CAMEL

Manual for Version 0.0

March 26, 2024

Contents

Part I
Introduction and Setup

by Sergio Madrid

1 Introduction

Welcome to the documentation for CAMEL, a high-performance C library for symbolic and numerical computation. This manual provides a comprehensive reference for the library, including installation instructions, usage examples, and performance analysis.

You might now be wondering "What is CAMEL?" and "Why should I use it?" Let's start by answering these questions. CAMEL aims to be a powerful and efficient library for symbolic and numerical computation, as well as a versatile tool for a wide range of applications, including scientific computing, data analysis, and machine learning. The library places a focus on performance and ease of use and is designed to be a versatile tool for a wide range of applications, including scientific computing, data structure manipulation, and machine learning. The library is written in C, which makes it easy to integrate with other languages and systems, and it is designed to be efficient and scalable, making it suitable for both small-scale and large-scale computations.

This manual aims not only to provide a comprehensive reference for the usage of the library, but also a detailed and rigurous mathematical background for the algorithms and data structures implemented in the library, so that the reader can understand the inner workings of the library, and have a theoretical background before using the library or having a look at the source code. This manual assumes that the reader has a basic understanding of mathematics and computer science, and is familiar with the C programming language. At the points where the manual assumes a deeper understanding of a topic, it will provide references to external sources where the reader can learn more about it.

1.1 Design Goals

CAMEL first appeared as a personal project of mine (Sergio Madrid) to expand my mathematical knowledge by implementing a wide range of mathematical algorithms and data structures, but over time, the scope of the library has expanded until what it is now, but since I began making it, most of the main goals of the library have remained the same.

The first main goal of mine was to create an understandable and intuitive API that would give easy access to the library's features, and a low entry point for anyone interested in the topics covered by the library. I also wished to provide easy to read an understand source code, so that anyone interested in the implementation of the algorithms, whether a beginner or a seasoned veteran, could easily understand them, and that way feel encouraged to keep digging deeper instead of ending up discouraged. Finally, I aimed to provide a high-performance library, with a focus on performance and efficiency, so that the library could be used in a wide range of applications, from small-scale computations or academic demonstration, to large-scale computing, data analysis or machine learning.

Finally, I aimed to provide full symbolic and numerical computation capabilities throughout all modules of the library, based all on the same core for easy interoperation.

1.2 Main Features

The library is divided into several modules, each of which provides a set of related functions and data structures. These modules are explained thoroughly in Part II, and this section will serve as a brief overview of the main features of the library. The modules are as follows:

1. **Core**: The core module provides the building blocks for the entire library, including basic types, error handling, memory management, data structures, expression parsing, and other core tools.

2. **Algebra**: The algebra module includes linear algebra, abstract algebra, polynomial manipulation, and other algebra tools.
3. **Calculus**: The calculus module provides from simple functions, to complex calculus tools such as signal processing, differential equations, and other tools.
4. **Geometry**: The geometry module offers general geometric tools that range from simple geometric operations to complex geometric algorithms.
5. **Number Theory**: The nujber theory module deals with everything related to integers, including prime numbers, sets, or other number theory tools.
6. **Statistics**: The statistics module is filled with statistical tools, including probability distributions, hypothesis testing, and other statistical tools.

1.3 License

CAMEL is licensed under the MIT License, which is a permissive open-source license that allows you to use the library for any purpose, including commercial applications, as long as you include the original copyright notice.

2 Installation

This section provides detailed instructions for installing CAMEL on your system. The library is designed to be easy to install and use, and it is compatible with a wide range of systems and compilers. The following sections provide step-by-step instructions for installing the library on various platforms, including Windows, macOS, and Linux.

2.1 Prerequisites

Before installing CAMEL, you will need to have the following software installed on your system:

- **C Compiler**: You will need a C compiler to build the library. The library is compatible with a wide range of compilers, including GCC, Clang, and MSVC.
- **Make**: You will need the `make` utility to build the library. The library includes a Makefile that automates the build process, making it easy to compile the library on a wide range of systems.
- **Git**: You will optionally need Git to clone the library's source code from the repository. Git is a version control system that is widely used for open-source projects, and it is available for all major operating systems.

2.2 Installation Steps

You can install CAMEL in one of two ways: by downloading the source code from the repository or by downloading a precompiled binary. The following sections provide detailed instructions for each method.

2.2.1 Downloading the Source Code

To download the source code from the repository, you will need to have Git installed on your system. Once you have Git installed, you can clone the repository by running the following command in your terminal:

```
git clone https://github.com/srmadrid/camel.git
```

This will create a new directory called `camel` in your current working directory, which contains the source code for the library. Navigate to the `camel` directory and run the following command to build the library:

2.3 Verifying the Installation

To verify that the library has been installed correctly, create a directory `bin/os` in `camel/test`, where `os` is the name of your operating system.Then you can run the following command in your terminal to build the test suite:

```
make test
```

Once it has compiled, you can run the test suite executable, which will be located in the created directory, to verify that the library is working correctly.

3 Getting Started

3.1 Hello World with Camel

3.2 Basic Concepts

Part II
Library Reference

by Sergio Madrid

4 Notation

This manual includes complex mathematical explanations and algorithms, and to make it easier to understand, we will use a set of notation and conventions that will be used throughout the manual:

- **Sets**: Sets will be denoted with capital letters, such as A, B, C, etc.
- **Vectors**: Vectors will be denoted with lowercase bold letters, such as $\mathbf{v}$, $\mathbf{w}$, $\mathbf{x}$, etc. And their components will be denoted with subscripts, such as v_1, v_2, v_3, etc.
- **Matrices**: Matrices will be denoted with uppercase bold letters, such as $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, etc. And their components will be denoted with subscripts, A_{ij}, where i is the row and j is the column.
- **Scalars**: Scalars will be denoted with lowercase letters, such as a, b, c, etc.

5 Naming Conventions

The library uses a set of naming conventions to make the code more readable and to provide a consistent API. The following sections provide an overview of the naming conventions used in the library.

6 Core

The **core** module provides the building blocks for the entire library, including basic types, error handling, memory management, data structures, expression parsing, and other core tools. The following sections provide an overview of the core module, as well as detailed descriptions of its components.

6.1 Basic Features

6.1.1 Elementary Typedefs

For ease of usage, CAMEL provides a set of elementary typedefs for basic types:

- `u8`: An unsigned 8-bit integer.
- `u16`: An unsigned 16-bit integer.
- `u32`: An unsigned 32-bit integer.
- `u64`: An unsigned 64-bit integer.
- `i8`: A signed 8-bit integer.
- `i16`: A signed 16-bit integer.
- `i32`: A signed 32-bit integer.
- `i64`: A signed 64-bit integer.
- `f32`: A 32-bit floating-point number.

- `f64`: A 64-bit floating-point number.
- `b8`: A boolean value.
- `b32`: A 32-bit boolean value.
- `cf32`: A 32-bit complex floating-point number.
- `cf64`: A 64-bit complex floating-point number.

6.1.2 Error Handling

6.1.3 Constants

6.1.4 Types

CAMEL also offers some helper functionalities for types which are used throughout the library. The most important of which is the `CML_NumericType` enum, which is used to represent numeric types:

`CML_NumericType`

Values:
- `CML_U8`: An unsigned 8-bit integer.
- `CML_U16`: An unsigned 16-bit integer.
- `CML_U32`: An unsigned 32-bit integer.
- `CML_U64`: An unsigned 64-bit integer.
- `CML_I8`: A signed 8-bit integer.
- `CML_I16`: A signed 16-bit integer.
- `CML_I32`: A signed 32-bit integer.
- `CML_I64`: A signed 64-bit integer.
- `CML_F32`: A 32-bit floating-point number.
- `CML_F64`: A 64-bit floating-point number.
- `CML_COMPLEXF32`: A 32-bit complex floating-point number.
- `CML_COMPLEXF64`: A 64-bit complex floating-point number.
- `CML_BIGINT`: An arbitrary-precision integer (corresponds to `CML_BigInt`).
- `CML_FRACTION`: An arbitrary-precision fraction (corresponds to `CML_Fraction`).
- `CML_COMPLEX`: An arbitrary-precision complex number (corresponds to `CML_Complex`).
- `CML_EXPRESSION`: A symbolic expression (corresponds to `CML_Expression`).
- `CML_MATRIX`: A matrix (corresponds to `CML_Matrix`).

Description:
Represents a numeric type, which can be used to specify the type of a variable or function parameter.

At a first glance, it might seem strange that the `CML_NumericType` enum includes the non-numeric `CML_Expression` type, but this is because the same operations that can be performed on a numeric type can be performed on an expression, and, as will be seen later, this allows for matrices to hold expressions which, in some areas, can be very useful. `CML_Matrix` is also included in the enum, as it is the essential numeric type for the library, and this allows for higher dimensional matrices to be created.

Additionally, a helper function is included to get the size of a numeric type in bytes:

`cml_numerictype_size`

Parameters:
- `CML_NumericType type`: The numeric type.

Returns:
- `u32`

Description:
Returns the size of the numeric type in bytes.

Valid Usage:
• `type` **must** be a valid `CML_NumericType` value.

6.2 Memory Management

6.3 Data Structures

The **data structures** module provides a set of common data structures, including arrays, lists, queues, stacks, and trees, as well as some more esoteric ones. The following sections provide an overview of the data structures provided by the library, as well as detailed descriptions of their usage and, in cases where it is necessary, their algorithms thoroughly explained.

6.3.1 String

The **string** data structure serves as a replacement for the standard C string, and provides a wide range of string manipulation functions. The string data structure is defined as follows:

`CML_String`

Fields:
• `char *data`: A pointer to the string's data.
• `u32 length`: The length of the string.
• `u32 capacity`: The allocated capacity of the string.
• `i32 refCount`: Remaining allowed references to the string.
• `CML_Allocator *allocator`: Allocator used for dynamic memory allocation within the string.

Description:
Represents a string data structure containing information about the string's data, length, capacity, and allocator. When `refCount` is -1, the string has infinite references.

Valid Usage:
• `allocator` **must** be a valid pointer to a `CML_Allocator` structure.

`cml_string_init`

Parameters:
• `CML_Allocator *allocator`: Pointer to the allocator to use in the string.
• `const char *input`: The initial data for the string.
• `CML_String *string`: The string to initialize.
Returns:
• `CML_Status`

Description:
Initializes a new string with the input string.

Valid Usage:
• `allocator` **must** be a valid pointer to a `CML_Allocator` structure.
• `input` **must** be a valid pointer to a null-terminated string.
• `string` **must** be a valid pointer to a `CML_String` structure.

Return Codes:
• `CML_SUCCESS`: The string was successfully initialized.
• `CML_ERR_NULL_PTR`: `allocator`, `input`, or `string` was `NULL`.
• `CML_ERR_MALLOC`: An error occurred during memory allocation.

`cml_string_init0`

Parameters:
• `CML_Allocator *allocator`: Pointer to the allocator to use in the string.
• `CML_String *string`: The string to initialize.
Returns:

• `CML_Status`

Description:
Initializes a new string with all values set to 0, and `data` set to `NULL`.

Valid Usage:
• `allocator` **must** be a valid pointer to a `CML_Allocator` structure.
• `string` **must** be a valid pointer to a `CML_String` structure.

