xtensor

Feb 19, 2018

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

We use a shared copyright model that enables all contributors to maintain the copyright on their contributions.

This software is licensed under the BSD-3-Clause license. See the LICENSE file for details.

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 xtensor 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 Using the Spack package

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

```
spack install xtensor
spack load --dependencies xtensor
```

1.1.4 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 0.15.4

- fix gcc-7 error w.r.t. the use of `assert` #648.

1.2.2 0.15.3

- add missing headers to cmake installation and tests #647.

1.2.3 0.15.2

- `xshape` implementation #572.
- xfixed container #586.
- protected `xcontainer::derived_cast` #627.
- const reference fix #632.
- `xgenerator` access operators fixed #643.
- contiguous layout optiimzation #645.

1.2.4 0.15.1

- `xarray_adaptor` fixed #618.
- `xtensor_adaptor` fixed #620.
- fix in `xreducer` steppers #622.
- documentation improved #621. #623. #625.
- warnings removed #624.

1.2.5 0.15.0

Breaking changes

- change `reshape` to `resize`, and add throwing `reshape` #598.
- moved to modern cmake #611.

New features

- unravel function #589.
- random access iterators #596.

Other changes

- upgraded to google/benchmark version 1.3.0 #583.
- `XTENSOR_ASSERT` renamed into `XTENSOR_TRY`, new `XTENSOR_ASSERT` #603.
- `adapt` fixed #604.
- VC14 warnings removed #608.
- `xfunctor_iterator` is now a random access iterator #609.
- removed `old-style-cast` warnings #610.

1.2.6 0.14.1

New features

- sort, argmin and argmax #549.
- `xscalar_expression_tag` #582.

Other changes

- accumulator improvements #570.
- benchmark cmake fixed #571.
- allocator_type added to container interface #573.
- allow conda-forge as fallback channel #575.

- arithmetic mixing optional assemblies and scalars fixed #578.
- arithmetic mixing optional assemblies and optionals fixed #579.
- `operator==` restricted to xtensor and xoptional expressions #580.

1.2.7 0.14.0

Breaking changes

- `xadapt` renamed into `adapt` #563.
- Naming consistency #565.

New features

- add `random::choice` #547.
- evaluation strategy and accumulators. #550.
- modulus operator #556.
- `adapt`: default overload for 1D arrays #560.
- Move semantic on `adapt` #564.

Other changes

- optional fixes to avoid ambiguous calls #541.
- narrative documentation about `xt::adapt` #544.
- `xfunction` refactoring #545.
- SIMD acceleration for AVX fixed #557.
- allocator fixes #558. #559.
- return type of `view::strides()` fixed #568.

1.2.8 0.13.2

- Support for complex version of `isclose` #512.
- Fixup static layout in `xstrided_view` #536.
- `xexpression::operator[]` now take support any type of sequence #537.
- Fixing `xinfo` issues for Visual Studio. #529.
- Fix const-correctness in `xstrided_view`. #526.

1.2.9 0.13.1

- More general floating point type #518.
- Do not require functor to be passed via rvalue reference #519.
- Documentation improved #520.
- Fix in xreducer #521.

1.2.10 0.13.0

Breaking changes

- The API for `xbuffer_adaptor` has changed. The template parameter is the type of the buffer, not just the value type #482.
- Change `edge_items` print option to `edgeitems` for better numpy consistency #489.
- xtensor now depends on `xtl` version *~0.3.3* #508.

New features

- Support for parsing the `npy` file format #465.
- Creation of optional expressions from value and boolean expressions (optional assembly) #496.
- Support for the explicit cast of expressions with different value types #491.

Other changes

- Addition of broadcasting bitwise operators #459.
- More efficient optional expression system #467.
- Migration of benchmarks to the Google benchmark framework #473.
- Container semantic and adaptor semantic merged #475.
- Various fixes and improvements of the strided views #480. #481.
- Assignment now performs basic type conversion #486.
- Workaround for a compiler bug in Visual Studio 2017 #490.
- MSVC 2017 workaround #492.
- The `size()` method for containers now returns the total number of elements instead of the buffer size, which may differ when the smallest stride is greater than `1` #502.
- The behavior of `linspace` with integral types has been made consistent with numpy #510.

1.2.11 0.12.1

- Fix issue with slicing when using heterogeneous integral types #451.

1.2.12 0.12.0

Breaking changes

- `xtensor` now depends on `xtl` version *0.2.x* #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.

1.2.13 0.11.3

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

1.2.14 0.11.2

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

1.2.15 0.11.1

- Fixes some warnings in julia and python bindings

1.2.16 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}}
```

Index Access

```
#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}};

std::cout << arr1(0, 0) << std::endl;

xt::xarray<int> arr2
  {1, 2, 3, 4, 5, 6, 7, 8, 9};

std::cout << arr2(0);
```

Outputs:

```
1.0
1
```

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.

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

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

(continues on next page)

(continued from previous page)

```
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.

```
#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 `resize` 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, `resize` 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 resizing 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, and the global size of the container has to stay the same.
- `resize()` resizes 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 resized. 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 resized, 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 resized, 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 resize 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 Adapting 1-D containers

xtensor can adapt one-dimensional containers in place, and provide them a tensor interface. Only random access containers can be adapted.

1.6.1 Adapting std::vector

The following example shows how to bring an `std::vector` into the expression system of *xtensor*:

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

std::vector<double> v = {1., 2., 3., 4., 5., 6. };
std::vector<std::size_t> shape = { 2, 3 };
auto a1 = xt::adapt(v, shape);

xt::xarray<double> a2 = {{ 1., 2., 3.},
                         { 4., 5., 6.}};
```

(continues on next page)

(continued from previous page)

```
xt::xarray<double> res = a1 + a2;
// res = {{ 2., 4., 6. }, { 8., 10., 12. }};
```

`v` is not copied into `a1`, so if you change a value in `a1`, you're actually changing the corresponding value in `v`:

```
a1(0, 0) = 20.;
// now v is { 20., 2., 3., 4., 5., 6. }
```

