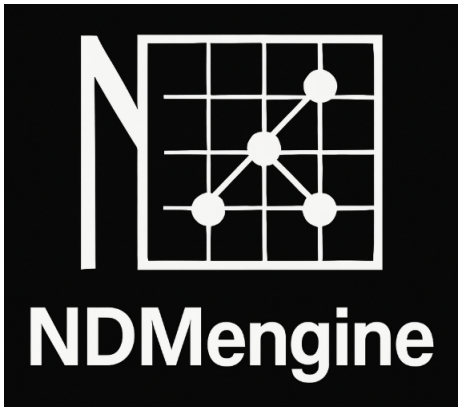

The N-Dimensional Matrix Engine Documentation

Marco Å. Ojeda

19/07/2025

NDMengine

Contents

1 Introduction **3**

2 Internal Structure and Indexing Mechanics **4**

3 Construction, Access, and Shape Introspection **6**

3.1 Constructor of the NDMatrix Class 6
3.2 Element Access via Multi-Dimensional Indexing 8
3.3 Flat Access and Size Introspection 11
3.4 Dimensions and Introspection and Matrix Reinitialization 13
3.5 Variadic Opertor Overloads for Intuitive Indexing 14
3.6 Reshaping and Iterator Interface 16
3.7 Submatrix Slicing via `view()` 18
3.8 Elementwise Arithmetic Operations 20
3.9 Scalar Arithmetic Operators (0-D) Broadcasting 22
3.10 Dimensional Reduction via `slice()` 26
3.11 Broadcasting Shape Inference 29
3.12 Index Conversion Utility Tools 31
3.12.1 `unravel_index` 31
3.12.2 `ravel_multi_index` 32

1 Introduction

NDMengine is a custom-built header-only matrix and data manipulation engine designed for high-dimensional numerical computing, developed under NOXIUM RESEARCH, a private project by Marco Å. Ojeda. The engine focuses on core tensor and matrix operations with a unique emphasis on N-dimensional shape broadcasting, modular memory control, and type-flexible storage using C++ templates.

At the heart of NDMengine lies the NDMatrix class template, a general-purpose data container that supports arbitrary-dimensional arrays (tensors). It manages its internal memory using `std::shared_ptr<T[]>`, enabling efficient deep copies and shared ownership models, while also providing robust type safety and flexibility via C++ templates. The library supports construction, reshaping, broadcasting, slicing, arithmetic, operations, scalar operations, and element-wise function application.

Key features include:

- Shape tracking through a `std::vector<size_t>` shape container and flattened memory layout.
- Static and dynamic broadcasting support.
- Operator overloading for arithmetic operations between matrices and scalars.
- Iterators and STL compatibility for range-based loops and container-style traversal.
- Functional style application of unary operations (e.g., `abs`, `sqrt`) through functor-style lambdas.
- Utility methods for reshaping, flattening, printing, and shape introspection.

The engine is structured for extensibility, with distinct internal function blocks. like broadcasting shape inference, index flattening and unflattening, and dimension compatibility checking. All components are modular and inline for header-only usage, allowing maximum portability. NDMengine is minimalistic in syntax and avoids dependencies, focusing on composability and performance. It can serve as a foundation for advanced linear algebra engines, GPU-accelerated frameworks, or DSLs in scientific computing, AI, or financial simulations. While it currently avoids advanced numerical methods or linear algebra libraries, its design accomodates future additions such as GPU targeting (e.g., via Metal or CUDA), custom DSL syntax parsing, or symbolic tensor operations.

This documentation marks the start of formalizing the engine's structure, interfaces, and design goals.

2 Internal Structure and Indexing Mechanics

This chapter details the private section of the `NDMatrix` class implementation. This internal design governs how the matrix stores its data, calculates access positions, and maintains consistency in memory layout. It is fundamental to understand how data access is both safe and efficient inside NDMengine.

Internally, three private members serve as the backbone for managing shape, strides and data. The `shape` vector keeps track of the extent of the matrix in each dimension. For instance, a shape like $\{3, 4, 5\}$ describes a $3 \times 4 \times 5$ tensor. The `strides` vector stores precomputed offsets that indicate how far to move in memory to reach the next element along a specific axis. This is essential for computing flattened indices in a row-major order. Finally, the `data` vector stores all matrix elements contiguously in a one-dimensional layout.

To access an element from a set N-dimensional indices, a flattened index must be computed. This is handled by two overloaded versions of `compute_flat_index`. One accepts an initializer list, while the other takes a standard vector. These functions verify that the number of indices matches the dimensionality of the matrix and that each index is within bounds. If the checks pass, the function calculates the flat index by summing the product of each index and its corresponding stride.

Pseudo Code: `compute_flat_index`

function `compute_flat_index`(indices)
    if `indices.size() != shape.size()`
        throw invalid argument exception
    `flat_index` $\leftarrow 0$
    for i from 0 **to** `shape.size() - 1`
        if `indices[i] >= shape[i]`
            throw out of range exception
        `flat_index` $\leftarrow$ `flat_index` + `indices[i]` $\times$ `strides[i]`
    return `flat_index`
end function

The method `compute_strides` calculates the stride vector from the matrix shape. It works in reverse order, from the last dimension toward the first, to ensure that the memory layout follows a row-major strategy. Starting with a stride of 1, it assigns this stride to the last dimension and multiplies it successively by each dimension size as it moves leftward through the shape. This prepares the stride vector for accurate index calculations.

Pseudo Code: `compute_strides`

function `compute_strides`()

`strides.resize(shape.size())` // Match number of dimensions
`stride` ← 1 // Start with stride 1 (innermost dimension)
for i from `shape.size() - 1` **down to** 0
`strides[i]` ← `stride` // Assign current stride
`stride` ← `stride` × `shape[i]` // Update stride for next dimension

end function

To better illustrate how these mechanisms operate in practice, consider the example of a matrix with shape = $\{2, 3\}$. The `compute_strides` function will iterate from the last dimension to the first, assigning strides as follows: Starting with stride = 1: for i = 1 (last dimension), strides[1] = 1; then stride *= shape[1] gives stride = 3; for i = 0, strides[0] = 3. Thus the final vector is $\{3, 1\}$. Now using `compute_flat_index` with an $\{1, 2\}$, the function checks that both indices are in range, then calculates the flat index as: flat_index= $1 \cdot 3 + 2 \cdot 1 = 3 + 2 = 5$. This index points to the 6th element in the 1D array (0-based indexing).

These internal mechanisms are tightly integrated to provide correctness and performance. Indexing errors are caught early with detailed exception messages. Memory is compact and contiguous, facilitating cache-friendly access patterns. All multi-dimensional operations, including slicing and reshaping, depend on this structure to function as expected. Eventually, STL compatibility and iterator support will make use of the same internal stride and shape layout to traverse elements efficiently in a way that feels idiomatic to C++ users. This serves as a conceptual bridge toward future extensibility of the engine. This design enforces both safety and speed, which are a cornerstone of the NDMengine philosophy.

