A proposal to add linear algebra support to the C++ standard library

Document #: P1385R7
Date: 2022-10-15
Project: Programming Language C++
Target: SG6
Reply-to: Guy Davidson <guy.cpp.wg21@gmail.com>
Bob Steagall <bob.steagall.cpp@gmail.com>

Abstract

This document proposes a set of fundamental linear algebra types and functions for the standard C++ library. The facilities described herein are pure additions, requiring no changes to existing implementations.

Revision history

R0 Initial version for pre-Kona mailing.

D1 Update for presentation at Kona includes operation traits.

R1 Update for post-Kona mailing; includes feedback from LEWG(I) and joint SG14/SG19 session.

Feedback:

At the Kona 2019 meeting, draft version D1 of this paper was reviewed by LEWG(I) and a joint session of SG14 and SG19. Both reviews were generally positive, several good suggestions were made, and some polls regarding future directions were taken.

LEWG(I) Polls and Feedback

[Wednesday 2019-02-20]
(http://wiki.edg.com/bin/view/Wg21kona2019/P1385)

1. We want 0-based indexing as opposed to 1-based indexing.
   (unanimous: 20)
2. We like having separate row_vector and column_vector types in addition to matrix.
```
SF F N A SA (21 present)
 3 0 5 4  4
```
3. We want explicitly named operations (e.g., dot and outer) in addition to operators.
```
SF F N A SA (21 present)
```

```
    8 5 2 1  0
```
4. Define engine/matrix classes in terms of mdspan + storage and mdspan concepts (e.g., extents), and expose an mdspan-esque. This implies that fs_ and dyn_ are combined into one template parameterized on extents (which are either static or dynamic).
```
   SF F N A SA (22 present)
    6 2 7 0  0
```

There were some additional requests:
+ Provide some implementation and usage experience.
+ Provide a comparison with prior art.
+ Explore the re-usability of `mdspan` and `common_type`.
+ Be careful of allowing specializations of traits types that are part of namespace `std`; be consistent with other traits.

Joint SG14/SG19 Session Feedback

[Friday 2019-02-22]
(http://wiki.edg.com/bin/view/Wg21kona2019/SG14MinutesP1385)
+ Stick to 0-based indexing, for compatibility with current practice, and also for performance reasons.
+ Provide a fixed-size engine whose memory is dynamically allocated.
+ In this session, there was very broad agreement that the one-vector approach advocated by LEWG(I) was the way to proceed.
+ Outer product computation is rare in practice, so, the vector-vector multiplication operator should return the inner product, and the outer product should be a named function.

Other Suggestions Gathered at the Meeting
+ Experiment with executors for concurrent operations.
+ Include an "audience table" (see [P1362R0](http://wg21.link/p1362r0), Section 4.4) showing feature levels and anticipated user sophistication for each.
+ Include tutorial material on how the library can be used and extended, with several illustrative examples.

R2 Update for Cologne meeting; includes feedback from Kona and monthly SIG conference calls.

— Emphasized proposed std::math namespace
— Replaced row_vector and column_vector types with a single vector type to represent both.
— Removed discussion regarding 0-based or 1-based indexing in favor of 0-based.
— Reduced number of customization points within namespace std to two.

D3 Last-minute update for Cologne meeting.

— Remove erroneous references to row_vector and column_vector in the R2 text.

Feedback:

At the Cologne 2019 meeting, a joint session of SG14, SG19, and SG6 was held on Friday 20-Jul-2019 and version R2 of this paper was presented. A vote was held in the afternoon, and the room reached consensus to forward P1385 to LEWG subject to reconciling implementation with P1673.

Joint SG14/SG6 Session Feedback
[Friday 2019-07-20]
(http://wiki.edg.com/bin/view/Wg21cologne2019/SG6P1385R2)
[Additional session here]
(http://wiki.edg.com/bin/view/Wg21cologne2019/SG14LA)
Forward P1385 to LEWG subject to reconciling implementation with P1673

```
SF F N A SA
9 4 9 2  2
```

In the intervening months, the authors of P1673 put together an initial implementation of the interface described therein, and provided it to the authors of this proposal. We are currently endeavoring to implement P1385 in terms of the interface expressed by P1673.

R3 Update for Belfast meeting.

— Remove more erroneous references to row_vector and column_vector.

Feedback:

The paper was reviewed by SG6 briefly and by LEWG(I). SG6 only require further review if a numeric question arises.

LEWG(I) Polls and Feedback [Wednesday 2020-11-06](http://wiki.edg.com/bin/view/Wg21belfast/P1385)

1. We want to be able to modify elements through `matrix_*_view`s (similar to `span`).

```
SF F N A SA (20 present)
 8 8 0 0  0
```

2. For matrix types with overloaded operators, we are comfortable with supporting hooks for expression templates but having no expression templates by default in the standard library.

```
SF F N A SA (20 present)
 2 4 4 2  0
```

3. Given what we've seen so far, we are comfortable with the customization mechanisms for overloaded operators on matrices.

```
SF F N A SA (20 present)
 2 4 7 0  2
```

CONSENSUS: Bring a revision with the guidance below to LEWGI for further design review.
+ These pieces of guidance from Kona seem not to have been addressed. Please address them in the next revision of the paper.

  - Define engine/matrix classes in terms of mdspan & storage and mdspan concepts (e.g. extents), and expose an mdspan-esque interface. This implies that fs_ and dyn_ are combined into one template parameterized on extents (which are either static or dynamic).
  - Add explicitly named operations (e.g. dot, outer) in addition to operators.
+ Ask SG6 to formulate a precise definition of `is_field`, `is_nc_ring` and `is_ring`, and consider what the answers should be for builtin types (e.g. signed integers, unsigned integers, and floating point types).
+ Make `numeric_traits` non-implementation-defined; for non-builtin/non-`std` types, assume the properties are false.
+ Separate `is_complex` or `is_specialization_of` into a separate paper.
+ Remove the `numeric_traits` helper traits (`is_field`, `is_nc_ring`, etc).
+ Make `matrix_*_view`'s copy constructor and copy assignment methods `noexcept`.
+ Add non-const `begin` and `end` to `matrix_*_view`.
+ Remove `assign` method from `matrix_*_view`.
+ Provide a way to modify elements through `matrix_*_view` types: one option is to add `matrix_*_ref` types that allow modification of the underlying elements.
+ Explore adding a submatrix view.
+ Make `index_type` `ptrdiff_t` and `size_type` `size_t`.
+ Develop a clear plan for `is_rectangular == false`.
+ Add `data` to engine types.
+ `is_fixed_size` and `is_resizable` are inverses of each other; explore combining.
+ Explore different designs for `numeric_traits`: either looking for embedded type aliases in the classes instead of a trait, or granular traits a la P1370.

R4 Update to R3 for post-Belfast mailing.

— Include feedback from reviews in Belfast.

R5 Update for pre-Prague mailing, based on feedback from Belfast.

— Removed element type predicate traits from the public interface.
— Removed is_complex from the public interface.
— Added mutating row, column, transpose, and submatrix “views” (in addition to the corresponding const “views”).
— Changed type of NTTPs for sizes to size_t.
— Changed index_type to size_type for indexing.
— Changed names formerly *_view to *_engine.
— Removed matrix_ prefix from non-owning engine names.
— Removed nested boolean attributes from engines and math objects.
— Renamed const_*_tag and mutable_*_tag tag types to readable_*_tag and writable_*_tag, respectively.

Feedback:

The paper was reviewed by SG6/SG14/SG19 briefly and by LEWG(I).

LEWG(I) Polls and Feedback [Wednesday 2020-11-06](http://wiki.edg.com/bin/view/Wg21prague/P1385)

1. We want `operator*(std::math::vector, std::math::vector)` in the standard library, even though some believe it is ambiguous.

```
SF F N A SA (25 present)
 5 6 2 1  6 (no consensus)
```

2. We want `operator*(matrix, vector)` `operator*(matrix, matrix)` in the standard library even if we won't have `operator*(vector, vector)`.

```
SF F N A SA (25 present)
10 4 2 3  2 (consensus)
```

3. We want overloaded operators (e.g. `operator*`, `operator+`, etc) for matrix/vector operations in the standard library.

```
SF F N A SA (25 present)
10 4 3 2  2 (consensus)
```

4. Assuming we have overloaded operators (e.g. `operator*`, `operator+`, etc) for matrix/vector operations in the standard library, we want their semantics to be customizable by users.

```
SF F N A SA (25 present)
 5 4 8 1  3 (no consensus)
```

5. We are okay with only providing submatrices/slices with non-owning semantics in the first version we ship.

```
SF F N A SA (17 present)
 8 4 2 2  0 (consensus)
```

CONSENSUS: Bring a revision of D1385R6 (DSL Linear Algebra Library), with the guidance below, to LEWGI for further design review.

+ Add further justification for why matrix operations need to be customizable by users; this should be a focus of the next discussion in an effort to increase consensus.
+ Remove `operator*(std::math::vector, std::math::vector)`.
+ `swap` on engines should be `noexcept`.
+ Bikeshed "view" in this paper on the LEWG mailing list.
+ Bikeshed the `t` and `h` member functions on the LEWG mailing list.
+ Make `is_resizable` a constexpr inline variable instead of a constepxr static function.
+ Use `extents` instead of `size_tuple`.
+ Explore alternatives to `initializer_list<initializer_list<>>` that enforce that the dimensions of all the inner lists are identical (follow-up with Eric Fiselier).
+ Demonstrate and clarify how the engine categories work and how you could use them to write generic functions that accept matrices of certain categories (follow up with Gašper Ažman).

R6 Update for post-Prague mailing, incorporating remaining feedback from Belfast.

— Added initable_*_tag to specify engine types for which construction and assignment from an initializer_list are acceptable.
— Removed iteration from the public interfaces of vector and all vector engines.

— Added free function templates begin(), end(), etc. to provide iteration over the elements of a vector object.
— Added support for basic_mdspan for the engine types, vector, and matrix.
— Reduced the number of non-owning, view-style engine types to two: vector_view_engine and matrix_view_engine (and consequently removed row_engine, column_engine, and transpose_engine).
— Added function templates inner_product() and outer_product().

R7 Update for 2022-10 mailing

— Expanded motivation to highlight the difference between an array and a matrix.
— Reduced design to withdraw the vector class and unify around a single matrix class.

Open issues

— Develop tutorial materials and examples (including examples demonstrating how to build engines and traits based on expression engines).
— Add wording and requirements tables.
— Add an audience table.
— Integrate BLAS interface from P1673 into reference implementation.