1.6.2 Adapting C-style arrays

xtensor provides two ways for adapting C-style array; the first one does not take the ownership of the array:

```
#include <cstddef>
#include "xtensor/xadapt.hpp"

void compute(double* data, std::size_t size)
{
    std::vector<std::size_t> shape = { size };
    auto a = xt::adapt(data, size, xt::no_ownership(), shape);
    a = a + a; // does not modify the size
}

int main()
{
    std::size_t size = 2;
    double* data = new double[size];
    for (int i = 0; i < size; i++)
        data[i] = i;
    std::cout << data << std::endl;
    // prints e.g. 0x557a363b7c20
    compute(data, size);
    std::cout << data << std::endl;
    // prints e.g. 0x557a363b7c20 (same pointer)
    for (int i = 0; i < size; i++)
        std::cout << data[i] << " ";
    std::cout << std::endl;
    // prints 0 2 (data is still available here)
}
```

However if you replace `xt::no_ownership` with `xt::acquire_ownership`, the adaptor will take the ownership of the array, meaning it will be deleted when the adaptor is destroyed:

```
#include <cstddef>
#include "xtensor/xarray.hpp"
#include "xtensor/xadapt.hpp"

void compute(double*& data, std::size_t size)
{
    // data pointer can be changed, hence double*&
    std::vector<std::size_t> shape = { size };
    auto a = xt::adapt(data, size, xt::acquire_ownership(), shape);
    xt::xarray<double> b {1., 2.};
    b.reshape({2, 1});
    a = a * b; // size has changed, shape is now { 2, 2 }
```

(continues on next page)

(continued from previous page)

```
}

int main()
{
    std::size_t size = 2;
    double* data = new double[size];
    for (int i = 0; i < size; i++)
        data[i] = i;
    std::cout << data << std::endl;
    // prints e.g. 0x557a363b7c20
    compute(data, size);
    std::cout << data << std::endl;
    // prints e.g. 0x557a363b8220 (pointer has changed)
    for (int i = 0; i < size * size; i++)
        std::cout << data[i] << " ";
    std::cout << std::endl;
    // prints e.g. 4.65504e-310 1 0 2 (data has been deleted and is now corrupted)
}
```

To safely get the computed data out of the function, you could pass an additional output parameter to `compute` in which you copy the result before exiting the function. Or you can create the adaptor before calling `compute` and pass it to the function:

```
#include <cstddef>
#include "xtensor/xarray.hpp"
#include "xtensor/xadapt.hpp"

template <class A>
void compute(A& a)
{
    xt::xarray<double> b {1., 2.};
    b.reshape({2, 1});
    a = a * b; // size has changed, shape is now { 2, 2 }
}

int main()
{
    std::size_t size = 2;
    double* data = new double[size];
    for (int i = 0; i < size; i++)
        data[i] = i;
    std::vector<std::size_t> shape = { size };
    auto a = xt::adapt(data, size, xt::acquire_ownership(), shape);
    compute(a);
    for (int i = 0; i < size * size; i++)
        std::cout << data[i] << " ";
    std::cout << std::endl;
    // prints 0 1 0 2
}
```

1.7 Operators and functions

1.7.1 Arithmetic operators

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

- unary `operator+`
- unary `operator-`
- `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}}
```

1.7.2 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*.

1.7.3 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 }
```

1.7.4 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.

1.7.5 Casting

xtensor will implicitly promote and/or cast tensor expression elements as needed, which suffices for most use-cases. But explicit casting can be performed via `cast`, which performs an element-wise `static_cast`.

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

xt::xarray<int> a = { 3, 5, 7 };

auto res = a / 2;
// => res = { 1, 2, 3 }

auto res2 = xt::cast<double>(a) / 2;
// => res2 = { 1.5, 2.5, 3.5 }
```

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

1.7.7 Accumulators

Similar to reducers, *xtensor* provides accumulators which are used to implement cumulative functions such as `cumsum` or `cumprod`. Accumulators can currently only work on a single axis. Additionally, the accumulators are not lazy and do not return an xexpression, but rather an evaluated `xarray` or `xtensor`.

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

xt::xarray<double> a = xt::ones<double>({5, 8, 3});
xt::xarray<double> res = xt::cumsum(a, 1);
```

(continues on next page)

(continued from previous page)

```
// => res.shape() = {5, 8, 3};
// => res(0, 0, 0) = 1
// => res(0, 7, 0) = 8
```

You can also call the `accumumulate` generator with your own accumulating function. For example, the implementation of cumsum is as follows:

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

xt::xarray<double> arr = some_init_function({5, 5, 5});
xt::xarray<double> res = xt::accumulate([](double a, double b) { return a + b; },
                                        arr,
                                        1);
```

1.7.8 Evaluation strategy

Generally, *xtensor* implements a *lazy execution model*, but under certain circumstances, a *greedy* execution model with immediate execution can be favorable. For example, reusing (and recomputing) the same values of a reducer over and over again if you use them in a loop can cost a lot of CPU cycles. Additionally, *greedy* execution can benefit from SIMD acceleration over reduction axes and is faster when the entire result needs to be computed.

Therefore, xtensor allows to select an `evaluation_strategy`. Currently, two evaluation strategies are implemented: `evaluation_strategy::immediate` and `evaluation_strategy::lazy`. When `immediate` evaluation is selected, the return value is not an xexpression, but an in-memory datastructure such as a xarray or xtensor (depending on the input values).

Choosing an evaluation_strategy is straightforward. For reducers:

... code:

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

xt::xarray<double> a = xt::ones<double>({3, 2, 4, 6, 5});
auto res = xt::sum(a, {1, 3}, xt::evaluation_strategy::immediate());
// or select the default:
// auto res = xt::sum(a, {1, 3}, xt::evaluation_strategy::lazy());
```

Note: for accumulators, only the `immediate` evaluation strategy is currently implemented.

1.7.9 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"
```

(continues on next page)

(continued from previous page)

```
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 }
```

1.8 Views

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

1.8.1 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
```

(continues on next page)

(continued from previous page)

```
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
```

1.8.2 Dynamic views

While the `xt::view` is a compile-time static expression, xtensor also contains a dynamic view in `xstridedview.hpp`. The dynamic view and the slice vector allow to dynamically push_back slices, so when the dimension is unknown at compile time, the slice vector can be built dynamically at runtime. All the same slices as in xview can be used.

