# xtensor

***Release***

**Oct 07, 2017**

# INSTALLATION

Multi-dimensional arrays with broadcasting and lazy computing.

*xtensor* is a C++ library meant for numerical analysis with multi-dimensional array expressions.

*xtensor* provides

- an extensible expression system enabling **lazy broadcasting**.
- an API following the idioms of the **C++ standard library**.
- tools to manipulate array expressions and build upon *xtensor*.

Containers of *xtensor* are inspired by NumPy, the Python array programming library. **Adaptors** for existing data structures to be plugged into our expression system can easily be written. In fact, *xtensor* can be used to **process numpy data structures inplace** using Python's buffer protocol. For more details on the numpy bindings, check out the xtensor-python project.

*xtensor* requires 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

CHAPTER 1

# Licensing



## 1.1 Installation

Although `xtensor` is a header-only library, we provide standardized means to install it, with package managers or with cmake.

Besides the xtendor 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 headers.

### 1.1.1 Using the conda package

A package for xtensor is available on the conda package manager.

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

### 1.1.2 Using the Debian package

A package for xtensor is available on Debian.

```
sudo apt-get install xtensor-dev
```

### 1.1.3 From source with cmake

You can also install `xtensor` from source with cmake. This requires that you have the xtl library installed on your system. 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
```

See the section of the documentation on *Build and configuration*, for more details on how to cmake options.

## 1.2 Changelog

### 1.2.1 0.12.0

#### Breaking changes

- `xtensor` now depends on `xtl`. #421.

#### New features

- `xtensor` has an optional dependency on `xsimd` for enabling simd acceleration #426.
- All expressions have an additional safe access function (`at`) #420.
- norm functions #440.
- `closure_pointer` used in iterators returning temporaries so their `operator->` can be correctly defined #446.
- expressions tags added so `xtensor` expression system can be extended #447.

#### Other changes

- Preconditions and exceptions #409.
- `isclose` is now symmetric #411.
- concepts added #414.
- narrowing cast for mixed arithmetic #432.
- `is_xexpression` concept fixed #439.
- `void_t` implementation fixed for compilers affected by C++14 defect CWG 1558 #448.

### 1.2.2 0.11.3

- Fixed bug in length-1 statically dimensioned tensor construction #431.

### 1.2.3 0.11.2

- Fixup compilation issue with latest clang compiler. (missing *constexpr* keyword) #407.

### 1.2.4 0.11.1

- Fixes some warnings in julia and python bindings

### 1.2.5 0.11.0

#### Breaking changes

- `xbegin` / `xend`, `xcbegin` / `xcend`, `xrbegin` / `xrend` and `xcrbegin` / `xcrend` methods replaced with classical `begin` / `end`, `cbegin` / `cend`, `rbegin` / `rend` and `crbegin` / `crend` methods. Old `begin` / `end` methods and their variants have been removed. #370.
- `xview` now uses a const stepper when its underlying expression is const. #385.

#### Other changes

- `xview` copy semantic and move semantic fixed. #377.
- `xoptional` can be implicitly constructed from a scalar. #382.
- build with Emscripten fixed. #388.
- STL version detection improved. #396.
- Implicit conversion between signed and unsigned integers fixed. #397.

## 1.3 Basic usage

**Initialize a 2-D array and compute the sum of one of its rows and a 1-D array.**

```
#include <iostream>
#include "xtensor/xarray.hpp"
#include "xtensor/xio.hpp"

xt::xarray<double> arr1
  {{1.0, 2.0, 3.0},
   {2.0, 5.0, 7.0},
   {2.0, 5.0, 7.0}};

xt::xarray<double> arr2
  {5.0, 6.0, 7.0};

xt::xarray<double> res = xt::view(arr1, 1) + arr2;

std::cout << res;
```

Outputs:

```
{7, 11, 14}
```

**Initialize a 1-D array and reshape it inplace.**

```
#include <iostream>
#include "xtensor/xarray.hpp"
#include "xtensor/xio.hpp"

xt::xarray<int> arr
  {1, 2, 3, 4, 5, 6, 7, 8, 9};

arr.reshape({3, 3});

std::cout << arr;
```

Outputs:

```
{{1, 2, 3},
 {4, 5, 6},
 {7, 8, 9}}
```

**Broadcasting the `xt::pow` universal functions.**

```
#include <iostream>
#include "xtensor/xarray.hpp"
#include "xtensor/xmath.hpp"
#include "xtensor/xio.hpp"

xt::xarray<double> arr1
  {1.0, 2.0, 3.0};

xt::xarray<unsigned int> arr2
  {4, 5, 6, 7};

arr2.reshape({4, 1});

xt::xarray<double> res = xt::pow(arr1, arr2);

std::cout << res;
```

Outputs:

```
{{1, 16, 81},
 {1, 32, 243},
 {1, 64, 729},
 {1, 128, 2187}}
```

## 1.4 Expressions and lazy evaluation

*xtensor* is more than an N-dimensional array library: it is an expression engine that allows numerical computation on any object implementing the expression interface. These objects can be in-memory containers such as `xarray<T>`

and `xtensor<T>`, but can also be backed by a database or a representation on the file system. This also enables creating adaptors as expressions for other data structures.

### 1.4.1 Expressions

Assume `x`, `y` and `z` are arrays of *compatible shapes* (we'll come back to that later), the return type of an expression such as `x + y * sin(z)` is **not an array**. The result is an `xexpression` which offers the same interface as an N-dimensional array but does not hold any value. Such expressions can be plugged into others to build more complex expressions:

```
auto f = x + y * sin(z);
auto f2 = w + 2 * cos(f);
```

The expression engine avoids the evaluation of intermediate results and their storage in temporary arrays, so you can achieve the same performance as if you had written a simple loop. Assuming `x`, `y` and `z` are one-dimensional arrays of length `n`,

```
xt::xarray<double> res = x + y * sin(z)
```

will produce quite the same assembly as the following loop:

```
xt::xarray<double> res(n);
for(size_t i = 0; i < n; ++i)
{
    res(i) = x(i) + y(i) * sin(z(i));
}
```

### 1.4.2 Lazy evaluation

An expression such as `x + y * sin(z)` does not hold the result. **Values are only computed upon access or when the expression is assigned to a container**. This allows to operate symbolically on very large arrays and only compute the result for the indices of interest:

```
// Assume x and y are xarrays each containing 1 000 000 objects
auto f = cos(x) + sin(y);

double first_res = f(1200);
double second_res = f(2500);
// Only two values have been computed
```

That means if you use the same expression in two assign statements, the computation of the expression will be done twice. Depending on the complexity of the computation and the size of the data, it might be convenient to store the result of the expression in a temporary variable:

```
// Assume x and y are small arrays
xt::xarray<double> tmp = cos(x) + sin(y);
xt::xarray<double> res1 = tmp + 2 * x;
xt::xarray<double> res2 = tmp - 2 * x;
```

### 1.4.3 Forcing evaluation

If you have to force the evaluation of an xexpression for some reason (for example, you want to have all results in memory to perform a sort, or use external BLAS functions) then you can use `xt::eval` on an xexpression.

Evaluating will either return an rvalue to a newly allocated container in the case of a xexpression, or a reference to a container in case you are evaluating a `xarray` or `xtensor`. Note that, in order to avoid copies, you should use an universal reference on the lefthand side (`auto&&`). For example:

```
xt::xarray<double> a = {1, 2, 3};
xt::xarray<double> b = {3, 2, 1};
auto calc = a + b; // unevaluated xexpression!
auto&& e = xt::eval(calc); // a rvalue container xarray!
// this just returns a reference to the existing container
auto&& a_ref = xt::eval(a);
```

### 1.4.4 Broadcasting

The number of dimensions of an `xexpression` and the sizes of these dimensions are provided by the `shape()` method, which returns a sequence of unsigned integers specifying the size of each dimension. We can operate on expressions of different shapes of dimensions in a elementwise fashion. Broadcasting rules of *xtensor* are similar to those of Numpy and libdynd.

In an operation involving two arrays of different dimensions, the array with the lesser dimensions is broadcast across the leading dimensions of the other. For example, if `A` has shape `(2, 3)`, and `B` has shape `(4, 2, 3)`, the result of a broadcasted operation with `A` and `B` has shape `(4, 2, 3)`.

```
   (2, 3) # A
(4, 2, 3) # B
---------
(4, 2, 3) # Result
```

The same rule holds for scalars, which are handled as 0-D expressions. If *A* is a scalar, the equation becomes:

```
       () # A
(4, 2, 3) # B
---------
(4, 2, 3) # Result
```

If matched up dimensions of two input arrays are different, and one of them has size `1`, it is broadcast to match the size of the other. Let's say B has the shape `(4, 2, 1)` in the previous example, so the broadcasting happens as follows:

```
   (2, 3) # A
(4, 2, 1) # B
---------
(4, 2, 3) # Result
```

### 1.4.5 Expression interface

All `xexpression` s in *xtensor* provide at least the following interface:

#### Shape

- `dimension()` returns the number of dimension of the expression.
- `shape()` returns the shape of the expression.

```cpp
#include <vector>
#include "xtensor/xarray.hpp"

std::vector<size_t> shape = {3, 2, 4};
xt::xarray<double> a(shape);
size_t d = a.dimension();
const std::vector<size_t>& s = a.shape();
bool res = (d == shape.size()) && (s == shape);
// => res = true
```

### Element access

- `operator()` is an access operator which can take multiple integral arguments or none.
- `at()` is similar to `operator()` but checks that its number of arguments does not exceed the number of dimensions, and performs bounds check. This should not be used where you expect `operator()` to perform broadcasting.
- `operator[]` has two overloads: one that takes a single integral argument and is equivalent to the call of `operator()` with one argument, and one with a single multi-index argument, which can be of size determined at runtime. This operator also supports braced initializer arguments.
- `element()` is an access operator which takes a pair of iterators on a container of indices.

```cpp
#include <vector>
#inclde "xtensor/xarray.hpp"

// xt::xarray<double> a = ...
std::vector<size_t> index = {1, 1, 1};
double v1 = a(1, 1, 1);
double v2 = a[index],
double v3 = a.element(index.begin(), index.end());
// => v1 = v2 = v3
```

### Iterators

- `begin()` and `end()` return instances of `xiterator` which can be used to iterate over all the elements of the expression. The layout of the iteration can be specified through the `layout_type` template parameter, accepted values are `layout_type::row_major` and `layout_type::column_major`. If not specified, `DEFAULT_LAYOUT` is used. This iterator pair permits to use algorithms of the STL with `xexpression` as if they were simple containers.
- `begin(shape)` and `end(shape)` are similar but take a *broadcasting shape* as an argument. Elements are iterated upon in `DEFAULT_LAYOUT` if no `layout_type` template parameter is specified. Certain dimensions are repeated to match the provided shape as per the rules described above.
- `rbegin()` and `rend()` return instances of `xiterator` which can be used to iterate over all the elements of the reversed expression. As `begin()` and `end()`, the layout of the iteration can be specified through the `layout_type` parameter.
- `rbegin(shape)` and `rend(shape)` are the reversed counterpart of `begin(shape)` and `end(shape)`.

## 1.5 Arrays and tensors

### 1.5.1 Internal memory layout

A multi-dimensional array of *xtensor* consists of a contiguous one-dimensional buffer combined with an indexing scheme that maps unsigned integers to the location of an element in the buffer. The range in which the indices can vary is specified by the *shape* of the array.

The scheme used to map indices into a location in the buffer is a strided index scheme. In such a scheme, the index `(i0, ..., in)` corresponds to the offset `sum(ik * sk)` from the beginning of the one-dimensional buffer, where `(s0, ..., sn)` are the *strides* of the array. Some particular cases of strided schemes implement well known memory layouts:

- the row-major layout (or C layout) is a strided index scheme where the strides grow from right to left
- the column-major layout (or Fortran layout) is a strided index scheme where the strides grow from left to right

`xtensor` provides a `layout_type` enum that helps to specify the layout used by multi-dimensional arrays. This enum can be used in two ways:

- at compile time, as a template argument. The value `layout_type::dynamic` allows to specify any strided index scheme at runtime (including row-major and column-major schemes), whereas `layout_type::row_major` and `layout_type::column_major` fixes the strided index scheme and disable `reshape` and constructor overloads taking a set of strides or a layout value as parameter. The default value of the template parameter is `DEFAULT_LAYOUT`.
- at runtime if the previous template parameter was set to `layout_type::dynamic`. In that case, `reshape` and constructor overloads allow to aspecify a set of strides or a layout value to avoid strides computation. If neither strides nor layout is specified when instantiating or reshaping a multi-dimensional array, strides corresponding to `DEFAULT_LAYOUT` are used.

The following example shows how to initialize a multi-dimensional array of dynamic layout with specified strides:

```
#include <vector>
#include "xtensor/xarray.hpp"

std::vector<size_t> shape = { 3, 2, 4 };
std::vector<size_t> strides = { 8, 4, 1 };
xt::xarray<double, layout_type::dynamic> a(shape, strides);
```

However, this requires to carefully compute the strides to avoid buffer overflow when accessing elements of the array. We can use the layout shortcut to specify the strides instead of computing them:

```
#include <vector>
#include "xtensor/xarray.hpp"

std::vector<size_t> shape = { 3, 2, 4 };
xt::xarray<double, layout_type::dynamic> a(shape, xt::layout::row_major);
```

If the layout of the array can be fixed at compile time, we can even do simpler:

```
#include <vector>
#include "xtensor/xarray.hpp"

std::vector<size_t> shape = { 3, 2, 4 };
xt::xarray<double, layout_type::row_major> a(shape);
```

However, in that latter case, the layout of the array is forced to `row_major` at compile time, and therefore cannot be changed at runtime.

### 1.5.2 Runtime vs Compile-time dimensionality

Two container classes implementing multi-dimensional arrays are provided: `xarray` and `xtensor`.

- `xarray` can be reshaped dynamically to any number of dimensions. It is the container that is the most similar to numpy arrays.
- `xtensor` has a dimension set at compilation time, which enables many optimizations. For example, shapes and strides of `xtensor` instances are allocated on the stack instead of the heap.

Let's use `xtensor` instead of `xarray` in the previous example:

```
#include <array>
#include "xtensor/xtensor.hpp"

std::array<size_t, 3> shape = { 3, 2, 4 };
xt::xtensor<double, 3, layout_type::row_major> a(shape);
```

`xarray` and `xtensor` containers are both `xexpression` s and can be involved and mixed in mathematical expressions, assigned to each other etc... They provide an augmented interface compared to other `xexpression` types:

- Each method exposed in `xexpression` interface has its non-const counterpart exposed by both `xarray` and `xtensor`.
- `reshape()` reshapes the container in place, that is, if the global size of the container doesn't change, no memory allocation occurs.
- `transpose()` transposes the container in place, that is, no memory allocation occurs.
- `strides()` returns the strides of the container, used to compute the position of an element in the underlying buffer.

### 1.5.3 Performance

The dynamic dimensionality of `xarray` comes at a cost. Since the dimension is unknown at build time, the sequences holding shape and strides of `xarray` instances are heap-allocated, which makes it significantly more expansive than `xtensor`. Shape and strides of `xtensor` are stack-allocated which makes them more efficient.

More generally, the library implements a `promote_shape` mechanism at build time to determine the optimal sequence type to hold the shape of an expression. The shape type of a broadcasting expression whose members have a dimensionality determined at compile time will have a stack-allocated shape. If a single member of a broadcasting expression has a dynamic dimension (for example an `xarray`), it bubbles up to entire broadcasting expression which will have a heap allocated shape. The same hold for views, broadcast expressions, etc...

### 1.5.4 Aliasing and temporaries

In some cases, an expression should not be directly assigned to a container. Instead, it has to be assigned to a temporary variable before being copied into the destination container. This occurs when the destination container is involved in the expression and has to be reshaped. This phenomenon is known as *aliasing*.

To prevent this, *xtensor* assigns the expression to a temporary variable before copying it. In the case of `xarray`, this results in an extra dynamic memory allocation and copy.

However, if the left-hand side is not involved in the expression being assigned, no temporary variable should be required. *xtensor* cannot detect such cases automatically and applies the "temporary variable rule" by default. A mechanism is provided to forcibly prevent usage of a temporary variable:

```
#include "xtensor/xarray.hpp"
#include "xtensor/xnoalias.hpp"

// a, b, and c are xt::xarrays previously initialized
xt::noalias(b) = a + c;
// Even if b has to be reshaped, a+c will be assigned directly to it
// No temporary variable will be involved
```

#### Example of aliasing

The aliasing phenomenon is illustrated in the following example:

```
#include <vector>
#include "xtensor/xarray.hpp"

std::vector<size_t> a_shape = {3, 2, 4};
xt::xarray<double> a(a_shape);

std::vector<size_t> b_shape = {2, 4};
xt::xarray<double> b(b_shape);

b = a + b;
// b appears on both left-hand and right-hand sides of the statement
```

In the above example, the shape of `a + b` is `{ 3, 2, 4 }`. Therefore, `b` must first be reshaped, which impacts how the right-hand side is computed.

If the values of `b` were copied into the new buffer directly without an intermediary variable, then we would have `new_b(0, i, j) == old_b(i, j) for (i,j) in [0,1] x [0, 3]`. After the reshape of `bb`, `a(0, i, j) + b(0, i, j)` is assigned to `b(0, i, j)`, then, due to broadcasting rules, `a(1, i, j) + b(0, i, j)` is assigned to `b(1, i, j)`. The issue is `b(0, i, j)` has been changed by the previous assignment.

## 1.6 Operators and functions

### 1.6.1 Arithmetic operators

*xtensor* provides overloads of traditional arithmetic operators for `xexpression` objects:

- unary `operator+`
- unary `operator-`
- `operator+`
- `operator-`
- `operator*`
- `operator/`

All these operators are element-wise operators and apply the lazy broadcasting rules explained in a previous section.

```
#incude "xtensor/xarray.hpp"

xt::xarray<int> a = {{1, 2}, {3, 4}};
xt::xarray<int> b = {1, 2};
```

```
xt::xarray<int> res = 2 * (a + b);
// => res = {{4, 8}, {8, 12}}
```

### 1.6.2 Logical operators

*xtensor* also provides overloads of the logical operators:

- `operator!`
- `operator||`
- `operator&&`

Like arithmetic operators, these logical operators are element-wise operators and apply the lazy broadcasting rules. In addition to these element-wise logical operators, *xtensor* provides two reducing boolean functions:

- `any(E&& e)` returns `true` if any of `e` elements is truthy, `false` otherwise.
- `all(E&& e)` returns `true` if all alements of `e` are truthy, `false` otherwise.

and an element-wise ternary function (similar to the `:  ?` ternary operator):

- `where(E&& b, E1&& e&, E2&& e2)` returns an `xexpression` whose elements are those of `e1` when corresponding elements of `b` are thruthy, and those of `e2` otherwise.

```
#include "xtensor/xarray.hpp"

xt::xarray<bool> b = { false, true, true, false };
xt::xarray<int> a1 = { 1,   2,  3,  4 };
xt::xarray<int> a2 = { 11, 12, 13, 14 };

xt::xarray<int> res = xt::where(b, a1, a2);
// => res = { 11, 2, 3, 14 }
```

Unlike in `numpy.where`, `xt::where` takes full advantage of the lazyness of *xtensor*.

### 1.6.3 Comparison operators

*xtensor* provides overloads of the inequality operators:

- `operator<`
- `operator<=`
- `operator>`
- `operator>=`

These overloads of inequality operators are quite different from the standard C++ inequality operators: they are element-wise operators returning boolean `xexpression`:

```
#include "xtensor/xarray.hpp"

xt::xarray<int> a1 = {  1, 12,  3, 14 };
xt::xarray<int> a2 = { 11,  2, 13, 4  };
xt::xarray<bool> comp = a1 < a2;
// => comp = { true, false, true, false }
```

However, equality operators are similar to the traditional ones in C++:

- `operator==(const E1& e1, const E2& e2)` returns `true` if `e1` and `e2` hold the same elements.
- `operator!=(const E1& e1, const E2& e2)` returns `true` if `e1` and `e2` don't hold the same elements.

Element-wise equality comparison can be achieved through the `xt::equal` function.

```
#include "xtensor/xarray.hpp"

xt::xarray<int> a1 = {  1,  2, 3, 4};
xt::xarray<int> a2 = { 11, 12, 3, 4};

bool res = (a1 == a2);
// => res = false

xt::xarray<bool> re = xt::equal(a1, a2);
// => re = { false, false, true, true }
```

### 1.6.4 Mathematical functions

*xtensor* provides overloads for many of the standard mathematical functions:

- basic functions: `abs`, `remainder`, `fma`, ...
- exponential functions: `exp`, `expm1`, `log`, `log1p`, ...
- power functions: `pow`, `sqrt`, `cbrt`, ...
- trigonometric functions: `sin`, `cos`, `tan`, ...
- hyperbolic functions: `sinh`, `cosh`, `tanh`, ...
- Error and gamma functions: `erf`, `erfc`, `tgamma`, `lgamma`, ....
- Nearest integer floating point operations: `ceil`, `floor`, `trunc`, ...

See the API reference for a comprehensive list of available functions. Like operators, the mathematical functions are element-wise functions and apply the lazy broadcasting rules.

### 1.6.5 Reducers

*xtensor* provides reducers, that is, means for accumulating values of tensor expressions over prescribed axes. The return value of a reducer is an `xexpression` with the same shape as the input expression, with the specified axes removed.

```
#include "xtensor/xarray.hpp"
#include "xtensor/xmath.hpp"

xt::xarray<double> a = ones<double>({3, 2, 4, 6, 5 });
xt::xarray<double> res = xt::sum(a, {1, 3});
// => res.shape() = { 3, 4, 5 };
// => res(0, 0, 0) = 12
```

You can also call the `reduce` generator with your own reducing function:

```
#include "xtensor/xarray.hpp"
#include "xtensor/xreducer.hpp"

xt::xarray<double> a = some_init_function({3, 2, 4, 6, 5});
xt::xarray<double> res = reduce([](double a, double b) { return a*a + b*b; },
                                a,
                                {1, 3});
```

### 1.6.6 Universal functions and vectorization

*xtensor* provides utilities to **vectorize any scalar function** (taking multiple scalar arguments) into a function that will perform on `xexpression` s, applying the lazy broadcasting rules which we described in a previous section. These functions are called `xfunction` s. They are *xtensor*'s counterpart to numpy's universal functions.

Actually, all arithmetic and logical operators, inequality operator and mathematical functions we described before are `xfunction` s.

The following snippet shows how to vectorize a scalar function taking two arguments:

```
#include "xtensor/xarray.hpp"
#include "xtensor/xvectorize.hpp"

int f(int a, int b)
{
    return a + 2 * b;
}

auto vecf = xt::vectorize(f);
xt::xarray<int> a = { 11, 12, 13 };
xt::xarray<int> b = {  1,  2,  3 };
xt::xarray<int> res = vecf(a, b);
// => res = { 13, 16, 19 }
```

## 1.7 Views

Views are used to adapt the shape of an `xexpression` without changing it, nor copying it. *xtensor* provides two kinds of views.

### 1.7.1 Sliced views

Sliced views consist of the combination of the `xexpression` to adapt, and a list of `slice` s that specify how the shape must be adapted. Sliced views are implemented by the `xview` class. Objects of this type should not be instantiated directly, but though the `view` helper function.

Slices can be specified in the following ways:

- selection in a dimension by specifying an index (unsigned integer)
- `range(min, max)`, a slice representing an interval
- `range(min, max, step)`, a slice representing a stepped interval
- `all()`, a slice representing all the elements of a dimension
- `newaxis()`, a slice representing an additional dimension of length one

```
#include <vector>
#include "xtensor/xarray.hpp"
#include "xtensor/xview.hpp"

std::vector<size_t> shape = {3, 2, 4};
xt::xarray<int> a(shape);

// View with same number of dimensions
auto v1 = xt::view(a, xt::range(1, 3), xt::all(), xt::range(1, 3));
// => v1.shape() = { 2, 2, 2 }
// => v1(0, 0, 0) = a(1, 0, 1)
// => v1(1, 1, 1) = a(2, 1, 2)

// View reducing the number of dimensions
auto v2 = xt::view(a, 1, xt::all(), xt::range(0, 4, 2));
// => v2.shape() = { 2, 2 }
// => v2(0, 0) = a(1, 0, 0)
// => v2(1, 1) = a(1, 1, 2)

// View increasing the number of dimensions
auto v3 = xt::view(a, xt::all(), xt::all(), xt::newaxis(), xt::all());
// => v3.shape() = { 3, 2, 1, 4 }
// => v3(0, 0, 0, 0) = a(0, 0, 0)
```

`xview` does not perform a copy of the underlying expression. This means if you modify an element of the `xview`, you are actually also altering the underlying expression.