Introduction

Linear algebra is a mathematical discipline of ever-increasing importance, with direct application to a wide variety of problem domains, such as signal processing, computer graphics, medical imaging, scientific simulations, machine learning, analytics, financial modeling, and high-performance computing. And yet, despite the relevance of linear algebra to so many aspects of modern computing, the C++ standard library does not include any linear algebra facilities. This paper proposes to remedy this deficit for C++26.

This paper should be read after P1166, in which we describe a high-level set of expectations for what a linear algebra library should contain.

Goals

We expect that typical users of a standard linear algebra library are likely to value two features above all else: ease-of-use (including expressiveness), and high performance out of the box. This set of users will expect the ability to compose arithmetical expressions of linear algebra objects similar to what one might find in a textbook; indeed, this has been deemed a "must-have" feature by several participants in SG14 Linear Algebra SIG conference calls. And for a given arithmetical expression, they will expect run-time computational performance that is close to what they could obtain with an equivalent sequence of function calls to a more "traditional" linear algebra library, such as LAPACK, Blaze, Eigen, etc.

There also exists a set of linear algebra “super-users” who will value most highly a third feature – the ability to customize underlying infrastructure in order to maximize performance for specific problems and computing platforms. These users seek the highest possible run-time performance, and to achieve it, require the ability to customize any and every portion of the library's computational infrastructure.

With these high-level user requirements in mind, in this paper we propose an interface specification intended to achieve the following goals:

1. To provide a matrix vocabulary types for representing the mathematical objects and fundamental operations relevant to linear algebra;

2. To provide a public interface for linear algebra expressions that is intuitive, teachable, and mimics the expressiveness of traditional mathematical notation to the greatest reasonable extent;

3. To exhibit out-of-the-box performance in the neighborhood of that of that exhibited by an equivalent sequence of function calls to a more traditional linear algebra library, such as LAPACK, Blaze, Eigen, etc.;

4. To provide a set of building blocks that manage the source, ownership, lifetime, layout, and access of the memory required to represent the linear algebra vocabulary types;

5. To provide straightforward facilities and techniques for customization that enable users to optimize performance for their specific problem domain on their specific hardware; and,

6. To provide a reasonable level of granularity for customization so that developers only have to implement a minimum set of types and functions to integrate their performance enhancements with the rest of the linear algebra facilities described here.

Definitions

When discussing linear algebra and related topics for a proposal such as this, it is important to note that there are several overloaded terms (such as *matrix*, *vector*, *dimension*, and *rank*) which must be defined and disambiguated if such discussions are to be productive. These terms have specific meanings in mathematics, as well as different, but confusingly similar, meanings to C++ programmers.

In the following sections we provide definitions for relevant mathematical concepts, C++ type design concepts, and describe how this proposal employs those overloaded terms in various contexts.

Mathematical terms

In order to facilitate subsequent discussion, we first provide the following *informal* set of definitions for important mathematical concepts:

1. A **vector space** is a collection of **vectors**, where vectors are objects that may be added together and multiplied by scalars. Euclidean vectors are an example of a vector space,

typically used to represent displacements, as well as physical quantities such as force or momentum. Linear algebra is concerned primarily with the study of vector spaces.

2. The **dimension** of a vector space is the minimum number of coordinates required to specify any point within the space.

3. A **matrix** is a rectangular array of numbers, symbols, or expressions, arranged in rows and columns. A matrix having *m* rows and *n* columns is said to have size *m* x *n*. Although matrices can be used to solve systems of simultaneous linear equations, they are most commonly used to represent linear transformations, solve linear least squares problems, and to explore and/or manipulate the properties of vector spaces.

4. A **vector** is a matrix with only one row or one column. Although the vector is traditionally introduced as a tuple of scalars, and a matrix as a tuple of vectors, as far as theorems of linear algebra are concerned there is no difference between a single column matrix and a column vector, nor a single row matrix and a row vector.

5. The **rank** of a matrix is the dimension of the vector space spanned by its columns, which is equal to the dimension of the vector space spanned by its rows. The rank is also equal to the maximum number of linearly-independent columns and rows.

6. An **element** of a matrix is an individual member (number, symbol, expression) of the rectangular array comprising the matrix, lying at the intersection of a single row and a single column. In traditional mathematical notation, row and column indexing is 1-based, where rows are indexed from 1 to *m* and columns are indexed from 1 to *n*. Given some matrix *A*, element a_{11} refers to the element in the upper left-hand corner of the array and element a_{mn} refers to the element in the lower right-hand corner.

7. A **row vector** is a matrix containing a single row; in other words, a matrix of size *1* x *n*. In many applications of linear algebra, row vectors represent spatial vectors.

8. A **column vector** is a matrix containing a single column; in other words, a matrix of size *m* x *1*. In many applications of linear algebra, column vectors represent spatial vectors.

9. **Element transforms** are non-arithmetical operations that modify the relative positions of elements in a matrix, such as transpose, column exchange, and row exchange.

10. **Element arithmetic** refers to arithmetical operations that read or modify the values of individual elements independently of other elements, such as assigning a value to a specific element or multiplying a row by some value.

11. **Matrix arithmetic** refers to the assignment, addition, subtraction, negation, multiplication, and determinant operations defined for matrices, row vectors, and column vectors as wholes.

12. A **rectangular matrix** is a matrix requiring a full *m* x *n* representation; that is, a matrix not possessing a special form, such as identity, triangular, band, etc.

13. A **square matrix** is a matrix where the number of rows equals the number of columns.

14. The **identity matrix** is a square matrix where all elements on the diagonal are equal to one and all off-diagonal elements are equal to zero.

15. A **triangular matrix** is a matrix where all elements above or below the diagonal are zero; those with non-zero elements above the diagonal are called *upper triangular*, while those with non-zero elements below the diagonal are called *lower triangular*.

16. A **band matrix** is a sparse matrix whose non-zero entries are confined to a diagonal band, lying on the main diagonal and zero or more diagonals on either side.

17. **Decompositions** are complex sequences of arithmetic operations, element arithmetic, and element transforms performed upon a matrix that expose important mathematical properties of that matrix. Several types of decomposition are often performed in solving least-squares problems.

18. **Eigen-decompositions** are decompositions performed upon a symmetric matrix in order to compute the eigenvalues and eigenvectors of that matrix; this is often performed when solving problems involving linear dynamic systems.

Product

The operations on a matrix of addition and subtraction, along with scalar multiplication, are carried out element-wise and are commutative. The product operation is not carried out element-wise and is therefore not commutative.

Given a product A.B=C, the result C can only be calculated where the number of columns in matrix A is the same as the number of rows in matrix B. To calculate the product requires the use of the dot product.

The dot product is calculated by memberwise multiplication of the elements of a row vector by the elements of a column vector, and summing the results.

(Sum i = 1-c, AiBi)

Each element of the result matrix is the dot product of the corresponding row and column vectors of the operands, thus:

```
A = a11 a12 a13    B = b11 b12 b13    C = ar1.bc1 ar1.bc2 ar1.bc3
    a21 a22 a23        b21 b22 b23        ar2.bc1 ar2.bc2 ar2.bc3
    a31 a32 a33        b31 b32 b33        ar3.bc1 ar3.bc2 ar3.bc3
```

When a matrix consisting of a single row is multiplied by a matrix consisting of a single column, this is called an inner product, thus:

```
A = a11 a12 a13    B = b11            C = ar1.bc1
```

```
                          b21
                         b31
```

The result is a matrix with a single row and a single column, which is a scalar value. When a matrix consisting of a single column is multiplied by a matrix consisting of a single row, this is called an outer product, thus:

```
A = a11              B = b11 b12 b13    C = ar1.bc1 ar1.bc2 ar1.bc3
    a21                                     ar2.bc1 ar2.bc2 ar2.bc3
    a31                                     ar3.bc1 ar3.bc2 ar3.bc3
```

The result is a matrix whose row count is that of matrix A, and whose column count is that of matrix B. The dot product used to calculate the value of each element is a single memberwise multiplication; no summation is required.

Terms pertaining to C++ types

The following are terms used in this proposal that describe various aspects of how the mathematical concepts described above in Section 3.1 might be implemented:

1. An **array** is a data structure representing an indexable collection of objects (elements) such that each element is identified by at least one index. An array is said to be *one-dimensional* if its elements are accessible with a single index; a *multi-dimensional* array is an array for which more than one index is required to access its elements.

2. The **dimension** of an array refers to the number of indices required to access an element of that array. The **rank** of an array is a synonym for its dimension.

3. This proposal uses the term **MathObj** to refer generically to one of the C++ types described herein representing matrices (i.e.,`matrix`). These are the public-facing types developers will use in their code.

4. An **engine** is an implementation type that manages the resources associated with a *MathObj* instance. This includes, at a minimum, the storage-related aspects of, and access to, the elements of a *MathObj*. It could also include execution-related aspects, such as an execution context. In this proposal, an engine object is a private member of a *MathObj*. Other than as a template parameter, engines are not part of a *MathObj*'s public interface.

5. The adjective **dense** refers to a *MathObj* representation where storage is allocated for every element.

6. The adjective **sparse** refers to a *MathObj* representation where storage is allocated only for non-zero elements;

7. **Storage** is used by this proposal as a synonym for memory.

8. **Traits** refers to a stateless class template that provides some set of services, normalizing those services over its set of template parameters.

9. **Row size** and **column size** refer to the number of rows and columns, respectively, that a *MathObj* represents, which must be less than or equal to its row and column capacities, defined below.

10. **Row capacity** and **column capacity** refer to the maximum number of rows and columns, respectively, that a *MathObj* can possibly represent.

11. **Fixed-size** (FS) refers to an engine type whose row and column sizes are fixed at instantiation time and constant thereafter.

12. **Fixed-capacity** (FC) refers to an engine type whose row and column capacities are fixed at instantiation time and constant thereafter.

13. **Dynamically re-sizable** (DR) refers to an engine type whose row and column sizes and capacities may be changed at run time.

Overloaded terms

This section describes how we use certain overloaded terms in this proposal and in future works.

Matrix

The term *matrix* is frequently used by C++ programmers to mean a general-purpose array of arbitrary size. For example, one of the authors worked at a company where it was common practice to refer to 4x4 arrays as "4-dimensional matrices."

In this proposal, we use the word *array* only to mean a data structure whose elements are accessible using one or more indices, and which has no invariants pertaining to higher-level or mathematical meaning.

We use *matrix* to mean the mathematical object as defined above in Section 3.1, and `` `matrix` `` (in monospaced font) to mean the C++ class template that implements the mathematical object. We sometimes use `` `MathObj` `` (in monospaced font) in some of the component interface code and text below to generically refer to a `` `matrix` `` object.

