

---

# **librapid**

***Release v0.6.11***

**Toby Davis**

**Apr 05, 2023**



# CONTENTS

<b>1</b>	<b>What is LibRapid?</b>	<b>1</b>
1.1	Getting Started . . . . .	1
1.1.1	Installation . . . . .	1
1.1.2	Your First Program . . . . .	1
1.1.2.1	Your First Program: Explained . . . . .	2
1.2	CMake Integration . . . . .	3
1.2.1	Installation . . . . .	3
1.2.2	CMake Options . . . . .	3
1.3	Tutorials . . . . .	3
1.4	API Reference . . . . .	3
1.4.1	Arrays, Matrices and Linear Algebra . . . . .	4
1.4.1.1	Array Class Listing . . . . .	4
1.4.2	Vectors . . . . .	7
1.4.2.1	Vector Listing . . . . .	7
1.4.3	Multi-Precision Arithmetic . . . . .	18
1.4.3.1	Multi-Precision Listing . . . . .	19
1.5	Caution . . . . .	19
1.5.1	Array Referencing Issues . . . . .	19
<b>2</b>	<b>Why use LibRapid?</b>	<b>21</b>
2.1	A Small Example . . . . .	21
<b>3</b>	<b>Current Development Stage</b>	<b>23</b>
<b>4</b>	<b>Roadmap</b>	<b>25</b>
<b>5</b>	<b>Licencing</b>	<b>27</b>
	<b>Index</b>	<b>29</b>



## WHAT IS LIBRAPID?

LibRapid is a high performance Array library for C++. It supports a wide range of calculations and operations, useful classes and functions, and even supports CUDA! It uses SIMD instructions and multithreading where possible, achieving incredible performance on all operations.

Getting Started Write your first program with LibRapid.

CMake Integration See all available CMake options to make the most of LibRapid's features.

API Reference View LibRapid's API and documentation.

Tutorials Learn how to use some of LibRapid's features.

Performance and Benchmarks View LibRapid's benchmark results.

Caution **Learn about potential issues that may occur with LibRapid**

## 1.1 Getting Started

### 1.1.1 Installation

To use LibRapid in your CMake project, first clone the project: `git clone --recursive https://github.com/LibRapid/libRapid.git`

Next, add the following to your `CMakeLists.txt`

```
add_subdirectory(librapid)
target_link_libraries(yourTarget PUBLIC librapid)
```

That's it! LibRapid will now be compiled and linked with your project!

### 1.1.2 Your First Program

```
1 #include <librapid>
2 namespace lrc = librapid;
3
4 int main() {
5     lrc::Array<int> myFirstArray = lrc::fromData({{1, 2, 3, 4},
6                                                  {5, 6, 7, 8}});
7 }
```

(continues on next page)

(continued from previous page)

```

8   lrc::Array<int> mySecondArray = lrc::fromData({{8, 7, 6, 5},
9                                           {4, 3, 2, 1}});
10
11   fmt::print("{}\n\n", myFirstArray);
12   fmt::print("{}\n", mySecondArray);
13
14   fmt::print("Sum of two Arrays:\n{}\n", myFirstArray + mySecondArray);
15   fmt::print("First row of my Array: {}\n", myFirstArray[0]);
16   fmt::print("First row of my Array: {}\n", myFirstArray[0] + mySecondArray[1]);
17
18   return 0;
19 }

```

### 1.1.2.1 Your First Program: Explained

```

1  #include <librapid>
2  namespace lrc = librapid;

```

The first line here allows you to use all of LibRapid's features in your file. The second line isn't required, but it makes your code shorter and quicker to type.

```

5  lrc::Array<int> myFirstArray = lrc::fromData({{1, 2, 3, 4},
6                                           {5, 6, 7, 8}});
7
8  lrc::Array<int> mySecondArray = lrc::fromData({{8, 7, 6, 5},
9                                           {4, 3, 2, 1}});

```

These lines create two Array instances from a list of values. Both arrays are 2-dimensional and have 2 rows and 4 columns.

```

11  fmt::print("{}\n\n", myFirstArray);
12  fmt::print("{}\n", mySecondArray);

```

Here, we print out the Arrays we just created. Try changing the numbers to see how the formatting changes!

```

14  fmt::print("Sum of two Arrays:\n{}\n", myFirstArray + mySecondArray);

```

This line performs a simple arithmetic operation on our Arrays and prints the result.

```

15  fmt::print("First row of my Array: {}\n", myFirstArray[0]);
16  fmt::print("First row of my Array: {}\n", myFirstArray[0] + mySecondArray[1]);

```

As you can see, Array instances can be indexed with the traditional square bracket notation. This means you can easily access sub-arrays of higher-dimensional array objects.

## 1.2 CMake Integration

### 1.2.1 Installation

Link librapid like any other CMake library:

Clone the repository: `git clone --recursive https://github.com/LibRapid/libRapid.git`

Add the following to your `CMakeLists.txt`

```
add_subdirectory(librapid)
target_link_libraries(yourTarget PUBLIC librapid)
```

### 1.2.2 CMake Options

When using LibRapid in your CMake project, the following options are configurable:

- `LIBRAPID_BUILD_EXAMPLES` => `OFF` (Build examples?)
- `LIBRAPID_BUILD_TESTS` => `OFF` (Build tests?)
- `LIBRAPID_STRICT` => `OFF` (Force warnings into errors?)
- `LIBRAPID_QUIET` => `OFF` (Disable warnings)
- `LIBRAPID_GET_BLAS` => `OFF` (Clone a prebuilt version of OpenBLAS?)
- `LIBRAPID_USE_CUDA` => `ON` (Automatically search for CUDA?)
- `LIBRAPID_USE_OMP` => `ON` (Automatically search for OpenMP?)
- `LIBRAPID_USE_MULTIPREC` => `OFF` (Include multiprecision library – more on this elsewhere in documentation)
- `LIBRAPID_OPTIMISE_SMALL_ARRAYS` => `OFF` (Optimise small arrays?)
- `LIBRAPID_FAST_MATH` => `OFF` (Use potentially less accurate operations to increase performance)

## 1.3 Tutorials

## 1.4 API Reference

---

**Important:** This list is **INCOMPLETE!** If you think something is missing, try searching for it first. If you still can't find it, please open an issue on the [LibRapid GitHub repository](#).