Summary

- The private section includes `shape`, `strides`, and `data` vectors to manage dimensions, memory layout, and data storage.
- `compute_flat_index` translates N-dimensional indices into a single flat index, with bounds, checking and support for multiple input types.
- `compute_strides` calculates stride values in reverse to follow a row-major memory layout.
- Row-major layout ensures efficient, contiguous storage for optimized access patterns.
- Early exceptions handling provides robustness by catching indexing mismatches or violations.
- These mechanisms form the low-level foundation for multi-dimensional operations in NDMengine.

3 Construction, Access, and Shape Introspection

3.1 Constructor of the NDMatrix Class

This section begins with the constructor of the NDMatrix class, which is responsible for initializing the matrix structure and memory layout based on user-specified dimensionality and an optional initial value. This function forms the foundation upon which all higher-level behavior of the engine is built.

Pseudo Code: NDMatrix Constructor

constructor NDMatrix(dims, init)
    shape ← dims
    `compute_strides`() // Precompute how indices map to flat memory
    total_size ← `accumulate`(shape.begin(), shape.end(), 1, multiply)
    data.resize(total_size, init) // Initialize data vector with total size, filled with init
end constructor

The constructor takes two parameters: a `std::vector<size_t>` called `dims`, and an optional value `init` of type `T` (default-initialized if not specified). The `dims` vector defines the shape of the tensor, specifying the extent of each dimension in the N-dimensional space. For example, passing {3, 4, 5} creates a $3 \times 4 \times 5$ tensor. The `init` parameter defines the value with which all elements in the matrix will be initialized.

Initialization begins by assigning the `dims` vector to the `shape` member variable using initializer list syntax. Once shape is known, the constructor immediately calls the internal `compute_strides()` method. This step is crucial - it computes a mapping for each N-dimensional coordinate to a flat memory offset using a row-major memory layout. The strides are stored in a separate vector and are necessary for the engine to resolve multi-dimensional access requests into a 1D memory representation.

After computing the strides, the constructor proceeds to determine the total number of elements the matrix will store. This is done using the standard library function `std::accumulate`, which multiplies all values in the `shape` vector. The operation starts with an identity value of 1 and applies the `std::multiplies<>` binary operator across the entire vector. The resulting `total_size` gives the full number of data points that the N-dimensional tensor contains.

Finally, the `data` vector is resized to `total_size`, and all entries are initialized with the value `init`. This guarantees that the matrix is immediately ready for use and that all entries are populated consistently. Since `data` is a flat vector, it stores all tensor entries in linear memory, ensuring good performance and compatibility with modern hardware cache lines.

This constructor is a critical part of the engine design. It ensures that the internal state is valid, coherent, and ready for efficient access immediately after object instantiation. It integrates with the private mechanisms (shape, strides, and data), tightly coupling the public interface to the underlying design philosophy of memory layout determinism and dimensional clarity.

To demonstrate how this constructor might be used in practice, consider the following abstract example:

```
// Include the NDMengine header
#include "NDMengine.hpp"

int main() {
    // Construct a 2x3 tensor filled with the value 7.0
    NDMatrix<double> tensor({2, 3}, 7.0);

    // The internal layout will reserve 6 elements: 2 * 3
    // Each element is initialized to 7.0

    return 0;
}
```

This example is intentionally simple to emphasize the constructor's semantics. The shape $\{2, 3\}$ defines the tensor dimensions, while the `init` value 7.0 fills every entry in the underlying storage. Internally, the constructor computes strides $\{3, 1\}$ for row-major ordering and allocates a flat `data` vector of size 6. This kind of construction pattern is foundational to how tensors are brought into existence and prepared for subsequent manipulation.

In this next example we have a more elaborate example using two multi-dimensional tensors constructed, and printing the contents using nested for loops:

```
// Include the NDMengine header
#include "NDMengine.hpp"

int main() {
    // Construct a 2x3 tensor filled with 1.5
    NDMatrix<double> tensorA({2, 3}, 1.5);

    // Construct a 3x2x2 tensor filled with -2.0
    NDMatrix<double> tensorB({3, 2, 2}, -2.0);

    std::cout << "tensorA (2x3):\n";
    for (size_t i = 0; i < 2; ++i) {
        for (size_t j = 0; j < 3; ++j) {
            std::cout << tensorA({i, j}) << " ";
        }
        std::cout << "\n";
    }

    std::cout << "\ntensorB (3x2x2):\n";
    for (size_t i = 0; i < 3; ++i) {
```

```
        std::cout << "Block " << i << ":\n";
        for (size_t j = 0; j < 2; ++j) {
            for (size_t k = 0; k < 2; ++k) {
                std::cout << tensorB({i, j, k}) << " ";
            }
            std::cout << "\n";
        }
        std::cout << "\n";
    }

    return 0;
}
```

This demonstrates how to construct and inspect two multi-dimensional tensors using the NDMatrix constructor. The first tensor, `tensorA`, is a simple 2×3 matrix filled with the value 1.5. The second tensor, `tensorB`, is a more complex $3 \times 2 \times 2$ tensor initialized with the value -2.0. Both tensors are created solely using the constructor, which computes the internal shape, strides, and allocates memory accordingly. To verify the internal structure and initialization, the contents are printed using nested `for` loops: two levels for the 2D tensor and three levels for the 3D tensor. Each element is accessed using coordinate-based indexing, and the output is formatted in blocks to reflect the hierarchical structure of the data. This example highlights how the constructor supports immediate use of high-dimensional tensors without requiring additional setup or manual memory handling.

3.2 Element Access via Multi-Dimensional Indexing

The next segment of code of the NDMengine defines overloaded `at` member functions that serve as a flexible, safe, and intuitive access mechanism for elements stored in a flattened 1-dimensional array representing an N-dimensional matrix. The logic supports both constant and mutable access through different types of argument inputs, allowing access using either `std::initializer_list<size_t>` or `std::vector<size_t>`. These overloads enable the user to access data in a natural multi-dimensional indexing fashion while abstracting the details of flattening N-dimensional indices into a single linear offset.

The first overload provides a non-const, mutable reference to an element in the underlying flattened data container. It accepts an `std::initializer_list<size_t>` argument, which allows the user to write intuitive syntax such as `matrix.at(i,j)=value;`. Internally, it computes the linear index using a function named `compute_flat_index`, which translates the multi-dimensional indices into a flat index. This function is responsible for handling dimensionality consistency and ensuring index validity. The result is used to directly index into the `data` container, returning a mutable reference.