Vector

Likewise, many C++ programmers incorrectly use the term *vector* as a synonym for "dynamically re-sizable array." This bad habit is reinforced by the unfortunate naming of `std::vector`.

This proposal uses the term *vector* to mean an element of a vector space, per Section 3.1 above. Further, we also mean *vector* generically to have both of the meanings set out in 3.1

Dimension

In linear algebra, a vector space *V* is said to be of *dimension n*, or be *n-dimensional*, if there exist *n* linearly independent vectors which span *V*. This is another way of saying that *n* is the minimum number of coordinates required to specify any point in *V*. However, in common programming parlance, *dimension* refers to the number of indices used to access an element in an array.

We use the term dimension both ways in this proposal, but try to do so consistently and in a way that is clear from the context. For example, a rotation matrix used by a game engine is a two-dimensional data structure composed of three-dimensional row and column vectors. A vector describing an electric field is an example of a one-dimensional data structure that could be implemented as a three-dimensional column vector.

Rank

The *rank* of a matrix is the dimension of the vector space spanned by its columns (or rows), which corresponds to the maximal number of linearly independent columns (or rows) of that matrix. Rank also has another meaning in tensor analysis, where it is commonly used as a synonym for a tensor's *order*.

However, rank also has a meaning in computer science where it is used as a synonym for dimension. In the C++ standard at [*meta.unary.prop.query*], rank is described as the number of dimensions of `` `T` `` if `` `T` `` names an array, otherwise it is zero.

We avoid using the term *rank* in this proposal in the context of linear algebra, except as a quantity that might result from performing certain decompositions wherein the mathematical rank of a matrix is computed.

Scope

We contend that the best approach for standardizing a set of linear algebra components for C++23 will be one that is layered, iterative, and incremental. This paper is quite deliberately a "basic linear algebra-only" proposal; it describes what we believe is a foundational layer providing the minimum set of components and arithmetic operations necessary to provide a reasonable, basic level of functionality.

Higher-level functionality can be specified in terms of the interfaces described here, and we encourage succession papers to explore this possibility.

Functional requirements

The foundational layer, as described here, should include the minimal set of types and functions required to perform matrix arithmetic in finite dimensional spaces. This includes:

+ A matrix class template;

+ Arithmetic operations for addition, subtraction, negation, and multiplication of matrices;

+ Arithmetic operations for scalar multiplication of matrices;

+ Well-defined facilities for integrating new element types;

+ Well-defined facilities for creating and integrating custom engines; and,

+ Well-defined facilities for creating and integrating custom arithmetic operations.

Considered but excluded

Tensors
There has been a great deal of interest expressed in specifying an interface for general-purpose tensor processing in which linear algebra facilities fall out as a special case. We exclude this idea from this proposal for two reasons. First, given the practical realities of standardization work, the enormous scope of such an effort would very likely delay introduction of linear algebra facilities until C++29 or later.

Second, and more importantly, implementing matrices as derived types or specializations of a general-purpose tensor type is bad type design. Consider the following: a tensor is (informally) an array of mathematical objects (numbers or functions) such that its elements transform according to certain rules under a coordinate system change. In a *p*-dimensional space, a tensor of rank *n* will have p^n elements. In particular, a rank-2 tensor in a *p*-dimensional space may be represented by a *p* x *p* matrix having certain invariants related to coordinate transformation not possessed by all *p* x *p* matrices.

These defining characteristics of a tensor lead us to the crux of the issue: every rank-2 tensor can be represented by a square matrix, but not every square matrix represents a tensor. As one quickly realizes, only a small fraction of all possible matrices are representations of rank-2 tensors.

All of this is a long way of saying that the class invariants governing a matrix type are quite different from those governing a tensor type, and as such, the public interfaces of such types will also differ substantially.

From this we conclude that matrices are not Liskov-substitutable for rank-2 tensors, and therefore as matter of good type design, matrices and tensors should be implemented as distinct types, perhaps with appropriate inter-conversion operations.

This situation is analogous to the age-old object-oriented design question: when designing a group of classes that represent geometric shapes, is a square a kind of rectangle? In other words, should class `square` be publicly derived from class `rectangle`? Mathematically, yes, a square *is* a rectangle. But from the perspective of good interface design, class `square` is not substitutable for class `rectangle` and is usually best implemented as a distinct type having no IS-A relationship with `rectangle`.

Quaternions and octonions

There has also been interest expressed in including other useful mathematical objects, such as quaternions and octonions, as part of a standard linear algebra library. Although element storage for these types might be implemented using the engines described in this proposal, quaternions and octonions represent mathematical concepts that are fundamentally different from those of matrices and vectors.

As with tensors, the class invariants and public interfaces for quaternions and octonions would be substantially different from that of the linear algebra components. Liskov substitutability would not be possible, and therefore quaternions and octonions should be implemented as types distinct from the linear algebra types.

Design aspects

The following describe several important aspects of the problem domain affecting the design of the proposed interface. Importantly, these aspects are orthogonal, and are addressable through judicious combinations of template parameters and implementation type design.

Memory source

Perhaps the first question to be answered is that of the source of memory in which elements will reside. One can easily imagine multiple sources of memory:

+ Elements reside in an external buffer allocated from the global heap.

+ Elements reside in an external buffer allocated by a custom allocator and/or specialized heap.

+ Elements reside in an external fixed-size buffer that exists independently of the *MathObj*, not allocated from a heap, and which has a lifetime greater than that of the *MathObj*.

+ Elements reside in a fixed-size buffer that is a member of the *MathObj* itself.

+ Elements reside collectively in a set of buffers distributed across multiple machines.

Addressing model

It is also possible that the memory used by a *MathObj* might be addressed using what the standard calls a *pointer-like type*, also known as a *fancy pointer*.

For example, consider an element buffer existing in a shared memory segment managed by a custom allocator. In this case, the allocator might employ a fancy pointer type that performs location-independent addressing based on a segment index and an offset into that segment.

One can also imagine a fancy pointer that is a handle to a memory resource existing somewhere on a network, and addressing operations require first mapping that resource into the local address space, perhaps by copying over the network or by some magic sequence of RPC invocations.

Memory ownership

The next important questions pertain to memory ownership. Should the memory in which elements reside be deallocated, and if so, what object is responsible for performing the deallocation?

A *MathObj* might own the memory in which it stores its elements, or it might employ some non-owning view type, like `mdspan`, to manipulate elements owned by some other object.

Capacity and resizability

As with `std::string` and `std::vector`, it is occasionally useful for a *MathObj* to have excess storage capacity in order to reduce the number of re-allocations required by anticipated future resizing operations. Some linear algebra libraries, like LAPACK, account for the fact that a *MathObj*'s capacity may be different than its size. This capability was of critical importance to the success of one author's prior work in functional MRI image analysis.

In other problem domains, like Cartesian geometry, *MathObj*s are small and always of the same size. In this case, the size and capacity are equal, and there is no need for a *MathObj* to maintain or manage excess capacity.

Element layout

There are many ways to arrange the elements of a matrix in memory, the most common in C++ being row-major dense rectangular. In Fortran-based libraries, the two-dimensional arrays used to represent matrices are usually column-major. There are also special arrangements of elements for upper/lower triangular and banded diagonal matrices that are both row-major and column-major. These arrangements of elements have been well-known for many years, and libraries like LAPACK in the hands of a knowledgeable user can use them to implement code that is optimal in both time and space.

Element access and indexing

In keeping with the goal of supporting a natural syntax, and in analogy with the indexing operations provided by the random-access standard library containers, it seems reasonable to provide both const and non-const indexing for reading and writing individual elements.

Element type

C++ supports a relatively narrow range of arithmetic types, lacking direct support for arbitrary precision numbers and fixed-point numbers, among others. Libraries exist to implement these types, and they should not be precluded from use in a standard linear algebra library. It is possible that individual elements of a *MathObj* may allocate memory, and therefore an implementation cannot assume that element types have trivial constructors or destructors.

Mixed-element-type expressions

In general, when multiple built-in arithmetic types are present in an arithmetical expression, the resulting type will have a precision greater than or equal to that of the type with greatest precision in the expression. In other words, to the greatest reasonable extent, information is preserved.

We contend that a similar principle should apply to expressions involving *MathObj*s where more than one element type is present. Arithmetic operations involving *MathObj*s should, to the greatest reasonable extent, preserve element-wise information.

For example, just as the result of multiplying a `float` by a `double` is a `double`, the result multiplying a matrix-of-`float` by a matrix-of-`double` should be a matrix-of-`double`. We call the process of determining the resulting element type **element promotion**.

Mixed-engine expressions

In analogy with element type, *MathObj* expressions may include mixed storage management strategies, as implemented by their corresponding engine types. For example, consider the case of a fixed-size matrix multiplied by a dynamically-resizable matrix. What is the engine type of the resulting matrix?

Expression involving mixed engine types should not limit the availability of basic arithmetic operations. This means that there should be a mechanism for determining the engine type of the result of such expressions. We call the process of determining the resulting engine type **engine promotion**.

We contend that in most cases, the resulting engine type should be at least as "general" as the most "general" of the two engine types. For example, one could make the argument that a dynamically-resizable engine is more general than a fixed-size engine, and therefore the resulting engine type in an expression involving both these engine types should be a dynamically-resizable engine.

However, there are cases in which it may be possible to choose a more performant engine at compile time. For example, consider the case adding a fixed-size matrix and a dynamically-resizable matrix. Although size checking must be performed at run time, the resulting engine might be specified as fixed-size.

Arithmetic customization

In pursuit of optimal performance, developers may want to customize specific arithmetic operations, such as matrix-matrix or matrix-vector multiplication. Customization might be based on things like element layout in memory, fixed-size -vs- dynamically resizable, special hardware capabilities, etc.

One such possible optimization is the use of multiple cores, perhaps distributed across a network, to carry out multiplication on very large pairs of matrices, particularly in situations where the operation is used to produce a third matrix rather than modify one of the operands; the matrix multiplication operation is particularly amenable to this approach.

Developers may also wish to make use of SIMD intrinsics to enable parallel evaluation of matrix multiplication. This is common in game development environments where programs are written for very specific platforms, where the make and model of processor is well defined. This would impact on element layout and storage. Such work has already been demonstrated in paper N4454.

It is possible that two operands may be associated with different arithmetic customizations. We call the process of determining which of those two customizations to employ when performing the actual arithmetic operations **operation traits promotion**.

Linear algebra and `constexpr`

The fundamental set of operations for linear algebra can all be implemented in terms of a subset of the algorithms defined in the `<algorithm>` header, all of which are marked `constexpr` since C++20. Matrix and vector initialization is of course also possible at compile time for objects whose sizes are known at compile time.