---

Arrays, Matrices and Linear Algebra   Multidimensional arrays, matrices, linear algebra and more.

Vectors   Fixed-size vectors and supported operations.

Complex Numbers   Complex numbers and their operations.

Mathematics   General mathematical operations that work on most data types.

Multi-Precision Arithmetic   Arbitrary-precision integers, floating points and rationals.

Utilities   Utility functions and classes to support development.

## 1.4.1 Arrays, Matrices and Linear Algebra

The main feature of LibRapid is its high-performance array library. It provides an intuitive way to perform highly efficient operations on arrays and matrices in C++.

### 1.4.1.1 Array Class Listing

```
template<typename ShapeType_, typename StorageType_>
```

```
class ArrayContainer
```

#### Public Types

```
using StorageType = StorageType_
```

```
using ShapeType = ShapeType_
```

```
using StrideType = Stride<size_t, 32>
```

```
using SizeType = typename ShapeType::SizeType
```

```
using Scalar = typename StorageType::Scalar
```

```
using Packet = typename typetraits::TypeInfo<Scalar>::Packet
```

#### Public Functions

```
ArrayContainer()
```

Default constructor.

```
template<typename T>  
ArrayContainer(const std::initializer_list<T> &data)
```

```
template<typename T>  
explicit ArrayContainer(const std::vector<T> &data)
```

```
explicit ArrayContainer(const ShapeType &shape)
```

Constructs an array container from a shape

**Parameters** **shape** – The shape of the array container

```
ArrayContainer(const ShapeType &shape, const Scalar &value)
```

Create an array container from a shape and a scalar value. The scalar value represents the value the memory is initialized with.

**Parameters**

- **shape** – The shape of the array container
- **value** – The value to initialize the memory with



explicit **ArrayContainer**(const Scalar &value)

Allows for a fixed-size array to be constructed with a fill value

**Parameters** **value** – The value to fill the array with

explicit **ArrayContainer**(ShapeType &&shape)

Construct an array container from a shape, which is moved, not copied.

**Parameters** **shape** – The shape of the array container

**ArrayContainer**(const ArrayContainer &other) = default

Construct an array container from another array container.

**Parameters** **other** – The array container to copy.

**ArrayContainer**(ArrayContainer &&other) noexcept = default

Construct an array container from a temporary array container.

**Parameters** **other** – The array container to move.

```
template<typename desc, typename Functor_, typename...
Args> ArrayContainer (const detail::Function< desc, Functor_, Args...
> &function) LIBRAPID_RELEASE_NOEXCEPT
```

Construct an array container from a function object. This will assign the result of the function to the array container, evaluating it accordingly.

#### Template Parameters

- **desc** – The assignment descriptor
- **Functor\_** – The function type
- **Args** – The argument types of the function

**Parameters** **function** – The function to assign

**ArrayContainer &operator=**(const ArrayContainer &other) = default

Assign an array container to this array container.

**Parameters** **other** – The array container to copy.

**Returns** A reference to this array container.

**ArrayContainer &operator=**(const Scalar &value)

**ArrayContainer &operator=**(ArrayContainer &&other) noexcept = default

Assign a temporary array container to this array container.

**Parameters** **other** – The array container to move.

**Returns** A reference to this array container.

```
template<typename desc, typename Functor_, typename ...Args>
ArrayContainer &operator= (const detail::Function<desc, Functor_, Args...> &function)
```

Assign a function object to this array container. This will assign the result of the function to the array container, evaluating it accordingly.

#### Template Parameters

- **Functor\_** – The function type
- **Args** – The argument types of the function

**Parameters** **function** – The function to assign

**Returns** A reference to this array container.

```
template<typename TransposeType>  
ArrayContainer &operator=(const Transpose<TransposeType> &transpose)
```

```
template<typename T>  
detail::CommaInitializer<ArrayContainer> operator<<(const T &value)
```

Allow ArrayContainer objects to be initialized with a comma separated list of values. This makes use of the CommaInitializer class

**Template Parameters** **T** – The type of the values

**Parameters** **value** – The value to set in the Array object

**Returns** The comma initializer object

```
auto operator[](int64_t index) const
```

Access a sub-array of this ArrayContainer instance. The sub-array will reference the same memory as this ArrayContainer instance.

**See also:**

ArrayView

**Parameters** **index** – The index of the sub-array

**Returns** A reference to the sub-array (ArrayView)

```
auto operator[](int64_t index)
```

```
Scalar get() const
```

```
ShapeType::SizeType ndim() const noexcept
```

Return the number of dimensions of the ArrayContainer object

**Returns** Number of dimensions of the ArrayContainer

```
const ShapeType &shape() const noexcept
```

Return the shape of the array container. This is an immutable reference.

**Returns** The shape of the array container.

```
const StorageType &storage() const noexcept
```

Return the StorageType object of the ArrayContainer

**Returns** The StorageType object of the ArrayContainer

```
StorageType &storage() noexcept
```

Return the StorageType object of the ArrayContainer

**Returns** The StorageType object of the ArrayContainer

```
Packet packet(size_t index) const
```

Return a Packet object from the array's storage at a specific index.

**Parameters** **index** – The index to get the packet from

**Returns** A Packet object from the array's storage at a specific index

Scalar **scalar**(size\_t index) const

Return a Scalar from the array's storage at a specific index.

