linalg.alg.blas3.trsm],
// solve multiple triangular linear systems

// [linalg.alg.blas3.trsm.left],
// solve multiple triangular linear systems
// with triangular matrix on the left
```

```
template<class in_matrix_1_t,
         class Triangle,
         class DiagonalStorage,
         class in_matrix_2_t,
         class out_matrix_t>
void triangular_matrix_matrix_left_solve(
  in_matrix_1_t A,
  Triangle t,
  DiagonalStorage d,
  in_matrix_2_t B,
  out_matrix_t X);
template<class ExecutionPolicy,
         class in_matrix_1_t,
         class Triangle,
         class DiagonalStorage,
         class in_matrix_2_t,
         class out_matrix_t>
void triangular_matrix_matrix_left_solve(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
  DiagonalStorage d,
  in_matrix_2_t B,
  out_matrix_t X);
template<class in_matrix_1_t,
         class Triangle,
         class DiagonalStorage,
         class inout_matrix_t>
void triangular_matrix_matrix_left_solve(
  in_matrix_1_t A,
  Triangle t,
  DiagonalStorage d,
  inout_matrix_t B);
template<class ExecutionPolicy,
         class in_matrix_1_t,
         class Triangle,
         class DiagonalStorage,
         class inout_matrix_t>
void triangular_matrix_matrix_left_solve(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
  DiagonalStorage d,
  inout_matrix_t B);

// [linalg.alg.blas3.trsm.right],
// solve multiple triangular linear systems
// with triangular matrix on the right
template<class in_matrix_1_t,
         class Triangle,
         class DiagonalStorage,
         class in_matrix_2_t,
         class out_matrix_t>
void triangular_matrix_matrix_right_solve(
```

```
    in_matrix_1_t A,
    Triangle t,
    DiagonalStorage d,
    in_matrix_2_t B,
    out_matrix_t X);
  template<class ExecutionPolicy,
           class in_matrix_1_t,
           class Triangle,
           class DiagonalStorage,
           class in_matrix_2_t,
           class out_matrix_t>
  void triangular_matrix_matrix_right_solve(
    ExecutionPolicy&& exec,
    in_matrix_t A,
    Triangle t,
    DiagonalStorage d,
    in_matrix_t B,
    out_matrix_t X);
  template<class in_matrix_1_t,
           class Triangle,
           class DiagonalStorage,
           class inout_matrix_t>
  void triangular_matrix_matrix_right_solve(
    in_matrix_1_t A,
    Triangle t,
    DiagonalStorage d,
    inout_matrix_t B);
  template<class ExecutionPolicy,
           class in_matrix_1_t,
           class Triangle,
           class DiagonalStorage,
           class inout_matrix_t>
  void triangular_matrix_matrix_right_solve(
    ExecutionPolicy&& exec,
    in_matrix_1_t A,
    Triangle t,
    DiagonalStorage d,
    inout_matrix_t B);
}
```

Tag classes [linalg.tags]

Storage order tags [linalg.tags.order]

```
struct column_major_t { };
inline constexpr column_major_t column_major = { };

struct row_major_t { };
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` below.

Triangle tags [linalg.tags.triangle]

Some linear algebra algorithms 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.

```
struct upper_triangle_t { };
inline constexpr upper_triangle_t upper_triangle = { };

struct lower_triangle_t { };
inline constexpr lower_triangle_t lower_triangle = { };
```

These tag classes specify whether algorithms and other users of a matrix (represented as a `basic_mdspan`) 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 [linalg.tags.diagonal]

```
struct implicit_unit_diagonal_t { };
inline constexpr implicit_unit_diagonal_t
  implicit_unit_diagonal = { };

struct explicit_diagonal_t { };
inline constexpr explicit_diagonal_t explicit_diagonal = { };
```

These tag classes specify what algorithms and other users of a matrix 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").

The tag `explicit_diagonal_t` indicates that algorithms and other users of the viewer may access the matrix's diagonal entries directly.

Layouts for general and packed matrix types [linalg.layouts]

`layout_blas_general` [linalg.layouts.general]

`layout_blas_general` is a `basic_mdspan` layout mapping policy. Its `StorageOrder` template parameter determines whether the matrix's data layout is column major or row major.

`layout_blas_general<column_major_t>` represents a column-major matrix layout, where the stride between consecutive rows is always one, and the stride between consecutive columns may be greater than or equal to the number of rows. *[Note:* This is a generalization of `layout_left`. *--end note]*

`layout_blas_general<row_major_t>` represents a row-major matrix layout, where the stride between consecutive rows may be greater than or equal to the number of columns, and the stride between consecutive columns is always one. *[Note:* This is a generalization of `layout_right`. *--end note]*

[Note:

`layout_blas_general` represents exactly the data layout assumed by the General (GE) matrix type in the BLAS' C binding. It has two advantages:

1. Unlike `layout_left` and `layout_right`, any "submatrix" (subspan of consecutive rows and consecutive columns) of a matrix with `layout_blas_general<StorageOrder>` layout also has `layout_blas_general<StorageOrder>` layout.

2. Unlike `layout_stride`, it always has compile-time unit stride in one of the matrix's two extents.

BLAS functions call the possibly nonunit stride of the matrix the "leading dimension" of that matrix. For example, a BLAS function argument corresponding to the leading dimension of the matrix `A` is called `LDA`, for "leading dimension of the matrix A."

--end note]

```
template<class StorageOrder>
class layout_blas_general {
public:
  template<class Extents>
  struct mapping {
  private:
    Extents extents_; // exposition only
    const typename Extents::index_type stride_{}; // exposition only

  public:
    constexpr mapping(const Extents& e,
      const typename Extents::index_type s);

    template<class OtherExtents>
    constexpr mapping(const mapping<OtherExtents>& e) noexcept;

    typename Extents::index_type
    operator() (typename Extents::index_type i,
                typename Extents::index_type j) const;
```

```
    constexpr typename Extents::index_type
    required_span_size() const noexcept;

    typename Extents::index_type
    stride(typename Extents::index_type r) const noexcept;

    template<class OtherExtents>
    bool operator==(const mapping<OtherExtents>& m) const noexcept;

    template<class OtherExtents>
    bool operator!=(const mapping<OtherExtents>& m) const noexcept;

    Extents extents() const noexcept;

    static constexpr bool is_always_unique();
    static constexpr bool is_always_contiguous();
    static constexpr bool is_always_strided();

    constexpr bool is_unique() const noexcept;
    constexpr bool is_contiguous() const noexcept;
    constexpr bool is_strided() const noexcept;
  };
};
```

- *Constraints:*
  - `StorageOrder` is either `column_major_t` or `row_major_t`.
  - `Extents` is a specialization of `extents`.
  - `Extents::rank()` equals 2.

```
constexpr mapping(const Extents& e,
  const typename Extents::index_type s);
```

- *Requires:*
  - If `StorageOrder` is `column_major_t`, then `s` is greater than or equal to `e.extent(0)`. Otherwise, if `StorageOrder` is `row_major_t`, then `s` is greater than or equal to `e.extent(1)`.
- *Effects:* Initializes `extents_` with `e`, and initializes `stride_` with `s`.

[Note:

The BLAS Standard requires that the stride be one if the corresponding matrix dimension is zero. We do not impose this requirement here, because it is specific to the BLAS. if an implementation dispatches to a BLAS function, then the implementation must impose the requirement at run time.

--end note]

```c++
template<class OtherExtents>
constexpr mapping(const mapping<OtherExtents>& e) noexcept;
```

* *Constraints:*

  * `OtherExtents` is a specialization of `extents`.

  * `OtherExtents::rank()` equals 2.

* *Effects:* Initializes `extents_` with `m.extents_`, and initializes `stride_` with `m.stride_`.

```c++
typename Extents::index_type
operator() (typename Extents::index_type i,
            typename Extents::index_type j) const;
```

* *Requires:*

  * 0 ≤ `i` < `extent(0)`, and

  * 0 ≤ `j` < `extent(1)`.

* *Returns:*

  * If `StorageOrder` is `column_major_t`, then `i + stride(1)*j`;

  * else, if `StorageOrder` is `row_major_t`, then `stride(0)*i + j`.

```c++
template<class OtherExtents>
bool operator==(const mapping<OtherExtents>& m) const;
```

* *Constraints:* `OtherExtents::rank()` equals `rank()`.

* *Returns:* `true` if and only if for 0 ≤ `r` < `rank()`, `m.extent(r)` equals `extent(r)` and `m.stride(r)` equals `stride(r)`.

```c++
template<class OtherExtents>
bool operator!=(const mapping<OtherExtents>& m) const;
```

* *Constraints:* `OtherExtents::rank()` equals `rank()`.

* *Returns:* * *Returns:* `true` if and only if there exists `r` with 0 ≤ `r` < `rank()` such that `m.extent(r)` does not equal `extent(r)` or `m.stride(r)` does not equal `stride(r)`.

```c++
typename Extents::index_type
stride(typename Extents::index_type r) const noexcept;
```

* *Returns:*
  * If `StorageOrder` is `column_major_t`, `stride_` if `r` equals 1, else 1;
  * else, if `StorageOrder` is `row_major_t`, `stride_` if `r` equals 0, else 1.

```
constexpr typename Extents::index_type
required_span_size() const noexcept;
```

* *Returns:* `stride(0)*stride(1)`.

```
Extents extents() const noexcept;
```

* *Effects:* Equivalent to `return extents_;`.

```
static constexpr bool is_always_unique();
```

* *Returns:* `true`.

```
static constexpr bool is_always_contiguous();
```

* *Returns:* `false`.

```
static constexpr bool is_always_strided();
```

* *Returns:* `true`.

```
constexpr bool is_unique() const noexcept;
```

* *Returns:* `true`.

```
constexpr bool is_contiguous() const noexcept;
```

* *Returns:*
  * If `StorageOrder` is `column_major_t`, then `true` if `stride(1)` equals `extent(0)`, else `false`;

    - else, if `StorageOrder` is `row_major_t`, then `true` if `stride(0)` equals `extent(1)`, else `false`.

```
constexpr bool is_strided() const noexcept;
```

- *Returns:* `true`.

`layout_blas_packed` [linalg.layouts.packed]

`layout_blas_packed` is a `basic_mdspan` layout mapping policy that represents a square matrix that stores only the entries in one triangle, in a packed contiguous format. Its `Triangle` template parameter determines whether an `basic_mdspan` with this layout stores the upper or lower triangle of the matrix. Its `StorageOrder` template parameter determines whether the layout packs the matrix's elements in column-major or row-major order.

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.

[Note:

`layout_blas_packed` describes the data layout used by the BLAS' Symmetric Packed (SP), Hermitian Packed (HP), and Triangular Packed (TP) matrix types.

If `transposed`'s input has layout `layout_blas_packed`, the return type also has layout `layout_blas_packed`, but with opposite `Triangle` and `StorageOrder`. For example, the transpose of a packed column-major upper triangle, is a packed row-major lower triangle.

--end note]

```
template<class Triangle,
         class StorageOrder>
class layout_blas_packed {
public:
  template<class Extents>
  struct mapping {
  private:
    Extents extents_; // exposition only

  public:
    constexpr mapping(const Extents& e);

    template<class OtherExtents>
    constexpr mapping(const mapping<OtherExtents>& e) noexcept;

    typename Extents::index_type
    operator() (typename Extents::index_type i,
```

```
                 typename Extents::index_type j) const;

    template<class OtherExtents>
    bool operator==(const mapping<OtherExtents>& m) const noexcept;

    template<class OtherExtents>
    bool operator!=(const mapping<OtherExtents>& m) const noexcept;

    constexpr typename Extents::index_type
    stride(typename Extents::index_type r) const noexcept;

    constexpr typename Extents::index_type
    required_span_size() const noexcept;

    constexpr Extents extents() const noexcept;

    static constexpr bool is_always_unique();
    static constexpr bool is_always_contiguous();
    static constexpr bool is_always_strided();

    constexpr bool is_unique() const noexcept;
    constexpr bool is_contiguous() const noexcept;
    constexpr bool is_strided() const noexcept;
};
```

- *Constraints:*
  - `Triangle` is either `upper_triangle_t` or `lower_triangle_t`.
  - `StorageOrder` is either `column_major_t` or `row_major_t`.
  - `Extents` is a specialization of `extents`.
  - `Extents::rank()` equals 2.

```
constexpr mapping(const Extents& e);
```

- *Requires:* `e.extent(0)` equals `e.extent(1)`.
- *Effects:* Initializes `extents_` with `e`.

```
template<class OtherExtents>
constexpr mapping(const mapping<OtherExtents>& e);
```

- *Constraints:*
  - `OtherExtents` is a specialization of `extents`.
  - `OtherExtents::rank()` equals 2.

- *Effects:* Initializes `extents_` with `e`.

```
typename Extents::index_type
operator() (typename Extents::index_type i,
            typename Extents::index_type j) const;
```

- *Requires:*
    - 0 ≤ `i` < `extent(0)`, and
    - 0 ≤ `j` < `extent(1)`.
- *Returns:* Let `N` equal `extent(0)`. Then:
    - If `StorageOrder` is `column_major_t` and
        - if `Triangle` is `upper_triangle_t`, then `i + j(j+1)/2` if `i` >= `j`, else `j + i(i+1)/2`;
        - else, if `Triangle` is `lower_triangle_t`, then `i + Nj - j(j+1)/2` if `i` <= `j`, else `j + Ni - i(i+1)/2`;
    - else, if `StorageOrder` is `row_major_t` and
        - if `Triangle` is `upper_triangle_t`, then `j + Ni - i(i+1)/2` if `j` <= `i`, else `i + Nj - j(j+1)/2`;
        - else, if `Triangle` is `lower_triangle_t`, then `j + i(i+1)/2` if `j` >= `i`, else `i + j(j+1)/2`.

```
template<class OtherExtents>
bool operator==(const mapping<OtherExtents>& m) const;
```

- *Constraints:* `OtherExtents::rank()` equals `rank()`.
- *Returns:* `true` if and only if for 0 ≤ `r` < `rank()`, `m.extent(r)` equals `extent(r)`.

```
template<class OtherExtents>
bool operator!=(const mapping<OtherExtents>& m) const;
```

- *Constraints:* `OtherExtents::rank()` equals `rank()`.
- *Returns:* `true` if and only if there exists `r` with 0 ≤ `r` < `rank()` such that `m.extent(r)` does not equal `extent(r)`.

```
constexpr typename Extents::index_type
stride(typename Extents::index_type r) const noexcept;
```

- *Returns:* 1 if `extent(0)` is less than 2, else 0.

```
constexpr typename Extents::index_type
required_span_size() const noexcept;
```

- *Returns:* `extent(0)*(extent(0) - 1)/2`.

```
constexpr Extents extents() const noexcept;
```

- *Effects:* Equivalent to `return extents_;`.

```
static constexpr bool is_always_unique();
```

- *Returns:* `false`.

```
static constexpr bool is_always_contiguous();
```

- *Returns:* `true`.

```
static constexpr bool is_always_strided();
```

- *Returns:* `false`.

```
constexpr bool is_unique() const noexcept;
```

- *Returns:* `true` if `extent(0)` is less than 2, else `false`.

```
constexpr bool is_contiguous() const noexcept;
```

- *Returns:* `true`.

```
constexpr bool is_strided() const noexcept;
```

- *Returns:* `true` if `extent(0)` is less than 2, else `false`.

Scaled in-place transformation [linalg.scaled]

The `scaled` function takes a value `alpha` and a `basic_mdspan` `x`, and returns a new read-only `basic_mdspan` with the same domain as `x`, that represents the elementwise product of `alpha` with each element of `x`.

