A proposal to add a matrix type to the C++ standard library

Document #:	P1385R8
Date:	2023-05-15
Project:	Programming Language C++
Audience:	SG6, LEWG

Reply-to:	Guy Davidson <guy.cpp.wg21@gmail.com> Bob Steagall <bob.steagall.cpp@gmail.com>

Contents

Revision History

- R8 *Update for pre-Varna 2023 mailing*
  - Refactored the `matrix_storage_engine` interface to be similar to that of `mdspan`.
  - Added support for fixed-size, dynamically-allocated matrices.
  - Added support for writable view engines, as appropriate.
  - Updated the convenience aliases for declaring `matrix` objects.
  - Replaced the idea of operation traits with that of *custom operation traits*.
- 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.
- R6 *Update for post-Prague mailing*
  - Added `initable_*_tag` to specify types for which construction and assignment from an `initializer_list` are acceptable.
  - 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()`.
  - 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.
- 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.
- R4 *Update to R3 for post-Belfast mailing.*
  - Include feedback from reviews in Belfast.
- R3 *Update for Belfast 2019 meeting.*
  - Remove more erroneous references to `row_vector` and `column_vector`.
- D3 *Last-minute update for Cologne 2019 meeting*
  - Remove erroneous references to `row_vector` and `column_vector` in the R2 text.
- R2 *Update for pre-Cologne 2019 mailing*
  - 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.
- R1 *Update for post-Kona mailing.*
  - Includes feedback from LEWG(I) and joint SG14/SG19 session.
- D1
  - Update for presentation at Kona 2019 that includes operation traits.
- R0
  - Intial version for pre-Kona 2019 mailing.

Abstract

This document proposes a matrix type and supporting facilities to enable basic linear algebra functionality for the standard C++ library. The facilities proposed herein are pure addtions, requiring no changes to existing implementations.

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.

1 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.

1.1 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 meetings and 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 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.

2 Definitions

When discussing linear algebra and related topics for a proposal like this one, 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.

2.1 Mathematical terms

In order to facilitate subsequent discussion, we first offer the following *informal* set of definitions for the relevant 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 thevector 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 given row and 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 $a_{m,n}$ 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.

2.2 Overloaded mathematical 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 Mathematical terms, and `matrix` (in monospaced font) to mean the C++ class template that implements the mathematical object.

Vector

Likewise, many C++ programmers often 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, as described above in Mathematical terms above. Further, we also mean *vector* generically to have both of the meanings set out in that section.

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.

2.3 Terms pertaining to C++ types

The following are terms used in this proposal that describe various aspects of how the mathematical concepts described above 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 by 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. An **engine** is an implementation type that manages the resources associated with a `matrix` instance. This includes, at a minimum, the storage-related aspects of, and access to, the elements of that `matrix`. It could also include execution-related aspects, such as an execution context. In this proposal, an engine object is a private member of a `matrix`. Other than as a template parameter, engines are not part of the `matrix` public interface.
4. The adjective **dense** refers to a `matrix` representation where storage is allocated for every element.
5. The adjective **sparse** refers to a `matrix` representation where storage is allocated only for non-zero elements;
6. **Storage** is used by this proposal as a synonym for memory.
7. **Traits** refers to a stateless class template that provides some set of services, normalizing those services over its set of template parameters.
8. **Row size** and **column size** refer to the number of rows and columns, respectively, that a `matrix` represents, which must be less than or equal to its row and column capacities, defined below.
9. **Row capacity** and **column capacity** refer to the maximum number of rows and columns, respectively, that a `matrix` can possibly represent.
10. **Fixed-size** (FS) refers to an engine type whose row and column sizes are fixed at instantiation time and constant thereafter.
11. **Fixed-capacity** (FC) refers to an engine type whose row and column capacities are fixed at instantiation time and constant thereafter.
12. **Dynamically re-sizable** (DR) refers to an engine type whose row and column sizes and capacities may be changed at run time.

3 Scope

We contend that the best approach for standardizing a set of linear algebra components for C++26 will be one that is layered, iterative, and incremental. This paper is quite deliberately a "basic matrix arithmetic-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.

3.1 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 matrix addition, subtraction, negation, and multiplication;

— Arithmetic operations for scalar multiplication and division 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.

3.2 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.

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.

4 Design considerations

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.

4.1 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 `matrix`, not allocated from a heap, and which has a lifetime greater than that of the `matrix`;

— Elements reside in a fixed-size buffer that is a member of the `matrix` itself;

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

4.2 Addressing model

It is also possible that the memory used by a `matrix` 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.

4.3 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 `matrix` 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.