**Parameters** **index** – The index to get the scalar from

**Returns** A Scalar from the array's storage at a specific index

void **writePacket**(size\_t index, const Packet &value)

Write a Packet object to the array's storage at a specific index

**Parameters**

- **index** – The index to write the packet to
- **value** – The value to write to the array's storage

void **write**(size\_t index, const Scalar &value)

Write a Scalar to the array's storage at a specific index

**Parameters**

- **index** – The index to write the scalar to
- **value** – The value to write to the array's storage

std::string **str**(const std::string &format = "{}") const

Return a string representation of the array container \format The format to use for the string representation

**Returns** A string representation of the array container

template<typename **desc**, typename **Functor\_**, typename ...**Args**>

auto **operator**=(const detail::Function<desc, Functor\_, Args...> &function) -> ArrayContainer&

template<typename **TransposeType**>

auto **operator**=(const Transpose<TransposeType> &transpose) -> ArrayContainer&

template<typename **T**>

auto **operator**<<(const T &value) -> detail::CommaInitializer<ArrayContainer>

## 1.4.2 Vectors

LibRapid provides a highly optimised fixed-size vector library which supports all primitive types as well as user-defined ones (assuming they implement the required operations).

### 1.4.2.1 Vector Listing

template<typename **Scalar**, int64\_t **Dims** = 3>

class **GenericVector**

The implementation for the Vector class. It is capable of representing an n-dimensional vector with any data type and storage type. By default, the storage type is a Vc Vector, but can be replaced with custom types for different functionality.

**Template Parameters**

- **Scalar** – The type of each element of the vector
- **Dims** – The number of dimensions of the vector
- **StorageType** – The type of the storage for the vector

## Public Types

using **StorageType** = Scalar[Dims]

## Public Functions

**GenericVector**() = default

Default constructor.

explicit **GenericVector**(const StorageType &arr)

Create a Vector object from a StorageType object

**Parameters** **arr** – The StorageType object to construct from

template<typename **S**, int64\_t **D**>

**GenericVector**(const GenericVector<**S**, **D**> &other)

Construct a Vector from another Vector with potentially different dimensions, scalar type and storage type

### Template Parameters

- **S** – The scalar type of the other vector
- **D** – The number of dimensions of

**Parameters** **other** – The other vector to construct from

template<typename ...**Args**, std::enable\_if\_t<sizeof...(Args) == Dims, int> = 0>

**GenericVector**(**Args**... args)

Construct a Vector object from n values, where n is the number of dimensions of the vector

**Template Parameters** **Args** – Parameter pack template type

**Parameters** **args** – The values to construct the vector from

template<typename ...**Args**, int64\_t **size** = sizeof...(Args), typename std::enable\_if\_t<size != Dims, int> = 0>

**GenericVector**(**Args**... args)

Construct a Vector object from an arbitrary number of arguments. See other vector constructors for more information

### Template Parameters

- **Args** – Parameter pack template type
- **size** – Number of arguments passed

**Parameters** **args** – Values

template<typename **T**, std::enable\_if\_t<std::is\_convertible\_v<**T**, Scalar>, int> = 0>

**GenericVector**(const std::initializer\_list<**T**> &list)

Construct a Vector object from an std::initializer\_list

**Template Parameters** **T** – The type of each element of the initializer list

**Parameters** **list** – The initializer list to construct from

**GenericVector**(const GenericVector &other) = default

Create a Vector from another vector instance

**Parameters** **other** – Vector to copy values from

**GenericVector**(GenericVector &&other) noexcept = default

Move constructor for Vector objects

**Parameters** **other** – Vector to move

GenericVector &**operator**=(const GenericVector &other) = default

Assignment operator for Vector objects

**Parameters** **other** – Vector to copy values from

**Returns** Reference to this

GenericVector &**operator**=(GenericVector &&other) noexcept = default

Assignment move constructor for Vector objects

**Parameters** **other** – Vector to move

**Returns** Reference to this

const Scalar &**operator**[](int64\_t index) const

Access a specific element of the vector

**Parameters** **index** – The index of the element to access

**Returns** Reference to the element

Scalar &**operator**[](int64\_t index)

Access a specific element of the vector

**Parameters** **index** – The index of the element to access

**Returns** Reference to the element

template<typename **T**, int64\_t **d**>

GenericVector &**operator**+=(const GenericVector<T, d> &other)

Add a vector to this vector, element-by-element

**Parameters** **other** – The vector to add

**Returns** Reference to this

template<typename **T**, int64\_t **d**>

GenericVector &**operator**-= (const GenericVector<T, d> &other)

Subtract a vector from this vector, element-by-element

**Parameters** **other** – The vector to subtract

**Returns** Reference to this

template<typename **T**, int64\_t **d**>

GenericVector &**operator**\*=(const GenericVector<T, d> &other)

Multiply this vector by another vector, element-by-element

**Parameters** **other** – The vector to multiply by

**Returns** Reference to this

template<typename **T**, int64\_t **d**>

GenericVector &**operator**/=(const GenericVector<T, d> &other)

Divide this vector by another vector, element-by-element

**Parameters** **other** – The vector to divide by

**Returns** Reference to this

```
template<typename T, std::enable_if_t<std::is_convertible_v<T, Scalar>, int> = 0>
GenericVector &operator+=(const T &value)
```

Add a scalar to this vector, element-by-element

**Parameters** **other** – The scalar to add

**Returns** Reference to this

```
template<typename T, std::enable_if_t<std::is_convertible_v<T, Scalar>, int> = 0>
GenericVector &operator-=(const T &value)
```

Subtract a scalar from this vector, element-by-element

**Parameters** **other** – The scalar to subtract

**Returns** Reference to this

```
template<typename T, std::enable_if_t<std::is_convertible_v<T, Scalar>, int> = 0>
GenericVector &operator*=(const T &value)
```