[*Example:*

```
// z = alpha * x + y
void z_equals_alpha_times_x_plus_y(
  mdspan<double, extents<dynamic_extent>> z,
  const double alpha,
  mdspan<double, extents<dynamic_extent>> x,
  mdspan<double, extents<dynamic_extent>> y)
{
  add(scaled(alpha, x), y, y);
}

// w = alpha * x + beta * y
void w_equals_alpha_times_x_plus_beta_times_y(
  mdspan<double, extents<dynamic_extent>> w,
  const double alpha,
  mdspan<double, extents<dynamic_extent>> x,
  const double beta,
  mdspan<double, extents<dynamic_extent>> y)
{
  add(scaled(alpha, x), scaled(beta, y), w);
}
```

--*end example*]

[Note:

An implementation could dispatch to a function in the BLAS library, by noticing that the first argument has an `accessor_scaled` `Accessor` type. It could use this information to extract the appropriate run-time value(s) of the relevant BLAS function arguments (e.g., `ALPHA` and/or `BETA`), by calling `accessor_scaled::scaling_factor`.

--end note]

Class template `accessor_scaled` [linalg.scaled.accessor_scaled]

The class template `accessor_scaled` is a `basic_mdspan` accessor policy whose reference type represents the product of a fixed value (the "scaling factor") and its nested `basic_mdspan` accessor's reference. It is part of the implementation of `scaled`.

The exposition-only class template `scaled_scalar` represents a read-only value, which is the product of a fixed value (the "scaling factor") and the value of a reference to an element of a `basic_mdspan`. *[Note:* The value is read only to avoid confusion with the definition of "assigning to a scaled scalar." *--end note]* `scaled_scalar` is part of the implementation of `scaled_accessor`.

```c++
template<class ScalingFactor,
         class Reference>
class scaled_scalar { // exposition only
private:
  const ScalingFactor scaling_factor;
  Reference value;
  using result_type =
    decltype (scaling_factor * value);

public:
  scaled_scalar(const ScalingFactor& s, Reference v);

  operator result_type() const;
};
```

- *Requires:*
  - `ScalingFactor` and `Reference` shall be *Cpp17CopyConstructible*.
- *Constraints:*
  - The expression `scaling_factor * value` is well formed.

```c++
scaled_scalar(const ScalingFactor& s, Reference v);
```

- *Effects:* Initializes `scaling_factor` with `s`, and initializes `value` with `v`.

```c++
operator result_type() const;
```

- *Effects:* Equivalent to `return scaling_factor * value;`.

The class template `accessor_scaled` is a `basic_mdspan` accessor policy whose reference type represents the product of a scaling factor and its nested `basic_mdspan` accessor's reference.

```c++
template<class ScalingFactor,
         class Accessor>
class accessor_scaled {
public:
  using element_type  = Accessor::element_type;
  using pointer       = Accessor::pointer;
  using reference     =
    scaled_scalar<ScalingFactor, Accessor::reference>;
  using offset_policy =
    accessor_scaled<ScalingFactor, Accessor::offset_policy>;

  accessor_scaled(const ScalingFactor& s, Accessor a);
```

```
  reference access(pointer p, ptrdiff_t i) const noexcept;

  offset_policy::pointer
  offset(pointer p, ptrdiff_t i) const noexcept;

  element_type* decay(pointer p) const noexcept;

  ScalingFactor scaling_factor() const;

private:
  const ScalingFactor scaling_factor_; // exposition only
  Accessor accessor; // exposition only
};
```

- *Requires:*
  - `ScalingFactor` and `Accessor` shall be *Cpp17CopyConstructible*.
  - `Accessor` shall meet the `basic_mdspan` accessor policy requirements (see *[mdspan.accessor.reqs]* in P0009).

```
accessor_scaled(const ScalingFactor& s, Accessor a);
```

- *Effects:* Initializes `scaling_factor_` with `s`, and initializes `accessor` with `a`.

```
reference access(pointer p, ptrdiff_t i) const noexcept;
```

- *Effects:* Equivalent to `return reference(scaling_factor_, accessor.access(p, i));`.

```
offset_policy::pointer
offset(pointer p, ptrdiff_t i) const noexcept;
```

- *Effects:* Equivalent to `return accessor.offset(p, i);`.

```
element_type* decay(pointer p) const noexcept;
```

- *Effects:* Equivalent to `return accessor.decay(p);`.

```
ScalingFactor scaling_factor() const;
```

- *Effects:* Equivalent to `return scaling_factor_;`.

`scaled` [linalg.scaled.scaled]

The `scaled` function takes a value `alpha` and a `basic_mdspan` `x`, and returns a new read-only `basic_mdspan` with the same domain as `x`, that represents the elementwise product of `alpha` with each element of `x`.

```
template<class ScalingFactor,
         class ElementType,
         class Extents,
         class Layout,
         class Accessor>
/* see below */
scaled(
  const ScalingFactor& s,
  const basic_mdspan<ElementType, Extents, Layout, Accessor>& a);
```

Let `R` name the type `basic_mdspan<ReturnElementType, Extents, Layout, ReturnAccessor>`, where

- `ReturnElementType` is either `ElementType` or `const ElementType`; and
- `ReturnAccessor` is:
  - if `Accessor` is `accessor_scaled<NestedScalingFactor, NestedAccessor>` for some `NestedScalingFactor` and `NestedAccessor`, then either `accessor_scaled<ProductScalingFactor, NestedAccessor>` or `accessor_scaled<ScalingFactor, Accessor>`, where `ProductScalingFactor` is `decltype(s * a.accessor().scaling_factor())`;
  - else, `accessor_scaled<ScalingFactor, Accessor>`.
- *Effects:*
  - If `Accessor` is `accessor_scaled<NestedScalingFactor, NestedAccessor>` and `ReturnAccessor` is `accessor_scaled<ProductScalingFactor, NestedAccessor>`, then equivalent to `return R(a.data(), a.mapping(), ReturnAccessor(product_s, a.accessor().nested_accessor()));`, where `product_s` equals `s * a.accessor().scaling_factor()`;
  - else, equivalent to `return R(a.data(), a.mapping(), ReturnAccessor(s, a.accessor()));`.
- *Remarks:* The elements of the returned `basic_mdspan` are read only.

[Note:

The point of `ReturnAccessor` is to give implementations freedom to optimize applying `accessor_scaled` twice in a row. However, implementations are not required to optimize arbitrary combinations of nested `accessor_scaled` interspersed with other nested accessors.

The point of `ReturnElementType` is that, based on P0009R10, it may not be possible to deduce the const version of `Accessor` for use in `accessor_scaled`. In general, it may not be correct or efficient to use an

`Accessor` meant for a nonconst `ElementType`, with `const ElementType`. This is because `Accessor::reference` may be a type other than `ElementType&`. Thus, we cannot require that the return type have `const ElementType` as its element type, since that might not be compatible with the given `Accessor`. However, in some cases, like `accessor_basic`, it is possible to deduce the const version of `Accessor`. Regardless, users are not allowed to modify the elements of the returned `basic_mdspan`.

--end note]

[*Example:*

```
void test_scaled(basic_mdspan<double, extents<10>> a)
{
  auto a_scaled = scaled(5.0, a);
  for(int i = 0; i < a.extent(0); ++i) {
    assert(a_scaled(i) == 5.0 * a(i));
  }
}
```

--end example]

Conjugated in-place transformation [linalg.conj]

The `conjugated` function takes a `basic_mdspan` `x`, and returns a new read-only `basic_mdspan` `y` with the same domain as `x`, whose elements are the complex conjugates of the corresponding elements of `x`. If the element type of `x` is not `complex<R>` for some `R`, then `y` is a read-only view of the elements of `x`.

[Note:

An implementation could dispatch to a function in the BLAS library, by noticing that the `Accessor` type of a `basic_mdspan` input has type `accessor_conjugate`, and that its nested `Accessor` type is compatible with the BLAS library. If so, it could set the corresponding `TRANS*` BLAS function argument accordingly and call the BLAS function.

--end note]

Class template `accessor_conjugate` [linalg.conj.accessor_conjugate]

The class template `accessor_conjugate` is a `basic_mdspan` accessor policy whose reference type represents the complex conjugate of its nested `basic_mdspan` accessor's reference.

The exposition-only class template `conjugated_scalar` represents a read-only value, which is the complex conjugate of the value of a reference to an element of a `basic_mdspan`. *[Note:* The value is read only to avoid confusion with the definition of "assigning to the conjugate of a scalar." *--end note]* `conjugated_scalar` is part of the implementation of `accessor_conjugate`.

```
template<class Reference,
         class ElementType>
class conjugated_scalar { // exposition only
public:
```

```
  conjugated_scalar(Reference v);

  operator ElementType() const;

private:
  Reference val;
};
```

- *Requires:* `Reference` shall be *Cpp17CopyConstructible*.
- *Constraints:*
  - The expression `conj(val)` is well formed and is convertible to `ElementType`. *[Note:* This implies that `ElementType` is `complex<R>` for some type `R`. *--end note]*

```
conjugated_scalar(Reference v);
```

- *Effects:* Initializes `val` with `v`.

```
operator T() const;
```

- *Effects:* Equivalent to `return conj(val);`.

```
template<class Accessor>
class accessor_conjugate {
private:
  Accessor acc; // exposition only

public:
  using element_type  = typename Accessor::element_type;
  using pointer       = typename Accessor::pointer;
  using reference     = /* see below */;
  using offset_policy = /* see below */;

  accessor_conjugate(Accessor a);

  reference access(pointer p, ptrdiff_t i) const
    noexcept(noexcept(reference(acc.access(p, i))));

  typename offset_policy::pointer
  offset(pointer p, ptrdiff_t i) const
    noexcept(noexcept(acc.offset(p, i)));

  element_type* decay(pointer p) const
    noexcept(noexcept(acc.decay(p)));
```

```
  Accessor nested_accessor() const;
};
```

- *Requires:*
  - `Accessor` shall be *Cpp17CopyConstructible*.
  - `Accessor` shall meet the `basic_mdspan` accessor policy requirements (see *[mdspan.accessor.reqs]* in P0009R10).

```
using reference = /* see below */;
```

If `element_type` is `complex<R>` for some `R`, then this names `conjugated_scalar<typename Accessor::reference, element_type>`. Otherwise, it names `typename Accessor::reference`.

```
using offset_policy = /* see below */;
```

If `element_type` is `complex<R>` for some `R`, then this names `accessor_conjugate<typename Accessor::offset_policy, element_type>`. Otherwise, it names `typename Accessor::offset_policy>`.

```
accessor_conjugate(Accessor a);
```

- *Effects:* Initializes `acc` with `a`.

```
reference access(pointer p, ptrdiff_t i) const
  noexcept(noexcept(reference(acc.access(p, i))));
```

- *Effects:* Equivalent to `return reference(acc.access(p, i));`.

```
typename offset_policy::pointer
offset(pointer p, ptrdiff_t i) const
  noexcept(noexcept(acc.offset(p, i)));
```

- *Effects:* Equivalent to `return acc.offset(p, i);`.

```
element_type* decay(pointer p) const
  noexcept(noexcept(acc.decay(p)));
```

- *Effects:* Equivalent to `return acc.decay(p);`.

```
Accessor nested_accessor() const;
```

- *Effects:* Equivalent to `return acc;`.

`conjugated` [linalg.conj.conjugated]

```
template<class ElementType,
         class Extents,
         class Layout,
         class Accessor>
/* see-below */
conjugated(
  basic_mdspan<ElementType, Extents, Layout, Accessor> a);
```

Let `R` name the type `basic_mdspan<ReturnElementType, Extents, Layout, ReturnAccessor>`, where

- `ReturnElementType` is either `ElementType` or `const ElementType`; and
- `ReturnAccessor` is:
  - if `Accessor` is `accessor_conjugate<NestedAccessor>` for some `NestedAccessor`, then either `NestedAccessor` or `accessor_conjugate<Accessor>`,
  - else if `ElementType` is `complex<U>` or `const complex<U>` for some `U`, then `accessor_conjugate<Accessor>`,
  - else either `accessor_conjugate<Accessor>` or `Accessor`.
- *Effects:*
  - If `Accessor` is `accessor_conjugate<NestedAccessor>` and `ReturnAccessor` is `NestedAccessor`, then equivalent to `return R(a.data(), a.mapping(), a.nested_accessor());`;
  - else, if `ReturnAccessor` is `accessor_conjugate<Accessor>`, then equivalent to `return R(a.data(), a.mapping(), accessor_conjugate<Accessor>(a.accessor()));`;
  - else, equivalent to `return R(a.data(), a.mapping(), a.accessor());`.
- *Remarks:* The elements of the returned `basic_mdspan` are read only.

[Note:

The point of `ReturnAccessor` is to give implementations freedom to optimize applying `accessor_conjugate` twice in a row. However, implementations are not required to optimize arbitrary combinations of nested `accessor_conjugate` interspersed with other nested accessors.

--end note]

[*Example:*

```
void test_conjugated_complex(
  basic_mdspan<complex<double>, extents<10>> a)
{
  auto a_conj = conjugated(a);
  for(int i = 0; i < a.extent(0); ++i) {
    assert(a_conj(i) == conj(a(i));
  }
  auto a_conj_conj = conjugated(a_conj);
  for(int i = 0; i < a.extent(0); ++i) {
    assert(a_conj_conj(i) == a(i));
  }
}

void test_conjugated_real(
  basic_mdspan<double, extents<10>> a)
{
  auto a_conj = conjugated(a);
  for(int i = 0; i < a.extent(0); ++i) {
    assert(a_conj(i) == a(i));
  }
  auto a_conj_conj = conjugated(a_conj);
  for(int i = 0; i < a.extent(0); ++i) {
    assert(a_conj_conj(i) == a(i));
  }
}
```

--end example]

Transpose in-place transformation [linalg.transp]

`layout_transpose` is a `basic_mdspan` layout mapping policy that swaps the rightmost two indices, extents, and strides (if applicable) of any unique `basic_mdspan` layout mapping policy.

The `transposed` function takes a rank-2 `basic_mdspan` representing a matrix, and returns a new read-only `basic_mdspan` representing the transpose of the input matrix.

[Note:

An implementation could dispatch to a function in the BLAS library, by noticing that the first argument has a `layout_transpose` `Layout` type, and/or an `accessor_conjugate` (see below) `Accessor` type. It could use this information to extract the appropriate run-time value(s) of the relevant `TRANS*` BLAS function arguments.

--end note]

`layout_transpose` [linalg.transp.layout_transpose]

`layout_transpose` is a `basic_mdspan` layout mapping policy that swaps the rightmost two indices, extents, and strides (if applicable) of any unique `basic_mdspan` layout mapping policy.

```
template<class InputExtents>
using transpose_extents_t = /* see below */; // exposition only
```

For `InputExtents` a specialization of `extents`, `transpose_extents_t<InputExtents>` names the `extents` type `OutputExtents` such that

- `InputExtents::static_extent(InputExtents::rank()-1)` equals `OutputExtents::static_extent(OutputExtents::rank()-2)`,

- `InputExtents::static_extent(InputExtents::rank()-2)` equals `OutputExtents::static_extent(OutputExtents::rank()-1)`, and

- `InputExtents::static_extent(r)` equals `OutputExtents::static_extent(r)` for 0 ≤ `r` < `InputExtents::rank()-2`.

- *Requires:* `InputExtents` is a specialization of `extents`.

- *Constraints:* `InputExtents::rank()` is at least 2.

```
template<class InputExtents>
transpose_extents_t<InputExtents>
transpose_extents(const InputExtents in); // exposition only
```

- *Constraints:* `InputExtents::rank()` is at least 2.