Return Codes:
• `CML_SUCCESS`: The string was successfully initialized.
• `CML_ERR_NULL_PTR`: `allocator` or `string` was `NULL`.

cml_string_destroy

Parameters:
• `void *string`: The string to destroy.
Returns:
• `void`

Description:
Destroys a string, freeing its internal memory.

Valid Usage:
• `string` **must** be a valid pointer to a `CML_String` structure.

cml_string_temp

Parameters:
• `CML_Allocator *allocator`: Pointer to the allocator to use in the string.
• `const char *input`: The data for the string.
Returns:
• `CML_String*`

Description:
Creates a temporary string (`refCount` = 1) with the input string. It is dynamically allocated using the provided allocator.

Valid Usage:
• `allocator` **must** be a valid pointer to a `CML_Allocator` structure.
• `input` **must** be a valid pointer to a null-terminated string.

Return Codes:
• `CML_String*`: A pointer to the newly created string.
• `NULL`: An error occurred during memory allocation.

cml_string_checkref

Parameters:
• `CML_String **string`: The string to check.
Returns:
• `void`

Description:
Checks if the string has any remaining references, and if not, destroys it.

Valid Usage:
• `string` **must** be a valid pointer to a pointer to a `CML_String` structure, and `*string` **must** be a valid pointer to a `CML_String` structure.

cml_string_copy

Parameters:
• `CML_String *input`: The source string.
• `CML_String *out`: The destination string.
Returns:
• `CML_Status`

Description:
Copies the input string into the out string.

Valid Usage:
• `input` **must** be a valid pointer to a `CML_String` structure.
• `out` **must** be a valid pointer to a `CML_String` structure. It need not be initialized.

Return Codes:
• `CML_SUCCESS`: The string was successfully copied.
• `CML_ERR_NULL_PTR`: `input` or `out` was `NULL`.
• `CML_ERR_MALLOC`: An error occurred during memory allocation.
• `CML_ERR_REALLOC`: An error occurred during memory reallocation.

cml_string_ncopy

Parameters:
• `CML_String *input`: The source string.
• `u32 n`: The number of characters to copy.
• `CML_String *out`: The destination string.
Returns:
• `CML_Status`

Description:
Copies the first `n` characters of the input string into the out string.

Valid Usage:
• `input` **must** be a valid pointer to a `CML_String` structure.
• `out` **must** be a valid pointer to a `CML_String` structure. It need not be initialized.

Return Codes:
• `CML_SUCCESS`: The string was successfully copied.
• `CML_ERR_NULL_PTR`: `input` or `out` was `NULL`.
• `CML_ERR_MALLOC`: An error occurred during memory allocation.
• `CML_ERR_REALLOC`: An error occurred during memory reallocation.

cml_string_copy_char

Parameters:
• `const char *input`: The source string.
• `CML_String *out`: The destination string.
Returns:
• `CML_Status`

Description:
Copies the input `char` string into the out string.

Valid Usage:
• `input` **must** be a valid pointer to a null-terminated string.
• `out` **must** be a valid pointer to a `CML_String` structure. It need not be initialized.

Return Codes:
• `CML_SUCCESS`: The string was successfully copied.
• `CML_ERR_NULL_PTR`: `input` or `out` was `NULL`.
• `CML_ERR_MALLOC`: An error occurred during memory allocation.
• `CML_ERR_REALLOC`: An error occurred during memory reallocation.

cml_string_ncopy_char

Parameters:
- `const char *input`: The source string.
- `u32 n`: The number of characters to copy.
- `CML_String *out`: The destination string.

Returns:
- `CML_Status`

Description:
Copies the first `n` characters of the input `char` string into the out string.

Valid Usage:
- `input` **must** be a valid pointer to a null-terminated string.
- `out` **must** be a valid pointer to a `CML_String` structure. It need not be initialized.

Return Codes:
- `CML_SUCCESS`: The string was successfully copied.
- `CML_ERR_NULL_PTR`: `input` or `out` was `NULL`.
- `CML_ERR_MALLOC`: An error occurred during memory allocation.
- `CML_ERR_REALLOC`: An error occurred during memory reallocation.

cml_string_cat

Parameters:
- `CML_String *input`: The source string.
- `CML_String *out`: The destination string.

Returns:
- `CML_Status`

Description:
Concatenates the input string into the out string.

Valid Usage:
- `input` **must** be a valid pointer to a `CML_String` structure.
- `out` **must** be a valid pointer to an initialized `CML_String` structure.

Return Codes:
- `CML_SUCCESS`: The string was successfully concatenated.
- `CML_ERR_NULL_PTR`: `input` or `out` was `NULL`.
- `CML_ERR_REALLOC`: An error occurred during memory reallocation.

cml_string_ncat

Parameters:
- `CML_String *input`: The source string.
- `u32 n`: The number of characters to concatenate.
- `CML_String *out`: The destination string.

Returns:
- `CML_Status`

Description:
Concatenates the first `n` characters of the input string into the out string.

Valid Usage:
- `input` **must** be a valid pointer to a `CML_String` structure.
- `out` **must** be a valid pointer to an initialized `CML_String` structure.

Return Codes:
- `CML_SUCCESS`: The string was successfully concatenated.
- `CML_ERR_NULL_PTR`: `input` or `out` was `NULL`.
- `CML_ERR_REALLOC`: An error occurred during memory reallocation.

cml_string_cat_char

Parameters:
• `const char *input`: The source string.
• `CML_String *out`: The destination string.
Returns:
• `CML_Status`

Description:
Concatenates the input `char` string into the out string.

Valid Usage:
• `input` **must** be a valid pointer to a null-terminated string.
• `out` **must** be a valid pointer to an initialized `CML_String` structure.

Return Codes:
• `CML_SUCCESS`: The string was successfully concatenated.
• `CML_ERR_NULL_PTR`: `input` or `out` was `NULL`.
• `CML_ERR_REALLOC`: An error occurred during memory reallocation.

cml_string_ncat_char

Parameters:
• `const char *input`: The source string.
• `u32 n`: The number of characters to concatenate.
• `CML_String *out`: The destination string.
Returns:
• `CML_Status`

Description:
Concatenates the first `n` characters of the input `char` string into the out string.

Valid Usage:
• `input` **must** be a valid pointer to a null-terminated string.
• `out` **must** be a valid pointer to an initialized `CML_String` structure.

Return Codes:
• `CML_SUCCESS`: The string was successfully concatenated.
• `CML_ERR_NULL_PTR`: `input` or `out` was `NULL`.
• `CML_ERR_REALLOC`: An error occurred during memory reallocation.

cml_string_eq

Parameters:
• `CML_String *s1`: The first string.
• `CML_String *s2`: The second string.
Returns:
• `b8`: whether the strings are equal.

Description:
Compares two strings for equality.

Valid Usage:
• `s1` **must** be a valid pointer to a `CML_String` structure.
• `s2` **must** be a valid pointer to a `CML_String` structure.

cml_string_eq_char

Parameters:
• `CML_String *s1`: The first string.
• `const char *s2`: The second string.
Returns:

• `b8`: whether the strings are equal.

Description:
Compares a string and a `char` string for equality.

Valid Usage:
• `s1` **must** be a valid pointer to a `CML_String` structure.
• `s2` **must** be a valid pointer to a null-terminated string.

`cml_string_debug`

Parameters:
• `CML_String *expected`: The expected string.
• `CML_String *got`: The got string.
• `b8 verbose`: Whether to print the internal information of the strings.
Returns:
• `char*`

Description:
Returns a `char` string with the debug information of the strings. Designed to be used directly in `printf`.

Valid Usage:
• `expected` **must** be a valid pointer to a `CML_String` structure.
• `got` **must** be a valid pointer to a `CML_String` structure.

6.3.2 Stack

A **stack** is a data structure that follows the Last In, First Out (LIFO) principle, and is used to store data in a way that the last element added is the first one to be removed. The stack data structure is defined as follows:

`CML_Stack`

Fields:
• `void *data`: A pointer to the stack's data.
• `u32 length`: The length of the stack.
• `u32 capacity`: The allocated capacity of the stack.
• `u32 stride`: The size of each element in the stack in bytes.
• `CML_Allocator *allocator`: Allocator used for dynamic memory allocation within the stack.
• `void (*destroyFn)(void *)`: Function to destroy the elements of the stack.

Description:
Represents a stack data structure containing information about the stack's data, length, capacity, and allocator.

Valid Usage:
• `allocator` **must** be a valid pointer to a `CML_Allocator` structure.

`_cml_stack_init`

Parameters:
• `CML_Allocator *allocator`: Pointer to the allocator to use in the stack.
• `u32 capacity`: The initial capacity of the stack.
• `u32 stride`: The size of each element in the stack in bytes.
• `void (*destroyFn)(void *)`: Function to destroy the elements of the stack.
• `CML_Stack *stack`: The stack to initialize.
Returns:
• `CML_Status`

Description:
Initializes a new stack. This function is not meant to be called directly, but rather through the `cml_stack_init` or `cml_stack_init_default` macros.

Valid Usage:
• `allocator` **must** be a valid pointer to a `CML_Allocator` structure.
• `capacity` **must** be greater than 0.
• `destroyFn` can be `NULL` if the elements of the stack do not need to be destroyed. Otherwise, it **must** be a valid pointer to a destroy function.
• `stack` **must** be a valid pointer to a `CML_Stack` structure.

Return Codes:
• `CML_SUCCESS`: The stack was successfully initialized.
• `CML_ERR_NULL_PTR`: `allocator` or `stack` was `NULL`.
• `CML_ERR_INVALID_CAPACITY`: `capacity` was less than or equal to 0.
• `CML_ERR_MALLOC`: An error occurred during memory allocation.

cml_stack_init

Parameters:
• `allocator`: Pointer to the allocator to use in the stack.
• `capacity`: The initial capacity of the stack.
• `type`: The type of the elements in the stack.
• `destroyFn`: Function to destroy the elements of the stack.
• `stack`: The stack to initialize.

Description:
Initializes a new stack.

Valid Usage:
• `allocator` **must** be a valid pointer to a `CML_Allocator` structure.
• `capacity` **must** be greater than 0.
• `destroyFn` can be `NULL` if the elements of the stack do not need to be destroyed. Otherwise, it **must** be a valid pointer to a destroy function.
• `stack` **must** be a valid pointer to a `CML_Stack` structure.

Return Codes:
• `CML_SUCCESS`: The stack was successfully initialized.
• `CML_ERR_NULL_PTR`: `allocator` or `stack` was `NULL`.
• `CML_ERR_INVALID_CAPACITY`: `capacity` was less than or equal to 0.
• `CML_ERR_MALLOC`: An error occurred during memory allocation.

cml_stack_init_default

Parameters:
• `allocator`: Pointer to the allocator to use in the stack.
• `type`: The type of the elements in the stack.
• `destroyFn`: Function to destroy the elements of the stack.
• `stack`: The stack to initialize.

Description:
Initializes a new stack with a default capacity of 2.

Valid Usage:
• `allocator` **must** be a valid pointer to a `CML_Allocator` structure.
• `destroyFn` can be `NULL` if the elements of the stack do not need to be destroyed. Otherwise, it **must** be a valid pointer to a destroy function.
• `stack` **must** be a valid pointer to a `CML_Stack` structure.

Return Codes:
• `CML_SUCCESS`: The stack was successfully initialized.
• `CML_ERR_NULL_PTR`: `allocator` or `stack` was `NULL`.