Multiply this vector by a scalar, element-by-element

**Parameters** **other** – The scalar to multiply by

**Returns** Reference to this

```
template<typename T, std::enable_if_t<std::is_convertible_v<T, Scalar>, int> = 0>
GenericVector &operator/=(const T &value)
```

Divide this vector by a scalar, element-by-element

**Parameters** **other** – The scalar to divide by

**Returns** Reference to this

```
GenericVector operator-() const
```

Negate this vector

**Returns** Vector with all elements negated

```
template<typename T, int64_t d>
GenericVector cmp(const GenericVector<T, d> &other, const char *mode) const
```

Compare two vectors for equality. Available modes are:

- "eq" - Check for equality
- "ne" - Check for inequality
- "lt" - Check if each element is less than the corresponding element in the other
- "le" - Check if each element is less than or equal to the corresponding element in the other
- "gt" - Check if each element is greater than the corresponding element in the other
- "ge" - Check if each element is greater than or equal to the corresponding element in the other

**Parameters**

- **other** – The vector to compare to
- **mode** – The comparison mode

**Returns** Vector with each element set to 1 if the comparison is true, 0 otherwise

```
template<typename T>
```

GenericVector **cmp**(const T &value, const char \*mode) const

Compare a vector and a scalar for equality. Available modes are:

- "eq" - Check for equality
- "ne" - Check for inequality
- "lt" - Check if each element is less than the scalar
- "le" - Check if each element is less than or equal to the scalar
- "gt" - Check if each element is greater than the scalar
- "ge" - Check if each element is greater than or equal to the scalar

**Parameters**

- **value** – The scalar to compare to
- **mode** – The comparison mode

**Returns** Vector with each element set to 1 if the comparison is true, 0 otherwise

template<typename T, int64\_t d>

GenericVector **operator**<(const GenericVector<T, d> &other) const

Equivalent to calling `cmp(other, "lt")`

**See also:**

[\*cmp\(\)\*](#)

**Parameters** **other** – The vector to compare to

**Returns** See [\*cmp\(\)\*](#)

template<typename T, int64\_t d>

GenericVector **operator**<=(const GenericVector<T, d> &other) const

Equivalent to calling `cmp(other, "le")`

**See also:**

[\*cmp\(\)\*](#)

**Parameters** **other** – The vector to compare to

**Returns** See [\*cmp\(\)\*](#)

template<typename T, int64\_t d>

GenericVector **operator**>(const GenericVector<T, d> &other) const

Equivalent to calling `cmp(other, "gt")`

**See also:**

[\*cmp\(\)\*](#)

**Parameters** **other** – The vector to compare to

**Returns** See [\*cmp\(\)\*](#)

template<typename T, int64\_t d>

GenericVector **operator>=**(const GenericVector<T, d> &other) const

Equivalent to calling `cmp(other, "ge")`

**See also:**

[\*cmp\(\)\*](#)

**Parameters** **other** – The vector to compare to

**Returns** See [\*cmp\(\)\*](#)

template<typename **T**, int64\_t **d**>

GenericVector **operator==**(const GenericVector<T, d> &other) const

Equivalent to calling `cmp(other, "eq")`

**See also:**

[\*cmp\(\)\*](#)

**Parameters** **other** – The vector to compare to

**Returns** See [\*cmp\(\)\*](#)

template<typename **T**, int64\_t **d**>

GenericVector **operator!=**(const GenericVector<T, d> &other) const

Equivalent to calling `cmp(other, "ne")`

**See also:**

[\*cmp\(\)\*](#)

**Parameters** **other** – The vector to compare to

**Returns** See [\*cmp\(\)\*](#)

template<typename **T**, std::enable\_if\_t<std::is\_convertible\_v<T, Scalar>, int> = 0>

GenericVector **operator<**(const T &other) const

Equivalent to calling `cmp(other, "lt")`

**See also:**

[\*cmp\(\)\*](#)

**Parameters** **value** – The scalar to compare to

**Returns** See [\*cmp\(\)\*](#)

template<typename **T**, std::enable\_if\_t<std::is\_convertible\_v<T, Scalar>, int> = 0>

GenericVector **operator<=**(const T &other) const

Equivalent to calling `cmp(other, "le")`

**See also:**

[\*cmp\(\)\*](#)

**Parameters** **value** – The scalar to compare to

**Returns** See [\*cmp\(\)\*](#)

template<typename **T**, std::enable\_if\_t<std::is\_convertible\_v<T, Scalar>, int> = 0>



GenericVector **operator>**(const T &other) const

Equivalent to calling `cmp(other, "gt")`

**See also:**

[\*cmp\(\)\*](#)

**Parameters** **value** – The scalar to compare to

**Returns** See [\*cmp\(\)\*](#)

template<typename T, std::enable\_if\_t<std::is\_convertible\_v<T, Scalar>, int> = 0>

GenericVector **operator>=**(const T &other) const

Equivalent to calling `cmp(other, "ge")`

**See also:**

[\*cmp\(\)\*](#)

**Parameters** **value** – The scalar to compare to

**Returns** See [\*cmp\(\)\*](#)

template<typename T, std::enable\_if\_t<std::is\_convertible\_v<T, Scalar>, int> = 0>

GenericVector **operator==**(const T &other) const

Equivalent to calling `cmp(other, "eq")`

**See also:**

[\*cmp\(\)\*](#)

**Parameters** **value** – The scalar to compare to

**Returns** See [\*cmp\(\)\*](#)

template<typename T, std::enable\_if\_t<std::is\_convertible\_v<T, Scalar>, int> = 0>

GenericVector **operator!=**(const T &other) const

Equivalent to calling `cmp(other, "ne")`

**See also:**

[\*cmp\(\)\*](#)