- *Returns:* An `extents` object `out` such that

  - `out.extent(in.rank()-1)` equals `in.extent(in.rank()-2)`,

  - `out.extent(in.rank()-2)` equals `in.extent(in.rank()-1)`, and

  - `out.extent(r)` equals `in.extent(r)` for 0 ≤ `r` < `in.rank()-2`.

```
template<class Layout>
class layout_transpose {
public:
  template<class Extents>
  struct mapping {
  private:
    using nested_mapping_type =
      typename Layout::template mapping<
        transpose_extents_t<Extents>>; // exposition only
    nested_mapping_type nested_mapping_; // exposition only

  public:
```

```
    mapping(const nested_mapping_type& map);

    ptrdiff_t operator() (ptrdiff_t i, ptrdiff_t j) const
      noexcept(noexcept(nested_mapping_(j, i)));

    nested_mapping_type nested_mapping() const;

    template<class OtherExtents>
    bool operator==(const mapping<OtherExtents>& m) const;

    template<class OtherExtents>
    bool operator!=(const mapping<OtherExtents>& m) const;

    Extents extents() const noexcept;

    typename Extents::index_type required_span_size() const
      noexcept(noexcept(nested_mapping_.required_span_size()));

    bool is_unique() const
      noexcept(noexcept(nested_mapping_.is_unique()));

    bool is_contiguous() const
      noexcept(noexcept(nested_mapping_.is_contiguous()));

    bool is_strided() const
      noexcept(noexcept(nested_mapping_.is_strided()));

    static constexpr bool is_always_unique();

    static constexpr bool is_always_contiguous();

    static constexpr bool is_always_strided();

    typename Extents::index_type
    stride(typename Extents::index_type r) const
      noexcept(noexcept(nested_mapping_.stride(r)));
  };
};
```

- *Requires:*
  - `Layout` shall meet the `basic_mdspan` layout mapping policy requirements. *[Note:* See *[mdspan.layout.reqs]* in P0009R10. *--end note]*
- *Constraints:*
  - For all specializations `E` of `extents` with `E::rank()` equal to 2, `typename Layout::template mapping<E>::is_always_unique()` is `true`.

```
mapping(const nested_mapping_type& map);
```

- *Effects:* Initializes `nested_mapping_` with `map`.

```
ptrdiff_t operator() (ptrdiff_t i, ptrdiff_t j) const
  noexcept(noexcept(nested_mapping_(j, i)));
```

- *Effects:* Equivalent to `return nested_mapping_(j, i);`.

```
nested_mapping_type nested_mapping() const;
```

- *Effects:* Equivalent to `return nested_mapping_;`.

```
template<class OtherExtents>
bool operator==(const mapping<OtherExtents>& m) const;
```

- *Constraints:* `OtherExtents::rank()` equals `rank()`.

- *Effects:* Equivalent to `nested_mapping_ == m.nested_mapping_;`.

```
template<class OtherExtents>
bool operator!=(const mapping<OtherExtents>& m) const;
```

- *Constraints:* `OtherExtents::rank()` equals `rank()`.

- *Effects:* Equivalent to `nested_mapping_ != m.nested_mapping_;`.

```
Extents extents() const noexcept;
```

- *Effects:* Equivalent to `return transpose_extents(nested_mapping_.extents());`.

```
typename Extents::index_type
required_span_size() const
  noexcept(noexcept(nested_mapping_.required_span_size()));
```

- *Effects:* Equivalent to `return nested_mapping_.required_span_size();'.

```
bool is_unique() const
  noexcept(noexcept(nested_mapping_.is_unique()));
```

- *Effects:* Equivalent to `return nested_mapping_.is_unique();'.

```
bool is_contiguous() const
  noexcept(noexcept(nested_mapping_.is_contiguous()));
```

- *Effects:* Equivalent to `return nested_mapping_.is_contiguous();'.

```
bool is_strided() const
  noexcept(noexcept(nested_mapping_.is_strided()));
```

- *Effects:* Equivalent to `return nested_mapping_.is_strided();'.

```
static constexpr bool is_always_unique();
```

- *Effects:* Equivalent to `return nested_mapping_type::is_always_unique();'.

```
static constexpr bool is_always_contiguous();
```

- *Effects:* Equivalent to `return nested_mapping_type::is_always_contiguous();'.

```
static constexpr bool is_always_strided();
```

- *Effects:* Equivalent to `return nested_mapping_type::is_always_strided();'.

```
typename Extents::index_type
stride(typename Extents::index_type r) const
  noexcept(noexcept(nested_mapping_.stride(r)));
```

- *Constraints:* `is_always_strided()` is `true`.

- *Effects:* Equivalent to `return nested_mapping_.stride(s);', where `s`is 0 if`r`is 1 and`s`is 1 if`r` is 0.

`transposed` [linalg.transp.transposed]

The `transposed` function takes a rank-2 `basic_mdspan` representing a matrix, and returns a new read-only `basic_mdspan` representing the transpose of the input matrix. The input matrix's data are not modified, and the returned `basic_mdspan` accesses the input matrix's data in place. If the input `basic_mdspan`'s layout is already `layout_transpose<L>` for some layout `L`, then the returned `basic_mdspan` has layout `L`. Otherwise, the returned `basic_mdspan` has layout `layout_transpose<L>`, where `L` is the input `basic_mdspan`'s layout.

```
template<class ElementType,
         class Extents,
         class Layout,
         class Accessor>
/* see-below */
transposed(
  basic_mdspan<ElementType, Extents, Layout, Accessor> a);
```

Let `ReturnExtents` name the type `transpose_extents_t<Extents>`. Let `R` name the type `basic_mdspan<ReturnElementType, ReturnExtents, ReturnLayout, Accessor>`, where

- `ReturnElementType` is either `ElementType` or `const ElementType`; and
- `ReturnLayout` is:
  - if `Layout` is `layout_blas_packed<Triangle, StorageOrder>`, then `layout_blas_packed<OppositeTriangle, OppositeStorageOrder>`, where
    - `OppositeTriangle` names the type `conditional_t<is_same_v<Triangle, upper_triangle_t>, lower_triangle_t, upper_triangle_t>`, and
    - `OppositeStorageOrder` names the type `conditional_t<is_same_v<StorageOrder, column_major_t>, row_major_t, column_major_t>`;
  - else, if `Layout` is `layout_transpose<NestedLayout>` for some `NestedLayout`, then either `NestedLayout` or `layout_transpose<Layout>`,
  - else `layout_transpose<Layout>`.
- *Effects:*
  - If `Layout` is `layout_blas_packed<Triangle, StorageOrder>`, then equivalent to `return R(a.data(), ReturnMapping(a.mapping().extents()), a.accessor());`;
  - else, if `Layout` is `layout_transpose<NestedLayout>` and `ReturnLayout` is `NestedLayout`, then equivalent to `return R(a.data(), a.mapping().nested_mapping(), a.accessor());`;
  - else, equivalent to `return R(a.data(), ReturnMapping(a.mapping()), a.accessor);`, where `ReturnMapping` names the type `typename layout_transpose<Layout>::template mapping<ReturnExtents>`.
- *Remarks:* The elements of the returned `basic_mdspan` are read only.

[Note:

Implementations may optimize applying `layout_transpose` twice in a row. However, implementations need not optimize arbitrary combinations of nested `layout_transpose` interspersed with other nested layouts.

--end note]

[Example:

```
void test_transposed(basic_mdspan<double, extents<3, 4>> a)
{
  const ptrdiff_t num_rows = a.extent(0);
  const ptrdiff_t num_cols = a.extent(1);

  auto a_t = transposed(a);
  assert(num_rows == a_t.extent(1));
  assert(num_cols == a_t.extent(0));
  assert(a.stride(0) == a_t.stride(1));
  assert(a.stride(1) == a_t.stride(0));

  for(ptrdiff_t row = 0; row < num_rows; ++row) {
    for(ptrdiff_t col = 0; col < num_rows; ++col) {
      assert(a(row, col) == a_t(col, row));
    }
  }

  auto a_t_t = transposed(a_t);
  assert(num_rows == a_t_t.extent(0));
  assert(num_cols == a_t_t.extent(1));
  assert(a.stride(0) == a_t_t.stride(0));
  assert(a.stride(1) == a_t_t.stride(1));

  for(ptrdiff_t row = 0; row < num_rows; ++row) {
    for(ptrdiff_t col = 0; col < num_rows; ++col) {
      assert(a(row, col) == a_t_t(row, col));
    }
  }
}
```

--*end example*]

Conjugate transpose transform [linalg.conj_transp]

The `conjugate_transposed` function returns a conjugate transpose view of an object. This combines the effects of `transposed` and `conjugated`.

```
template<class ElementType,
         class Extents,
         class Layout,
         class Accessor>
/* see-below */
conjugate_transposed(
  basic_mdspan<ElementType, Extents, Layout, Accessor> a);
```

- *Effects:* Equivalent to `return conjugated(transposed(a));`.
- *Remarks:* The elements of the returned `basic_mdspan` are read only.

[*Example:*

```
void test_conjugate_transposed(
  basic_mdspan<complex<double>, extents<3, 4>> a)
{
  const ptrdiff_t num_rows = a.extent(0);
  const ptrdiff_t num_cols = a.extent(1);

  auto a_ct = conjugate_transposed(a);
  assert(num_rows == a_ct.extent(1));
  assert(num_cols == a_ct.extent(0));
  assert(a.stride(0) == a_ct.stride(1));
  assert(a.stride(1) == a_ct.stride(0));

  for(ptrdiff_t row = 0; row < num_rows; ++row) {
    for(ptrdiff_t col = 0; col < num_rows; ++col) {
      assert(a(row, col) == conj(a_ct(col, row)));
    }
  }

  auto a_ct_ct = conjugate_transposed(a_ct);
  assert(num_rows == a_ct_ct.extent(0));
  assert(num_cols == a_ct_ct.extent(1));
  assert(a.stride(0) == a_ct_ct.stride(0));
  assert(a.stride(1) == a_ct_ct.stride(1));

  for(ptrdiff_t row = 0; row < num_rows; ++row) {
    for(ptrdiff_t col = 0; col < num_rows; ++col) {
      assert(a(row, col) == a_ct_ct(row, col));
      assert(conj(a_ct(col, row)) == a_ct_ct(row, col));
    }
  }
}
```

--end example]

Algorithms [linalg.algs]

Requirements [linalg.algs.reqs]

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_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_mdspan` with a non-`const` element type and a unique layout.
- `out_vector*_t` is a rank-1 `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_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_mdspan` with a non-`const` element type.
- `out_matrix*_t` is a rank-2 `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_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_mdspan` with a non-`const` element type and a unique layout.
- `out_object*_t` is a rank-1 or rank-2 `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`.
- `in_*_t` template parameters may deduce a `const` lvalue reference or a (non-`const`) rvalue reference to a `basic_mdspan`.
- `inout_*_t` and `out_*_t` template parameters may deduce a `const` lvalue reference to a `basic_mdspan`, or a (non-`const`) rvalue reference to a `basic_mdspan`.

BLAS 1 functions [linalg.algs.blas1]

[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 naïve 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 [linalg.algs.blas1.givens]

Compute Givens rotation [linalg.algs.blas1.givens.lartg]

```
template<class Real>
void givens_rotation_setup(const Real a,
                           const Real b,
                           Real& c,
                           Real& s,
                           Real& r);

template<class Real>
void givens_rotation_setup(const complex<Real>& a,
                           const complex<Real>& a,
                           Real& c,
                           complex<Real>& s,
                           complex<Real>& r);
```

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 expression uses `conj` as mathematical notation, not as code. *--end note]*

[Note: This function corresponds to the LAPACK function `xLARTG`. The BLAS variant `xROTG` takes four arguments -- `a`, `b`, `c`, and `s`-- and overwrites the input `a` with `r`. We have chosen `xLARTG`'s interface because it separates input and output, and to encourage following `xLARTG`'s more careful implementation. *--end note]*

[Note: `givens_rotation_setup` 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`. Assigns to `r` the Euclidean norm of the two-component vector formed by `a` and `b`.
- *Throws:* Nothing.

Apply a computed Givens rotation to vectors [linalg.algs.blas1.givens.rot]

```
template<class inout_vector_1_t,
         class inout_vector_2_t,
         class Real>
```

```
void givens_rotation_apply(
  inout_vector_1_t x,
  inout_vector_2_t y,
  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 x,
  inout_vector_2_t y,
  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 x,
  inout_vector_2_t y,
  const Real c,
  const complex<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 x,
  inout_vector_2_t y,
  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:* `x.extent(0)` equals `y.extent(0)`.
- *Constraints:*
  - `Real` is `float`, `double`, or `long double`.
  - For the overloads that take the last argument `s` as `Real`, for `i` in the domain of `x` and `j` in the domain of `y`, the expressions `x(i) = c*x(i) + s*y(j)` and `y(j) = c*y(j) - s*x(i)` are well formed.

- For the overloads that take the last argument `s` as `const complex<Real>`, for `i` in the domain of `x` and `j` in the domain of `y`, the expressions `x(i) = c*x(i) + s*y(j)` and `y(j) = c*y(j) - conj(s)*x(i)` are well formed.

- *Mandates:* If neither `x.static_extent(0)` nor `y.static_extent(0)` equals `dynamic_extent`, then `x.static_extent(0)` equals `y.static_extent(0)`.

- *Effects:* Applies the plane (Givens) rotation specified by `c` and `s` to the input vectors `x` and `y`, 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 [linalg.algs.blas1.swap]

```
template<class inout_object_1_t,
         class inout_object_2_t>
void swap_elements(inout_object_1_t x,
                   inout_object_2_t y);

template<class ExecutionPolicy,
         class inout_object_1_t,
         class inout_object_2_t>
void swap_elements(ExecutionPolicy&& exec,
                   inout_object_1_t x,
                   inout_object_2_t y);
```

[Note: These functions correspond to the BLAS function `xSWAP`. *--end note]*

- *Requires:* For all `r` in 0, 1, ..., `x.rank()` - 1, `x.extent(r)` equals `y.extent(r)`.

- *Constraints:*

  - `x.rank()` equals `y.rank()`.

  - `x.rank()` is no more than 2.

  - For `i...` in the domain of `x` and `y`, the expression `x(i...) = y(i...)` is well formed.

- *Mandates:* For all `r` in 0, 1, ..., `x.rank()` - 1, if neither `x.static_extent(r)` nor `y.static_extent(r)` equals `dynamic_extent`, then `x.static_extent(r)` equals `y.static_extent(r)`.

- *Effects:* Swap all corresponding elements of the objects `x` and `y`.

Multiply the elements of an object in place by a scalar [linalg.algs.blas1.scal]

```
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 [linalg.algs.blas1.copy]

```
template<class in_object_t,
         class out_object_t>
void copy(in_object_t x,
          out_object_t y);

template<class ExecutionPolicy,
         class in_object_t,
         class out_object_t>
void copy(ExecutionPolicy&& exec,
          in_object_t x,
          out_object_t y);
```

[Note: These functions correspond to the BLAS function `xCOPY`. --*end note]*

- *Requires:* For all `r` in 0, 1, ..., `x.rank()` - 1, `x.extent(r)` equals `y.extent(r)`.
- *Constraints:*
  - `x.rank()` equals `y.rank()`.
  - For all `i...` in the domain of `x` and `y`, the expression `y(i...) = x(i...)` is well formed.
- *Mandates:* For all `r` in 0, 1, ..., `x.rank()` - 1, if neither `x.static_extent(r)` nor `y.static_extent(r)` equals `dynamic_extent`, then `x.static_extent(r)` equals `y.static_extent(r)`.
- *Effects:* Overwrite each element of `y` with the corresponding element of `x`.