- `CML_ERR_MALLOC`: An error occurred during memory allocation.

cml_stack_destroy

Parameters:
- `void *stack`: The stack to destroy.

Returns:
- `void`

Description:
Destroys a stack, freeing its internal memory and calling the destroy function on each element.

Valid Usage:
- `stack` **must** be a valid pointer to a `CML_Stack` structure.

cml_stack_resize

Parameters:
- `u32 capacity`: The new capacity of the stack.
- `CML_Stack *out`: The stack to resize.

Returns:
- `CML_Status`

Description:
Resizes the stack to the new capacity.

Valid Usage:
- `capacity` **must** be greater than 0.
- `out` **must** be a valid pointer to a `CML_Stack` structure.

Return Codes:
- `CML_SUCCESS`: The stack was successfully resized.
- `CML_ERR_NULL_PTR`: `stack` was `NULL`.
- `CML_ERR_INVALID_CAPACITY`: `capacity` was less than or equal to 0.
- `CML_ERR_REALLOC`: An error occurred during memory reallocation.

cml_stack_push

Parameters:
- `void *element`: The element to push.
- `CML_Stack *out`: The stack to push into.

Returns:
- `CML_Status`

Description:
Pushes an element into the stack.

Valid Usage:
- `element` **must** be a valid pointer to an element of the stack's type.
- `out` **must** be a valid pointer to a `CML_Stack` structure.

Return Codes:
- `CML_SUCCESS`: The element was successfully pushed.
- `CML_ERR_NULL_PTR`: `element` or `stack` was `NULL`.
- `CML_ERR_REALLOC`: An error occurred during memory reallocation.

cml_stack_pop

Parameters:
- `CML_Stack *stack`: The stack to pop from.
- `void *out`: The element to pop into.

Returns:
• `CML_Status`

Description:
Pops an element from the stack and stores it in `out`.

Valid Usage:
• `stack` **must** be a valid pointer to a `CML_Stack` structure.
• `out` **must** be a valid pointer to an element of the stack's type.

Return Codes:
• `CML_SUCCESS`: The element was successfully popped.
• `CML_ERR_NULL_PTR`: `stack` or `out` was `NULL`.
• `CML_ERR_EMPTY_STRUCTURE`: The stack was empty.
• `CML_ERR_REALLOC`: An error occurred during memory reallocation.

cml_stack_peek

Parameters:
• `CML_Stack *stack`: The stack to peek from.
Returns:
• `void*`

Description:
Peeks at the top element of the stack and returns a pointer to it.

Valid Usage:
• `stack` **must** be a valid pointer to a `CML_Stack` structure.

Return Codes:
• `void*`: If `stack` was not `NULL`.
• `NULL`: If `stack` was `NULL` or empty.

cml_stack_eq

Parameters:
• `CML_Stack *stack1`: The first stack.
• `CML_Stack *stack2`: The second stack.
Returns:
• `b8`: whether the stacks are equal.

Description:
Compares two stacks for equality.

Valid Usage:
• `stack1` **must** be a valid pointer to a `CML_Stack` structure.
• `stack2` **must** be a valid pointer to a `CML_Stack` structure.

cml_stack_debug

Parameters:
• `CML_Stack *expected`: The expected stack.
• `CML_Stack *got`: The got stack.
• `b8 verbose`: Whether to print the internal information of the stacks.
Returns:
• `char*`

Description:
Returns a `char` string with the debug information of the stacks. Designed to be used directly in `printf`.

Valid Usage:
• `expected` **must** be a valid pointer to a `CML_Stack` structure.
• `got` **must** be a valid pointer to a `CML_Stack` structure.

6.3.3 Queue

6.3.4 Dynamic Array

A **dynamic array** is a data structure that allows for the storage of a variable number of elements by dynamically resizing the array as needed. The dynamic array data structure is defined as follows:

`CML_DArray`

Fields:
- `void *data`: A pointer to the dynamic array's data.
- `u32 length`: The length of the dynamic array.
- `u32 capacity`: The allocated capacity of the dynamic array.
- `u32 stride`: The size of each element in the dynamic array in bytes.
- `CML_Allocator *allocator`: Allocator used for dynamic memory allocation within the dynamic array.
- `void (*destroyFn)(void *)`: Function to destroy the elements of the dynamic array.

Description:
Represents a dynamic array data structure containing information about the dynamic array's data, length, capacity, and allocator.

Valid Usage:
- `allocator` **must** be a valid pointer to a `CML_Allocator` structure.

`_cml_darray_init`

Parameters:
- `CML_Allocator *allocator`: Pointer to the allocator to use in the dynamic array.
- `u32 capacity`: The initial capacity of the dynamic array.
- `u32 stride`: The size of each element in the dynamic array in bytes.
- `void (*destroyFn)(void *)`: Function to destroy the elements of the dynamic array.
- `CML_DArray *darray`: The dynamic array to initialize.

Returns:
- `CML_Status`

Description:
Initializes a new dynamic array. This function is not meant to be called directly, but rather through the `cml_darray_init` or `cml_darray_init_default` macros.

Valid Usage:
- `allocator` **must** be a valid pointer to a `CML_Allocator` structure.
- `capacity` **must** be greater than 0.
- `destroyFn` can be `NULL` if the elements of the dynamic array do not need to be destroyed. Otherwise, it **must** be a valid pointer to a destroy function.
- `darray` **must** be a valid pointer to a `CML_DArray` structure.

Return Codes:
- `CML_SUCCESS`: The dynamic array was successfully initialized.
- `CML_ERR_NULL_PTR`: `allocator` or `darray` was `NULL`.
- `CML_ERR_INVALID_CAPACITY`: `capacity` was less than or equal to 0.
- `CML_ERR_MALLOC`: An error occurred during memory allocation.

`cml_darray_init`

Parameters:
- `allocator`: Pointer to the allocator to use in the dynamic array.
- `capacity`: The initial capacity of the dynamic array.
- `type`: The type of the elements in the dynamic array.
- `destroyFn`: Function to destroy the elements of the dynamic array.

• `darray`: The dynamic array to initialize.

Description:
Initializes a new dynamic array.

Valid Usage:
• `allocator` **must** be a valid pointer to a `CML_Allocator` structure.
• `capacity` **must** be greater than 0.
• `destroyFn` can be `NULL` if the elements of the dynamic array do not need to be destroyed. Otherwise, it **must** be a valid pointer to a destroy function.
• `darray` **must** be a valid pointer to a `CML_DArray` structure.

Return Codes:
• `CML_SUCCESS`: The dynamic array was successfully initialized.
• `CML_ERR_NULL_PTR`: `allocator` or `darray` was `NULL`.
• `CML_ERR_INVALID_CAPACITY`: `capacity` was less than or equal to 0.
• `CML_ERR_MALLOC`: An error occurred during memory allocation.

cml_darray_init_default

Parameters:
• `allocator`: Pointer to the allocator to use in the dynamic array.
• `type`: The type of the elements in the dynamic array.
• `destroyFn`: Function to destroy the elements of the dynamic array.
• `darray`: The dynamic array to initialize.

Description:
Initializes a new dynamic array with a default capacity of 2.

Valid Usage:
• `allocator` **must** be a valid pointer to a `CML_Allocator` structure.
• `destroyFn` can be `NULL` if the elements of the dynamic array do not need to be destroyed. Otherwise, it **must** be a valid pointer to a destroy function.
• `darray` **must** be a valid pointer to a `CML_DArray` structure.

Return Codes:
• `CML_SUCCESS`: The dynamic array was successfully initialized.
• `CML_ERR_NULL_PTR`: `allocator` or `darray` was `NULL`.
• `CML_ERR_MALLOC`: An error occurred during memory allocation.

cml_darray_destroy

Parameters:
• `void *darray`: The dynamic array to destroy.
Returns:
• `void`

Description:
Destroys a dynamic array, freeing its internal memory and calling the destroy function on each element.

Valid Usage:
• `darray` **must** be a valid pointer to a `CML_DArray` structure.

cml_darray_resize

Parameters:
• `u32 capacity`: The new capacity of the dynamic array.
• `CML_DArray *out`: The dynamic array to resize.
Returns:
• `CML_Status`

Description:
Resizes the dynamic array to the new capacity.

Valid Usage:
• `capacity` **must** be greater than 0.
• `out` **must** be a valid pointer to a `CML_DArray` structure.

Return Codes:
• `CML_SUCCESS`: The dynamic array was successfully resized.
• `CML_ERR_NULL_PTR`: `out` was `NULL`.
• `CML_ERR_INVALID_CAPACITY`: `capacity` was less than or equal to 0.
• `CML_ERR_REALLOC`: An error occurred during memory reallocation.

cml_darray_push

Parameters:
• `void *element`: The element to push.
• `CML_DArray *out`: The dynamic array to push into.
Returns:
• `CML_Status`

Description:
Pushes an element into the dynamic array.

Valid Usage:
• `element` **must** be a valid pointer to an element of the dynamic array's type.
• `out` **must** be a valid pointer to a `CML_DArray` structure.

Return Codes:
• `CML_SUCCESS`: The element was successfully pushed.
• `CML_ERR_NULL_PTR`: `element` or `out` was `NULL`.
• `CML_ERR_REALLOC`: An error occurred during memory reallocation.

cml_darray_insert

Parameters:
• `void *element`: The element to insert.
• `u32 index`: The index to insert the element at.
• `CML_DArray *out`: The dynamic array to insert into.
Returns:
• `CML_Status`

Description:
Inserts an element into the dynamic array at the specified index.

Valid Usage:
• `element` **must** be a valid pointer to an element of the dynamic array's type.
• `index` **must** be less than or equal to the length of the dynamic array.
• `out` **must** be a valid pointer to a `CML_DArray` structure.

Return Codes:
• `CML_SUCCESS`: The element was successfully inserted.
• `CML_ERR_NULL_PTR`: `element` or `out` was `NULL`.
• `CML_ERR_INVALID_INDEX`: `index` was greater than the length of the dynamic array.
• `CML_ERR_REALLOC`: An error occurred during memory reallocation.

cml_darray_pop

Parameters:
• `CML_DArray *darray`: The dynamic array to pop from.
• `void *out`: The element to pop into.
Returns:

• CML_Status

Description:
Pops an element from the dynamic array and stores it in `out`.

Valid Usage:
• `darray` **must** be a valid pointer to a `CML_DArray` structure.
• `out` **must** be a valid pointer to a space to store an element of the dynamic array's type.

Return Codes:
• `CML_SUCCESS`: The element was successfully popped.
• `CML_ERR_NULL_PTR`: `darray` or `out` was `NULL`.
• `CML_ERR_EMPTY_STRUCTURE`: The dynamic array was empty.
• `CML_ERR_REALLOC`: An error occurred during memory reallocation.

cml_darray_remove

Parameters:
• `u32 index`: The index to remove the element at.
• `CML_DArray *darray`: The dynamic array to remove from.
• `void *out`: The element to remove into.
Returns:
• `CML_Status`

Description:
Removes an element from the dynamic array at the specified index and stores it in `out`.

Valid Usage:
• `index` **must** be less than the length of the dynamic array.
• `darray` **must** be a valid pointer to a `CML_DArray` structure.
• `out` **must** be a valid pointer to a space to store an element of the dynamic array's type.