**Parameters** **value** – The scalar to compare to

**Returns** See [\*cmp\(\)\*](#)

Scalar **mag2**() const

Calculate the magnitude of this vector squared

**Returns** The magnitude squared

Scalar **mag**() const

Calculate the magnitude of this vector

**Returns** The magnitude

inline Scalar **invMag**() const

Calculate 1/mag(this)

**Returns** 1/mag(this)

GenericVector **norm**() const

Calculate the normalized version of this vector

**Returns** The normalized vector

Scalar **dot**(const GenericVector &other) const

Calculate the dot product of this vector and another

**Parameters** **other** – The other vector

**Returns** The dot product

GenericVector **cross**(const GenericVector &other) const

Calculate the cross product of this vector and another

**Parameters** **other** – The other vector

**Returns** The cross product

GenericVector **proj**(const GenericVector &other) const

Project this vector onto another vector

**Parameters** **other** – The vector to project onto

**Returns** The projection of this vector onto the other vector

explicit **operator bool**() const

Cast this vector to a boolean. This is equivalent to calling *mag2()* != 0

**Returns** True if the magnitude of this vector is not 0, false otherwise

GenericVector<Scalar, 2> **xy**() const

Add two Vector objects and return the result

**Template Parameters**

- **Scalar** – The type of the scalar
- **Dims** – The number of dimensions
- **StorageType** – The type of the storage

**Parameters**

- **lhs** – The left hand side of the addition
- **rhs** – The right hand side of the addition

**Returns** The result of the addition

GenericVector<Scalar, 2> **yx**() const

GenericVector<Scalar, 2> **xz**() const

GenericVector<Scalar, 2> **zx**() const

GenericVector<Scalar, 2> **yz**() const

GenericVector<Scalar, 2> **zy**() const

GenericVector<Scalar, 3> **xyz**() const  
 GenericVector<Scalar, 3> **xzy**() const  
 GenericVector<Scalar, 3> **yxz**() const  
 GenericVector<Scalar, 3> **yzx**() const  
 GenericVector<Scalar, 3> **zxy**() const  
 GenericVector<Scalar, 3> **zyx**() const  
 GenericVector<Scalar, 3> **xyw**() const  
 GenericVector<Scalar, 3> **xwy**() const  
 GenericVector<Scalar, 3> **yxw**() const  
 GenericVector<Scalar, 3> **ywx**() const  
 GenericVector<Scalar, 3> **wxy**() const  
 GenericVector<Scalar, 3> **wyx**() const  
 GenericVector<Scalar, 3> **xzw**() const  
 GenericVector<Scalar, 3> **xwz**() const  
 GenericVector<Scalar, 3> **zxw**() const  
 GenericVector<Scalar, 3> **zwx**() const  
 GenericVector<Scalar, 3> **wxz**() const  
 GenericVector<Scalar, 3> **wzx**() const  
 GenericVector<Scalar, 3> **yzw**() const  
 GenericVector<Scalar, 3> **ywz**() const  
 GenericVector<Scalar, 3> **zyw**() const  
 GenericVector<Scalar, 3> **zwy**() const  
 GenericVector<Scalar, 3> **wyz**() const  
 GenericVector<Scalar, 3> **wzy**() const  
 GenericVector<Scalar, 4> **xyzw**() const  
 GenericVector<Scalar, 4> **xywz**() const  
 GenericVector<Scalar, 4> **xzyw**() const  
 GenericVector<Scalar, 4> **xzwy**() const  
 GenericVector<Scalar, 4> **xwyz**() const  
 GenericVector<Scalar, 4> **xwzy**() const  
 GenericVector<Scalar, 4> **yxzw**() const

GenericVector<Scalar, 4> **y<sub>xwz</sub>**() const

GenericVector<Scalar, 4> **yz<sub>xw</sub>**() const

GenericVector<Scalar, 4> **yz<sub>wx</sub>**() const

GenericVector<Scalar, 4> **y<sub>wxz</sub>**() const

GenericVector<Scalar, 4> **y<sub>wzx</sub>**() const

GenericVector<Scalar, 4> **z<sub>xyw</sub>**() const

GenericVector<Scalar, 4> **z<sub>xwy</sub>**() const

GenericVector<Scalar, 4> **z<sub>ywx</sub>**() const

GenericVector<Scalar, 4> **z<sub>ywx</sub>**() const

GenericVector<Scalar, 4> **z<sub>wxy</sub>**() const

GenericVector<Scalar, 4> **z<sub>wyx</sub>**() const

GenericVector<Scalar, 4> **w<sub>xyz</sub>**() const

GenericVector<Scalar, 4> **w<sub>xzy</sub>**() const

GenericVector<Scalar, 4> **w<sub>yxz</sub>**() const

GenericVector<Scalar, 4> **w<sub>yzx</sub>**() const

GenericVector<Scalar, 4> **w<sub>zxy</sub>**() const

GenericVector<Scalar, 4> **w<sub>zyx</sub>**() const

const StorageType &**data**() const

Return the underlying storage type

**Returns** The underlying storage type

StorageType &**data**()

Return the underlying storage type

**Returns** The underlying storage type

Scalar **x**() const

Access the x component of this vector

**Returns** The x component of this vector

Scalar **y**() const

Access the y component of this vector

**Returns** The y component of this vector

Scalar **z**() const

Access the z component of this vector

**Returns** The z component of this vector

Scalar **w()** const

Access the w component of this vector

**Returns** The w component of this vector

void **x**(Scalar val)

Set the x component of this vector

**Parameters** **val** – The new value of the x component

void **y**(Scalar val)

Set the y component of this vector

**Parameters** **val** – The new value of the y component

void **z**(Scalar val)

Set the z component of this vector

**Parameters** **val** – The new value of the z component

void **w**(Scalar val)

Set the w component of this vector

**Parameters** **val** – The new value of the w component

std::string **str**(const std::string &formatString = "{}") const

Convert a vector into a string representation &#8212; “(x, y, z, w, ...)”