Interface description

In this section, we describe the various types, operators, and functions comprising the proposed interface. The reader should note that the descriptions below are by no means ready for wording; rather, they are intended to foster further discussions and refinements, and to serve as a guide for hardy souls attempting to build implementations from this specification.

Overview

At the highest level, the interface is divided into four broad categories:

1. **Engines**, which are implementation types that manage the resources associated with a *MathObj* instance, including memory ownership and lifetime, as well as element access; and,

2. **MathObjs**, which provide a unified interface intended to model a corresponding mathematical abstraction (i.e. `matrix`);

3. **Operators**, which provide the desired mathematical syntax and carry out the promised arithmetic.

4. **Operation traits** act as a "container" for element promotion, engine promotion, and arithmetic traits (described below) and provide the "glue" that connects the engines, *MathObjs*, and the operators. This traits type is a template parameter to the *MathObj* types, and provides a way to inform an operator of the set of available arithmetic traits to be used when deciding how to perform an arithmetic operation.

At a lower level are a number of supporting traits types employed by the operation traits to determine the return type of the operator and perform the corresponding arithmetic operation. There are several such traits types:

+ **Element promotion traits** determine the resulting element type of an arithmetic operation involving two *elements*.

+ **Engine promotion traits** determine the resulting engine type of an arithmetic operation involving *matrix* objects. As part of that process, this traits type uses the element promotion traits to determine the element type of the resulting engine.

+ **Arithmetic traits** determine the type and value of a *MathObj* resulting from an arithmetical operation. As part of that process, this traits type uses the engine promotion traits to determine the engine type of the resulting *MathObj*. Having determined the result type, the arithmetic traits also have a member function that carries out the actual computations.

And finally, **operation selector traits** provide the means by which an arithmetic operator selects the operation traits that will perform the arithmetic. In the case where each operand has the same operation traits, the decision is simple. However, it is possible that the operands may be instantiated with different operation traits types, and so the operator uses the operation selector traits to decide which operation traits type to use for computing its result. The proposed traits class `std::matrix_operation_traits` is a library customization point.

Template parameter nomenclature

In order to avoid excessive visual noise in the code displayed in subsequent sections of this paper, we use the following abbreviation-based naming conventions for template parameters:

+ Parameter names `T`, `T1`, `T2`, `U`, `U1`, and `U2` represent element types.

+ Parameter names `ET`, `ET1`, and `ET2` represent engine types.

+ Parameter names `OT`, `OT1`, and `OT2` represent operation traits types.

+ Parameter names `OP`, `OP1`, and `OP2` represent the operand types deduced by an arithmetic operator.

+ Parameter names `AT`, `AT1`, and `AT2` represent allocator types.

+ Parameter names `C`, `C1`, and `C2` represent the number of columns in a fixed-size matrix or matrix engine.

+ Parameter names `R`, `R1`, and `R2` represent the number of rows in a fixed-size matrix or matrix engine.

+ Parameter name `MCT` represents a matrix engine's category tag type.

+ Parameter name `VFT` represents a view engine's functionality type (e.g., row, column, submatrix, etc.).

Header <linear_algebra> synopsis

```
#include <cstdint>
#include <complex>
#include <initializer_list>
#include <mdspan>
#include <tuple>
#include <type_traits>

namespace std {
//- Tags that describe engines and their capabilities.
//
struct scalar_engine_tag;

struct readable_matrix_engine_tag;
struct writable_matrix_engine_tag;
struct initable_matrix_engine_tag;
struct resizable_matrix_engine_tag;

//- A trivial engine that represents a scalar operand.
//
template<class T>
struct scalar_engine;

//- Owning engines with fixed-size internal storage.
//
template<class T, size_t R, size_t C>
class fs_matrix_engine;

//- Owning engines with dynamically-allocated external storage.
//
template<class T, class AT = allocator<T>>
class dr_matrix_engine;

//- Non-owning, view-style engine; tag to distinguish partial
specializations
//  of them; and related alias templates.
```

```
//
template<class ET, class MCT, class VFT>
class matrix_view_engine;

struct subvector_view_tag;
struct column_view_tag;
struct row_view_tag;
struct submatrix_view_tag;
struct transpose_view_tag;

template<class ET, class VCT>
using column_engine = matrix_view_engine<ET, MCT,
column_view_tag>;

template<class ET, class VCT>
using row_engine = matrix_view_engine<ET, MCT, row_view_tag>;

template<class ET, class MCT>
using submatrix_engine = matrix_view_engine<ET, MCT,
submatrix_view_tag>;

template<class ET, class MCT>
using transpose_engine = matrix_view_engine<ET, MCT,
transpose_view_tag>;

//- The default element promotion, engine promotion, and
arithmetic operation
//  traits for the four basic arithmetic operations.
//
struct matrix_operation_traits;

//- The primary math object type, matrix.
//
template<class ET, class OT=matrix_operation_traits>
class matrix;

//- Math object element promotion traits, per arithmetical
operation.
//
template<class T1>
struct matrix_negation_element_traits;
template<class T1, class T2>
struct matrix_addition_element_traits;
template<class T1, class T2>
struct matrix_subtraction_element_traits;
template<class T1, class T2>
struct matrix_multiplication_element_traits;
```

```
//- Math object engine promotion traits, per arithmetical
operation.
//
template<class OT, class ET1>
struct matrix_negation_engine_traits;
template<class OT, class ET1, class ET2>
struct matrix_addition_engine_traits;
template<class OT, class ET1, class ET2>
struct matrix_subtraction_engine_traits;
template<class OT, class ET1, class ET2>
struct matrix_multiplication_engine_traits;

//- Math object arithmetic traits, per arithmetical operation.
//
template<class OT, class OP1>
struct matrix_negation_traits;
template<class OT, class OP1, class OP2>
struct matrix_addition_traits;
template<class OT, class OP1, class OP2>
struct matrix_subtraction_traits;
template<class OT, class OP1, class OP2>
struct matrix_multiplication_traits;

//- A traits type that chooses between two operation traits types
in the binary
//  arithmetic operators and free functions that act like binary
operators.
//  This traits class is a customization point.
//
template<class OT1, class OT2>
struct matrix_operation_traits_selector;

//- Addition operator
//
template<class ET1, class OT1, class ET2, class OT2>
auto  operator +(matrix<ET1, OT1> const& m1, matrix<ET2, OT2>
const& m2);

//- Subtraction operator
//
template<class ET1, class OT1, class ET2, class OT2>
auto  operator -(matrix<ET1, OT1> const& m1, matrix<ET2, OT2>
const& m2);

//- Negation operator
//
template<class ET1, class OT1, class ET2, class OT2>
auto  operator -(matrix<ET1, OT1> const& m1);
```

```
//- Matrix*Scalar multiplication operators
//
template<class ET1, class OT1, class S2>
auto  operator *(matrix<ET1, OT1> const& m1, S2 const& s2);

template<class S1, class ET2, class OT2>
auto  operator *(S1 const& s1, matrix<ET2, OT2> const& m2);

//- Matrix*Matrix multiplication operator
//
template<class ET1, class OT1, class ET2, class OT2>
auto  operator *(matrix<ET1, OT1> const& m1, matrix<ET2, OT2>
const& m2);

//- Convenience aliases for matrix objects based on
//  dynamically-resizable engines.
//
template<class T, class AT = allocator<T>>
using dyn_matrix = matrix<dr_matrix_engine<T, AT>,
matrix_operation_traits>;

//- Convenience aliases for matrix objects based on fixed-size
engines.
//
template<class T, int32_t R, int32_t C>
using fs_matrix = matrix<fs_matrix_engine<T, R, C>,
matrix_operation_traits>;

}   //- namespace std
```

Engine Types

The over-arching purpose of the engine types is to perform resource management on behalf of an associated *MathObj* instance that owns the engine. At a minimum, all of the engine types provide a basic interface for const element indexing, determining row and column sizes, and determining row and column capacities. They also export public type aliases which specify their element type, whether or not they are dense, whether or not they are rectangular, whether or not they are resizable, whether or not their memory layout is row-major, and a 2-tuple for describing sizes and capacities.

It is important to note that an engine's resource management duties are primarily related to storage. To that end, an engine may own the storage it manages and control its lifetime, or it may be non-owning and represent a view of storage owned by some other object.

One can also imagine engines that manage resources related to execution. This is an area of ongoing work and not yet addressed in this proposal.

fs_matrix_engine<T, R, C>

Class template `fs_matrix_engine<T, R, C>` implements a fixed-size, fixed-capacity engine for matrices having `R` rows and `C` columns. In addition to the basic engine interface, it provides member functions for mutable element indexing, swapping engine contents, swapping columns, and swapping rows.

```
template<class T, size_t R, size_t C>
class fs_matrix_engine
{
  public:
    //- Types
    //
    using engine_category = initable_matrix_engine_tag;
    using element_type    = T;
    using value_type      = remove_cv_t<T>;
    using pointer         = element_type*;
    using const_pointer   = element_type const*;
    using reference       = element_type&;
    using const_reference = element_type const&;
    using difference_type = ptrdiff_t;
    using size_type       = size_t;
    using size_tuple      = tuple<size_type, size_type>;
    using span_type       = mdspan<element_type, R, C>;
    using const_span_type = mdspan<element_type const, R, C>;

    //- Construct/copy/destroy
    //
    ~fs_matrix_engine() noexcept = default;

    constexpr fs_matrix_engine();
    constexpr fs_matrix_engine(fs_matrix_engine&&) noexcept = 
default;
    constexpr fs_matrix_engine(fs_matrix_engine const&) = default;

    template<class T2, size_t R2, size_t C2>        @(_see note_)@
    constexpr fs_matrix_engine(fs_matrix_engine<T2, R2, C2> const& 
rhs);
    template<class ET2>                             @(_see note_)@
    constexpr fs_matrix_engine(ET2 const& rhs);
    template<class T2>                              @(_see note_)@
    constexpr 
fs_matrix_engine(initializer_list<initializer_list<T2>> rhs);

    constexpr fs_matrix_engine&     operator =(fs_matrix_engine&&) 
noexcept
         = default;
    constexpr fs_matrix_engine&     operator =(fs_matrix_engine 
const&)
```

```
        = default;

    template<class T2, size_t R2, size_t C2>        @(_see note_)@
    constexpr fs_matrix_engine&     operator =
        (fs_matrix_engine<T2, R2, C2> const& rhs);
    template<class ET2>                             @(_see note_)@
    constexpr fs_matrix_engine&     operator =(ET2 const& rhs);
    template<class T2>                              @(_see note_)@
    constexpr fs_matrix_engine&     operator =
        (initializer_list<initializer_list<T2>> rhs);

    //- Capacity
    //
    constexpr size_type     columns() const noexcept;
    constexpr size_type     rows() const noexcept;
    constexpr size_tuple    size() const noexcept;

    constexpr size_type     column_capacity() const noexcept;
    constexpr size_type     row_capacity() const noexcept;
    constexpr size_tuple    capacity() const noexcept;

    //- Element access
    //
    constexpr reference         operator ()(size_type i, size_type
j);
    constexpr const_reference   operator ()(size_type i, size_type
j) const;

    //- Data access
    //
    constexpr span_type         span() noexcept;
    constexpr const_span_type   span() const noexcept;

    //- Modifiers
    //
    constexpr void      swap(fs_matrix_engine& rhs) noexcept;
    constexpr void      swap_columns(size_type j1, size_type j2)
noexcept;
    constexpr void      swap_rows(size_type i1, size_type i2)
noexcept;
};
```

