Linear Algebra for the Standard C++ Library

(A Proposal)

Bob Steagall

C++Now 2019

Linear Algebra for the Standard C++ Library

(A Proposal)

Sponsored by: The American East Const Association of America®

Bob Steagall

C++Now 2019

Overview

- Some background
- High-level goals
- Some important definitions
- Design aspects
- Scope and requirements
- Interface design
- How it works
- Customizing behavior
- Ongoing / future work

Overview

- Discussion of P1385
  - *A proposal to add linear algebra support to the C++ standard library*
  - http://wg21.link/P1385
- Co-author Guy Davidson
- SG14 Linear Algebra Study Group

Some Background

What is Linear Algebra?

- **Linear algebra**
  - The branch of mathematics concerning linear equations and linear functions and their representations through matrices and vector spaces.

- Central to many areas of mathematics
  - For example, modern treatments of geometry

- Useful in science and engineering
  - Allows modeling many phenomena and computing efficiently with such models
  - Used in many domains (computer graphics, machine learning, finance, analytics, medical imaging, signal processing, nuclear simulations, etc.)

User Requirements

- Everyone
  - Ease-of-use
  - Expressiveness
  - Performance

- Super-users
  - Customization
  - Support for non-traditional computing environments

High-Level Goals – General

- Provide a set of linear algebra vocabulary types

- Provide a public interface that is
  - Intuitive
  - Teachable
  - Customizable

  -- and --

  - Mimics traditional mathematical notation

- Exhibit competitive out-of-box performance

High-Level Goals – Customization

- Provide a set of building blocks for
  - Managing memory (source, ownership, lifetime, layout, access)
  - Managing other resources (e.g., execution context)
  - Possibly representing other interesting math types (e.g., tensors, quaternions)

- Provide *straightforward* tools for customization
  - Enable users to optimize performance for their specific problem/hardware

- Provide a *reasonable* level of granularity for customization
  - Users have to implement a minimum set of types/functions

Some Important Definitions

Mathematical Terms

- **Linear algebra** is primarily the study of vector spaces.

- **Vector space**
  - 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

- **Dimension** of a vector space
  - The number of coordinates required to specify any point within the space

Mathematical Terms

- **Matrix**
  - A rectangular arrangement of numbers, symbols, or expressions organized in rows and columns
  - A matrix having *R* rows and *C* columns is said to have size *R* x *C*
  - Matrices provide a useful way of representing linear transformations from one vector space to another

- **Element**
  - An individual member of the rectangular arrangement comprising the matrix
  - Rows are traditionally indexed from 1 to *R*, and columns from 1 to *C*
  - In matrix A, element a_{11} appears in the upper left-hand corner, while element a_{RC} appears in the lower right-hand corner.

Mathematical Terms

- **Row vector**
  - A matrix containing a single row – a matrix of size *1* x *C*
  - The rows of a matrix are sometimes called row vectors

- **Column vector**
  - A matrix containing a single column – a matrix of size *R* x *1*
  - The columns of a matrix are sometimes called column vectors

- **Rank** (of a matrix)
  - The dimension of the vector space spanned by its rows/columns
  - Also equal to the maximum number of linearly-independent rows/columns

Mathematical Terms

- **Element transforms**
  - Non-arithmetic operations that modify the relative positions of elements in a matrix, such as transpose, column exchange, and row exchange
- **Element arithmetic**
  - Arithmetical operations that read or modify the values of individual elements independently of other elements
- **Matrix arithmetic**
  - Assignment, addition, subtraction, negation, and multiplication operations defined for matrices and vectors as wholes

Mathematical Terms

- **Decompositions**
  - Complex sequences of arithmetic operations, element arithmetic, and element transforms performed upon a matrix to determine important mathematical properties of that matrix

- **Eigen-decompositions**
  - Sequences of operations performed upon a symmetric matrix in order to compute the eigenvalues and eigenvectors of that matrix

Terms Regarding C++ Types

- **Math object**
  - Generically, one of the C++ types `matrix` or `vector` described here
- **Storage**
  - A synonym for memory
- **Dense**
  - A math object representation with storage allocated for every element
- **Sparse**
  - A math object representation with storage allocated only for non-zero elements

Terms Regarding C++ Types

- **Engines** are implementation types that manage the **resources** associated with a math object
  - Element storage ownership and lifetime
  - Access to individual elements
  - Resizing/reserving, if appropriate
  - Execution context
- In this interface design, an engine object is a private member of a containing math object
- Other than as a template parameter, engines are not part of a math object's public interface

Terms Regarding C++ Types

- **Traits**
  - A (usually) stateless class or class template whose members provide an interface normalized over some set of types or template parameters
  - Often appear as parameters in class/function templates
- **Row capacity / column capacity**
  - The maximum number of rows/columns that a math object could *possibly* have
- **Row size / column size**
  - The number of rows/columns that a math object *actually* has
  - Must be less than or equal to corresponding row/column capacities

Terms Regarding C++ Types

- **Fixed-size**
  - An engine type whose row/column sizes are fixed *and* known at compile time
- **Fixed-capacity**
  - An engine type whose row/column capacities are fixed *and* known at compile time
- **Dynamically re-sizable**
  - An engine type whose row/column sizes/capacities are set at run time

Disambiguating Math and C++ Terms

- *Matrix* is frequently (ab)used by C/C++ programmers to mean a general purpose array with some arbitrary number of indices

- We use *matrix* to mean the mathematical object
  - And `matrix` to mean the C++ class template that models a *matrix*

- We use the term *array* to mean a
  - Single- or multi-dimensional array in the C++ sense
  - No invariants pertaining to higher-level or mathematical meaning

Disambiguating Math and C++ Terms

- *Vector* is frequently (ab)used by C/C++ programmers to mean a dynamically-resizable one-dimensional array

- We use *vector* to mean the mathematical object
  - And `vector` to mean the C++ class template that models a *vector*

- We use the term *linear array* to mean a
  - Single-dimensional array in the C++ sense, having
  - No invariants pertaining to a higher-level or mathematical meaning

Disambiguating Math and C++ Terms

- In programming, *dimension* refers to the number of indices required to access an element of an array

- In linear algebra, a vector space *V* is *n-dimensional* if there exists *n* linearly independent vectors that span *V*

- We use *dimension* both ways
  - A vector describing a point in an electric field is a one-dimensional data structure implemented as a three-dimensional vector
  - A rotation matrix used by a game engine is two-dimensional data structure composed of three-dimensional row and column vectors

Disambiguating Math and C++ Terms

- The *rank* of a matrix is the dimension of the vector space spanned by its rows/columns
  - In tensor analysis, rank is often used as a synonym for a tensor's *order*
- The C++ standard uses the term *rank* as a synonym for *dimension*
  - [*meta.unary.prop.query*]: rank is the number of dimensions of T, if T names an array, otherwise it is zero
- We avoid using *rank*

Design Aspects

Design Aspects – Memory

- Location
  - In an external buffer allocated from the global heap or custom allocator
  - In an internal buffer that is a member of the math object itself
  - Collectively in a set of buffers distributed across multiple processes/machines

- Addressing model
  - Memory might be addressed via *fancy pointer* (e.g., shared / distributed /elsewhere)

- Ownership
  - A math object might own and manage its memory
  - A math object might use a const/mutable view to memory managed by another object

Design Aspects – Memory

- Capacity and resizability
  - In some problem domains, it is useful for a math object to have excess storage capacity, so that resizes do not require reallocations
  - In other problem domains (like graphics) math objects are small and never resize
- Element layout
  - In C/C++, the default is row-major dense rectangular
  - In Fortran, the default is column-major dense rectangular
  - Upper/lower triangular; banded
  - Sparse