The second overload is a const-qualified version of the first. It uses the same signature but appends `const` to the end of the method declaration, which makes it callable on const-qualified instances of the object. This version is necessary to maintain const-correctness, ensuring that constant matrices cannot be modified accidentally. It returns a const reference to the element

located at the computed flat index, which means that while the element can be read, it cannot be modified through this interface.

The third and fourth overloads extend the indexing capabilities by supporting dynamic indexing via a `std::vector<size_t>`. These functions serve a similar purpose to the previous two, but they are designed to handle use cases such as slicing operations or runtime-generated indices where the number of dimensions is not known at compile-time. The mutable version, declared as `T& at(const std::vector<size_t>& indices)`, returns a modifiable reference to the data element, while the const-qualified counterpart `const T& at(const std::vector<size_t>& indices) const` returns a read-only reference.

All overloads rely on the core `compute_flat_index` function, which is assumed to validate bounds and handle the index translation appropriately. This design abstracts the complexity of N-dimensional layout handling and provides a user-friendly, type-safe interface for accessing matrix data. By using initializer lists and vector for input, the code supports both compile-time and runtime flexible indexing schemes, an essential feature for modern matrix and tensor manipulation libraries. The separation into const and non-const versions ensures the interface adheres to good C++ design practices, offering robustness, clarity and safety.

Pseudocode: Multi-Dimensional Element Access

```
// Returns a read-only reference to the element at the given N-dimensional index.
// Input: a list of dimension indices like [i, j, k]
// Output: a const reference to the data element at that index
function getElementConstByList(indices):
    flatIndex = computeFlatIndex(indices)
return data[flatIndex]

// Returns a mutable reference to the element using dynamic vector-based indexing.
// Input: a dynamic list of indices (e.g., from slicing or runtime-generated list)
// Output: a reference to the data element at that index
function getElementByVector(indices):
    flatIndex = computeFlatIndex(indices)
return data[flatIndex]


// Returns a read-only reference using dynamic vector-based indexing.
// Input: a dynamic list of indices
// Output: a const reference to the data element
function getElementConstByVector(indices):
    flatIndex = computeFlatIndex(indices)
return data[flatIndex]
```

To make this more understandable, we have an example that directly demonstrates the use of the `at` function from NDMengine in the context of, let's say, credit modeling. We will be computing the probability of default (PD) for a single borrower using logistic regression, the

borrower's features are stored in a 2D matrix, and we access them them via `at({i, j})`. In credit modeling, borrower data is often stored in a 2D matrix structure where each row represents a borrower and each column represents a feature, such as income, loan-to-value ratio, or credit history length. To compute the probability of default (PD) for a specific borrower, we must access these features in a clean and safe way. The `at({i, j})` function in NDMengine enables this by abstracting away flat index calculations and offering an intuitive interface for accessing elements in N-dimensional containers. In the following example, we calculate PD using logistic regression for one borrower (row 0) by accessing their features through `at({0, j})`.

```
// Assume an NDMengine 2D matrix with borrower data (1 borrower, 3 features)
NDMatrix<double> borrower_matrix({1, 3});  // 1 row, 3 columns

// Populate the features for borrower 0
borrower_matrix.at({0, 0}) = 45000;   // income
borrower_matrix.at({0, 1}) = 0.85;    // loan-to-value ratio
borrower_matrix.at({0, 2}) = 3;       // credit history length

// Logistic regression weights (pre-trained)
double w0 = -3.5;                  // intercept
double w1 = -0.00005;              // weight for income
double w2 = 2.5;                   // weight for loan-to-value
double w3 = -0.2;                  // weight for credit history

// Access borrower features using NDMengine's at() function
double income = borrower_matrix.at({0, 0});
double ltv = borrower_matrix.at({0, 1});
double history = borrower_matrix.at({0, 2});

// Compute linear score
double linear_score = w0
                    + w1 * income
                    + w2 * ltv
                    + w3 * history;

// Apply logistic function to obtain PD
double probability_of_default = 1.0 / (1.0 + exp(-linear_score));

// Output result
print("Borrower Probability of Default (PD): ", probability_of_default);
```

This example directly illustrates the role of the `at` function as a safe and elegant accessor for structured financial data. Instead of working with raw arrays and computing linear indices manually, we rely on `at({row, column})` to extract each borrower attribute. This makes the

code easier to read, less error-prone, and more scalable, especially when working with multiple borrowers or higher-dimensional financial models.

3.3 Flat Access and Size Introspection

This segment of code from the NDMengine provides foundational mechanisms for interacting directly with the underlying linear storage of an N-dimensional matrix. It offers both introspection capabilities—specifically the ability to query the total number of elements in the matrix—and direct flat index access to the raw, linearized representation of the matrix data. These functions are critical for internal performance optimizations, debugging utilities, and low-level manipulation, especially when higher-order abstractions like multidimensional `at(`$\{i, j, k\}$`)` calls are not required or need to by bypassed.

The `size()` function is a const-qualified method that returns the total number of scalar elements stored in the matrix, regardless of its dimensionality. This value represents the product of all dimension lengths. It forwards the call to the underlying `data` container. This function allows external users of the matrix to understand how much memory or how many scalar slots are being used. It is essential for iterators, memory checks, and range validations when performing bulk operations.

Following the `size()` function, the two overloaded `operator[]` functions provide direct access to the flattened data array using a single linear index. The first overload, `T& operator[](size_t i)`, returns a mutable reference to the element at the given index `i`. This is a non-const method, which permits modification of the underlying data. It is especially useful for scenarios that involve direct linear traversal of the matrix data, for instance, during memory copying, file output, or transformation operations applied across the entire matrix irrespective of its original structure.

The second overload, `const T& operator[](size_t i) const`, offers const-correctness for accessing the internal data. It ensures that read-only contexts - such as evaluations on const-qualified matrices - can still benefit from flat indexing for performance or inspection purposes. This const overload guarantees that users cannot modify data when the matrix is treated as an immutable object, thus preserving encapsulation and safety.

Together, these functions reflect the layered design philosophy of NDMengine. While the engine provides high-level, safe abstractions for N-dimensional indexing through functions like at `at(`$\{i, j, k\}$`)`, it also exposes low-level tools for developers who need precise control or performance optimization. The flat index access through `operator[]` allows interoperabilty with legacy systems, efficient iteration, and tight control in performance-critical code paths. The `size()` function complements this by allowing external code to dynamically query the bounds of safe iteration over the internal buffer. Importantly, all access is row-major, meaning that the data is stored and interpreted ina layout consistent with the majority of C++ matrix libraries. This design choice improves interoperabilty and reduces surprises for developers familiar with conventional memory layouts.