Add vectors or matrices elementwise [linalg.algs.blas1.add]

```
template<class in_object_1_t,
         class in_object_2_t,
```

```
         class out_object_t>
void 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 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:* For all `r` in 0, 1, ..., `x.rank()` - 1,
  - `x.extent(r)` equals `z.extent(r)`.
  - `y.extent(r)` equals `z.extent(r)`.
- *Constraints:*
  - `x.rank()`, `y.rank()`, and `z.rank()` are all equal.
  - For `i...` in the domain of `x`, `y`, and `z`, the expression `z(i...) = x(i...) + y(i...)` is well formed.
- *Mandates:* For all `r` in 0, 1, ..., `x.rank()` - 1,
  - if neither `x.static_extent(r)` nor `z.static_extent(r)` equals `dynamic_extent`, then `x.static_extent(r)` equals `z.static_extent(r)`; and
  - if neither `y.static_extent(r)` nor `z.static_extent(r)` equals `dynamic_extent`, then `y.static_extent(r)` equals `z.static_extent(r)`.
  - if neither `x.static_extent(r)` nor `y.static_extent(r)` equals `dynamic_extent`, then `x.static_extent(r)` equals `y.static_extent(r)`;
- *Effects*: Compute the elementwise sum z = x + y.

Dot product of two vectors [linalg.algs.blas1.dot]

Nonconjugated dot product of two vectors [linalg.algs.blas1.dot.dotu]

[Note: The functions in this section correspond to the BLAS functions `xDOT` (for real element types) and `xDOTU` (for complex element types). --*end note]*

Nonconjugated dot product with specified result type

```
template<class in_vector_1_t,
         class in_vector_2_t,
         class T>
T dot(in_vector_1_t v1,
      in_vector_2_t v2,
      T init);
template<class ExecutionPolicy,
         class in_vector_1_t,
         class in_vector_2_t,
         class T>
T dot(ExecutionPolicy&& exec,
      in_vector_1_t v1,
      in_vector_2_t v2,
      T init);
```

- *Requires:*
  - `T` shall be *Cpp17MoveConstructible*.
  - `init + v1(0)*v2(0)` shall be convertible to `T`.
  - `v1.extent(0)` equals `v2.extent(0)`.
- *Constraints:* For all `i` in the domain of `v1` and `v2` and for `val` of type `T&`, the expression `val += v1(i)*v2(i)` is well formed.
- *Mandates:* If neither `v1.static_extent(0)` nor `v2.static_extent(0)` equals `dynamic_extent`, then `v1.static_extent(0)` equals `v2.static_extent(0)`.
- *Effects:* Let `N` be `v1.extent(0)`. If `N` is zero, returns `init`, else returns /GENERALIZED_SUM/(`plus<>()`, `init`, `v1(0)*v2(0)`, ..., `v1(N-1)*v2(N-1)`).
- *Remarks:* If `in_vector_t::element_type` and `T` are both floating-point types or complex versions thereof, and if `T` has higher precision than `in_vector_type::element_type`, then intermediate terms in the sum use `T`'s precision or greater.

[Note: Like `reduce`, `dot` applies binary `operator+` in an unspecified order. This may yield a nondeterministic result for non-associative or non-commutative `operator+` such as floating-point addition. However, implementations may perform extra work to make the result deterministic. They may do so for all `dot` overloads, or just for specific `ExecutionPolicy` types. *--end note]*

[Note: Users can get `xDOTC` behavior by giving the second argument as the result of `conjugated`. Alternately, they can use the shortcut `dotc` below. *--end note]*

Nonconjugated dot product with default result type

```
template<class in_vector_1_t,
         class in_vector_2_t>
auto dot(in_vector_1_t v1,
         in_vector_2_t v2) -> /* see-below */;
```

```
template<class ExecutionPolicy,
         class in_vector_1_t,
         class in_vector_2_t>
auto dot(ExecutionPolicy&& exec,
         in_vector_1_t v1,
         in_vector_2_t v2) -> /* see-below */;
```

- *Effects:* Let `T` be `decltype(v1(0)*v2(0))`. Then, the two-parameter overload is equivalent to `dot(v1, v2, T{});`, and the three-parameter overload is equivalent to `dot(exec, v1, v2, T{});`.

Conjugated dot product of two vectors [linalg.algs.blas1.dot.dotc]

[Note:

The functions in this section correspond to the BLAS functions `xDOT` (for real element types) and `xDOTC` (for complex element types).

`dotc` exists to give users reasonable default inner product behavior for both real and complex element types.

--end note]

Conjugated dot product with specified result type

```
template<class in_vector_1_t,
         class in_vector_2_t,
         class T>
T dotc(in_vector_1_t v1,
       in_vector_2_t v2,
       T init);
template<class ExecutionPolicy,
         class in_vector_1_t,
         class in_vector_2_t,
         class T>
T dotc(ExecutionPolicy&& exec,
       in_vector_1_t v1,
       in_vector_2_t v2,
       T init);
```

- *Effects:* The three-argument overload is equivalent to `dot(v1, conjugated(v2), init);`. The four-argument overload is equivalent to `dot(exec, v1, conjugated(v2), init);`.

Conjugated dot product with default result type

```
template<class in_vector_1_t,
         class in_vector_2_t>
auto dotc(in_vector_1_t v1,
          in_vector_2_t v2) -> /* see-below */;
template<class ExecutionPolicy,
         class in_vector_1_t,
```

```
         class in_vector_2_t>
auto dotc(ExecutionPolicy&& exec,
          in_vector_1_t v1,
          in_vector_2_t v2) -> /* see-below */;
```

- *Effects:* If `in_vector_2_t::element_type` is `complex<R>` for some `R`, let `T` be `decltype(v1(0)*conj(v2(0)))`; else, let `T` be `decltype(v1(0)*v2(0))`. Then, the two-parameter overload is equivalent to `dotc(v1, v2, T{});`, and the three-parameter overload is equivalent to `dotc(exec, v1, v2, T{});`.

Scaled sum of squares of a vector's elements [linalg.algs.blas1.ssq]

```
template<class T>
struct sum_of_squares_result {
  T scaling_factor;
  T scaled_sum_of_squares;
};
template<class in_vector_t,
         class T>
sum_of_squares_result<T> vector_sum_of_squares(
  in_vector_t v,
  sum_of_squares_result init);
template<class ExecutionPolicy,
         class in_vector_t,
         class T>
sum_of_squares_result<T> vector_sum_of_squares(
  ExecutionPolicy&& exec,
  in_vector_t v,
  sum_of_squares_result init);
```

[Note: These functions correspond to the LAPACK function `xLASSQ`. *--end note]*

- *Requires:*
  - `T` shall be *Cpp17MoveConstructible* and *Cpp17LessThanComparable*.
  - `abs(x(0))` shall be convertible to `T`.
- *Constraints:* For all `i` in the domain of `v`, and for `absxi`, `f`, and `ssq` of type `T`, the expression `ssq = ssq + (absxi / f)*(absxi / f)` is well formed.
- *Effects:* Returns two values:
  - `scaling_factor`: the maximum of `init.scaling_factor` and `abs(x(i))` for all `i` in the domain of `v`; and
  - `scaled_sum_of_squares`: a value such that `scaling_factor * scaling_factor * scaled_sum_of_squares` equals the sum of squares of `abs(x(i))` plus `init.scaling_factor * init.scaling_factor * init.scaled_sum_of_squares`.

- *Remarks:* If `in_vector_t::element_type` is a floating-point type or a complex version thereof, and if `T` is a floating-point type, then
  - if `T` has higher precision than `in_vector_type::element_type`, then intermediate terms in the sum use `T`'s precision or greater; and
  - any guarantees regarding overflow and underflow of `vector_sum_of_squares` are implementation-defined.

Euclidean norm of a vector [linalg.algs.blas1.nrm2]

Euclidean norm with specified result type

```
template<class in_vector_t,
         class T>
T vector_norm2(in_vector_t v,
               T init);
template<class ExecutionPolicy,
         class in_vector_t,
         class T>
T vector_norm2(ExecutionPolicy&& exec,
               in_vector_t v,
               T init);
```

[Note: These functions correspond to the BLAS function `xNRM2`. *--end note]*

- *Requires:*
  - `T` shall be *Cpp17MoveConstructible*.
  - `init + abs(v(0))*abs(v(0))` shall be convertible to `T`.
- *Constraints:* For all `i` in the domain of `v` and for `val` of type `T&`, the expressions `val += abs(v(i))*abs(v(i))` and `sqrt(val)` are well formed. *[Note:* This does not imply a recommended implementation for floating-point types. See *Remarks* below. *--end note]*
- *Effects:* Returns the Euclidean norm (also called 2-norm) of the vector `v`.
- *Remarks:* If `in_vector_t::element_type` is a floating-point type or a complex version thereof, and if `T` is a floating-point type, then
  - if `T` has higher precision than `in_vector_type::element_type`, then intermediate terms in the sum use `T`'s precision or greater; and
  - any guarantees regarding overflow and underflow of `vector_norm2` are implementation-defined.

[Note: A suggested implementation of this function for floating-point types `T`, is to return the `scaled_sum_of_squares` result from `vector_sum_of_squares(x, {0.0, 1.0})`. *--end note]*

Euclidean norm with default result type

```
template<class in_vector_t>
auto vector_norm2(in_vector_t v) -> /* see-below */;
template<class ExecutionPolicy,
         class in_vector_t>
auto vector_norm2(ExecutionPolicy&& exec,
                  in_vector_t v) -> /* see-below */;
```

- *Effects:* Let `T` be `decltype(abs(v(0)) * abs(v(0)))`. Then, the one-parameter overload is equivalent to `vector_norm2(v, T{});`, and the two-parameter overload is equivalent to `vector_norm2(exec, v, T{});`.

Sum of absolute values of vector elements [linalg.algs.blas1.asum]

Sum of absolute values with specified result type

```
template<class in_vector_t,
         class T>
T vector_abs_sum(in_vector_t v,
                 T init);
template<class ExecutionPolicy,
         class in_vector_t,
         class T>
T vector_abs_sum(ExecutionPolicy&& exec,
                 in_vector_t v,
                 T init);
```

[Note: This function corresponds to the BLAS functions `SASUM`, `DASUM`, `CSASUM`, and `DZASUM`. The different behavior for complex element types is based on the observation that this lower-cost approximation of the one-norm serves just as well as the actual one-norm for many linear algebra algorithms in practice. *--end note]*

- *Requires:*
  - `T` shall be *Cpp17MoveConstructible*.
  - `init + v1(0)*v2(0)` shall be convertible to `T`.
- *Constraints:* For all `i` in the domain of `v` and for `val` of type `T&`, the expression `val += abs(v(i))` is well formed.
- *Effects:* Let `N` be `v.extent(0)`.
  - If `N` is zero, returns `init`.
  - Else, if `in_vector_t::element_type` is `complex<R>` for some `R`, then returns /GENERALIZED_SUM/(`plus<>()`, `init`, `abs(real(v(0))) + abs(imag(v(0)))`, ..., `abs(real(v(N-1))) + abs(imag(v(N-1)))`).
  - Else, returns /GENERALIZED_SUM/(`plus<>()`, `init`, `abs(v(0))`, ..., `abs(v(N-1))`).

- *Remarks:* If `in_vector_t::element_type` is a floating-point type or a complex version thereof, if `T` is a floating-point type, and if `T` has higher precision than `in_vector_type::element_type`, then intermediate terms in the sum use `T`'s precision or greater.

Sum of absolute values with default result type

```
template<class in_vector_t>
auto vector_abs_sum(in_vector_t v) -> /* see-below */;
template<class ExecutionPolicy,
         class in_vector_t>
auto vector_abs_sum(ExecutionPolicy&& exec,
                    in_vector_t v) -> /* see-below */;
```

- *Effects:* Let `T` be `decltype(abs(v(0)))`. Then, the one-parameter overload is equivalent to `vector_abs_sum(v, T{});`, and the two-parameter overload is equivalent to `vector_abs_sum(exec, v, T{});`.

Index of maximum absolute value of vector elements [linalg.algs.blas1.iamax]

```
template<class in_vector_t>
ptrdiff_t idx_abs_max(in_vector_t v);

template<class ExecutionPolicy,
         class in_vector_t>
ptrdiff_t 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.

- *Effects:* 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`.

Frobenius norm of a matrix [linalg.algs.blas1.matfrobnorm]

Frobenius norm with specified result type

```
template<class in_matrix_t,
         class T>
T matrix_frob_norm(
  in_matrix_t A,
  T init);
template<class ExecutionPolicy,
         class in_matrix_t,
         class T>
```

```
T matrix_frob_norm(
  ExecutionPolicy&& exec,
  in_matrix_t A,
  T init);
```

- *Requires:*
  - `T` shall be *Cpp17MoveConstructible*.
  - `init + abs(A(0,0))*abs(A(0,0))` shall be convertible to `T`.
- *Constraints:* For all `i,j` in the domain of `A` and for `val` of type `T&`, the expressions `val += abs(A(i,j))*abs(A(i,j))` and `sqrt(val)` are well formed. *[Note:* This does not imply a recommended implementation for floating-point types. See *Remarks* below. *--end note]*
- *Effects:* Returns the Frobenius norm of the matrix `A`, that is, the square root of the sum of squares of the absolute values of the elements of `A`.
- *Remarks:* If `in_matrix_t::element_type` is a floating-point type or a complex version thereof, and if `T` is a floating-point type, then
  - if `T` has higher precision than `in_matrix_type::element_type`, then intermediate terms in the sum use `T`'s precision or greater; and
  - any guarantees regarding overflow and underflow of `matrix_frob_norm` are implementation-defined.

Frobenius norm with default result type

```
template<class in_matrix_t>
auto matrix_frob_norm(
  in_matrix_t A) -> /* see-below */;
template<class ExecutionPolicy,
         class in_matrix_t>
auto matrix_frob_norm(
  ExecutionPolicy&& exec,
  in_matrix_t A) -> /* see-below */;
```

- *Effects:* Let `T` be `decltype(abs(A(0,0)) * abs(A(0,0)))`. Then, the one-parameter overload is equivalent to `matrix_frob_norm(A, T{});`, and the two-parameter overload is equivalent to `matrix_frob_norm(exec, A, T{});`.

One norm of a matrix [linalg.algs.blas1.matonenorm]

One norm with specified result type

```
template<class in_matrix_t,
         class T>
T matrix_one_norm(
```

```
  in_matrix_t A,
  T init);
template<class ExecutionPolicy,
         class in_matrix_t,
         class T>
T matrix_one_norm(
  ExecutionPolicy&& exec,
  in_matrix_t A,
  T init);
```

- *Requires:*
  - `T` shall be *Cpp17MoveConstructible* and *Cpp17LessThanComparable*.
  - `abs(A(0,0))` shall be convertible to `T`.
- *Constraints:* For all `i,j` in the domain of `A` and for `val` of type `T&`, the expression `val += abs(A(i,j))` is well formed.
- *Effects:*
  - If `A.extent(1)` is zero, returns `init`;
  - Else, returns the one norm of the matrix `A`, that is, the maximum over all columns of `A`, of the sum of the absolute values of the elements of the column.
- *Remarks:* If `in_matrix_t::element_type` is a floating-point type or a complex version thereof, if `T` is a floating-point type, and if `T` has higher precision than `in_matrix_type::element_type`, then intermediate terms in each sum use `T`'s precision or greater.

One norm with default result type

```
template<class in_matrix_t>
auto matrix_one_norm(
  in_matrix_t A) -> /* see-below */;
template<class ExecutionPolicy,
         class in_matrix_t>
auto matrix_one_norm(
  ExecutionPolicy&& exec,
  in_matrix_t A) -> /* see-below */;
```

- *Effects:* Let `T` be `decltype(abs(A(0,0)) * abs(A(0,0)))`. Then, the one-parameter overload is equivalent to `matrix_one_norm(A, T{});`, and the two-parameter overload is equivalent to `matrix_one_norm(exec, A, T{});`.

Infinity norm of a matrix [linalg.algs.blas1.matinfnorm]

Infinity norm with specified result type

```
template<class in_matrix_t,
         class T>
T matrix_inf_norm(
  in_matrix_t A,
  T init);
template<class ExecutionPolicy,
         class in_matrix_t,
         class T>
T matrix_inf_norm(
  ExecutionPolicy&& exec,
  in_matrix_t A,
  T init);
```

- *Requires:*
    - `T` shall be *Cpp17MoveConstructible* and *Cpp17LessThanComparable*.
    - `abs(A(0,0))` shall be convertible to `T`.
- *Constraints:* For all `i,j` in the domain of `A` and for `val` of type `T&`, the expression `val += abs(A(i,j))` is well formed.
- *Effects:*
    - If `A.extent(0)` is zero, returns `init`;
    - Else, returns the infinity norm of the matrix `A`, that is, the maximum over all rows of `A`, of the sum of the absolute values of the elements of the row.
- *Remarks:* If `in_matrix_t::element_type` is a floating-point type or a complex version thereof, if `T` is a floating-point type, and if `T` has higher precision than `in_matrix_type::element_type`, then intermediate terms in each sum use `T`'s precision or greater.