Design Aspects – Elements

- Element types
  - C++ provides only a small set of arithmetic types
  - Sometimes other types are desirable
    - Fixed-point, arbitrary precision floating point, elastic precision, complex, etc.
    - Individual elements may allocate memory – can't assume trivial element types

- Expressions with mixed element types
  - In general, when multiple primitive types are present in a arithmetic expression, the resulting type is the "largest" of all the types
  - Information should be preserved
  - The process of determining the resulting element type is **element promotion**

Design Aspects – Arithmetic

- Expressions with mixed engine types
  - Consider fixed-size matrix multiplied by a dynamically-resizable matrix
  - The resulting engine should be at least as "general" as the "most general" of all the engine types participating in the expression
  - Determining the resulting engine type is called **engine promotion**
- Arithmetic expressions
  - Users may want to optimize specific operations
    - SIMD-based matrix-matrix and matrix-vector multiplication
  - Two operands may be associated with different customizations
  - Determining the customization to employ is **operation traits promotion**

Scope and Requirements

Scope

- The best approach for standardizing a set of linear algebra components for C++23 will be one that is **layered**, **iterative**, and **incremental**

- This proposal (P1385) is deliberately one for basic linear algebra only
  - Describes the minimum set of components and arithmetic operations necessary to provide a reasonable, basic level of functionality

- Higher-level functionality can be built upon the interfaces described the proposal
  - **We encourage succession papers to explore this possibility!**

Required Functionality – Abstract

- Provide the minimal set of types and functions required to perform basic matrix arithmetic in finite dimensional spaces
  - Facilities for determining if a type T can be a matrix element
  - Types that that implement engines
  - Types that model matrices, row vectors, and column vectors
  - Support for element transforms
  - Support for element arithmetic
  - Support for (possibly mixed-type) matrix arithmetic

- Make it easy-to-use and extensible

Required Functionality - Concrete

- Model the mathematical ideas
  - Traits types to validate element type
  - Class templates that implement engines
  - Class templates that represent matrices and vectors
  - Arithmetic operators for addition, subtraction, and negation of matrices and vectors
  - Arithmetic operators for scalar multiplication of matrices and vectors
  - Arithmetic operators for non-scalar multiplication of matrices and vectors

Required Functionality - Concrete

- Make it flexible
  - Use traits types to ensure mixed-type arithmetic expressions are supported

- Make it extensible, with well-defined, (relatively) easy-to-use
  - Facilities for integrating new element types
  - Facilities for integrating custom engines
  - Facilities for integrating custom implementations of arithmetic operations

- Minimize customization points in/under namespace `std`
  - This design requires only <u>two</u>

Considered but Excluded

- Tensors
  - Every rank-2 tensor can be represented by a square matrix, but not every square matrix is a tensor
  - The class invariants and public interface are very different from those of a matrix
  - Matrices are not Liskov-substitutable for tensors

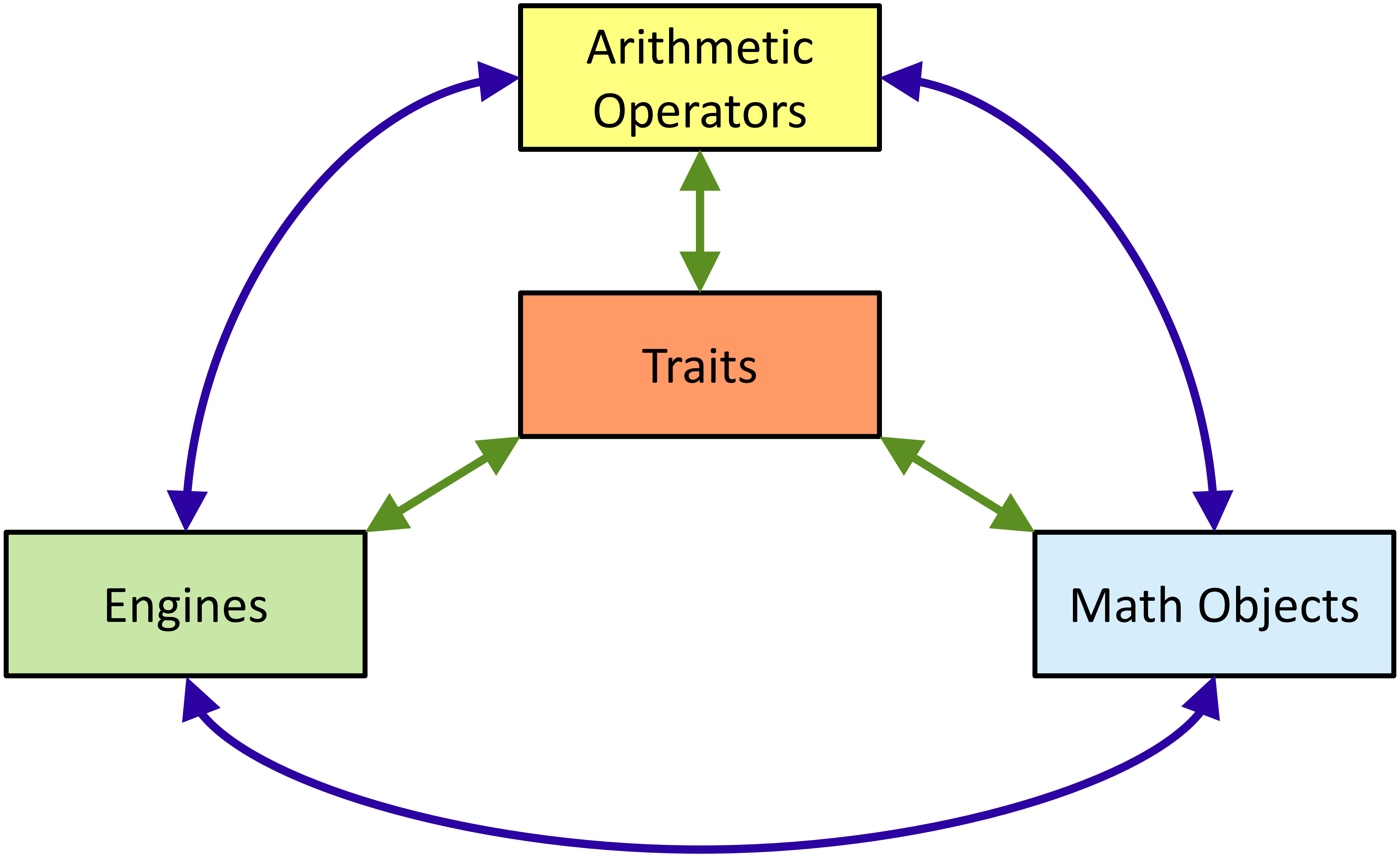
- Quaternions
  - Quaternions model math concepts very different from vectors
  - Class invariants and public interface are also very different from that of vectors
  - Vectors are not Liskov-substitutable for quaternions

Interface and Components

Interface Overview – Type Categories

- **Engines** are implementation types that manage resources
  - Memory management, ownership, and lifetime control
  - Element access and update
- **Math objects** (`vector` and `matrix`) model mathematical abstractions
  - Use engines to manage elements
  - Present a consolidated interface to the arithmetic operators
- **Operators** provide the desired syntax
  - Addition, subtract, multiplication, and negation
- **Traits** types support the engines, math object, and operators
  - Perform type <u>and</u> value computations

Interface Overview – Type Categories



Interface Overview – Traits Support

- **Numeric traits**
  - Specify and test the properties of numeric types
  - Customization point permitting partial/full specialization by the user

- **Element promotion traits**
  - Determine the resulting element type of an *element* arithmetic operation

- **Engine promotion traits**
  - Determine the resulting engine type of a *matrix* arithmetic operation

Interface Overview – Traits Support

- **Arithmetic traits**
  - Determine the resulting **type** and **value** of an arithmetic operation

- **Operation traits**
  - A “container” for element promotion, engine promotion, and arithmetic traits
  - Template parameter to `matrix` and `vector`