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

auto a = xt::xarray<int>::from_shape({3, 2, 3, 4, 5});

// note that `a` has to be passed into the slice_vector constructor
xt::slice_vector sv(a, xt::range(0, 1), xt::newaxis());
sv.push_back(1);
sv.push_back(xt::all());
// there is also a shorthand syntax: sv.append(1, xt::all());

auto v1 = xt::dynamic_view(a, sv);
// v1 has the same behavior as the static view
```

The dynamic_view is implemented on top of the strided_view, which is very efficient on contigous memory (e.g. xtensor or xarray) but less efficient on xexpressions.

1.8.3 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/xindex_view.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 }
```

(continues on next page)

(continued from previous page)

```
b += 100;
// => a = {{101, 5, 3}, {104, 105, 6}}
```

1.8.4 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/xindex_view.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}}
```

1.8.5 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/xindex_view.hpp"

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

1.8.6 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)
```

1.8.7 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}};
```

1.9 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.

1.9.1 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.

1.9.2 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

1.9.3 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.

1.9.4 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.

1.9.5 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.

1.10 Missing values

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

1.10.1 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.

1.10.2 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}}
```

1.10.3 Optional assemblies

The classes `xoptional_assembly` and `xoptional_assembly_adaptor` provide containers and adaptors holding missing values that are optimized for element-wise operations. Contrary to `xtensor_optional` and `xarray_optional`, the optional assemblies hold two expressions, one holding the values, the other holding the mask for the missing values. The difference between `xoptional_assembly` and `xoptional_assembly_adaptor` is that the first one is the owner of the two expressions while the last one holds a reference on at least one of the two expressions.

```
xarray<double> v
    {{ 1.0, 2.0 },
     { 3.0, 4.0 }};

xarray<bool> hv
    {{ true, true  },
     { true, false }};

xoptional_assembly<xarray<double>, xarray<bool>>
assembly(v, vh);
std::cout << assembly << std::endl;
```