Here is pseudocode representing the logic from the code. It reflects the core behaviors: total size introspection and flat indexing for internal matrix data access.

```
// Function to return the total number of elements in the matrix
function size():
    return internal_data_array.length

// Function to return a mutable reference to the element at flat index i
function get_element(i):
    return reference to internal_data_array[i]

// Function to return a read-only reference to the element at flat index i
function get_element_const(i):
    return read-only reference to internal_data_array[i]
```

This pseudocode assumes that the internal data structure is a 1D array stored in a row-major order. It captures both const and non-const versions of the subscript operator for safe, context-aware access. These methods are typically used when the user already knows the flat index and wants fast access without the overhead of multi-dimensional index computation.

3.4 Dimensions and Introspection and Matrix Reinitialization

We introduce two utility functions that offer essential support for inspecting and modifying the structure and contents of an `NDMatrix` instance: `dimensions()` and `fill()`. These methods are part of the public interface, designed to be lightweight yet powerful tools for external interaction with the matrix's internal state.

`dimensions()`: **Exposing the Tensor Shape**. The `dimensions()` function provides direct access to the internal `shape` vector, which describes the extent of the matrix along each axis. The return type is a `const std::vector<size_t>&`, ensuring that users may inspect but not modify the shape directly.

This method is useful in scenarios where one needs to query the structure of a tensor for conditional logic, algorithm design, or debugging. Since the shape vector is stored in row-major order, the elements of the returned vector corresponds to the dimension sizes in left-to-right axis order. For instance, a tensor constructed with dimensions $\{3, 4, 5\}$ represents a $3 \times 4 \times 5$ tensor, with 3 as the outermost dimension and 5 as the innermost.

```
// Example usage
NDMatrix<float> tensor({2, 3, 4});
std::vector<size_t> dims = tensor.dimensions(); // dims = {2, 3, 4}
```

This method serves as a crucial tool in external modules or layers that dynamically inspect tensor structures, especially in adaptive algorithms, broadcasting logic, or when interfacing with external visualization or serialization tools.

`fill()`: **Reinitialization Matrix Contents**. The `fill(const T& value)` function provides an efficient mechanism for overwriting all elements in the internal data vector with a single scalar value. This operation is useful for matrix resets, testing, or clearing intermediate computation buffers.

Internally, it leverages `std::fill` to traverse the `data` vector from `begin()` to `end()`, setting each element to the specified value. Since `data` is a flat contiguous array storing all tensor entries in row-major order, this function affects the entire matrix uniformly, regardless of its shape.

The signature accepts a `const T&`, ensuring that large values are passed efficiently by reference, while allowing for immutable value semantics. This reflects the same type `T` as used in the `NDMatrix<T>` template, guaranteeing consistency and type safety.

```
NDMatrix<int> matrix({4, 4}, 0);
matrix.fill(9); // All 16 elements are now 9
```

The `fill()` method does not alter the shape or strides; it solely replaces the value in the memory buffer. It is particularly useful in iterative or simulation contexts where the tensor structure remains constant, but contents must be cleared or reset between stages. This utility function adheres to NDMengine's philosophy of providing low-friction, high-performance tools for matrix manipulation, blending STL idioms with tensor semantics in a clean, declarative syntax.

3.5 Variadic Opertor Overloads for Intuitive Indexing

To provide users with a more natural and ergonomic interface for element access, NDMengine implements a variadic overload of the function call operator `operator()`. These overloads allow users to access elements using familiar multi-index syntax like `matrix(i, j, k)` instead of explicitly calling `at(`$\{i, j, k\}$`)`.

Purpose and design. The variadic `operator()` serves as syntactic sugar around the underlying `.at(...)` function. It simplifies the experience of working with high-dimensional tensors by removing the need to manually construct `std::initializer_list<size_t>` or `std::vector<size_t>` constainers. Instead, the user can pass a sequence of indices directly as arguments.

For example, instead of writing:

```
value = matrix.at({1, 2, 3});
```

one can simply write:

```
value = matrix(1, 2, 3);
```

This is not only reduces verbosity but also mirrors the syntax seen in scientific computing environments such as NumPy and MATLAB.

Type-Safety Through `static_assert`. To ensure correctness, both overloads include a compile-time check using `static_assert`. Specifically, it verifies that each argument in the parameter pack is **convertible to `size_t`**, the internal type used for indexing. This protects against common programming errors such as passing floating-point numbers or other incompatible types.

```
static_assert((std::is_convertible_v<Args, size_t> && ...),
"All indices must be size_t-convertible.");
```

This expression uses fold syntax over a parameter pack to apply the `std::is_convertible` trait to each index. If any argument fails this check, compilation will fail with a clear diagnostic message. This level of static type enforcement upholds the engine's goal of robust and safe tensor manipulation.

Forwarding to `.at(...)` with Static Casts. Both the const and non-const versions of `operator()` internally forward the unpacked indices to the `.at(...)` function by wrapping them in a `std::initializer_list<size_t>`, after applying `static_cast<size_t>` to each one. This guarantees consistent type treatment and avoids ambiguity.

```
return at({static_cast<size_t>(args)...});
```

The use of braces in this context builds the initializer list required by the `.at()` method, allowing the code to take full advantage of the engine's existing bounds-checked indexing mechanism. The result is a seamless bridge between natural syntax and safe internal logic.

Const-Correctness and Overload Symmetry. There are two overloads of the operator:
(I) Mutable version - allows modifying the matrix entry at a given index.

```
T& operator()(Args... args);
```

(II) Const-qualified version - Enables access on const qualified instances without allowing mutation.

```
const T& operator()(Args.. args) const;
```

This separation respects C++ const-correctness rules and ensures that the NDMengine matrix behaves intuitively across both mutable and immutable contexts. It also aligns with thd design philosophy seen in the `.at()` function family, reinforcing uniformity across access patterns.

Practical Example - Here is a practical example using `operator()` to access and modify elements in a 3x3 matrix:

```
NDMatrix<int> mat({3, 3}, 0);   // 3x3 matrix initialized with zeros
mat(1, 2) = 42;                 // Set the value at row 1, column 2
int val = mat(1, 2);            // retrieve the same value
std::cout << val << std::endl;  // Outputs: 42
```

This usage is ideal for interactive work, high-performance loops, or algorithmic indexing where expressiveness and performance must coexist.

3.6 Reshaping and Iterator Interface

NDMengine includes powerful utilities for transforming and traversing matrix data while maintaining the core guarantees of memory layout and performance. This section introduces two key facilities: the `reshape()` function, which allows dimensional reinterpretation of matrix data without physical reallocation, and the iterator interface, which enables seamless integration with STL algorithms and idiomatic C++ patterns such as range-based `for` loops.