```
#include <vector>
#include "xtensor/xarray.hpp"
#include "xtensor/xview.hpp"

std::vector<size_t> shape = {3, 2, 4};
xt::xarray<int> a(shape, 0);

auto v1 = xt::view(a, 1, xt::all(), xt::range(1, 3));
v1(0, 0) = 1;
// => a(1, 0, 1) = 1
```

### 1.7.2 Index views

Index views are one-dimensional views of an `xexpression`, containing the elements whose positions are specified by a list of indices. Like for sliced views, the elements of the underlying `xexpression` are not copied. Index views should be built with the `index_view` helper function.

```
#include "xtensor/xarray.hpp"
#include "xtensor/xindexview.hpp"

xt::xarray<double> a = {{1, 5, 3}, {4, 5, 6}};
auto b = xt::index_view(a, {{0,0}, {1, 0}, {0, 1}});
// => b = { 1, 4, 5 }
b += 100;
// => a = {{101, 5, 3}, {104, 105, 6}}
```

### 1.7.3 Filter views

Filters are one-dimensional views holding elements of an `xexpression` that verify a given condition. Like for other views, the elements of the underlying `xexpression` are not copied. Filters should be built with the `filter` helper function.

```
#include "xtensor/xarray.hpp"
#include "xtensor/xindexview.hpp"

xt::xarray<double> a = {{1, 5, 3}, {4, 5, 6}};
auto v = xt::filter(a, a >= 5);
// => v = { 5, 5, 6 }
v += 100;
// => a = {{1, 105, 3}, {4, 105, 106}}
```

### 1.7.4 Filtration

Sometimes, the only thing you want to do with a filter is to assign it a scalar. Though this can be done as shown in the previous section, this is not the *optimal* way to do it. *xtensor* provides a specially optimized mechanism for that, called filtration. A filtration IS NOT an `xexpression`, the only methods it provides are scalar and computed scalar assignments.

```
#include "xtensor/xarray.hpp"
#include "xtensor/xindexview.hpp"

xt::xarray<double> a = {{1, 5, 3}, {4, 5, 6}};
filtration(a, a >= 5) += 100;
// => a = {{1, 105, 3}, {4, 105, 106}}
```

### 1.7.5 Broadcasting views

Another type of view provided by *xtensor* is *broadcasting view*. Such a view broadcast an expression to the specified shape. As long as the view is not assigned to an array, no memory allocation or copy occurs. Broadcasting views should be built with the `broadcast` helper function.

```
#include <vector>
#include "xtensor/xarray.hpp"
#include "xtensor/xbroadcast.hpp"

std::vector<size_t> s1 = { 2, 3 };
std::vector<size_t> s2 = { 3, 2, 3 };

xt::xarray<int> a1(s1);
auto bv = xt::broadcast(a1, s2);
// => bv(0, 0, 0) = bv(1, 0, 0) = bv(2, 0, 0) = a(0, 0)
```

### 1.7.6 Complex views

In the case of tensor containing complex numbers, *xtensor* provides views returning `xexpression` corresponding to the real and imaginary parts of the complex numbers. Like for other views, the elements of the underlying `xexpression` are not copied.

Functions `xt::real` and `xt::imag` respectively return views on the real and imaginary part of a complex expression. The returned value is an expression holding a closure on the passed argument.

- The constness and value category (rvalue / lvalue) of `real(a)` is the same as that of `a`. Hence, if `a` is a non-const lvalue, `real(a)` is an non-const lvalue reference, to which one can assign a real expression.
- If `a` has complex values, the same holds for `imag(a)`. The constness and value category of `imag(a)` is the same as that of `a`.
- If `a` has real values, `imag(a)` returns `zeros(a.shape())`.

```
#include <complex>
#include "xtensor/xarray.hpp"
#include "xtensor/xcomplex.hpp"

using namespace std::complex_literals;

xarray<std::complex<double>> e =
    {{1.0       , 1.0 + 1.0i},
     {1.0 - 1.0i, 1.0       }};

real(e) = zeros<double>({2, 2});
// => e = {{0.0, 0.0 + 1.0i}, {0.0 - 1.0i, 0.0}};
```

## 1.8 Expression builders

*xtensor* provides functions to ease the build of common N-dimensional expressions. The expressions returned by these functions implement the lazyness of *xtensor*, that is, they don't hold any value. Values are computed upon request.

### 1.8.1 Ones and zeros

- `zeros(shape)`: generates an expression containing zeros of the specified shape.
- `ones(shape)`: generates an expression containing ones of the specified shape.
- `eye(shape, k=0)`: generates an expression of the specified shape, with ones on the k-th diagonal.
- `eye(n, k = 0)`: generates a n x n expression with ones on the k-th diagonal.

### 1.8.2 Numerical ranges

- `arange(start=0, stop, step=1)`: generates numbers evenly spaced within given half-open interval.
- `linspace(start, stop, num_samples)`: generates num_samples evenly spaced numbers over given interval.
- `logspace(start, stop, num_samples)`: generates num_samples evenly spaced on a log scale over given interval

### 1.8.3 Joining expressions

- `concatenate(tuple, axis=0)`: concatenates a list of expressions along the given axis.
- `stack(tuple, axis=0)`: stacks a list of expressions along the given axis.

### 1.8.4 Random distributions

- `rand(shape, lower, upper)`: generates an expression of the specified shape, containing uniformly distributed random numbers in the half-open interval [lower, upper).
- `randint(shape, lower, upper)`: generates an expression of the specified shape, containing uniformly distributed random integers in the half-open interval [lower, upper).
- `randn(shape, mean, std_dev)`: generates an expression of the specified shape, containing numbers sampled from the Normal random number distribution.

### 1.8.5 Meshes

- `meshgrid(x1, x2,...)`` : generates N-D coordinate expressions given one-dimensional coordinate arrays `x1`, `x2`... If specified vectors have lengths `Ni = len(xi)`, meshgrid returns `(N1, N2, N3,..., Nn)`-shaped arrays, with the elements of xi repeated to fill the matrix along the first dimension for x1, the second for x2 and so on.

## 1.9 Missing values

`xtensor` handles missing values and comprises specialized container types for an optimized support of missing values.

### 1.9.1 Optional expressions

Support of missing values in xtensor is primarily provided through the `xoptional` value type and the `xtensor_optional` and `xarray_optional` containers. In the following example, we instantiate a 2-D tensor with a missing value:

```
xtensor_optional<double, 2> m
    {{ 1.0 ,       2.0         },
     { 3.0 , missing<double>() }};
```

This code is semantically equivalent to

```
xtensor<xoptional<double>, 2> m
    {{ 1.0 ,       2.0         },
     { 3.0 , missing<double>() }};
```

The `xtensor_optional` container is optimized to handle missing values. Internally, instead of holding a single container of optional values, it holds an array of `double` and a boolean container where each value occupies a single bit instead of `sizeof(bool)` bytes.

The `xtensor_optional::reference` typedef, which is the return type of `operator()` is a reference proxy which can be used as an lvalue for assigning new values in the array. It happens to be an instance of `xoptional<T, B>` where `T` and `B` are actually the reference types of the underlying storage for values and boolean flags.

This technique enables performance improvements in mathematical operations over boolean arrays including SIMD optimizations, and reduces the memory footprint of optional arrays. It should be transparent to the user.

### 1.9.2 Operating on missing values

Arithmetic operators and mathematical universal functions are overloaded for optional values so that they can be operated upon in the same way as regular scalars.

```
xtensor_optional<double, 2> a
    {{ 1.0 ,       2.0         },
     { 3.0 , missing<double>() }};

xtensor<double, 1> b
    { 1.0, 2.0 };

// `b` is broadcasted to match the shape of `a`
std::cout << a + b << std::endl;
```

outputs:

```
{{  2,   4},
 {  4, N/A}}
```

### 1.9.3 Handling expressions with missing values

Functions `has_value(E&& e)` and `value(E&& e)` return expressions corresponding to the underlying value and flag of optional elements. When `e` is an lvalue, `value(E&& e)` and `has_value(E&& e)` are lvalues too.

```
xtensor_optional<double, 2> a
    {{ 1.0 ,       2.0         },
     { 3.0 , missing<double>() }};


xtensor<bool, 2> b = has_value(a);

std::cout << b << std::endl;
```

outputs:

```
{{  true,  true},
 {  true, false}}
```

## 1.10 Expressions and semantic

### 1.10.1 xexpression

Defined in `xtensor/xexpression.hpp`
**template** <class *D*>
**class** `xt::xexpression`

Base class for xexpressions.

The xexpression class is the base class for all classes representing an expression that can be evaluated to a multidimensional container with tensor semantic. Functions that can apply to any xexpression regardless of its specific type should take a xexpression argument.

**Template Parameters**

- `E`: The derived type.

Subclassed by *xt::xsemantic_base< D >*

#### Downcast functions

auto **derived_cast**()
Returns a reference to the actual derived type of the xexpression.

Returns a constant reference to the actual derived type of the xexpression.

auto **derived_cast**() **const**
Returns a constant reference to the actual derived type of the xexpression.

### 1.10.2 xsemantic_base

Defined in `xtensor/xsemantic.hpp`
**template** <class *D*>
**class** `xt::`**xsemantic_base**
Base interface for assignable xexpressions.

The *xsemantic_base* class defines the interface for assignable xexpressions.

**Template Parameters**

- `D`: The derived type, i.e. the inheriting class for which *xsemantic_base* provides the interface.

Inherits from *xt::xexpression< D >*

Subclassed by *xt::xadaptor_semantic< D >*, *xt::xcontainer_semantic< D >*, *xt::xview_semantic< D >*

#### Computed assignement

**template** <**class** E>
auto **operator+=**(**const** E &*e*)
Adds the scalar `e` to `*this`.

**Return** a reference to `*this`.

**Parameters**

- `e`: the scalar to add.

**template** <**class** E>
auto **operator-=**(**const** E &*e*)
Subtracts the scalar `e` from `*this`.

**Return** a reference to `*this`.

**Parameters**

- `e`: the scalar to subtract.

**template** <**class** E>
auto **operator*=**(**const** E &*e*)
Multiplies `*this` with the scalar `e`.

**Return** a reference to `*this`.

**Parameters**

- `e`: the scalar involved in the operation.

**template <class** E>
auto **operator/=**(**const** E &*e*)
Divides `*this` by the scalar `e`.

**Return** a reference to `*this`.

**Parameters**

- `e`: the scalar involved in the operation.

**template <class** E>
auto **operator+=**(**const** *xexpression*<E> &*e*)
Adds the xexpression `e` to `*this`.

**Return** a reference to `*this`.

**Parameters**

- `e`: the xexpression to add.

**template <class** E>
auto **operator-=**(**const** *xexpression*<E> &*e*)
Subtracts the xexpression `e` from `*this`.

**Return** a reference to `*this`.

**Parameters**

- `e`: the xexpression to subtract.

**template <class** E>
auto **operator*=**(**const** *xexpression*<E> &*e*)
Multiplies `*this` with the xexpression `e`.

**Return** a reference to `*this`.

**Parameters**

- `e`: the xexpression involved in the operation.

**template <class** E>
auto **operator/=**(**const** *xexpression*<E> &*e*)
Divides `*this` by the xexpression `e`.

**Return** a reference to `*this`.

**Parameters**

- `e`: the xexpression involved in the operation.

#### Assign functions

**template** <**class** E>
auto **assign**(**const** *xexpression*<E> &*e*)
Assigns the xexpression `e` to `*this`.

Ensures no temporary will be used to perform the assignment.

**Return** a reference to `*this`.

**Parameters**

- `e`: the xexpression to assign.

**template** <**class** E>
auto **plus_assign**(**const** *xexpression*<E> &*e*)
Adds the xexpression `e` to `*this`.

Ensures no temporary will be used to perform the assignment.

**Return** a reference to `*this`.

**Parameters**

- `e`: the xexpression to add.

**template** <**class** E>
auto **minus_assign**(**const** *xexpression*<E> &*e*)
Subtracts the xexpression `e` to `*this`.

Ensures no temporary will be used to perform the assignment.

**Return** a reference to `*this`.

**Parameters**

- `e`: the xexpression to subtract.

**template** <**class** E>
auto **multiplies_assign**(**const** *xexpression*<E> &*e*)
Multiplies `*this` with the xexpression `e`.

Ensures no temporary will be used to perform the assignment.

**Return** a reference to `*this`.

**Parameters**

- `e`: the xexpression involved in the operation.

**template** <**class** E>
auto **divides_assign**(**const** *xexpression*<E> &*e*)
Divides `*this` by the xexpression `e`.

Ensures no temporary will be used to perform the assignment.

**Return** a reference to `*this`.

**Parameters**

- `e`: the xexpression involved in the operation.

### 1.10.3 xcontainer_semantic

Defined in `xtensor/xsemantic.hpp`
**template** <class *D*>

**class** xt::**xcontainer_semantic**

Implementation of the *xsemantic_base* interface for dense multidimensional containers.

The *xcontainer_semantic* class is an implementation of the *xsemantic_base* interface for dense multidimensional containers.

**Template Parameters**

- D: the derived type

Inherits from *xt::xsemantic_base< D >*

#### Assign functions

auto **assign_temporary**(temporary_type &&*tmp*)

Assigns the temporary tmp to *this.

**Return** a reference to *this.

**Parameters**

- tmp: the temporary to assign.

### 1.10.4 xadaptor_semantic

Defined in xtensor/xsemantic.hpp

**template** <class *D*>

**class** xt::**xadaptor_semantic**

Implementation of the *xsemantic_base* interface for dense multidimensional container adaptors.

The *xadaptor_semantic* class is an implementation of the *xsemantic_base* interface for dense multidimensional container adaptors.

**Template Parameters**

- D: the derived type

Inherits from *xt::xsemantic_base< D >*

#### Assign functions

auto **assign_temporary**(temporary_type &&*tmp*)

Assigns the temporary tmp to *this.

**Return** a reference to *this.

**Parameters**

- tmp: the temporary to assign.

### 1.10.5 xview_semantic

Defined in `xtensor/xsemantic.hpp`
**template** <class *D*>
**class** `xt::xview_semantic`
Implementation of the *xsemantic_base* interface for multidimensional views.

The *xview_semantic* is an implementation of the *xsemantic_base* interface for multidimensional views.

**Template Parameters**

- `D`: the derived type

Inherits from *xt::xsemantic_base< D >*

#### Assign functions

auto **assign_temporary**(temporary_type &&*tmp*)
Assigns the temporary `tmp` to `*this`.

**Return** a reference to `*this`.

**Parameters**

- `tmp`: the temporary to assign.

### 1.10.6 xeval

Defined in `xtensor/xeval.hpp`
**template <class** T>
auto `xt::eval`(T &&*t*)
Force evaluation of xexpression.

```
xarray<double> a = {1,2,3,4};
auto&& b = xt::eval(a); // b is a reference to a, no copy!
auto&& c = xt::eval(a + b); // c is xarray<double>, not an xexpression
```

**Return** xarray or xtensor depending on shape type

## 1.11 Containers and views

### 1.11.1 layout

Defined in `xtensor/xlayout.hpp`

**enum** `xt::layout_type`
layout_type enum for xcontainer based xexpressions

*Values:*

**dynamic** = 0x00
dynamic layout_type: you can reshape to row major, column major, or use custom strides

**any** = 0xFF
layout_type compatible with all others

**row_major** = 0x01
row major layout_type

**column_major** = 0x02
column major layout_type

**template** <class... *Args*>
**constexpr** *layout_type* xt::**compute_layout**(Args... *args*)
Implementation of the following logical table:

```
  | d | a | r | c |
--+---+---+---+---+
d | d | d | d | d |
a | d | a | r | c |
r | d | r | r | d |
c | d | c | d | c |
d = dynamic, a = any, r = row_major, c = column_major.
```

Using bitmasks to avoid nested if-else statements.

**Return** the output layout, computed with the previous logical table.

**Parameters**

- `args`: the input layouts.

### 1.11.2 xcontainer

Defined in `xtensor/xcontainer.hpp`
**template** <class *D*>
**class** xt::**xcontainer**
Base class for dense multidimensional containers.

The xcontainer class defines the interface for dense multidimensional container classes. It does not embed any data container, this responsibility is delegated to the inheriting classes.

**Template Parameters**

- `D`: The derived type, i.e. the inheriting class for which xcontainer provides the interface.

Inherits from *xt::xiterable< D >*

Subclassed by *xt::xstrided_container< D >*

#### Size and shape

auto **size**() **const**
Returns the number of element in the container.

**constexpr** auto **dimension**() **const**
Returns the number of dimensions of the container.

auto **shape**() **const**
Returns the shape of the container.

auto **strides**() **const**
Returns the strides of the container.

auto **backstrides**() **const**
Returns the backstrides of the container.

### Data

auto **operator[]**(**const** xindex &*index*)
Returns a reference to the element at the specified position in the container.

**Parameters**

- `index`: a sequence of indices specifying the position in the container. Indices must be unsigned integers, the number of indices in the list should be equal or greater than the number of dimensions of the container.

auto **operator[]**(**const** xindex &*index*) **const**
Returns a constant reference to the element at the specified position in the container.

**Parameters**

- `index`: a sequence of indices specifying the position in the container. Indices must be unsigned integers, the number of indices in the list should be equal or greater than the number of dimensions of the container.

auto **data**()
Returns a reference to the buffer containing the elements of the container.

auto **data**() **const**
Returns a constant reference to the buffer containing the elements of the container.

auto **raw_data**()
Returns the offset to the first element in the container.

auto **raw_data_offset**() **const**
Returns the offset to the first element in the container.

**template** <class... *Args*>
auto **operator()**(Args... *args*)
Returns a reference to the element at the specified position in the container.

**Parameters**

- `args`: a list of indices specifying the position in the container. Indices must be unsigned integers, the number of indices should be equal or greater than the number of dimensions of the container.

**template** <class... *Args*>
auto **operator()**(Args... *args*) **const**
Returns a constant reference to the element at the specified position in the container.

**Parameters**

- `args`: a list of indices specifying the position in the container. Indices must be unsigned integers, the number of indices should be equal or greater than the number of dimensions of the container.

**template** <class... *Args*>

auto **at**(Args... *args*)

Returns a reference to the element at the specified position in the expression, after dimension and bounds checking.

**Parameters**

- `args`: a list of indices specifying the position in the function. Indices must be unsigned integers, the number of indices should be equal to the number of dimensions of the expression.

**Exceptions**

- `std::out_of_range`: if the number of argument is greater than the number of dimensions or if indices are out of bounds.

**template** <class... *Args*>
auto **at**(Args... *args*) **const**

Returns a constant reference to the element at the specified position in the expression, after dimension and bounds checking.

**Parameters**

- `args`: a list of indices specifying the position in the function. Indices must be unsigned integers, the number of indices should be equal to the number of dimensions of the expression.

**Exceptions**

- `std::out_of_range`: if the number of argument is greater than the number of dimensions or if indices are out of bounds.

**template** <**class** It>
auto **element**(It *first*, It *last*)

Returns a reference to the element at the specified position in the container.

**Parameters**

- `first`: iterator starting the sequence of indices
- `last`: iterator ending the sequence of indices The number of indices in the sequence should be equal to or greater than the number of dimensions of the container.

**template** <**class** It>
auto **element**(It *first*, It *last*) **const**

Returns a reference to the element at the specified position in the container.

**Parameters**

- `first`: iterator starting the sequence of indices
- `last`: iterator ending the sequence of indices The number of indices in the sequence should be equal to or greater than the number of dimensions of the container.

### Broadcasting

**template** <**class** S>
bool **broadcast_shape**(S &*shape*) **const**

Broadcast the shape of the container to the specified parameter.

**Return** a boolean indicating whether the broadcasting is trivial

**Parameters**

- `shape`: the result shape

**template <class** S>
bool **is_trivial_broadcast**(**const** S &*strides*) **const**
Compares the specified strides with those of the container to see whether the broadcasting is trivial.

**Return** a boolean indicating whether the broadcasting is trivial

### 1.11.3 xstrided_container

Defined in `xtensor/xcontainer.hpp`
**template** <class *D*>
**class** `xt::`**xstrided_container**
Partial implementation of xcontainer that embeds the strides and the shape.

The *xstrided_container* class is a partial implementation of the xcontainer interface that embed the strides and the shape of the multidimensional container. It does not embed the data container, this responsibility is delegated to the inheriting classes.

**Template Parameters**

- `D`: The derived type, i.e. the inheriting class for which *xstrided_container* provides the partial imlpementation of xcontainer.

Inherits from *xt::xcontainer< D >*

#### Public Functions

**template <class** S = shape_type>
void **reshape**(**const** S &*shape*, bool *force* = false)
Reshapes the container.

**Parameters**

- `shape`: the new shape
- `force`: force reshaping, even if the shape stays the same (default: false)

**template <class** S = shape_type>
void **reshape**(**const** S &*shape*, *layout_type l*)
Reshapes the container.

**Parameters**

- `shape`: the new shape
- `l`: the new layout_type

**template <class** S = shape_type>
void **reshape**(**const** S &*shape*, **const** strides_type &*strides*)
Reshapes the container.

**Parameters**

- `shape`: the new shape

- `strides`: the new strides

*layout_type* **layout**() **const**
Return the layout_type of the container.

**Return** layout_type of the container

### 1.11.4 xiterable

Defined in `xtensor/xiterable.hpp`
**template** <class *D*>
**class** `xt::`**xconst_iterable**
Base class for multidimensional iterable constant expressions.

The *xconst_iterable* class defines the interface for multidimensional constant expressions that can be iterated.

**Template Parameters**

- `D`: The derived type, i.e. the inheriting class for which *xconst_iterable* provides the interface.

Subclassed by *xt::xiterable< D >*

#### Constant iterators

**template** <*layout_type L*>
auto **begin**() **const**
Returns a constant iterator to the first element of the expression.

**Template Parameters**

- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

**template** <*layout_type L*>
auto **end**() **const**
Returns a constant iterator to the element following the last element of the expression.

**Template Parameters**

- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

**template** <*layout_type L*>
auto **cbegin**() **const**
Returns a constant iterator to the first element of the expression.

**Template Parameters**

- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

**template** <*layout_type L*>
auto **cend**() **const**
Returns a constant iterator to the element following the last element of the expression.

**Template Parameters**

- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

### Constant reverse iterators

**template** <*layout_type L*>
auto **rbegin**() **const**
Returns a constant iterator to the first element of the reversed expression.

**Template Parameters**

- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

**template** <*layout_type L*>
auto **rend**() **const**
Returns a constant iterator to the element following the last element of the reversed expression.

**Template Parameters**

- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

**template** <*layout_type L*>
auto **crbegin**() **const**
Returns a constant iterator to the first element of the reversed expression.

**Template Parameters**

- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

**template** <*layout_type L*>
auto **crend**() **const**
Returns a constant iterator to the element following the last element of the reversed expression.

**Template Parameters**

- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

### Constant broadcast iterators

**template** <class *S*, *layout_type L*>
auto **begin**(**const** S &*shape*) **const**
Returns a constant iterator to the first element of the expression.

The iteration is broadcasted to the specified shape.

**Parameters**

- `shape`: the shape used for broadcasting

**Template Parameters**

- `S`: type of the `shape` parameter.
- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

**template** <class *S*, *layout_type L*>
auto **end**(**const** S &*shape*) **const**
Returns a constant iterator to the element following the last element of the expression.

The iteration is broadcasted to the specified shape.

**Parameters**

- `shape`: the shape used for broadcasting

**Template Parameters**

- `S`: type of the `shape` parameter.
- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

**template** <class *S*, *layout_type L*>
auto **cbegin**(**const** S &*shape*) **const**

Returns a constant iterator to the first element of the expression.

The iteration is broadcasted to the specified shape.

**Parameters**

- `shape`: the shape used for broadcasting

**Template Parameters**

- `S`: type of the `shape` parameter.
- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

**template** <class *S*, *layout_type L*>
auto **cend**(**const** S &*shape*) **const**

Returns a constant iterator to the element following the last element of the expression.

The iteration is broadcasted to the specified shape.

**Parameters**

- `shape`: the shape used for broadcasting

**Template Parameters**

- `S`: type of the `shape` parameter.
- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

### Unnamed Group

**template** <class *S*, *layout_type L*>
auto **rbegin**(**const** S &*shape*) **const**

Constant reverse broadcast iterators.

Returns a constant iterator to the first element of the reversed expression. The iteration is broadcasted to the specified shape.

**Parameters**

- `shape`: the shape used for broadcasting

**Template Parameters**

- `S`: type of the `shape` parameter.
- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

**template** <class *S*, *layout_type L*>
auto **rend**(**const** S &*shape*) **const**

Returns a constant iterator to the element following the last element of the reversed expression.

The iteration is broadcasted to the specified shape.

**Parameters**

- `shape`: the shape used for broadcasting

**Template Parameters**

- `S`: type of the `shape` parameter.
- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

**template** <class *S*, *layout_type L*>
auto **crbegin**(**const** S &*shape*) **const**

Returns a constant iterator to the first element of the reversed expression.

The iteration is broadcasted to the specified shape.

**Parameters**

- `shape`: the shape used for broadcasting

**Template Parameters**

- `S`: type of the `shape` parameter.
- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

**template** <class *S*, *layout_type L*>
auto **crend**(**const** S &*shape*) **const**

Returns a constant iterator to the element following the last element of the reversed expression.

The iteration is broadcasted to the specified shape.

**Parameters**

- `shape`: the shape used for broadcasting

**Template Parameters**

- `S`: type of the `shape` parameter.
- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

**template** <class *D*>
**class** xt::**xiterable**

Base class for multidimensional iterable expressions.

The xiterable class defines the interface for multidimensional expressions that can be iterated.

**Template Parameters**

- `D`: The derived type, i.e. the inheriting class for which xiterable provides the interface.

Inherits from *xt::xconst_iterable< D >*

Subclassed by *xt::xcontainer< D >*

### Iterators

**template** <*layout_type L*>
auto **begin**()

Returns an iterator to the first element of the expression.

**Template Parameters**

- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

**template** <*layout_type L*>
auto **end**()

Returns an iterator to the element following the last element of the expression.

**Template Parameters**

- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

### Broadcast iterators

**template** <class *S*, *layout_type* *L*>
auto **begin**(**const** S &*shape*)

Returns an iterator to the first element of the expression.

The iteration is broadcasted to the specified shape.

**Parameters**

- `shape`: the shape used for broadcasting

**Template Parameters**

- `S`: type of the `shape` parameter.
- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

**template** <class *S*, *layout_type* *L*>
auto **end**(**const** S &*shape*)

Returns an iterator to the element following the last element of the expression.

The iteration is broadcasted to the specified shape.

**Parameters**

- `shape`: the shape used for broadcasting

**Template Parameters**

- `S`: type of the `shape` parameter.
- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

### Reverse iterators

**template** <*layout_type* *L*>
auto **rbegin**()

Returns an iterator to the first element of the reversed expression.

**Template Parameters**

- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

**template** <*layout_type* *L*>
auto **rend**()

Returns an iterator to the element following the last element of the reversed expression.

**Template Parameters**

- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

#### Reverse broadcast iterators

**template** <class *S*, *layout_type L*>
auto **rbegin**(**const** S &*shape*)
Returns an iterator to the first element of the reversed expression.

The iteration is broadcasted to the specified shape.

**Parameters**

- `shape`: the shape used for broadcasting

**Template Parameters**

- `S`: type of the `shape` parameter.
- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

**template** <class *S*, *layout_type L*>
auto **rend**(**const** S &*shape*)
Returns an iterator to the element following the last element of the reversed expression.

The iteration is broadcasted to the specified shape.

**Parameters**

- `shape`: the shape used for broadcasting

**Template Parameters**

- `S`: type of the `shape` parameter.
- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

### 1.11.5 xarray

Defined in `xtensor/xarray.hpp`
**template** <class *EC*, *layout_type L*, class *SC*>
**class** xt::**xarray_container**
Dense multidimensional container with tensor semantic.

The *xarray_container* class implements a dense multidimensional container with tensor semantic.

**See** *xarray*

**Template Parameters**

- `EC`: The type of the container holding the elements.
- `L`: The layout_type of the container.
- `SC`: The type of the containers holding the shape and the strides.

Inherits from *xt::xstrided_container< xarray_container< EC, L, SC > >*, *xt::xcontainer_semantic< xarray_container< EC, L, SC > >*

#### Constructors

**xarray_container**()
Allocates an uninitialized *xarray_container* that holds 0 element.

**xarray_container**(**const** shape_type &*shape*, *layout_type* *l* = L)
Allocates an uninitialized *xarray_container* with the specified shape and layout_type.

**Parameters**

- `shape`: the shape of the *xarray_container*
- `l`: the layout_type of the *xarray_container*

**xarray_container**(**const** shape_type &*shape*, const_reference *value*, *layout_type* *l* = L)
Allocates an *xarray_container* with the specified shape and layout_type.

Elements are initialized to the specified value.

**Parameters**

- `shape`: the shape of the *xarray_container*
- `value`: the value of the elements
- `l`: the layout_type of the *xarray_container*

**xarray_container**(**const** shape_type &*shape*, **const** strides_type &*strides*)
Allocates an uninitialized *xarray_container* with the specified shape and strides.

**Parameters**

- `shape`: the shape of the *xarray_container*
- `strides`: the strides of the *xarray_container*

**xarray_container**(**const** shape_type &*shape*, **const** strides_type &*strides*, const_reference *value*)
Allocates an uninitialized *xarray_container* with the specified shape and strides.

Elements are initialized to the specified value.

**Parameters**

- `shape`: the shape of the *xarray_container*
- `strides`: the strides of the *xarray_container*
- `value`: the value of the elements

**xarray_container**(container_type &&*data*, inner_shape_type &&*shape*, inner_strides_type &&*strides*)
Allocates an *xarray_container* by moving specified data, shape and strides.

**Parameters**

- `data`: the data for the *xarray_container*
- `shape`: the shape of the *xarray_container*
- `strides`: the strides of the *xarray_container*

**xarray_container**(**const** value_type &*t*)
Allocates an *xarray_container* that holds a single element initialized to the specified value.

**Parameters**

- `t`: the value of the element

### Constructors from initializer list

**xarray_container**(nested_initializer_list_t<value_type, 1> *t*)
Allocates a one-dimensional *xarray_container*.

**Parameters**

- `t`: the elements of the *xarray_container*

**xarray_container**(nested_initializer_list_t<value_type, 2> *t*)
Allocates a two-dimensional *xarray_container*.

**Parameters**

- `t`: the elements of the *xarray_container*

**xarray_container**(nested_initializer_list_t<value_type, 3> *t*)
Allocates a three-dimensional *xarray_container*.

**Parameters**

- `t`: the elements of the *xarray_container*

**xarray_container**(nested_initializer_list_t<value_type, 4> *t*)
Allocates a four-dimensional *xarray_container*.

**Parameters**

- `t`: the elements of the *xarray_container*

**xarray_container**(nested_initializer_list_t<value_type, 5> *t*)
Allocates a five-dimensional *xarray_container*.

**Parameters**

- `t`: the elements of the *xarray_container*

### Extended copy semantic

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

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

**typedef** xt::**xarray**
Alias template on *xarray_container* with default parameters for data container type and shape / strides container type.

This allows to write

```
xt::xarray<double> a = {{1., 2.}, {3., 4.}};
```

instead of the heavier syntax

```
xt::xarray_container<std::vector<double>, std::vector<std::size_t>> a = ...
```

**Template Parameters**

- `T`: The value type of the elements.
- `L`: The layout_type of the *xarray_container* (default: row_major).
- `A`: The allocator of the container holding the elements.
- `SA`: The allocator of the containers holding the shape and the strides.

### 1.11.6 xarray_adaptor

Defined in `xtensor/xarray.hpp`

**template** <class *EC*, *layout_type* *L*, class *SC*>
**class** `xt::`**xarray_adaptor**

Dense multidimensional container adaptor with tensor semantic.

The *xarray_adaptor* class implements a dense multidimensional container adaptor with tensor semantic. It is used to provide a multidimensional container semantic and a tensor semantic to stl-like containers.

**Template Parameters**

- `EC`: The container type to adapt.
- `L`: The layout_type of the adaptor.
- `SC`: The type of the containers holding the shape and the strides.

Inherits from *xt::xstrided_container< xarray_adaptor< EC, L, SC > >*, *xt::xadaptor_semantic< xarray_adaptor< EC, L, SC > >*

#### Constructors

**xarray_adaptor**(container_closure_type *data*)

Constructs an *xarray_adaptor* of the given stl-like container.

**Parameters**

- `data`: the container to adapt

**xarray_adaptor**(container_closure_type *data*, **const** shape_type &*shape*, *layout_type* *l* = L)

Constructs an *xarray_adaptor* of the given stl-like container, with the specified shape and layout_type.

**Parameters**

- `data`: the container to adapt
- `shape`: the shape of the *xarray_adaptor*
- `l`: the layout_type of the *xarray_adaptor*

**xarray_adaptor**(container_closure_type *data*, **const** shape_type &*shape*, **const** strides_type &*strides*)

Constructs an *xarray_adaptor* of the given stl-like container, with the specified shape and strides.

**Parameters**

- `data`: the container to adapt
- `shape`: the shape of the *xarray_adaptor*
- `strides`: the strides of the *xarray_adaptor*

#### Extended copy semantic

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

### 1.11.7 xadapt (xarray_adaptor)

Defined in `xtensor/xadapt.hpp`
**template** <**class** C, **class** SC, layout_type *L* = DEFAULT_LAYOUT>
std::enable_if_t<!detail::is_array< SC >::value, *xarray_adaptor*<C, L, SC>> `xt::xadapt`(C &*container*, **const** SC &*shape*, *layout_type* *l* = L)
Constructs an *xarray_adaptor* of the given stl-like container, with the specified shape and layout.

**Parameters**

- `container`: the container to adapt
- `shape`: the shape of the *xarray_adaptor*
- `l`: the layout_type of the *xarray_adaptor*

**template** <**class** C, **class** SC>
std::enable_if_t<!detail::is_array< SC >::value, *xarray_adaptor*<C, *layout_type*::dynamic, SC>> `xt::xadapt`(C &*container*, **const** SC &*shape*, **const** SC &*strides*)
Constructs an *xarray_adaptor* of the given stl-like container, with the specified shape and strides.

**Parameters**

- `container`: the container to adapt
- `shape`: the shape of the *xarray_adaptor*
- `strides`: the strides of the *xarray_adaptor*

**template** <**class** P, **class** O, **class** SC, layout_type *L* = DEFAULT_LAYOUT, **class** A = std::allocator<std::remove_pointer

std::enable_if_t<!detail::is_array< SC >::value, *xarray_adaptor*<xbuffer_adaptor<std::remove_pointer_t<P>, O, A>, L, SC>> `xt::xa`

Constructs an *xarray_adaptor* of the given dynamically allocated C array, with the specified shape and layout.

**Parameters**

- `pointer`: the pointer to the beginning of the dynamic array
- `size`: the size of the dynamic array
- `ownership`: indicates whether the adaptor takes ownership of the array. Possible values are `no_ownerhsip()` or `accept_ownership()`
- `shape`: the shape of the *xarray_adaptor*
- `l`: the layout_type of the *xarray_adaptor*
- `alloc`: the allocator used for allocating / deallocating the dynamic array

**template** <**class** P, **class** O, **class** SC, **class** A = std::allocator<std::remove_pointer_t<P>>>

std::enable_if_t<!detail::is_array< SC >::value, *xarray_adaptor*<xbuffer_adaptor<std::remove_pointer_t<P>, O, A>, *layout_type*::dyna

Constructs an *xarray_adaptor* of the given dynamically allocated C array, with the specified shape and layout.

**Parameters**

- `pointer`: the pointer to the beginning of the dynamic array
- `size`: the size of the dynamic array
- `ownership`: indicates whether the adaptor takes ownership of the array. Possible values are `no_ownerhsip()` or `accept_ownership()`
- `shape`: the shape of the *xarray_adaptor*
- `strides`: the strides of the *xarray_adaptor*
- `alloc`: the allocator used for allocating / deallocating the dynamic array

### 1.11.8 xtensor

Defined in `xtensor/xtensor.hpp`

**template** <class *EC*, size_t *N*, *layout_type* *L*>
**class** `xt::xtensor_container`

Dense multidimensional container with tensor semantic and fixed dimension.

The *xtensor_container* class implements a dense multidimensional container with tensor semantic and fixed dimension

**See** *xtensor*

**Template Parameters**

- `EC`: The type of the container holding the elements.
- `N`: The dimension of the container.
- `L`: The layout_type of the tensor.

Inherits from *xt::xstrided_container< xtensor_container< EC, N, L > >*, *xt::xcontainer_semantic< xtensor_container< EC, N, L > >*

### Constructors

**xtensor_container**()
Allocates an uninitialized *xtensor_container* that holds 0 element.

**xtensor_container**(nested_initializer_list_t<value_type, N> *t*)
Allocates an *xtensor_container* with nested initializer lists.

**xtensor_container**(**const** shape_type &*shape*, *layout_type* *l* = L)
Allocates an uninitialized *xtensor_container* with the specified shape and layout_type.

**Parameters**

- `shape`: the shape of the *xtensor_container*
- `l`: the layout_type of the *xtensor_container*

**xtensor_container**(**const** shape_type &*shape*, const_reference *value*, *layout_type* *l* = L)
Allocates an *xtensor_container* with the specified shape and layout_type.

Elements are initialized to the specified value.

**Parameters**

- `shape`: the shape of the *xtensor_container*
- `value`: the value of the elements
- `l`: the layout_type of the *xtensor_container*

**xtensor_container**(**const** shape_type &*shape*, **const** strides_type &*strides*)
Allocates an uninitialized *xtensor_container* with the specified shape and strides.

**Parameters**

- `shape`: the shape of the *xtensor_container*
- `strides`: the strides of the *xtensor_container*

**xtensor_container**(**const** shape_type &*shape*, **const** strides_type &*strides*, const_reference *value*)
Allocates an uninitialized *xtensor_container* with the specified shape and strides.

Elements are initialized to the specified value.

**Parameters**

- `shape`: the shape of the *xtensor_container*
- `strides`: the strides of the *xtensor_container*
- `value`: the value of the elements

**xtensor_container**(container_type &&*data*, inner_shape_type &&*shape*, inner_strides_type &&*strides*)
Allocates an *xtensor_container* by moving specified data, shape and strides.

**Parameters**

- `data`: the data for the *xtensor_container*
- `shape`: the shape of the *xtensor_container*
- `strides`: the strides of the *xtensor_container*

#### Extended copy semantic

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

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

**typedef** xt::**xtensor**
Alias template on *xtensor_container* with default parameters for data container type.

This allows to write

```
xt::xtensor<double, 2> a = {{1., 2.}, {3., 4.}};
```

instead of the heavier syntax

```
xt::xtensor_container<std::vector<double>, 2> a = ...
```

**Template Parameters**

- `T`: The value type of the elements.
- `N`: The dimension of the tensor.
- `L`: The layout_type of the tensor (default: row_major).
- `A`: The allocator of the containers holding the elements.

### 1.11.9 xtensor_adaptor

Defined in `xtensor/xtensor.hpp`
**template** <class *EC*, std::size_t *N*, *layout_type L*>
**class** xt::**xtensor_adaptor**
Dense multidimensional container adaptor with tensor semantic and fixed dimension.

The *xtensor_adaptor* class implements a dense multidimensional container adaptor with tensor semantic and fixed dimension. It is used to provide a multidimensional container semantic and a tensor semantic to stl-like containers.

**Template Parameters**

- `EC`: The container type to adapt.
- `N`: The dimension of the adaptor.
- `L`: The layout_type of the adaptor.

Inherits from *xt::xstrided_container< xtensor_adaptor< EC, N, L > >*, *xt::xadaptor_semantic< xtensor_adaptor< EC, N, L > >*

### Constructors

**xtensor_adaptor**(container_closure_type *data*)
Constructs an *xtensor_adaptor* of the given stl-like container.

**Parameters**

- `data`: the container to adapt

**xtensor_adaptor**(container_closure_type *data*, **const** shape_type &*shape*, *layout_type* *l* = layout_type::row_major)
Constructs an *xtensor_adaptor* of the given stl-like container, with the specified shape and layout_type.

**Parameters**

- `data`: the container to adapt
- `shape`: the shape of the *xtensor_adaptor*
- `l`: the layout_type of the *xtensor_adaptor*

**xtensor_adaptor**(container_closure_type *data*, **const** shape_type &*shape*, **const** strides_type &*strides*)
Constructs an *xtensor_adaptor* of the given stl-like container, with the specified shape and strides.

**Parameters**

- `data`: the container to adapt
- `shape`: the shape of the *xtensor_adaptor*
- `strides`: the strides of the *xtensor_adaptor*

### Extended copy semantic

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

## 1.11.10 xadapt (xtensor_adaptor)

Defined in `xtensor/xadapt.hpp`
**template** <**class** C, std::size_t *N*, layout_type *L* = DEFAULT_LAYOUT>
*xtensor_adaptor*<C, N, L> `xt::xadapt`(C &*container*, **const** std::array<**typename** C::size_type, N> &*shape*, *layout_type* *l* = L)
Constructs an *xtensor_adaptor* of the given stl-like container, with the specified shape and layout_type.

**Parameters**

- `container`: the container to adapt
- `shape`: the shape of the *xtensor_adaptor*
- `l`: the layout_type of the *xtensor_adaptor*

**template** <**class** C, std::size_t *N*>

*xtensor_adaptor*<C, N, *layout_type*::dynamic> `xt::xadapt`(C &*container*, **const** std::array<**typename** C::size_type, N> &*shape*, **const** std::array<**typename** C::size_type, N> &*strides*)

Constructs an *xtensor_adaptor* of the given stl-like container, with the specified shape and strides.

**Parameters**

- `container`: the container to adapt
- `shape`: the shape of the *xtensor_adaptor*
- `strides`: the strides of the *xtensor_adaptor*

**template** <**class** P, std::size_t *N*, **class** O, layout_type *L* = DEFAULT_LAYOUT, **class** A = std::allocator<std::remove_pointe
*xtensor_adaptor*<xbuffer_adaptor<std::remove_pointer_t<P>, O, A>, N, L> `xt::xadapt`(P &*pointer*, **typename** A::size_type *size*, O *ownership*, **const** std::array<**typename** A::size_type, N> &*shape*, *layout_type* *l* = L, **const** A &*alloc* = A())

Constructs an *xtensor_adaptor* of the given dynamically allocated C array, with the specified shape and layout.

**Parameters**

- `pointer`: the pointer to the beginning of the dynamic array
- `size`: the size of the dynamic array
- `ownership`: indicates whether the adaptor takes ownership of the array. Possible values are `no_ownerhsip()` or `accept_ownership()`
- `shape`: the shape of the *xtensor_adaptor*
- `l`: the layout_type of the *xtensor_adaptor*
- `alloc`: the allocator used for allocating / deallocating the dynamic array

**template** <**class** P, std::size_t *N*, **class** O, **class** A = std::allocator<std::remove_pointer_t<P>>>

*xtensor_adaptor*<xbuffer_adaptor<std::remove_pointer_t<P>, O, A>, N, *layout_type*::dynamic> `xt::xadapt`(P *&pointer*, **typename** A::size_type *size*, O *ownership*, **const** std::array<**typename** A::size_type, N> *&shape*, **const** std::array<**typename** A::size_type, N> *&strides*, **const** A *&alloc* = A())

Constructs an *xtensor_adaptor* of the given dynamically allocated C array, with the specified shape and layout.

**Parameters**

- `pointer`: the pointer to the beginning of the dynamic array
- `size`: the size of the dynamic array
- `ownership`: indicates whether the adaptor takes ownership of the array. Possible values are `no_ownerhsip()` or `accept_ownership()`
- `shape`: the shape of the *xtensor_adaptor*
- `strides`: the strides of the *xtensor_adaptor*
- `alloc`: the allocator used for allocating / deallocating the dynamic array

### 1.11.11 xview

Defined in `xtensor/xview.hpp`

**template** <class *CT*, class... *S*>
**class** `xt::xview`

Multidimensional view with tensor semantic.

The xview class implements a multidimensional view with tensor semantic. It is used to adapt the shape of an xexpression without changing it. xview is not meant to be used directly, but only with the *view* helper functions.

**See** *view*, range, *all*, *newaxis*

**Template Parameters**

- `CT`: the closure type of the *xexpression* to adapt

- `S`: the slices type describing the shape adaptation

Inherits from *xt::xview_semantic< xview< CT, S... > >*, *xt::xiterable< xview< CT, S... > >*

### Extended copy semantic

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

### Constructor

**template <class** CTA, **class** FSL, class... *SL*>
**xview**(CTA &&*e*, FSL &&*first_slice*, SL&&... *slices*)
Constructs a view on the specified xexpression.

Users should not call directly this constructor but use the view function instead.

**See** *view*

**Parameters**

- `e`: the xexpression to adapt
- `first_slice`: the first slice describing the view
- `slices`: the slices list describing the view

### Size and shape

auto **dimension**() **const**
Returns the number of dimensions of the view.

auto **size**() **const**
Returns the size of the expression.

auto **shape**() **const**
Returns the shape of the view.

auto **slices**() **const**
Returns the slices of the view.

*layout_type* **layout**() **const**
Returns the slices of the view.

### Data

**template** <class... *Args*>
auto **operator()**(Args... *args*)
Returns a reference to the element at the specified position in the view.

**Parameters**

- `args`: a list of indices specifying the position in the view. Indices must be unsigned integers, the number of indices should be equal or greater than the number of dimensions of the view.

**template** <class... *Args*>
auto **at**(Args... *args*)

Returns a reference to the element at the specified position in the expression, after dimension and bounds checking.

**Parameters**

- `args`: a list of indices specifying the position in the function. Indices must be unsigned integers, the number of indices should be equal to the number of dimensions of the expression.

**Exceptions**

- `std::out_of_range`: if the number of argument is greater than the number of dimensions or if indices are out of bounds.

**template** <class... *Args*>
auto **operator()**(Args... *args*) **const**

Returns a constant reference to the element at the specified position in the view.

**Parameters**

- `args`: a list of indices specifying the position in the view. Indices must be unsigned integers, the number of indices should be equal or greater than the number of dimensions of the view.

**template** <class... *Args*>
auto **at**(Args... *args*) **const**

Returns a constant reference to the element at the specified position in the expression, after dimension and bounds checking.

**Parameters**

- `args`: a list of indices specifying the position in the function. Indices must be unsigned integers, the number of indices should be equal to the number of dimensions of the expression.

**Exceptions**

- `std::out_of_range`: if the number of argument is greater than the number of dimensions or if indices are out of bounds.

**template <class** T>
auto **data**() **const**

Returns the data holder of the underlying container (only if the view is on a realized container).

*xt::eval* will make sure that the underlying xexpression is on a realized container.

**template <class** T>
auto **strides**() **const**

Return the strides for the underlying container of the view.

**template <class** T>
auto **raw_data**() **const**

Return the pointer to the underlying buffer.

**template <class** T>
auto **raw_data_offset**() **const**

Return the offset to the first element of the view in the underlying container.

### Broadcasting

**template <class** ST>

bool **broadcast_shape**(ST &*shape*) **const**
Broadcast the shape of the view to the specified parameter.

**Return** a boolean indicating whether the broadcasting is trivial

**Parameters**

- `shape`: the result shape

**template <class** ST>
bool **is_trivial_broadcast**(**const** ST &*strides*) **const**
Compares the specified strides with those of the view to see whether the broadcasting is trivial.

**Return** a boolean indicating whether the broadcasting is trivial

**template <class** E, class... *S*>
auto `xt::view`(E &&*e*, S&&... *slices*)
Constructs and returns a view on the specified xexpression.

Users should not directly construct the slices but call helper functions instead.

**See** range, *all*, *newaxis*

**Parameters**

- `e`: the xexpression to adapt
- `slices`: the slices list describing the view

Defined in `xtensor/xslice.hpp`

**Warning:** doxygenfunction: Unable to resolve multiple matches for function "xt::range" with arguments (T, T) in doxygen xml output for project "xtensor" from directory: ../xml. Potential matches:

```
- template <class A, class B, class C>
  auto xt::range(A, B, C)
- template <class A, class B>
  auto xt::range(A, B)
```

**Warning:** doxygenfunction: Unable to resolve multiple matches for function "xt::range" with arguments (T, T, T) in doxygen xml output for project "xtensor" from directory: ../xml. Potential matches:

```
- template <class A, class B, class C>
  auto xt::range(A, B, C)
- template <class A, class B>
  auto xt::range(A, B)
```

auto `xt::all`()
Returns a slice representing a full dimension, to be used as an argument of view function.

**See** *view*

auto `xt::newaxis`()
Returns a slice representing a new axis of length one, to be used as an argument of view function.

**See** *view*

### 1.11.12 xbroadcast

Defined in `xtensor/xbroadcast.hpp`
**template** <class *CT*, class *X*>
**class** `xt::xbroadcast`

Broadcasted xexpression to a specified shape.

The xbroadcast class implements the broadcasting of an *xexpression* to a specified shape. xbroadcast is not meant to be used directly, but only with the *broadcast* helper functions.

**See** *broadcast*

**Template Parameters**

- `CT`: the closure type of the *xexpression* to broadcast
- `X`: the type of the specified shape.

Inherits from *xt::xexpression< xbroadcast< CT, X > >*, *xt::xconst_iterable< xbroadcast< CT, X > >*

#### Constructor

**template** <**class** CTA, **class** S>
**xbroadcast**(CTA &&*e*, S &&*s*)

Constructs an xbroadcast expression broadcasting the specified *xexpression* to the given shape.

**Parameters**

- `e`: the expression to broadcast
- `s`: the shape to apply

#### Size and shape

auto **size**() **const**

Returns the size of the expression.

auto **dimension**() **const**

Returns the number of dimensions of the expression.

auto **shape**() **const**

Returns the shape of the expression.

*layout_type* **layout**() **const**

Returns the layout_type of the expression.

#### Data

auto **operator[]**(**const** xindex &*index*) **const**

Returns a constant reference to the element at the specified position in the expression.

**Parameters**

- `index`: a sequence of indices specifying the position in the function. Indices must be unsigned integers, the number of indices in the sequence should be equal or greater than the number of dimensions of the container.

**template** <class... *Args*>
auto **operator()**(Args... *args*) **const**

Returns a constant reference to the element at the specified position in the expression.

**Parameters**

- `args`: a list of indices specifying the position in the function. Indices must be unsigned integers, the number of indices should be equal or greater than the number of dimensions of the expression.

**template** <class... *Args*>
auto **at**(Args... *args*) **const**

Returns a constant reference to the element at the specified position in the expression, after dimension and bounds checking.

**Parameters**

- `args`: a list of indices specifying the position in the function. Indices must be unsigned integers, the number of indices should be equal to the number of dimensions of the expression.

**Exceptions**

- `std::out_of_range`: if the number of argument is greater than the number of dimensions or if indices are out of bounds.

**template <class** It>
auto **element**(It, It *last*) **const**

Returns a constant reference to the element at the specified position in the expression.

**Parameters**

- `first`: iterator starting the sequence of indices
- `last`: iterator ending the sequence of indices The number of indices in the sequence should be equal to or greater than the number of dimensions of the function.

### Broadcasting

**template <class** S>
bool **broadcast_shape**(S &*shape*) **const**

Broadcast the shape of the function to the specified parameter.

**Return** a boolean indicating whether the broadcasting is trivial

**Parameters**

- `shape`: the result shape

**template <class** S>
bool **is_trivial_broadcast**(**const** S &*strides*) **const**

Compares the specified strides with those of the container to see whether the broadcasting is trivial.

**Return** a boolean indicating whether the broadcasting is trivial

**template <class** E, **class** S>

auto xt::**broadcast**(E &&*e*, **const** S &*s*)

Returns an *xexpression* broadcasting the given expression to a specified shape.

The returned expression either hold a const reference to e or a copy depending on whether e is an lvalue or an rvalue.

**Template Parameters**

- e: the *xexpression* to broadcast
- s: the specified shape to broadcast.

### 1.11.13 xindexview

Defined in xtensor/xindexview.hpp

**template** <class *CT*, class *I*>
**class** xt::**xindexview**

View of an xexpression from vector of indices.

The xindexview class implements a flat (1D) view into a multidimensional xexpression yielding the values at the indices of the index array. xindexview is not meant to be used directly, but only with the *index_view* and *filter* helper functions.

**See** *index_view*, *filter*

**Template Parameters**

- CT: the closure type of the *xexpression* type underlying this view
- I: the index array type of the view

Inherits from *xt::xview_semantic< xindexview< CT, I > >*, *xt::xiterable< xindexview< CT, I > >*

#### Extended copy semantic

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

The extended assignment operator.

#### Constructor

**template** <**class** I2>
**xindexview**(CT *e*, I2 &&*indices*)

Constructs an xindexview, selecting the indices specified by *indices*.

The resulting xexpression has a 1D shape with a length of n for n indices.

**Parameters**

- e: the underlying xexpression for this view
- indices: the indices to select

### Size and shape

auto **size**() **const**
Returns the size of the xindexview.

auto **dimension**() **const**
Returns the number of dimensions of the xindexview.

auto **shape**() **const**
Returns the shape of the xindexview.

### Data

**template** <class... *Args*>
auto **operator()**(size_type *idx*, Args...) **const**
Returns the element at the specified position in the xindexview.

**Parameters**

- `idx`: the position in the view

**template** <**class** It>
auto **element**(It *first*, It)
Returns a reference to the element at the specified position in the xindexview.

**Parameters**

- `first`: iterator starting the sequence of indices The number of indices in the sequence should be equal to or greater 1.

### Broadcasting

**template** <**class** O>
bool **broadcast_shape**(O &*shape*) **const**
Broadcast the shape of the xindexview to the specified parameter.

**Return** a boolean indicating whether the broadcasting is trivial

**Parameters**

- `shape`: the result shape

**template** <**class** O>
bool **is_trivial_broadcast**(**const** O&) **const**
Compares the specified strides with those of the container to see whether the broadcasting is trivial.

**Return** a boolean indicating whether the broadcasting is trivial

**template** <class *ECT*, class *CCT*>
**class** xt::**xfiltration**
Filter of a xexpression for fast scalar assign.

The xfiltration class implements a lazy filtration of a multidimentional *xexpression*, optimized for scalar and computed scalar assignments. Actually, the *xfiltration* class IS NOT an *xexpression* and the scalar and computed scalar assignments are the only method it provides. The filtering condition is not evaluated until the filtration is assigned.

xfiltration is not meant to be used directly, but only with the *filtration* helper function.

**See** *filtration*

**Template Parameters**

- `ECT`: the closure type of the *xexpression* type underlying this filtration
- `CCR`: the closure type of the filtering *xexpression* type

### Extended copy semantic

**template <class** E>
auto **operator=**(**const** E &*e*)
Assigns the scalar `e` to `*this`.

**Return** a reference to \ *this.

**Parameters**

- `e`: the scalar to assign.

### Constructor

**xfiltration**(ECT *e*, CCT *condition*)
Constructs a xfiltration on the given expression `e`, selecting the elements matching the specified `condition`.

**Parameters**

- `e`: the *xexpression* to filter.
- `condition`: the filtering *xexpression* to apply.

### Computed assignement

**template <class** E>
auto **operator+=**(**const** E &*e*)
Adds the scalar `e` to `*this`.

**Return** a reference to `*this`.

**Parameters**

- `e`: the scalar to add.

**template <class** E>
auto **operator-=**(**const** E &*e*)
Subtracts the scalar `e` from `*this`.

**Return** a reference to `*this`.

**Parameters**

- `e`: the scalar to subtract.

**template <class** E>

auto **operator*=**(**const** E &*e*)
Multiplies `*this` with the scalar `e`.

**Return** a reference to `*this`.

**Parameters**

- `e`: the scalar involved in the operation.

**template <class** E>
auto **operator/=**(**const** E &*e*)
Divides `*this` by the scalar `e`.

**Return** a reference to `*this`.

**Parameters**

- `e`: the scalar involved in the operation.

**template <class** E, **class** I>
auto `xt::index_view`(E &&*e*, I &&*indices*)
creates an indexview from a container of indices.

Returns a 1D view with the elements at *indices* selected.

```
xarray<double> a = {{1,5,3}, {4,5,6}};
b = index_view(a, {{0, 0}, {1, 0}, {1, 1}});
std::cout << b << std::endl; // {1, 4, 5}
b += 100;
std::cout << a << std::endl; // {{101, 5, 3}, {104, 105, 6}}
```

**Parameters**

- `e`: the underlying xexpression
- `indices`: the indices to select

**template <class** E, **class** O>
auto `xt::filter`(E &&*e*, O &&*condition*)
creates a view into *e* filtered by *condition*.

Returns a 1D view with the elements selected where *condition* evaluates to *true*. This is equivalent to

```
{index_view(e, where(condition));}
```

The returned view is not optimal if you just want to assign a scalar to the filtered elements. In that case, you should consider using the *filtration* function instead.

```
xarray<double> a = {{1,5,3}, {4,5,6}};
b = filter(a, a >= 5);
std::cout << b << std::endl; // {5, 5, 6}
```

**Parameters**

- `e`: the underlying xexpression
- `condition`: xexpression with shape of *e* which selects indices

**See** *filtration*

**template <class** E, **class** C>

auto `xt::filtration`(E &&*e*, C &&*condition*)
creates a filtration of `e` filtered by *condition*.

Returns a lazy filtration optimized for scalar assignment. Actually, scalar assignment and computed scalar assignments are the only available methods of the filtration, the filtration IS NOT an *xexpression*.