Return Codes:
• `CML_SUCCESS`: The element was successfully removed.
• `CML_ERR_NULL_PTR`: `darray` or `out` was `NULL`.
• `CML_ERR_INVALID_INDEX`: `index` was greater than or equal to the length of the dynamic array.
• `CML_ERR_REALLOC`: An error occurred during memory reallocation.

cml_darray_get

Parameters:
• `u32 index`: The index to get the element at.
• `CML_DArray *out`: The dynamic array to get from.
Returns:
• `void*`

Description:
Gets an element from the dynamic array at the specified index and returns a pointer to it.

Valid Usage:
• `index` **must** be less than the length of the dynamic array.
• `out` **must** be a valid pointer to a `CML_DArray` structure.

Return Codes:
• `void*`: If `index` was less than the length of the dynamic array.
• `NULL`: If `index` was greater than or equal to the length of the dynamic array, `out` was `NULL` or empty.

cml_darray_set

Parameters:
• `u32 index`: The index to set the element at.
• `void *element`: The element to set.
• `CML_DArray *out`: The dynamic array to set into.
Returns:
• `CML_Status`

Description:
Sets an element in the dynamic array at the specified index, and destroys the previous element if necessary.

Valid Usage:
• `index` **must** be less than the length of the dynamic array.
• `element` **must** be a valid pointer to an element of the dynamic array's type.
• `out` **must** be a valid pointer to a `CML_DArray` structure.

Return Codes:
• `CML_SUCCESS`: The element was successfully set.
• `CML_ERR_NULL_PTR`: `element`, `out` or `out->data` was `NULL`.
• `CML_ERR_INVALID_INDEX`: `index` was greater than or equal to the length of the dynamic array.

cml_darray_eq

Parameters:
• `CML_DArray *darray1`: The first dynamic array.
• `CML_DArray *darray2`: The second dynamic array.
Returns:
• `b8`: whether the dynamic arrays are equal.

Description:
Compares two dynamic arrays for equality.

Valid Usage:
• `darray1` **must** be a valid pointer to a `CML_DArray` structure.
• `darray2` **must** be a valid pointer to a `CML_DArray` structure.

cml_darray_debug

Parameters:
• `CML_DArray *expected`: The expected dynamic array.
• `CML_DArray *got`: The got dynamic array.
• `b8 verbose`: Whether to print the internal information of the dynamic arrays.
Returns:
• `char*`

Description:
Returns a `char` string with the debug information of the dynamic arrays. Designed to be used directly in `printf`.

Valid Usage:
• `expected` **must** be a valid pointer to a `CML_DArray` structure.
• `got` **must** be a valid pointer to a `CML_DArray` structure.

6.3.5 Singly Linked List

6.3.6 Doubly Linked List

6.3.7 Binary Tree

6.3.8 Binary Search Tree

6.3.9 Hash Table

6.3.10 Graph

6.4 Expression Parsing

6.5 Arbitrary Precision Arithmetic

6.6 Matrix Numeric Type

Matrices are the basic numeric type used in CAMEL. They are used to represent scalars, vectors and matrices and are used as a way of unifying most of the numerical operations in the library. The matrix numeric type is defined as follows:

`CML_Matrix`

Fields:
- `void *data`: A pointer to the matrix's data.
- `CML_NumericType type`: The type of the matrix's data.
- `u32 rows`: The number of rows in the matrix.
- `u32 columns`: The number of columns in the matrix.
- `b8 rowmajor`: Whether the matrix is row-major. If `false`, the matrix is column-major.
- `CML_Allocator *allocator`: Allocator used for dynamic memory allocation within the matrix.

Description:
Represents a matrix numeric type containing information about the matrix's data, number of rows, and number of columns.

Valid Usage:
- `data` **must** be a valid pointer to the matrix's data.

Matrices can hold the types enumerated in `CML_NumericType`. The matrix numeric type is used to represent scalars (when the number of rows and columns is 1), vectors (when the number of rows or columns is 1), and matrices (when the number of rows and columns is greater than 1). The matrix numeric type is used to unify most of the numerical operations in the library, such as addition, subtraction, multiplication, division and others. The matrix uses the following functions to perform these operations:

`cml_matrix_init`

Parameters:
- `CML_Allocator *allocator`: Allocator used for dynamic memory allocation within the matrix.
- `u32 rows`: The number of rows in the matrix.
- `u32 columns`: The number of columns in the matrix.
- `b8 rowmajor`: Whether the matrix is row-major.
- `CML_NumericType type`: The type of the matrix's data.
- `CML_Matrix *out`: The matrix to initialize.

Returns:
- `CML_Status`

Description:
Initializes a new matrix and sets everything to 0. Keep in mind that if more complex values are used, such as `CML_BigInt` or other library defined types, these will not be

initialized, the position of their memory is just set to 0.

Valid Usage:
• `out` **must** be a valid pointer to a `CML_Matrix` structure.
• `allocator` **must** be a valid pointer to a `CML_Allocator` structure.
• `type` **must** be a valid `CML_NumericType`.
• `rows` **must** be greater than 0.
• `columns` **must** be greater than 0.

Return Codes:
• `CML_SUCCESS`: The matrix was successfully initialized.
• `CML_ERR_NULL_PTR`: `out` or `allocator` was `NULL`.
• `CML_ERR_INVALID_SIZE`: `rows` or `columns` was 0.
• `CML_ERR_CALLOC`: An error occurred during memory allocation.

cml_matrix_init0

Parameters:
• `CML_Allocator *allocator`: Allocator used for dynamic memory allocation within the matrix.
• `u32 rows`: The number of rows in the matrix.
• `u32 columns`: The number of columns in the matrix.
• `b8 rowmajor`: Whether the matrix is row-major.
• `CML_NumericType type`: The type of the matrix's data.
• `CML_Matrix *out`: The matrix to initialize.
Returns:
• `CML_Status`

Description:
Initializes a new matrix and sets everything to 0. In contrast to `cml_matrix_init`, this function initializes more complex types, such as `CML_BigInt` or other library defined types, to 0.

Valid Usage:
• `out` **must** be a valid pointer to a `CML_Matrix` structure.
• `allocator` **must** be a valid pointer to a `CML_Allocator` structure.
• `type` **must** be a valid `CML_NumericType`.
• `rows` **must** be greater than 0.
• `columns` **must** be greater than 0.

Return Codes:
• `CML_SUCCESS`: The matrix was successfully initialized.
• `CML_ERR_NULL_PTR`: `out` or `allocator` was `NULL`.
• `CML_ERR_INVALID_SIZE`: `rows` or `columns` was 0.
• `CML_ERR_CALLOC`: An error occurred during memory allocation.

cml_matrix_destroy

Parameters:
• `CML_Matrix *matrix`: The matrix to destroy.
Returns:
• `void`

Description:
Destroys a matrix, freeing its internal memory. If it contains more complex types, such as `CML_BigInt` or other library defined types, it will call the destroy function on each element.

Valid Usage:
• `matrix` **must** be a valid pointer to a `CML_Matrix` structure.

cml_matrix_set

Parameters:
• `void *element`: The element to set into the matrix.
• `u32 row`: The row to set the element at.
• `u32 column`: The column to set the element at.
• `CML_Matrix *out`: The matrix to set into.
Returns:
• `CML_Status`

Description:
Sets an element in the matrix at the specified row and column. If the matrix contains more complex types, such as `CML_BigInt` or other library defined types, it will take ownership of the element, and the user should no longer use it.

Valid Usage:
• `element` **must** be a valid pointer to an element of the matrix's type.
• `row` **must** be less than the number of rows in the matrix but greater than or equal to 0.
• `column` **must** be less than the number of columns in the matrix but greater than or equal to 0.
• `out` **must** be a valid pointer to a `CML_Matrix` structure.

Return Codes:
• `CML_SUCCESS`: The element was successfully set.
• `CML_ERR_NULL_PTR`: `element` or `out` was `NULL`.
• `CML_ERR_INVALID_INDEX`: `row` or `column` was out of bounds.

cml_matrix_get

Parameters:
• `u32 row`: The row to get the element from.
• `u32 column`: The column to get the element from.
• `const CML_Matrix *out`: The matrix to get from.
Returns:
• `void*`

Description:
Gets an element from the matrix at the specified row and column and returns a pointer to it.

Valid Usage:
• `row` **must** be less than the number of rows in the matrix but greater than or equal to 0.
• `column` **must** be less than the number of columns in the matrix but greater than or equal to 0.
• `out` **must** be a valid pointer to a `CML_Matrix` structure.

cml_matrix_select

Parameters:
• `CML_Allocator *allocator`: Allocator used for dynamic memory allocation within the new matrix.
• `const CML_Matrix *A`: The matrix to select from.
• `CML_Matrix *p`: The row permutation vector. For example, if $\mathbf{p} = [2 \quad 0 \quad 1]$ (or $\mathbf{p}^T$, it does not matter if it is a row or a column vector), then the new matrix will be $\begin{bmatrix} \mathbf{A}_{2*} \\ \mathbf{A}_{0*} \\ \mathbf{A}_{1*} \end{bmatrix}$, where $\mathbf{A}_{i*}$ is the i-th row of $\mathbf{A}$.
• `CML_Matrix *q`: The column permutation vector. For example, if $\mathbf{q} = [2 \quad 0 \quad 1]$ (or $\mathbf{q}^T$, it does not matter if it is a row or a column vector), then the new matrix will be

$[\mathbf{A}_{*2} \quad \mathbf{A}_{*0} \quad \mathbf{A}_{*1}]$, where $\mathbf{A}_{*i}$ is the i-th column of $\mathbf{A}$.
• `b8 rowmajor`: Whether the new matrix is row-major.
• `CML_Matrix *out`: The new matrix to select into.
Returns:
• `CML_Status`

Description:
Selects rows and columns from a matrix and stores them in a new matrix. If the matrix contains more complex types, such as `CML_BigInt` or other library defined types, they will be copied into the new matrix. If either the input permutation vectors are `NULL`, the identity permutation will be used for that vector (i.e., $\mathbf{p} \vee \mathbf{q} = [0 \quad 1 \quad \cdots \quad n-1]$, where n is the number of rows or columns in the matrix). If the type of the permutation vectors is not `CML_U32`, a conversion will be attempted. In general, if both permutation vectors are `NULL`, the new matrix will be a copy of the input matrix. The allocator can be `NULL`, in which case the allocator of `A` will be used.

Valid Usage:
• `allocator` **may** be `NULL`, in which case the allocator of `A` will be used.
• `A` **must** be a valid pointer to a `CML_Matrix` structure.
• `out` **must** be a valid pointer to a `CML_Matrix` structure.
• `p` and `q` can be `NULL`, and if not `NULL`, it is recommended that they are of type `CML_U32`.

Return Codes:
• `CML_SUCCESS`: The new matrix was successfully selected.
• `CML_ERR_NULL_PTR`: `A` or `out` was `NULL`.
• `CML_ERR_INVALID_PERMUTATION`: The permutation vectors were out of bounds.
• Any error code from `cml_matrix_init` (used to initialize the new matrix or `p` or `q`, if either is `NULL`).

cml_matrix_add

Parameters:
• `CML_Allocator *allocator`: Allocator used for dynamic memory allocation within the new matrix.
• `const CML_Matrix *left`: The left matrix ($\mathbf{A}$).
• `const CML_Matrix *right`: The right matrix ($\mathbf{B}$).
• `b8 rowmajor`: Whether the new matrix is row-major.
• `CML_Matrix *out`: The new matrix to store the result of the addition ($\mathbf{C}$).
Returns:
• `CML_Status`

Description:
Adds two matrices and stores the result in a new matrix. This new matrix is initialized using the input allocator, or the allocator of the left matrix if the input allocator is `NULL`. If one of the inputs is a scalar (i.e., a matrix with one row and one column) and the other is not, the scalar will be added to each element of the other matrix.