- **Operation selector traits**
  - Used by operators to select the result’s operation traits type
  - Customization point, permitting partial/full specialization by the user

Interface Overview – Traits Support

- Implementation-specific private traits types (many)

- Employ the usual host of fundamental metaprogramming tools
  - Traits types
  - Partial specialization
  - Variable templates
  - Type detection idiom

Code Overview

Type Declarations – Numeric/Element Traits

```
namespace std::math {
...

//- A traits type that supplies important information about a numerical type.  Note that
//  this traits class is a customization point.
//
template<class T>   struct number_traits;

//- Predicate traits for matrix element type inquiries.
//
template<class T>   struct is_complex;
template<class T>   struct is_field;
template<class T>   struct is_ring;
template<class T>   struct is_nc_ring;

template<class T>   struct is_matrix_element;

...
}
```

Type Declarations – Engine Tags

```
namespace std::math {
...

//- Some type tags for specifying how engines should behave.
//
using scalar_engine_tag              = integral_constant<int, 0>;

using const_vector_engine_tag        = integral_constant<int, 1>;
using mutable_vector_engine_tag      = integral_constant<int, 2>;
using resizable_vector_engine_tag    = integral_constant<int, 3>;

using const_matrix_engine_tag        = integral_constant<int, 4>;
using mutable_matrix_engine_tag      = integral_constant<int, 5>;
using resizable_matrix_engine_tag    = integral_constant<int, 6>;




...
}
```

Type Declarations – Engines

```
namespace std::math {
...

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

//- Owning engines with fixed-size internal storage.
//
template<class T, int32_t N>             class fs_vector_engine;
template<class T, int32_t R, int32_t C> class fs_matrix_engine;

//- Non-owning view-style engine.
//
template<class ET>      class matrix_column_view;
template<class ET>      class matrix_row_view;
template<class ET>      class matrix_transpose_view;


...
}
```

Type Declarations – Operation Traits and Math Objects

```
namespace std::math {
...

//- The default element promotion, engine promotion, and arithmetic operation traits for
//  the four basic arithmetic operations, rolled up under a consolidated traits type.
//
struct matrix_operation_traits;


//- Primary mathematical object types.
//
template<class ET, class OT=matrix_operation_traits>  class vector;
template<class ET, class OT=matrix_operation_traits>  class matrix;


...
}
```

Type Declarations – Element and Engine Promotion Traits

```
namespace std::math {
...

//- 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;


...
}
```

Type Declarations – Arithmetic and Operation Traits

```
namespace std::math {
...

//- Math object arithmetic traits.
//
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 (e.g., outer_product()).
//  Note that this traits class is a customization point.
//
template<class T1, class T2>    struct matrix_operation_traits_selector;


...
}
```

Operators – Addition, Subtraction, Negation

```
namespace std::math {
...

//- Addition
//
template<class ET1, class OT1, class ET2, class OT2>
inline auto  operator +(vector<ET1, OT1> const& v1, vector<ET2, OT2> const& v2);

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



...
}
```

Operators – Addition, Subtraction, Negation

```
namespace std::math {
...

//- Subtraction
//
template<class ET1, class OT1, class ET2, class OT2>
inline auto  operator -(vector<ET1, OT1> const& v1, vector<ET2, OT2> const& v2);

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



...
}
```

Operators – Addition, Subtraction, Negation

```
namespace std::math {
...

//- Negation
//
template<class ET1, class OT1>
inline auto  operator -(vector<ET1, OT1> const& v1);

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



...
}
```

Operators – Scalar/Math Type Multiplication

```
namespace std::math {
...

//- Vector*Scalar
//
template<class ET1, class OT1, class S2>
inline auto  operator *(vector<ET1, OT1> const& v1, S2 const& s2);

template<class S1, class ET2, class OT2>
inline auto  operator *(S1 const& s1, vector<ET2, OT2> const& v2);

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

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

...
}
```

Operators – Math Type / Math Type Multiplication

```
namespace std::math {
...

//- Vector*Matrix
//
template<class ET1, class OT1, class ET2, class OT2>
inline auto  operator *(vector<ET1, OT1> const& v1, matrix<ET2, OT2> const& m2);

//- Matrix*Vector
//
template<class ET1, class OT1, class ET2, class OT2>
inline auto  operator *(matrix<ET1, OT1> const& m1, vector<ET2, OT2> const& v2);


...
}
```

Operators – Math Type / Math Type Multiplication

```
namespace std::math {
...

//- Vector*Vector
//
template<class ET1, class OT1, class ET2, class OT2>
inline auto  operator *(vector<ET1, OT1> const& v1, vector<ET2, OT2> const& v2);

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



...
}
```

Convenience Aliases

```
namespace std::math {

//- Aliases for vector and matrix objects based on dynamically-resizable engines.
//
template<class T, class A = allocator<T>>
using dyn_vector = vector<dr_vector_engine<T, A>, matrix_operation_traits>;

template<class T, class A = allocator<T>>
using dyn_matrix = matrix<dr_matrix_engine<T, A>, matrix_operation_traits>;


//- Aliases for vector and matrix objects based on fixed-size engines.
//
template<class T, int32_t N>
using fs_vector = vector<fs_vector_engine<T, N>, matrix_operation_traits>;

template<class T, int32_t R, int32_t C>
using fs_matrix = matrix<fs_matrix_engine<T, R, C>, matrix_operation_traits>;

...
}
```

Numeric Traits

Interface Overview – Numeric Traits

- **Numeric traits**
  - Specify and test the properties of numeric types
  - Customization point intended to be partially/fully specializable by the user

```
//- A traits type that supplies important information about a numerical type.  Note that
//  this traits class is a customization point.
//
template<class T>   struct number_traits;

//- Predicate traits for matrix element type inquiries.
//
template<class T>   struct is_complex;
template<class T>   struct is_field;
template<class T>   struct is_ring;
template<class T>   struct is_nc_ring;

template<class T>   struct is_matrix_element;
```

Numeric Traits – number_traits<T>

```
namespace detail {

//- Support types for a possible implementation.
//
struct builtin_number_traits
{
    using is_field     = true_type;
    using is_ring      = true_type;
    using is_nc_ring   = true_type;
};

struct non_number_traits
{
    using is_field     = false_type;
    using is_ring      = false_type;
    using is_nc_ring   = false_type;
};

...
}
```

Numeric Traits – number_traits<T>

```
namespace detail {
...

//- More support for a possible implementation.
//
template<class T>
using scalar_number_traits_helper_t =
            conditional_t<is_arithmetic_v<T>, builtin_number_traits, non_number_traits>;
}


//- A traits type that supplies important information about a numerical type.
//
template<class T>
struct number_traits : public detail::scalar_number_traits_helper_t<T> {};

template<class T>
struct number_traits<complex<T>> : public number_traits<T> {};
```

Number, Traits – `is_field<T>`, `is_matrix_element<T>`

```cpp
//- Predicate trait types for fields
//
template<class T>
struct is_field : public bool_constant<number_traits<T>::is_field::value> {};

template<class T> constexpr bool  is_field_v = is_field<T>::value;


//- Predicate traits type for matrix elements.
//
template<class T>
struct is_matrix_element : public bool_constant<is_arithmetic_v<T> || is_field_v<T>> {};

template<class T>
constexpr bool  is_matrix_element_v = is_matrix_element<T>::value;
```

Engines

Interface Overview – Engines

- **Engines** are implementation types that manage resources
  - Memory management, ownership, and lifetime control
  - Element access

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

//- Owning engines with fixed-size internal storage.
//
template<class T, int32_t N>             class fs_vector_engine;
template<class T, int32_t R, int32_t C> class fs_matrix_engine;

