

---

**xtensor-r**

**Feb 22, 2019**



---

# INSTALLATION

---

<b>1</b>	<b>Enabling R arrays in your C++ libraries</b>	<b>3</b>
<b>2</b>	<b>Licensing</b>	<b>5</b>
2.1	Installation of the xtensor-r C++ library . . . . .	5
2.2	Installation of the R package . . . . .	6
2.3	Basic Usage . . . . .	6
2.4	Arrays and tensors . . . . .	7
2.5	R Peculiarities . . . . .	7
2.6	API reference . . . . .	8
2.7	Releasing xtensor-r . . . . .	11
2.8	Releasing the R Package . . . . .	12



R bindings for the `xtensor` C++ multi-dimensional array library.

What are `xtensor` and `xtensor-r`?

- `xtensor` is a C++ library for multi-dimensional arrays enabling numpy-style broadcasting and lazy computing.
- `xtensor-r` enables inplace use of R arrays with all the benefits from `xtensor`
  - C++ universal functions and broadcasting
  - STL - compliant APIs.

The [numpy to xtensor cheat sheet](#) from the `xtensor` documentation shows how numpy APIs translate to C++ with `xtensor`.

The Python bindings for `xtensor` are based on the [Rcpp](#) C++ library, which enables seamless interoperability between C++ and Python.



---

## Enabling R arrays in your C++ libraries

---

Instead of exposing new types to R, `xtensor-r` enables the use of R data structures from C++ using R's C API.

In addition to the basic accessors and iterators of `xtensor` containers, it also enables using R arrays with `xtensor`'s expression system.

Besides `xtensor-r` provides an API to create *Universal functions* from simple scalar functions from your C++ code.

`xtensor` and `xtensor-r` require a modern C++ compiler supporting C++14. The following C++ compilers are supported:

- On Windows platforms, Visual C++ 2015 Update 2, or more recent
- On Unix platforms, gcc 4.9 or a recent version of Clang



We use a shared copyright model that enables all contributors to maintain the copyright on their contributions. This software is licensed under the BSD-3-Clause license. See the LICENSE file for details.

## 2.1 Installation of the `xtensor-r` C++ library

`xtensor-r` is a header-only C++ library. We maintain the conda package for `xtensor-r` and its dependencies.

Besides the `xtensor-r` headers, all these methods place the `cmake` project configuration file in the right location so that third-party projects can use `cmake`'s `find_package` to locate `xtensor-r` headers.

### 2.1.1 Using the conda package

A package for `xtensor-r` is available on the conda package manager.

```
conda install -c conda-forge xtensor-r
```

### 2.1.2 From source with `cmake`

You can also install `xtensor-r` from source with `cmake`. On Unix platforms, from the source directory:

```
mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=/path/to/prefix ..
make install
```

On Windows platforms, from the source directory:

```
mkdir build
cd build
cmake -G "NMake Makefiles" -DCMAKE_INSTALL_PREFIX=/path/to/prefix ..
nmake
nmake install
```

## 2.2 Installation of the R package

We provide a R package for Xtensor on both conda and CRAN (Comprehensive R Archive Network). The packaging boilerplate for the R package is available at <https://github.com/QuantStack/Xtensor.R>.

To install the R package with conda:

```
conda install r-xtensor -c conda-forge
```

To install the R package from CRAN:

```
R CMD INSTALL xtensor
```

or from the GitHub repository using devtools:

```
devtools::install_github("QuantStack/Xtensor.R")
```

---

**Note:** A key difference between the version of the R package available on CRAN and conda is that

- the CRAN package *vendors* the headers of `xtl`, `xsimd`, `xtensor`, and `xtensor-r`.
- the conda package *depends* on the conda packages for `xtl`, `xsimd`, `xtensor` and `xtensor-r`.

When installing the package from CRAN, it is still possible to drop the vendored dependencies by adding the `--configure-args='--novendor'` option to the `R CMD INSTALL` command, or by defining the `NO_VENDOR_XTENSOR` environment variable to `YES`.

---

## 2.3 Basic Usage

### 2.3.1 Example : Use an algorithm of the C++ library on a R array inplace

**C++ code**

```
#include <numeric> // Standard library import for std::accumulate
#define STRICT_R_HEADERS // Otherwise a PI macro is defined in R
#include "xtensor/xmath.hpp" // xtensor import for the C++ universal_
↪ functions
#include "xtensor-r/rarray.hpp" // R bindings
#include <Rcpp.h>

using namespace Rcpp;

// [[Rcpp::plugins(cpp14)]]
```

(continues on next page)

(continued from previous page)

```

// [[Rcpp::export]]
double sum_of_sines(xt::rarray<double>& m)
{
    auto sines = xt::sin(m); // sines does not actually hold values.
    return std::accumulate(sines.cbegin(), sines.cend(), 0.0);
}

```

**R code:**

```

v <- matrix(0:14, nrow=3, ncol=5)
s <- sum_of_sines(v)
s

```

**Outputs**

```
1.2853996391883833
```

Note that R places some restriction on what C++ types are usable. We've noted the differences in *R Peculiarities*.