Operation:
$\mathbf{C} = \mathbf{A} + \mathbf{B}$. If only one of the inputs is a scalar, either $A_{ij} = a$ or $B_{ij} = b$.

Algorithm:
For each element C_{ij} of $\mathbf{C}$, $C_{ij} = A_{ij} + B_{ij}$, or, if only one of the inputs is a scalar, $C_{ij} = A_{ij} + b$ or $C_{ij} = a + B_{ij}$.

Valid Usage:
• `allocator` **may** be `NULL`, in which case the allocator of `left` will be used.
• `left` **must** be a valid pointer to a `CML_Matrix` structure.
• `right` **must** be a valid pointer to a `CML_Matrix` structure.
• `left` and `right` **must** have the same number of rows and columns, or one of them must be a scalar.
• `left` and `right` **must** have the same type.

• `out` **must** be a valid pointer to a `CML_Matrix` structure.

Return Codes:
• `CML_SUCCESS`: The new matrix was successfully added.
• `CML_ERR_NULL_PTR`: `left`, `right` or `out` was `NULL`.
• `CML_ERR_INCOMPATIBLE_TYPES`: `left` and `right` had different types.
• `CML_ERR_INCOMPATIBLE_SIZE`: `left` and `right` had different sizes and neither was a scalar.
• Any error code from `cml_matrix_init` (used to initialize the new matrix).

cml_matrix_add_inplace

Parameters:
• `const CML_Matrix *right`: The right matrix (**A**).
• `CML_Matrix *out`: The left matrix (**B**).
Returns:
• `CML_Status`

Description:
Adds two matrices and stores the result in the out matrix. If the left matrix is a scalar (i.e., a matrix with one row and one column) and the right matrix is not, the scalar will be added to each element of the right matrix.

Operation:
$\mathbf{B} = \mathbf{B} + \mathbf{A}$. If only the right input is a scalar, $A_{ij} = a$.

Algorithm:
For each element B_{ij} of $\mathbf{B}$, $B_{ij} = B_{ij} + A_{ij}$, or, if only the right input is a scalar, $B_{ij} = B_{ij} + a$.

Valid Usage:
• `right` **must** be a valid pointer to a `CML_Matrix` structure.
• `out` **must** be a valid pointer to a `CML_Matrix` structure.
• `out` and `right` **must** have the same number of rows and columns, or right must be a scalar.
• `out` and `right` **must** have the same type.

Return Codes:
• `CML_SUCCESS`: The new matrix was successfully added.
• `CML_ERR_NULL_PTR`: `right` or `out` was `NULL`.
• `CML_ERR_INCOMPATIBLE_TYPES`: `out` and `right` had different types.
• `CML_ERR_INCOMPATIBLE_SIZE`: `out` and `right` had different sizes and `right` was not a scalar.

cml_matrix_sub

Parameters:
• `CML_Allocator *allocator`: Allocator used for dynamic memory allocation within the new matrix.
• `const CML_Matrix *left`: The left matrix (**A**).
• `const CML_Matrix *right`: The right matrix (**B**).
• `b8 rowmajor`: Whether the new matrix is row-major.
• `CML_Matrix *out`: The new matrix to store the result of the subtraction (**C**).
Returns:
• `CML_Status`

Description:
Subtracts two matrices and stores the result in a new matrix. This new matrix is initialized using the input allocator, or the allocator of the left matrix if the input allocator is `NULL`. If one of the inputs is a scalar (i.e., a matrix with one row and one column) and the other is not, the scalar will be subtracted from each element of the

other matrix.

Operation:
$\mathbf{C} = \mathbf{A} - \mathbf{B}$. If only one of the inputs is a scalar, either $A_{ij} = a$ or $B_{ij} = b$.

Algorithm:
For each element C_{ij} of $\mathbf{C}$, $C_{ij} = A_{ij} - B_{ij}$, or, if only one of the inputs is a scalar, $C_{ij} = A_{ij} - b$ or $C_{ij} = a - B_{ij}$.

Valid Usage:
• `allocator` **may** be `NULL`, in which case the allocator of `left` will be used.
• `left` **must** be a valid pointer to a `CML_Matrix` structure.
• `right` **must** be a valid pointer to a `CML_Matrix` structure.
• `left` and `right` **must** have the same number of rows and columns, or one of them must be a scalar.
• `left` and `right` **must** have the same type.
• `out` **must** be a valid pointer to a `CML_Matrix` structure.

Return Codes:
• `CML_SUCCESS`: The new matrix was successfully subtracted.
• `CML_ERR_NULL_PTR`: `left`, `right` or `out` was `NULL`.
• `CML_ERR_INCOMPATIBLE_TYPES`: `left` and `right` had different types.
• `CML_ERR_INCOMPATIBLE_SIZE`: `left` and `right` had different sizes and neither was a scalar.
• Any error code from `cml_matrix_init` (used to initialize the new matrix).

cml_matrix_sub_inplace

Parameters:
• `const CML_Matrix *right`: The right matrix (**A**).
• `CML_Matrix *out`: The left matrix (**B**).
Returns:
• `CML_Status`

Description:
Subtracts two matrices and stores the result in the out matrix. If the left matrix is a scalar (i.e., a matrix with one row and one column) and the right matrix is not, the scalar will be subtracted from each element of the right matrix.

Operation:
$\mathbf{B} = \mathbf{B} - \mathbf{A}$. If only the right input is a scalar, $A_{ij} = a$.

Algorithm:
For each element B_{ij} of $\mathbf{B}$, $B_{ij} = B_{ij} - A_{ij}$, or, if only the right input is a scalar, $B_{ij} = B_{ij} - a$.

Valid Usage:
• `right` **must** be a valid pointer to a `CML_Matrix` structure.
• `out` **must** be a valid pointer to a `CML_Matrix` structure.
• `out` and `right` **must** have the same number of rows and columns, or right must be a scalar.
• `out` and `right` **must** have the same type.

Return Codes:
• `CML_SUCCESS`: The new matrix was successfully subtracted.
• `CML_ERR_NULL_PTR`: `right` or `out` was `NULL`.
• `CML_ERR_INCOMPATIBLE_TYPES`: `out` and `right` had different types.
• `CML_ERR_INCOMPATIBLE_SIZE`: `out` and `right` had different sizes and `right` was not a scalar.

7 Algebra

7.1 Linear Algebra

Linear algebra is the branch of mathematics concerning linear equations and linear maps and their representations through matrices and vector spaces. Inside of the Camel library, linear algebra is divided into two categories: fixed-size linear algebra and variable-size linear algebra. Fixed-size linear algebra is used for small, fixed-size matrices and vectors up to 4x4, while variable-size linear algebra is used for larger, variable-size matrices and vectors.

7.1.1 Fixed-Size Linear Algebra

The **fixed-size linear algebra** functions are designed to be used in performance-critical applications where the size of the matrices and vectors doesn't exceed 4x4, such as in graphics programming. The fixed-size linear algebra objects are stored in column-major order to be compatible with OpenGL and other graphics libraries. For further optimization, the fixed-size linear algebra module offers header-only or compiled versions of the functions, depending on the user's needs. By using the standard `cml_...` functions, the user can use the header-only version, while by using the `cmlc_...` functions, the user can use the compiled version.

The fixed-size linear algebra vectors are defined as follows:

CML_Vector2

Fields:
- struct:
  - `f32 x`: The x component of the vector.
  - `f32 y`: The y component of the vector.
- `f32 array[2]`: The vector as an array.

Description:
Represents a 2-dimensional vector.

CML_Vector3

Fields:
- struct:
  - `f32 x`: The x component of the vector.
  - `f32 y`: The y component of the vector.
  - `f32 z`: The z component of the vector.
- `f32 array[3]`: The vector as an array.

Description:
Represents a 3-dimensional vector.

CML_Vector4

Fields:
- struct:
  - `f32 x`: The x component of the vector.
  - `f32 y`: The y component of the vector.
  - `f32 z`: The z component of the vector.
  - `f32 w`: The w component of the vector.
- `f32 array[4]`: The vector as an array.

Description:
Represents a 4-dimensional vector.

The fixed-size linear algebra matrices are defined as follows:

CML_Matrix2x2

Fields:
- struct:
  - `f32 m00`: The element at row 0, column 0 of the matrix.
  - `f32 m10`: The element at row 1, column 0 of the matrix.
  - `f32 m01`: The element at row 0, column 1 of the matrix.
  - `f32 m11`: The element at row 1, column 1 of the matrix.
- `f32 array[4]`: The matrix as an array.

Description:
Represents a 2x2 matrix.

CML_Matrix3x3

Fields:
- struct:
  - `f32 m00`: The element at row 0, column 0 of the matrix.
  - `f32 m10`: The element at row 1, column 0 of the matrix.
  - `f32 m20`: The element at row 2, column 0 of the matrix.
  - `f32 m01`: The element at row 0, column 1 of the matrix.
  - `f32 m11`: The element at row 1, column 1 of the matrix.
  - `f32 m21`: The element at row 2, column 1 of the matrix.
  - `f32 m02`: The element at row 0, column 2 of the matrix.
  - `f32 m12`: The element at row 1, column 2 of the matrix.
  - `f32 m22`: The element at row 2, column 2 of the matrix.
- `f32 array[9]`: The matrix as an array.

Description:
Represents a 3x3 matrix.

CML_Matrix4x4

Fields:
- struct:
  - `f32 m00`: The element at row 0, column 0 of the matrix.
  - `f32 m10`: The element at row 1, column 0 of the matrix.
  - `f32 m20`: The element at row 2, column 0 of the matrix.
  - `f32 m30`: The element at row 3, column 0 of the matrix.
  - `f32 m01`: The element at row 0, column 1 of the matrix.
  - `f32 m11`: The element at row 1, column 1 of the matrix.
  - `f32 m21`: The element at row 2, column 1 of the matrix.
  - `f32 m31`: The element at row 3, column 1 of the matrix.
  - `f32 m02`: The element at row 0, column 2 of the matrix.
  - `f32 m12`: The element at row 1, column 2 of the matrix.
  - `f32 m22`: The element at row 2, column 2 of the matrix.
  - `f32 m32`: The element at row 3, column 2 of the matrix.
  - `f32 m03`: The element at row 0, column 3 of the matrix.
  - `f32 m13`: The element at row 1, column 3 of the matrix.
  - `f32 m23`: The element at row 2, column 3 of the matrix.
  - `f32 m33`: The element at row 3, column 3 of the matrix.
- `f32 array[16]`: The matrix as an array.

Description:
Represents a 4x4 matrix.

CML_Matrix2x3

Fields:
- struct:
  - `f32 m00`: The element at row 0, column 0 of the matrix.
  - `f32 m10`: The element at row 1, column 0 of the matrix.
  - `f32 m01`: The element at row 0, column 1 of the matrix.

`f32 m11`: The element at row 1, column 1 of the matrix.
`f32 m02`: The element at row 0, column 2 of the matrix.
`f32 m12`: The element at row 1, column 2 of the matrix.
- `f32 array[6]`: The matrix as an array.