//- Non-owning view-style engine.
//
template<class ET>      class matrix_column_view;
template<class ET>      class matrix_row_view;
template<class ET>      class matrix_transpose_view;
```

DR Matrix Engine – Nested Type Aliases

```
template<class T, class AT>
class dr_matrix_engine
{
    static_assert(is_matrix_element_v<T>);

  public:
    using engine_category = resizable_matrix_engine_tag;
    using element_type    = T;
    using value_type      = T;
    using allocator_type  = AT;
    using reference       = T&;
    using pointer         = typename allocator_traits<AT>::pointer;
    using const_reference = T const&;
    using const_pointer   = typename allocator_traits<AT>::const_pointer;
    using difference_type = ptrdiff_t;
    using index_type      = ptrdiff_t;
    using size_type       = ptrdiff_t;
    using size_tuple      = tuple<size_type, size_type>;
    ...
};
```

DR Matrix Engine – Nested Type Aliases

```
template<class T, class AT>
class dr_matrix_engine
{
  public:
    ...

    using is_fixed_size   = false_type;
    using is_resizable    = true_type;

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

    using column_view_type    = matrix_column_view<dr_matrix_engine>;
    using row_view_type       = matrix_row_view<dr_matrix_engine>;
    using transpose_view_type = matrix_transpose_view<dr_matrix_engine>;

    ...
};
```

DR Matrix Engine – Special Member Functions and Constructors

```
template<class T, class AT>
class dr_matrix_engine
{
  public:
    ...

    ~dr_matrix_engine();
    dr_matrix_engine();
    dr_matrix_engine(dr_matrix_engine&& rhs) noexcept;
    dr_matrix_engine(dr_matrix_engine const& rhs);

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

    dr_matrix_engine(size_type rows, size_type cols);
    dr_matrix_engine(size_type rows, size_type cols, size_type rowcap, size_type colcap);

    ...
};
```

DR Matrix Engine – Const Element Access and Properties

```
template<class T, class AT>
class dr_matrix_engine
{
  public:
    ...

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

    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;

    ...
};
```

DR Matrix Engine – Mutable Element Access

```
template<class T, class AT>
class dr_matrix_engine
{
  public:
    ...

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

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

    void    swap(dr_matrix_engine& other) noexcept;
    void    swap_columns(index_type c1, index_type c2);
    void    swap_rows(index_type r1, index_type r2);

    ...
};
```

DR Matrix Engine – Capacity and Size Management

```
template<class T, class AT>
class dr_matrix_engine
{
  public:
    ...

    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);

    ...
};
```

DR Matrix Engine – Possible Private Implementation

```
template<class T, class AT>
class dr_matrix_engine
{
    ...

  private:
    pointer         mp_elems;
    size_type       m_rows;
    size_type       m_cols;
    size_type       m_rowcap;
    size_type       m_colcap;
    allocator_type  m_alloc;

    ...
};
```

FS Matrix Engine – Nested Type Aliases

```
template<class T, int32_t R, int32_t C>
class fs_matrix_engine
{
    static_assert(is_matrix_element_v<T>);
    static_assert(R >= 1  &&  C >= 1);

  public:
    using engine_category = 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 = ptrdiff_t;
    using index_type      = int_fast32_t;
    using size_type       = int_fast32_t;
    using size_tuple      = tuple<size_type, size_type>;
    ...
};
```

FS Matrix Engine – Nested Type Aliases

```
template<class T, int32_t R, int32_t C>
class fs_matrix_engine
{
  public:
    ...

    using is_fixed_size   = true_type;
    using is_resizable    = false_type;

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

    using column_view_type    = matrix_column_view<fs_matrix_engine>;
    using row_view_type       = matrix_row_view<fs_matrix_engine>;
    using transpose_view_type = matrix_transpose_view<fs_matrix_engine>;

    ...
};
```

FS Matrix Engine – Special Member Functions

```
template<class T, int32_t R, int32_t C>
class fs_matrix_engine
{
  public:
    ...

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

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

    ...
};
```

FS Matrix Engine – Const Element Access and Properties

```
template<class T, int32_t R, int32_t C>
class fs_matrix_engine
{
  public:
    ...

    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;

    ...
};
```

FS Matrix Engine – Mutable Element Access

```
template<class T, int32_t R, int32_t C>
class fs_matrix_engine
{
  public:
    ...

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

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

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

    ...
};
```

FS Matrix Engine – Possible Private Implementation

```
template<class T, int32_t R, int32_t C>
class fs_matrix_engine
{
    ...

  private:
    T   ma_elems[R*C];
};
```

Matrix Transpose View – Nested Type Aliases

```
template<class ET>
class matrix_transpose_view
{
    static_assert(detail::is_matrix_engine_v<ET>);

  public:
    using engine_type     = ET;
    using engine_category = const_matrix_engine_tag;
    using element_type    = typename engine_type::element_type;
    using value_type      = typename engine_type::value_type;
    using reference       = typename engine_type::const_reference;
    using pointer         = typename engine_type::const_pointer;
    using const_reference = typename engine_type::const_reference;
    using const_pointer   = typename engine_type::const_pointer;
    using difference_type = typename engine_type::difference_type;
    using index_type      = typename engine_type::index_type;
    using size_type       = typename engine_type::size_type;
    using size_tuple      = typename engine_type::size_tuple;
     ...
};
```

Matrix Transpose View – Nested Type Aliases

```
template<class ET>
class matrix_transpose_view
{
  public:
    ...

    using is_fixed_size   = typename engine_type::is_fixed_size;
    using is_resizable    = false_type;

    using is_column_major = typename engine_type::is_row_major;
    using is_dense        = typename engine_type::is_dense;
    using is_rectangular  = typename engine_type::is_rectangular;
    using is_row_major    = typename engine_type::is_column_major;

    using column_view_type    = matrix_column_view<matrix_transpose_view>;
    using row_view_type       = matrix_row_view<matrix_transpose_view>;
    using transpose_view_type = matrix_transpose_view<matrix_transpose_view>;

    ...
};
```

Matrix Transpose View – Const Element Access and Properties

```
template<class ET>
class matrix_transpose_view
{
  public:
    ...

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

    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;

    ...
};
```

Matrix Transpose View – Possible Private Implementation

```
template<class ET>
class matrix_transpose_view
{
    ...

  private:
    engine_type const*  mp_other;

    ...
};
```

matrix_operation_traits

Interface Overview – Math Objects

- **Math objects** (`vector` and `matrix`) model mathematical abstractions
  - Use engines to manage elements
  - Use operation traits to suggest arithmetic implementation
  - Present a consolidated interface to the arithmetic operators

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

//- Primary mathematical object types.
//
template<class ET, class OT=matrix_operation_traits>  class vector;
template<class ET, class OT=matrix_operation_traits>  class matrix;
```

Matrix Operation Traits – Element Promotion

```
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>;

    ...
};
```

Matrix Operation Traits – Engine Promotion

```
struct matrix_operation_traits
{
    ...

    //- 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>;

    ...
};
```

Matrix Operation Traits – Arithmetic

```
struct matrix_operation_traits
{
    ...

    //- 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>;
};
```

vector

Vector – Nested Type Aliases

```
template<class ET, class OT>
class vector
{
    static_assert(detail::is_vector_engine_v<ET>);

  public:
    using engine_type     = ET;
    using element_type    = typename engine_type::element_type;
    using reference       = typename engine_type::reference;
    using const_reference = typename engine_type::const_reference;
    using iterator        = typename engine_type::iterator;
    using const_iterator  = typename engine_type::const_iterator;
    using index_type      = typename engine_type::index_type;
    using size_type       = typename engine_type::size_type;

    ...
};
```