outputs:

```
{{ 1,  2 },
 { 3, N/A}}
```

1.10.4 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}}
```

1.11 Expressions and semantic

1.11.1 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 xt::*xexpression*`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 xt::*xexpression*`derived_cast() const`

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

1.11.2 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::xcontainer_semantic< D >*, *xt::xview_semantic< D >*

Computed assignement

template <class E>
auto xt::*xsemantic_base*`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 xt::*xsemantic_base*`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 xt::*xsemantic_base*`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 xt::*xsemantic_base*`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 xt::*xsemantic_base*`operator%=`(**const** E &*e*)
Computes the remainder of `*this` after division by the scalar `e`.

Return a reference to `*this`.

Parameters

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

template <class E>
auto xt::*xsemantic_base*`operator&=`(**const** E &*e*)
Computes the bitwise and of `*this` and the scalar `e` and assigns it to `*this`.

Return a reference to `*this`.

Parameters

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

template <class E>
auto xt::*xsemantic_base*`operator|=`(**const** E &*e*)
Computes the bitwise or of `*this` and the scalar `e` and assigns it to `*this`.

Return a reference to `*this`.

Parameters

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

template <class E>
auto xt::*xsemantic_base*`operator^=`(**const** E &*e*)
Computes the bitwise xor of `*this` and the scalar `e` and assigns it to `*this`.

Return a reference to `*this`.

Parameters

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

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

Return a reference to `*this`.

Parameters

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

`template <class` E>
auto xt::*xsemantic_base*`operator%=`(`const` *xexpression*<E> &*e*)
Computes the remainder of `*this` after division by the xexpression `e`.

Return a reference to `*this`.

Parameters

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

`template <class` E>
auto xt::*xsemantic_base*`operator&=`(`const` *xexpression*<E> &*e*)
Computes the bitwise and of `*this` and the xexpression `e` and assigns it to `*this`.

Return a reference to `*this`.

Parameters

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

`template <class` E>
auto xt::*xsemantic_base*`operator|=`(`const` *xexpression*<E> &*e*)
Computes the bitwise or of `*this` and the xexpression `e` and assigns it to `*this`.

Return a reference to `*this`.

Parameters

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

`template <class` E>
auto xt::*xsemantic_base*`operator^=`(`const` *xexpression*<E> &*e*)
Computes the bitwise xor of `*this` and the xexpression `e` and assigns it to `*this`.

Return a reference to `*this`.

Parameters

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

Assign functions

`template <class` E>
auto xt::*xsemantic_base*`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 xt::*xsemantic_base*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 xt::*xsemantic_base*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 xt::*xsemantic_base*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 xt::*xsemantic_base*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.

template <**class** E>
auto xt::*xsemantic_base*modulus_assign(**const** *xexpression*<E> &*e*)

Computes the remainder of `*this` after division 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.

1.11.3 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 xt::*xcontainer_semantic*::**assign_temporary**(temporary_type &&*tmp*)

Assigns the temporary `tmp` to `*this`.

Return a reference to `*this`.

Parameters

- `tmp`: the temporary to assign.

1.11.4 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 xt::*xview_semantic*::**assign_temporary**(temporary_type &&*tmp*)

Assigns the temporary `tmp` to `*this`.

Return a reference to `*this`.

Parameters

- `tmp`: the temporary to assign.

1.11.5 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

1.12 Containers and views

1.12.1 layout

Defined in `xtensor/xlayout.hpp`

enum xt::**layout_type**

layout_type enum for xcontainer based xexpressions

Values:

xt**dynamic** = 0x00

dynamic layout_type: you can resize to row major, column major, or use custom strides

xt**any** = 0xFF

layout_type compatible with all others

xt**row_major** = 0x01

row major layout_type

xt**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.

1.12.2 xcontainer

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

Base class for dense multidimensional containers.

Base class for dense multidimensional optional assemblies.

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.

The *xoptional_assembly_base* class defines the interface for dense multidimensional optional assembly classes. Optional assembly classes hold optional values and are optimized for tensor operations. *xoptional_assembly_base* 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.

Template Parameters

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

Inherits from *xt::xiterable< D >*

Subclassed by *xt::xstrided_container< D >*

Size and shape

auto xt::*xcontainer*`size() const`
Returns the number of element in the container.

constexpr auto xt::*xcontainer*`dimension() const`
Returns the number of dimensions of the container.

constexpr auto xt::*xcontainer*`shape() const`
Returns the shape of the container.

constexpr auto xt::*xcontainer*`strides() const`
Returns the strides of the container.

constexpr auto xt::*xcontainer*`backstrides() const`
Returns the backstrides of the container.

Data

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

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

auto xt::*xcontainer*`raw_data()`
Returns the offset to the first element in the container.

auto xt::*xcontainer*`raw_data_offset()` `const`
Returns the offset to the first element in the container.

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

Parameters

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

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 xt::*xcontainer*`at`(Args... *args*) `const`
Returns a constant reference to the element at the specified position in the container, after dimension and bounds checking.

Parameters

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

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` S>
auto xt::*xcontainer*`operator[]`(`const` S &*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.

template <class S>
auto xt::*xcontainer***operator[]**(**const** S &*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.

template <class It>
auto xt::*xcontainer***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 xt::*xcontainer***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 xt::*xcontainer***broadcast_shape**(S &*shape*, bool *reuse_cache* = false) **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 xt::*xcontainer***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

1.12.3 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 xt::*xstrided_container*__resize__(S &&*shape*, bool *force* = false)

resizes 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 xt::*xstrided_container*__resize__(S &&*shape*, *layout_type* *l*)

resizes the container.

Parameters

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

template <class S = shape_type>
void xt::*xstrided_container*__resize__(S &&*shape*, **const** strides_type &*strides*)

Resizes the container.

Parameters

- `shape`: the new shape
- `strides`: the new strides

template <class S = shape_type>
void xt::*xstrided_container*__reshape__(S &&*shape*, *layout_type* *layout* = base_type::static_layout)

Reshapes the container and keeps old elements.

Parameters

- `shape`: the new shape (has to have same number of elements as the original container)
- `layout`: the layout to compute the strides (defaults to static layout of the container, or for a container with dynamic layout to DEFAULT_LAYOUT)

layout_type xt::*xstrided_container*__layout__() **const**

Return the layout_type of the container.

Return layout_type of the container

1.12.4 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 xt::*xconst_iterable*`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 xt::*xconst_iterable*`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 xt::*xconst_iterable*`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 xt::*xconst_iterable*`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 xt::*xconst_iterable*`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 xt::*xconst_iterable*`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 xt::*xconst_iterable*`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 xt::*xconst_iterable*`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 xt::*xconst_iterable*`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 xt::*xconst_iterable*`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 xt::*xconst_iterable*`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 xt::*xconst_iterable*`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 xt::*xconst_iterable*`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 xt::*xconst_iterable*`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 xt::*xconst_iterable*`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 xt::*xconst_iterable*`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 >*, *xt::xoptional_assembly_base< D >*

Iterators

template <*layout_type L*>
auto xt::*xiterable*`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 xt::*xiterable*`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 xt::*xiterable*__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 xt::*xiterable*__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 xt::*xiterable*__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 xt::*xiterable*__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 xt::*xiterable*__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 xt::*xiterable*__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`.

1.12.5 xarray

Defined in `xtensor/xarray.hpp`
template <class *EC*, *layout_type L*, class *SC*, class *Tag*>
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.
- `Tag`: The expression tag.

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

Constructors

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

xt::*xarray_container*`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*

xt::*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*

xt::*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*

xt::*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

xt::*xarray_container*`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*

xt::*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

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

Parameters

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

xt::*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*

xt::*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*

xt::*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*

xt::*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>
xt::*xarray_container*`xarray_container`(`const` *xexpression*<E> &*e*)
The extended copy constructor.

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

Public Functions

template <class S>
xarray_container<EC, L, SC, Tag> xt::*xarray_container*`from_shape`(S &&*s*)
Allocates and returns an *xarray_container* with the specified shape.

Parameters

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

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.

typedef xt::**xarray_optional**
Alias template on *xarray_container* for handling missing values.

Template Parameters

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

1.12.6 xarray_adaptor

Defined in `xtensor/xarray.hpp`
template <class *EC*, *layout_type* *L*, class *SC*, class *Tag*>