Description:
Represents a 2x3 matrix.

CML_Matrix2x4

Fields:
- `struct`:
  `f32 m00`: The element at row 0, column 0 of the matrix.
  `f32 m10`: The element at row 1, column 0 of the matrix.
  `f32 m01`: The element at row 0, column 1 of the matrix.
  `f32 m11`: The element at row 1, column 1 of the matrix.
  `f32 m02`: The element at row 0, column 2 of the matrix.
  `f32 m12`: The element at row 1, column 2 of the matrix.
  `f32 m03`: The element at row 0, column 3 of the matrix.
  `f32 m13`: The element at row 1, column 3 of the matrix.
- `f32 array[8]`: The matrix as an array.

Description:
Represents a 2x4 matrix.

CML_Matrix3x2

Fields:
- `struct`:
  `f32 m00`: The element at row 0, column 0 of the matrix.
  `f32 m10`: The element at row 1, column 0 of the matrix.
  `f32 m20`: The element at row 2, column 0 of the matrix.
  `f32 m01`: The element at row 0, column 1 of the matrix.
  `f32 m11`: The element at row 1, column 1 of the matrix.
  `f32 m21`: The element at row 2, column 1 of the matrix.
- `f32 array[6]`: The matrix as an array.

Description:
Represents a 3x2 matrix.

CML_Matrix3x4

Fields:
- `struct`:
  `f32 m00`: The element at row 0, column 0 of the matrix.
  `f32 m10`: The element at row 1, column 0 of the matrix.
  `f32 m20`: The element at row 2, column 0 of the matrix.
  `f32 m01`: The element at row 0, column 1 of the matrix.
  `f32 m11`: The element at row 1, column 1 of the matrix.
  `f32 m21`: The element at row 2, column 1 of the matrix.
  `f32 m02`: The element at row 0, column 2 of the matrix.
  `f32 m12`: The element at row 1, column 2 of the matrix.
  `f32 m22`: The element at row 2, column 2 of the matrix.
  `f32 m03`: The element at row 0, column 3 of the matrix.
  `f32 m13`: The element at row 1, column 3 of the matrix.
  `f32 m23`: The element at row 2, column 3 of the matrix.
- `f32 array[12]`: The matrix as an array.

Description:
Represents a 3x4 matrix.

CML_Matrix4x2

Fields:

- struct:
  - `f32 m00`: The element at row 0, column 0 of the matrix.
  - `f32 m10`: The element at row 1, column 0 of the matrix.
  - `f32 m20`: The element at row 2, column 0 of the matrix.
  - `f32 m30`: The element at row 3, column 0 of the matrix.
  - `f32 m01`: The element at row 0, column 1 of the matrix.
  - `f32 m11`: The element at row 1, column 1 of the matrix.
  - `f32 m21`: The element at row 2, column 1 of the matrix.
  - `f32 m31`: The element at row 3, column 1 of the matrix.
- `f32 array[8]`: The matrix as an array.

Description:
Represents a 4x2 matrix.

CML_Matrix4x3

Fields:

- struct:
  - `f32 m00`: The element at row 0, column 0 of the matrix.
  - `f32 m10`: The element at row 1, column 0 of the matrix.
  - `f32 m20`: The element at row 2, column 0 of the matrix.
  - `f32 m30`: The element at row 3, column 0 of the matrix.
  - `f32 m01`: The element at row 0, column 1 of the matrix.
  - `f32 m11`: The element at row 1, column 1 of the matrix.
  - `f32 m21`: The element at row 2, column 1 of the matrix.
  - `f32 m31`: The element at row 3, column 1 of the matrix.
  - `f32 m02`: The element at row 0, column 2 of the matrix.
  - `f32 m12`: The element at row 1, column 2 of the matrix.
  - `f32 m22`: The element at row 2, column 2 of the matrix.
  - `f32 m32`: The element at row 3, column 2 of the matrix.
- `f32 array[12]`: The matrix as an array.

Description:
Represents a 4x3 matrix.

During the rest of this section, any reference to a vector will be done using `CML_Vectorn` (or `CML_VECTORN` in macros), where `n` is the number of components of the vector. Similarly, any reference to a matrix will be done using `CML_Matrixnxm` (or `CML_MATRIXNXM` in macros), where `n` is the number of rows and `m` is the number of columns of the matrix, or using `CML_Matrixnxn` if the matrix is necessarily square.

For the creation of vectors and matrices, the following macros are provided:

CML_VECTORN_ZERO

Description:
Sets the vector to zero.

CML_VECTORN_I

Description:
Sets the vector to the i-th unit vector.

CML_VECTORN_J

Description:
Sets the vector to the j-th unit vector.

CML_VECTORN_K

Description:
Sets the vector to the k-th unit vector.

CML_VECTORN_L

Description:
Sets the vector to the l-th unit vector.

CML_VECTORN

Parameters:
- `x`: The x component of the vector.
- `y`: The y component of the vector.
- `z`: The z component of the vector (only for 3D and 4D vectors).
- `w`: The w component of the vector (only for 4D vectors).

Description:
Sets the vector to the specified components.

CML_MATRIXNXM_ZERO

Description:
Sets the matrix to zero.

CML_MATRIXNXM_IDENTITY

Description:
Sets the matrix to the identity matrix.

CML_MATRIXNXM_ONE

Description:
Sets the matrix to the one matrix.

CML_MATRIXNXM

Parameters:
- `m00`: The element at row 0, column 0 of the matrix.
- `m01`: The element at row 0, column 1 of the matrix.
- `m02`: The element at row 0, column 2 of the matrix (if applicable).
- `m03`: The element at row 0, column 3 of the matrix (if applicable).
- `m10`: The element at row 1, column 0 of the matrix.
- `m11`: The element at row 1, column 1 of the matrix.
- `m12`: The element at row 1, column 2 of the matrix (if applicable).
- `m13`: The element at row 1, column 3 of the matrix (if applicable).
- `m20`: The element at row 2, column 0 of the matrix (if applicable).
- `m21`: The element at row 2, column 1 of the matrix (if applicable).
- `m22`: The element at row 2, column 2 of the matrix (if applicable).
- `m23`: The element at row 2, column 3 of the matrix (if applicable).
- `m30`: The element at row 3, column 0 of the matrix (if applicable).
- `m31`: The element at row 3, column 1 of the matrix (if applicable).
- `m32`: The element at row 3, column 2 of the matrix (if applicable).
- `m33`: The element at row 3, column 3 of the matrix (if applicable).

Description:
Sets the matrix to the specified elements row-wise, for more intuitive initialization.

All function in this module expect valid pointers, and all arguments to hold values, except the `out` parameter. Knwoing this, all valid usage parts will be skipped for brevity. The fixed-size linear algebra functions are defined as follows:

cml_vectorn_add

Parameters:
• `const CML_Vectorn *v`: The first vector.
• `const CML_Vectorn *w`: The second vector.
• `CML_Vectorn *out`: The vector to store the result.
Returns:
• `void`

Description:
Adds two vectors.

Operation:
Standard vector sum: **out** $= \mathbf{v} + \mathbf{w}$

Algorithm:
For each component i of the vectors, $out_i = v_i + w_i$.

cml_vectorn_add_f32

Parameters:
• `const CML_Vectorn *v`: The vector.
• `f32 t`: The scalar.
• `CML_Vectorn *out`: The vector to store the result.
Returns:
• `void`

Description:
Adds a scalar to a vector.

Operation:
"Broadcast" sum: **out** $= \mathbf{v} + t$

Algorithm:
For each component i of the vector, $out_i = v_i + t$.

cml_vectorn_sub

Parameters:
• `const CML_Vectorn *v`: The first vector.
• `const CML_Vectorn *w`: The second vector.
• `CML_Vectorn *out`: The vector to store the result.
Returns:
• `void`

Description:
Subtracts two vectors.

Operation:
Standard vector subtraction: **out** $= \mathbf{v} - \mathbf{w}$

Algorithm:
For each component i of the vectors, $out_i = v_i - w_i$.

cml_vectorn_sub_f32

Parameters:
• `const CML_Vectorn *v`: The vector.
• `f32 t`: The scalar.
• `CML_Vectorn *out`: The vector to store the result.
Returns:
• `void`

Description:
Subtracts a scalar from a vector.

Operation:
"Broadcast" subtraction: $\mathbf{out} = \mathbf{v} - t$

Algorithm:
For each component i of the vector, $out_i = v_i - t$.

cml_vectorn_scale

Parameters:
- `const CML_Vectorn *v`: The vector.
- `f32 t`: The scalar.
- `CML_Vectorn *out`: The vector to store the result.

Returns:
- `void`

Description:
Scales a vector by a scalar.

Operation:
Standard vector scaling: $\mathbf{out} = t\mathbf{v}$

Algorithm:
For each component i of the vector, $out_i = tv_i$.

cml_vectorn_mod

Parameters:
- `const CML_Vectorn *v`: The vector.

Returns:
- `f32`: the modulus of the vector.

Description:
Returns the modulus of a vector.

Operation:
Standard vector modulus: $\|\mathbf{v}\|_2 = \sqrt{\sum_{i=0}^{n-1} v_i^2}$.

Algorithm:
For each component i of the vector, accumulate the square of the component. Return the square root of the accumulated value.

cml_vectorn_mod2

Parameters:
- `const CML_Vectorn *v`: The vector.

Returns:
- `f32`: the squared modulus of the vector.

Description:
Returns the squared modulus of a vector.

Operation:
Standard vector squared modulus: $\|\mathbf{v}\|_2^2 = \sum_{i=0}^{n-1} v_i^2$.

Algorithm:
For each component i of the vector, accumulate the square of the component. Return the accumulated value.

cml_vectorn_norm

Parameters:
- `const CML_Vectorn *v`: The vector.
- `CML_Vectorn *out`: The vector to store the result.

Returns:
• `void`

Description:
Normalizes a vector.

Operation:
Standard vector normalization: $\mathbf{out} = \hat{\mathbf{v}} = \frac{\mathbf{v}}{\|\mathbf{v}\|_2}$.

Algorithm:
Calculate $\|\mathbf{v}\|_2$. Then, for each component i of the vector, $out_i = \frac{v_i}{\|\mathbf{v}\|_2}$.

cml_vectorn_dot

Parameters:
• `const CML_Vectorn *v`: The first vector.
• `const CML_Vectorn *w`: The second vector.
Returns:
• `f32`: the dot product of the vectors.

Description:
Returns the dot product of two vectors.

Operation:
Standard vector dot product: $\mathbf{v} \cdot \mathbf{w} = \sum_{i=0}^{n-1} v_i w_i$.

Algorithm:
For each component i of the vectors, accumulate the product of the components. Return the accumulated value.

cml_vector3_cross

Parameters:
• `const CML_Vector3 *v`: The first vector.
• `const CML_Vector3 *w`: The second vector.
• `CML_Vector3 *out`: The vector to store the result.
Returns:
• `void`

Description:
Returns the cross product of two 3D vectors.

Operation:
Standard vector cross product: $\mathbf{out} = \mathbf{v} \times \mathbf{w} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ v_x & v_y & v_z \\ w_x & w_y & w_z \end{vmatrix}$.

Algorithm:
$out_x = v_y w_z - v_z w_y$,
$out_y = v_z w_x - v_x w_z$,
$out_z = v_x w_y - v_y w_x$.

cml_vectorn_distance

Parameters:
• `const CML_Vectorn *v`: The first vector.
• `const CML_Vectorn *w`: The second vector.
Returns:
• `f32`: the distance between the vectors.