## 2.4 Arrays and tensors

`xtensor-r` provides two container types wrapping R arrays: `rarray` and `rtensor`. They are the counterparts to `xarray` and `xtensor` containers.

### 2.4.1 rarray

Like `xarray`, `rarray` has a dynamic shape. This means that you can reshape the R array on the C++ side and see this change reflected on the R side. `rarray` doesn't make a copy of the shape, but reads it each time it is needed. Therefore, if a reference on a `rarray` is kept in the C++ code and the corresponding R array is then reshaped in the R code, this modification will reflect in the `rarray`.

### 2.4.2 rtensor

Like `xtensor`, `rtensor` has a static stack-allocated shape. This means that the shape of the R array is copied into the shape of the `rtensor` upon creation. As a consequence, reshapes are not reflected across languages. However, this drawback is offset by a more effective computation of shape and broadcast.

## 2.5 R Peculiarities

R supports only a subset of types available in C++. The types which are natively supported with `xtensor-R` are `int` (32 bit), `double`, `complex`, `byte`. Using other types (such as a C++ `float`) in a `rarray` or `rtensor` will fail to compile.

Below is a mapping from R to `xtensor`:

R type	xtensor type
Integer Vector	rarray<int> or rarray<int32_t>
Real Vector	rarray<double>
Complex Vector	rarray<std::complex<double>>
Raw Vector	rarray<Rbyte> or rarray<uint8_t>
Logical Vector	Not supported at the moment

Note that, for illustration purposes, `rarray` was used above. The `rtensor` container works just as well.

### 2.5.1 R's NA and NaN values

R encodes missing values (NAs) as special values of the underlying type.

Xtensor-r supports R's missing values with the `rarray_optional` and `rtensor_optional` counterparts to `rarray` and `rtensor` which enable operations with missing values with the `xoptional` API.

### 2.5.2 R defining PI as macro

Ancient versions of S used to define a `PI` macro. This macro collides with `xtensor`, as `xtensor` is using `PI` as a variable name in the numeric constants.

If you're encountering this issue, either reorder your headers so that the `xmath` header of `xtensor` is included before `Rcpp` / `xtensor-r` or use the following define before including `xtensor-r` and `Rcpp`:

```
#define STRICT_R_HEADERS
```

This prevents the `PI` macro definition.

## 2.6 API reference

### 2.6.1 Containers

#### `rarray`

```
template <class T>
class rarray : public xt::rcontainer<rarray<T>>, public xcontainer_semantic<rarray<T>>
```

Multidimensional container providing the `xtensor` container semantics to an R array.

`rarray` is similar to the `xarray` container in that it has a dynamic dimensionality. Reshapes of a `rarray` container are reflected in the underlying R array.

See *rtensor*

#### Template Parameters

- `T`: The type of the element stored in the `rarray`.

## Constructors

**rarray** (**const** shape\_type &shape)

Allocates an uninitialized rarray with the specified shape.

### Parameters

- shape: the shape of the rarray

**rarray** (**const** shape\_type &shape, const\_reference value)

Allocates a rarray with the specified shape.

Elements are initialized to the specified value.

### Parameters

- shape: the shape of the rarray
- value: the value of the elements

**rarray** (**const** value\_type &t)

Allocates a rarray with nested initializer lists.

**template** <class S>

**rarray**<T> **from\_shape** (S &&shape)

Allocates and returns an rarray with the specified shape.

### Parameters

- shape: the shape of the rarray

## Copy semantic

**rarray** (**const** self\_type &rhs)

The copy constructor.

auto **operator=** (**const** self\_type &rhs)

The assignment operator.

## Extended copy semantic

**template** <class E>

**rarray** (**const** xexpression<E> &e)

The extended copy constructor.

**template** <class E>

auto **operator=** (**const** xexpression<E> &e)

The extended assignment operator.

## rtensor

**template** <class T, std::size\_t N>

```
class rtensor : public xt::rcontainer<rtensor<T, N>>, public xcontainer_semantic<rtensor<T, N>>
```

Multidimensional container providing the xtensor container semantics wrapping an R array.

rtensor is similar to the xtensor container in that it has a static dimensionality.