`reshape()`: **Changing Dimensional Views Without Data Movement**. The `reshape(const std::vector<size_t>& new_shape)` method allows users to reinterpret the existing data buffer under a new shape, provided that the total number of elements remains unchanged. This is analogous to reshaping functionality in other numerical computing libraries, such as NumPy's `.reshape()`.

Internally, `reshape()` computes the new total number of elements implied by the `new_shape` argument using `std::accumulate` with `std::multiplies<>`. It then verifies that this total matches the current number of elements stored in the `data` vector.

```
size_t new_total = std::accumulate(new_shape.begin(), new_shape.end(), size_t(1), std::mu
if (new_total != data.size()) {
    throw std::invalid_argument("([NDMatrix Error - std::invalid_argument])::  reshape():
}
```

This validation is essential. Since the `reshape()` function does not reallocate or resize the underlying buffer, the operation is only semantically meaningful if the new shape is consistent with the existing memory layout. If the shapes mismatch, the function throws a clear `std::invalid_argument` exception with a descriptive message to aid in debugging.

If the shape is valid, the engine updates the internal `shape` vector and recomputes the `strides` to reflect the new layout:

```
shape = new_shape;
compute_strides();
```

This approach makes reshaping an $\mathbf{O(1)}$ operation with no memory allocation, ideal for high-performance workflows where tensors must be reused across operations with different structural interpretations.

```
NDMatrix<float> tensor({2, 3, 4});
// Total size = 24.
tensor.reshape({4, 2, 3}); // Reinterprets the same data buffer as a 4x2x3 tensor
```

Iterator and Reverse Iterator Support. To maximize interoperability with the Standard Template Library (STL) and range-based syntax, NDMengine exposes a complete suit of forward and reverse iterators on the `data` container. These iterators are thin wrappers around the underlying `std::vector<T>` and follow conventional C++ idioms.

The following functions expose standard forward iterators:

```
auto begin()       -> data.begin()
auto end()         -> data.end()
auto begin() const -> data.begin()
auto end()   const -> data.end()
```

These allow direct integration with STL algorithms and idiomatic loops:

```
NDMatrix<int> mat({2, 2}, 1);
std::for_each(mat.begin(), mat.end(), [](int& x) { x *= 2; });
```

Constant Iterators

```
auto cbegin() const -> data.cbegin()
auto cend()   const -> data.cend()
```

These enable read-only traversal and support const-qualified matrix instances.
Reverse Iterators. NDMengine also supports reverse traversal using:

```
auto rbegin()       -> data.rbegin()
auto rend()         -> data.rend()
auto rbegin() const -> data.rbegin()
auto rend()   const -> data.rend()
```

And their const-qualified counterparts:

```
auto crbegin() const -> data.crbegin()
auto crend()   const -> data.crend()
```

These iterators are essential when operations need to be applied in reverse order or when implementing reverse algorithms efficiently.

3.7 Submatrix Slicing via `view()`

To enable selective access to subregions of high-dimensional matrices, NDMengine introduces the `view()` method - a slicing utility that extracts a contiguous N-dimensional submatrix from the current matrix and returns it as a newly constructed `NDMatrix<T>`. This version of slicing performs a deep copy of the relevant entries, ensuring data isolation between the original and the result.

Slicing is a core feature in tensor-based computation, widely used in machine learning, physics simulations, and financial modeling. The `view()` method provides a foundational implementation of this concept by supporting rectangular submatrix extraction via two vectors: **(I)** `start`: the coordinate (inclusive) where the slice begins in each dimension, and **(II)** `extent`: the shape of the submatrix (i.e., how many elements to extract along each dimension).

For example:

```
matrix.view({1, 0}, {2, 2});
```

returns a `2x2` submatrix at row 1, column 0.

Safey: Dimensional and Boundary Checks - Before any slicing occurs, `view()` performs rigorous validation of inputs to ensure correctness and prevent undefined behavior.

(I). Dimension count consistency: Both `start` and `extent` must match the number of dimensions in the original matrix:

```
if (start.size() != shape.size() || extent.size() != shape.size()) {
    throw std::invalid_argument("([NDMatrix Error - std::invalid_argument]):: view(): sta
}
```

(II). Boundary protection: The function verifies the slice does not exceed matrix bounds in any dimension:

```
for (size_t i = 0; i < shape.size(); ++i) {
    if (start[i] + extent[i] > shape[i]) {
        throw std::out_of_range("([NDMatrix Error - std::out_of_range]):: view(): requested s
    }
}
```

These checks catch errors early and precisely.

After validation, a new `NDMatrix<T>` is constructed to hold the sliced data, with its shape set to the user-provided `extent`.

```
NDMatrix<T> result(extent);
```

A recursive lambda named `copy` is then defined using `std::function`, which traverses the N-dimensional index space of the submatrix. For each coordinate in the target space, it: **(I)** Computes the corresponding index in the source matrix by offsetting from `start`, **(II)** Copies the element from the original matrix to the result matrix using multi-dimensional indexing.

The recursion builds dimension by dimension. Once the full index is constructed (`dim == shape.size()`), it performs the actual element assignment:

```
result.at({idx_new.begin(), idx_new.end()}) = at({idx_this.begin(), idx_this.end()});
```

This approach is both flexible and dimension-agnostic, enabling the same function to handle slicing across arbitrary N-dimensions. The recursive call kicks off with:

```
copy(0, start, std::vector<size_t>(shape.size(), 0));
```

where `idx_this` and `idx_new` track positions in the source and target matrices, respectively. This version of `view()` performs a deep copy. Future extensions could support shallow views or reference-based slicing for performance-critical applications. Recursive traversal avoids hard-coded loops or template metaprogramming, providing simplicity and generality at the cost of some overhead - acceptable for a first implementation. Exception messages are crafted to clearly indicate misalignment or overreach in slicing requests, aiding debugging in larger systems.

Practical Example:

```
NDMatrix<float> tensor({4, 4}, 1.0);

// Fill a 4x4 matrix with a gradient
int count = 0;
for (size_t i = 0; i < 4; ++i)
    for (size_t j = 0; j < 4; ++j)
        tensor.at({i, j}) = count++;

// Extract a 2x2 submatrix from position (1,1)
NDMatrix<float> sub = tensor.view({1, 1}, {2, 2});
```

This operation returns a 2x2 matrix containing the values from the region defined by rows 1–2 and columns 1–2 in the original matrix.

3.8 Elementwise Arithmetic Operations

NDMengine supports elementwise arithmetic between tensors of equal shape via operator overloading. This allows users to express numerical computations in a concise and readable manner, mirroring the mathematical form of tensor algebra while maintaining full type safety and memory control. The supported operators are:

- `operator+`: elementwise addition.
- `operator-`: elementwise subtraction.
- `operator*`: elementwise multiplication.
- `operator/`: elementwise divison.