**Parameters** **formatString** – The format string to use for each component

**Returns** A string representation of this vector

template<typename **T**, int64\_t **d**>

auto **operator**+=(const GenericVector<T, d> &other) -> GenericVector&

template<typename **T**, int64\_t **d**>

auto **operator**+=(const GenericVector<T, d> &other) -> GenericVector&

template<typename **T**, int64\_t **d**>

auto **operator**+=(const GenericVector<T, d> &other) -> GenericVector&

template<typename **T**, int64\_t **d**>

auto **operator**+=(const GenericVector<T, d> &other) -> GenericVector&

template<typename **T**, std::enable\_if\_t<std::is\_convertible\_v<T, Scalar>, int>>

auto **operator**+=(const T &value) -> GenericVector&

template<typename **T**, std::enable\_if\_t<std::is\_convertible\_v<T, Scalar>, int>>

auto **operator**+=(const T &value) -> GenericVector&

template<typename **T**, std::enable\_if\_t<std::is\_convertible\_v<T, Scalar>, int>>

auto **operator**+=(const T &value) -> GenericVector&

template<typename **T**, std::enable\_if\_t<std::is\_convertible\_v<T, Scalar>, int>>

auto **operator**+=(const T &value) -> GenericVector&

template<typename **T**, int64\_t **d**>

auto **cmp**(const GenericVector<T, d> &other, const char \*mode) const -> GenericVector

template<typename **T**>

```
auto cmp(const T &value, const char *mode) const -> GenericVector

template<typename T, int64_t d>
auto operator<(const GenericVector<T, d> &other) const -> GenericVector

template<typename T, int64_t d>
auto operator<=(const GenericVector<T, d> &other) const -> GenericVector

template<typename T, int64_t d>
auto operator>(const GenericVector<T, d> &other) const -> GenericVector

template<typename T, int64_t d>
auto operator>=(const GenericVector<T, d> &other) const -> GenericVector

template<typename T, int64_t d>
auto operator==(const GenericVector<T, d> &other) const -> GenericVector

template<typename T, int64_t d>
auto operator!=(const GenericVector<T, d> &other) const -> GenericVector

template<typename T, std::enable_if_t<std::is_convertible_v<T, Scalar>, int>>
auto operator<(const T &other) const -> GenericVector

template<typename T, std::enable_if_t<std::is_convertible_v<T, Scalar>, int>>
auto operator<=(const T &other) const -> GenericVector

template<typename T, std::enable_if_t<std::is_convertible_v<T, Scalar>, int>>
auto operator>(const T &other) const -> GenericVector

template<typename T, std::enable_if_t<std::is_convertible_v<T, Scalar>, int>>
auto operator>=(const T &other) const -> GenericVector

template<typename T, std::enable_if_t<std::is_convertible_v<T, Scalar>, int>>
auto operator==(const T &other) const -> GenericVector

template<typename T, std::enable_if_t<std::is_convertible_v<T, Scalar>, int>>
auto operator!=(const T &other) const -> GenericVector
```

## Protected Attributes

```
StorageType m_data = {}
```

### 1.4.3 Multi-Precision Arithmetic

LibRapid has support for [MPIR](#) and [MPFR](#), which support arbitrary-precision integers, floating points and rationals.

We provide a simple wrapper around these libraries, enabling all mathematical operations to be performed on these data types – you don’t even need to use a different function name!

---

### 1.4.3.1 Multi-Precision Listing

---

**Warning:** doxygenclass: Cannot find class “librapid::mpz” in doxygen xml output for project “librapid” from directory: ../xml

---

**Warning:** doxygenclass: Cannot find class “librapid::mpq” in doxygen xml output for project “librapid” from directory: ../xml

---

**Warning:** doxygenclass: Cannot find class “librapid::mpf” in doxygen xml output for project “librapid” from directory: ../xml

---

**Warning:** doxygenclass: Cannot find class “librapid::mpfr” in doxygen xml output for project “librapid” from directory: ../xml

---

## 1.5 Caution

**Warning:** LibRapid developers had to make certain decisions regarding the underlying data layout used by the library. We made these decisions with the best interests of the library in mind, and while they may improve performance or usability, they may also incur adverse side effects.

While the developers of LibRapid may not be aware of all the side effects of their design choices, we have done our best to identify and justify those we know of.

### 1.5.1 Array Referencing Issues

LibRapid uses lazy evaluation to reduce the number of intermediate variables and copies required for any given operation, significantly improving performance. A side effect of this is that combined operations store references to Array objects.

As a result, if any of the referenced Array instances go out of scope before the lazy object is evaluated, an invalid memory location will be accessed, incurring a segmentation fault.

The easiest fix for this is to make sure you evaluate temporary results in time, though this is easier said than done. LibRapid aims to identify when a lazy object is using an invalid value and notify the user, but this will not work in all cases.

The code below will cause a segmentation fault since `testArray` will go out of scope upon returning from the function while the returned object contains two references to the array.

```
1  /* References invalid memory
2  vvvv */
3  auto doesThisBreak() {
4      lrc::Array<float> testArray(lrc::Shape({3, 3}));
5      testArray << 1, 2, 3, 4, 5, 6, 7, 8, 9;
6      return testArray + testArray;
7  }
```

```
1  /* Changed
2  -----vvv----- */
3  lrc::Array<float> doesThisBreak() {
4      lrc::Array<float> testArray(lrc::Shape({3, 3}));
5      testArray << 1, 2, 3, 4, 5, 6, 7, 8, 9;
6      return testArray + testArray;
7  }
```



## WHY USE LIBRAPID?

LibRapid aims to provide a cohesive ecosystem of functions that interoperate with each other, allowing for faster development and faster code execution.