Unlike the rarray container, rtensor cannot be reshaped with a different number of dimensions and reshapes are not reflected on the R side. However, rtensor has benefits compared to rarray in terms of performances. rtensor shapes are stack-allocated which makes iteration upon rtensor faster than with pyarray.

See *rarray*

### Template Parameters

- T: The type of the element stored in the rarray.

### Constructors

```
rtensor ()
```

Allocates an uninitialized rtensor.

```
rtensor (nested_initializer_list_t<T, N> t)
```

Allocates a rtensor with a nested initializer list.

```
rtensor (const shape_type &shape)
```

Allocates an uninitialized rtensor with the specified shape and layout.

#### Parameters

- shape: the shape of the rtensor
- l: the layout\_type of the rtensor

```
rtensor (const shape_type &shape, const_reference value)
```

Allocates a rtensor with the specified shape and layout.

Elements are initialized to the specified value.

#### Parameters

- shape: the shape of the rtensor
- value: the value of the elements
- l: the layout\_type of the rtensor

### Copy semantic

```
rtensor (const self_type &rhs)
```

The copy constructor.

```
auto operator= (const self_type &rhs)
```

The assignment operator.

### Extended copy semantic

```
template <class E>
rtensor (const xexpression<E> &e)
    The extended copy constructor.
```

```
template <class E>
auto operator= (const xexpression<E> &e)
    The extended assignment operator.
```

### roptional

```
template <class RC>
class rcontainer_optional : public xoptional_assembly_base<rcontainer_optional<RC>>, public xcontainer_semantic
    Multidimensional container of optional values providing the xtensor container semantics to an R array.

    rcontainer_optional is not meant to be used directly, but through the aliases rarray_optional<T>
    and rtensor_optional<T, N>.
```

Depending on the value type, optional values are reference proxies on R's NA\_INTEGER, NA\_LOGICAL, NA\_REAL, NA\_STRING, or R\_NilValue.

Besides support for optionality, *rarray\_optional* and *rtensor\_optional* are similar to the *rarray* and *rtensor* respectively, with respect to dynamic and static dimensionality.

#### Template Parameters

- T: The type of the element stored in the rarray.

## 2.6.2 Numpy-style universal functions

### rvectorize

```
template <class R, class... Args>
rvectorizer<R (*) (Args...), R, Args...> xt :: rvectorize
    R (*)Args...Create numpy-style universal function from scalar function.
```

## 2.7 Releasing xtensor-r

### 2.7.1 Releasing a new version

From the master branch of xtensor-r

- Make sure that you are in sync with the master branch of the upstream remote.
- In file `xtensor_r_config.hpp`, set the macros for `XTENSOR_R_VERSION_MAJOR`, `XTENSOR_R_VERSION_MINOR` and `XTENSOR_R_VERSION_PATCH` to the desired values.
- Update the readme file w.r.t. dependencies on xtensor and pybind11.
- Stage the changes (`git add`), commit the changes (`git commit`) and add a tag of the form `Major.minor.patch`. It is important to not add any other content to the tag name.
- Push the new commit and tag to the main repository. (`git push`, and `git push --tags`)

## 2.7.2 Updating the conda-forge recipe

xtensor-r has been packaged for the conda package manager. Once the new tag has been pushed on GitHub, edit the conda-forge recipe for xtensor in the following fashion:

- Update the version number to the new Major.minor.patch.
- Set the build number to 0.
- Update the hash of the source tarball.
- Check for the versions of the dependencies.
- Optionally, rerender the conda-forge feedstock.

## 2.8 Releasing the R Package

To build a package tarball for R, simply clone <https://github.com/QuantStack/Xtensor.R> and run

```
R CMD build .
```

from the root of the source directory.

To produce a package tarball *vendoring* the xtl, xsimd and xtensor dependencies (for e.g. CRAN), clone <https://github.com/QuantStack/Xtensor.R> and run

```
./configure # Downloads xtl, xtensor, xsimd, and xtensor-r.  
R CMD build .
```

from the root of the source directory.

## X

`xt::rarray (C++ class), 8`  
`xt::rarray::from_shape (C++ function), 9`  
`xt::rarray::operator= (C++ function), 9`  
`xt::rarray::rarray (C++ function), 9`  
`xt::rcontainer_optional (C++ class), 11`  
`xt::rtensor (C++ class), 9`  
`xt::rtensor::operator= (C++ function), 10, 11`  
`xt::rtensor::rtensor (C++ function), 10, 11`  
`xt::rvectorize (C++ function), 11`