dr_matrix_engine<T, AT>

Class template `dr_matrix_engine<T, AT>` implements an engine for matrices whose sizes and capacities can be changed at runtime. In addition to the basic engine interface, it provides member functions for mutable element indexing, swapping engine contents, swapping columns, swapping rows, and resizing.

```
template<class T, class AT>
class dr_matrix_engine
{
  public:
    //- Types
    //
    using engine_category = resizable_matrix_engine_tag;
    using element_type    = T;
    using value_type      = remove_cv_t<T>;
    using allocator_type  = AT;
    using pointer         = typename
allocator_traits<AT>::pointer;
    using const_pointer   = typename
allocator_traits<AT>::const_pointer;
    using reference       = element_type&;
    using const_reference = element_type const&;
    using difference_type = ptrdiff_t;
    using size_type       = size_t;
    using size_tuple      = tuple<size_type, size_type>;
    using span_type       = basic_mdspan<T,
@_implementation-define_@>;
    using const_span_type = basic_mdspan<T const,
@_implementation-define_@>;

    //- Construct/copy/destroy
    //
    ~dr_matrix_engine() noexcept;

    dr_matrix_engine();
    dr_matrix_engine(dr_matrix_engine&& rhs) noexcept;
    dr_matrix_engine(dr_matrix_engine const& rhs);
    dr_matrix_engine(size_type rows, size_type cols);
    dr_matrix_engine(size_type rows, size_type cols, size_type
rowcap, size_type colcap);

    template<class ET2>                    @(_see note_)@
    dr_matrix_engine(ET2 const& rhs);
    template<class T2>                     @(_see note_)@
    dr_matrix_engine(initializer_list<initializer_list<T2>> rhs);

    dr_matrix_engine&   operator =(dr_matrix_engine&&) noexcept;
    dr_matrix_engine&   operator =(dr_matrix_engine const&);

    template<class ET2>                    @(_see note_)@
    dr_matrix_engine&   operator =(ET2 const& rhs);
    template<class T2>                     @(_see note_)@
    dr_matrix_engine&   operator =
        (initializer_list<initializer_list<T2>> rhs);
```

```
    //- Capacity
    //
    size_type   columns() const noexcept;
    size_type   rows() const noexcept;
    size_tuple  size() const noexcept;

    size_type   column_capacity() const noexcept;
    size_type   row_capacity() const noexcept;
    size_tuple  capacity() const noexcept;

    void        reserve(size_type rowcap, size_type colcap);
    void        resize(size_type rows, size_type cols);
    void        resize(size_type rows, size_type cols, size_type
rowcap, size_type colcap);

    //- Element access
    //
    reference       operator ()(size_type i, size_type j);
    const_reference operator ()(size_type i, size_type j) const;

    //- Data access
    //
    span_type       span() noexcept;
    const_span_type span() const noexcept;

    //- Modifiers
    //
    void    swap(dr_matrix_engine& other) noexcept;
    void    swap_columns(size_type c1, size_type c2) noexcept;
    void    swap_rows(size_type r1, size_type r2) noexcept;
};
```

matrix_view_engine<ET, MCT, VFT>

Class template matrix_view_engine<ET, MCT, VFT> implements a non-owning engine that implements at least the readable matrix engine interface, and possibly the writable matrix engine interface, depending on the underlying engine type ET and the tag type MCT. Its purpose is to provide a transpose view of a matrix (when template parameter VFT is transpose_view_tag), or some contiguous range of rows and columns from some matrix object (when template parameter MFT is submatrix_view_tag).

```
template<class ET, class MCT>
class matrix_view_engine<ET, MCT, submatrix_view_tag>
{
    static_assert(is_matrix_engine_v<ET>);
    static_assert(is_matrix_engine_tag<MCT>);
```

```
  public:
    //- Types
    //
    using engine_category = MCT;
    using element_type    = typename ET::element_type;
    using value_type      = typename ET::value_type;
    using pointer         = @_implementation-defined_@;
    using const_pointer   = typename ET::const_pointer;
    using reference       = @_implementation-defined_@;
    using const_reference = typename ET::const_reference;
    using difference_type = typename ET::difference_type;
    using size_type       = typename ET::size_type;
    using size_tuple      = typename ET::size_tuple;
    using span_type       = @_implementation-defined_@;     (@_see
note_@)
    using const_span_type = @_implementation-defined_@;     (@_see
note_@)

    //- Construct/copy/destroy
    //
    ~matrix_view_engine() noexcept = default;

    constexpr matrix_view_engine();
    constexpr matrix_view_engine(matrix_view_engine&&) noexcept
        = default;
    constexpr matrix_view_engine(matrix_view_engine const&)
noexcept
        = default;

    constexpr matrix_view_engine&     operator =
(matrix_view_engine&&)
        noexcept = default;
    constexpr matrix_view_engine&     operator
=(matrix_view_engine const&)
        noexcept = default;

    template<class ET2                      (@_see note_@)
    constexpr matrix_view_engine&     operator =(ET2 const& rhs);
    template<class U>                       (@_see note_@)
    constexpr matrix_view_engine&     operator =
        (initializer_list<initializer_list<U>> list);

    //- Capacity
    //
    constexpr size_type     columns() const noexcept;
    constexpr size_type     rows() const noexcept;
    constexpr size_tuple    size() const noexcept;
```

```
    constexpr size_type     column_capacity() const noexcept;
    constexpr size_type     row_capacity() const noexcept;
    constexpr size_tuple    capacity() const noexcept;

    //- Element access
    //
    constexpr reference     operator ()(size_type i, size_type j)
const;

    //- Data access
    //
    constexpr span_type     span() const noexcept;      (@_see
note_@)

    //- Modifiers
    //
    constexpr void          swap(matrix_view_engine& rhs);
};
```

If the matrix engine category tag `MCT` is `readable_vector_engine_tag`, then nested type aliases `pointer` and `reference` are equivalent to `const_pointer` and `const_reference`, respectively.

If the matrix engine category tag `MCT` is `writable_vector_engine_tag`, then nested type aliases `pointer` and `reference` are equivalent to `typename ET::pointer` and `typename ET::reference`, respectively.

Math object types

`matrix<ET, OT>`

Class template `matrix<ET, OT>` represents a matrix, with element type and resource management implemented by the engine type `ET`, and arithmetic operations specified by the operation traits type `OT`. If the underlying engine type provides dynamic resizing, then this class will as well.

```
template<class ET, class OT>
class matrix
{
  public:
    //- Types
    //
    using engine_type           = ET;
    using element_type          = typename
engine_type::element_type;
    using value_type            = typename engine_type::value_type;
```

```
    using reference           = typename engine_type::reference;
    using const_reference     = typename
engine_type::const_reference;
    using difference_type     = typename
engine_type::difference_type;
    using size_type           = typename engine_type::size_type;
    using size_tuple          = typename engine_type::size_tuple;

    using column_type         =
        matrix<column_engine<engine_type, @_see note_@>, OT>;
    using const_column_type   =
        matrix<column_engine<engine_type,
readable_matrix_engine_tag>, OT>;

    using row_type            =
        matrix<row_engine<engine_type, @_see note_@>, OT>;
    using const_row_type      =
        matrix<row_engine<engine_type,
readable_matrix_engine_tag>, OT>;

    using submatrix_type      =
        matrix<submatrix_engine<engine_type, @_see note_@>, OT>;
    using const_submatrix_type =
        matrix<submatrix_engine<engine_type,
readable_matrix_engine_tag>, OT>;

    using transpose_type      =
        matrix<transpose_engine<engine_type, @_see note_@>, OT>;
    using const_transpose_type =
        matrix<transpose_engine<engine_type,
readable_matrix_engine_tag>, OT>;

    using hermitian_type      =
        conditional_t<@_see note_@, matrix, transpose_type>;
    using const_hermitian_type =
        conditional_t<@_see note_@, matrix, const_transpose_type>;

    using span_type           = @_implementation-defined_@;
(@_see note_@)
    using const_span_type     = @_implementation-defined_@;
(@_see note_@)

    //- Construct/copy/destroy
    //
    ~matrix() noexcept = default;

    constexpr matrix() = default;
    constexpr matrix(matrix&&) noexcept = default;
```

```
    constexpr matrix(matrix const&) = default;

    template<class ET2, class OT2>
    constexpr matrix(matrix<ET2, OT2> const& src);
    template<class U>                                       (@_see
note_@)
    constexpr matrix(initializer_list<initializer_list<U>> rhs);

    constexpr matrix(size_tuple size);                      (@_see
note_@)
    constexpr matrix(size_type rows, size_type cols);       (@_see
note_@)
    constexpr matrix(size_tuple size, size_tuple cap);      (@_see
note_@)
    constexpr matrix(size_type rows, size_type cols,
                     size_type rowcap, size_type colcap);   (@_see
note_@)

    constexpr matrix&   operator =(matrix&&) noexcept = default;
    constexpr matrix&   operator =(matrix const&) = default;
    template<class ET2, class OT2>
    constexpr matrix&   operator =(matrix<ET2, OT2> const& rhs);
    template<class U>                                       (@_see
note_@)
    constexpr matrix&   operator
=(initializer_list<initializer_list<U>> rhs);

    //- Capacity
    //
    static constexpr bool   is_resizable() noexcept;
    constexpr size_type     columns() const noexcept;
    constexpr size_type     rows() const noexcept;
    constexpr size_tuple    size() const noexcept;

    constexpr size_type     column_capacity() const noexcept;
    constexpr size_type     row_capacity() const noexcept;
    constexpr size_tuple    capacity() const noexcept;

    constexpr void      reserve(size_tuple cap);
(@_see note_@)
    constexpr void      reserve(size_type rowcap, size_type
colcap);
        (@_see note_@)

    constexpr void      resize(size_tuple size);
(@_see note_@)
    constexpr void      resize(size_type rows, size_type cols);
(@_see note_@)
```

```
    constexpr void      resize(size_tuple size, size_tuple
cap);(@_see note_@)
    constexpr void      resize(size_type rows, size_type cols,
                               size_type rowcap, size_type
colcap);
        (@_see note_@)

    //- Element access
    //
    constexpr reference             operator ()(size_type i,
size_type j);
    constexpr const_reference       operator ()(size_type i,
size_type j) const;

    constexpr column_type           column(size_type j) noexcept;
    constexpr const_column_type     column(size_type j) const
noexcept;
    constexpr row_type              row(size_type i) noexcept;
    constexpr const_row_type        row(size_type i) const
noexcept;
    constexpr submatrix_type        submatrix(size_type ri,
size_type rn,
                                              size_type ci,
size_type cn)
                                         noexcept;
    constexpr const_submatrix_type  submatrix(size_type ri,
size_type rn,
                                              size_type ci,
size_type cn)
                                         const noexcept;
    constexpr transpose_type        t() noexcept;
    constexpr const_transpose_type  t() const noexcept;
    constexpr hermitian_type        h();
    constexpr const_hermitian_type  h() const;

    //- Data access
    //
    constexpr engine_type&          engine() noexcept;
    constexpr engine_type const&    engine() const noexcept;

    constexpr span_type             span() noexcept;        (@_see
note_@)
    constexpr const_span_type       span() const noexcept;  (@_see
note_@)

    //- Modifiers
    //
```

```
    constexpr void      swap(matrix& rhs) noexcept;
    constexpr void      swap_columns(size_type i, size_type j)
noexcept;
          (@_see note_@)
    constexpr void      swap_rows(size_type i, size_type j)
noexcept;
          (@_see note_@)
};
```