Each operator returns a new `NDMatrix<T>` containing the result of applying the operation element-by-element across two operand tensors.

Each arithmetic operator overload begins by checking whether the operand tensors (`*this` and `other`) share the exact same shape. This strict shape equality requirement enforces the invariant that elementwise operations must operate on structurally identical tensors.

```
if (shape != other.shape()) {
    throw std::invalid_argument("([NDMatrix Error - std::invalid_argument]):: operator+:
}
```

This early check ensures mathematical consistency and prevents hard-to-diagnose runtime errors caused by mismatched dimensions. The exception messages are descriptive and tagged with `NDMatrix Error`, aiding debugging in complex numeric workflows.

If the shapes are compatible, a new matrix named `result` is allocated with the same shape. Then, the operation is applied across the flattened `data` buffer using a simple for-loop over the linear index space:

```
NDMatrix result(shape);
for (size_t i = 0; i < data.size(); ++i){
    result.data[i] = data[i] <op> other.data[i];
}
```

where `<op>` is `+`, `-`, `*`, or `/` depending on the operator overload. This approach avoids unnecessary abstractions and gives full control over iteration, aligning with the engine's performance and predictability goals. Each operation is performed in-place on the result buffer, while preserving the immutability of the source operands.

Operator Details.

`operator+` - Performs component-wise addition of corresponding elements from the two matrices:

```
result[i] = data[i] + other.data[i];
```

`operator-` - Computes elementwise difference:

```
 result[i] = data[i] - other.data[i];
```

`operaor*` - Applies pointwise multiplication

```
 result[i] = data[i] * other.data[i];
```

`operator/` - Performs elementwise division

```
 result[i] = data[i] / other.data[i];
```

Example Usage:

```
NDMatrix<double> A({2, 2}, 1.0);
NDMatrix<double> B({2, 2}, 2.0);

NDMatrix<double> C = A + B;  // {3.0, 3.0, 3.0, 3.0}
NDMatrix<double> D = B - A;  // {1.0, 1.0, 1.0, 1.0}
NDMatrix<double> E = A * B;  // {2.0, 2.0, 2.0, 2.0}
NDMatrix<double> F = B / A;  // {2.0, 2.0, 2.0, 2.0}
```

Each operation returns a new matrix without mutating the inputs, enabling safe composition and chaining of arithmetic expressions.

3.9 Scalar Arithmetic Operators (0-D) Broadcasting

Scalar operators enable fast, elementwise arithmetic between an `NDMatrix<T>` and a single scalar value of the same `T`. Conceptually, the scalar is treated as 0-dimensional tensor broadcast across every element. These operators are provided as const member functions that do not mutate the source matrix; each returns a freshly allocated `NDMatrix<T>` with the same shape.

Signatures (member functions; scalar on the right-hand side):

```
NDMatrix operator+(const T& scalar) const;
NDMatrix operator-(const T& scalar) const;
NDMatrix operator*(const T& scalar) const;
NDMatrix operator/(const T& scalar) const;
```

The behavior and guarantees of scalar operators are as follows: the source matrix (*this) is never modified; instead, the operation produces and returns a new matrix. The result always preserves the original shape, with strides recomputed to match. Each scalar operation runs in linear time relative to the number of elements, with an additional linear amount of storage allocated for the new result, so the complexity is $\Theta(N)$ where N equals the size of the data vector.

The element type `T` must support the corresponding arithmetic operator (`+`, `-`,`*`, or `/`). When the type is an integer, the division operator follows integer division semantics. Division by zero follows the rule of the type itself: it is undefined for built-in integer types by may throw or behave differently from custom numeric classes, so callers should guard against this case when necessary.

In terms of semantics, the scalar is conceptually broadcoast across the matrix as a zero-dimensional tensor. This means that the same scalar value is applied to each element in the row-major memory order. Because the operators return new matrices, results are chainable in expressions such as `(A * 1.5) + 0.01`, which combine multiple scalar transformations cleanly in a single line.

From an implementation perspective, a fresh result matrix is first conducted with the same shape as the source. Then a single flat loop iterates over all indices from zero to the size of the data array, applying the arithmetic operator to each element: `result.data[i] = data[i] <op> scalar`. All scalar operator functions are declared `const`, which means they can also be invoked on cost-qualified `NDMatrix<T>` instances.

These members overloads handle the form `matrix <op> scalar`. If symmetric behavior such as `scalar <op> matrix` (for example, `2.0 * M`) is desired, non-member overloads can be provided that forward to the existing member versions, although such extensions are not required for the current implementation and are noted here only for completeness.

Pseudocode: Scalar Operators

```
// Add scalar to every element, returning a new matrix
function addScalar(matrix, scalar):
    result ← NDMatrix(matrix.shape)
    for i from 0 to matrix.data.size - 1:
```

```
        result.data[i] ← matrix.data[i] + scalar
    return result

// Subtract scalar from every element
function subScalar(matrix, scalar):
    result ← NDMatrix(matrix.shape)
    for i from 0 to matrix.data.size - 1:
        result.data[i] ← matrix.data[i] - scalar
    return result

// Multiply every element by scalar
function mulScalar(matrix, scalar):
    result ← NDMatrix(matrix.shape)
    for i from 0 to matrix.data.size - 1:
        result.data[i] ← matrix.data[i] * scalar
    return result

// Divide every element by scalar
function divScalar(matrix, scalar):
    // Caller should ensure scalar != 0 (or that T's division is defined)
    result ← NDMatrix(matrix.shape)
    for i from 0 to matrix.data.size - 1:
        result.data[i] ← matrix.data[i] / scalar
    return result
```