Vector – Nested Type Aliases

```
template<class ET, class OT>
class vector
{
  public:
    ...

    using transpose_type  = vector const&;
    using hermitian_type  = conditional_t<is_complex_v<element_type>, vector, transpose_type>;

    using is_fixed_size   = typename engine_type::is_fixed_size;
    using is_resizable    = typename engine_type::is_resizable;

    using is_column_major = typename engine_type::is_column_major;
    using is_dense        = typename engine_type::is_dense;
    using is_rectangular  = typename engine_type::is_rectangular;
    using is_row_major    = typename engine_type::is_row_major;

    ...
};
```

Vector – Special Member Functions

```
template<class ET, class OT>
class vector
{
  public:
    ...

    ~vector() = default;

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

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

    ...
};
```

Vector – Other Constructors and Assignment Operators

```
template<class ET, class OT>
class vector
{
  public:
    ...

    template<class ET2, class OT2>
    constexpr vector(vector<ET2, OT2> const& src);

    template<class ET2 = ET, detail::enable_if_resizable<ET, ET2> = true>
    constexpr vector(size_type elems);

    template<class ET2 = ET, detail::enable_if_resizable<ET, ET2> = true>
    constexpr vector(size_type elems, size_type elemcap);

    template<class ET2, class OT2>
    constexpr vector&     operator =(vector<ET2, OT2> const& rhs);

    ...
};
```

Vector – Const Element Access and Properties

```
template<class ET, class OT>
class vector
{
  public:
    ...

    constexpr const_reference   operator ()(index_type i) const;
    constexpr const_iterator    begin() const noexcept;
    constexpr const_iterator    end() const noexcept;

    constexpr size_type         capacity() const noexcept;
    constexpr index_type        elements() const noexcept;
    constexpr size_type         size() const noexcept;

    constexpr transpose_type    t() const;
    constexpr hermitian_type    h() const;

    ...
};
```

Vector – Mutable Element Operations

```
template<class ET, class OT>
class vector
{
  public:
    ...

    constexpr reference operator ()(index_type i);
    constexpr iterator  begin() noexcept;
    constexpr iterator  end() noexcept;

    constexpr void      assign(vector const& rhs);
    template<class ET2, class OT2>
    constexpr void      assign(vector<ET2, OT2> const& rhs);

    constexpr void      swap(vector& rhs) noexcept;
    constexpr void      swap_elements(index_type i, index_type j) noexcept;

    ...
};
```

Vector – Size and Capacity Management

```
template<class ET, class OT>
class vector
{
  public:
    ...

    template<class ET2 = ET, detail::enable_if_resizable<ET, ET2> = true>
    constexpr void      reserve(size_type elemcap);

    template<class ET2 = ET, detail::enable_if_resizable<ET, ET2> = true>
    constexpr void      resize(size_type elems);

    template<class ET2 = ET, detail::enable_if_resizable<ET, ET2> = true>
    constexpr void      resize(size_type elems, size_type elemcap);

    ...
};
```

Vector – Private Implementation

```
template<class ET, class OT>
class vector
{
    ...

  private:
    engine_type     m_engine;
};
```

matrix

Matrix – Nested Type Aliases

```
template<class ET, class OT>
class matrix
{
    static_assert(detail::is_matrix_engine_v<ET>);

  public:
    using engine_type     = ET;
    using element_type    = typename engine_type::element_type;
    using reference       = typename engine_type::reference;
    using const_reference = typename engine_type::const_reference;
    using index_type      = typename engine_type::index_type;
    using size_type       = typename engine_type::size_type;
    using size_tuple      = typename engine_type::size_tuple;

    ...
};
```

Matrix – Nested Type Aliases

```
template<class ET, class OT>
class matrix
{
  public:
    ...

    using column_type     = vector<matrix_column_view<engine_type>, OT>;
    using row_type        = vector<matrix_row_view<engine_type>, OT>;
    using transpose_type  = matrix<matrix_transpose_view<engine_type>, OT>;
    using hermitian_type  = conditional_t<is_complex_v<element_type>, matrix, transpose_type>;

    using is_fixed_size   = typename engine_type::is_fixed_size;
    using is_resizable    = typename engine_type::is_resizable;

    using is_column_major = typename engine_type::is_column_major;
    using is_dense        = typename engine_type::is_dense;
    using is_rectangular  = typename engine_type::is_rectangular;
    using is_row_major    = typename engine_type::is_row_major;

    ...
};
```

Matrix – Special Member Functions

```
template<class ET, class OT>
class matrix
{
  public:
    ...

    ~matrix() = default;
    constexpr matrix() = default;
    constexpr matrix(matrix&&) noexcept = default;
    constexpr matrix(matrix const&) = default;

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

    ...
};
```

Matrix – Other Constructors and Assignment

```
template<class ET, class OT>
class matrix
{
    ...
    template<class ET2, class OT2>
    matrix(matrix<ET2, OT2> const& src);
    template<class ET2, class OT2>
    constexpr matrix&       operator =(matrix<ET2, OT2> const& rhs);

    template<class ET2 = ET, detail::enable_if_resizable<ET, ET2> = true>
    constexpr matrix(size_tuple size);

    template<class ET2 = ET, detail::enable_if_resizable<ET, ET2> = true>
    constexpr matrix(size_type rows, size_type cols);

    template<class ET2 = ET, detail::enable_if_resizable<ET, ET2> = true>
    constexpr matrix(size_tuple size, size_tuple cap);

    template<class ET2 = ET, detail::enable_if_resizable<ET, ET2> = true>
    constexpr matrix(size_type rows, size_type cols, size_type rowcap, size_type colcap);
    ...
};
```

Matrix – Const Element Access and Properties

```
template<class ET, class OT>
class matrix
{
  public:
    ...
    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 column_type       column(index_type j) const noexcept;
    constexpr row_type          row(index_type i) const noexcept;
    constexpr transpose_type    t() const;
    constexpr hermitian_type    h() const;
    ...
};
```

Matrix – Mutable Element Operations

```
template<class ET, class OT>
class matrix
{
  public:
    ...
    constexpr reference operator ()(index_type i, index_type j);

    constexpr void      assign(matrix const& rhs);
    template<class ET2, class OT2>
    constexpr void      assign(matrix<ET2, OT2> const& rhs);

    template<class ET2 = ET, detail::enable_if_mutable<ET, ET2> = true>
    constexpr void      swap(matrix& rhs) noexcept;

    template<class ET2 = ET, detail::enable_if_mutable<ET, ET2> = true>
    constexpr void      swap_columns(index_type i, index_type j) noexcept;

    template<class ET2 = ET, detail::enable_if_mutable<ET, ET2> = true>
    constexpr void      swap_rows(index_type i, index_type j) noexcept;
    ...
};
```

Matrix – Capacity Management

```
template<class ET, class OT>
class matrix
{
  public:
    ...

    template<class ET2 = ET, detail::enable_if_resizable<ET, ET2> = true>
    constexpr void  reserve(size_tuple cap);

    template<class ET2 = ET, detail::enable_if_resizable<ET, ET2> = true>
    constexpr void  reserve(size_type rowcap, size_type colcap);

    ...
};
```

Matrix – Size Management

```
template<class ET, class OT>
class matrix
{
  public:
    ...

    template<class ET2 = ET, detail::enable_if_resizable<ET, ET2> = true>
    constexpr void  resize(size_tuple size);

    template<class ET2 = ET, detail::enable_if_resizable<ET, ET2> = true>
    constexpr void  resize(size_type rows, size_type cols);

    ...
};
```

Matrix – Size and Capacity Management

```
template<class ET, class OT>
class matrix
{
  public:
    ...

    template<class ET2 = ET, detail::enable_if_resizable<ET, ET2> = true>
    constexpr void  resize(size_tuple size, size_tuple cap);

    template<class ET2 = ET, detail::enable_if_resizable<ET, ET2> = true>
    constexpr void  resize(size_type rows, size_type cols, size_type rowcap, size_type colcap);

    ...
};
```

Matrix – Private Implementation

```
template<class ET, class OT>
class matrix
{
    ...

  private:
    engine_type     m_engine;
};
```

How Does it Work?