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 closure for the container type to adapt.

- `L`: The layout_type of the adaptor.
- `SC`: The type of the containers holding the shape and the strides.
- `Tag`: The expression tag.

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

Constructors

xt::*xarray_adaptor*`xarray_adaptor`(container_type &&*data*)
Constructs an *xarray_adaptor* of the given stl-like container.

Parameters

- `data`: the container to adapt

xt::*xarray_adaptor*`xarray_adaptor`(`const` container_type &*data*)
Constructs an *xarray_adaptor* of the given stl-like container.

Parameters

- `data`: the container to adapt

`template <class` D>
xt::*xarray_adaptor*`xarray_adaptor`(D &&*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*

`template <class` D>
xt::*xarray_adaptor*`xarray_adaptor`(D &&*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 xt::*xarray_adaptor*`operator=`(`const` *xexpression*<E> &*e*)
The extended assignment operator.

1.12.7 adapt (xarray_adaptor)

Defined in `xtensor/xadapt.hpp`

template <**class** C, **class** SC, layout_type *L* = layout_type::row_major, **typename** std::enable_if_t<!detail::is_array< std::dec
xarray_adaptor<xtl::closure_type_t<C>, L, std::decay_t<SC>> xt::**adapt**(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, **class** SS, **typename** std::enable_if_t<!detail::is_array< std::decay_t< SC >>::value, int > = 0
xarray_adaptor<xtl::closure_type_t<C>, *layout_type*::dynamic, std::decay_t<SC>> xt::**adapt**(C &&*container*, SC &&*shape*, SS &&*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* = layout_type::row_major, **class** A = std::allocator<std::remove_point
xarray_adaptor<xbuffer_adaptor<xtl::closure_type_t<P>, O, A>, L, SC> xt::**adapt**(P &&*pointer*, **typename** A::size_type *size*, O *ownership*, **const** SC &*shape*, *layout_type* *l* = L, **const** A &*alloc* = A())

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 `acquire_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** SS, **class** A = std::allocator<std::remove_pointer_t<std::remove_reference_t

xarray_adaptor<xbuffer_adaptor<xtl::closure_type_t<P>, O, A>, *layout_type*::dynamic, std::decay_t<SC>> xt::**adapt**(P &&*pointer*, **typename** A::size_type *size*, O *ownership*, SC &&*shape*, SS &&*strides*, **const** A &*alloc* = A())

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 `acquire_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

1.12.8 xtensor

Defined in `xtensor/xtensor.hpp`

template <class *EC*, size_t *N*, *layout_type* *L*, class *Tag*>
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.
- `Tag`: The expression tag.

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

Constructors

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

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

xt::*xtensor_container*`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*

xt::*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*

xt::*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*

xt::*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

xt::*xtensor_container*`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>
xt::*xtensor_container***xtensor_container**(**const** *xexpression*<E> &*e*)
The extended copy constructor.

template <class E>
auto xt::*xtensor_container***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.

typedef xt::**xtensor_optional**
Alias template on *xtensor_container* for handling missing values.

Template Parameters

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

1.12.9 xtensor_adaptor

Defined in `xtensor/xtensor.hpp`
template <class *EC*, std::size_t *N*, *layout_type* *L*, class *Tag*>

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 closure for the container type to adapt.
- `N`: The dimension of the adaptor.
- `L`: The layout_type of the adaptor.
- `Tag`: The expression tag.

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

Constructors

xt::*xtensor_adaptor***xtensor_adaptor**(container_type &&*data*)

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

Parameters

- `data`: the container to adapt

xt::*xtensor_adaptor***xtensor_adaptor**(**const** container_type &*data*)

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

Parameters

- `data`: the container to adapt

template <**class** D>
xt::*xtensor_adaptor***xtensor_adaptor**(D &&*data*, **const** shape_type &*shape*, *layout_type* *l* = L)

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*

template <**class** D>
xt::*xtensor_adaptor***xtensor_adaptor**(D &&*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 xt::*xtensor_adaptor***operator=**(**const** *xexpression*<E> &*e*)
The extended assignment operator.

1.12.10 adapt (xtensor_adaptor)

Defined in `xtensor/xadapt.hpp`
template <**class** C, layout_type *L* = layout_type::row_major>
xtensor_adaptor<C, 1, L> xt**::adapt**(C &&*container*, *layout_type* *l* = L)
Constructs a 1D *xtensor_adaptor* of the given stl-like container, with the specified layout_type.

Parameters

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

template <**class** C, **class** SC, layout_type *L* = layout_type::row_major, **typename** std::enable_if_t<!detail::is_array< std::dec
xarray_adaptor<xtl::closure_type_t<C>, L, std::decay_t<SC>> xt**::adapt**(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, **class** SS, **typename** std::enable_if_t<!detail::is_array< std::decay_t< SC >>::value, int > = 0
xarray_adaptor<xtl::closure_type_t<C>, *layout_type*::dynamic, std::decay_t<SC>> xt**::adapt**(C &&*container*, SC &&*shape*, SS &&*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, layout_type *L* = layout_type::row_major, **class** A = std::allocator<std::remove_pointer_t<std::re
xtensor_adaptor<xbuffer_adaptor<xtl::closure_type_t<P>, O, A>, 1, L> xt**::adapt**(P &&*pointer*, **typename** A::size_type *size*, O *ownership*, *layout_type* *l* = L, **const** A &*alloc* = A())
Constructs a 1D *xtensor_adaptor* of the given dynamically allocated C array, with the specified 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 `acquire_ownership()`
- `l`: the layout_type of the *xtensor_adaptor*
- `alloc`: the allocator used for allocating / deallocating the dynamic array

template <**class** P, **class** O, **class** SC, layout_type *L* = layout_type::row_major, **class** A = std::allocator<std::remove_point
xarray_adaptor<xbuffer_adaptor<xtl::closure_type_t<P>, O, A>, L, SC> xt::**adapt**(P &&*pointer*, **typename** A::size_type *size*, O *ownership*, **const** SC &*shape*, *layout_type* *l* = L, **const** A &*alloc* = A())

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 `acquire_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** SS, **class** A = std::allocator<std::remove_pointer_t<std::remove_reference_t
xarray_adaptor<xbuffer_adaptor<xtl::closure_type_t<P>, O, A>, *layout_type*::dynamic, std::decay_t<SC>> xt::**adapt**(P &&*pointer*, **typename** A::size_type *size*, O *ownership*, SC &&*shape*, SS &&*strides*, **const** A &*alloc* = A())

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 `acquire_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

1.12.11 xoptional_assembly_base

Defined in `xtensor/xoptional_assembly_base.hpp`
template <class *D*>
class xt::**xoptional_assembly_base**
Inherits from *xt::xiterable< D >*

Size and shape

auto xt::*xoptional_assembly_base*`size()` **const**
Returns the number of element in the optional assembly.

auto **constexpr** xt::*xoptional_assembly_base*`dimension()` **const**
Returns the number of dimensions of the optional assembly.

auto xt::*xoptional_assembly_base*`shape()` **const**
Returns the shape of the optional assembly.

auto xt::*xoptional_assembly_base*`strides()` **const**
Returns the strides of the optional assembly.

auto xt::*xoptional_assembly_base*`backstrides()` **const**
Returns the backstrides of the optional assembly.

Data

template <class... *Args*>
auto xt::*xoptional_assembly_base*`operator()`(Args... *args*)
Returns a reference to the element at the specified position in the optional assembly.

Parameters

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

template <class... *Args*>
auto xt::*xoptional_assembly_base*`operator()`(Args... *args*) **const**
Returns a constant reference to the element at the specified position in the optional assembly.

Parameters

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

template <class... *Args*>
auto xt::*xoptional_assembly_base*__at__(Args... *args*)

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

Parameters

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

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 xt::*xoptional_assembly_base*__at__(Args... *args*) **const**

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

Parameters

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

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 S>
auto xt::*xoptional_assembly_base*__operator[]__(**const** S &*index*)

Returns a reference to the element at the specified position in the optional assembly.

Parameters

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

template <class S>
auto xt::*xoptional_assembly_base*__operator[]__(**const** S &*index*) **const**

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

Parameters

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

template <class It>
auto xt::*xoptional_assembly_base*__element__(It *first*, It *last*)

Returns a reference to the element at the specified position in the optional assembly.

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 optional assembly.

template <class It**>**
auto xt::*xoptional_assembly_base*`element`(It *first*, It *last*) **const**
Returns a constant reference to the element at the specified position in the optional assembly.

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 optional assembly.

Broadcasting

template <class S**>**
bool xt::*xoptional_assembly_base*`broadcast_shape`(S &*shape*, bool *reuse_cache* = false) **const**
Broadcast the shape of the optional assembly to the specified parameter.

Return a boolean indicating whether the broadcasting is trivial

Parameters

- `shape`: the result shape

template <class S**>**
bool xt::*xoptional_assembly_base*`is_trivial_broadcast`(**const** S &*strides*) **const**
Compares the specified strides with those of the optional assembly to see whether the broadcasting is trivial.

Return a boolean indicating whether the broadcasting is trivial

Public Functions

template <class S = shape_type**>**
void xt::*xoptional_assembly_base*`resize`(**const** S &*shape*, bool *force* = false)
Resizes the optional assembly.

Parameters

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

template <class S = shape_type**>**
void xt::*xoptional_assembly_base*`resize`(**const** S &*shape*, *layout_type* *l*)
Resizes the optional assembly.

Parameters

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

template <class S = shape_type**>**
void xt::*xoptional_assembly_base*`resize`(**const** S &*shape*, **const** strides_type &*strides*)
Resizes the optional assembly.

Parameters

- `shape`: the new shape
- `strides`: the new strides

template <class S = shape_type>
void xt::*xoptional_assembly_base*`reshape`(**const** S &*shape*, *layout_type* *layout* = static_layout)
Reshapes the optional assembly.

Parameters

- `shape`: the new shape
- `layout`: the new layout

layout_type xt::*xoptional_assembly_base*`layout`() **const**
Return the layout_type of the container.

Return layout_type of the container

auto xt::*xoptional_assembly_base*`value`()
Return an expression for the values of the optional assembly.

auto xt::*xoptional_assembly_base*`value`() **const**
Return a constant expression for the values of the optional assembly.

auto xt::*xoptional_assembly_base*`has_value`()
Return an expression for the missing mask of the optional assembly.

auto xt::*xoptional_assembly_base*`has_value`() **const**
Return a constant expression for the missing mask of the optional assembly.

1.12.12 xoptional_assembly

Defined in `xtensor/xoptional_assembly.hpp`
template <class VE, **class** FE>