For the nested type aliases `column_type` and `row_type`: if `typename ET::engine_category` is equal to `readable_matrix_engine_tag`, then the matrix engine tag type to be used as a template argument to `column_engine` and `row_engine`, respectively, is `readable_vector_engine_tag`. Otherwise, it is `writable_vector_engine_tag`.

For nested type aliases `transpose_type` and `submatrix_type`: if `typename ET::engine_category` is equal to `readable_matrix_engine_tag`, then the matrix engine tag type to be used as a template argument to `transpose_engine` and `submatrix_engine`, respectively, is `readable_matrix_engine_tag`. Otherwise, it is `writable_vector_engine_tag`.

Operation traits

matrix_operation_traits

Class `matrix_operation_traits` is a traits-style template parameter to `matrix`. Its purpose is to associate sets of element promotion traits, engine promotion traits, and arithmetic traits with a *MathObj* so that those traits may be conveyed into an arithmetic operator.

```
struct matrix_operation_traits
{
    //- Default element promotion traits.
    //
    template<class T1>
    using element_negation_traits =
matrix_negation_element_traits<T1>;

    template<class T1, class T2>
    using element_addition_traits =
matrix_addition_element_traits<T1, T2>;

    template<class T1, class T2>
    using element_subtraction_traits =
        matrix_subtraction_element_traits<T1, T2>;
```

```
    template<class T1, class T2>
    using element_multiplication_traits =
        matrix_multiplication_element_traits<T1, T2>;

    //- Default engine promotion traits.
    //
    template<class OTR, class ET1>
    using engine_negation_traits =
matrix_negation_engine_traits<OTR, ET1>;

    template<class OTR, class ET1, class ET2>
    using engine_addition_traits =
        matrix_addition_engine_traits<OTR, ET1, ET2>;

    template<class OTR, class ET1, class ET2>
    using engine_subtraction_traits =
        matrix_subtraction_engine_traits<OTR, ET1, ET2>;

    template<class OTR, class ET1, class ET2>
    using engine_multiplication_traits =
        matrix_multiplication_engine_traits<OTR, ET1, ET2>;

    //- Default arithmetic operation traits.
    //
    template<class OP1, class OTR>
    using negation_traits = matrix_negation_traits<OP1, OTR>;

    template<class OTR, class OP1, class OP2>
    using addition_traits = matrix_addition_traits<OTR, OP1, OP2>;

    template<class OTR, class OP1, class OP2>
    using subtraction_traits = matrix_subtraction_traits<OTR, OP1,
OP2>;

    template<class OTR, class OP1, class OP2>
    using multiplication_traits =
matrix_multiplication_traits<OTR, OP1, OP2>;
};
```

This traits type is a customization point. Users may override the default functionality it provides by creating a custom operation traits class in their own namespace, and defining only those members necessary to implement the desired custom behavior.

`matrix_operation_traits_selector<OT1, OT2>`

Class template `matrix_operation_traits_selector<OT1, OT2>` is used by the binary arithmetic operators to select the operation traits type to be used in performing an arithmetic operation. The selection is based on the operation traits types of the two operands.

```
//- Primary template and expected specializations.
//
template<class T1, class T2>
struct matrix_operation_traits_selector;

template<class T1>
struct matrix_operation_traits_selector<T1, T1>
{
    using traits_type = T1;
};

template<class T1>
struct matrix_operation_traits_selector<T1,
matrix_operation_traits>
{
    using traits_type = T1;
};

template<class T1>
struct matrix_operation_traits_selector<matrix_operation_traits,
T1>
{
    using traits_type = T1;
};

template<>
struct matrix_operation_traits_selector<matrix_operation_traits,
matrix_operation_traits>
{
    using traits_type = matrix_operation_traits;
};

//- Convenience alias.
//
template<class T1, class T2>
using matrix_operation_traits_selector_t =
        typename matrix_operation_traits_selector<T1,
T2>::traits_type;
```

Element promotion traits

Element promotion traits are used by the library to determine the resulting element type of an arithmetical expression having one or two *MathObj* operands.

matrix_negation_element_traits<T1>

Class template `matrix_negation_element_traits<T1>` implements the default traits type for determining the element type of the *MathObj* instance resulting from negating a given *MathObj* instance.

Alias template `matrix_negation_element_t<OT, T1, T2>` is used by the library to return the nested type `OT::element_negation_traits<T1>`.

```
template<class T1>
struct matrix_negation_element_traits
{
    using element_type = decltype(-declval<T1>());
};

template<class OT, class T1>
using matrix_negation_element_t = ...;          //-
Implementation-defined
```

matrix_addition_element_traits<T1, T2>

Class template `matrix_addition_element_traits<T1, T2>` implements the default traits type for determining the element type of a *MathObj* instance resulting from the addition of two other *MathObj* instances.

Alias template `matrix_addition_element_t<OT, T1, T2>` is used by the library to obtain the nested type `OT::element_addition_traits<T1, T2>`.

```
template<class T1, class T2>
struct matrix_addition_element_traits
{
    using element_type = decltype(declval<T1>() + declval<T2>());
};

template<class OT, class T1, class T2>
using matrix_addition_element_t = ...;          //-
Implementation-defined
```

matrix_subtraction_element_traits<T1, T2>

Class template `matrix_subtraction_element_traits<T1, T2>` implements the default traits type for determining the element type of a *MathObj* instance resulting from the subtraction of two other *MathObj* instances.

Alias template `matrix_subtraction_element_t<OT, T1, T2>` is used by the library to obtain the nested type `OT::element_subtraction_traits<T1, T2>`.

```
template<class T1, class T2>
struct matrix_subtraction_element_traits
{
    using element_type = decltype(declval<T1>() - declval<T2>());
};

template<class OT, class T1, class T2>
using matrix_subtraction_element_t = ...;       //-
Implementation-defined
```

`matrix_multiplication_element_traits<T1, T2>`

Class template `matrix_multiplication_element_traits<T1, T2>` implements the default traits type for determining the element type of a *MathObj* instance resulting from the multiplication of two other *MathObj* instances.

Alias template `matrix_multiplication_element_t<OT, T1, T2>` is used by the library to obtain the nested type `OT::element_multiplication_traits<T1, T2>`.

```
template<class T1, class T2>
struct matrix_multiplication_element_traits
{
    using element_type = decltype(declval<T1>() * declval<T2>());
};

template<class OT, class T1, class T2>
using matrix_multiplication_element_t = ...;    //-
Implementation-defined
```

Engine promotion traits

Engine promotion traits are used by the arithmetic traits to determine the resulting engine types in an arithmetical expression.

`matrix_negation_engine_traits<OT, ET1>`

Class template `matrix_negation_engine_traits<OT, ET1>` implements a traits type that determines the resulting engine type when negating a *MathObj*.

Alias template `matrix_negation_engine_t<OT, ET1>` is used by the library to obtain the nested type `OT::engine_negation_traits<ET1>`.

```
template<class OT, class ET1>
struct matrix_negation_engine_traits
{
    using element_type =
        matrix_negation_element_t<OT, typename ET1::element_type>;
```

```
    using engine_type  = ...;                        //-
Implementation-defined
};

template<class OT, class ET1>
using matrix_negation_engine_t = ...;            //-
Implementation-defined
```

matrix_addition_engine_traits<OT, ET1, ET2>

Class template `matrix_addition_engine_traits<OT, ET1, ET2>` implements a traits type that determines the resulting engine type when adding two compatible *MathObj*s.

Alias template `matrix_addition_engine_t<OT, ET1, ET2>` is used by the library to obtain the nested type `OT::element_addition_traits<ET1, ET2>`.

```
template<class OT, class ET1, class ET2>
struct matrix_addition_engine_traits
{
    using element_type =
        matrix_addition_element_t<OT,
                                  typename ET1::element_type,
                                  typename ET2::element_type>;

    using engine_type  = ...;                        //-
Implementation-defined
};

template<class OT, class ET1, class ET2>
using matrix_addition_engine_t = detail::engine_add_type_t<OT,
ET1, ET2>;
```

matrix_subtraction_engine_traits<OT, ET1, ET2>

Class template `matrix_subtraction_engine_traits<OT, ET1, ET2>` implements a traits type that determines the resulting engine type when subtracting two compatible *MathObj*s.

Alias template `matrix_subtraction_engine_t<OT, ET1, ET2>` is used by the library to obtain the nested type `OT::element_subtraction_traits<ET1, ET2>`.

```
template<class OT, class ET1, class ET2>
struct matrix_subtraction_engine_traits
{
    using element_type =
        matrix_subtraction_element_t<OT,
                                     typename ET1::element_type,
                                     typename ET2::element_type>;
```

```
    using engine_type  = ...;                       //-
Implementation-defined
};

template<class OT, class ET1, class ET2>
using matrix_subtraction_engine_t = ...;        //-
Implementation-defined
```

matrix_multiplication_engine_traits<OT, ET1, ET2>

Class template `matrix_multiplication_engine_traits<OT, ET1, ET2>` implements a traits type that determines the resulting engine type when multiplying two compatible *MathObj*s.