Description:
Returns the distance between two vectors.

Operation:
Standard vector distance: $d(\mathbf{v}, \mathbf{w}) = \|\mathbf{v} - \mathbf{w}\|_2 = \sqrt{\sum_{i=0}^{n-1}(v_i - w_i)^2}$.

Algorithm:
For each component i of the vectors, accumulate the square of the difference between the components. Return the square root of the accumulated value.

cml_vectorn_distance2

Parameters:
• `const CML_Vectorn *v`: The first vector.
• `const CML_Vectorn *w`: The second vector.
Returns:
• `f32`: the squared distance between the vectors.

Description:
Returns the squared distance between two vectors.

Operation:
Standard vector squared distance: $d^2(\mathbf{v}, \mathbf{w}) = \|\mathbf{v} - \mathbf{w}\|_2^2 = \sum_{i=0}^{n-1}(v_i - w_i)^2$.

Algorithm:
For each component i of the vectors, accumulate the square of the difference between the components. Return the accumulated value.

cml_vectorn_angle

Parameters:
• `const CML_Vectorn *v`: The first vector.
• `const CML_Vectorn *w`: The second vector.
Returns:
• `f32`: the angle between the vectors.

Description:
Returns the angle between two vectors.

Operation:
Standard vector angle: $\theta = \arccos\left(\frac{\mathbf{v}\cdot\mathbf{w}}{\|\mathbf{v}\|_2\|\mathbf{w}\|_2}\right)$.

Algorithm:
Return the arccosine of the dot product of the vectors divided by the product of their moduli.

cml_vectorn_project

Parameters:
• `const CML_Vectorn *v`: The first vector.
• `const CML_Vectorn *w`: The second vector.
• `CML_Vectorn *out`: The vector to store the result.
Returns:
• `void`

Description:
Projects a vector onto another.

Operation:
Standard vector projection: $\mathbf{out} = \frac{\mathbf{v}\cdot\mathbf{w}}{\|\mathbf{w}\|_2^2}\mathbf{w}$.

Algorithm:
Calculate $\frac{\mathbf{v}\cdot\mathbf{w}}{\|\mathbf{w}\|_2^2}$. Then, for each component i of the vectors, $out_i = \frac{\mathbf{v}\cdot\mathbf{w}}{\|\mathbf{w}\|_2^2}w_i$.

cml_vectorn_reflect

Parameters:
• `const CML_Vectorn *v`: The vector.
• `const CML_Vectorn *n`: The normal vector.
• `CML_Vectorn *out`: The vector to store the result.
Returns:
• `void`

Description:
Reflects a vector across the suface defined by a normal vector.

Operation:
Standard vector reflection: $\mathbf{out} = \mathbf{v} - \frac{2(\mathbf{v}\cdot\mathbf{n})}{\|\mathbf{n}\|_2}\mathbf{n}$.

Algorithm:
Calculate $\frac{2(\mathbf{v}\cdot\mathbf{n})}{\|\mathbf{n}\|_2}$. Then, for each component i of the vectors, $out_i = v_i - \frac{2(\mathbf{v}\cdot\mathbf{n})}{\|\mathbf{n}\|_2} n_i$.

cml_vectorn_eq

Parameters:
• `const CML_Vectorn *v`: The first vector.
• `const CML_Vectorn *w`: The second vector.
Returns:
• `b8`: whether the vectors are equal.

Description:
Compares two vectors for equality.

Operation:
Standard vector equality: $\mathbf{v} = \mathbf{w} \Leftrightarrow \forall i, v_i = w_i$.

Algorithm:
For each component i of the vectors, if $v_i \neq w_i$, return `false`. Return `true` otherwise.

cml_vectorn_debug

Parameters:
• `const CML_Vectorn *expected`: The expected vector.
• `const CML_Vectorn *got`: The got vector.
Returns:
• `char*`

Description:
Returns a `char` string with the debug information of the vectors. Designed to be used directly in `printf`.

cml_matrixnxm_add

Parameters:
• `const CML_Matrixnxm *A`: The left matrix.
• `const CML_Matrixnxm *B`: The right matrix.
• `CML_Matrixnxm *OUT`: The matrix to store the result.
Returns:
• `void`

Description:
Adds two matrices.

Operation:
Standard matrix sum: $\mathbf{OUT} = \mathbf{A} + \mathbf{B}$

Algorithm:
For each element i, j of the matrices, $OUT_{ij} = A_{ij} + B_{ij}$.

cml_matrixnxm_sub

Parameters:
- `const CML_Matrixnxm *A`: The left matrix.
- `const CML_Matrixnxm *B`: The right matrix.
- `CML_Matrixnxm *OUT`: The matrix to store the result.

Returns:
- `void`

Description:
Subtracts two matrices.

Operation:
Standard matrix subtraction: $\mathbf{OUT} = \mathbf{A} - \mathbf{B}$

Algorithm:
For each element i, j of the matrices, $OUT_{ij} = A_{ij} - B_{ij}$.

cml_matrixnxm_scale

Parameters:
- `const CML_Matrixnxm *A`: The matrix.
- `f32 t`: The scalar.
- `CML_Matrixnxm *OUT`: The matrix to store the result.

Returns:
- `void`

Description:
Scales a matrix by a scalar.

Operation:
Standard matrix scaling: $\mathbf{OUT} = t\mathbf{A}$

Algorithm:
For each element i, j of the matrix, $OUT_{ij} = tA_{ij}$.

cml_matrixnxn_mul

Parameters:
- `const CML_Matrixnxn *A`: The left matrix.
- `const CML_Matrixnxn *B`: The right matrix.
- `CML_Matrixnxn *OUT`: The matrix to store the result.

Returns:
- `void`

Description:
Multiplies two square matrices.

Operation:
Standard matrix multiplication: $\mathbf{OUT} = \mathbf{AB}$

Algorithm:
For each element i, j of the matrices, $OUT_{ij} = \sum_{k=0}^{n-1} A_{ik}B_{kj}$.

cml_matrixnxm_mul_matrixmxp

Parameters:
- `const CML_Matrixnxm *A`: The left matrix.
- `const CML_Matrixmxp *B`: The right matrix.
- `CML_Matrixnxp *OUT`: The matrix to store the result.

Returns:
- `void`

Description:
Multiplies a matrix by a matrix.

Operation:
Standard matrix multiplication: $\mathbf{OUT} = \mathbf{AB}$

Algorithm:
For each element i, j of the matrices, $OUT_{ij} = \sum_{k=0}^{m-1} A_{ik} B_{kj}$.

cml_matrixnxm_mul_vectorm

Parameters:
- `const CML_Matrixnxm *A`: The matrix.
- `const CML_Vectorn *v`: The vector.
- `CML_Vectorn *out`: The vector to store the result.

Returns:
- `void`

Description:
Multiplies a matrix by a vector.

Operation:
Standard matrix-vector multiplication: $\mathbf{out} = \mathbf{Av}$

Algorithm:
For each element i of the vector, $out_i = \sum_{j=0}^{m-1} A_{ij} v_j$.

cml_vectorn_mul_matrixnxm

Parameters:
- `const CML_Vectorn *v`: The vector.
- `const CML_Matrixnxm *A`: The matrix.
- `CML_Vectorn *out`: The vector to store the result.

Returns:
- `void`

Description:
Multiplies a vector by a matrix.

Operation:
Standard vector-matrix multiplication: $\mathbf{out}^T = \mathbf{v}^T\mathbf{A}$

Algorithm:
For each element i of the vector, $out_i = \sum_{j=0}^{n-1} v_j A_{ji}$.

cml_matrixnxn_det

Parameters:
- `const CML_Matrixnxn *A`: The matrix.

Returns:
- `f32`: the determinant of the matrix.

Description:
Returns the determinant of a square matrix.

Operation:
Standard matrix determinant: $\det(\mathbf{A}) = |\mathbf{A}| = \sum_{i=0}^{n-1} (-1)^i A_{0i} \det(\mathbf{A}_{0i})$.

Algorithm:
For 2×2 matrices, return $A_{00}A_{11} - A_{01}A_{10}$. For 3×3 matrices, return the determinant calculated using Sarrus' rule. For 4×4 matrices, return the determinant calculated using the Laplace expansion.

cml_matrixnxn_inv

Parameters:
- `const CML_Matrixnxn *A`: The matrix.
- `CML_Matrixnxn *OUT`: The matrix to store the result.

Returns:
- `CML_Status`

Description:
Inverts a square matrix if possible.

Operation:
Standard matrix inversion: $\mathbf{OUT} = \mathbf{A}^{-1} = \frac{1}{\det(\mathbf{A})}\text{adj}(\mathbf{A})$.

Algorithm:
Calculate the determinant of the matrix. If the determinant is zero, return `CML_ERR_SINGULAR_MATRIX`. Calculate the adjugate of the matrix and the inverse of the determinant. For each element i, j of the matrix, $OUT_{ij} = \frac{1}{\det(\mathbf{A})}\text{adj}(\mathbf{A})_{ij}$.

Return Codes:
- `CML_SUCCESS`: The matrix was successfully inverted.
- `CML_ERR_SINGULAR_MATRIX`: The matrix is singular and cannot be inverted.

cml_matrixnxm_transpose

Parameters:
- `const CML_Matrixnxm *A`: The matrix.
- `CML_Matrixmxn *OUT`: The matrix to store the result.

Returns:
- `void`

Description:
Transposes a matrix.

Operation:
Standard matrix transposition: $\mathbf{OUT} = \mathbf{A}^T$

Algorithm:
For each element i, j of the matrices, $OUT_{ij} = A_{ji}$.

cml_matrixnxn_trace

Parameters:
- `const CML_Matrixnxn *A`: The matrix.

Returns:
- `f32`: the trace of the matrix.

Description:
Returns the trace of a square matrix.

Operation:
Standard matrix trace: $\text{tr}(\mathbf{A}) = \sum_{i=0}^{n-1} A_{ii}$.

Algorithm:
For each diagonal element i, i of the matrix, accumulate the value. Return the accumulated value.

cml_matrixnxm_eq

Parameters:
- `const CML_Matrixnxm *A`: The first matrix.
- `const CML_Matrixnxm *B`: The second matrix.

Returns:

• `b8`: whether the matrices are equal.

Description:
Compares two matrices for equality.

Operation:
Standard matrix equality: $\mathbf{A} = \mathbf{B} \Leftrightarrow \forall i, j, A_{ij} = B_{ij}$.

Algorithm:
For each element i, j of the matrices, if $A_{ij} \neq B_{ij}$, return `false`. Return `true` otherwise.

cml_matrixnxm_debug

Parameters:
• `const CML_Matrixnxm *expected`: The expected matrix.
• `const CML_Matrixnxm *got`: The got matrix.
Returns:
• `char*`

Description:
Returns a `char` string with the debug information of the matrices. Designed to be used directly in `printf`.

Camel also provides a set of functions to create transformation matrices. All of these functions are defined as column-major, like the rest of the module. Having a vector $\mathbf{v} = (x, y, z, w)$, the transformation matrix $\mathbf{T}$ is applied to $\mathbf{v}$ as $\mathbf{v}' = \mathbf{T}\mathbf{v}$. When possible, the follwoing functions will show the 4×4 transformation matrices; however, to obtain the 3×3 transformation matrices, the user can simply ignore the last row and column, and for the 2×2 transformation matrices, the user can ignore the last two rows and columns. When the transformation is different for the different dimensions, the function will provide the separate functions for each dimension that requires it.