Infinity norm with default result type

```
template<class in_matrix_t>
auto matrix_inf_norm(
  in_matrix_t A) -> /* see-below */;
template<class ExecutionPolicy,
         class in_matrix_t>
auto matrix_inf_norm(
  ExecutionPolicy&& exec,
  in_matrix_t A) -> /* see-below */;
```

- *Effects:* Let `T` be `decltype(abs(A(0,0)) * abs(A(0,0)))`. Then, the one-parameter overload is equivalent to `matrix_inf_norm(A, T{});`, and the two-parameter overload is equivalent to `matrix_inf_norm(exec, A, T{});`.

BLAS 2 functions [linalg.algs.blas2]

General matrix-vector product [linalg.algs.blas2.gemv]

[Note: These functions correspond to the BLAS function `xGEMV`. *--end note]*

The following requirements apply to all functions in this section.

- *Requires:*
  - `A.extent(1)` equals `x.extent(0)`.
  - `A.extent(0)` equals `y.extent(0)`.
  - `y.extent(0)` equals `z.extent(0)` (if applicable).
- *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 (if applicable).
- *Mandates:*
  - If neither `A.static_extent(1)` nor `x.static_extent(0)` equals `dynamic_extent`, then `A.static_extent(1)` equals `x.static_extent(0)`.
  - If neither `A.static_extent(0)` nor `y.static_extent(0)` equals `dynamic_extent`, then `A.static_extent(0)` equals `y.static_extent(0)`.
  - If neither `y.static_extent(0)` nor `z.static_extent(0)` equals `dynamic_extent`, then `y.static_extent(0)` equals `z.static_extent(0)` (if applicable).

Overwriting matrix-vector product

```
template<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`.

[*Example:*

```
constexpr ptrdiff_t num_rows = 5;
constexpr ptrdiff_t num_cols = 6;

// y = 3.0 * A * x
void scaled_matvec_1(
  mdspan<double, extents<num_rows, num_cols>> A,
  mdspan<double, extents<num_cols>> x,
  mdspan<double, extents<num_rows>> y)
{
  matrix_vector_product(scaled(3.0, A), x, y);
}

// y = 3.0 * A * x + 2.0 * y
void scaled_matvec_2(
  mdspan<double, extents<num_rows, num_cols>> A,
  mdspan<double, extents<num_cols>> x,
  mdspan<double, extents<num_rows>> y)
{
  matrix_vector_product(scaled(3.0, A), x,
                        scaled(2.0, y), y);
}

// z = 7.0 times the transpose of A, times y
void scaled_matvec_2(mdspan<double, extents<num_rows, num_cols>> A,
  mdspan<double, extents<num_rows>> y,
  mdspan<double, extents<num_cols>> z)
{
  matrix_vector_product(scaled(7.0, transposed(A)), y, z);
}
```

--*end example*]

Updating matrix-vector product

```
template<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);
```

- *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 [linalg.algs.blas2.symv]

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

The following requirements apply to all functions in this section.

- *Requires:*
  - `A.extent(0)` equals `A.extent(1)`.
  - `A.extent(1)` equals `x.extent(0)`.
  - `A.extent(0)` equals `y.extent(0)`.
  - `y.extent(0)` equals `z.extent(0)` (if applicable).
- *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 (if applicable).
- *Mandates:*
  - If neither `A.static_extent(0)` nor `A.static_extent(1)` equals `dynamic_extent`, then `A.static_extent(0)` equals `A.static_extent(1)`.
  - If neither `A.static_extent(1)` nor `x.static_extent(0)` equals `dynamic_extent`, then `A.static_extent(1)` equals `x.static_extent(0)`.
  - If neither `A.static_extent(0)` nor `y.static_extent(0)` equals `dynamic_extent`, then `A.static_extent(0)` equals `y.static_extent(0)`.
  - If neither `y.static_extent(0)` nor `z.static_extent(0)` equals `dynamic_extent`, then `y.static_extent(0)` equals `z.static_extent(0)` (if applicable).

- *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

```
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

```
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);
```

- *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 [linalg.algs.blas2.hemv]

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

The following requirements apply to all functions in this section.

- *Requires:*"
  - `A.extent(0)` equals `A.extent(1)`.
  - `A.extent(1)` equals `x.extent(0)`.
  - `A.extent(0)` equals `y.extent(0)`.
  - `y.extent(0)` equals `z.extent(0)` (if applicable).
- *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.
- *Mandates:*
  - If neither `A.static_extent(0)` nor `A.static_extent(1)` equals `dynamic_extent`, then `A.static_extent(0)` equals `A.static_extent(1)`.
  - If neither `A.static_extent(1)` nor `x.static_extent(0)` equals `dynamic_extent`, then `A.static_extent(1)` equals `x.static_extent(0)`.
  - If neither `A.static_extent(0)` nor `y.static_extent(0)` equals `dynamic_extent`, then `A.static_extent(0)` equals `y.static_extent(0)`.
  - If neither `y.static_extent(0)` nor `z.static_extent(0)` equals `dynamic_extent`, then `y.static_extent(0)` equals `z.static_extent(0)` (if applicable).
- *Remarks:*

- The functions will only access the triangle of `A` specified by the `Triangle` argument `t`.

- If `inout_matrix_t::element_type` is `complex<RA>` for some `RA`, then the functions will assume for indices `i,j` outside that triangle, that `A(j,i)` equals `conj(A(i,j))`. Otherwise, the functions will assume that `A(j,i)` equals `A(i,j)`.

Overwriting Hermitian matrix-vector product

```
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 expression `y(i) += A(i,j)*x(j)` is well formed; and
  - if `in_matrix_type::element_type` is `complex<RA>` for some `RA`, then the expression `y(i) += conj(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 Hermitian matrix-vector product

```
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);
```

- *Constraints:* For `i,j` in the domain of `A`:
  - the expression `z(i) = y(i) + A(i,j)*x(j)` is well formed; and
  - if `in_matrix_t::element_type` is `complex<RA>` for some `RA`, then the expression `z(i) = y(i) + conj(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`.

Triangular matrix-vector product [linalg.algs.blas2.trmv]

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

The following requirements apply to all functions in this section.

- *Requires:*
  - `A.extent(0)` equals `A.extent(1)`.
  - `A.extent(0)` equals `y.extent(0)`.
  - `A.extent(1)` equals `x.extent(0)` (if applicable).
  - `y.extent(0)` equals `z.extent(0)` (if applicable).
- *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.
  - `y.rank()` equals 1.
  - `x.rank()` equals 1 (if applicable).

    - `z.rank()` equals 1 (if applicable).

- *Mandates:*

    - If neither `A.static_extent(0)` nor `A.static_extent(1)` equals `dynamic_extent`, then `A.static_extent(0)` equals `A.static_extent(1)`.

    - If neither `A.static_extent(0)` nor `y.static_extent(0)` equals `dynamic_extent`, then `A.static_extent(0)` equals `y.static_extent(0)`.

    - If neither `A.static_extent(1)` nor `x.static_extent(0)` equals `dynamic_extent`, then `A.static_extent(1)` equals `x.static_extent(0)` (if applicable).

    - If neither `y.static_extent(0)` nor `z.static_extent(0)` equals `dynamic_extent`, then `y.static_extent(0)` equals `z.static_extent(0)` (if applicable).

- *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 [linalg.algs.blas2.trmv]

```
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`.

In-place triangular matrix-vector product [linalg.algs.blas2.trmv.in-place]

```
template<class in_matrix_t,
         class Triangle,
         class DiagonalStorage,
         class inout_vector_t>
void triangular_matrix_vector_product(
  in_matrix_t A,
  Triangle t,
  DiagonalStorage d,
  inout_vector_t y);
```

- *Requires:* `A.extent(1)` equals `y.extent(0)`.
- *Constraints:* For `i,j` in the domain of `A`, the expression `y(i) += A(i,j)*y(j)` is well formed.
- *Mandates:* If neither `A.static_extent(1)` nor `y.static_extent(0)` equals `dynamic_extent`, then `A.static_extent(1)` equals `y.static_extent(0)`.
- *Effects:* Overwrites `y` (on output) with the product of the matrix `A` with the vector `y` (on input).

Updating triangular matrix-vector product [linalg.algs.blas2.trmv.up]

```
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);
```

- *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 [linalg.algs.blas2.trsv]

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

The following requirements apply to all functions in this section.

- *Requires:*
  - `A.extent(0)` equals `A.extent(1)`.
  - `A.extent(1)` equals `b.extent(0)`.
- *Constraints:*
  - `A.rank()` equals 2.
  - `b.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.
- *Mandates:*
  - If neither `A.static_extent(0)` nor `A.static_extent(1)` equals `dynamic_extent`, then `A.static_extent(0)` equals `A.static_extent(1)`.
  - If neither `A.static_extent(1)` nor `b.static_extent(0)` equals `dynamic_extent`, then `A.static_extent(1)` equals `b.static_extent(0)`.
- *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]

Not-in-place triangular solve [linalg.algs.blas2.trsv.not-in-place]

```
template<class in_matrix_t,
         class Triangle,
         class DiagonalStorage,
         class in_vector_t,
         class out_vector_t>
void triangular_matrix_vector_solve(
  in_matrix_t A,
  Triangle t,
  DiagonalStorage d,
  in_vector_t b,
  out_vector_t x);
template<class ExecutionPolicy,
         class in_matrix_t,
         class Triangle,
         class DiagonalStorage,
         class in_vector_t,
         class out_vector_t>
void triangular_matrix_vector_solve(
  ExecutionPolicy&& exec,
  in_matrix_t A,
  Triangle t,
  DiagonalStorage d,
  in_vector_t b,
  out_vector_t x);
```

- *Requires:*
    - `A.extent(0)` equals `x.extent(0)`.
- *Constraints:*
    - `x.rank()` equals 1.
    - If `r` is in the domain of `x` and `b`, then the expression `x(r) = b(r)` is well formed.
    - If `r` is in the domain of `x` and `c` 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.
- *Mandates:*
    - If neither `A.static_extent(0)` nor `x.static_extent(0)` equals `dynamic_extent`, then `A.static_extent(0)` equals `x.static_extent(0)`.
- *Effects:* Assigns to the elements of `x` the result of solving the triangular linear system Ax=b.

In-place triangular solve [linalg.algs.blas2.trsv.in-place]

```
template<class in_matrix_t,
         class Triangle,
         class DiagonalStorage,
         class inout_vector_t>
void triangular_matrix_vector_solve(
  in_matrix_t A,
  Triangle t,
  DiagonalStorage d,
  inout_vector_t b);
template<class ExecutionPolicy,
         class in_matrix_t,
         class Triangle,
         class DiagonalStorage,
         class inout_vector_t>
void triangular_matrix_vector_solve(
  ExecutionPolicy&& exec,
  in_matrix_t A,
  Triangle t,
  DiagonalStorage d,
  inout_vector_t b);
```

[Note:

Performing triangular solve in place hinders parallelization. However, other `ExecutionPolicy`-specific optimizations, such as vectorization, are still possible. This is why the `ExecutionPolicy` overload exists.

--end note]

- *Requires:*
  - `A.extent(0)` equals `b.extent(0)`.
- *Constraints:*
  - If `r` and `c` are in the domain of `b`, then the expression `b(r) -= A(r,c)*b(c)` is well formed.
  - If `r` is in the domain of `b` and `DiagonalStorage` is `explicit_diagonal_t`, then the expression `b(r) /= A(r,r)` is well formed.
- *Mandates:*
  - If neither `A.static_extent(0)` nor `b.static_extent(0)` equals `dynamic_extent`, then `A.static_extent(0)` equals `b.static_extent(0)`.
- *Effects:* Overwrites `b` with the result of solving the triangular linear system Ax=b for x.

Rank-1 (outer product) update of a matrix [linalg.algs.blas2.rank1]

Nonsymmetric nonconjugated rank-1 update [linalg.algs.blas2.rank1.geru]

```
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) and `xGERU` (for complex element types). *--end note]*

* *Requires:*
  * `A.extent(0)` equals `x.extent(0)`.
  * `A.extent(1)` equals `y.extent(0)`.
* *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.
* *Mandates:*
  * If neither `A.static_extent(0)` nor `x.static_extent(0)` equals `dynamic_extent`, then `A.static_extent(0)` equals `x.static_extent(0)`.
  * If neither `A.static_extent(1)` nor `y.static_extent(0)` equals `dynamic_extent`, then `A.static_extent(1)` equals `y.static_extent(0)`.
* *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 the result of `conjugated`. Alternately, they can use the shortcut `matrix_rank_1_update_c` below. *--end note]*

Nonsymmetric conjugated rank-1 update [linalg.algs.blas2.rank1.gerc]

```
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);
```

[Note: This function corresponds to the BLAS functions `xGER` (for real element types) and `xGERC` (for complex element types). *--end note]*

- *Effects:* Equivalent to `matrix_rank_1_update(x, conjugated(y), A);`.

Rank-1 update of a Symmetric matrix [linalg.algs.blas2.rank1.syr]

```
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);
template<class T,
         class in_vector_t,
         class inout_matrix_t,
         class Triangle>
void symmetric_matrix_rank_1_update(
  T alpha,
  in_vector_t x,
  inout_matrix_t A,
  Triangle t);
template<class ExecutionPolicy,
         class T,
         class in_vector_t,
```

```
         class inout_matrix_t,
         class Triangle>
void symmetric_matrix_rank_1_update(
  ExecutionPolicy&& exec,
  T alpha,
  in_vector_t x,
  inout_matrix_t A,
  Triangle t);
```

[Note:

These functions correspond to the BLAS functions `xSYR` and `xSPR`.

They take an optional scaling factor `alpha`, because it would be impossible to express the update C = C - x x^T otherwise.

--end note]

- *Requires:*
  - `A.extent(0)` equals `A.extent(1)`.
  - `A.extent(0)` equals `x.extent(0)`.
- *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 overloads without `alpha`:
    - For `i,j` in the domain of `A`, the expression `A(i,j) += x(i)*x(j)` is well formed.
  - For overloads with `alpha`:
    - For `i,j` in the domain of `C`, and `i,k` and `k,i` in the domain of `A`, the expression `C(i,j) += alpha*A(i,k)*A(j,k)` is well formed.
- *Mandates:*
  - If neither `A.static_extent(0)` nor `A.static_extent(1)` equals `dynamic_extent`, then `A.static_extent(0)` equals `A.static_extent(1)`.
  - If neither `A.static_extent(0)` nor `x.static_extent(0)` equals `dynamic_extent`, then `A.static_extent(0)` equals `x.static_extent(0)`.
- *Effects:*
  - Overloads without `alpha` assign to `A` on output, the elementwise sum of `A` on input, with (the outer product of `x` and `x`).

  - Overloads with `alpha` assign to `A` on output, the elementwise sum of `A` on input, with alpha times (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 [linalg.algs.blas2.rank1.her]