Let’s Add Two Matrices

```
//- Create a couple of 4x4 matrices
//
dyn_matrix<float>        m1(4, 4);
fs_matrix<double, 4, 4>  m2;
```

Let’s Add Two Matrices

```
//- Create a couple of 4x4 matrices
//
matrix<dr_matrix_engine<float, allocator<double>>, matrix_operation_traits>  m1(4, 4);
matrix<fs_matrix_engine<double, 4, 4>, matrix_operation_traits>              m2;
```

Let’s Add Two Matrices

```
//- Create a couple of 4x4 matrices
//
matrix<dr_matrix_engine<float, allocator<double>>, matrix_operation_traits>  m1(4, 4);
matrix<fs_matrix_engine<double, 4, 4>, matrix_operation_traits>              m2;

//- Set the values of their elements
//
f(m1);
f(m2);
```

Let's Add Two Matrices

```
//- Create a couple of 4x4 matrices
//
matrix<dr_matrix_engine<float, allocator<double>>, matrix_operation_traits>  m1(4, 4);
matrix<fs_matrix_engine<double, 4, 4>, matrix_operation_traits>             m2;

//- Set the values of their elements
//
f(m1);
f(m2);

//- Add them together.  What is the type of mr?  Specifically,
//    What is the element type of mr?
//    What is the engine type of mr?
//    What is the operation traits type of mr?
//
auto    mr = m1 + m2;
```

Let’s Add Two Matrices

```
//- Create a couple of 4x4 matrices
//
matrix<dr_matrix_engine<float, allocator<double>>, matrix_operation_traits>  m1(4, 4);
matrix<fs_matrix_engine<double, 4, 4>, matrix_operation_traits>             m2;

//- Set the values of their elements
//
f(m1);
f(m2);

//- Add them together.  What is the type of mr?  Specifically,
//    What is the element type of mr?
//    What is the engine type of mr?
//    What is the operation traits type of mr?
//
auto    mr = m1 + m2;
```

Matrix Addition Operator

```
//- The addition operator, which relies to the addition traits to do the actual work.
//
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);
}
```

Matrix Addition Operator

```
//- The addition operator, which relies to the addition traits to do the actual work.
//
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);
}

//  op_traits = ?
```

Operation Traits Selector

```
//- Alias template interface to selector trait.
//
template<class T1, class T2>
using matrix_operation_traits_selector_t =
                                typename matrix_operation_traits_selector<T1,T2>::traits_type;


//- Selector trait primary template
//
template<class T1, class T2>
struct matrix_operation_traits_selector;


//- Partial specialization for equal operation traits types
//
template<class T1>
struct matrix_operation_traits_selector<T1, T1>
{
    using traits_type = T1;
};
```

Operation Traits Selector

```
//- Specializations involving matrix_operation_traits.
//
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;
};
```

Matrix Addition Operator

```
//- The addition operator, which relies to the addition traits to do the actual work.
//
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);
}

//  op_traits  = matrix_operation_traits
```

Matrix Addition Operator

```
//- The addition operator, which relies to the addition traits to do the actual work.
//
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);
}

//  op_traits  = matrix_operation_traits
//  op1_type   = matrix<dr_matrix_engine<float, allocator<float>>, matrix_operation_traits>
```

Matrix Addition Operator

```
//- The addition operator, which relies to the addition traits to do the actual work.
//
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);
}

//  op_traits  = matrix_operation_traits
//  op1_type   = matrix<dr_matrix_engine<float, allocator<float>>, matrix_operation_traits>
//  op2_type   = matrix<fs_matrix_engine<double, 4, 4>, matrix_operation_traits>
```

Matrix Addition Operator

```
//- The addition operator, which relies to the addition traits to do the actual work.
//
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);
}

//  op_traits  = matrix_operation_traits
//  op1_type   = matrix<dr_matrix_engine<float, allocator<float>>, matrix_operation_traits>
//  op2_type   = matrix<fs_matrix_engine<double, 4, 4>, matrix_operation_traits>
//
//  add_traits = matrix_addition_traits<matrix_operation_traits,
//                   matrix<dr_matrix_engine<float, allocator<float>>, matrix_operation_traits>,
//                   matrix<fs_matrix_engine<double, 4, 4>, matrix_operation_traits>>
```

Matrix Addition Traits

```
//- The matrix_addition_traits type is an arithmetic traits type that provides the default
//  mechanism for determining the resulting type, and computing the result, of a matrix/matric
//  or vector/vector addition.
//
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& v1, matrix<ET2, OT2> const& v2);
};
```

Matrix Addition Traits

```
//- The matrix_addition_traits type is an arithmetic traits type that provides the default
//  mechanism for determining the resulting type, and computing the result, of a matrix/matric
//  or vector/vector addition.
//
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& v1, matrix<ET2, OT2> const& v2);
};

//  engine_type = ?
```

Matrix Addition Engine Traits

```
//- The matrix_addition_engine_traits type provides the default mechanism for determining the
//  correct engine type for a matrix/matrix addition.  This is the primary template.
//
template<class OT, class ET1, class ET2>
struct matrix_addition_engine_traits
{
    static_assert(detail::engines_match_v<ET1, ET2>);

    using element_type_1 = typename ET1::element_type;
    using element_type_2 = typename ET2::element_type;
    using element_type   = matrix_addition_element_t<OT, element_type_1, element_type_2>;
    using engine_type    = conditional_t<detail::is_matrix_engine_v<ET1>,
                                         dr_matrix_engine<element_type, allocator<element_type>>,
                                         dr_vector_engine<element_type, allocator<element_type>>>;
};
```

Matrix Addition Engine Traits

```
//- Traits type matrix_addition_engine_traits partially specialized for the case of
//
//    dr_matrix_engine + fs_matrix_engine.
//
template<class OT, class T1, class A1, class T2, int32_t R2, int32_t C2>
struct matrix_addition_engine_traits<OT,
                                     dr_matrix_engine<T1, A1>,
                                     fs_matrix_engine<T2, R2, C2>>
{
    using element_type = matrix_addition_element_t<OT, T1, T2>;
    using alloc_type   = detail::rebind_alloc_t<A1, element_type>;
    using engine_type  = dr_matrix_engine<element_type, alloc_type>;
};
```

Matrix Addition Engine Traits

```
//- Traits type matrix_addition_engine_traits partially specialized for the case of
//
//    dr_matrix_engine + fs_matrix_engine.
//
template<class OT, class T1, class A1, class T2, int32_t R2, int32_t C2>
struct matrix_addition_engine_traits<OT,
                                     dr_matrix_engine<T1, A1>,
                                     fs_matrix_engine<T2, R2, C2>>
{
    using element_type = matrix_addition_element_t<OT, T1, T2>;
    using alloc_type   = detail::rebind_alloc_t<A1, element_type>;
    using engine_type  = dr_matrix_engine<element_type, alloc_type>;
};

//  element_type = ?
```

Matrix Element Addition Traits

```
//- The matrix_addition_elment_traits type provides the default mechanism for determining
//  the result of adding two elements of (possibly) different types.
//
template<class T1, class T2>
struct matrix_addition_element_traits
{
    using element_type = decltype(declval<T1>() + declval<T2>());
};
```

Matrix Element Addition Traits

```
//- The matrix_addition_elment_traits type provides the default mechanism for determining
//  the result of adding two elements of (possibly) different types.
//
template<class T1, class T2>
struct matrix_addition_element_traits
{
    using element_type = decltype(declval<T1>() + declval<T2>());
};

//  element_type = decltype(declval<float>() + declval<double>())
//               = decltype(float&& + double&&)
//               = double
```

