CAMEL

Manual for Version 0.0

February 8, 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 Core 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.
- **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

6.2.1 String

The string data structures 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` struc-

ture.

_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.

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:

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