```
xarray<double> a = {{1,5,3}, {4,5,6}};
filtration(a, a >= 5) += 2;
std::cout << a << std::endl; // {{1, 7, 3}, {4, 7, 8}}
```

**Parameters**

- `e`: the *xexpression* to filter
- `condition`: the filtering *xexpression*

### 1.11.14 xfunctorview

Defined in `xtensor/xfunctorview.hpp`
**template** <class *F*, class *CT*>
**class** `xt::xfunctorview`
View of an xexpression .

The xfunctorview class is an expression addressing its elements by applying a functor to the corresponding element of an underlying expression. Unlike e.g. xgenerator, an xfunctorview is an lvalue. It is used e.g. to access real and imaginary parts of complex expressions.

xfunctorview is not meant to be used directly, but through helper functions such as *real* or *imag*.

**See** *real*, *imag*

**Template Parameters**

- `F`: the functor type to be applied to the elements of specified expression.
- `CT`: the closure type of the *xexpression* type underlying this view

Inherits from *xt::xview_semantic< xfunctorview< F, CT > >*

#### Constructors

**xfunctorview**(CT *e*)
Constructs an xfunctorview expression wrappering the specified *xexpression*.

**Parameters**

- `e`: the underlying expression

**template <class** Func, **class** E>
**xfunctorview**(Func &&*func*, E &&*e*)
Constructs an xfunctorview expression wrappering the specified *xexpression*.

**Parameters**

- `func`: the functor to be applied to the elements of the underlying expression.
- `e`: the underlying expression

### Extended copy semantic

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

### Size and shape

auto **size**() **const**
Returns the size of the expression.

auto **dimension**() **const**
Returns the number of dimensions of the expression.

auto **shape**() **const**
Returns the shape of the expression.

*layout_type* **layout**() **const**
Returns the layout_type of the expression.

### Data

auto **operator[]**(**const** xindex &*index*)
Returns a reference to the element at the specified position in the expression.

**Parameters**

- `index`: a sequence of indices specifying the position in the function. Indices must be unsigned integers, the number of indices in the sequence should be equal or greater than the number of dimensions of the container.

auto **operator[]**(**const** xindex &*index*) **const**
Returns a constant reference to the element at the specified position in the expression.

**Parameters**

- `index`: a sequence of indices specifying the position in the function. Indices must be unsigned integers, the number of indices in the sequence should be equal or greater than the number of dimensions of the container.

**template** <class... *Args*>
auto **operator()**(Args... *args*)
Returns a reference to the element at the specified position in the expression.

**Parameters**

- `args`: a list of indices specifying the position in the function. Indices must be unsigned integers, the number of indices should be equal or greater than the number of dimensions of the expression.

**template** <class... *Args*>
auto **at**(Args... *args*)
Returns a reference to the element at the specified position in the expression, after dimension and bounds checking.

**Parameters**

- `args`: a list of indices specifying the position in the function. Indices must be unsigned integers, the number of indices should be equal to the number of dimensions of the expression.

**Exceptions**

- `std::out_of_range`: if the number of argument is greater than the number of dimensions or if indices are out of bounds.

**template <class** IT>
auto **element** (IT *first*, IT *last*)

Returns a reference to the element at the specified position in the expression.

**Parameters**

- `first`: iterator starting the sequence of indices
- `last`: iterator ending the sequence of indices The number of indices in the sequence should be equal to or greater than the number of dimensions of the function.

**template** <class... *Args*>
auto **operator()** (Args... *args*) **const**

Returns a constant reference to the element at the specified position in the expression.

**Parameters**

- `args`: a list of indices specifying the position in the function. Indices must be unsigned integers, the number of indices should be equal or greater than the number of dimensions of the expression.

**template** <class... *Args*>
auto **at** (Args... *args*) **const**

Returns a constant reference to the element at the specified position in the expression, after dimension and bounds checking.

**Parameters**

- `args`: a list of indices specifying the position in the function. Indices must be unsigned integers, the number of indices should be equal to the number of dimensions of the expression.

**Exceptions**

- `std::out_of_range`: if the number of argument is greater than the number of dimensions or if indices are out of bounds.

**template <class** IT>
auto **element** (IT *first*, IT *last*) **const**

Returns a constant reference to the element at the specified position in the expression.

**Parameters**

- `first`: iterator starting the sequence of indices
- `last`: iterator ending the sequence of indices The number of indices in the sequence should be equal to or greater than the number of dimensions of the function.

### Broadcasting

**template <class** S>

bool **broadcast_shape**(S &*shape*) **const**
Broadcast the shape of the function to the specified parameter.

**Return** a boolean indicating whether the broadcasting is trivial

**Parameters**

- `shape`: the result shape

**template** <**class** S>
bool **is_trivial_broadcast**(**const** S &*strides*) **const**
Compares the specified strides with those of the container to see whether the broadcasting is trivial.

**Return** a boolean indicating whether the broadcasting is trivial

### Iterators

**template** <*layout_type* *L* = DL>
auto **begin**()
Returns an iterator to the first element of the expression.

**Template Parameters**

- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

**template** <*layout_type* *L* = DL>
auto **end**()
Returns an iterator to the element following the last element of the expression.

**Template Parameters**

- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

**template** <*layout_type* *L* = DL>
auto **begin**() **const**
Returns a constant iterator to the first element of the expression.

**Template Parameters**

- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

**template** <*layout_type* *L* = DL>
auto **end**() **const**
Returns a constant iterator to the element following the last element of the expression.

**Template Parameters**

- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

**template** <*layout_type* *L* = DL>
auto **cbegin**() **const**
Returns a constant iterator to the first element of the expression.

**Template Parameters**

- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

**template** <*layout_type* *L* = DL>

auto **cend**() **const**

Returns a constant iterator to the element following the last element of the expression.

**Template Parameters**

- L: layout used for the traversal. Default value is DEFAULT_LAYOUT.

### Broadcast iterators

**template** <**class** S, *layout_type L*>
auto **begin**(**const** S &*shape*)

Returns a constant iterator to the first element of the expression.

The iteration is broadcasted to the specified shape.

**Parameters**

- shape: the shape used for broadcasting

**Template Parameters**

- S: type of the shape parameter.
- L: layout used for the traversal. Default value is DEFAULT_LAYOUT.

**template** <**class** S, *layout_type L*>
auto **end**(**const** S &*shape*)

Returns a constant iterator to the element following the last element of the expression.

The iteration is broadcasted to the specified shape.

**Parameters**

- shape: the shape used for broadcasting

**Template Parameters**

- S: type of the shape parameter.
- L: layout used for the traversal. Default value is DEFAULT_LAYOUT.

**template** <**class** S, *layout_type L*>
auto **begin**(**const** S &*shape*) **const**

Returns a constant iterator to the first element of the expression.

The iteration is broadcasted to the specified shape.

**Parameters**

- shape: the shape used for broadcasting

**Template Parameters**

- S: type of the shape parameter.
- L: layout used for the traversal. Default value is DEFAULT_LAYOUT.

**template** <**class** S, *layout_type L*>
auto **end**(**const** S &*shape*) **const**

Returns a constant iterator to the element following the last element of the expression.

The iteration is broadcasted to the specified shape.

**Parameters**

- shape: the shape used for broadcasting

**Template Parameters**

- `S`: type of the `shape` parameter.
- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

**template <class** S, *layout_type L*>
auto **cbegin**(**const** S &*shape*) **const**

Returns a constant iterator to the first element of the expression.

The iteration is broadcasted to the specified shape.

**Parameters**

- `shape`: the shape used for broadcasting

**Template Parameters**

- `S`: type of the `shape` parameter.
- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

**template <class** S, *layout_type L*>
auto **cend**(**const** S &*shape*) **const**

Returns a constant iterator to the element following the last element of the expression.

The iteration is broadcasted to the specified shape.

**Parameters**

- `shape`: the shape used for broadcasting

**Template Parameters**

- `S`: type of the `shape` parameter.
- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

### Reverse iterators

**template** <*layout_type L* = DL>
auto **rbegin**()

Returns an iterator to the first element of the reversed expression.

**Template Parameters**

- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

**template** <*layout_type L* = DL>
auto **rend**()

Returns an iterator to the element following the last element of the reversed expression.

**Template Parameters**

- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

**template** <*layout_type L* = DL>
auto **rbegin**() **const**

Returns a constant iterator to the first element of the reversed expression.

**Template Parameters**

- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

**template** <*layout_type* L = DL>
auto **rend**() **const**

Returns a constant iterator to the element following the last element of the reversed expression.

**Template Parameters**

- L: layout used for the traversal. Default value is DEFAULT_LAYOUT.

**template** <*layout_type* L = DL>
auto **crbegin**() **const**

Returns a constant iterator to the first element of the reversed expression.

**Template Parameters**

- L: layout used for the traversal. Default value is DEFAULT_LAYOUT.

**template** <*layout_type* L = DL>
auto **crend**() **const**

Returns a constant iterator to the element following the last element of the reversed expression.

**Template Parameters**

- L: layout used for the traversal. Default value is DEFAULT_LAYOUT.

### Reverse broadcast iterators

**template** <**class** S, *layout_type* L>
auto **rbegin**(**const** S &*shape*)

Returns an iterator to the first element of the expression.

The iteration is broadcasted to the specified shape.

**Parameters**

- shape: the shape used for broadcasting

**Template Parameters**

- S: type of the shape parameter.
- L: layout used for the traversal. Default value is DEFAULT_LAYOUT.

**template** <**class** S, *layout_type* L>
auto **rend**(**const** S &*shape*)

Returns an iterator to the element following the last element of the reversed expression.

The iteration is broadcasted to the specified shape.

**Parameters**

- shape: the shape used for broadcasting

**Template Parameters**

- S: type of the shape parameter.
- L: layout used for the traversal. Default value is DEFAULT_LAYOUT.

**template** <**class** S, *layout_type* L>
auto **rbegin**(**const** S &*shape*) **const**

Returns a constant iterator to the first element of the reversed expression.

The iteration is broadcasted to the specified shape.

**Parameters**

- `shape`: the shape used for broadcasting

**Template Parameters**

- `S`: type of the `shape` parameter.
- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

**template <class** S, *layout_type L*>
auto **rend**(**const** S&) **const**

Returns a constant iterator to the element following the last element of the reversed expression.

**Parameters**

- `shape`: the shape used for broadcasting

**Template Parameters**

- `S`: type of the `shape` parameter.
- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

**template <class** S, *layout_type L*>
auto **crbegin**(**const** S&) **const**

Returns a constant iterator to the first element of the reversed expression.

The iteration is broadcasted to the specified shape.

**Parameters**

- `shape`: the shape used for broadcasting

**Template Parameters**

- `S`: type of the `shape` parameter.
- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

**template <class** S, *layout_type L*>
auto **crend**(**const** S &*shape*) **const**

Returns a constant iterator to the element following the last element of the reversed expression.

**Parameters**

- `shape`: the shape used for broadcasting

**Template Parameters**

- `S`: type of the `shape` parameter.
- `L`: layout used for the traversal. Default value is `DEFAULT_LAYOUT`.

Defined in `xtensor/xcomplex.hpp`

**template <class** E>
**decltype(auto) xt::real(E && e)**

Returns an *xexpression* representing the real part of the given expression.

The returned expression either hold a const reference to `e` or a copy depending on whether `e` is an lvalue or an rvalue.

**Template Parameters**

- `e`: the *xexpression*

**template <class** E>
**decltype(auto) xt::imag(E && e)**
Returns an *xexpression* representing the imaginary part of the given expression.

The returned expression either hold a const reference to `e` or a copy depending on whether `e` is an lvalue or an rvalue.

**Template Parameters**

- `e`: the *xexpression*

## 1.12 Functions and generators

### 1.12.1 xfunction

Defined in `xtensor/xfunction.hpp`
**template** <class *F*, class *R*, class... *CT*>
**class** `xt::`**xfunction**
Multidimensional function operating on xexpression.

Th xfunction class implements a multidimensional function operating on xexpression.

**Template Parameters**

- `F`: the function type
- `R`: the return type of the function
- `CT`: the closure types for arguments of the function

Inherits from *xt::xexpression< xfunction< F, R, CT... > >*, *xt::xconst_iterable< xfunction< F, R, CT... > >*

#### Constructor

**template <class** Func>
**xfunction**(Func &&*f*, CT... *e*)
Constructs an xfunction applying the specified function to the given arguments.

**Parameters**

- `f`: the function to apply
- `e`: the *xexpression* arguments

#### Size and shape

auto **size**() **const**
Returns the size of the expression.

auto **dimension**() **const**
Returns the number of dimensions of the function.

auto **shape**() **const**
Returns the shape of the xfunction.

*layout_type* **layout**() **const**
Returns the layout_type of the xfunction.

### Data

**template** <class... *Args*>
auto **operator()**(Args... *args*) **const**
Returns a constant reference to the element at the specified position in the function.

**Parameters**

- `args`: a list of indices specifying the position in the function. Indices must be unsigned integers, the number of indices should be equal or greater than the number of dimensions of the function.

**template** <class... *Args*>
auto **at**(Args... *args*) **const**
Returns a constant reference to the element at the specified position in the expression, after dimension and bounds checking.

**Parameters**

- `args`: a list of indices specifying the position in the function. Indices must be unsigned integers, the number of indices should be equal to the number of dimensions of the expression.

**Exceptions**

- `std::out_of_range`: if the number of argument is greater than the number of dimensions or if indices are out of bounds.

**template <class** It>
auto **element**(It *first*, It *last*) **const**
Returns a constant reference to the element at the specified position in the function.

**Parameters**

- `first`: iterator starting the sequence of indices
- `last`: iterator ending the sequence of indices The number of indices in the sequence should be equal to or greater than the number of dimensions of the container.

### Broadcasting

**template <class** S>
bool **broadcast_shape**(S &*shape*) **const**
Broadcast the shape of the function to the specified parameter.

**Return** a boolean indicating whether the broadcasting is trivial

**Parameters**

- `shape`: the result shape

**template <class** S>
bool **is_trivial_broadcast**(**const** S &*strides*) **const**
Compares the specified strides with those of the container to see whether the broadcasting is trivial.

**Return** a boolean indicating whether the broadcasting is trivial

### 1.12.2 xreducer

Defined in `xtensor/xreducer.hpp`
**template** <class *F*, class *CT*, class *X*>
**class** `xt::xreducer`

Reducing function operating over specified axes.

The xreducer class implements an *xexpression* applying a reducing function to an *xexpression* over the specified axes.

The reducer's result_type is deduced from the result type of function `F::reduce_functor_type` when called with elements of the expression

**See** *reduce*

**Template Parameters**

- `F`: a tuple of functors (class xreducer_functors or compatible)
- `CT`: the closure type of the *xexpression* to reduce
- `X`: the list of axes

**Template Parameters**

- `CT.`:

Inherits from *xt::xexpression< xreducer< F, CT, X > >*, *xt::xconst_iterable< xreducer< F, CT, X > >*

#### Constructor

**template** <**class** Func, **class** CTA, **class** AX>
**xreducer**(Func &&*func*, CTA &&*e*, AX &&*axes*)

Constructs an xreducer expression applying the specified function to the given expression over the given axes.

**Parameters**

- `func`: the function to apply
- `e`: the expression to reduce
- `axes`: the axes along which the reduction is performed

#### Size and shape

auto **size**() **const**

Returns the size of the expression.

auto **dimension**() **const**

Returns the number of dimensions of the expression.

auto **shape**() **const**

Returns the shape of the expression.

*layout_type* **layout**() **const**

Returns the shape of the expression.

### Data

auto **operator[]**(**const** xindex &*index*) **const**

Returns a constant reference to the element at the specified position in the reducer.

**Parameters**

- `index`: a sequence of indices specifying the position in the reducer. Indices must be unsigned integers, the number of indices in the sequence should be equal or greater than the number of dimensions of the reducer.

**template** <class... *Args*>
auto **operator()**(Args... *args*) **const**

Returns a constant reference to the element at the specified position in the reducer.

**Parameters**

- `args`: a list of indices specifying the position in the reducer. Indices must be unsigned integers, the number of indices should be equal or greater than the number of dimensions of the reducer.

**template** <class... *Args*>
auto **at**(Args... *args*) **const**

Returns a constant reference to the element at the specified position in the expression, after dimension and bounds checking.

**Parameters**

- `args`: a list of indices specifying the position in the function. Indices must be unsigned integers, the number of indices should be equal to the number of dimensions of the expression.

**Exceptions**

- `std::out_of_range`: if the number of argument is greater than the number of dimensions or if indices are out of bounds.

**template <class** It>
auto **element**(It *first*, It *last*) **const**

Returns a constant reference to the element at the specified position in the reducer.

**Parameters**

- `first`: iterator starting the sequence of indices
- `last`: iterator ending the sequence of indices The number of indices in the sequence should be equal to or greater than the number of dimensions of the reducer.

### Broadcasting

**template <class** S>
bool **broadcast_shape**(S &*shape*) **const**

Broadcast the shape of the reducer to the specified parameter.

**Return** a boolean indicating whether the broadcasting is trivial

**Parameters**

- `shape`: the result shape

**template <class** S>
bool **is_trivial_broadcast**(**const** S &*strides*) **const**
Compares the specified strides with those of the container to see whether the broadcasting is trivial.

**Return** a boolean indicating whether the broadcasting is trivial

**template <class** F, **class** E, **class** X>
auto xt::**reduce**(F &&*f*, E &&*e*, X &&*axes*)
Returns an *xexpression* applying the speficied reducing function to an expresssion over the given axes.

The returned expression either hold a const reference to `e` or a copy depending on whether `e` is an lvalue or an rvalue.

**Parameters**

- `f`: the reducing function to apply.
- `e`: the *xexpression* to reduce.
- `axes`: the list of axes.

### 1.12.3 xgenerator

Defined in `xtensor/xgenerator.hpp`
**template <class** F, **class** R, **class** S>
**class** xt::**xgenerator**
Multidimensional function operating on indices.

The xgenerator class implements a multidimensional function, generating a value from the supplied indices.

**Template Parameters**

- `F`: the function type
- `R`: the return type of the function
- `S`: the shape type of the generator

Inherits from *xt::xexpression< xgenerator< F, R, S > >*, *xt::xconst_iterable< xgenerator< F, R, S > >*

#### Constructor

**template <class** Func>
**xgenerator**(Func &&*f*, **const** S &*shape*)
Constructs an xgenerator applying the specified function over the given shape.

**Parameters**

- `f`: the function to apply
- `shape`: the shape of the xgenerator

#### Size and shape

auto **size**() **const**
Returns the size of the expression.

auto **dimension**() **const**
Returns the number of dimensions of the function.

auto **shape**() **const**
Returns the shape of the xgenerator.

### Data

**template** <class... *Args*>
auto **operator()**(Args... *args*) **const**
Returns the evaluated element at the specified position in the function.

**Parameters**

- `args`: a list of indices specifying the position in the function. Indices must be unsigned integers, the number of indices should be equal or greater than the number of dimensions of the function.

**template** <class... *Args*>
auto **at**(Args... *args*) **const**
Returns a constant reference to the element at the specified position in the expression, after dimension and bounds checking.

**Parameters**

- `args`: a list of indices specifying the position in the function. Indices must be unsigned integers, the number of indices should be equal to the number of dimensions of the expression.

**Exceptions**

- `std::out_of_range`: if the number of argument is greater than the number of dimensions or if indices are out of bounds.

**template** <**class** It>
auto **element**(It *first*, It *last*) **const**
Returns a constant reference to the element at the specified position in the function.

**Parameters**

- `first`: iterator starting the sequence of indices
- `last`: iterator ending the sequence of indices The number of indices in the sequence should be equal to or greater than the number of dimensions of the container.

### Broadcasting

**template** <**class** O>
bool **broadcast_shape**(O &*shape*) **const**
Broadcast the shape of the function to the specified parameter.

**Return** a boolean indicating whether the broadcasting is trivial

**Parameters**

- `shape`: the result shape

**template** <**class** O>
bool **is_trivial_broadcast**(**const** O&) **const**
Compares the specified strides with those of the container to see whether the broadcasting is trivial.

**Return** a boolean indicating whether the broadcasting is trivial

### 1.12.4 xbuilder

Defined in `xtensor/xbuilder.hpp`

**template <class** T, **class** S>
auto `xt::ones`(S *shape*)

Returns an *xexpression* containing ones of the specified shape.

**Template Parameters**

- `shape`: the shape of the returned expression.

**template <class** T, **class** I, std::size_t *L*>
auto `xt::ones`(**const** I (&*shape*)[L])
**template <class** T, **class** S>
auto `xt::zeros`(S *shape*)

Returns an *xexpression* containing zeros of the specified shape.

**Template Parameters**

- `shape`: the shape of the returned expression.

**template <class** T, **class** I, std::size_t *L*>
auto `xt::zeros`(**const** I (&*shape*)[L])
**template <class** T = bool>
auto `xt::eye`(**const** std::vector<std::size_t> &*shape*, int *k* = 0)

Generates an array with ones on the diagonal.

**Return** xgenerator that generates the values on access

**Parameters**

- `shape`: shape of the resulting expression
- `k`: index of the diagonal. 0 (default) refers to the main diagonal, a positive value refers to an upper diagonal, and a negative value to a lower diagonal.

**Template Parameters**

- `T`: value_type of xexpression

**template <class** T = bool>
auto `xt::eye`(std::size_t *n*, int *k* = 0)

Generates a (n x n) array with ones on the diagonal.

**Return** xgenerator that generates the values on access

**Parameters**

- `n`: length of the diagonal.
- `k`: index of the diagonal. 0 (default) refers to the main diagonal, a positive value refers to an upper diagonal, and a negative value to a lower diagonal.

**Template Parameters**

- `T`: value_type of xexpression

**template <class** T>

auto xt::**arange**(T *start*, T *stop*, T *step* = 1)

Generates numbers evenly spaced within given half-open interval [start, stop).

**Return** xgenerator that generates the values on access

**Parameters**

- start: start of the interval
- stop: stop of the interval
- step: stepsize

**Template Parameters**

- T: value_type of xexpression

**template <class** T>
auto xt::**arange**(T *stop*)

Generate numbers evenly spaced within given half-open interval [0, stop) with a step size of 1.

**Return** xgenerator that generates the values on access

**Parameters**

- stop: stop of the interval

**Template Parameters**

- T: value_type of xexpression

**template <class** T>
auto xt::**linspace**(T *start*, T *stop*, std::size_t *num_samples* = 50, bool *endpoint* = true)

Generates *num_samples* evenly spaced numbers over given interval.

**Return** xgenerator that generates the values on access

**Parameters**

- start: start of interval
- stop: stop of interval
- num_samples: number of samples (defaults to 50)
- endpoint: if true, include endpoint (defaults to true)

**Template Parameters**

- T: value_type of xexpression

**template <class** T>
auto xt::**logspace**(T *start*, T *stop*, std::size_t *num_samples*, T *base* = 10, bool *endpoint* = true)

Generates *num_samples* numbers evenly spaced on a log scale over given interval.

**Return** xgenerator that generates the values on access

**Parameters**

- start: start of interval (pow(base, start) is the first value).
- stop: stop of interval (pow(base, stop) is the final value, except if endpoint = false)
- num_samples: number of samples (defaults to 50)
- base: the base of the log space.

- `endpoint`: if true, include endpoint (defaults to true)

**Template Parameters**

- `T`: value_type of xexpression

**template <class** E>
auto `xt::diag`(E &&*arr*, int *k* = 0)

xexpression with values of arr on the diagonal, zeroes otherwise

```
xt::xarray<double> a = {1, 5, 9};
auto b = xt::diag(a); // => {{1, 0, 0},
                      //     {0, 5, 0},
                      //     {0, 0, 9}}
```

**Return** xexpression function with shape n x n and arr on the diagonal

**Parameters**

- `arr`: the 1D input array of length n
- `k`: the offset of the considered diagonal

**template <class** E>
auto `xt::diagonal`(E &&*arr*, int *offset* = 0, std::size_t *axis_1* = 0, std::size_t *axis_2* = 1)

Returns the elements on the diagonal of arr If arr has more than two dimensions, then the axes specified by axis_1 and axis_2 are used to determine the 2-D sub-array whose diagonal is returned.

The shape of the resulting array can be determined by removing axis1 and axis2 and appending an index to the right equal to the size of the resulting diagonals.