Matrix Addition Engine Traits

```
//- Traits type matrix_addition_engine_traits partially specialized for the case of
//
//    dr_matrix_engine + fs_matrix_engine.
//
template<class OT, class T1, class A1, class T2, int32_t R2, int32_t C2>
struct matrix_addition_engine_traits<OT,
                                     dr_matrix_engine<T1, A1>,
                                     fs_matrix_engine<T2, R2, C2>>
{
    using element_type = matrix_addition_element_t<OT, T1, T2>;
    using alloc_type   = detail::rebind_alloc_t<A1, element_type>;
    using engine_type  = dr_matrix_engine<element_type, alloc_type>;
};

//- In this example,
//
//  element_type = double
```

Matrix Addition Engine Traits

```
//- Traits type matrix_addition_engine_traits partially specialized for the case of
//
//    dr_matrix_engine + fs_matrix_engine.
//
template<class OT, class T1, class A1, class T2, int32_t R2, int32_t C2>
struct matrix_addition_engine_traits<OT,
                                     dr_matrix_engine<T1, A1>,
                                     fs_matrix_engine<T2, R2, C2>>
{
    using element_type = matrix_addition_element_t<OT, T1, T2>;
    using alloc_type   = detail::rebind_alloc_t<A1, element_type>;
    using engine_type  = dr_matrix_engine<element_type, alloc_type>;
};

//  element_type = double
//  alloc_type   = allocator<double>
```

Matrix Addition Engine Traits

```
//- Traits type matrix_addition_engine_traits partially specialized for the case of
//
//    dr_matrix_engine + fs_matrix_engine.
//
template<class OT, class T1, class A1, class T2, int32_t R2, int32_t C2>
struct matrix_addition_engine_traits<OT,
                                     dr_matrix_engine<T1, A1>,
                                     fs_matrix_engine<T2, R2, C2>>
{
    using element_type = matrix_addition_element_t<OT, T1, T2>;
    using alloc_type   = detail::rebind_alloc_t<A1, element_type>;
    using engine_type  = dr_matrix_engine<element_type, alloc_type>;
};

//  element_type = double
//  alloc_type   = allocator<double>
//  engine_type  = dr_matrix_engine<double, allocator<double>>
```

Matrix Addition Traits

```
//- The standard addition traits type provides the default mechanism for computing the result
//  of a matrix/matrix or vector/vector addition.
//
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& v1, matrix<ET2, OT2> const& v2);
};

//  engine_type = dr_matrix_engine<double, allocator<double>>
```

Matrix Addition Traits

```
//- The standard addition traits type provides the default mechanism for computing the result
//  of a matrix/matrix or vector/vector addition.
//
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& v1, matrix<ET2, OT2> const& v2);
};

//  engine_type = dr_matrix_engine<double, allocator<double>>
//  op_traits   = matrix_operation_traits
```

Matrix Addition Traits

```
//- The standard addition traits type provides the default mechanism for computing the result
//  of a matrix/matrix or vector/vector addition.
//
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& v1, matrix<ET2, OT2> const& v2);
};

//  engine_type = dr_matrix_engine<double, allocator<double>>
//  op_traits   = matrix_operation_traits
//  result_type = matrix<dr_matrix_engine<double, allocator<double>>, matrix_operation_traits>
```

Matrix Addition Operator

```
//- The addition operator, which relies to the addition traits to do the actual work.
//
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);
}

//  op_traits  = matrix_operation_traits
//  op1_type   = matrix<dr_matrix_engine<float, allocator<float>>, matrix_operation_traits>
//  op2_type   = matrix<fs_matrix_engine<double, 4, 4>, matrix_operation_traits>
//
//  add_traits = matrix_addition_traits<matrix_operation_traits,
//                   matrix<dr_matrix_engine<float, allocator<float>>, matrix_operation_traits>,
//                   matrix<fs_matrix_engine<double, 4, 4>, matrix_operation_traits>>
```

Matrix Addition Operator

```
//- The addition operator, which relies to the addition traits to do the actual work.
//
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);
}

//  op_traits  = matrix_operation_traits
//  op1_type   = matrix<dr_matrix_engine<float, allocator<float>>, matrix_operation_traits>
//  op2_type   = matrix<fs_matrix_engine<double, 4, 4>, matrix_operation_traits>
//
//  add_traits = matrix_addition_traits<matrix_operation_traits,
//                   matrix<dr_matrix_engine<float, allocator<float>>, matrix_operation_traits>,
//                   matrix<fs_matrix_engine<double, 4, 4>, matrix_operation_traits>>
```

Matrix Addition Traits

```
//- The standard addition traits type provides the default mechanism for computing the result
//  of a matrix/matrix or vector/vector addition.
//
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& v1, matrix<ET2, OT2> const& v2);
};

//  engine_type = dr_matrix_engine<double, allocator<double>>
//  op_traits   = matrix_operation_traits
//  result_type = matrix<dr_matrix_engine<double, allocator<double>>, matrix_operation_traits>
```

Matrix Addition Traits – add( )

```
template<class OT, class ET1, class OT1, class ET2, class OT2>  inline auto
matrix_addition_traits<OT, matrix<ET1, OT1>, matrix<ET2, OT2>>::add
(matrix<ET1, OT1> const& m1, matrix<ET2, OT2> const& m2) -> result_type
{
    //- Code would go here to ensure that m1.size() == m2.size()...

    result_type     mr;

    //- Code would go here to ensure that mr.size() == m1.size()...

    //- Add the elements
    //
    for (auto i = 0;  i < m1.rows();  ++i)
    {
        for (auto j = 0;  j < m1.columns();  ++j)
        {
            mr(i, j) = m1(i, j) + m2(i, j);
        }
    }
    return mr;
}
```

Matrix Addition Traits – add( )

```
template<class OT, class ET1, class OT1, class ET2, class OT2>  inline auto
matrix_addition_traits<OT, matrix<ET1, OT1>, matrix<ET2, OT2>>::add
(matrix<ET1, OT1> const& m1, matrix<ET2, OT2> const& m2) -> result_type
{
    //- Code would go here to ensure that m1.size() == m2.size()...

    result_type     mr;

    //- Code would go here to ensure that mr.size() == m1.size()...

    //- Add the elements
    //
    for (auto i = 0;  i < m1.rows();  ++i)
    {
        for (auto j = 0;  j < m1.columns();  ++j)
        {
            mr(i, j) = m1(i, j) + m2(i, j);
        }
    }
    return mr;
}
```

Matrix Addition Operator

```
//- The addition operator, which relies to the addition traits to do the actual work.
//
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);
}

//  op_traits  = matrix_operation_traits
//  op1_type   = matrix<dr_matrix_engine<float, allocator<float>>, matrix_operation_traits>
//  op2_type   = matrix<fs_matrix_engine<double, 4, 4>, matrix_operation_traits>
//
//  add_traits = matrix_addition_traits<matrix_operation_traits,
//                   matrix<dr_matrix_engine<float, allocator<float>>, matrix_operation_traits>,
//                   matrix<fs_matrix_engine<double, 4, 4>, matrix_operation_traits>>
```

Let's Add Two Matrices

```
//- Create a couple of 4x4 matrices
//
matrix<dr_matrix_engine<float, allocator<double>>, matrix_operation_traits>  m1(4, 4);
matrix<fs_matrix_engine<double, 4, 4>, matrix_operation_traits>              m2;

//- Set the values of their elements
//
f(m1);
f(m2);

//- Add them together.
//      What is the element type of mr?           double
//      What is the engine type of mr?            dr_matrix_engine<double, allocator<double>>
//      What is the operation traits type of mr?  matrix_operation_traits
//
//      mr --> matrix<dr_matrix_engine<double, allocator<double>>, matrix_operation_traits>
//
auto    mr = m1 + m2;
```

Customization

Custom Element Type

Custom Element Type

```
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);
};
```