```
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);
template<class T,
         class in_vector_t,
         class inout_matrix_t,
         class Triangle>
void hermitian_matrix_rank_1_update(
  T alpha,
  in_vector_t x,
  inout_matrix_t A,
  Triangle t);
template<class ExecutionPolicy,
         class T,
         class in_vector_t,
         class inout_matrix_t,
         class Triangle>
void hermitian_matrix_rank_1_update(
  ExecutionPolicy&& exec,
  T alpha,
  in_vector_t x,
  inout_matrix_t A,
  Triangle t);
```

[Note:

These functions correspond to the BLAS functions `xHER` and `xHPR`.

They take an optional scaling factor `alpha`, because it would be impossible to express the update A = A - x x^H otherwise.

--end note]

- *Requires:*
  - `A.extent(0)` equals `A.extent(1)`.
  - `A.extent(0)` equals `x.extent(0)`.
- *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 overloads without `alpha`:
    - For `i,j` in the domain of `A`:
      - if `in_vector_t::element_type` is `complex<RX> for some` RX`, then the expression` A(i,j) += x(i)*conj(x(j))` is well formed;
      - else, the expression `A(i,j) += x(i)*x(j)` is well formed.
  - For overloads with `alpha`:
    - For `i,j` in the domain of `A`:
      - if `in_vector_t::element_type` is `complex<RX> for some` RX`, then the expression` A(i,j) += alpha*x(i)*conj(x(j))` is well formed;
      - else, the expression `A(i,j) += alpha*x(i)*x(j)` is well formed.
- *Mandates:*
  - If neither `A.static_extent(0)` nor `A.static_extent(1)` equals `dynamic_extent`, then `A.static_extent(0)` equals `A.static_extent(1)`.
  - If neither `A.static_extent(0)` nor `x.static_extent(0)` equals `dynamic_extent`, then `A.static_extent(0)` equals `x.static_extent(0)`.
- *Effects:*
  - Overloads without `alpha` assign to `A` on output, the elementwise sum of `A` on input, with (the outer product of `x` and the conjugate of `x`).
  - Overloads with `alpha` assign to `A` on output, the elementwise sum of `A` on input, with alpha times (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`.
  - If `inout_matrix_t::element_type` is `complex<RA>` for some `RA`, then the functions will assume for indices `i,j` outside that triangle, that `A(j,i)` equals `conj(A(i,j))`. Otherwise, the functions will assume that `A(j,k)` equals `A(i,j)`.

Rank-2 update of a symmetric matrix [linalg.algs.blas2.rank2.syr2]

```
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:*
  - `A.extent(0)` equals `A.extent(1)`.
  - `A.extent(0)` equals `x.extent(0)`.
  - `A.extent(0)` equals `y.extent(0)`.
- *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.
- *Mandates:*

  - If neither `A.static_extent(0)` nor `A.static_extent(1)` equals `dynamic_extent`, then `A.static_extent(0)` equals `A.static_extent(1)`.
  - If neither `A.static_extent(0)` nor `x.static_extent(0)` equals `dynamic_extent`, then `A.static_extent(0)` equals `x.static_extent(0)`.
  - If neither `A.static_extent(0)` nor `y.static_extent(0)` equals `dynamic_extent`, then `A.static_extent(0)` equals `y.static_extent(0)`.

- *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 [linalg.algs.blas2.rank2.her2]

```
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:*

  - `A.extent(0)` equals `A.extent(1)`.
  - `A.extent(0)` equals `x.extent(0)`.
  - `A.extent(0)` equals `y.extent(0)`.

- *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`:
    - If `in_vector_2_t::element_type` is `complex<RY>` for some `RY`,
      - if `in_vector_1_t::element_type` is `complex<RX>` for some `RX`, then the expression `A(i,j) += x(i)*conj(y(j)) + y(i)*conj(x(j))` is well formed;
      - else, the expression `A(i,j) += x(i)*conj(y(j)) + y(i)*x(j)` is well formed;
    - else,
      - if `in_vector_1_t::element_type` is `complex<RX>` for some `RX`, then the expression `A(i,j) += x(i)*y(j) + y(i)*conj(x(j))` is well formed;
      - else, the expression `A(i,j) += x(i)*y(j)) + y(i)*x(j)` is well formed.
- *Mandates:*
  - If neither `A.static_extent(0)` nor `A.static_extent(1)` equals `dynamic_extent`, then `A.static_extent(0)` equals `A.static_extent(1)`.
  - If neither `A.static_extent(0)` nor `x.static_extent(0)` equals `dynamic_extent`, then `A.static_extent(0)` equals `x.static_extent(0)`.
  - If neither `A.static_extent(0)` nor `y.static_extent(0)` equals `dynamic_extent`, then `A.static_extent(0)` equals `y.static_extent(0)`.
- *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`.
  - If `inout_matrix_t::element_type` is `complex<RA>` for some `RA`, then the functions will assume for indices `i,j` outside that triangle, that `A(j,i)` equals `conj(A(i,j))`. Otherwise, the functions will assume that `A(j,i)` equals `A(i,j)`.

BLAS 3 functions [linalg.algs.blas3]

General matrix-matrix product [linalg.algs.blas3.gemm]

[Note: These functions correspond to the BLAS function `xGEMM`. *--end note]*

The following requirements apply to all functions in this section.

- *Requires:*

- `C.extent(0)` equals `E.extent(0)` (if applicable).
- `C.extent(1)` equals `E.extent(1)` (if applicable).
- `A.extent(1)` equals `B.extent(0)`.
- `A.extent(0)` equals `C.extent(0)`.
- `B.extent(1)` equals `C.extent(1)`.

- *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.
- *Mandates:*
  - For all `r` in 0, 1, ..., `C.rank()` - 1, if neither `C.static_extent(r)` nor `E.static_extent(r)` equals `dynamic_extent`, then `C.static_extent(r)` equals `E.static_extent(r)` (if applicable).
  - If neither `A.static_extent(1)` nor `B.static_extent(0)` equals `dynamic_extent`, then `A.static_extent(1)` equals `B.static_extent(0)`.
  - If neither `A.static_extent(0)` nor `C.static_extent(0)` equals `dynamic_extent`, then `A.static_extent(0)` equals `C.static_extent(0)`.
  - If neither `B.static_extent(1)` nor `C.static_extent(1)` equals `dynamic_extent`, then `B.static_extent(1)` equals `C.static_extent(1)`.

Overwriting general matrix-matrix product

```
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

```
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);
```

- *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 [linalg.algs.blas3.symm]

[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:*

    - `A.extent(0)` equals `A.extent(1)`.

- `C.extent(0)` equals `E.extent(0)` (if applicable).
- `C.extent(1)` equals `E.extent(1)` (if applicable).

* *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.
* *Mandates:*
  - If neither `A.static_extent(0)` nor `A.static_extent(1)` equals `dynamic_extent`, then `A.static_extent(0)` equals `A.static_extent(1)`.
  - For all `r` in 0, 1, ..., `C.rank()` - 1, if neither `C.static_extent(r)` nor `E.static_extent(r)` equals `dynamic_extent`, then `C.static_extent(r)` equals `E.static_extent(r)` (if applicable).
* *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)`.
  - *Remarks:* `C` and `E` (if applicable) may refer to the same matrix. If so, then they must have the same layout.

The following requirements apply to all overloads of `symmetric_matrix_left_product`.

* *Requires:*
  - `A.extent(1)` equals `B.extent(0)`,
  - `A.extent(0)` equals `C.extent(0)`, and
  - `B.extent(1)` equals `C.extent(1)`.
* *Mandates:*
  - If neither `A.static_extent(1)` nor `B.static_extent(0)` equals `dynamic_extent`, then `A.static_extent(1)` equals `B.static_extent(0)`;
  - if neither `A.static_extent(0)` nor `C.static_extent(0)` equals `dynamic_extent`, then `A.static_extent(0)` equals `C.static_extent(0)`; and
  - if neither `B.static_extent(1)` nor `C.static_extent(1)` equals `dynamic_extent`, then `B.static_extent(1)` equals `C.static_extent(1)`.

The following requirements apply to all overloads of `symmetric_matrix_right_product`.

- *Requires:*
  - `B.extent(1)` equals `A.extent(0)`,
  - `B.extent(0)` equals `C.extent(0)`, and
  - `A.extent(1)` equals `C.extent(1)`.
- *Mandates:*
  - If neither `B.static_extent(1)` nor `A.static_extent(0)` equals `dynamic_extent`, then `B.static_extent(1)` equals `A.static_extent(0)`;
  - if neither `B.static_extent(0)` nor `C.static_extent(0)` equals `dynamic_extent`, then `B.static_extent(0)` equals `C.static_extent(0)`; and
  - if neither `A.static_extent(1)` nor `C.static_extent(1)` equals `dynamic_extent`, then `A.static_extent(1)` equals `C.static_extent(1)`.

Overwriting symmetric matrix-matrix left product [linalg.algs.blas3.symm.ov.left]

```
template<class in_matrix_1_t,
         class Triangle,
         class in_matrix_2_t,
         class out_matrix_t>
void symmetric_matrix_left_product(
  in_matrix_1_t A,
  Triangle t,
  in_matrix_2_t B,
  out_matrix_t C);
template<class ExecutionPolicy,
         class in_matrix_1_t,
         class Triangle,
         class in_matrix_2_t,
         class out_matrix_t>
void symmetric_matrix_left_product(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
  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`.

Overwriting symmetric matrix-matrix right product [linalg.algs.blas3.symm.ov.right]

```
template<class in_matrix_1_t,
         class Triangle,
         class in_matrix_2_t,
         class out_matrix_t>
void symmetric_matrix_right_product(
  in_matrix_1_t A,
  Triangle t,
  in_matrix_2_t B,
  out_matrix_t C);
template<class ExecutionPolicy,
         class in_matrix_1_t,
         class Triangle,
         class in_matrix_2_t,
         class out_matrix_t>
void symmetric_matrix_right_product(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
  in_matrix_2_t B,
  out_matrix_t C);
```

- *Constraints:* 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:* Assigns to the elements of the matrix `C` the product of the matrices `B` and `A`.

Updating symmetric matrix-matrix left product [linalg.algs.blas3.symm.up.left]

```
template<class in_matrix_1_t,
         class Triangle,
         class in_matrix_2_t,
         class in_matrix_3_t,
         class out_matrix_t>
void symmetric_matrix_left_product(
  in_matrix_1_t A,
  Triangle t,
  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 in_matrix_2_t,
         class in_matrix_3_t,
         class out_matrix_t>
void symmetric_matrix_left_product(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
```

```
  in_matrix_2_t B,
  in_matrix_3_t E,
  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) += 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`.

Updating symmetric matrix-matrix right product [linalg.algs.blas3.symm.up.right]

```
template<class in_matrix_1_t,
         class Triangle,
         class in_matrix_2_t,
         class in_matrix_3_t,
         class out_matrix_t>
void symmetric_matrix_right_product(
  in_matrix_1_t A,
  Triangle t,
  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 in_matrix_2_t,
         class in_matrix_3_t,
         class out_matrix_t>
void symmetric_matrix_right_product(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
  in_matrix_2_t B,
  in_matrix_3_t E,
  out_matrix_t C);
```

- *Constraints:* 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:* assigns to the elements of the matrix `C` on output, the elementwise sum of `E` and the product of the matrices `B` and `A`.

Hermitian matrix-matrix product [linalg.algs.blas3.hemm]

[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:*
  - `A.extent(0)` equals `A.extent(1)`.
  - `C.extent(0)` equals `E.extent(0)` (if applicable).
  - `C.extent(1)` equals `E.extent(1)` (if applicable).
- *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.
- *Mandates:*
  - If neither `A.static_extent(0)` nor `A.static_extent(1)` equals `dynamic_extent`, then `A.static_extent(0)` equals `A.static_extent(1)`.
  - For all `r` in 0, 1, ..., `C.rank()` - 1, if neither `C.static_extent(r)` nor `E.static_extent(r)` equals `dynamic_extent`, then `C.static_extent(r)` equals `E.static_extent(r)` (if applicable).
- *Remarks:*
  - The functions will only access the triangle of `A` specified by the `Triangle` argument `t`.
  - If `in_matrix_1_t::element_type` is `complex<RA>` for some `RA`, then the functions will assume for indices `i,j` outside that triangle, that `A(j,i)` equals `conj(A(i,j))`. Otherwise, the functions will assume that `A(j,i)` equals `A(i,j)`.
  - `C` and `E` (if applicable) may refer to the same matrix. If so, then they must have the same layout.

The following requirements apply to all overloads of `hermitian_matrix_left_product`.

- *Requires:*
  - `A.extent(1)` equals `B.extent(0)`,
  - `A.extent(0)` equals `C.extent(0)`, and
  - `B.extent(1)` equals `C.extent(1)`.

- *Mandates:*
  - If neither `A.static_extent(1)` nor `B.static_extent(0)` equals `dynamic_extent`, then `A.static_extent(1)` equals `B.static_extent(0)`;
  - if neither `A.static_extent(0)` nor `C.static_extent(0)` equals `dynamic_extent`, then `A.static_extent(0)` equals `C.static_extent(0)`; and
  - if neither `B.static_extent(1)` nor `C.static_extent(1)` equals `dynamic_extent`, then `B.static_extent(1)` equals `C.static_extent(1)`.

The following requirements apply to all overloads of `hermitian_matrix_right_product`.

- *Requires:*
  - `B.extent(1)` equals `A.extent(0)`,
  - `B.extent(0)` equals `C.extent(0)`, and
  - `A.extent(1)` equals `C.extent(1)`.
- *Mandates:*
  - If neither `B.static_extent(1)` nor `A.static_extent(0)` equals `dynamic_extent`, then `B.static_extent(1)` equals `A.static_extent(0)`;
  - if neither `B.static_extent(0)` nor `C.static_extent(0)` equals `dynamic_extent`, then `B.static_extent(0)` equals `C.static_extent(0)`; and
  - if neither `A.static_extent(1)` nor `C.static_extent(1)` equals `dynamic_extent`, then `A.static_extent(1)` equals `C.static_extent(1)`.

Overwriting Hermitian matrix-matrix left product [linalg.algs.blas3.hemm.ov.left]

```
template<class in_matrix_1_t,
         class Triangle,
         class in_matrix_2_t,
         class out_matrix_t>
void hermitian_matrix_left_product(
  in_matrix_1_t A,
  Triangle t,
  in_matrix_2_t B,
  out_matrix_t C);
template<class ExecutionPolicy,
         class in_matrix_1_t,
         class Triangle,
         class in_matrix_2_t,
         class out_matrix_t>
void hermitian_matrix_left_product(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
```

```
  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`.

Overwriting Hermitian matrix-matrix right product [linalg.algs.blas3.hemm.ov.right]

```
template<class in_matrix_1_t,
         class Triangle,
         class in_matrix_2_t,
         class out_matrix_t>
void hermitian_matrix_right_product(
  in_matrix_1_t A,
  Triangle t,
  in_matrix_2_t B,
  out_matrix_t C);
template<class ExecutionPolicy,
         class in_matrix_1_t,
         class Triangle,
         class in_matrix_2_t,
         class out_matrix_t>
void hermitian_matrix_right_product(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
  in_matrix_2_t B,
  out_matrix_t C);
```

- *Constraints:* 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:* Assigns to the elements of the matrix `C` the product of the matrices `B` and `A`.

Updating Hermitian matrix-matrix left product [linalg.algs.blas3.hemm.up.left]

```
template<class in_matrix_1_t,
         class Triangle,
         class in_matrix_2_t,
         class in_matrix_3_t,
         class out_matrix_t>
void hermitian_matrix_left_product(
  in_matrix_1_t A,
  Triangle t,
  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 in_matrix_2_t,
         class in_matrix_3_t,
         class out_matrix_t>
void hermitian_matrix_left_product(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
  in_matrix_2_t B,
  in_matrix_3_t E,
  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) += 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`.

Updating Hermitian matrix-matrix right product [linalg.algs.blas3.hemm.up.right]

```
template<class in_matrix_1_t,
         class Triangle,
         class in_matrix_2_t,
         class in_matrix_3_t,
         class out_matrix_t>
void hermitian_matrix_right_product(
  in_matrix_1_t A,
  Triangle t,
  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 in_matrix_2_t,
         class in_matrix_3_t,
         class out_matrix_t>
void hermitian_matrix_right_product(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
  in_matrix_2_t B,
  in_matrix_3_t E,
  out_matrix_t C);
```

- *Constraints:* 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:* Assigns to the elements of the matrix `C` on output, the elementwise sum of `E` and the product of the matrices `B` and `A`.