class xt::**xoptional_assembly**
Dense multidimensional container holding optional values, optimized for tensor operations.

The *xoptional_assembly* class implements a dense multidimensional container hoding optional values. This container is optimized of tensor operations: contrary to xarray_optional, *xoptional_assembly* holds two separated expressions, one for the values, the other for the missing mask.

Template Parameters

- `VE`: The type of expression holding the values.
- `FE`: The type of expression holding the missing mask.

Inherits from *xt::xoptional_assembly_base< xoptional_assembly< VE, FE > >*, *xt::xcontainer_semantic< xoptional_assembly< VE, FE > >*

Constructors

xt::*xoptional_assembly*`xoptional_assembly()`
Allocates an uninitialized *xoptional_assembly* that holds 0 element.

xt::*xoptional_assembly*`xoptional_assembly(const` shape_type *&shape*, *layout_type l* = base_type::static_layout)
Allocates an uninitialized *xoptional_assembly* with the specified shape and layout_type.

Parameters

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

xt::*xoptional_assembly*`xoptional_assembly(const` shape_type *&shape*, `const` value_type *&value*, *layout_type l* = base_type::static_layout)
Allocates an *xoptional_assembly* with the specified shape and layout_type.

Elements are initialized to the specified value.

Parameters

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

xt::*xoptional_assembly*`xoptional_assembly(const` shape_type *&shape*, `const` strides_type *&strides*)
Allocates an uninitialized *xoptional_assembly* with the specified shape and strides.

Parameters

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

xt::*xoptional_assembly*`xoptional_assembly(const` shape_type *&shape*, `const` strides_type *&strides*, `const` value_type *&value*)
Allocates an uninitialized *xoptional_assembly* with the specified shape and strides.

Elements are initialized to the specified value.

Parameters

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

xt::*xoptional_assembly*`xoptional_assembly(const` VE *&ve*)
Allocates an *xoptional_assembly* from the specified value expression.

The flag expression is initialized as if no value is missing.

Parameters

- `ve`: the expression holding the values

xt::*xoptional_assembly*`xoptional_assembly`(VE &&*ve*)
Allocates an *xoptional_assembly* from the specified value expression.

The flag expression is initialized as if no value is missing. The value expression is moved inside the *xoptional_assembly* and is therefore not available after the *xoptional_assembly* has been constructed.

Parameters

- `ve`: the expression holding the values

`template <class` OVE, `class` OFE, typename = concept_check <is_xexpression<OVE>::*value* && is_xexpression<OFE>
xt::*xoptional_assembly*`xoptional_assembly`(OVE &&*ove*, OFE &&*ofe*)
Allocates an *xoptional_assembly* from the specified value expression and missing mask expression.

Parameters

- `ove`: the expression holding the values
- `ofe`: the expression holding the missing mask

xt::*xoptional_assembly*`xoptional_assembly`(`const` value_type &*value*)
Allocates an *xoptional_assembly* that holds a single element initialized to the specified value.

Parameters

- `value`: the value of the element

Constructors from initializer list

xt::*xoptional_assembly*`xoptional_assembly`(nested_initializer_list_t<value_type, 1> *t*)
Allocates a one-dimensional *xoptional_assembly*.

Parameters

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

xt::*xoptional_assembly*`xoptional_assembly`(nested_initializer_list_t<value_type, 2> *t*)
Allocates a two-dimensional *xoptional_assembly*.

Parameters

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

xt::*xoptional_assembly*`xoptional_assembly`(nested_initializer_list_t<value_type, 3> *t*)
Allocates a three-dimensional *xoptional_assembly*.

Parameters

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

xt::*xoptional_assembly*`xoptional_assembly`(nested_initializer_list_t<value_type, 4> *t*)
Allocates a four-dimensional *xoptional_assembly*.

Parameters

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

xt::*xoptional_assembly*`xoptional_assembly`(nested_initializer_list_t<value_type, 5> *t*)
Allocates a five-dimensional *xoptional_assembly*.

Parameters

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

Extended copy semantic

`template <class` E>
xt::*xoptional_assembly*`xoptional_assembly`(`const` *xexpression*<E> &*e*)
The extended copy constructor.

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

Public Functions

`template <class` S>
xoptional_assembly<VE, FE> xt::*xoptional_assembly*`from_shape`(S &&*s*)
Allocates and returns an *xoptional_assembly* with the specified shape.

Parameters

- `s`: the shape of the *xoptional_assembly*

1.12.13 xoptional_assembly_adaptor

Defined in `xtensor/xoptional_assembly.hpp`
`template <class` VEC, `class` FEC>
`class` xt::`xoptional_assembly_adaptor`
Dense multidimensional adaptor holding optional values, optimized for tensor operations.

The *xoptional_assembly_adaptor* class implements a dense multidimensional adaptor hoding optional values. It is used to provide an optional expression semantic to two tensor expressions, one holding the value, the hother holding the missing mask.

Template Parameters

- `VEC`: The closure for the type of expression holding the values.
- `FE`: The closure for the type of expression holding the missing mask.

Inherits from *xt::xoptional_assembly_base< xoptional_assembly_adaptor< VEC, FEC > >*, *xt::xcontainer_semantic< xoptional_assembly_adaptor< VEC, FEC > >*

Constructors

`template <class` OVE, `class` OFE>
xt::*xoptional_assembly_adaptor*`xoptional_assembly_adaptor`(OVE &&*ve*, OFE &&*fe*)
Constructs an *xoptional_assembly_adaptor* of the given value and missing mask expressions.

Parameters

- `ve`: the expression holding the values
- `fe`: the expression holding the missing mask

Extended copy semantic

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

1.12.14 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 xt::*xview***operator=**(**const** *xexpression*<E> &*e*)
The extended assignment operator.

Constructor

template <class CTA, **class** FSL, class... *SL*>
xt::*xview***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 xt::*xview*`dimension() const`
Returns the number of dimensions of the view.

auto xt::*xview*`size() const`
Returns the size of the expression.

auto xt::*xview*`shape() const`
Returns the shape of the view.

auto xt::*xview*`slices() const`
Returns the slices of the view.

layout_type xt::*xview*`layout() const`
Returns the slices of the view.

Data

`template` <class... *Args*>
auto xt::*xview*`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 xt::*xview*`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 xt::*xview*`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 xt::*xview*`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 xt::*xview***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 xt::*xview***strides**() **const**

Return the strides for the underlying container of the view.

template <class T>
auto xt::*xview***raw_data**() **const**

Return the pointer to the underlying buffer.

template <class T>
auto xt::*xview***raw_data_offset**() **const**

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

Broadcasting

template <class ST>
bool xt::*xview***broadcast_shape**(ST &*shape*, bool *reuse_cache* = false) **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 xt::*xview***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`
template <class A, **class** B>
auto xt**::range**(A *min_val*, B *max_val*)

template <**class** A, **class** B, **class** C>
auto xt::**range**(A *min_val*, B *max_val*, C *step*)

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*

1.12.15 xstrided_view

Defined in `xtensor/xstrided_view.hpp`
template <class *CT*, class *S*, class *CD*>
class xt::**xstrided_view**
View of an xexpression using strides.

The *xstrided_view* class implements a view utilizing an offset and strides into a multidimensional xcontainer. The xstridedview is currently used to implement `transpose`.

See *strided_view*, *transpose*

Template Parameters

- `CT`: the closure type of the *xexpression* type underlying this view
- `CD`: the closure type of the underlying data container

Inherits from *xt::xview_semantic< xstrided_view< CT, S, CD > >*, *xt::xiterable< xstrided_view< CT, S, CD > >*

Extended copy semantic

template <**class** E>
auto xt::*xstrided_view*^{**operator=**}(**const** *xexpression*<E> &*e*)
The extended assignment operator.

Constructor

xt::*xstrided_view*^{**xstrided_view**}(CT *e*, S &&*shape*, S &&*strides*, std::size_t *offset*, *layout_type* *layout*)
Constructs an *xstrided_view*.

Parameters

- `e`: the underlying xexpression for this view
- `shape`: the shape of the view
- `strides`: the strides of the view
- `offset`: the offset of the first element in the underlying container
- `layout`: the layout of the view

Size and shape

auto xt::*xstrided_view*`size() const`
Returns the size of the *xstrided_view*.

auto xt::*xstrided_view*`dimension() const`
Returns the number of dimensions of the *xstrided_view*.

auto xt::*xstrided_view*`shape() const`
Returns the shape of the *xstrided_view*.

Data

`template` <class... *Args*>
auto xt::*xstrided_view*`operator()`(Args... *args*) `const`
Returns the element at the specified position in the *xstrided_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 xt::*xstrided_view*`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 xt::*xstrided_view*`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 xt::*xstrided_view*`element`(It *first*, It *last*)
Returns a reference to the element at the specified position in the *xstrided_view*.

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 the number of dimensions of the container..

Broadcasting

template <class O>
bool xt::*xstrided_view*__broadcast_shape__(O &*shape*, bool *reuse_cache* = false) **const**
Broadcast the shape of the *xstrided_view* to the specified parameter.

Return a boolean indicating whether the broadcasting is trivial

Parameters

- `shape`: the result shape

template <class O>
bool xt::*xstrided_view*__is_trivial_broadcast__(**const** O &*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** I>
auto xt::__strided_view__(E &&*e*, I &&*shape*, I &&*strides*, std::size_t *offset* = 0, *layout_type* *layout* = *layout_type*::dynamic)
Construct a strided view from an xexpression, shape, strides and offset.

Return the view

Parameters

- `e`: xexpression
- `shape`: the shape of the view
- `strides`: the new strides of the view
- `offset`: the offset of the first element in the underlying container
- `layout`: the new layout of the expression

Template Parameters

- `E`: type of xexpression
- `I`: shape and strides type

template <class E>
auto xt::__transpose__(E &&*e*)
Returns a transpose view by reversing the dimensions of xexpression e.

Parameters

- `e`: the input expression

template <class E, **class** S, **class** Tag = check_policy::none>
auto xt::__transpose__(E &&*e*, S &&*permutation*, Tag *check_policy* = Tag())
Returns a transpose view by permuting the xexpression e with `permutation`.

Parameters

- `e`: the input expression
- `permutation`: the sequence containing permutation
- `check_policy`: the check level (check_policy::full() or check_policy::none())

Template Parameters

- `Tag`: selects the level of error checking on permutation vector defaults to check_policy::none.

class xt::**slice_vector**

The *slice_vector* is used as part of the dynamic_view interface.

It offers an interface to dynamically push_back or append slices to dynamically create a view into a xexpression.

This data structure is necessary as the slices of xview are compile-time static. If you do not know how many dimensions your expression has at compile-time, using dynamic_view can be useful.

Internally, the dynamic_view is implemented using the strided_view adapter, which is very fast for data structures in contigous memory (such as xarrays and xtensors).

Usage example:

```
xt::xarray<double> a = {{1, 2, 3}, {4, 5, 6}};
xt::slice_vector sv(a, xt::range(0, 1));
sv.push_back(xt::all());
```

Inherits from std::vector< std::array< long int, 3 > >

Public Functions

template <**class** E, class... *Args*>
xt::*slice_vector***slice_vector**(**const** *xexpression*<E> &*e*, Args... *args*)

Constructor for slice vector.

Parameters

- `e`: xexpression
- `args`: xslices, integer, all or newaxis

template <class... *Args*>
xt::*slice_vector***slice_vector**(**const** dynamic_shape<std::size_t> &*shape*, Args... *args*)

Alternative constructor taking a shape.

Parameters

- `shape`: the shape

template <**class** T, class... *Args*>
void xt::*slice_vector***append**(**const** T &*s*, Args... *args*)

Convenience function to append one or multiple slices to the *slice_vector*.

```
xt::slice_vector sv(a, xt::range(0, 1));
sv.append(xt::all(), xt::newaxis(), 0, 12));
```

template <**class** T>
void xt::*slice_vector***push_back**(**const** xslice<T> &*s*)

Push back a single slice, integer, all or newaxis.

Parameters

- `s`: slice

template <class E, **class** S>
auto xt::**dynamic_view**(E &&*e*, S &&*slices*)

Function to create a dynamic view from a xexpression and a *slice_vector*.