Alias template `matrix_multiplication_engine_t<OT, ET1, ET2>` is used by the library to obtain the nested type `OT::element_multiplication_traits<ET1, ET2>`.

```
template<class OT, class ET1, class ET2>
struct matrix_multiplication_engine_traits
{
    using element_type =
        matrix_multiplication_element_t<OT,
                                        typename
ET1::element_type,
                                        typename
ET2::element_type>;

    using engine_type  = ...;                       //-
Implementation-defined.
};

template<class OT, class ET1, class ET2>
using matrix_multiplication_engine_t = ...;     //-
Implementation-defined
```

Arithmetic traits

This section defines a set of arithmetic traits types for negation, addition, subtraction, and multiplication. The purpose of each of these traits types is threefold:

1. to determine the element type of the resulting *MathObj*;

2. to determine the engine type of the resulting *MathObj*; and

3. to carry out the arithmetical operation and return its result.

The idea here is that arithmetic operators (described below) simply forward
to the appropriate traits type, which does the heavy lifting.

matrix_negation_traits<OT, OP1>

Class template `matrix_negation_traits<OT, OP1>` is an arithmetic traits type that performs the negation of a *MathObj* and returns the result in another *MathObj* having an implementation-defined engine type. There are two partial specializations to support the two overloaded negation operators described below.

Alias template `matrix_negation_traits_t<OT, OP1>` is used by the library to obtain the nested type `OT::negation_traits<OP1>`.

```
template<class OT, class ET1, class OT1>
struct matrix_negation_traits<OT, matrix<ET1, OT1>>
{
    using engine_type = matrix_negation_engine_t<OT, ET1>;
    using op_traits   = OT;
    using result_type = matrix<engine_type, op_traits>;

    static result_type  negate(matrix<ET1, OT1> const& v1);
};

template<class OT, class OP1>
using matrix_negation_traits_t = ...;       //-
Implementation-defined
```

matrix_addition_traits<OT, OP1, OP2>

Class template `matrix_addition_traits<OT, OP1, OP2>` is an arithmetic traits type that performs the addition of two compatible *MathObj*s and returns the result in a *MathObj* having an implementation-defined engine type. There are two partial specializations to support the two overloaded addition operators described below.

Alias template `matrix_addition_traits_t<OT, OP1, OP2>` is used by the library to obtain the nested type `OT::addition_traits<OP1, OP2>`.

```
template<class OT, class ET1, class OT1, class ET2, class OT2>
struct matrix_addition_traits<OT, matrix<ET1, OT1>, matrix<ET2,
OT2>>
{
    using engine_type = matrix_addition_engine_t<OT, ET1, ET2>;
    using op_traits   = OT;
    using result_type = matrix<engine_type, op_traits>;

    static result_type  add
        (matrix<ET1, OT1> const& m1, matrix<ET2, OT2> const& m2);
};
```

```
template<class OT, class OP1, class OP2>
using matrix_addition_traits_t = ...;        //-
Implementation-defined
```

matrix_subtraction_traits<OT, OP1, OP2>

Class template `matrix_subtraction_traits<OT, OP1, OP2>` is an arithmetic traits type that performs the subtraction of two compatible *MathObj*s and returns the result in a *MathObj* having an implementation-defined engine type. There are two partial specializations to support the two overloaded subtraction operators described below.

Alias template `matrix_subtraction_traits_t<OT, OP1, OP2>` is used by the library to obtain the nested type `OT::subtraction_traits<OP1, OP2>`.

```
template<class OT, class ET1, class OT1, class ET2, class OT2>
struct matrix_subtraction_traits<OT, matrix<ET1, OT1>, matrix<ET2,
OT2>>
{
    using engine_type = matrix_subtraction_engine_t<OT, ET1, ET2>;
    using op_traits   = OT;
    using result_type = matrix<engine_type, op_traits>;

    static result_type  subtract
        (matrix<ET1, OT1> const& m1, matrix<ET2, OT2> const& m2);
};

template<class OT, class OP1, class OP2>
using matrix_subtraction_traits_t = ...;        //-
Implementation-defined
```

matrix_multiplication_traits<OT, OP1, OP2>

Class template `matrix_multiplication_traits<OT, OP1, OP2>` is an arithmetic traits type that performs the multiplication of two compatible *MathObj*s and returns the result in a *MathObj* having an implementation-defined engine type. There are eight partial specializations to support the eight binary multiplication operators described below.

Alias template `matrix_multiplication_traits_t<OT, OP1, OP2>` is used by the library to obtain the nested type `OT::multiplication_traits<OP1, OP2>`.

```
//- matrix*scalar
//
template<class OT, class ET1, class OT1, class T2>
struct matrix_multiplication_traits<OT, matrix<ET1, OT1>, T2>
{
    using scalar_type = detail::element_tag<T2>;
```

```
    using engine_type = matrix_multiplication_engine_t<OT, ET1,
scalar_type>;
    using op_traits   = OT;
    using result_type = matrix<engine_type, op_traits>;

    static result_type  multiply(matrix<ET1, OT1> const& m1, T2
const& s2);
};

//- scalar*matrix
//
template<class OT, class T1, class ET2, class OT2>
struct matrix_multiplication_traits<OT, T1, matrix<ET2, OT2>>
{
    using scalar_type = detail::element_tag<T1>;
    using engine_type = matrix_multiplication_engine_t<OT,
scalar_type, ET2>;
    using op_traits   = OT;
    using result_type = matrix<engine_type, op_traits>;

    static result_type  multiply(T1 const& s1, matrix<ET2, OT2>
const& m2);
};

template<class OT, class ET1, class OT1, class ET2, class OT2>
struct matrix_multiplication_traits<OT, matrix<ET1, OT1>,
matrix<ET2, OT2>>
{
    using engine_type = matrix_multiplication_engine_t<OT, ET1,
ET2>;
    using op_traits   = OT;
    using result_type = matrix<engine_type, op_traits>;

    static result_type  multiply
        (matrix<ET1, OT1> const& m1, matrix<ET2, OT2> const& m2);
};

template<class OT, class OP1, class OP2>
using matrix_multiplication_traits_t = ...;     //-
Implementation-defined
```

Arithmetic operators

The arithmetic operators provide syntax that mimics common mathematical notation, with computation executed by an arithmetic traits type specified by one of the operands' operation traits template parameters.

Readers will note that the return types of the overloaded operators described below are left unspecified. This is a deliberate choice so that implementers have the freedom to choose whatever default technique for evaluating expressions they desire; for example, by returning temporary objects, or by using expression templates, or perhaps by some hybrid technique.

```
//- Negation
//
template<class ET1, class OT1>
inline auto
operator -(matrix<ET1, OT1> const& m1)
{
    using op1_type   = matrix<ET1, OT1>;
    using op_traits  = OT1;
    using neg_traits = matrix_negation_traits_t<op_traits,
op1_type>;

    return neg_traits::negate(m1);
}

//- Addition
//
template<class ET1, class OT1, class ET2, class OT2>
inline auto
operator +(matrix<ET1, OT1> const& m1, matrix<ET2, OT2> const& m2)
{
    using op_traits  = matrix_operation_traits_selector_t<OT1,
OT2>;
    using op1_type   = matrix<ET1, OT1>;
    using op2_type   = matrix<ET2, OT2>;
    using add_traits =
        matrix_addition_traits_t<op_traits, op1_type, op2_type>;

    return add_traits::add(m1, m2);
}

//- Subtraction
//
template<class ET1, class OT1, class ET2, class OT2>
inline auto
operator -(matrix<ET1, OT1> const& m1, matrix<ET2, OT2> const& m2)
{
    using op_traits  = matrix_operation_traits_selector_t<OT1,
OT2>;
    using op1_type   = matrix<ET1, OT1>;
    using op2_type   = matrix<ET2, OT2>;
    using sub_traits =
        matrix_subtraction_traits_t<op_traits, op1_type,
op2_type>;
```

```
    return sub_traits::subtract(m1, m2);
}

//- Multiplication
//- matrix*scalar and scalar*matrix
//
template<class ET1, class OT1, class S2>
inline auto
operator *(matrix<ET1, OT1> const& m1, S2 const& s2)
{
    static_assert(is_matrix_element_v<S2>);

    using op_traits  = OT1;
    using op1_type   = matrix<ET1, OT1>;
    using op2_type   = S2;
    using mul_traits =
        matrix_multiplication_traits_t<op_traits, op1_type,
op2_type>;

    return mul_traits::multiply(m1, s2);
}

template<class S1, class ET2, class OT2>
inline auto
operator *(S1 const& s1, matrix<ET2, OT2> const& m2)
{
    static_assert(is_matrix_element_v<S1>);

    using op_traits  = OT2;
    using op1_type   = S1;
    using op2_type   = matrix<ET2, OT2>;
    using mul_traits =
        matrix_multiplication_traits_t<op_traits, op1_type,
op2_type>;

    return mul_traits::multiply(s1, m2);
}

//- matrix*matrix
//
template<class ET1, class OT1, class ET2, class OT2>
inline auto
operator *(matrix<ET1, OT1> const& m1, matrix<ET2, OT2> const& m2)
{
    using op_traits  = matrix_operation_traits_selector_t<OT1,
OT2>;
    using op1_type   = matrix<ET1, OT1>;
```

```
    using op2_type    = matrix<ET2, OT2>;
    using mul_traits =
        matrix_multiplication_traits_t<op_traits, op1_type,
op2_type>;

    return mul_traits::multiply(m1, m2);
}
```

Customization

The library provides for several forms of customization: custom element types, custom element promotion, custom engines, and custom arithmetical operations. The following sections show examples of each.

Integrating a new element type

Suppose that you have created a new type that models a real number in some way and you wish for that type to be used as a matrix element:

```
class new_num
{
  public:
    new_num();
    new_num(new_num&&) = default;
    new_num(new_num const&) = default;
    template<class U>   new_num(U other);

    new_num&    operator =(new_num&&) = default;
    new_num&    operator =(new_num const&) = default;
    template<class U>   new_num&    operator =(U rhs);

    new_num     operator -() const;
    new_num     operator +() const;

    new_num&    operator +=(new_num rhs);
    new_num&    operator -=(new_num rhs);
    new_num&    operator *=(new_num rhs);
    new_num&    operator /=(new_num rhs);

    template<class U>   new_num&    operator +=(U rhs);
    template<class U>   new_num&    operator -=(U rhs);
    template<class U>   new_num&    operator *=(U rhs);
    template<class U>   new_num&    operator /=(U rhs);

    ...
};
```

```
inline bool operator ==(NewNum lhs, NewNum rhs);
template<class U> inline bool operator ==(NewNum lhs, U rhs);
template<class U> inline bool operator ==(U lhs, NewNum rhs);
...
//- other comparison operators...
...
//- other arithmetic operators...
...
                  inline new_num operator *(new_num lhs, new_num
rhs);
template<class U> inline new_num operator *(new_num lhs, U rhs);
template<class U> inline new_num operator *(U lhs, new_num rhs);
```

Assuming that this type works as intended, and that all arithmetic interactions with other types are handled the set of operator overloads that you provide, then all that is required for the library to accept `new_num` as an element type is to create a specialization of `number_traits`:

```
template<>
struct std::math::number_traits<new_num>
{
    using is_field    = true_type;
    using is_nc_ring  = true_type;
    using is_ring     = true_type;
};
```

By stating that `new_num` models at least a non-commutative ring, and by ensuring that its arithmetic operators functions as promised, the library's traits types will allow compiliation to succeed.