```
xt::xarray<double> a = {{1, 2, 3},
                        {4, 5, 6}
                        {7, 8, 9}};
auto b = xt::diagonal(a); // => {1, 5, 9}
```

**Return** xexpression with values of the diagonal

**Parameters**

- `arr`: the input array
- `offset`: offset of the diagonal from the main diagonal. Can be positive or negative.
- `axis_1`: Axis to be used as the first axis of the 2-D sub-arrays from which the diagonals should be taken.
- `axis_2`: Axis to be used as the second axis of the 2-D sub-arrays from which the diagonals should be taken.

**template <class** E>
auto `xt::tril`(E &&*arr*, int *k* = 0)

Extract lower triangular matrix from xexpression.

The parameter k selects the offset of the diagonal.

**Return** xexpression containing lower triangle from arr, 0 otherwise

**Parameters**

- `arr`: the input array

- `k`: the diagonal above which to zero elements. 0 (default) selects the main diagonal, k < 0 is below the main diagonal, k > 0 above.

**template <class** E>
auto `xt::triu`(E &&*arr*, int *k* = 0)

Extract upper triangular matrix from xexpression.

The parameter k selects the offset of the diagonal.

**Return** xexpression containing lower triangle from arr, 0 otherwise

**Parameters**

- `arr`: the input array
- `k`: the diagonal below which to zero elements. 0 (default) selects the main diagonal, k < 0 is below the main diagonal, k > 0 above.

**template <class** E>
auto `xt::flip`(E &&*arr*, std::size_t *axis*)

Reverse the order of elements in an xexpression along the given axis.

Note: A NumPy/Matlab style `flipud(arr)` is equivalent to `xt::flip(arr, 0)`, `fliplr(arr)` to `xt::flip(arr, 1)`.

**Return** xexpression evaluating to reversed array

**Parameters**

- `arr`: the input xexpression
- `axis`: the axis along which elements should be reversed

### 1.12.5 xrandom

Defined in `xtensor/xrandom.hpp`

default_engine_type &`xt::random::get_default_random_engine`()

Returns a reference to the default random number engine.

void `xt::random::seed`(seed_type *seed*)

Seeds the default random number generator with `seed`.

**Parameters**

- `seed`: The seed

**template <class** T, **class** S, **class** E = random::default_engine_type>
auto `xt::random::rand`(**const** S &*shape*, T *lower* = 0, T *upper* = 1, E &*engine* = random::get_default_random_engine())

xexpression with specified `shape` containing uniformly distributed random numbers in the interval from `lower` to `upper`, excluding upper.

Numbers are drawn from `std::uniform_real_distribution`.

**Parameters**

- `shape`: shape of resulting xexpression
- `lower`: lower bound
- `upper`: upper bound

- `engine`: random number engine

**Template Parameters**

- `T`: number type to use

**template <class** T, **class** S, **class** E = random::default_engine_type>
auto `xt::random::`**randint**(**const** S &*shape*, T *lower* = 0, T *upper* = std::numeric_limits< T >::max(), E &*engine* = random::get_default_random_engine())

xexpression with specified `shape` containing uniformly distributed random integers in the interval from `lower` to `upper`, excluding upper.

Numbers are drawn from `std::uniform_int_distribution`.

**Parameters**

- `shape`: shape of resulting xexpression
- `lower`: lower bound
- `upper`: upper bound
- `engine`: random number engine

**Template Parameters**

- `T`: number type to use

**template <class** T, **class** S, **class** E = random::default_engine_type>
auto `xt::random::`**randn**(**const** S &*shape*, T *mean* = 0, T *std_dev* = 1, E &*engine* = random::get_default_random_engine())

xexpression with specified `shape` containing numbers sampled from the Normal (Gaussian) random number distribution with mean `mean` and standard deviation `std_dev`.

Numbers are drawn from `std::normal_distribution`.

**Parameters**

- `shape`: shape of resulting xexpression
- `mean`: mean of normal distribution
- `std_dev`: standard deviation of normal distribution
- `engine`: random number engine

**Template Parameters**

- `T`: number type to use

## 1.13 Mathematical functions

### 1.13.1 Operators and related functions

Defined in `xtensor/xmath.hpp`
**template <class** E>
auto `xt::`**operator+**(E &&*e*)

Identity.

Returns an *xfunction* for the element-wise identity of *e*.

**Return** an *xfunction*

**Parameters**

- e: an *xexpression*

**template <class** E>
auto xt::**operator-**(E &&*e*)

Opposite.

Returns an *xfunction* for the element-wise opposite of *e*.

**Return** an *xfunction*

**Parameters**

- e: an *xexpression*

**template <class** E1, **class** E2>
auto xt::**operator+**(E1 &&*e1*, E2 &&*e2*)

Addition.

Returns an *xfunction* for the element-wise addition of *e1* and *e2*.

**Return** an *xfunction*

**Parameters**

- e1: an *xexpression* or a scalar
- e2: an *xexpression* or a scalar

**template <class** E1, **class** E2>
auto xt::**operator-**(E1 &&*e1*, E2 &&*e2*)

Substraction.

Returns an *xfunction* for the element-wise substraction of *e2* to *e1*.

**Return** an *xfunction*

**Parameters**

- e1: an *xexpression* or a scalar
- e2: an *xexpression* or a scalar

**template <class** E1, **class** E2>
auto xt::**operator***(E1 &&*e1*, E2 &&*e2*)

Multiplication.

Returns an *xfunction* for the element-wise multiplication of *e1* by *e2*.

**Return** an *xfunction*

**Parameters**

- e1: an *xexpression* or a scalar
- e2: an *xexpression* or a scalar

**template <class** E1, **class** E2>
auto xt::**operator/**(E1 &&*e1*, E2 &&*e2*)

Division.

Returns an *xfunction* for the element-wise division of *e1* by *e2*.

**Return** an *xfunction*

**Parameters**

- e1: an *xexpression* or a scalar
- e2: an *xexpression* or a scalar

**template <class** E1, **class** E2>
auto xt::**operator||**(E1 &&*e1*, E2 &&*e2*)

Or.

Returns an *xfunction* for the element-wise or of *e1* and *e2*.

**Return** an *xfunction*

**Parameters**

- e1: an *xexpression* or a scalar
- e2: an *xexpression* or a scalar

**template <class** E1, **class** E2>
auto xt::**operator&&**(E1 &&*e1*, E2 &&*e2*)

And.

Returns an *xfunction* for the element-wise and of *e1* and *e2*.

**Return** an *xfunction*

**Parameters**

- e1: an *xexpression* or a scalar
- e2: an *xexpression* or a scalar

**template <class** E>
auto xt::**operator!**(E &&*e*)

Not.

Returns an *xfunction* for the element-wise not of *e*.

**Return** an *xfunction*

**Parameters**

- e: an *xexpression*

**template <class** E1, **class** E2, **class** E3>
auto xt::**where**(E1 &&*e1*, E2 &&*e2*, E3 &&*e3*)

Ternary selection.

Returns an *xfunction* for the element-wise ternary selection (i.e. operator ? :) of *e1*, *e2* and *e3*.

**Return** an *xfunction*

**Parameters**

- e1: a boolean *xexpression*
- e2: an *xexpression* or a scalar
- e3: an *xexpression* or a scalar

**template <class** T>
auto xt::**nonzero**(**const** T &*arr*)

return vector of indices where T is not zero

**Return** vector of *index_types* where arr is not equal to zero

**Parameters**

- arr: input array

**template <class** T>
auto xt::**where**(**const** T &*condition*)

return vector of indices where condition is true (equivalent to *nonzero(condition)*)

**Return** vector of *index_types* where condition is not equal to zero

**Parameters**

- `condition`: input array

**template <class** E>
bool `xt`**::any**(E &&*e*)

Any.

Returns true if any of the values of *e* is truthy, false otherwise.

**Return** a boolean

**Parameters**

- `e`: an *xexpression*

**template <class** E>
bool `xt`**::all**(E &&*e*)

Any.

Returns true if all of the values of *e* are truthy, false otherwise.

**Return** a boolean

**Parameters**

- `e`: an *xexpression*

**template <class** E1, **class** E2>
auto `xt`**::operator<**(E1 &&*e1*, E2 &&*e2*)

Lesser than.

Returns an *xfunction* for the element-wise lesser than comparison of *e1* and *e2*.

**Return** an *xfunction*

**Parameters**

- `e1`: an *xexpression* or a scalar
- `e2`: an *xexpression* or a scalar

**template <class** E1, **class** E2>
auto `xt`**::operator<=**(E1 &&*e1*, E2 &&*e2*)

Lesser or equal.

Returns an *xfunction* for the element-wise lesser or equal comparison of *e1* and *e2*.

**Return** an *xfunction*

**Parameters**

- `e1`: an *xexpression* or a scalar
- `e2`: an *xexpression* or a scalar

**template <class** E1, **class** E2>
auto `xt`**::operator>**(E1 &&*e1*, E2 &&*e2*)

Greater than.

Returns an *xfunction* for the element-wise greater than comparison of *e1* and *e2*.

**Return** an *xfunction*

**Parameters**

- `e1`: an *xexpression* or a scalar
- `e2`: an *xexpression* or a scalar

**template <class** E1, **class** E2>
auto xt**::operator>=**(E1 &&*e1*, E2 &&*e2*)

Greater or equal.

Returns an *xfunction* for the element-wise greater or equal comparison of *e1* and *e2*.

**Return** an *xfunction*

**Parameters**

- e1: an *xexpression* or a scalar
- e2: an *xexpression* or a scalar

**template <class** E1, **class** E2>
bool xt**::operator==**(**const** *xexpression*<E1> &*e1*, **const** *xexpression*<E2> &*e2*)

Equality.

Returns true if *e1* and *e2* have the same shape and hold the same values. Unlike other comparison operators, this does not return an *xfunction*.

**Return** a boolean

**Parameters**

- e1: an *xexpression* or a scalar
- e2: an *xexpression* or a scalar

**template <class** E1, **class** E2>
bool xt**::operator!=**(**const** *xexpression*<E1> &*e1*, **const** *xexpression*<E2> &*e2*)

Inequality.

Returns true if *e1* and *e2* have different shapes or hold the different values. Unlike other comparison operators, this does not return an *xfunction*.

**Return** a boolean

**Parameters**

- e1: an *xexpression* or a scalar
- e2: an *xexpression* or a scalar

**template <class** E1, **class** E2>
auto xt**::equal**(E1 &&*e1*, E2 &&*e2*)

Element-wise equality.

Returns an *xfunction* for the element-wise equality of *e1* and *e2*.

**Return** an *xfunction*

**Parameters**

- e1: an *xexpression* or a scalar
- e2: an *xexpression* or a scalar

**template <class** E1, **class** E2>
auto xt**::not_equal**(E1 &&*e1*, E2 &&*e2*)

Element-wise inequality.

Returns an *xfunction* for the element-wise inequality of *e1* and *e2*.

**Return** an *xfunction*

**Parameters**

- e1: an *xexpression* or a scalar

- `e2`: an *xexpression* or a scalar

| | |
|---|---|
| *operator+* | identity |
| *operator-* | opposite |
| *operator+* | addition |
| *operator-* | substraction |
| *operator** | multiplication |
| *operator/* | division |
| *operator\|\|* | logical or |
| *operator&&* | logical and |
| *operator!* | logical not |
| *where* | ternary selection |
| *nonzero* | indices selection |
| *where* | indices selection |
| *any* | return true if any value is truthy |
| *all* | return true if all the values are truthy |
| *operator<* | element-wise lesser than |
| *operator<=* | element-wise less or equal |
| *operator>* | element-wise greater than |
| *operator>=* | element-wise greater or equal |
| *operator==* | expression equality |
| *operator!=* | expression inequality |
| *equal* | element-wise equality |
| *not_equal* | element-wise inequality |

### 1.13.2 Basic functions

**xtensor** provides the following basic functions for xexpressions and scalars:

Defined in `xtensor/xmath.hpp`

**template <class** E>
auto `xt::`**abs**(E &&*e*)

Absolute value function.

Returns an *xfunction* for the element-wise absolute value of *e*.

**Return** an *xfunction*

**Parameters**

- `e`: an *xexpression*

**template <class** E>
auto `xt::`**fabs**(E &&*e*)

Absolute value function.

Returns an *xfunction* for the element-wise absolute value of *e*.

**Return** an *xfunction*

**Parameters**

- `e`: an *xexpression*

**template <class** E1, **class** E2>
auto `xt::`**fmod**(E1 &&*e1*, E2 &&*e2*)

Remainder of the floating point division operation.

Returns an *xfunction* for the element-wise remainder of the floating point division operation *e1 / e2*.

**Return** an *xfunction*

**Note** e1 and e2 can't be both scalars.

**Parameters**

- `e1`: an *xexpression* or a scalar
- `e2`: an *xexpression* or a scalar

**template <class** E1, **class** E2>
auto `xt::remainder`(E1 &&*e1*, E2 &&*e2*)

Signed remainder of the division operation.

Returns an *xfunction* for the element-wise signed remainder of the floating point division operation *e1 / e2*.

**Return** an *xfunction*

**Note** e1 and e2 can't be both scalars.

**Parameters**

- `e1`: an *xexpression* or a scalar
- `e2`: an *xexpression* or a scalar

**template <class** E1, **class** E2, **class** E3>
auto `xt::fma`(E1 &&*e1*, E2 &&*e2*, E3 &&*e3*)

Fused multiply-add operation.

Returns an *xfunction* for *e1 * e2 + e3* as if to infinite precision and rounded only once to fit the result type.

**Return** an *xfunction*

**Note** e1, e2 and e3 can't be scalars every three.

**Parameters**

- `e1`: an *xfunction* or a scalar
- `e2`: an *xfunction* or a scalar
- `e3`: an *xfunction* or a scalar

**template <class** E1, **class** E2>
auto `xt::maximum`(E1 &&*e1*, E2 &&*e2*)

Elementwise maximum.

Returns an *xfunction* for the element-wise maximum between e1 and e2.

**Return** an *xfunction*

**Parameters**

- `e1`: an *xexpression*
- `e2`: an *xexpression*

**template <class** E1, **class** E2>
auto `xt::minimum`(E1 &&*e1*, E2 &&*e2*)

Elementwise minimum.

Returns an *xfunction* for the element-wise minimum between e1 and e2.

**Return** an *xfunction*

**Parameters**

- `e1`: an *xexpression*
- `e2`: an *xexpression*

**template <class** E1, **class** E2>
auto xt::**fmax**(E1 &&*e1*, E2 &&*e2*)

Maximum function.

Returns an *xfunction* for the element-wise maximum of *e1* and *e2*.

**Return** an *xfunction*

**Note** e1 and e2 can't be both scalars.

**Parameters**

- e1: an *xexpression* or a scalar
- e2: an *xexpression* or a scalar

**template <class** E1, **class** E2>
auto xt::**fmin**(E1 &&*e1*, E2 &&*e2*)

Minimum function.

Returns an *xfunction* for the element-wise minimum of *e1* and *e2*.

**Return** an *xfunction*

**Note** e1 and e2 can't be both scalars.

**Parameters**

- e1: an *xexpression* or a scalar
- e2: an *xexpression* or a scalar

**template <class** E1, **class** E2>
auto xt::**fdim**(E1 &&*e1*, E2 &&*e2*)

Positive difference function.

Returns an *xfunction* for the element-wise positive difference of *e1* and *e2*.

**Return** an *xfunction*

**Note** e1 and e2 can't be both scalars.

**Parameters**

- e1: an *xexpression* or a scalar
- e2: an *xexpression* or a scalar

**template <class** E1, **class** E2, **class** E3>
auto xt::**clip**(E1 &&*e1*, E2 &&*lo*, E3 &&*hi*)

Clip values between hi and lo.

Returns an *xfunction* for the element-wise clipped values between lo and hi

**Return** a *xfunction*

**Parameters**

- e1: an *xexpression* or a scalar
- lo: a scalar
- hi: a scalar

**template <class** E>
auto xt::**sign**(E &&*e*)

Returns an element-wise indication of the sign of a number.

If the number is positive, returns +1. If negative, -1. If the number is zero, returns 0.

**Return** an *xfunction*

**Parameters**

- `e`: an *xexpression*

| | |
|---|---|
| *abs* | absolute value |
| *fabs* | absolute value |
| *fmod* | remainder of the floating point division operation |
| *remainder* | signed remainder of the division operation |
| *fma* | fused multiply-add operation |
| *minimum* | element-wise minimum |
| *maximum* | element-wise maximum |
| *fmin* | element-wise minimum for floating point values |
| *fmax* | element-wise maximum for floating point values |
| *fdim* | element-wise positive difference |
| *clip* | element-wise clipping operation |
| *sign* | element-wise indication of the sign |

### 1.13.3 Exponential functions

**xtensor** provides the following exponential functions for xexpressions:

Defined in `xtensor/xmath.hpp`

**template <class** E>
auto `xt::`**exp**(E &&*e*)

Natural exponential function.

Returns an *xfunction* for the element-wise natural exponential of *e*.

**Return** an *xfunction*

**Parameters**

- `e`: an *xexpression*

**template <class** E>
auto `xt::`**exp2**(E &&*e*)

Base 2 exponential function.

Returns an *xfunction* for the element-wise base 2 exponential of *e*.

**Return** an *xfunction*

**Parameters**

- `e`: an *xexpression*

**template <class** E>
auto `xt::`**expm1**(E &&*e*)

Natural exponential minus one function.

Returns an *xfunction* for the element-wise natural exponential of *e*, minus 1.

**Return** an *xfunction*

**Parameters**

- `e`: an *xexpression*

**template <class** E>

auto xt::**log**(E &&*e*)
Natural logarithm function.

Returns an *xfunction* for the element-wise natural logarithm of *e*.

**Return** an *xfunction*

**Parameters**

- e: an *xexpression*

**template <class** E>
auto xt::**log2**(E &&*e*)
Base 2 logarithm function.

Returns an *xfunction* for the element-wise base 2 logarithm of *e*.

**Return** an *xfunction*

**Parameters**

- e: an *xexpression*

**template <class** E>
auto xt::**log10**(E &&*e*)
Base 10 logarithm function.

Returns an *xfunction* for the element-wise base 10 logarithm of *e*.

**Return** an *xfunction*

**Parameters**

- e: an *xexpression*

**template <class** E>
auto xt::**log1p**(E &&*e*)
Natural logarithm of one plus function.

Returns an *xfunction* for the element-wise natural logarithm of *e*, plus 1.

**Return** an *xfunction*

**Parameters**

- e: an *xexpression*

| | |
|---|---|
| *exp* | natural exponential function |
| *exp2* | base 2 exponential function |
| *expm1* | natural exponential function, minus one |
| *log* | natural logarithm function |
| *log2* | base 2 logarithm function |
| *log10* | base 10 logarithm function |
| *log1p* | natural logarithm of one plus function |

### 1.13.4 Power functions

**xtensor** provides the following power functions for xexpressions and scalars:

Defined in xtensor/xmath.hpp
**template <class** E1, **class** E2>
auto xt::**pow**(E1 &&*e1*, E2 &&*e2*)
Power function.

Returns an *xfunction* for the element-wise value of of *e1* raised to the power *e2*.

**Return** an *xfunction*

**Note** e1 and e2 can't be both scalars.

**Parameters**

- `e1`: an *xexpression* or a scalar
- `e2`: an *xexpression* or a scalar

**template <class** E>
auto `xt::sqrt`(E &&*e*)

Square root function.

Returns an *xfunction* for the element-wise square root of *e*.

**Return** an *xfunction*

**Parameters**

- `e`: an *xexpression*

**template <class** E>
auto `xt::cbrt`(E &&*e*)

Cubic root function.

Returns an *xfunction* for the element-wise cubic root of *e*.

**Return** an *xfunction*

**Parameters**

- `e`: an *xexpression*

**template <class** E1, **class** E2>
auto `xt::hypot`(E1 &&*e1*, E2 &&*e2*)

Hypotenuse function.

Returns an *xfunction* for the element-wise square root of the sum of the square of *e1* and *e2*, avoiding overflow and underflow at intermediate stages of computation.

**Return** an *xfunction*

**Note** e1 and e2 can't be both scalars.

**Parameters**

- `e1`: an *xexpression* or a scalar
- `e2`: an *xexpression* or a scalar

| | |
|---|---|
| *pow* | power function |
| *sqrt* | square root function |
| *cbrt* | cubic root function |
| *hypot* | hypotenuse function |

### 1.13.5 Trigonometric functions

**xtensor** provides the following trigonometric functions for xexpressions and scalars:

Defined in `xtensor/xmath.hpp`
**template <class** E>

auto xt::**sin**(E &&*e*)
Sine function.

Returns an *xfunction* for the element-wise sine of *e* (measured in radians).

**Return** an *xfunction*

**Parameters**

- e: an *xexpression*

**template <class** E>
auto xt::**cos**(E &&*e*)
Cosine function.

Returns an *xfunction* for the element-wise cosine of *e* (measured in radians).

**Return** an *xfunction*

**Parameters**

- e: an *xexpression*

**template <class** E>
auto xt::**tan**(E &&*e*)
Tangent function.

Returns an *xfunction* for the element-wise tangent of *e* (measured in radians).

**Return** an *xfunction*

**Parameters**

- e: an *xexpression*

**template <class** E>
auto xt::**asin**(E &&*e*)
Arcsine function.

Returns an *xfunction* for the element-wise arcsine of *e*.

**Return** an *xfunction*

**Parameters**

- e: an *xexpression*

**template <class** E>
auto xt::**acos**(E &&*e*)
Arccosine function.

Returns an *xfunction* for the element-wise arccosine of *e*.

**Return** an *xfunction*

**Parameters**

- e: an *xexpression*

**template <class** E>
auto xt::**atan**(E &&*e*)
Arctangent function.

Returns an *xfunction* for the element-wise arctangent of *e*.

**Return** an *xfunction*

**Parameters**

- e: an *xexpression*

**template <class** E1, **class** E2>

auto xt::**atan2**(E1 &&*e1*, E2 &&*e2*)

Artangent function, using signs to determine quadrants.

Returns an *xfunction* for the element-wise arctangent of *e1 / e2*, using the signs of arguments to determine the correct quadrant.

**Return** an *xfunction*

**Note** e1 and e2 can't be both scalars.

**Parameters**

- e1: an *xexpression* or a scalar
- e2: an *xexpression* or a scalar

| | |
|---|---|
| *sin* | sine function |
| *cos* | cosine function |
| *tan* | tangent function |
| *asin* | arc sine function |
| *acos* | arc cosine function |
| *atan* | arc tangent function |
| *atan2* | arc tangent function, determining quadrants |

### 1.13.6 Hyperbolic functions

**xtensor** provides the following hyperbolic functions for xexpressions:

Defined in xtensor/xmath.hpp

**template <class** E>
auto xt::**sinh**(E &&*e*)

Hyperbolic sine function.

Returns an *xfunction* for the element-wise hyperbolic sine of *e*.

**Return** an *xfunction*

**Parameters**

- e: an *xexpression*

**template <class** E>
auto xt::**cosh**(E &&*e*)

Hyperbolic cosine function.

Returns an *xfunction* for the element-wise hyperbolic cosine of *e*.

**Return** an *xfunction*

**Parameters**

- e: an *xexpression*

**template <class** E>
auto xt::**tanh**(E &&*e*)

Hyperbolic tangent function.

Returns an *xfunction* for the element-wise hyperbolic tangent of *e*.

**Return** an *xfunction*

**Parameters**

- e: an *xexpression*

**template <class** E>
auto xt::**asinh**(E &&*e*)
Inverse hyperbolic sine function.

Returns an *xfunction* for the element-wise inverse hyperbolic sine of *e*.

**Return** an *xfunction*

**Parameters**

- e: an *xexpression*

**template <class** E>
auto xt::**acosh**(E &&*e*)
Inverse hyperbolic cosine function.

Returns an *xfunction* for the element-wise inverse hyperbolic cosine of *e*.

**Return** an *xfunction*

**Parameters**

- e: an *xexpression*

**template <class** E>
auto xt::**atanh**(E &&*e*)
Inverse hyperbolic tangent function.

Returns an *xfunction* for the element-wise inverse hyperbolic tangent of *e*.

**Return** an *xfunction*

**Parameters**

- e: an *xexpression*

| | |
|---|---|
| *sinh* | hyperbolic sine function |
| *cosh* | hyperbolic cosine function |
| *tanh* | hyperbolic tangent function |
| *asinh* | inverse hyperbolic sine function |
| *acosh* | inverse hyperbolic cosine function |
| *atanh* | inverse hyperbolic tangent function |

### 1.13.7 Error and gamma functions

**xtensor** provides the following error and gamma functions for xexpressions:

Defined in xtensor/xmath.hpp
**template <class** E>
auto xt::**erf**(E &&*e*)
Error function.

Returns an *xfunction* for the element-wise error function of *e*.

**Return** an *xfunction*

**Parameters**

- e: an *xexpression*

**template <class** E>
auto xt::**erfc**(E &&*e*)
Complementary error function.

Returns an *xfunction* for the element-wise complementary error function of *e*, whithout loss of precision for large argument.

**Return** an *xfunction*

**Parameters**

- e: an *xexpression*

**template <class** E>
auto xt::**tgamma**(E &&*e*)

Gamma function.

Returns an *xfunction* for the element-wise gamma function of *e*.

**Return** an *xfunction*

**Parameters**

- e: an *xexpression*

**template <class** E>
auto xt::**lgamma**(E &&*e*)

Natural logarithm of the gamma function.

Returns an *xfunction* for the element-wise logarithm of the asbolute value fo the gamma function of *e*.

**Return** an *xfunction*

**Parameters**

- e: an *xexpression*

| | |
|---|---|
| *erf* | error function |
| *erfc* | complementary error function |
| *tgamma* | gamma function |
| *lgamma* | natural logarithm of the gamma function |

### 1.13.8 Nearest integer floating point operations

**xtensor** provides the following rounding operations for xexpressions:

Defined in xtensor/xmath.hpp
**template <class** E>
auto xt::**ceil**(E &&*e*)

ceil function.

Returns an *xfunction* for the element-wise smallest integer value not less than *e*.

**Return** an *xfunction*

**Parameters**

- e: an *xexpression*

**template <class** E>
auto xt::**floor**(E &&*e*)

floor function.

Returns an *xfunction* for the element-wise smallest integer value not greater than *e*.

**Return** an *xfunction*

**Parameters**

- e: an *xexpression*

**template <class** E>
auto xt::**trunc**(E &&*e*)

trunc function.

Returns an *xfunction* for the element-wise nearest integer not greater in magnitude than *e*.

**Return** an *xfunction*

**Parameters**

- e: an *xexpression*

**template <class** E>
auto xt::**round**(E &&*e*)

round function.

Returns an *xfunction* for the element-wise nearest integer value to *e*, rounding halfway cases away from zero, regardless of the current rounding mode.

**Return** an *xfunction*

**Parameters**

- e: an *xexpression*

**template <class** E>
auto xt::**nearbyint**(E &&*e*)

nearbyint function.

Returns an *xfunction* for the element-wise rounding of *e* to integer values in floating point format, using the current rounding mode. nearbyint never raises FE_INEXACT error.

**Return** an *xfunction*

**Parameters**

- e: an *xexpression*

**template <class** E>
auto xt::**rint**(E &&*e*)

rint function.

Returns an *xfunction* for the element-wise rounding of *e* to integer values in floating point format, using the current rounding mode. Contrary to nearbyint, rint may raise FE_INEXACT error.

**Return** an *xfunction*

**Parameters**

- e: an *xexpression*

| | |
|---|---|
| *ceil* | nearest integers not less |
| *floor* | nearest integers not greater |
| *trunc* | nearest integers not greater in magnitude |
| *round* | nearest integers, rounding away from zero |
| *nearbyint* | nearest integers using current rounding mode |
| *rint* | nearest integers using current rounding mode |

### 1.13.9 Classification functions

**xtensor** provides the following classification functions for xexpressions and scalars:

Defined in `xtensor/xmath.hpp`

**template <class** E>
auto xt::**isfinite**(E &&*e*)
finite value check

Returns an *xfunction* for the element-wise finite value check tangent of *e*.

**Return** an *xfunction*

**Parameters**

- e: an *xexpression*

**template <class** E>
auto xt::**isinf**(E &&*e*)
infinity check

Returns an *xfunction* for the element-wise infinity check tangent of *e*.

**Return** an *xfunction*

**Parameters**

- e: an *xexpression*

**template <class** E>
auto xt::**isnan**(E &&*e*)
NaN check.

Returns an *xfunction* for the element-wise NaN check tangent of *e*.

**Return** an *xfunction*

**Parameters**

- e: an *xexpression*

**template <class** E1, **class** E2>
auto xt::**isclose**(E1 &&*e1*, E2 &&*e2*, double *rtol* = 1e-05, double *atol* = 1e-08, bool *equal_nan* = false)
Element-wise closeness detection.

Returns an *xfunction* that evaluates to true if the elements in e1 and e2 are close to each other according to parameters atol and rtol. The equation is: std::abs(a - b) <= (m_atol + m_rtol * std::abs(b)).

**Return** an *xfunction*

**Parameters**

- e1: input array to compare
- e2: input array to compare
- rtol: the relative tolerance parameter (default 1e-05)
- atol: the absolute tolerance parameter (default 1e-08)
- equal_nan: if true, isclose returns true if both elements of e1 and e2 are NaN

**template <class** E1, **class** E2>
auto xt::**allclose**(E1 &&*e1*, E2 &&*e2*, double *rtol* = 1e-05, double *atol* = 1e-08)
Check if all elements in *e1* are close to the corresponding elements in *e2*.

Returns true if all elements in e1 and e2 are close to each other according to parameters atol and rtol.

**Return** a boolean

**Parameters**

- e1: input array to compare

- `e2`: input arrays to compare
- `rtol`: the relative tolerance parameter (default 1e-05)
- `atol`: the absolute tolerance parameter (default 1e-08)

| | |
|---|---|
| *isfinite* | checks for finite values |
| *isinf* | checks for infinite values |
| *isnan* | checks for NaN values |
| *isclose* | element-wise closeness detection |
| *allclose* | closeness reduction |

### 1.13.10 Reducing functions

**xtensor** provides the following reducing functions for xexpressions:

Defined in `xtensor/xmath.hpp`
**`template <class E, class X>`**
auto `xt::sum`(E &&*e*, X &&*axes*)
Sum of elements over given axes.

Returns an *xreducer* for the sum of elements over given *axes*.

**Return** an *xreducer*

**Parameters**

- `e`: an *xexpression*
- `axes`: the axes along which the sum is performed (optional)

**`template <class E, class X>`**
auto `xt::prod`(E &&*e*, X &&*axes*)
Product of elements over given axes.

Returns an *xreducer* for the product of elements over given *axes*.

**Return** an *xreducer*

**Parameters**

- `e`: an *xexpression*
- `axes`: the axes along which the product is computed (optional)

**`template <class E, class X>`**
auto `xt::mean`(E &&*e*, X &&*axes*)
Mean of elements over given axes.

Returns an *xreducer* for the mean of elements over given *axes*.

**Return** an *xexpression*

**Parameters**

- `e`: an *xexpression*
- `axes`: the axes along which the mean is computed (optional)

Defined in `xtensor/xnorm.hpp`
**`template <class E, class X>`**

auto xt::**norm_l0**(E &&*e*, X &&*axes*)

L0 (count) pseudo-norm of an array-like argument over given axes.

Returns an *xreducer* for the L0 pseudo-norm of the elements across given *axes*.

**Return** an *xreducer* When no axes are provided, the norm is calculated over the entire array. In this case, the reducer represents a scalar result, otherwise an array of appropriate dimension.

**Parameters**

- e: an *xexpression*
- axes: the axes along which the norm is computed (optional)

**template <class** E, **class** X>
auto xt::**norm_l1**(E &&*e*, X &&*axes*)

L1 norm of an array-like argument over given axes.

Returns an *xreducer* for the L1 norm of the elements across given *axes*.

**Return** an *xreducer* When no axes are provided, the norm is calculated over the entire array. In this case, the reducer represents a scalar result, otherwise an array of appropriate dimension.

**Parameters**

- e: an *xexpression*
- axes: the axes along which the norm is computed (optional)

**template <class** E, **class** X>
auto xt::**norm_sq**(E &&*e*, X &&*axes*)

Squared L2 norm of an array-like argument over given axes.

Returns an *xreducer* for the squared L2 norm of the elements across given *axes*.

**Return** an *xreducer* When no axes are provided, the norm is calculated over the entire array. In this case, the reducer represents a scalar result, otherwise an array of appropriate dimension.

**Parameters**

- e: an *xexpression*
- axes: the axes along which the norm is computed (optional)

**template <class** E, **class** X>
auto xt::**norm_l2**(E &&*e*, X &&*axes*)

L2 norm of an array-like argument over given axes.

Returns an *xreducer* for the L2 norm of the elements across given *axes*.

**Return** an *xreducer* (specifically: `sqrt(norm_sq(e, axes))`)

**Parameters**

- e: an *xexpression*
- axes: the axes along which the norm is computed

**template <class** E, **class** X>
auto xt::**norm_linf**(E &&*e*, X &&*axes*)

Infinity (maximum) norm of an array-like argument over given axes.

Returns an *xreducer* for the infinity norm of the elements across given *axes*.

**Return** an *xreducer* When no axes are provided, the norm is calculated over the entire array. In this case, the reducer represents a scalar result, otherwise an array of appropriate dimension.

**Parameters**

- e: an *xexpression*

- `axes`: the axes along which the norm is computed (optional)

**template <class** E, **class** X>
auto `xt::norm_lp_to_p`(E &&*e*, double *p*, X &&*axes*)

p-th power of the Lp norm of an array-like argument over given axes.

Returns an *xreducer* for the p-th power of the Lp norm of the elements across given *axes*.

**Return** an *xreducer* When no axes are provided, the norm is calculated over the entire array. In this case, the reducer represents a scalar result, otherwise an array of appropriate dimension.

**Parameters**

- `e`: an *xexpression*
- `p`:
- `axes`: the axes along which the norm is computed (optional)

**template <class** E, **class** X>
auto `xt::norm_lp`(E &&*e*, double *p*, X &&*axes*)

Lp norm of an array-like argument over given axes.

Returns an *xreducer* for the Lp norm (p != 0) of the elements across given *axes*.

**Return** an *xreducer* When no axes are provided, the norm is calculated over the entire array. In this case, the reducer represents a scalar result, otherwise an array of appropriate dimension.

**Parameters**

- `e`: an *xexpression*
- `p`:
- `axes`: the axes along which the norm is computed (optional)

**template <class** E, XTENSOR_REQUIRE< is_xexpression< E >::value >>
auto `xt::norm_induced_l1`(E &&*e*)

Induced L1 norm of a matrix.

Returns an *xreducer* for the induced L1 norm (i.e. the maximum of the L1 norms of e's columns).

**Return** an *xreducer*

**Parameters**

- `e`: a 2D *xexpression*

**template <class** E, XTENSOR_REQUIRE< is_xexpression< E >::value >>
auto `xt::norm_induced_linf`(E &&*e*)

Induced L-infinity norm of a matrix.

Returns an *xreducer* for the induced L-infinity norm (i.e. the maximum of the L1 norms of e's rows).

**Return** an *xreducer*

**Parameters**

- `e`: a 2D *xexpression*

| | |
|---|---|
| *sum* | sum of elements over given axes |
| *prod* | product of elements over given axes |
| *mean* | mean of elements over given axes |
| *norm_l0* | L0 pseudo-norm over given axes |
| *norm_l1* | L1 norm over given axes |
| *norm_sq* | Squared L2 norm over given axes |
| *norm_l2* | L2 norm over given axes |
| *norm_linf* | Infinity norm over given axes |
| *norm_lp_to_p* | p_th power of Lp norm over given axes |
| *norm_lp* | Lp norm over given axes |
| *norm_induced_l1* | Induced L1 norm of a matrix |
| *norm_induced_linf* | Induced L-infinity norm of a matrix |

## 1.14 Compiler workarounds

This page tracks the workarounds for the various compiler issues that we encountered in the development. This is mostly of interest for developers interested in contributing to xtensor.

### 1.14.1 Visual Studio 2015 and `std::enable_if`

With Visual Studio, `std::enable_if` evaluates its second argument, even if the condition is false. This is the reason for the presence of the indirection in the implementation of the `xfunction_type_t` meta-function.

### 1.14.2 GCC-4.9 and Clang < 3.8 and constexpr `std::min` and `std::max`

`std::min` and `std::max` are not constexpr in these compilers. In `xio.hpp`, we locally define a `XTENSOR_MIN` macro used instead of `std::min`. The macro is undefined right after it is used.

### 1.14.3 Clang < 3.8 matching `initializer_list` with static arrays

Old versions of Clang don't handle overload resolution with braced initializer lists correctly: braced initializer lists are not properly matched to static arrays. This prevent compile-time detection of the length of a braced initializer list.

A consequence is that we need to use stack-allocated shape types in these cases. Workarounds for this compiler bug arise in various files of the code base. Everywhere, the handling of *Clang < 3.8* is wrapped with checks for the `X_OLD_CLANG` macro.

### 1.14.4 GCC < 5.1 and `std::is_trivially_default_constructible`

The version of the STL shipped with versions of GCC older than 5.1 are missing a number of type traits, such as `std::is_trivially_default_constructible`. However, for some of them, equivalent type traits with different names are provided, such as `std::has_trivial_default_constructor`.

In this case, we polyfill the proper standard names using the deprecated `std::has_trivial_default_constructor`. This must also be done when the compiler is clang when it makes use of the GCC implementation of the STL, which is the default behavior on linux. Properly detecting the version of the GCC STL used by clang cannot be done with the `__GNUC__` macro, which are overridden by clang. Instead, we check for the definition of the macro `_GLIBCXX_USE_CXX11_ABI` which is only defined with GCC versions greater than 5.

### 1.14.5 GCC-6 and the signature of `std::isnan` and `std::isinf`

We are not directly using `std::isnan` or `std::isinf` for the implementation of `xt::isnan` and `xt::isinf`, as a workaround to the following bug in GCC-6 for the following reason.

- C++11 requires that the `<cmath>` header declares `bool std::isnan(double)` and `bool std::isinf(double)`.
- C99 requires that the `<math.h>` header declares `int ::isnan(double)` and `int ::isinf(double)`.

These two definitions would clash when importing both headers and using namespace std.

As of version 6, GCC detects whether the obsolete functions are present in the C `<math.h>` header and uses them if they are, avoiding the clash. However, this means that the function might return int instead of bool as C++11 requires, which is a bug.

## 1.15 Build and configuration

### 1.15.1 Build

`xtensor` build supports the following options:

- `BUILD_TESTS`: enables the `xtest` and `xbenchmark` targets (see below).
- `DOWNLOAD_GTEST`: downloads `gtest` and builds it locally instead of using a binary installation.
- `GTEST_SRC_DIR`: indicates where to find the `gtest` sources instead of downloading them.
- `XTENSOR_ENABLE_ASSERT`: activates the assertions in `xtensor`.
- `XTENSOR_CHECK_DIMENSION`: turns on `XTENSOR_ENABLE_ASSERT` and activates dimensions check in `xtensor`. Note that the dimensions check should not be activated if you expect `operator()` to perform broadcasting.
- `XTENSOR_USE_XSIMD`: enables simd acceleration in `xtensor`. This requires that you have xsimd installed on your system.

All these options are disabled by default. Enabling `DOWNLOAD_GTEST` or setting `GTEST_SRC_DIR` enables `BUILD_TESTS`.

If the `BUILD_TESTS` option is enabled, the following targets are available:

- xtest: builds an run the test suite.
- xbenchmark: builds and runs the benchmarks.

For instance, building the test suite of `xtensor` with assertions enabled:

```
mkdir build
cd build
cmake -DBUILD_TESTS=ON -DXTENSOR_ENABLE_ASSERT=ON ../
make xtest
```

Building the test suite of `xtensor` where the sources of `gtest` are located in e.g. `/usr/share/gtest`:

```
mkdir build
cd build
cmake -DGTEST_SRC_DIR=/usr/share/gtest ../
make xtest
```

### 1.15.2 Configuration

`xtensor` can be configured via macros, which must be defined *before* including any of its header. Here is a list of available macros:

- `XTENSOR_ENABLE_ASSERT`: enables assertions in xtensor, such as bound check.
- `XTENSOR_ENABLE_CHECK_DIMENSION`: enables the dimensions check in `xtensor`. Note that this option should not be turned on if you expect `operator()` to perform broadcasting.
- `XTENSOR_USE_XSIMD`: enables simd acceleration in `xtensor`. This requires that you have xsimd installed on your system.
- `DEFAULT_DATA_CONTAINER(T, A)`: defines the type used as the default data container for tensors and arrays. `T` is the `value_type` of the container and `A` its `allocator_type`.
- `DEFAULT_SHAPE_CONTAINER(T, EA, SA)`: defines the type used as the default shape container for tensors and arrays. `T` is the `value_type` of the data container, `EA` its `allocator_type`, and `SA` is the `allocator_type` of the shape container.
- `DEFAULT_LAYOUT`: defines the default layout (row_major, column_major, dynamic) for tensors and arrays. We *strongly* discourage using this macro, which is provided for testing purpose. Prefer defining alias types on tensor and array containers instead.

## 1.16 Extending xtensor

`xtensor` provides means to plug external data structures into its expression engine without copying any data.

### 1.16.1 Adapting one-dimensional containers

You may want to use your own one-dimensional container as a backend for tensor data containers and even for the shape or the strides. This is the simplest structure to plug into `xtensor`. In the following example, we define new container and adaptor types for user-specified storage and shape types.

```
// Assuming container_type and shape_type are third-party library containers
using my_array_type = xt::xarray_container<container_type, shape_type>;
using my_adaptor_type = xt::xarray_adaptor<container_type, shape_type>;

// Or, working with a fixed number of dimensions
using my_tensor_type = xt::xtensor_container<container_type, 3>;
using my_adaptor_type = xt::xtensor_adaptor<container_type, 3>;
```

These new types will have all the features of the core `xt::xtensor` and `xt::xarray` types. `xt::xarray_container` and `xt::xtensor_container` embed the data container, while `xt::xarray_adaptor` and `xt::xtensor_adaptor` hold a reference on an already initialized container.

A requirement for the user-specified containers is to provide a minimal `std::vector`-like interface, that is:

- usual typedefs for STL sequences
- random access methods (`operator[]`, `front`, `back` and `data`)
- iterator methods (`begin`, `end`, `cbegin`, `cend`)
- `size` and `resize` methods

`xtensor` does not require that the container has a contiguous memory layout, only that it provides the aforementioned interface. In fact, the container could even be backed by a file on the disk, a database or a binary message.

### 1.16.2 Structures that embed shape and strides

Some structures may gather data container, shape and strides, making them impossible to plug into `xtensor` with the method above. This section illustrates how to adapt such structures with the following simple example:

```
template <class T>
struct raw_tensor
{
    using container_type = std::vector<T>;
    using shape_type = std::vector<std::size_t>;
    container_type m_data;
    shape_type m_shape;
    shape_type m_strides;
    shape_type m_backstrides;
    static constexpr layout_type layout = layout_type::dynamic;
};
```

#### Define inner types

The following tells `xtensor` which types must be used for getting shape, strides, and data:

```
template <class T>
struct xcontainer_inner_types<raw_tensor<T>>
{
    using container_type = typename raw_tensor<T>::container_type;
    using inner_shape_type = typename raw_tensor<T>::shape_type;
    using inner_strides_type = inner_shape_type;
    using inner_backstrides_type = inner_shape_type;
    using shape_type = inner_shape_type;
    using strides_type = inner_shape_type;
    using backstrides_type = inner_shape_type;
    static constexpr layout_type layout = raw_tensor<T>::layout;
};
```

The `inner_XXX_type` are the types used to store and read the shape, strides an backstrides, while the other ones are used for reshaping. Most of the time, they will be the same; differences come when inner types cannot be instantiated out of the box (because they are linked to python buffer for instance).

Next, bring all the iterable features with this simple definition:

```
template <class T>
struct xiterable_inner_types<raw_tensor<T>>
    : xcontainer_iterable_types<raw_tensor<T>>
{
};
```

#### Inherit

Next step is to inherit from the `xcontainer` and the `xcontainer_semantic` classes:

```
template <class T>
class raw_tensor_adaptor : public xcontainer<raw_tensor_adaptor<T>>,
                           public xcontainer_semantic<raw_tensor_adaptor<T>>
{
    ...
};
```

Thanks to the previous structures definition, inheriting from `xcontainer` brings almost all the container API available in the other entities of `xtensor`, while inheriting from `xtensor_semantic` brings the support for mathematical operations.

NOTE: if we were to design a class that takes a reference on `raw_tensor` instead of embedding an instance, we would inherit from `xadaptor_semantic` instead of `xcontainer_semantic`.

### Define semantic

`xtensor` classes have full value semantic, so you may define the constructors specific to your structures, and use the default copy and move constructors and assign operators. Note these last ones *must* be declared as they are declared as `protected` in the base class.

```
template <class T>
class raw_tensor_adaptor : public xcontainer<raw_tensor_adaptor<T>>,
                           public xcontainer_semantic<raw_tensor_adaptor<T>>
{

public:

    using self_type = raw_tensor_adaptor<T>;
    using base_type = xcontainer<self_type>;
    using semantic_base = xcontainer_semantic<self_type>;

    // ... specific constructors here

    raw_tensor_adaptor(const raw_tensor_adaptor&) = default;
    raw_tensor_adaptor& operator=(const raw_tensor_adaptor&) = default;

    raw_tensor_adaptor(raw_tensor_adaptor&&) = default;
    raw_tensor_adaptor& operator=(raw_tensor_adaptor&&) = default;

    template <class E>
    raw_tensor_type(const xexpression<E>& e)
        : base_type()
    {
        semantic_base::assign(e);
    }

    template <class E>
    self_type& operator=(const xexpression<E>& e)
    {
        return semantic_base::operator=(e);
    }
};
```

The last two methods are extended copy constructor and assign operator. They allow to write things like

```
using tensor_type = raw_tensor_adaptor<double>;
tensor_type a, b, c;
// .... init a, b and c
tnesor_type d = a + b - c;
```

#### Implement the reshape methods

The next methods to define are the overloads of `reshape`. `xtensor` provides utilities functions to compute strides based on the shape and the layout, so the implementation of the `reshape` overloads is straightforward:

```
#include "xtensor/xstrides.hpp" // for utitilities functions

template <class T>
void reshape(const shape_type& shape)
{
    if(m_shape != shape)
        reshape(shape, layout::row_major);
}

template <class T>
void reshape(const shape_type& shape, layout l)
{
    m_raw.m_shape = shape;
    m_raw.m_strides.resize(shape.size());
    m_raw.m_backstrides.resize(shape.size());
    size_type data_size = compute_strides(m_shape, l, m_strides, m_backstrides);
    m_raw.m_data.resize(data_size);
}

template <class T>
void reshape(const shape_type& shape, const strides_type& strides)
{
    m_raw.m_shape = shape;
    m_raw.m_strides = strides;
    m_raw.m_backstrides.resize(shape.size());
    adapt_strides(m_raw.m_shape, m_raw.m_strides, m_raw.m_backstrides);
    m_raw.m_data.resize(compute_size(m_shape));
}
```

#### Implement private accessors

`xcontainer` assume the following methods are implemented in its inheriting class:

```
inner_shape_type& shape_impl();
const inner_shape_type& shape_impl() const;

inner_strides_type& strides_impl();
const inner_strides_type& strides_impl() const;

inner_backstrides_type& backstrides_impl();
const inner_backstrides_type& backstrides_impl() const;
```

However, since `xcontainer` provides a public API for getting the shape and the strides, these methods should be declared `protected` or `private` and `xcontainer` should be declared as a friend class so that it can access them.

### 1.16.3 Embedding a full tensor structure

You may need to plug structures that already provide n-dimensional access methods, instead of a one-dimensional container with a strided index scheme. This section illustrates how to adapt such structures with the following (minimal) API:

```
template <class T>
class table
{

public:

    using shape_type = std::vector<std::size_t>;

    const shape_type& shape() const;

    template <class... Args>
    T& operator()(Args... args);

    template <class... Args>
    const T& operator()(Args... args) const;

    template <class It>
    T& element(It first, It last);

    template <class It>
    const T& element(It first, It last) const;
};
```

### Define inner types

The following definitions are required:

```
template <class T>
struct xcontainer_inner_type<table<T>>
{
    using temporary_type = table<T>;
};

template <class T>
struct xiterable_inner_types<table<T>>
{
    using inner_shape_type = typename table<T>::shape_type;
    using stepper = xindexed_stepper<table<T>, false>;
    using const_stepper = xindexed_stepper<table<T>, true>;
};
```

### Inheritance

Next step is to inherit from the `xiterable` and `xcontainer_semantic` classes, and to define a bunch of typedefs.

```
template<class T>
class table_adaptor : public xiterable<table_adaptor<T>>,
                      public xcontainer_semantic<table_adaptor<T>>
{

public:

    using self_type = table<T>;
```

```
    using value_type = T;
    using reference = T&;
    using const_reference = const T&;
    using pointer = T*;
    using const_pointer = const T*;
    using size_type = std::size_t;
    using difference_type = std::ptrdiff_t;

    using inner_shape_type = typename table<T>::shape_type;
    using inner_stride_stype = inner_shape_type;
    using shape_type = inner_shape_type;
    using strides_type = inner_strides_type;

    using iterable_base = xexpression_iterable<self_type>;
    using stepper = typename iterable_base::stepper;
    using const_stepper = typename iterable_base::const_stepper;
};
```

The iterator and stepper used here may not be the most optimal for `table`, however they are guaranteed to work as long as `table` provides an access operator based on indices.

NOTE: we inherit from `xcontainer_semantic` because we assume the `table_adaptor` class embeds an instance of `table`. If it tooks a reference on it, we would inherit from `xadaptor_semantic` instead.

### Define semantic

As for one-dimensional containers adaptors, you must define constructors and at least declare default copy and move constuctor and assign operator. You also must define extended copy constructor and assign operator.

```
template <class T>
class table_adaptor : public xiterable<table_adaptor<T>>,
                      public xcontainer_semantic<table_adaptor<T>>
{

public:

    // .... typedefs
    // .... specific constructors

    table_adaptor(const table_adaptor&) = default;
    table_adaptor& operator=(const table_adaptor&) = default;

    table_adaptor(table_adaptor&&) = default;
    table_adaptor& operator=(table_adaptor&&) = default;

    template <class E>
    table_adaptor(const xexpression<E>& e)
        : base_type()
    {
        semantic_base::assign(e);
    }

    template <class E>
    self_type& operator=(const xexpression<E>& e)
    {
        return semantic_base::operator=(e);
```

```
    }
};
```

### Implement access operators

`xtensor` requires that the following access operators are defined

```
template <class... Args>
reference operator()(Args... args)
{
    // Should forward to table<T>:operator()(args...)
}

template <class... Args>
const_reference operator()(Args... args) const
{
    // Should forward to table<T>::operator()(args...)
}

reference operator[](const xindex& index)
{
    return element(index.cbegin(), index.cend());
}

const_reference operator[](const xindex& index) const
{
    return element(index.cbegin(), index.cend());
}

reference operator[](size_type i)
{
    return operator()(i);
}

const_reference operator[](size_type i) const
{
    return operator()(i);
}

template <class It>
reference element(It first, It last)
{
    // Should forward to table<T>::element(first, last)
}

template <class It>
const_reference element(It first, It last)
{
    // Should forward to table<T>::element(first, last)
}
```

### Implement broadcast mechanic

This part is relatively straightforward:

```
size_type dimension() const
{
    return shape().size();
}

const shape_type& shape() const
{
    // Should forward to table<T>::shape()
}

template <class S>
bool broadcast_shape(const S& s) const
{
    // Available in "xtensor/xtrides.hpp"
    return xt::broadcast_shape(shape(), s);
}

template <class S>
bool is_trivial_broadcast(const S& str) const noexcept
{
    return false;
}
```

### Implement reshape overloads

This is very similar to what must be done for one-dimensional containers, except you may ignore the layout and the strides in the implementation. However, these overloads are still required.

### Provide a stepper API

The last required step is to provide a stepper API, on which are built iterators.

```
template <class ST>
stepper stepper_begin(const ST& s)
{
    size_type offset = s.size() - dimension();
    return stepper(this, offset);
}

template <class ST>
stepper stepper_end(const ST& s)
{
    size_type offset = s.size() - dimension();
    return stepper(this, offset, true);
}

template <class ST>
const_stepper stepper_begin(const ST& s) const
{
    size_type offset = s.size() - dimension();
    return const_stepper(this, offset);
}