For example, LibRapid implements a wide range of mathematical functions which can operate on primitive types, multi-precision types, vectors, and arrays. Due to the way these functions are implemented, a single function call can be used to operate on all of these types, reducing code duplication.

### 2.1 A Small Example

To prove the point made above, let's take a look at a simple example. Here, we have a function that maps a value from one range to another:

```
1 // Standard "double" implementation
2 double map(double val, double start1, double stop1, double start2, double stop2) {
3     return start2 + (stop2 - start2) * ((val - start1) / (stop1 - start1));
4 }
5
6 // map(0.5, 0, 1, 0, 10) = 5
7 // map(10, 0, 100, 0, 1) = 0.1
8 // map(5, 0, 10, 0, 100) = 50
```

This function will accept integers, floats and doubles, but nothing else can be used, limiting its functionality.

Of course, this could be templated to accept other types, but if you passed a `std::vector<double>` to this function, for example, you'd have to create an edge case to support it. **This is where LibRapid comes in.**

Look at the function below:

```
1 // An extremely versatile mapping function (used within LibRapid!)
2 template<typename V, typename B1, typename E1, typename B2, typename E2>
3 V map(V val, B1 start1, E1 stop1, B2 start2, E2 stop2) {
4     return start2 + (stop2 - start2) * ((val - start1) / (stop1 - start1));
5 }
```

This may look excessively complicated with that many template parameters, but you don't actually need all of those! This just gives the greatest flexibility. This function can be called with **almost any LibRapid type!**

```
1 map(0.5, 0, 1, 0, 100); // . . . . . | 50
2 map(lrc::Vec2d(0.2, 0.8), 0, 1, 0, 100); // . . . . . | (20, 80)
3 map(0.5, 0, 1, 0, lrc::Vec2d(100, 200)); // . . . . . | (50, 100)
4 map(lrc::Vec2d(-1, -2), 1, 0, lrc::Vec2d(100, 300)); // . | (75, 250)
```

(continues on next page)

(continued from previous page)

```

5
6 // -----
7
8 using namespace lrc::literals; // To use "_f" suffix
9                               // (also requires multiprecision to be enabled)
10 // "0.5"_f in this case creates a multiprecision float :)
11 map("0.5"_f, "0"_f, "1"_f, "0"_f, "100"_f); // . . . . . | 50.0000000000000000
12
13 // -----
14
15 auto val      = lrc::fromData<float>({{1, 2}, {3, 4}});
16 auto start1   = lrc::fromData<float>({{0, 0}, {0, 0}});
17 auto end1     = lrc::fromData<float>({{10, 10}, {10, 10}});
18 auto start2   = lrc::fromData<float>({{0, 0}, {0, 0}});
19 auto end2     = lrc::fromData<float>({{100, 100}, {100, 100}});
20
21 fmt::print("{}\n", lrc::map(val, start1, end1, start2, end2));
22 // [[10 20]
23 //  [30 40]]

```

Note: LibRapid's built-in map function has even more functionality! See the [Map Function](#) details.

This is just one example of how LibRapid's functions can be used to make your code more concise and more efficient, and hopefully it's clear to see how powerful this could be when working with more complex functions and types.

## CURRENT DEVELOPMENT STAGE

At the current point in time, LibRapid C++ is under rapid development by me ([Pencilcaseman](#)).

I am currently doing my A-Levels and do not have time to work on the library as much as I would like, so if you or someone you know might be willing to support the development of the library, feel free to create a pull request or chat to us on [Discord](#). Any help is greatly appreciated!



## ROADMAP

The [Roadmap](#) is a rough outline of what I want to get implemented in the library and by what point, but **please don't count on features being implemented quickly** – I can't promise I'll have the time to implement everything as soon as I'd like... (I'll try my best though!)

If you have any feature requests or suggestions, feel free to create an issue describing it. I'll try to get it working as soon as possible. If you really need something implemented quickly, a small donation would be appreciated, and would allow me to bump it to the top of my list of features.



## **LICENCING**

LibRapid is produced under the MIT License, so you are free to use the library how you like for personal and commercial purposes, though this is subject to some conditions, which can be found in full here: [LibRapid License](#)