Custom element promotion

Suppose that you want the result of adding two `float` elements to be `double`. Then you would create the following custom types in your namespace:

```
//- Base template for custom element promotion
//
template<class T1, class T2>
struct element_add_traits_tst;

//- Promote any float/float addition to double.
//
template<>
struct element_add_traits_tst<float, float>
{
    using element_type = double;
};
```

```
//- Custom operation traits.
//
struct test_add_op_traits_tst
{
     template<class T1, class T2>
     using element_addition_traits = element_add_traits_tst<T1,
T2>;
};
```

The new operation traits could be used like this:

```
matrix<fs_matrix_engine<float, 2, 3>, add_op_traits_tst>
m1;
matrix<dr_matrix_engine<float, allocator<float>>,
add_op_traits_tst>        m2(2, 3);
matrix<dr_matrix_engine<float, allocator<float>>,
matrix_operation_traits>  m3(2, 3);

//- mr1 --> matrix<fs_matrix_engine<double, 2, 3>,
add_op_traits_tst>
//
auto mr1 = m1 + m1;

//- mr2 --> matrix<dr_matrix_engine<double, allocator<double>>,
add_op_traits_tst>
//
auto mr2 = m1 + m2;

//- mr3 --> matrix<dr_matrix_engine<double, allocator<double>>,
add_op_traits_tst>
//
auto mr3 = m1 + m3;
```

Note that this example assumes that an addition operation involving a fixed-size matrix and a dynamically-resizable matrix, or two dynamically-resizable matrices results in a dynamically-resizable matrix.

Integrating a new engine type

Suppose that you want to add a custom fixed-size matrix engine that is somehow different from `fs_matrix_engine`; perhaps it is instrumented in some way for debugging, or uses fixed-size storage that is external to the engine object. It might look like this:

```
template<class T, int32_t R, int32_t C>
class fs_matrix_engine_tst
{
  public:
    using engine_category = std::math::mutable_matrix_engine_tag;
```

```
    using element_type    = T;
    using value_type      = T;
    using reference       = T&;
    using pointer         = T*;
    using const_reference = T const&;
    using const_pointer   = T const*;
    using difference_type = std::ptrdiff_t;
    using index_type      = std::int_fast32_t;
    using size_type       = std::int_fast32_t;
    using size_tuple      = std::tuple<size_type, size_type>;

    using is_fixed_size   = std::true_type;
    using is_resizable    = std::false_type;

    using is_column_major = std::false_type;
    using is_dense        = std::true_type;
    using is_rectangular  = std::true_type;
    using is_row_major    = std::true_type;

    using column_view_type    =
        std::math::column_engine<fs_matrix_engine_tst>;
    using row_view_type       =
std::math::row_engine<fs_matrix_engine_tst>;
    using transpose_view_type =
        std::math::transpose_engine<fs_matrix_engine_tst>;

  public:
    constexpr fs_matrix_engine_tst();
    constexpr fs_matrix_engine_tst(fs_matrix_engine_tst&&) =
default;
    constexpr fs_matrix_engine_tst(fs_matrix_engine_tst const&) =
default;

    constexpr fs_matrix_engine_tst&     operator =
        (fs_matrix_engine_tst&&) = default;
    constexpr fs_matrix_engine_tst&     operator =
        (fs_matrix_engine_tst const&) = default;

    constexpr const_reference   operator ()(index_type i,
index_type j) const;

    constexpr index_type    columns() const noexcept;
    constexpr index_type    rows() const noexcept;
    constexpr size_tuple    size() const noexcept;

    constexpr size_type     column_capacity() const noexcept;
    constexpr size_type     row_capacity() const noexcept;
    constexpr size_tuple    capacity() const noexcept;
```

```
    constexpr reference     operator ()(index_type i, index_type
j);

    constexpr void      assign(fs_matrix_engine_tst const& rhs);
    template<class ET2>
    constexpr void      assign(ET2 const& rhs);

    constexpr void      swap(fs_matrix_engine_tst& rhs) noexcept;
    constexpr void      swap_columns(index_type j1, index_type
j2);
    constexpr void      swap_rows(index_type i1, index_type i2);

  private:
    ...         //- Implementation stuff
};
```

For each arithmetic operation in which you expect the new engine type to be involved, you will need to provide a specialization of the engine promotion traits for that operation. For example, let's assume that you're only interested in addition operations involving two operands having the new engine type, or where one operand has the standard fixed-size engine and the other has the new engine. Then your engine promotion traits might look like this:

```
//- Goal: Create a new fixed-size engine type and use it in
arithmetical expressions.
//
template<class OT, class ET1, class ET2>
struct engine_add_traits_tst;

template<class OT, class T1, int32_t R1, int32_t C1, class T2,
int32_t R2, int32_t C2>
struct engine_add_traits_tst<OT,
                             fs_matrix_engine_tst<T1, R1, C1>,
                             fs_matrix_engine_tst<T2, R2, C2>>
{
    using element_type = std::math::matrix_addition_element_t<OT,
T1, T2>;
    using engine_type  = fs_matrix_engine_tst<element_type, R1,
C1>;
};

template<class OT,
         class T1, int32_t R1, int32_t C1,
         class T2, int32_t R2, int32_t C2>
struct engine_add_traits_tst<OT,
                             fs_matrix_engine_tst<T1, R1, C1>,
```

```
                             std::math::fs_matrix_engine<T2, R2,
C2>>
{
    using element_type = std::math::matrix_addition_element_t<OT,
T1, T2>;
    using engine_type  = fs_matrix_engine_tst<element_type, R1,
C1>;
};

template<class OT,
         class T1, int32_t R1, int32_t C1,
         class T2, int32_t R2, int32_t C2>
struct engine_add_traits_tst<OT,
                             std::math::fs_matrix_engine<T1, R1,
C1>,
                             fs_matrix_engine_tst<T2, R2, C2>>
{
    using element_type = std::math::matrix_addition_element_t<OT,
T1, T2>;
    using engine_type   = fs_matrix_engine_tst<element_type, R1,
C1>;
};

//- This is a custom operation traits type!
//
struct add_op_traits_tst
{
    template<class T1, class T2>
    using element_addition_traits = element_add_traits_tst<T1,
T2>;

    template<class T1, class T2>
    using engine_addition_traits = engine_add_traits_tst<T1, T2>;
};
```

As we can see, these custom promotion traits dictate the resulting engine type for these particular cases. Resulting usage might look like this:

```
matrix<fs_matrix_engine<float, 2, 3>, matrix_operation_traits>
m1;
matrix<fs_matrix_engine_tst<float, 2, 3>, add_op_traits_tst>
m2;
matrix<dr_matrix_engine<float, allocator<float>>,
matrix_operation_traits>  m3(2, 3);

//- mr1 --> matrix<fs_matrix_engine<float, 2, 3>,
matrix_operation_traits>
//
```

```
auto    mr1 = m1 + m1;

//- mr2 --> matrix<fs_matrix_engine_tst<double, 2, 3>,
add_op_traits_tst>
//
auto    mr2 = m2 + m2;

//- mr3 --> matrix<fs_matrix_engine_tst<double, 2, 3>,
add_op_traits_tst>
//
auto    mr3 = m1 + m2;

//- mr4 --> matrix<dr_matrix_engine<double, allocator<double>>,
add_op_traits_tst>
//
auto    mr4 = m1 + m3;
```

Note that this example also assumes that an addition operation involving a fixed-size matrix and a dynamically-resizable matrix, or two dynamically-resizable matrices results in a dynamically-resizable matrix.

Customizing an arithmetic operation

Suppose that you want to specialize the addition function for the addition of two matrices that employ the custom engine above and whose sizes happen to be 3x4.

```
//- Goal: Call a specialized addition function for addition of
fixed-size matrix objects
// using the fixed-size test engine and having size 3x4.
//
template<class OT, class OP1, class OP2>
struct addition_traits_tst;

template<class OT>
struct addition_traits_tst<OT,
                           matrix<fs_matrix_engine_tst<double, 3,
4>, OT>,
                           matrix<fs_matrix_engine_tst<double, 3,
4>, OT>>
{
    using op_traits = OT;
    using engine_type = fs_matrix_engine_tst<double, 3, 4>;
    using result_type = matrix<engine_type, op_traits>;

    static result_type  add
        (matrix<fs_matrix_engine_tst<double, 3, 4>, OT> const& m1,
         matrix<fs_matrix_engine_tst<double, 3, 4>, OT> const&
m2);
```

```
};

//- This is a custom operation traits type!
//
struct test_add_op_traits_tst
{
    template<class T1, class T2>
    using element_addition_traits = element_add_traits_tst<T1,
T2>;

    template<class OT, class ET1, class ET2>
    using engine_addition_traits = engine_add_traits_tst<OT, ET1,
ET2>;

    template<class OT, class OP1, class OP2>
    using addition_traits = addition_traits_tst<OT, OP1, OP2>;
};
```

Actual usage might look like this:

```
matrix<fs_matrix_engine_tst<float, 3, 4>, add_op_traits_tst>
m1;
matrix<fs_matrix_engine_tst<double, 3, 4>, add_op_traits_tst>
m2;

//- mr1 --> matrix<fs_matrix_engine_tst<double, 3, 4>,
add_op_traits_tst>
//
auto    mr1 = m1 + m1;    //- Calls matrix_addition_traits::add()

//- mr2 --> matrix<fs_matrix_engine_tst<double, 3, 4>,
add_op_traits_tst>
//
auto    mr2 = m1 + m2;    //- Calls matrix_addition_traits::add()

//- mr3 --> matrix<fs_matrix_engine_tst<double, 3, 4>,
add_op_traits_tst>
//
auto    mr3 = m2 + m2;    //- Calls
matrix_addition_traits_tst::add()
```