Triangular matrix-matrix product [linalg.algs.blas3.trmm]

[Note: These functions correspond to the BLAS function `xTRMM`. --*end note]*

The following requirements apply to all functions in this section.

- *Requires:*
  - `A.extent(0)` equals `A.extent(1)`.
  - `C.extent(0)` equals `E.extent(0)` (if applicable).
  - `C.extent(1)` equals `E.extent(1)` (if applicable).
- *Constraints:*
  - `in_matrix_1_t` either has unique layout, or `layout_blas_packed` layout.
  - `in_matrix_2_t`, `in_matrix_3_t` (if applicable), `out_matrix_t`, and `inout_matrix_t` (if applicable) have unique layout.
  - If `in_matrix_1_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.
- *Mandates:*
  - If neither `A.static_extent(0)` nor `A.static_extent(1)` equals `dynamic_extent`, then `A.static_extent(0)` equals `A.static_extent(1)`.
  - For all `r` in 0, 1, ..., `C.rank()` - 1, if neither `C.static_extent(r)` nor `E.static_extent(r)` equals `dynamic_extent`, then `C.static_extent(r)` equals `E.static_extent(r)` (if applicable).
- *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]
  - `C` and `E` (if applicable) may refer to the same matrix. If so, then they must have the same layout.

The following requirements apply to all overloads of `triangular_matrix_left_product`.

* *Requires:*
  * `A.extent(1)` equals `B.extent(0)` (if applicable),
  * `A.extent(0)` equals `C.extent(0)`, and
  * `B.extent(1)` equals `C.extent(1)` (if applicable).
* *Mandates:*
  * If neither `A.static_extent(1)` nor `B.static_extent(0)` equals `dynamic_extent`, then `A.static_extent(1)` equals `B.static_extent(0)` (if applicable);
  * if neither `A.static_extent(0)` nor `C.static_extent(0)` equals `dynamic_extent`, then `A.static_extent(0)` equals `C.static_extent(0)`; and
  * if neither `B.static_extent(1)` nor `C.static_extent(1)` equals `dynamic_extent`, then `B.static_extent(1)` equals `C.static_extent(1)` (if applicable).

The following requirements apply to all overloads of `triangular_matrix_right_product`.

* *Requires:*
  * `B.extent(1)` equals `A.extent(0)` (if applicable),
  * `B.extent(0)` equals `C.extent(0)` (if applicable), and
  * `A.extent(1)` equals `C.extent(1)`.
* *Mandates:*
  * If neither `B.static_extent(1)` nor `A.static_extent(0)` equals `dynamic_extent`, then `B.static_extent(1)` equals `A.static_extent(0)` (if applicable);
  * if neither `B.static_extent(0)` nor `C.static_extent(0)` equals `dynamic_extent`, then `B.static_extent(0)` equals `C.static_extent(0)` (if applicable); and
  * if neither `A.static_extent(1)` nor `C.static_extent(1)` equals `dynamic_extent`, then `A.static_extent(1)` equals `C.static_extent(1)`.

Overwriting triangular matrix-matrix left product [linalg.algs.blas3.trmm.ov.left]

Not-in-place overwriting triangular matrix-matrix left product

```
template<class in_matrix_1_t,
         class Triangle,
         class DiagonalStorage,
         class in_matrix_2_t,
         class out_matrix_t>
void triangular_matrix_left_product(
  in_matrix_1_t A,
```

```
  Triangle t,
  DiagonalStorage d,
  in_matrix_2_t B,
  out_matrix_t C);
template<class ExecutionPolicy,
         class in_matrix_1_t,
         class Triangle,
         class DiagonalStorage,
         class in_matrix_2_t,
         class out_matrix_t>
void triangular_matrix_left_product(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
  DiagonalStorage d,
  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`.

In-place overwriting triangular matrix-matrix left product

```
template<class in_matrix_1_t,
         class Triangle,
         class DiagonalStorage,
         class inout_matrix_t>
void triangular_matrix_left_product(
  in_matrix_1_t A,
  Triangle t,
  DiagonalStorage d,
  inout_matrix_t C);
template<class ExecutionPolicy,
         class in_matrix_1_t,
         class Triangle,
         class DiagonalStorage,
         class inout_matrix_t>
void triangular_matrix_left_product(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
  DiagonalStorage d,
  inout_matrix_t C);
```

- *Requires:* `A.extent(1)` equals `C.extent(0)`.

- *Constraints:* For `i,j` and `k,j` in the domain of `C`, and `i,k` in the domain of `A`, the expression `C(i,j) += A(i,k)*C(k,j)` is well formed.

- *Mandates:* If neither `A.static_extent(1)` nor `C.static_extent(0)` equals `dynamic_extent`, then `A.static_extent(1)` equals `C.static_extent(0)`.

- *Effects:* Overwrites `C` on output with the product of the matrices `A` and `C` (on input).

Overwriting triangular matrix-matrix right product [linalg.algs.blas3.trmm.ov.right]

Not-in-place overwriting triangular matrix-matrix right product

```
template<class in_matrix_1_t,
         class Triangle,
         class DiagonalStorage,
         class in_matrix_2_t,
         class out_matrix_t>
void triangular_matrix_right_product(
  in_matrix_1_t A,
  Triangle t,
  DiagonalStorage d,
  in_matrix_2_t B,
  out_matrix_t C);
template<class ExecutionPolicy,
         class in_matrix_1_t,
         class Triangle,
         class DiagonalStorage,
         class in_matrix_2_t,
         class out_matrix_t>
void triangular_matrix_right_product(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
  DiagonalStorage d,
  in_matrix_2_t B,
  out_matrix_t C);
```

- *Constraints:* 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:* Assigns to the elements of the matrix `C` the product of the matrices `B` and `A`.

In-place overwriting triangular matrix-matrix right product

```
template<class in_matrix_1_t,
         class Triangle,
         class DiagonalStorage,
         class inout_matrix_t>
void triangular_matrix_right_product(
  in_matrix_1_t A,
  Triangle t,
  DiagonalStorage d,
  inout_matrix_t C);
```

```
template<class ExecutionPolicy,
         class in_matrix_1_t,
         class Triangle,
         class DiagonalStorage,
         class inout_matrix_t>
void triangular_matrix_right_product(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
  DiagonalStorage d,
  inout_matrix_t C);
```

- *Requires:* `C.extent(1)` equals `A.extent(0)`.
- *Constraints:* For `i,j` and `i,k` in the domain of `C`, and `k,j` in the domain of `A`, the expression `C(i,j) += C(i,k)*A(k,j)` is well formed.
- *Mandates:* If neither `C.static_extent(1)` nor `A.static_extent(0)` equals `dynamic_extent`, then `C.static_extent(1)` equals `A.static_extent(0).
- *Effects:* Overwrites `C` on output with the product of the matrices `C` (on input) and `A`.

Updating triangular matrix-matrix left product [linalg.algs.blas3.trmm.up.left]

```
template<class in_matrix_1_t,
         class Triangle,
         class DiagonalStorage,
         class in_matrix_2_t,
         class in_matrix_3_t,
         class out_matrix_t>
void triangular_matrix_left_product(
  in_matrix_1_t A,
  Triangle t,
  DiagonalStorage d,
  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 DiagonalStorage,
         class in_matrix_2_t,
         class in_matrix_3_t,
         class out_matrix_t>
void triangular_matrix_left_product(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
  DiagonalStorage d,
  in_matrix_2_t B,
```

```
  in_matrix_3_t E,
  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) += 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`.

Updating triangular matrix-matrix right product [linalg.algs.blas3.trmm.up.right]

```
template<class in_matrix_1_t,
         class Triangle,
         class DiagonalStorage,
         class in_matrix_2_t,
         class in_matrix_3_t,
         class out_matrix_t>
void triangular_matrix_right_product(
  in_matrix_1_t A,
  Triangle t,
  DiagonalStorage d,
  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 DiagonalStorage,
         class in_matrix_2_t,
         class in_matrix_3_t,
         class out_matrix_t>
void triangular_matrix_right_product(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
  DiagonalStorage d,
  in_matrix_2_t B,
  in_matrix_3_t E,
  out_matrix_t C);
```

- *Constraints:* 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:* Assigns to the elements of the matrix `C` on output, the elementwise sum of `E` and the product of the matrices `B` and `A`.

Rank-k update of a symmetric or Hermitian matrix [linalg.alg.blas3.rank-k]

[Note: Users can achieve the effect of the `TRANS` argument of these BLAS functions, by applying `transposed` or `conjugate_transposed` to the input matrix. *--end note]*

Rank-k symmetric matrix update [linalg.alg.blas3.rank-k.syrk]

```
template<class in_matrix_1_t,
         class inout_matrix_t,
         class Triangle>
void symmetric_matrix_rank_k_update(
  in_matrix_1_t A,
  inout_matrix_t C,
  Triangle t);
template<class ExecutionPolicy,
         class in_matrix_1_t,
         class inout_matrix_t,
         class Triangle>
void symmetric_matrix_rank_k_update(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  inout_matrix_t C,
  Triangle t);
template<class T,
         class in_matrix_1_t,
         class inout_matrix_t,
         class Triangle>
void symmetric_matrix_rank_k_update(
  T alpha,
  in_matrix_1_t A,
  inout_matrix_t C,
  Triangle t);
template<class ExecutionPolicy,
         class T,
         class in_matrix_1_t,
         class inout_matrix_t,
         class Triangle>
void symmetric_matrix_rank_k_update(
  ExecutionPolicy&& exec,
  T alpha,
  in_matrix_1_t A,
  inout_matrix_t C,
  Triangle t);
```

[Note:

These functions correspond to the BLAS function `xSYRK`.

They take an optional scaling factor `alpha`, because it would be impossible to express the update C = C - A A^T otherwise.

--end note]

- *Requires:*
  - `A.extent(0)` equals `C.extent(0)`.
  - `C.extent(0)` equals `C.extent(1)`.
- *Constraints:*
  - `A.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`, and `i,k` and `k,i` in the domain of `A`, the expression `C(i,j) += A(i,k)*A(j,k)` is well formed.
  - For `i,j` in the domain of `C`, and `i,k` and `k,i` in the domain of `A`, the expression `C(i,j) += alpha*A(i,k)*A(j,k)` is well formed (if applicable).
- *Mandates:*
  - If neither `A.static_extent(0)` nor `C.static_extent(0)` equals `dynamic_extent`, then `A.static_extent(0)` equals `C.static_extent(0)`.
  - If neither `C.static_extent(0)` nor `C.static_extent(1)` equals `dynamic_extent`, then `C.static_extent(0)` equals `C.static_extent(1)`.
- *Effects:*
  - Overloads without `alpha` assign to `C` on output, the elementwise sum of `C` on input with (the matrix product of `A` and the nonconjugated transpose of `A`).
  - Overloads with `alpha` assign to `C` on output, the elementwise sum of `C` on input with alpha times (the matrix product of `A` and the nonconjugated 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-k symmetric matrix update [linalg.alg.blas3.rank-k.herk]

```
template<class in_matrix_1_t,
         class inout_matrix_t,
         class Triangle>
void hermitian_matrix_rank_k_update(
  in_matrix_1_t A,
  inout_matrix_t C,
  Triangle t);
template<class ExecutionPolicy,
         class in_matrix_1_t,
         class inout_matrix_t,
```

```
          class Triangle>
void hermitian_matrix_rank_k_update(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  inout_matrix_t C,
  Triangle t);
template<class T,
          class in_matrix_1_t,
          class inout_matrix_t,
          class Triangle>
void hermitian_matrix_rank_k_update(
  T alpha,
  in_matrix_1_t A,
  inout_matrix_t C,
  Triangle t);
template<class ExecutionPolicy,
          class T,
          class in_matrix_1_t,
          class inout_matrix_t,
          class Triangle>
void hermitian_matrix_rank_k_update(
  ExecutionPolicy&& exec,
  T alpha,
  in_matrix_1_t A,
  inout_matrix_t C,
  Triangle t);
```

[Note:

These functions correspond to the BLAS function `xHERK`.

They take an optional scaling factor `alpha`, because it would be impossible to express the updates C = C - A A^T or C = C - A A^H otherwise.

--end note]

- *Requires:*
  - `A.extent(0)` equals `C.extent(0)`.
  - `C.extent(0)` equals `C.extent(1)`.
- *Constraints:*
  - `A.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 overloads without `alpha`: For `i,j` in the domain of `C`, and `i,k` and `k,i` in the domain of `A`,

        - if `in_matrix_1_t::element_type` is `complex<R>` for some `R`, then the expression `C(i,j) += A(i,k)*conj(A(j,k))` is well formed;
        - else, the expression `C(i,j) += A(i,k)*A(j,k)` is well formed.
    - For overloads with `alpha`: For `i,j` in the domain of `C`, and `i,k` and `k,i` in the domain of `A`,
        - if `in_matrix_1_t::element_type` is `complex<R>` for some `R`, then the expression `C(i,j) += alpha*A(i,k)*conj(A(j,k))` is well formed;
        - else, the expression `C(i,j) += alpha*A(i,k)*A(j,k)` is well formed.
- *Mandates:*
    - If neither `A.static_extent(0)` nor `C.static_extent(0)` equals `dynamic_extent`, then `A.static_extent(0)` equals `C.static_extent(0)`.
    - If neither `C.static_extent(0)` nor `C.static_extent(1)` equals `dynamic_extent`, then `C.static_extent(0)` equals `C.static_extent(1)`.
- *Effects:*
    - Overloads without `alpha` assign to `C` on output, the elementwise sum of `C` on input with (the matrix product of `A` and the conjugated transpose of `A`).
    - Overloads with `alpha` assign to `C` on output, the elementwise sum of `C` on input with alpha times (the matrix product of `A` and the 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 symmetric or Hermitian matrix [linalg.alg.blas3.rank2k]

[Note: Users can achieve the effect of the `TRANS` argument of these BLAS functions, by applying `transposed` or `conjugate_transposed` to the input matrices. *--end note]*

Rank-2k symmetric matrix update [linalg.alg.blas3.rank2k.syr2k]

```
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:*
  - `A.extent(0)` equals `C.extent(0)`.
  - `B.extent(1)` equals `C.extent(0)`.
  - `C.extent(0)` equals `C.extent(1)`.
- *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.
- *Mandates:*
  - If neither `A.static_extent(0)` nor `C.static_extent(0)` equals `dynamic_extent`, then `A.static_extent(0)` equals `C.static_extent(0)`.
  - If neither `B.static_extent(1)` nor `C.static_extent(0)` equals `dynamic_extent`, then `B.static_extent(1)` equals `C.static_extent(0)`.
  - If neither `C.static_extent(0)` nor `C.static_extent(1)` equals `dynamic_extent`, then `C.static_extent(0)` equals `C.static_extent(1)`.
- *Effects:* Assigns to `C` on output, the elementwise sum of `C` on input with (the matrix product of `A` and the nonconjugated transpose of `B`) and (the matrix product of `B` and the nonconjugated 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 Hermitian matrix update [linalg.alg.blas3.rank2k.her2k]

```
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:*
  - `A.extent(0)` equals `C.extent(0)`.
  - `B.extent(1)` equals `C.extent(0)`.
  - `C.extent(0)` equals `C.extent(1)`.
- *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`,
    - if `in_matrix_1_t::element_type` is `complex<RA>` for some `RA`, then
      - if `in_matrix_2_t::element_type` is `complex<RB>` for some `RB`, then the expression `C(i,j) += A(i,k)*conj(B(j,k)) + B(i,k)*conj(A(j,k))` is well formed;
      - else, the expression `C(i,j) += A(i,k)*B(j,k) + B(i,k)*conj(A(j,k))` is well formed;

        - else,
            - if `in_matrix_2_t::element_type` is `complex<RB>` for some `RB`, then the expression `C(i,j) += A(i,k)*conj(B(j,k)) + B(i,k)*A(j,k)` is well formed;
            - else, the expression `C(i,j) += A(i,k)*B(j,k) + B(i,k)*A(j,k)` is well formed.