template <class ST>
const_stepper stepper_end(const ST& s) const
```

```
{
    size_type offset = s.size() - dimension();
    return const_stepper(this, offset, true);
}
```

## 1.17 Releasing xtensor

### 1.17.1 Releasing a new version

From the master branch of xtensor

- Make sure that you are in sync with the master branch of the upstream remote.
- Update the changelog.
- In file `xtensor_config.hpp`, set the macros for `XTENSOR_VERSION_MAJOR`, `XTENSOR_VERSION_MINOR` and `XTENSOR_VERSION_PATCH` to the desired values.
- In file `README.md`, modify the binder link to point to the new release.
- 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`)

### 1.17.2 Updating the conda-forge recipe

xtensor 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.

## 1.18 From numpy to xtensor

### 1.18.1 Containers

Two container types are provided. `xarray` (dynamic number of dimensions) and `xtensor` (static number of dimensions).

| Python 3 - numpy | C++ 14 - xtensor |
|---|---|
| `np.array([[3, 4], [5, 6]])` | `xt::xarray<double>({{3, 4}, {5, 6}})`<br>`xt::xtensor<double, 2>({{3, 4}, {5, 6}})` |
| `arr.reshape([3, 4])` | `arr.reshape{{3, 4})` |

### 1.18.2 Initializers

Lazy helper functions return tensor expressions. Return types don't hold any value and are evaluated upon access or assignment. They can be assigned to a container or directly used in expressions.

| Python 3 - numpy | C++ 14 - xtensor |
|---|---|
| `np.linspace(1.0, 10.0, 100)` | `xt::linspace<double>(1.0, 10.0, 100)` |
| `np.logspace(2.0, 3.0, 4)` | `xt::logspace<double>(2.0, 3.0, 4)` |
| `np.arange(3, 7)` | `xt::arange(3, 7)` |
| `np.eye(4)` | `xt::eye(4)` |
| `np.zeros([3, 4])` | `xt::zeros<double>({3, 4})` |
| `np.ones([3, 4])` | `xt::ones<double>({3, 4})` |
| `np.meshgrid(x0, x1, x2, indexing='ij')` | `xt::meshgrid(x0, x1, x2)` |

xtensor's `meshgrid` implementation corresponds to numpy's `'ij'` indexing order.

### 1.18.3 Broadcasting

xtensor offers lazy numpy-style broadcasting, and universal functions. Unlike numpy, no copy or temporary variables are created.

| Python 3 - numpy | C++ 14 - xtensor |
|---|---|
| `a[:, np.newaxis]`<br>`a[:5, 1:]`<br>`a[5:1:-1, :]` | `xt::view(a, xt::all(), xt::newaxis())`<br>`xt::view(a, xt::range(_, 5), xt::range(1, _))`<br>`xt::view(a, xt::range(5, 1, -1), xt::all())` |
| `np.broadcast(a, [4, 5, 7])` | `xt::broadcast(a, {4, 5, 7})` |
| `np.vectorize(f)` | `xt::vectorize(f)` |
| `a[a > 5]` | `xt::filter(a, a > 5)` |
| `a[[0, 1], [0, 0]]` | `xt::index_view(a, {{0, 0}, {1, 0}})` |

### 1.18.4 Random

The random module provides simple ways to create random tensor expressions, lazily.

| Python 3 - numpy | C++ 14 - xtensor |
|---|---|
| `np.random.seed(0)` | `xt::random::seed(0)` |
| `np.random.randn(10, 10)` | `xt::random::randn<double>({10, 10})` |
| `np.random.randint(10, 10)` | `xt::random::randint<int>({10, 10})` |
| `np.random.rand(3, 4)` | `xt::random::rand<double>({3, 4})` |

### 1.18.5 Concatenation

Concatenating expressions does not allocate memory, it returns a tensor expression holding closures on the specified arguments.

| Python 3 - numpy | C++ 14 - xtensor |
|---|---|
| `np.stack([a, b, c], axis=1)` | `xt::stack(xtuple(a, b, c), 1)` |
| `np.concatenate([a, b, c], axis=1)` | `xt::concatenate(xtuple(a, b, c), 1)` |

### 1.18.6 Diagonal, triangular and flip

In the same spirit as concatenation, the following operations do not allocate any memory and do not modify the underlying xexpression.

| Python 3 - numpy | C++ 14 - xtensor |
|---|---|
| `np.diag(a)` | `xt::diag(a)` |
| `np.diagonal(a)` | `xt::diagonal(a)` |
| `np.triu(a)` | `xt::triu(a)` |
| `np.tril(a, k=1)` | `xt::tril(a, 1)` |
| `np.flip(a, axis=3)` | `xt::flip(a, 3)` |
| `np.flipud(a)` | `xt::flip(a, 0)` |
| `np.fliplr(a)` | `xt::flip(a, 1)` |

### 1.18.7 Iteration

xtensor follows the idioms of the C++ STL providing iterator pairs to iterate on arrays in different fashions.

| Python 3 - numpy | C++ 14 - xtensor |
|---|---|
| `for x in np.nditer(a):` | `for(auto it=a.begin(); it!=a.end(); ++it)` |
| Iterating over `a` with a prescribed broadcasting shape | `a.begin({3, 4})`<br>`a.end({3, 4})` |
| Iterating over `a` in a row-major fashion | `a.begin<layout_type::row_major>()`<br>`a.begin<layout_type::row_major>()` |
| Iterating over `a` in a column-major fashion | `a.begin<layout_type::column_major>()`<br>`a.end<layout_type::column_major>()` |

### 1.18.8 Logical

Logical universal functions are truly lazy. `xt::where(condition, a, b)` does not evaluate `a` where `condition` is falsy, and it does not evaluate `b` where `condition` is truthy.

| Python 3 - numpy | C++ 14 - xtensor |
|---|---|
| `np.where(a > 5, a, b)` | `xt::where(a > 5, a, b)` |
| `np.where(a > 5)` | `xt::where(a > 5)` |
| `np.any(a)` | `xt::any(a)` |
| `np.all(a)` | `xt::all(a)` |
| `np.logical_and(a, b)` | `a && b` |
| `np.logical_or(a, b)` | `a \|\| b` |
| `np.isclose(a, b)` | `xt::isclose(a, b)` |
| `np.allclose(a, b)` | `xt::allclose(a, b)` |

### 1.18.9 Comparisons

| Python 3 - numpy | C++ 14 - xtensor |
|---|---|
| `np.equal(a, b)` | `xt::equal(a, b)` |
| `np.not_equal(a)` | `xt::not_equal(a)` |
| `np.nonzero(a)` | `xt::nonzero(a)` |

### 1.18.10 Complex numbers

Functions `xt::real` and `xt::imag` respectively return views on the real and imaginary part of a complex expression. The returned value is an expression holding a closure on the passed argument.

| Python 3 - numpy | C++ 14 - xtensor |
|---|---|
| `np.real(a)` | `xt::real(a)` |
| `np.imag(a)` | `xt::imag(a)` |

- The constness and value category (rvalue / lvalue) of `real(a)` is the same as that of `a`. Hence, if `a` is a non-const lvalue, `real(a)` is an non-const lvalue reference, to which one can assign a real expression.
- If `a` has complex values, the same holds for `imag(a)`. The constness and value category of `imag(a)` is the same as that of `a`.
- If `a` has real values, `imag(a)` returns `zeros(a.shape())`.

### 1.18.11 Reducers

Reducers accumulate values of tensor expressions along specified axes. When no axis is specified, values are accumulated along all axes. Reducers are lazy, meaning that returned expressons don't hold any values and are computed upon access or assigmnent.

| Python 3 - numpy | C++ 14 - xtensor |
|---|---|
| `np.sum(a, axis=[0, 1])` | `xt::sum(a, {0, 1})` |
| `np.sum(a)` | `xt::sum(a)` |
| `np.prod(a, axis=1)` | `xt::prod(a, {1})` |
| `np.prod(a)` | `xt::prod(a)` |
| `np.mean(a, axis=1)` | `xt::mean(a, {1})` |
| `np.mean(a)` | `xt::mean(a)` |

More generally, one can use the `xt::reduce(function, input, axes)` which allows the specification of an arbitrary binary function for the reduction. The binary function must be cummutative and associative up to rounding errors.

### 1.18.12 Mathematical functions

xtensor universal functions are provided for a large set number of mathematical functions.

**Basic functions:**

| Python 3 - numpy | C++ 14 - xtensor |
|---|---|
| `np.absolute(a)` | `xt::abs(a)` |
| `np.sign(a)` | `xt::sign(a)` |
| `np.remainder(a, b)` | `xt::remainder(a, b)` |
| `np.clip(a, min, max)` | `xt::clip(a, min, max)` |
| | `xt::fma(a, b, c)` |

**Exponential functions:**

| Python 3 - numpy | C++ 14 - xtensor |
|---|---|
| `np.exp(a)` | `xt::exp(a)` |
| `np.expm1(a)` | `xt::expm1(a)` |
| `np.log(a)` | `xt::log(a)` |
| `np.log1p(a)` | `xt::log1p(a)` |

**Power functions:**

| Python 3 - numpy | C++ 14 - xtensor |
|---|---|
| `np.power(a, p)` | `xt::pow(a, b)` |
| `np.sqrt(a)` | `xt::sqrt(a)` |
| `np.cbrt(a)` | `xt::cbrt(a)` |

**Trigonometric functions:**

| Python 3 - numpy | C++ 14 - xtensor |
|---|---|
| `np.sin(a)` | `xt::sin(a)` |
| `np.cos(a)` | `xt::cos(a)` |
| `np.tan(a)` | `xt::tan(a)` |

**Hyperbolic functions:**

| Python 3 - numpy | C++ 14 - xtensor |
|---|---|
| `np.sinh(a)` | `xt::sinh(a)` |
| `np.cosh(a)` | `xt::cosh(a)` |
| `np.tang(a)` | `xt::tanh(a)` |

**Error and gamma functions:**

| Python 3 - numpy | C++ 14 - xtensor |
|---|---|
| `scipy.special.erf(a)` | `xt::erf(a)` |
| `scipy.special.gamma(a)` | `xt::tgamma(a)` |
| `scipy.special.gammaln(a)` | `xt::lgamma(a)` |

**Classification functions:**

| Python 3 - numpy | C++ 14 - xtensor |
|---|---|
| `np.isnan(a)` | `xt::isnan(a)` |
| `np.isinf(a)` | `xt::isinf(a)` |
| `np.isfinite(a)` | `xt::isfinite(a)` |

### 1.18.13 Linear algebra

Many functions found in the `numpy.linalg` module are implemented in xtensor-blas, a seperate package offering BLAS and LAPACK bindings, as well as a convenient interface replicating the `linalg` module.

Please note, however, that while we're trying to be as close to NumPy as possible, some features are not implemented yet. Most prominently that is broadcasting for all functions except for `dot`.

**Matrix and vector products**

| Python 3 - numpy | C++ 14 - xtensor |
|---|---|
| `np.dot(a, b)` | `xt::linalg::dot(a, b)` |
| `np.vdot(a, b)` | `xt::linalg::vdot(a, b)` |
| `np.outer(a, b)` | `xt::linalg::outer(a, b)` |
| `np.matrix_power(a, 123)` | `xt::linalg::matrix_power(a, 123)` |
| `np.kron(a, b)` | `xt::linalg::kron(a, b)` |

**Decompositions**

| Python 3 - numpy | C++ 14 - xtensor |
|---|---|
| `np.linalg.cholesky(a)` | `xt::linalg::cholesky(a)` |
| `np.linalg.qr(a)` | `xt::linalg::qr(a)` |
| `np.linalg.svd(a)` | `xt::linalg::svd(a)` |

**Matrix eigenvalues**

| Python 3 - numpy | C++ 14 - xtensor |
|---|---|
| `np.linalg.eig(a)` | `xt::linalg::eig(a)` |
| `np.linalg.eigvals(a)` | `xt::linalg::eigvals(a)` |
| `np.linalg.eigh(a)` | `xt::linalg::eigh(a)` |
| `np.linalg.eigvalsh(a)` | `xt::linalg::eigvalsh(a)` |

**Norms and other numbers**

| Python 3 - numpy | C++ 14 - xtensor |
|---|---|
| `np.linalg.norm(a, order=2)` | `xt::linalg::norm(a, 2)` |
| `np.linalg.cond(a)` | `xt::linalg::cond(a)` |
| `np.linalg.det(a)` | `xt::linalg::det(a)` |
| `np.linalg.matrix_rank(a)` | `xt::linalg::matrix_rank(a)` |
| `np.linalg.slogdet(a)` | `xt::linalg::slogdet(a)` |
| `np.trace(a)` | `xt::linalg::trace(a)` |

**Solving equations and inverting matrices**

| Python 3 - numpy | C++ 14 - xtensor |
|---|---|
| `np.linalg.inv(a)` | `xt::linalg::inv(a)` |
| `np.linalg.pinv(a)` | `xt::linalg::pinv(a)` |
| `np.linalg.solve(A, b)` | `xt::linalg::solve(A, b)` |
| `np.linalg.lstsq(A, b)` | `xt::linalg::lstsq(A, b)` |

## 1.19 Notable differences with numpy

xtensor and numpy are very different libraries in their internal semantics. While xtensor is a lazy expression system, Numpy manipulates in-memory containers, however, similarities in APIs are obvious. See e.g. the numpy to xtensor cheat sheet.

And this page tracks the subtle differences of behavior of numpy and xtensor.

### 1.19.1 Zero-dimensional arrays

With numpy, 0-D arrays are nearly indistinguishable from scalars. This led to some issues w.r.t. universal functions returning scalars with 0-D array inputs instead of actual arrays...

In xtensor, 0-D expressions are not implicitely convertible to scalar values. Values held by 0-D expressions can be accessed in the same way as values of higher dimensional arrays, that is with `operator[]`, `operator()` and `element`.

### 1.19.2 Meshgrid

Numpy's version of meshgrid supports two modes: the 'xy' indexing and the 'ij' indexing.

The following code

```
import numpy as np

x1, x2, x3, x4 = [1], [10, 20], [100, 200, 300], [1000, 2000, 3000, 4000]

ij = np.meshgrid(x1, x2, x3, x4, indexing='ij')
xy = np.meshgrid(x1, x2, x3, x4, indexing='xy')

print 'ij:', [m.shape for m in ij]
print 'xy:', [m.shape for m in xy]
```

would return

```
ij: [(1, 2, 3, 4), (1, 2, 3, 4), (1, 2, 3, 4), (1, 2, 3, 4)]
xy: [(2, 1, 3, 4), (2, 1, 3, 4), (2, 1, 3, 4), (2, 1, 3, 4)]
```

In other words, the 'xy' indexing, which is the default only reverses the first two dimensions compared to the 'ij' indexing.

xtensor's version of meshgrid corresponds to the 'ij' indexing.

### 1.19.3 The random module

Like most functions of xtensor, functions of the random module return expressions that don't hold any value.

Every time an element is accessed, a new random value is generated. To fix the values of a generator, it should be assigned to a container such as xarray or xtensor.

### 1.19.4 Missing values

Support of missing values in numpy can be emulated with the masked array module, which provides a means to handle arrays that have missing or invalid data.

Support of missing values in xtensor is done through a notion of optional values, implemented in `xoptional<T, B>`, which serves both as a value type for container and as a reference proxy for optimized storage types. See the section of the documentation on *Missing values*.

### 1.19.5 Strides

Strided containers of xtensor and numpy having the same exact memory layout may have different strides when accessing them through the `strides` attribute. The reason is an optimization in xtensor, which is to set the strides to `0` in dimensions of length `1`, which simplifies the implementation of broadcasting of universal functions.

## 1.20 Closure semantics

The `xtensor` library is a tensor expression library implementing numpy-style broadcasting and universal functions, but in a lazy fashion.

If `x` and `y` are two tensor expressions with compatible shapes, the result of `x + y` is not a tensor but an expression that does not hold any value. Values of `x + y` are computed upon access or when the result is assigned to a container such as `xt::xtensor` or `xt::xarray`. The same holds for most functions in xtensor, views, broadcasting views, etc.

In order to be able to perform the differed computation of `x + y`, the returned expression must hold references, const references or copies of the members `x` and `y`, depending on how arguments were passed to `operator+`. The actual types of held by the expressions are the **closure types**.

The concept of closure type is key in the implementation of `xtensor` and appears in all the expressions defined in xtensor, and the utility functions and meta functions complements the tools of the standard libary for the move semantics.

### 1.20.1 Basic rules for determinning closure types

The two main requirements are the following:

- when an argument passed to the function returning an expression (here, `operator+`) is an *rvalue*, the closure type is always a value and the `rvalue` is *moved*.
- when an argument passed to the function returning an expression is an *lvalue reference*, the closure type is a reference of the same type.

It is important for the closure type not to be a reference when the passed argument is an rvalue, which can result in dangling references.

Following the conventions of the C++ standard library for naming type traits, we provide two type traits classes providing an implementation of these rules in the `xutils.hpp` header, `closure`, and `const_closure`. The latter adds the const qualifier even when the provided argument is not const.

```
template <class S>
struct closure
{
    using underlying_type = std::conditional_t<
        std::is_const<std::remove_reference_t<S>>::value,
        const std::decay_t<S>,
        std::decay_t<S>>;
    using type = typename std::conditional<
        std::is_lvalue_reference<S>::value,
        underlying_type&,
        underlying_type>::type;
};

template <class S>
using closure_t = typename closure<S>::type;
```

The implementation for `const_closure` is slightly shorter.

```
template <class S>
struct const_closure
{
    using underlying_type = const std::decay_t<S>;
    using type = typename std::conditional<
        std::is_lvalue_reference<S>::value,
        underlying_type&,
        underlying_type>::type;
};
```

```
template <class S>
using const_closure_t = typename const_closure<S>::type;
```

Using this mechanism, we were able to

- avoid dangling references in nested expressions,
- hold references whenever possible,
- take advantage of the move semantics when holding references is not possible.

### 1.20.2 Closure types and scalar wrappers

A requirement for `xtensor` is the ability to mix scalars and tensors in tensor expressions. In order to do so, scalar values are wrapped into the `xscalar` wrapper, which is a cheap 0-D tensor expression holding a single scalar value.

For the xscalar to be a proper proxy on the scalar value, if actually holds a closure type on the scalar value.

The logics for this is encoded into xtensor's `xclosure` type trait.

```
template <class E, class EN = void>
struct xclosure
{
    using type = closure_t<E>;
};

template <class E>
struct xclosure<E, disable_xexpression<std::decay_t<E>>>
{
    using type = xscalar<closure_t<E>>;
};

template <class E>
using xclosure_t = typename xclosure<E>::type;
```

In doing so, we ensure const-correctness, we avoid dangling reference, and ensure that lvalues remain lvalues. The *const_xclosure* follows the same scheme:

```
template <class E, class EN = void>
struct const_xclosure
{
    using type = const_closure_t<E>;
};

template <class E>
struct const_xclosure<E, disable_xexpression<std::decay_t<E>>>
{
    using type = xscalar<std::decay_t<E>>;
};

template <class E>
using const_xclosure_t = typename const_xclosure<E>::type;
```

### 1.20.3 Writing functions that return expressions

*xtensor closure semantics are not meant to prevent users from doing mistakes, since it would also prevent them from doing something clever.*

This section covers cases where understanding C++ move semantics and xtensor closure semantics helps writing better code with xtensor.

### Returning evaluated or unevaluated expressions

A key feature of xtensor is that a function returning e.g. `x + y / z` where `x`, `y` and `z` are xtensor expressions does not actually perform any computation. It is only evaluated upon access or assignment. The returned expression holds values or references for `x`, `y` and `z` depending on the lvalue-ness of the variables passed to the expression, using the *closure semantics* described earlier. This may result in dangling references when using local variables of a function in an unevaluated expression, unless one properly forwards / move the variables.

---

**Note:** The following rule of thumbs prevents dangling references in the xtensor closure semantics:

- If the laziness is not important for your usecase, returning `xt::eval(x + y / z)` will return an evaluated container and avoid these complications.
- Otherwise, the key is to *move* lvalues that become invalid when leaving the current scope.

---

**Example: moving local variables and forwarding universal references**

Let us first consider the following implementation of the `mean` function in xtensor:

```
template <class E>
inline auto mean(E&& e) noexcept
{
    using value_type = typename std::decay_t<E>::value_type;
    auto size = e.size();
    auto s = sum(std::forward<E>(e));
    return std::move(s) / value_type(size);
}
```

The first thing to take into account is that the result of the final division is an expression, which performs the actual computation upon access or assignent.

- In order to perform the division, the expression must hold the values or references on the numerator and denominator.
- Since `s` is a local variable, it will be destroyed upon leaving the scope of the function, and more importantly it is an *lvalue*.
- A consequence of `s` being an lvalue and a local variable, is that the `s / value_type(size)` would end up holding a dangling const reference on `s`.
- Hence we must call return `std::move(s) / value_type(size)`.

The other place in this example where the C++ move semantics is used is the line `s = sum(std::forward<E>(e))`. The goal is to have the unevaluated `s` expression hold a const reference or a value for `e` depending on the lvalue-ness of the parameter passed to the function.

## 1.21 Related projects

### 1.21.1 xtensor-python

The xtensor-python project provides the implementation of container types compatible with `xtensor`'s expression system, `pyarray` and `pytensor` which effectively wrap numpy arrays, allowing operating on numpy arrays inplace.

### Example 1: Use an algorithm of the C++ library on a numpy array inplace

**C++ code**

```
#include <numeric>                          // Standard library import for std::accumulate
#include "pybind11/pybind11.h"              // Pybind11 import to define Python bindings
#include "xtensor/xmath.hpp"                // xtensor import for the C++ universal functions
#define FORCE_IMPORT_ARRAY                  // numpy C api loading
#include "xtensor-python/pyarray.hpp"       // Numpy bindings

double sum_of_sines(xt::pyarray<double> &m)
{
    auto sines = xt::sin(m);
    // sines does not actually hold any value
    return std::accumulate(sines.cbegin(), sines.cend(), 0.0);
}

PYBIND11_PLUGIN(xtensor_python_test)
{
    xt::import_numpy();
    pybind11::module m("xtensor_python_test", "Test module for xtensor python bindings");

    m.def("sum_of_sines", sum_of_sines,
          "Sum the sines of the input values");

    return m.ptr();
}
```

**Python code**

```
Python Code

import numpy as np
import xtensor_python_test as xt

a = np.arange(15).reshape(3, 5)
s = xt.sum_of_sines(v)
s
```

**Outputs**

```
1.2853996391883833
```

### Example 2: Create a universal function from a C++ scalar function

**C++ code**

```
#include "pybind11/pybind11.h"
#define FORCE_IMPORT_ARRAY
#include "xtensor-python/pyvectorize.hpp"
```

```cpp
#include <numeric>
#include <cmath>

namespace py = pybind11;

double scalar_func(double i, double j)
{
    return std::sin(i) - std::cos(j);
}

PYBIND11_PLUGIN(xtensor_python_test)
{
    xt::import_numpy();
    py::module m("xtensor_python_test", "Test module for xtensor python bindings");

    m.def("vectorized_func", xt::pyvectorize(scalar_func), "");

    return m.ptr();
}
```

**Python code**

```
import numpy as np
import xtensor_python_test as xt

x = np.arange(15).reshape(3, 5)
y = [1, 2, 3, 4, 5]
z = xt.vectorized_func(x, y)
z
```

**Outputs**

```
[[-0.540302,  1.257618,  1.89929 ,  0.794764, -1.040465],
 [-1.499227,  0.136731,  1.646979,  1.643002,  0.128456],
 [-1.084323, -0.583843,  0.45342 ,  1.073811,  0.706945]]
```

### 1.21.2 xtensor-python-cookiecutter

The xtensor-python-cookiecutter project helps extension authors create Python extension modules making use of *xtensor*.

It takes care of the initial work of generating a project skeleton with

- A complete setup.py compiling the extension module

A few examples included in the resulting project including

- A universal function defined from C++
- A function making use of an algorithm from the STL on a numpy array
- Unit tests
- The generation of the HTML documentation with sphinx

### 1.21.3 xtensor-julia

The xtensor-julia project provides the implementation of container types compatible with `xtensor`'s expression system, `jlarray` and `jltensor` which effectively wrap Julia arrays, allowing operating on Julia arrays inplace.

#### Example 1: Use an algorithm of the C++ library with a Julia array

**C++ code**

```
#include <numeric>                          // Standard library import for␣
↪std::accumulate
#include <cxx_wrap.hpp>                     // CxxWrap import to define Julia bindings
#include "xtensor-julia/jltensor.hpp"       // Import the jltensor container definition
#include "xtensor/xmath.hpp"                // xtensor import for the C++ universal␣
↪functions

double sum_of_sines(xt::jltensor<double, 2> m)
{
    auto sines = xt::sin(m);  // sines does not actually hold values.
    return std::accumulate(sines.cbegin(), sines.cend(), 0.0);
}

JULIA_CPP_MODULE_BEGIN(registry)
    cxx_wrap::Module mod = registry.create_module("xtensor_julia_test");
    mod.method("sum_of_sines", sum_of_sines);
JULIA_CPP_MODULE_END
```

**Julia code**

```
using xtensor_julia_test

arr = [[1.0 2.0]
       [3.0 4.0]]

s = sum_of_sines(arr)
s
```

**Outputs**

```
1.2853996391883833
```

#### Example 2: Create a numpy-style universal function from a C++ scalar function

**C++ code**

```
#include <cxx_wrap.hpp>
#include "xtensor-julia/jlvectorize.hpp"

double scalar_func(double i, double j)
{
    return std::sin(i) - std::cos(j);
}

JULIA_CPP_MODULE_BEGIN(registry)
```

```
    cxx_wrap::Module mod = registry.create_module("xtensor_julia_test");
    mod.method("vectorized_func", xt::jlvectorize(scalar_func));
JULIA_CPP_MODULE_END
```

**Julia code**

```
using xtensor_julia_test

x = [[ 0.0  1.0  2.0  3.0  4.0]
     [ 5.0  6.0  7.0  8.0  9.0]
     [10.0 11.0 12.0 13.0 14.0]]
y = [1.0, 2.0, 3.0, 4.0, 5.0]
z = xt.vectorized_func(x, y)
z
```

**Outputs**

```
[[-0.540302  1.257618  1.89929   0.794764 -1.040465],
 [-1.499227  0.136731  1.646979  1.643002  0.128456],
 [-1.084323 -0.583843  0.45342   1.073811  0.706945]]
```

### 1.21.4 xtensor-julia-cookiecutter

The xtensor-julia-cookiecutter project helps extension authors create Julia extension modules making use of *xtensor*.

It takes care of the initial work of generating a project skeleton with

- A complete read-to-use Julia package

A few examples included in the resulting project including

- A numpy-style universal function defined from C++
- A function making use of an algorithm from the STL on a numpy array
- Unit tests
- The generation of the HTML documentation with sphinx

### 1.21.5 xtensor-r

The xtensor-r project provides the implementation of container types compatible with `xtensor`'s expression system, `rarray` and `rtensor` which effectively wrap R arrays, allowing operating on R arrays inplace.

#### Example 1: Use an algorithm of the C++ library on a R array inplace

**C++ code**

```
#include <numeric>                     // Standard library import for std::accumulate
#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)]]

// [[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
```

### 1.21.6 xtensor-blas

The xtensor-blas project is an extension to the xtensor library, offering bindings to BLAS and LAPACK libraries through cxxblas and cxxlapack from the FLENS project. `xtensor-blas` powers the `xt::linalg` functionalities, which are the counterpart to numpy's `linalg` module.

### 1.21.7 xtensor-fftw

The xtensor-fftw project is an extension to the xtensor library, offering bindings to the fftw library. `xtensor-fftw` powers the `xt::fftw` functionalities, which are the counterpart to numpy's `fft` module.

# Index

## X