4.4 Capacity and resizability

As with `basic_string` and `vector`, it is occasionally useful for a `matrix` 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 `matrix` object'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, `matrix` objects are small and always of the same size. In this case, the size and capacity are equal, and there is no need for a `matrix` to maintain or manage excess capacity.

4.5 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.

4.6 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.

4.7 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 `matrix` may allocate memory, and therefore an implementation cannot assume that element types have trivial constructors or destructors.

4.8 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 principal should apply to expressions involving `matrix` objects where more than one element type is present. Arithmetic operations involving `matrix` operands 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*.

4.9 Mixed-element-layout expressions

This library defaults to a dense, row-major layout of elements in `matrix` engines. However, there may be times when it makes sense to specify a column-major layout for the instantiation of a `matrix` and/or the result of an operation.

For example, a developer may wish to multiply a matrix representing an affine transformation by a display list of vectors. The display list is actually a `matrix`, where each column of the matrix is a vector from the list. To maximize locality of reference, it might make sense for the transform matrix to be row-major, while the matrix representing the display list and the result are both column-major.

We call the process of specifying the element layout resulting from an arithmetic operation *layout promotion*.

4.10 Mixed-engine expressions

In analogy with element type, `matrix` 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?

Expressions 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.

4.11 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 representations, 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*.

4.12 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.

5 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 those hardy souls attempting to build implementations from this specification.

5.1 Overview

The public interface is divided into several broad categories:

1. **Tags**, used as template arguments to help specify behavior;
2. **Engines**, which are implementation types that manage the resources associated with a `matrix` instance, including memory ownership and lifetime, as well as element access;
3. **Custom operation traits**, which act as a "container" for optional, user-defined traits types that customize the ways in which element promotion, layout promotion, engine promotion, and arithmetic operations are to be carried out; and,
4. **The `matrix` class template**, which provides a unified interface intended to model the corresponding mathematical abstraction;
5. **Custom operation selection traits** provide the means by which an arithmetic operator selects the operation traits that will perform the arithmetic when one or both operands have custom operation traits. The traits class template `matrix_custom_operation_traits_selector` is a library customization point that allows the user to specify how the selection should be done;
6. **Operators**, which provide the desired syntax and carry out the intended arithmetic upon `matrix` (and scalar) operands.
7. **Alias templates**, which provide a number of convenient services:
   - Declaring commonly-used `matrix` specializations;
   - Selecting custom operation traits at compile-time;
   - Extracting the element type, layout type, engine type, and arithmetic traits type from a custom operation traits type.

5.1.1 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 `C`, `C1`, and `C2` represent the number of columns in a matrix engine.
- Parameter names `R`, `R1`, and `R2` represent the number of rows in a matrix engine.
- Parameter names `AT`, `AT1`, and `AT2` represent allocator types.
- Parameter names `LT`, `LT1`, and `LT2` represent matrix layout types.
- Parameter name `MVT` represents a view engine's functionality (e.g., transpose, row, column, submatrix, etc.).
- Parameter names `COT`, `COT1`, and `COT2` represent custom operation traits types.
- Parameter names `MT`, `MT1`, and `MT2` represent matrix types.
- Parameter names `S`, `S1`, and `S2` represent scalar operand types.