- *Mandates:*
    - If neither `A.static_extent(0)` nor `C.static_extent(0)` equals `dynamic_extent`, then `A.static_extent(0)` equals `C.static_extent(0)`.
    - If neither `B.static_extent(1)` nor `C.static_extent(0)` equals `dynamic_extent`, then `B.static_extent(1)` equals `C.static_extent(0)`.
    - If neither `C.static_extent(0)` nor `C.static_extent(1)` equals `dynamic_extent`, then `C.static_extent(0)` equals `C.static_extent(1)`.
- *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`.
    - If `inout_matrix_t::element_type` is `complex<RC>` for some `RC`, then the functions will assume for indices `i,j` outside that triangle, that `C(j,i)` equals `conj(C(i,j))`. Otherwise, the functions will assume that `C(j,i)` equals `C(i,j)`.

Solve multiple triangular linear systems [linalg.alg.blas3.trsm]

[Note: These functions correspond to the BLAS function `xTRSM`. The Reference BLAS does not have a `xTPSM` function. *--end note]*

The following requirements apply to all functions in this section.

- *Requires:*
    - For all `r` in 0, 1, ..., `B.rank()` - 1, `X.extent(r)` equals `B.extent(r)` (if applicable).
    - `A.extent(0)` equals `A.extent(1)`.
- *Constraints:*
    - `A.rank()` equals 2 and `B.rank()` equals 2.
    - `X.rank()` equals 2 (if applicable).
    - `in_matrix_1_t` either has unique layout, or `layout_blas_packed` layout.
    - `in_matrix_2_t` has unique layout (if applicable).
    - `out_matrix_t` has unique layout.

  - `inout_matrix_t` has unique layout (if applicable).
  - If `r,j` is in the domain of `X` and `B`, then the expression `X(r,j) = B(r,j)` is well formed (if applicable).
  - If `DiagonalStorage` is `explicit_diagonal_t`, and `i,j` is in the domain of `X`, then the expression `X(i,j) /= A(i,i)` is well formed (if applicable).
- *Mandates:*
  - For all `r` in 0, 1, ..., `X.rank()` - 1, if neither `X.static_extent(r)` nor `B.static_extent(r)` equals `dynamic_extent`, then `X.static_extent(r)` equals `B.static_extent(r)` (if applicable).
  - If neither `A.static_extent(0)` nor `A.static_extent(1)` equals `dynamic_extent`, then `A.static_extent(0)` equals `A.static_extent(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]

Solve multiple triangular linear systems with triangular matrix on the left [linalg.alg.blas3.trsm.left]

Not-in-place left solve of multiple triangular systems

```
template<class in_matrix_1_t,
         class Triangle,
         class DiagonalStorage,
         class in_matrix_2_t,
         class out_matrix_t>
void triangular_matrix_matrix_left_solve(
  in_matrix_1_t A,
  Triangle t,
  DiagonalStorage d,
  in_matrix_2_t B,
  out_matrix_t X);
template<class ExecutionPolicy,
         class in_matrix_1_t,
         class Triangle,
         class DiagonalStorage,
         class in_matrix_2_t,
         class out_matrix_t>
void triangular_matrix_matrix_left_solve(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
```

```
  DiagonalStorage d,
  in_matrix_2_t B,
  out_matrix_t X);
```

- *Requires:* `A.extent(1)` equals `B.extent(0)`.
- *Constraints:* If `i,j` and `i,k` are in the domain of `X`, then the expression `X(i,j) -= A(i,k) * X(k,j)` is well formed.
- *Mandates:* If neither `A.static_extent(1)` nor `B.static_extent(0)` equals `dynamic_extent`, then `A.static_extent(1)` equals `B.static_extent(0)`.
- *Effects:* Assigns to the elements of `X` the result of solving the triangular linear system(s) AX=B for X.

In-place left solve of multiple triangular systems

```
template<class in_matrix_1_t,
         class Triangle,
         class DiagonalStorage,
         class inout_matrix_t>
void triangular_matrix_matrix_left_solve(
  in_matrix_1_t A,
  Triangle t,
  DiagonalStorage d,
  inout_matrix_t B);
template<class ExecutionPolicy,
         class in_matrix_1_t,
         class Triangle,
         class DiagonalStorage,
         class inout_matrix_t>
void triangular_matrix_matrix_left_solve(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
  DiagonalStorage d,
  inout_matrix_t B);
```

[Note:

This algorithm makes it possible to compute factorizations like Cholesky and LU in place.

Performing triangular solve in place hinders parallelization. However, other `ExecutionPolicy`-specific optimizations, such as vectorization, are still possible. This is why the `ExecutionPolicy` overload exists.

--end note]

- *Requires:* `A.extent(1)` equals `B.extent(0)`.
- *Constraints:*

  - If `DiagonalStorage` is `explicit_diagonal_t`, and `i,j` is in the domain of `B`, then the expression `B(i,j) /= A(i,i)` is well formed (if applicable).

  - If `i,j` and `i,k` are in the domain of `X`, then the expression `B(i,j) -= A(i,k) * B(k,j)` is well formed.

- *Mandates:* If neither `A.static_extent(1)` nor `B.static_extent(0)` equals `dynamic_extent`, then `A.static_extent(1)` equals `B.static_extent(0)`.

- *Effects:* Overwrites `B` with the result of solving the triangular linear system(s) AX=B for X.

Solve multiple triangular linear systems with triangular matrix on the right [linalg.alg.blas3.trsm.right]

Not-in-place right solve of multiple triangular systems

```
template<class in_matrix_1_t,
         class Triangle,
         class DiagonalStorage,
         class in_matrix_2_t,
         class out_matrix_t>
void triangular_matrix_matrix_right_solve(
  in_matrix_1_t A,
  Triangle t,
  DiagonalStorage d,
  in_matrix_2_t B,
  out_matrix_t X);
template<class ExecutionPolicy,
         class in_matrix_1_t,
         class Triangle,
         class DiagonalStorage,
         class in_matrix_2_t,
         class out_matrix_t>
void triangular_matrix_matrix_right_solve(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
  DiagonalStorage d,
  in_matrix_2_t B,
  out_matrix_t X);
```

- *Requires:* `A.extent(1)` equals `B.extent(1)`.

- *Constraints:* If `i,j` and `i,k` are in the domain of `X`, then the expression `X(i,j) -= X(i,k) * A(k,j)` is well formed.

- *Mandates:* If neither `A.static_extent(1)` nor `B.static_extent(1)` equals `dynamic_extent`, then `A.static_extent(1)` equals `B.static_extent(1)`.

- *Effects:* Assigns to the elements of `X` the result of solving the triangular linear system(s) XA=B for X.

In-place right solve of multiple triangular systems

```
template<class in_matrix_1_t,
         class Triangle,
         class DiagonalStorage,
         class inout_matrix_t>
void triangular_matrix_matrix_right_solve(
  in_matrix_1_t A,
  Triangle t,
  DiagonalStorage d,
  inout_matrix_t B);
template<class ExecutionPolicy,
         class in_matrix_1_t,
         class Triangle,
         class DiagonalStorage,
         class inout_matrix_t>
void triangular_matrix_matrix_right_solve(
  ExecutionPolicy&& exec,
  in_matrix_1_t A,
  Triangle t,
  DiagonalStorage d,
  inout_matrix_t B);
```

[Note:

This algorithm makes it possible to compute factorizations like Cholesky and LU in place.

Performing triangular solve in place hinders parallelization. However, other `ExecutionPolicy`-specific optimizations, such as vectorization, are still possible. This is why the `ExecutionPolicy` overload exists.

--end note]

- *Requires:* `A.extent(1)` equals `B.extent(1)`.
- *Constraints:*
  - If `DiagonalStorage` is `explicit_diagonal_t`, and `i,j` is in the domain of `B`, then the expression `B(i,j) /= A(i,i)` is well formed (if applicable).
  - If `i,j` and `i,k` are in the domain of `X`, then the expression `B(i,j) -= B(i,k) * A(k,j)` is well formed.
- *Mandates:* If neither `A.static_extent(1)` nor `B.static_extent(1)` equals `dynamic_extent`, then `A.static_extent(1)` equals `B.static_extent(1)`.
- *Effects:* Overwrites `B` with the result of solving the triangular linear system(s) XA=B for X.

Examples

Cholesky factorization

This example shows how to compute the Cholesky factorization of a real symmetric positive definite matrix A stored as a `basic_mdspan` with a unique non-packed layout. The algorithm imitates `DPOTRF2` in LAPACK 3.9.0.

If `Triangle` is `upper_triangle_t`, then it computes the Cholesky factorization A = U^T U. Otherwise, it computes the Cholesky factorization A = L L^T. The function returns 0 if success, else k+1 if row/column k has a zero or NaN (not a number) diagonal entry.

```
#include <linalg>
#include <cmath>

template<class inout_matrix_t,
         class Triangle>
int cholesky_factor(inout_matrix_t A, Triangle t)
{
  using element_type = typename inout_matrix_t::element_type;
  constexpr element_type ZERO {};
  constexpr element_type ONE (1.0);
  const ptrdiff_t n = A.extent(0);

  if (n == 0) {
    return 0;
  }
  else if (n == 1) {
    if (A(0,0) <= ZERO || isnan(A(0,0))) {
      return 1;
    }
    A(0,0) = sqrt(A(0,0));
  }
  else {
    // Partition A into [A11, A12,
    //                   A21, A22],
    // where A21 is the transpose of A12.
    const ptrdiff_t n1 = n / 2;
    const ptrdiff_t n2 = n - n1;
    auto A11 = subspan(A, pair{0, n1}, pair{0, n1});
    auto A22 = subspan(A, pair{n1, n}, pair{n1, n});

    // Factor A11
    const int info1 = cholesky_factor(A11, t);
    if (info1 != 0) {
      return info1;
    }

    using std::linalg::symmetric_matrix_rank_k_update;
    using std::linalg::transposed;
    if constexpr (std::is_same_v<Triangle, upper_triangle_t>) {
      // Update and scale A12
      auto A12 = subspan(A, pair{0, n1}, pair{n1, n});
      using std::linalg::triangular_matrix_matrix_left_solve;
      triangular_matrix_matrix_left_solve(transposed(A11),
        upper_triangle, explicit_diagonal, A12);
      // A22 = A22 - A12^T * A12
      symmetric_matrix_rank_k_update(-ONE, transposed(A12),
                                     A22, t);
    }
```

```
    else {
      //
      // Compute the Cholesky factorization A = L * L^T
      //
      // Update and scale A21
      auto A21 = subspan(A, pair{n1, n}, pair{0, n1});
      using std::linalg::triangular_matrix_matrix_right_solve;
      triangular_matrix_matrix_right_solve(transposed(A11),
        lower_triangle, explicit_diagonal, A21);
      // A22 = A22 - A21 * A21^T
      symmetric_matrix_rank_k_update(-ONE, A21, A22, t);
    }

    // Factor A22
    const int info2 = cholesky_factor(A22, t);
    if (info2 != 0) {
      return info2 + n1;
    }
  }
}
```

Solve linear system using Cholesky factorization

This example shows how to solve a symmetric positive definite linear system Ax=b, using the Cholesky factorization computed in the previous example in-place in the matrix `A`. The example assumes that `cholesky_factor(A, t)` returned 0, indicating no zero or NaN pivots.

```
template<class in_matrix_t,
         class Triangle,
         class in_vector_t,
         class out_vector_t>
void cholesky_solve(
  in_matrix_t A,
  Triangle t,
  in_vector_t b,
  out_vector_t x)
{
  using std::linalg::transposed;
  using std::linalg::triangular_matrix_vector_solve;

  if constexpr (std::is_same_v<Triangle, upper_triangle_t>) {
    // Solve Ax=b where A = U^T U
    //
    // Solve U^T c = b, using x to store c.
    triangular_matrix_vector_solve(transposed(A), t,
                                   explicit_diagonal, b, x);
    // Solve U x = c, overwriting x with result.
    triangular_matrix_vector_solve(A, t, explicit_diagonal, x);
  }
  else {
    // Solve Ax=b where A = L L^T
```

```
    //
    // Solve L c = b, using x to store c.
    triangular_matrix_vector_solve(A, t, explicit_diagonal, b, x);
    // Solve L^T x = c, overwriting x with result.
    triangular_matrix_vector_solve(transposed(A), t,
                                   explicit_diagonal, x);
  }
}
```

Compute QR factorization of a tall skinny matrix

This example shows how to compute the QR factorization of a "tall and skinny" matrix V, using a cache-blocked algorithm based on rank-k symmetric matrix update and Cholesky factorization. "Tall and skinny" means that the matrix has many more rows than columns.

```
// Compute QR factorization A = Q R, with A storing Q.
template<class inout_matrix_t,
         class out_matrix_t>
int cholesky_tsqr_one_step(
  inout_matrix_t A, // A on input, Q on output
  out_matrix_t R)
{
  // One might use cache size, sizeof(element_type), and A.extent(1)
  // to pick the number of rows per block.  For now, we just pick
  // some constant.
  constexpr ptrdiff_t max_num_rows_per_block = 500;

  using R_element_type = typename out_matrix_t::element_type;
  constexpr R_element_type ZERO {};
  for(ptrdiff_t i = 0; i < R.extent(0); ++i) {
    for(ptrdiff_t j = 0; j < R.extent(1); ++j) {
      R(0,0) = ZERO;
    }
  }

  // Cache-blocked version of R = R + A^T * A.
  const ptrdiff_t num_rows = A.extent(0);
  ptrdiff_t rest_num_rows = num_rows;
  auto A_rest = A;
  while(A_rest.extent(0) > 0) {
    const ptrdiff num_rows_per_block =
      min(A.extent(0), max_num_rows_per_block);
    auto A_cur = subspan(A_rest, pair{0, num_rows_per_block}, all);
    A_rest = subspan(A_rest,
      pair{num_rows_per_block, A_rest.extent(0)}, all);
    // R = R + A_cur^T * A_cur
    using std::linalg::symmetric_matrix_rank_k_update;
    symmetric_matrix_rank_k_update(transposed(A_cur),
                                   R, upper_triangle);
  }
```

```
  const int info = cholesky_factor(R, upper_triangle);
  if(info != 0) {
    return info;
  }
  using std::linalg::triangular_matrix_matrix_left_solve;
  triangular_matrix_matrix_left_solve(R, upper_triangle, A);
  return info;
}

// Compute QR factorization A = Q R.  Use R_tmp as temporary R factor
// storage for iterative refinement.
template<class in_matrix_t,
         class out_matrix_1_t,
         class out_matrix_2_t,
         class out_matrix_3_t>
int cholesky_tsqr(
  in_matrix_t A,
  out_matrix_1_t Q,
  out_matrix_2_t R_tmp,
  out_matrix_3_t R)
{
  assert(R.extent(0) == R.extent(1));
  assert(A.extent(1) == R.extent(0));
  assert(R_tmp.extent(0) == R_tmp.extent(1));
  assert(A.extent(0) == Q.extent(0));
  assert(A.extent(1) == Q.extent(1));

  copy(A, Q);
  const int info1 = cholesky_tsqr_one_step(Q, R);
  if(info1 != 0) {
    return info1;
  }
  // Use one step of iterative refinement to improve accuracy.
  const int info2 = cholesky_tsqr_one_step(Q, R_tmp);
  if(info2 != 0) {
    return info2;
  }
  // R = R_tmp * R
  using std::linalg::triangular_matrix_left_product;
  triangular_matrix_left_product(R_tmp, upper_triangle,
                                 explicit_diagonal, R);
  return 0;
}
```