Custom Element Type

```
                   new_num  operator +(new_num lhs, new_num rhs);
template<class U>  new_num  operator +(new_num lhs, U rhs);
template<class U>  new_num  operator +(U lhs, new_num rhs);

                   new_num  operator -(new_num lhs, new_num rhs);
template<class U>  new_num  operator -(new_num lhs, U rhs);
template<class U>  new_num  operator -(U lhs, new_num rhs);

                   new_num  operator *(new_num lhs, new_num rhs);
template<class U>  new_num  operator *(new_num lhs, U rhs);
template<class U>  new_num  operator *(U lhs, new_num rhs);

                   new_num  operator /(new_num lhs, new_num rhs);
template<class U>  new_num  operator /(new_num lhs, U rhs);
template<class U>  new_num  operator /(U lhs, new_num rhs);
```

Custom Element Type

```
//- Goal: A matrix with elements of type new_num participates in arithmetic expressions.
//
```

Custom Element Type

```
//- Goal: A matrix with elements of type new_num participates in arithmetic expressions.
//
template<>
struct std::math::number_traits<new_num>
{
    using is_nc_ring = true_type;
    using is_ring    = true_type;
    using is_field   = true_type;
};
```

Custom Element Type

```
//- Goal: A matrix with elements of type new_num participates in arithmetic expressions.
//
template<>
struct std::math::number_traits<new_num>
{
    using is_nc_ring = true_type;
    using is_ring    = true_type;
    using is_field   = true_type;
};

//  template<class U>  new_num  operator +(new_num lhs, U rhs);
```

Custom Element Type

```
//- Goal: A matrix with elements of type new_num participates in arithmetic expressions.
//
template<>
struct std::math::number_traits<new_num>
{
    using is_nc_ring = true_type;
    using is_ring    = true_type;
    using is_field   = true_type;
};

//  template<class U>  new_num  operator +(new_num lhs, U rhs);

dyn_matrix<float>           m1(4, 4);
fs_matrix<new_num, 4, 4>    m2;

...
```

Custom Element Type

```
//- Goal: A matrix with elements of type new_num participates in arithmetic expressions.
//
template<>
struct std::math::number_traits<new_num>
{
    using is_nc_ring = true_type;
    using is_ring    = true_type;
    using is_field   = true_type;
};

//  template<class U>  new_num  operator +(new_num lhs, U rhs);

dyn_matrix<float>          m1(4, 4);
fs_matrix<new_num, 4, 4>   m2;

...

//- mr --> ?
//
auto mr = m1 + m2;
```

Custom Element Type

```
//- Goal: A matrix with elements of type new_num participates in arithmetic expressions.
//
template<>
struct std::math::number_traits<new_num>
{
    using is_nc_ring = true_type;
    using is_ring    = true_type;
    using is_field   = true_type;
};

//  template<class U>  new_num  operator +(new_num lhs, U rhs);

dyn_matrix<float>          m1(4, 4);
fs_matrix<new_num, 4, 4>   m2;

...

//- mr --> matrix<dr_matrix_engine<new_num, allocator<new_num>>, matrix_operation_traits>
//
auto mr = m1 + m2;
```

Custom Element Promotion

Custom Element Promotion

```
//- Goal: Promote any float/float addition to double.
//
```

Custom Element Promotion

```
//- Goal: Promote any float/float addition to double.
//
template<class T1, class T2>
struct element_add_traits_tst;
```

Custom Element Promotion

```
//- Goal: Promote any float/float addition to double.
//
template<class T1, class T2>
struct element_add_traits_tst;


template<>
struct element_add_traits_tst<float, float>
{
    using element_type = double;
};
```

Custom Element Promotion

```
//- Goal: Promote any float/float addition to double.
//
template<class T1, class T2>
struct element_add_traits_tst;


template<>
struct element_add_traits_tst<float, float>
{
    using element_type = double;
};

//- 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>;
};
```

Custom Element Promotion

```
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 --> ?
//
auto  mr1 = m1 + m1;

//- mr2 --> ?
//
auto  mr2 = m1 + m2;

//- mr3 --> ?
//
auto  mr3 = m1 + m3;
```

Custom Element Promotion

```
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;
```

Custom Engine Type

Custom Engine

```
//- Goal: Create a new fixed-size engine type and use it in arithmetic expressions.
//
```

Custom Engine

```
//- Goal: Create a new fixed-size engine type and use it in arithmetic expressions.
//
template<class T, int32_t R, int32_t C>
class fs_matrix_engine_tst
{...};
```

Custom Engine

```
//- Goal: Create a new fixed-size engine type and use it in arithmetic expressions.
//
template<class T, int32_t R, int32_t C>
class fs_matrix_engine_tst
{...};


template<class OT, class ET1, class ET2>
struct engine_add_traits_tst;
```

Custom Engine

```
//- Goal: Create a new fixed-size engine type and use it in arithmetic expressions.
//
template<class T, int32_t R, int32_t C>
class fs_matrix_engine_tst
{...};


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>;
};
```

Custom Engine

```
//- Goal: Create a new fixed-size engine type and use it in arithmetic expressions.
//
...
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>;
};
```

Custom Engine

```
//- Goal: Create a new fixed-size engine type and use it in arithmetic expressions.
//
...

//- 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>;
};
```

Custom Engine

```
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 --> ?
//
auto  mr1 = m1 + m1;

//- mr2 --> ?
//
auto  mr2 = m2 + m2;

//- mr3 --> ?
//
auto  mr3 = m1 + m2;

//- mr4 --> ?
//
auto  mr4 = m1 + m3;
```

Custom Engine

```
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;
```

Custom Arithmetic

Custom Arithmetic

```
//- Goal: Call a specialized addition function for addition of fixed-size matrix objects
//  using the fixed-size test engine and having size 3x4.
//
```

Custom Arithmetic

```
//- 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 OTR, class OP1, class OP2>
struct addition_traits_tst;
```

Custom Arithmetic

```
//- 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 OTR, class OP1, class OP2>
struct addition_traits_tst;


template<class OTR>
struct addition_traits_tst<OTR,
                           matrix<fs_matrix_engine_tst<double, 3, 4>, OTR>,
                           matrix<fs_matrix_engine_tst<double, 3, 4>, OTR>>
{
    using op_traits   = OTR;
    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>, OTR> const& m1,
                            matrix<fs_matrix_engine_tst<double, 3, 4>, OTR> const& m2);
};
```

Custom Arithmetic

```
//- Goal: Call a specialized addition function for addition of fixed-size matrix objects
//  using the fixed-size test engine and having size 3x4.
//
...

//- 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>;
};
```

Custom Arithmetic

```
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 --> ?
//
auto  mr1 = m1 + m1;

//- mr2 --> ?
//
auto  mr2 = m1 + m2;

//- mr3 --> ?
//
auto  mr3 = m2 + m2;
```

Custom Arithmetic

```
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;

//- mr2 --> matrix<fs_matrix_engine_tst<double, 3, 4>, add_op_traits_tst>
//
auto  mr2 = m1 + m2;

//- mr3 --> matrix<fs_matrix_engine_tst<double, 3, 4>, add_op_traits_tst>
//
auto  mr3 = m2 + m2;
```

Custom Arithmetic

```
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()
```

Ongoing/Future Work

Ongoing Work

- Concept-ification
- Integration with `mdspan`
- Integration with executors
- Support for concurrency (execution contexts)
- Proof-of-concept sets of engines and arithmetic traits that:
  - Employ expression templates
  - Demonstrate concurrent/distributed arithmetic

Questions?

Thank You for Attending!

Paper: wg21.link/p1385

Talk: github.com/BobSteagall/CppNow2019

Code: github.com/BobSteagall/wg21/linear_algebra/code

Blogs: bobsteagall.com (Bob)

hatcat.com (Guy)