Example: Financial Modeling - Credit Risk

Scenario: You maintain a borrower-by-feature matrix `X` ($N \times F$). Before fitting or scoring a logistic PD model, you standardize features by subtracting a global mean and dividing by a global standard deviation. Later, you stress the resulting PD logits by adding a macro shock and compute ECL using constant portfolio-level LGD and EAD scalars.

```
#include "NDMengine.hpp"
#include <cmath>
#include <vector>
#include <iostream>

int main() {
    // Toy dataset: 2 borrowers × 3 features
    NDMatrix<double> X({2, 3});
    X(0,0) = 45'000;  X(0,1) = 0.85;  X(0,2) = 3;   // income, LTV, credit history
    X(1,0) = 60'000;  X(1,1) = 0.70;  X(1,2) = 8;

    // Scalar standardization (global mean/std across all features, for illustration) ---
    double mu  = 20'000.0;     // global mean (toy)
```

```
    double sig = 20'000.0;     // global std  (toy, non-zero)

    NDMatrix<double> X_std = (X - mu) / sig;   // elementwise: (X_ij - mu) / sig

    // Logistic PD scoring for each borrower (simple hand-rolled dot product) ---
    // weights correspond to features [income, LTV, credit history]
    double b0 = -3.5;                   // intercept
    std::vector<double> w = {-0.5, 2.5, -0.2};

    NDMatrix<double> logit({2}, 0.0);   // one logit per borrower
    for (size_t i = 0; i < 2; ++i) {
        double s = b0;
        for (size_t j = 0; j < 3; ++j) {
            s += w[j] * X_std(i, j);
        }
        logit(i) = s;
    }

     // Macro stress via scalar shift on logits (0-D broadcast) ---
    double shock = 0.35;                // macro shock added to all borrowers
    NDMatrix<double> logit_stressed = logit + shock;

    // Convert logits to PDs
    NDMatrix<double> PD({2}, 0.0);
    for (size_t i = 0; i < PD.size(); ++i) {
        double z = logit_stressed[i];
        PD[i] = 1.0 / (1.0 + std::exp(-z));
    }

    // ECL calculation with scalar LGD and EAD ---
    double LGD = 0.45;                  // constant LGD
    double EAD = 100'000.0;             // constant EAD
    NDMatrix<double> ECL = PD * LGD * EAD;   // chained scalar multiplies

    for (size_t i = 0; i < ECL.size(); ++i) {
        std::cout << "Borrower " << i << " PD=" << PD[i]
                  << "  ECL=" << ECL[i] << "\n";
    }
    return 0;
}
```

If you decide to run this code yourself, the result should be:

```
Borrower 0 PD=0.00229454  ECL=103.255
Borrower 1 PD=0.00157804  ECL=71.0118
Program ended with exit code: 0
```

Why scalar ops matter here. Scalar operators play an important role in practical modeling. During preprocessing, expressions such as $\frac{X-\mu}{\sigma}$ standardize all features with only two scalar passes, without introducing additional loops beyond the flat iteration already inside the operators. In scenario analysis, a single macroeconomic shock applied to all logits can be expressed as a scalar addition `logit + shock`, which is exactly the kind of zero-dimensional broadcast these operators were designed to handle. Finally, in loss estimation, the expected credit loss can be written as `PD * LGD * EAD`, where the scalars are multiplied across the entire probability-of-default vector to produce an ECL per borrower. This changing of scalar multiplication works directly on the data without mutating intermediate arrays, keeping the workflow clean and efficient.

3.10 Dimensional Reduction via `slice()`

The `slice()` method provides a convenient way to reduce an N-dimensional tensor into an $(N-1)$-dimensional submatrix by fixing one axis at a specified index. Conceptually, this operation selects a single "slice" of the tensor orthogonal to the chosen axis. For example, given a 3D tensor with shape $\{3, 4, 5\}$, calling slice `slice(0,2)` fixes axis 0 at index 2 and returns a 2D matrix of shape $\{4, 5\}$, corresponding to the third "layer" along that axis.

Internally, the function validates the request before any data is copied. First, it checks that the requested axis is within the valid dimensional range. Second, it ensures that the index does not exceed the size of the selected axis. Violations trigger a `std::out_of_range` exception with a descriptive error message.

Once validated, the function constructs a new shape vector by copying all dimensions except the fixed axis. A new `NDMatrix` result object is then initialized with this reduced shape. To populate its data, the implementation uses a recursive lambda (`copy`) that traverses all dimensions of the target submatrix. Whenever the traversal reaches the fixed axis, the index is seto to the user-provided value. For all other dimensions,the recursion iterates fully, copying elements from the source into the result using safe multi-dimensional access through at `at()`.

This design provides flexibility across arbitrary dimensions without resorting to hardcoded loops or heavy template metaprogramming. The recursion keeps the implementation concise, dimension-agnostic, and easy to extend. Because the returned matrix is a deep copy, the slice is independent of the original tensor and modifications do not propagate back.

Pseudocode: `slice(axis, index)`

```
function slice(matrix, axis, index):
    if axis >= matrix.rank:
        throw out_of_range("axis is invalid")
    if index >= matrix.shape[axis]:
        throw out_of_range("index exceeds size of axis")

    // Build new shape (omit fixed axis)
    result_shape = []
    for d in 0 .. matrix.rank-1:
        if d != axis:
            result_shape.push(matrix.shape[d])

    result = NDMatrix(result_shape)

    // Recursive copy
    function copy(dim, src_index, dst_index):
        if dim == matrix.rank:
            result.at(dst_index) = matrix.at(src_index)
            return
        if dim == axis:
```

```
            src_index[dim] = index
            copy(dim+1, src_index, dst_index)
        else:
            dst_dim = (dim < axis) ? dim : dim - 1
            for i in 0 .. matrix.shape[dim]-1:
                src_index[dim] = i
                dst_index[dst_dim] = i
                copy(dim+1, src_index, dst_index)

    copy(0, [0]*matrix.rank, [0]*(matrix.rank-1))
    return result
```

Example: Financial Modeling - Credit Portfolio Analyst

In credit modeling, slicing is often required to extract a particular cross-section of a borrower-feature tensor. Suppose we store borrower data as a 3D tensor of shape $\{N, T, F\}$, where N is the number of borrowers, T is the number of time periods, and F is the number of features (income, LTV, credit history, etc.).

```
#include <iostream>
#include "NDMengine.hpp"

int main() {
    // Toy dataset: 2 borrowers × 3 time periods × 2 features
    NDMatrix<double> borrowers({2, 3, 2});
    double val = 1.0;
    for (size_t i = 0; i < 2; ++i)
        for (size_t t = 0; t < 3; ++t)
            for (size_t f = 0; f < 2; ++f)
                borrowers.at({i, t, f}) = val++;

    // Slice out borrower 1 across all periods and features
    NDMatrix<double> borrower1 = borrowers.slice(0, 1);

    std::cout << "Borrower 1 data (shape {3,2}):\n";
    for (size_t t = 0; t < 3; ++t) {
        for (size_t f = 0; f < 2; ++f)
            std::cout << borrower1.at({t, f}) << " ";
        std::cout << "\n";
    }

    return 0;
}
```

If you decide to run the code yourself, this should be the output:

```
Borrower 1 data (shape {3,2}):
7 8
9 10
11 12
```

In this example, `slice(0,1)` fixes the borrower axis (axis 0) at index 1, yielding a 2D submatrix of shape $\{3, 2\}$. This allows analysts to isolate the time-series features of a single borrower without manually indexing through the full 3D structure. In practice, this might be used to run a borrower-level probability-of-default trajectory, or to stress-test one obligor's features across time while leaving the rest of the dataset intact.