5.2 Header <matrix> synopsis

```
#include <array>
#include <complex>
#include <concepts>
#include <deque>
#include <initializer_list>
#include <mdspan>
#include <type_traits>
#include <vector>

namespace std::math {

//- Tags that describe element layout.
//
struct matrix_layout
{
    struct row_major;
    struct column_major;
    struct arbitrary;
};

//- Tags that describe the semantics of view engines.
//
struct matrix_view
{
    struct const_negation;
    struct const_conjugate;
    struct const_hermitian;
    struct identity;
    struct const_identity;
    struct transpose;
    struct const_transpose;
    struct column;
    struct const_column;
    struct row;
    struct const_row;
    struct submatrix;
    struct const_submatrix;
};

//- Engine types.
//
template<class T>
struct matrix_scalar_engine;

template<class T, size_t R, size_t C, class AT, class LT>
class matrix_storage_engine;

template<class ET, class MVT>
class matrix_view_engine;

//- The primary math object type, matrix, and (in)equality operators.
//
template<class ET, class COT = void>
class matrix;
```

```
template<class ET1, class COT1, class ET2, class COT2>
inline constexpr bool
operator ==(matrix<ET1, COT1> const& m1, matrix<ET2, COT2> const& m2);

template<class ET1, class COT1, class ET2, class COT2>
inline constexpr bool
operator !=(matrix<ET1, COT1> const& m1, matrix<ET2, COT2> const& m2);

//- A traits type that chooses between two custom operation traits types; this class is
//  a library customization point.
//
template<class OT1, class OT2>
struct matrix_custom_operation_traits_selector;

//- Arithmetic operators
//
template<class ET1, class COT1, class ET2, class COT2>
inline constexpr (see note)
operator +(matrix<ET1, COT1> const& m1, matrix<ET2, COT2> const& m2);

template<class ET1, class COT1, class ET2, class COT2>
inline constexpr (see note)
operator -(matrix<ET1, COT1> const& m1, matrix<ET2, COT2> const& m2);

template<class ET1, class COT1, class S2>
inline constexpr (see note)
operator *(matrix<ET1, COT1> const& m1, S2 const& s2);

template<class S1, class ET2, class COT2>
inline constexpr (see note)
operator *(S1 const& s1, matrix<ET2, COT2> const& m2);

template<class ET1, class COT1, class ET2, class COT2>
inline constexpr (see note)
operator *(matrix<ET1, COT1> const& m1, matrix<ET2, COT2> const& m2);

template<class ET1, class COT1, class S2>
inline constexpr (see note)
operator /(matrix<ET1, COT1> const& m1, S2 const& s2)

//- Alias templates for convenience when declaring matrix objects.
//
template<class T, size_t R, size_t C, class COT = void>
using fixed_size_matrix =
    matrix<matrix_storage_engine<T, R, C, void, matrix_layout::row_major>, COT>;

template<class T, size_t R, class COT = void>
using fixed_size_column_vector =
    matrix<matrix_storage_engine<T, R, 1, void, matrix_layout::column_major>, COT>;

template<class T, size_t C, class COT = void>
using fixed_size_row_vector =
    matrix<matrix_storage_engine<T, 1, C, void, matrix_layout::row_major>, COT>;
```

```
template<class T, size_t R, size_t C, class COT = void>
using general_matrix =
    matrix<matrix_storage_engine<T, R, C, allocator<T>, matrix_layout::row_major>, COT>;

template<class T, size_t R, class COT = void>
using general_column_vector =
    matrix<matrix_storage_engine<T, R, 1, allocator<T>, matrix_layout::column_major>, COT>;

template<class T, size_t C, class COT = void>
using general_row_vector =
    matrix<matrix_storage_engine<T, 1, C, allocator<T>, matrix_layout::row_major>, COT>;


template<class T, class COT = void>
using dynamic_matrix =
    matrix<matrix_storage_engine<T, dynamic_extent, dynamic_extent,
                                 allocator<T>, matrix_layout::row_major>, COT>;

template<class T, class COT = void>
using dynamic_column_vector =
    matrix<matrix_storage_engine<T, dynamic_extent, 1,
                                 allocator<T>, matrix_layout::column_major>, COT>;

template<class T, class COT = void>
using dynamic_row_vector =
    matrix<matrix_storage_engine<T, 1, dynamic_extent,
                                 allocator<T>, matrix_layout::row_major>, COT>;

//- Traits extractors, to help with creating custom operation traits.
//
template<class COT1, class COT2>
using select_custom_matrix_operation_traits_t = (see note);


template<typename COT, typename MT1, typename MT2>
using matrix_addition_element_traits_t = (see note);

template<typename COT, typename MT1, typename MT2>
using matrix_addition_layout_traits_t = (see note);

template<typename COT, typename MT1, typename MT2>
using matrix_addition_engine_traits_t = (see note);

template<typename COT, typename MT1, typename MT2>
using matrix_addition_arithmetic_traits_t = (see note);


template<typename COT, typename MT1, typename MT2>
using matrix_subtraction_element_traits_t = (see note);

template<typename COT, typename MT1, typename MT2>
using matrix_subtraction_layout_traits_t = (see note);

template<typename COT, typename MT1, typename MT2>
using matrix_subtraction_engine_traits_t = (see note);
```

```
template<typename COT, typename MT1, typename MT2>
using matrix_subtraction_arithmetic_traits_t = (see note);


template<typename COT, typename MT1, typename MT2>
using matrix_multiplication_element_traits_t = (see note);

template<typename COT, typename MT1, typename MT2>
using matrix_multiplication_layout_traits_t = (see note);

template<typename COT, typename MT1, typename MT2>
using matrix_multiplication_engine_traits_t = (see note);

template<typename COT, typename MT1, typename MT2>
using matrix_multiplication_arithmetic_traits_t = (see note);


template<typename COT, typename MT1, typename MT2>
using matrix_division_element_traits_t = (see note);

template<typename COT, typename MT1, typename MT2>
using matrix_division_layout_traits_t = (see note);

template<typename COT, typename MT1, typename MT2>
using matrix_division_engine_traits_t = (see note);

template<typename COT, typename MT1, typename MT2>
using matrix_division_arithmetic_traits_t = (see note);

}   //- Namespace std::math
```