The following functions are provided:

cml_matrixnxn_gen_scale

Parameters:
• `f32 x`: The scaling factor in the x axis.
• `f32 y`: The scaling factor in the y axis.
• `f32 z`: The scaling factor in the z axis.
• `CML_Matrixnxn *OUT`: The matrix to store the result.
Returns:
• `void`

Description:
Generates a scaling transformation matrix.

Operation:

$$\mathbf{OUT} = \begin{bmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

cml_matrixnxn_gen_invscale

Parameters:
• `const CML_Matrixnxn *scale`: The scaling transformation matrix.
• `CML_Matrixnxn *OUT`: The matrix to store the result.
Returns:
• `void`

Description:
Generates the inverse of a scaling transformation matrix.

Operation:
$$\mathbf{OUT} = \begin{bmatrix} \frac{1}{A_{00}} & 0 & 0 & 0 \\ 0 & \frac{1}{A_{11}} & 0 & 0 \\ 0 & 0 & \frac{1}{A_{22}} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

cml_matrix2x2_gen_shearx

Parameters:
• `f32 x`: The shear factor.
• `CML_Matrix2x2 *OUT`: The matrix to store the result.
Returns:
• `void`

Description:
Generates a shear transformation matrix in the x axis.

Operation:
$$\mathbf{OUT} = \begin{bmatrix} 1 & x \\ 0 & 1 \end{bmatrix}.$$

cml_matrix2x2_gen_sheary

Parameters:
• `f32 y`: The shear factor.
• `CML_Matrix2x2 *OUT`: The matrix to store the result.
Returns:
• `void`

Description:
Generates a shear transformation matrix in the y axis.

Operation:
$$\mathbf{OUT} = \begin{bmatrix} 1 & 0 \\ y & 1 \end{bmatrix}.$$

cml_matrix2x2_gen_invshear

Parameters:
• `const CML_Matrix2x2 *shear`: The shear transformation matrix.
• `CML_Matrix2x2 *OUT`: The matrix to store the result.
Returns:
• `void`

Description:
Generates the inverse of a shear transformation matrix.

Operation:
$$\mathbf{OUT} = \begin{bmatrix} 1 & -A_{01} \\ -A_{10} & 1 \end{bmatrix}.$$

cml_matrixnxn_gen_shearx

Parameters:
• `f32 y`: The shear factor in the y axis.
• `f32 z`: The shear factor in the z axis.
• `CML_Matrixnxn *OUT`: The matrix to store the result.
Returns:
• `void`

Description:
Generates a shear transformation matrix in the x axis.

Operation:

$$\mathbf{OUT} = \begin{bmatrix} 1 & y & z & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

cml_matrixnxn_gen_sheary

Parameters:
- `f32 x`: The shear factor in the x axis.
- `f32 z`: The shear factor in the z axis.
- `CML_Matrixnxn *OUT`: The matrix to store the result.

Returns:
- `void`

Description:
Generates a shear transformation matrix in the y axis.

Operation:

$$\mathbf{OUT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ x & 1 & z & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

cml_matrixnxn_gen_shearz

Parameters:
- `f32 x`: The shear factor in the x axis.
- `f32 y`: The shear factor in the y axis.
- `CML_Matrixnxn *OUT`: The matrix to store the result.

Returns:
- `void`

Description:
Generates a shear transformation matrix in the z axis.

Operation:

$$\mathbf{OUT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ x & y & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

cml_matrixnxn_gen_invshear

Parameters:
- `const CML_Matrixnxn *shear`: The shear transformation matrix.
- `CML_Matrixnxn *OUT`: The matrix to store the result.

Returns:
- `void`

Description:
Generates the inverse of a shear transformation matrix.

Operation:

$$\mathbf{OUT} = \begin{bmatrix} 1 & -A_{01} & -A_{02} & 0 \\ -A_{10} & 1 & -A_{12} & 0 \\ -A_{20} & -A_{21} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

cml_matrix4x4_gen_translation

Parameters:
• `f32 x`: The translation factor in the x axis.
• `f32 y`: The translation factor in the y axis.
• `f32 z`: The translation factor in the z axis.
• `CML_Matrix4x4 *OUT`: The matrix to store the result.
Returns:
• `void`

Description:
Generates a translation transformation matrix.

Operation:

$$\mathbf{OUT} = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

cml_matrix4x4_gen_invtranslation

Parameters:
• `const CML_Matrix4x4 *translation`: The translation transformation matrix.
• `CML_Matrix4x4 *OUT`: The matrix to store the result.
Returns:
• `void`

Description:
Generates the inverse of a translation transformation matrix.

Operation:

$$\mathbf{OUT} = \begin{bmatrix} 1 & 0 & 0 & -A_{03} \\ 0 & 1 & 0 & -A_{13} \\ 0 & 0 & 1 & -A_{23} \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

cml_matrix2x2_genlh_rotation

Parameters:
• `f32 angle`: The rotation angle in radians.
• `CML_Matrix2x2 *OUT`: The matrix to store the result.
Returns:
• `void`

Description:
Generates a left-handed rotation transformation matrix.

Operation:

$$\mathbf{OUT} = \begin{bmatrix} \cos(-\theta) & -\sin(-\theta) \\ \sin(-\theta) & \cos(-\theta) \end{bmatrix}.$$

cml_matrix2x2_genrh_rotation

Parameters:
• `f32 angle`: The rotation angle in radians.
• `CML_Matrix2x2 *OUT`: The matrix to store the result.
Returns:
• `void`

Description:
Generates a right-handed rotation transformation matrix.

Operation:

$$\mathbf{OUT} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}.$$

cml_matrix2x2_gen_invrotation

Parameters:
• `const CML_Matrix2x2 *rotation`: The rotation transformation matrix.
• `CML_Matrix2x2 *OUT`: The matrix to store the result.
Returns:
• `void`

Description:
Generates the inverse of a rotation transformation matrix.

Operation:
$$\mathbf{OUT} = \begin{bmatrix} A_{00} & -A_{01} \\ -A_{10} & A_{11} \end{bmatrix}.$$

cml_matrixnxn_genlh_rotationx

Parameters:
• `f32 angle`: The rotation angle in radians.
• `CML_Matrixnxn *OUT`: The matrix to store the result.
Returns:
• `void`

Description:
Generates a left-handed rotation transformation matrix around the x axis.

Operation:
$$\mathbf{OUT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(-\theta) & -\sin(-\theta) & 0 \\ 0 & \sin(-\theta) & \cos(-\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

cml_matrixnxn_genrh_rotationx

Parameters:
• `f32 angle`: The rotation angle in radians.
• `CML_Matrixnxn *OUT`: The matrix to store the result.
Returns:
• `void`

Description:
Generates a right-handed rotation transformation matrix around the x axis.

Operation:
$$\mathbf{OUT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

cml_matrixnxn_genlh_rotationy

Parameters:
• `f32 angle`: The rotation angle in radians.
• `CML_Matrixnxn *OUT`: The matrix to store the result.
Returns:
• `void`

Description:
Generates a left-handed rotation transformation matrix around the y axis.

Operation:
$$\mathbf{OUT} = \begin{bmatrix} \cos(-\theta) & 0 & \sin(-\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(-\theta) & 0 & \cos(-\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

cml_matrixnxn_genrh_rotationy

Parameters:
• `f32 angle`: The rotation angle in radians.
• `CML_Matrixnxn *OUT`: The matrix to store the result.
Returns:
• `void`

Description:
Generates a right-handed rotation transformation matrix around the y axis.

Operation:

$$\mathbf{OUT} = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

cml_matrixnxn_genlh_rotationz

Parameters:
• `f32 angle`: The rotation angle in radians.
• `CML_Matrixnxn *OUT`: The matrix to store the result.
Returns:
• `void`

Description:
Generates a left-handed rotation transformation matrix around the z axis.

Operation:

$$\mathbf{OUT} = \begin{bmatrix} \cos(-\theta) & -\sin(-\theta) & 0 & 0 \\ \sin(-\theta) & \cos(-\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

cml_matrixnxn_genrh_rotationz

Parameters:
• `f32 angle`: The rotation angle in radians.
• `CML_Matrixnxn *OUT`: The matrix to store the result.
Returns:
• `void`

Description:
Generates a right-handed rotation transformation matrix around the z axis.

Operation:

$$\mathbf{OUT} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

cml_matrixnxn_genlh_rotation

Parameters:
• `f32 angle`: The rotation angle in radians.
• `const CML_Vector3 *axis`: The axis of rotation.
• `CML_Matrixnxn *OUT`: The matrix to store the result.
Returns:
• `void`

Description:
Generates a left-handed rotation transformation matrix around an arbitrary axis.

Operation:
Having $t = 1 - \cos(-\theta)$, and x, y, z the components of the normalized axis of rotation,

$$\mathbf{OUT} = \begin{bmatrix} txx + \cos(-\theta) & txy - z\sin(-\theta) & txz + y\sin(-\theta) & 0 \\ txy + z\sin(-\theta) & tyy + \cos(-\theta) & tyz - x\sin(-\theta) & 0 \\ txz - y\sin(-\theta) & tyz + x\sin(-\theta) & tzz + \cos(-\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

cml_matrixnxn_genrh_rotation

Parameters:
- `f32 angle`: The rotation angle in radians.
- `const CML_Vector3 *axis`: The axis of rotation.
- `CML_Matrixnxn *OUT`: The matrix to store the result.

Returns:
- `void`

Description:
Generates a right-handed rotation transformation matrix around an arbitrary axis.

Operation:
Having $t = 1 - \cos(\theta)$, and x, y, z the components of the normalized axis of rotation,

$$\mathbf{OUT} = \begin{bmatrix} txx + \cos(\theta) & txy - z\sin(\theta) & txz + y\sin(\theta) & 0 \\ txy + z\sin(\theta) & tyy + \cos(\theta) & tyz - x\sin(\theta) & 0 \\ txz - y\sin(\theta) & tyz + x\sin(\theta) & tzz + \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

cml_matrixnxn_gen_invrotation

Parameters:
- `const CML_Matrixnxn *rotation`: The rotation transformation matrix.
- `CML_Matrixnxn *OUT`: The matrix to store the result.

Returns:
- `void`

Description:
Generates the inverse of a rotation transformation matrix.

Operation:
$\mathbf{OUT} = \mathbf{A}^T$.

7.1.2 Variable-Size Linear Algebra

7.2 Abstract Algebra

8 Calculus

9 Geometry

10 Number Theory

11 Statistics

Part III
Performance

by Sergio Madrid

Every single function in the library has been tested for performance. The following sections provide a detailed analysis of the performance of each function.

12 Core

12.1 Matrix Operations

12.1.1 Matrix Multiplication

Many variations of matrix multiplication were tested, and hypothesis testing was performed to determine the best algorithm for each case. With some optimizations, the best algorithm was found to be the standard matrix multiplication algorithm, with altered loop order to take advantage of the cache. Parallelization was also tested, but the overhead of creating threads was found to be too high for small matrices. The following table shows the performance of the function `cml_matrixnxn_mul_matrixnxn` for different matrix sizes.

Part IV
Examples

by Sergio Madrid