3.11 Broadcasting Shape Inference

`broadcast_shape(cost std::vector<size_t& other_shape>) const` computes the shape that results from broadcasting this tensor against another tensor. It follows NumPy-style trailing-axis rules: compare dimensions from the end; along each axis the sizes must be equal or one of them must be 1. Missing leading axes are treated as size 1 (so $0-D$ scalars and lower-rank tensors broadcast naturally). If any axis violates the rule, the function throws `std::invalid_argument` with a descriptive tag.

Let `A.shape = shape` (rank `n`) and `B.shape = other_shape` (rank `m`). Walk `i = 0..max(n,m)-1` from the last axis backward. Define

- `dim_a = (i < n) ?  shape[n-1-i] :  1`
- `dim_b = (i < m) ?  other_shape[m-1-i] :  1`

The resulting axis is

- `result[max_dim-1-i] = max(dim_a, dim_b)` if `dim_a == dim_b || dim_a == 1 || dim_b == 1`,
- otherwise throw (not broadcast-compatible).

Pseudocode:

```
function broadcast_shape(self_shape, other_shape):
    n ← length(self_shape)
    m ← length(other_shape)
    k ← max(n, m)
    result ← vector<size_t>(k)

    for i from 0 to k-1:
        a ← (i < n) ? self_shape[n-1-i]  : 1
        b ← (i < m) ? other_shape[m-1-i] : 1

        if a != b and a != 1 and b != 1:
            throw invalid_argument(
              "([NDMengine Error - std::invalid_argument]):: broadcast_shape(): shapes are no
            )

        result[k-1-i] ← max(a, b)

    return result
```

Complexity. `O(max(n,m))` time, `O(max(0,m))` extra space. No data copies - this is pure shape math.

We do have some exceptions:

- `std::invalid_argument` when any axis pair violates the rule (neither equal nor 1).
- Note: dimensions may be `0` (empty axes). Broadcasting treats `0` like any other size; the product of the result may be `0`, yielding an empty tensor - this is allowed and useful for degenerate slices.

A.shape	**B.shape**	**Result**	**Notes**
`{5, 3}`	`{1, 3}`	`{5, 3}`	Tail dims match (`3==3`); leading 5 vs implicit 1 → 5
`{4, 1, 7}`	`{3, 7}`	`{4, 3, 7}`	Compare from end: `7==7`, `1` vs `3` → `3`, leading `4` vs implicit `1` → `4`
`{6, 1, 1}`	`{1, 5}`	`{6, 1, 5}`	Middle remains 1, last expands to 5
`{8}`	`{1, 8}`	`{1, 8}`	Scalar-like vector against 2-D row
`{2, 3}`	`{2}`	**error**	Compare last: `3` vs `2` ⇒ incompatible (none is `1`)
`{}` (scalar)	`{10, 4}`	`{10, 4}`	0-D broadcasts to any shape
`{0, 7}`	`{1, 7}`	`{0, 7}`	Legal; total size 0 (empty view)

Table 1: Worked examples for `broadcast_shape` (NumPy-style trailing-axis rules).

In NDMengine, a scalar can be represented as rank-0 (`shape.size()==0`) or rank-k tensor with all ones. `broadcast_shape` treats missing axes as 1, so both conventions work for 0-D broadcasting. This is used in elemenwise binary operations with broadcasting (e.g., matrix vs row-vector; tensor vs scalar), preflight checks for used kernels (map/reduce) and preparing output buffers for lazily evaluated expressions (future work). The result is unique for given inputs; order of operands does not matter for the shape (commutative), though you'll still need correct index mapping when executing the operation.

3.12 Index Conversion Utility Tools

Every tensor operation in NDMengine ultimately reduces to addressing the correct slot in a 1D contiguous buffer. To achieve this consistently, the engine provides a set of index conversion utilities that map between multi-dimensional coordinates and flat indices. These tools are the backbone for advanced features such as slicing, broadcasting-aware iteration, and reduction operations.

Already introduced in the internal structure section, `compute_flat_index` takes an N-dimensional coordinate (either as `std::vector<size_t>` or `std::initializer_list<size_t>`) and returns the corresponding flat index in the data buffer. it performs: (*i*) Dimensionality check where the number of provided indices must match the rank, (*ii*) Bounds check where each index must be `< shape[d]` and (*iii*) Row-major accumulation where it multiplies each index by its stride and sums.

Pseudo code:

```
function compute_flat_index(indices):
    if indices.size() != shape.size():
        throw invalid_argument

    flat_index ← 0
    for d from 0 to shape.size()-1:
        if indices[d] >= shape[d]:
            throw out_of_range
        flat_index ← flat_index + indices[d] * strides[d]

    return flat_index
```

This ensures safety and consistency, and is reused internally by all accessors.

3.12.1 `unravel_index`

The inverse of flattening: given a flat index and a shape, return its multi-dimensional coordinate. This is essential for traversals, iterators, and mapping broadcasted outputs back to their sources.

Psuedo code:

```
function unravel_index(flat_index, shape, strides):
    indices ← vector(size=shape.size())

    for d from 0 to shape.size()-1:
        indices[d] ← (flat_index / strides[d]) % shape[d]

    return indices
```

3.12.2 `ravel_multi_index`

A specialized forward utility, similar to `compute_flat_index`, but explicitly accepts a multi-index and shape, and returns a flat index. Used for validating external coordinates against a given tensor's shape.

Pseudo code:

```
function ravel_multi_index(indices, shape):
    if indices.size() != shape.size():
        throw invalid_argument

    flat ← 0
    stride ← 1
    for d from shape.size()-1 downto 0:
        if indices[d] >= shape[d]:
            throw out_of_range
        flat += indices[d] * stride
        stride *= shape[d]

    return flat
```

Example:

```
NDMatrix<int> mat({3, 4}, 0); // shape (3x4)
auto flat = mat.compute_flat_index({2, 1}); // returns 2*4 + 1 = 9

// Now invert:
auto indices = mat.unravel_index(9); // returns {2, 1}
```

This roundtrip demonstrates that `compute_flat_index` and `unravel_index` form a consistent bijection for valid coordinates.

Consider a 2D matrix representing loan exposures: rows = borrowers, columns = scenarios.

```
NDMatrix<double> exposures({3, 5}); // 3 borrowers, 5 scenarios
exposures.at({0,0}) = 1000; // Borrower 0, Scenario 0
exposures.at({2,4}) = 5000; // Borrower 2, Scenario 4

// Access via flat index:
size_t flat = exposures.compute_flat_index({2,4}); // 14
double value = exposures[flat]; // same as exposures.at({2,4})

// Recover the indices for auditing:
std::vector<size_t> coord = exposures.unravel_index(flat); // {2,4}
```

In credit risk analysis, auditors often need to map flattened batch calculations back to specific borrowers/scenarios pairs. These utilities guarantee traceability between low-level iteration and domain-level coordinates.