## INDEX

### L

`librapid::array::ArrayContainer` (C++ class), 4  
`librapid::array::ArrayContainer::ArrayContainer`  
     (C++ function), 4, 5  
`librapid::array::ArrayContainer::get` (C++  
     function), 6  
`librapid::array::ArrayContainer::ndim` (C++  
     function), 6  
`librapid::array::ArrayContainer::operator=`  
     (C++ function), 5–7  
`librapid::array::ArrayContainer::operator<<`  
     (C++ function), 6, 7  
`librapid::array::ArrayContainer::operator[]`  
     (C++ function), 6  
`librapid::array::ArrayContainer::packet`  
     (C++ function), 6  
`librapid::array::ArrayContainer::Packet`  
     (C++ type), 4  
`librapid::array::ArrayContainer::scalar`  
     (C++ function), 6  
`librapid::array::ArrayContainer::Scalar`  
     (C++ type), 4  
`librapid::array::ArrayContainer::shape` (C++  
     function), 6  
`librapid::array::ArrayContainer::ShapeType`  
     (C++ type), 4  
`librapid::array::ArrayContainer::SizeType`  
     (C++ type), 4  
`librapid::array::ArrayContainer::storage`  
     (C++ function), 6  
`librapid::array::ArrayContainer::StorageType`  
     (C++ type), 4  
`librapid::array::ArrayContainer::str` (C++  
     function), 7  
`librapid::array::ArrayContainer::StrideType`  
     (C++ type), 4  
`librapid::array::ArrayContainer::write` (C++  
     function), 7  
`librapid::array::ArrayContainer::writePacket`  
     (C++ function), 7  
`librapid::GenericVector` (C++ class), 7  
`librapid::GenericVector::cmp` (C++ function), 10,  
     17  
`librapid::GenericVector::cross` (C++ function),  
     14  
`librapid::GenericVector::data` (C++ function), 16  
`librapid::GenericVector::dot` (C++ function), 14  
`librapid::GenericVector::GenericVector` (C++  
     function), 8  
`librapid::GenericVector::invMag` (C++ function),  
     13  
`librapid::GenericVector::m_data` (C++ member),  
     18  
`librapid::GenericVector::mag` (C++ function), 13  
`librapid::GenericVector::mag2` (C++ function), 13  
`librapid::GenericVector::norm` (C++ function), 14  
`librapid::GenericVector::operator bool` (C++  
     function), 14  
`librapid::GenericVector::operator!=` (C++  
     function), 12, 13, 18  
`librapid::GenericVector::operator*=` (C++  
     function), 9, 10, 17  
`librapid::GenericVector::operator+=` (C++  
     function), 9, 17  
`librapid::GenericVector::operator/=` (C++  
     function), 9, 10, 17  
`librapid::GenericVector::operator=` (C++ func-  
     tion), 9  
`librapid::GenericVector::operator==` (C++  
     function), 12, 13, 18  
`librapid::GenericVector::operator-` (C++ func-  
     tion), 10  
`librapid::GenericVector::operator-=` (C++  
     function), 9, 10, 17  
`librapid::GenericVector::operator>` (C++ func-  
     tion), 11, 12, 18  
`librapid::GenericVector::operator>=` (C++  
     function), 11, 13, 18  
`librapid::GenericVector::operator<` (C++ func-  
     tion), 11, 12, 18  
`librapid::GenericVector::operator<=` (C++  
     function), 11, 12, 18  
`librapid::GenericVector::operator[]` (C++  
     function), 9

librapid::GenericVector::proj (C++ *function*), 14  
librapid::GenericVector::StorageType (C++  
    *type*), 8  
librapid::GenericVector::str (C++ *function*), 17  
librapid::GenericVector::w (C++ *function*), 16, 17  
librapid::GenericVector::wxy (C++ *function*), 15  
librapid::GenericVector::wxyz (C++ *function*), 16  
librapid::GenericVector::wxz (C++ *function*), 15  
librapid::GenericVector::wxzy (C++ *function*), 16  
librapid::GenericVector::wyx (C++ *function*), 15  
librapid::GenericVector::wyxz (C++ *function*), 16  
librapid::GenericVector::wyz (C++ *function*), 15  
librapid::GenericVector::wyzx (C++ *function*), 16  
librapid::GenericVector::wzx (C++ *function*), 15  
librapid::GenericVector::wzxy (C++ *function*), 16  
librapid::GenericVector::wzy (C++ *function*), 15  
librapid::GenericVector::wzyx (C++ *function*), 16  
librapid::GenericVector::x (C++ *function*), 16, 17  
librapid::GenericVector::xwy (C++ *function*), 15  
librapid::GenericVector::xwyz (C++ *function*), 15  
librapid::GenericVector::xwz (C++ *function*), 15  
librapid::GenericVector::xwzy (C++ *function*), 15  
librapid::GenericVector::xy (C++ *function*), 14  
librapid::GenericVector::xyw (C++ *function*), 15  
librapid::GenericVector::xywz (C++ *function*), 15  
librapid::GenericVector::xyz (C++ *function*), 14  
librapid::GenericVector::xyzw (C++ *function*), 15  
librapid::GenericVector::xz (C++ *function*), 14  
librapid::GenericVector::xzw (C++ *function*), 15  
librapid::GenericVector::xzwy (C++ *function*), 15  
librapid::GenericVector::xzy (C++ *function*), 15  
librapid::GenericVector::xzyw (C++ *function*), 15  
librapid::GenericVector::y (C++ *function*), 16, 17  
librapid::GenericVector::ywx (C++ *function*), 15  
librapid::GenericVector::ywxz (C++ *function*), 16  
librapid::GenericVector::yz (C++ *function*), 15  
librapid::GenericVector::yzwx (C++ *function*), 16  
librapid::GenericVector::yx (C++ *function*), 14  
librapid::GenericVector::yxw (C++ *function*), 15  
librapid::GenericVector::yxwz (C++ *function*), 15  
librapid::GenericVector::yxz (C++ *function*), 15  
librapid::GenericVector::yxzw (C++ *function*), 15  
librapid::GenericVector::yz (C++ *function*), 14  
librapid::GenericVector::yzw (C++ *function*), 15  
librapid::GenericVector::yzwx (C++ *function*), 16  
librapid::GenericVector::yzx (C++ *function*), 15  
librapid::GenericVector::yzxw (C++ *function*), 16  
librapid::GenericVector::z (C++ *function*), 16, 17  
librapid::GenericVector::zwx (C++ *function*), 15  
librapid::GenericVector::zwxxy (C++ *function*), 16  
librapid::GenericVector::zwy (C++ *function*), 15  
librapid::GenericVector::zwyx (C++ *function*), 16  
librapid::GenericVector::zx (C++ *function*), 14  
librapid::GenericVector::zxw (C++ *function*), 15  
librapid::GenericVector::zxwy (C++ *function*), 16  
librapid::GenericVector::zxy (C++ *function*), 15  
librapid::GenericVector::zy (C++ *function*), 14  
librapid::GenericVector::zyw (C++ *function*), 15  
librapid::GenericVector::zywx (C++ *function*), 16  
librapid::GenericVector::zyx (C++ *function*), 15  
librapid::GenericVector::zyxw (C++ *function*), 16