

CAMEL

Manual for Version 0.0

February 13, 2024

Contents

I	Introduction and Setup	3
1	Introduction	3
1.1	Design Goals	3
1.2	Main Features	3
1.3	License	4
2	Installation	4
2.1	Prerequisites	4
2.2	Installation Steps	4
2.2.1	Downloading the Source Code	4
2.3	Verifying the Installation	5
3	Getting Started	5
3.1	Hello World with Camel	5
3.2	Basic Concepts	5
II	Library Reference	6
4	Notation	6
5	Naming Conventions	6
6	Core	6
6.1	Memory Management	6
6.2	Data Structures	6
6.2.1	String	6
6.2.2	Stack	12
6.2.3	Queue	15
6.2.4	Dynamic Array	15
6.2.5	Singly Linked List	20
6.2.6	Doubly Linked List	20
6.2.7	Binary Tree	20
6.2.8	Binary Search Tree	20
6.2.9	Hash Table	20
6.2.10	Graph	20
6.3	Expression Parsing	20
6.4	Arbitrary Precision Arithmetic	20
7	Algebra	20
7.1	Linear Algebra	20
7.1.1	Fixed-Size Linear Algebra	20
7.1.2	Variable-Size Linear Algebra	40
7.2	Abstract Algebra	40
8	Calculus	40
9	Geometry	40
10	Number Theory	40
11	Statistics	40
III	Performance	41
IV	Examples	42
12	Basic Examples	42
13	Advanced Usage	42
14	Exercises	42

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 rigorous 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 and 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 number 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

6.1 Memory Management

6.2 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.

6.2.1 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.

6.2.2 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.

6.2.3 Queue

6.2.4 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.

6.2.5 Singly Linked List**6.2.6 Doubly Linked List****6.2.7 Binary Tree****6.2.8 Binary Search Tree****6.2.9 Hash Table****6.2.10 Graph****6.3 Expression Parsing****6.4 Arbitrary Precision Arithmetic****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_Matrixn xn` 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. Knowing 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** = **v** + **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: $\mathbf{out} = \mathbf{v} + t$

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: $\mathbf{out} = \mathbf{v} - \mathbf{w}$

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_Vector *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_Vector *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_Vector *v`: The vector.
- `CML_Vector *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_Vector *v`: The first vector.
- `const CML_Vector *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 surface 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_matrixm xp`

Parameters:

- `const CML_Matrixnxm *A`: The left matrix.
- `const CML_Matrixm xp *B`: The right matrix.
- `CML_Matrixm xp *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_Vectorm *v`: The vector.
- `CML_Vectorm *out`: The vector to store the result.

Returns:

- `void`

Description:

Multiplies a matrix by a vector.

Operation:

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

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 following 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:

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

[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:

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

[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:

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

[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:

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

[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:

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

[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:

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

[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:

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

[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:

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

[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:

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

[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:

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

[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:

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

[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:

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

[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:

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

[**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:

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

[**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,

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

[**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.

Part IV

Examples

by Sergio Madrid

12 Basic Examples

13 Advanced Usage

14 Exercises