```
xt::xarray<double> a = {{1, 2, 3}, {4, 5, 6}};
xt::slice_vector sv(a, xt::range(0, 1));
sv.push_back(xt::range(0, 3, 2));
auto v = xt::dynamic_view(a, sv);
// ==> {{1, 3}}
```

Return initialized strided_view according to slices

Parameters

- `e`: xexpression
- `slices`: the slice vector

1.12.16 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>
xt::*xbroadcast***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 xt::*xbroadcast*`size() const`
Returns the size of the expression.

auto xt::*xbroadcast*`dimension() const`
Returns the number of dimensions of the expression.

auto xt::*xbroadcast*`shape() const`
Returns the shape of the expression.

layout_type xt::*xbroadcast*`layout() const`
Returns the layout_type of the expression.

Data

`template` <class... *Args*>
auto xt::*xbroadcast*`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 xt::*xbroadcast*`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` S>
auto xt::*xbroadcast*`operator[](const` S &*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` It>
auto xt::*xbroadcast*`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 xt::*xbroadcast***broadcast_shape**(S &*shape*, bool *reuse_cache* = false) **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 xt::*xbroadcast***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.

1.12.17 xindex_view

Defined in `xtensor/xindex_view.hpp`
template <class *CT*, class *I*>
class xt**::xindex_view**
View of an xexpression from vector of indices.

The *xindex_view* class implements a flat (1D) view into a multidimensional xexpression yielding the values at the indices of the index array. *xindex_view* 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< xindex_view< CT, I > >*, *xt::xiterable< xindex_view< CT, I > >*

Extended copy semantic

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

Constructor

template <**class** I2>
xt::*xindex_view*`xindex_view`(CT *e*, I2 &&*indices*)
Constructs an *xindex_view*, 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 xt::*xindex_view*`size`() **const**
Returns the size of the *xindex_view*.

auto xt::*xindex_view*`dimension`() **const**
Returns the number of dimensions of the *xindex_view*.

auto xt::*xindex_view*`shape`() **const**
Returns the shape of the *xindex_view*.

Data

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

Parameters

- `idx`: the position in the view

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

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 xt::*xindex_view*<**broadcast_shape**(O &*shape*, bool *reuse_cache* = false) **const**
Broadcast the shape of the *xindex_view* to the specified parameter.

Return a boolean indicating whether the broadcasting is trivial

Parameters

- `shape`: the result shape

template <**class** O>
bool xt::*xindex_view*<**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 xt::*xfiltration*<**operator=**(**const** E &*e*)
Assigns the scalar `e` to `*this`.

Return a reference to \ *this.

Parameters

- `e`: the scalar to assign.

Constructor

xt::*xfiltration*<**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 xt::*xfiltration***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 xt::*xfiltration***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 xt::*xfiltration***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 xt::*xfiltration***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 xt::*xfiltration***operator%=**(**const** E &*e*)
Computes the remainder of `*this` after division 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*

1.12.18 xfunctor_view

Defined in `xtensor/xfunctor_view.hpp`
template <class *F*, class *CT*>

class xt::**xfunctor_view**

View of an xexpression .

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

xfunctor_view 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< xfunctor_view< F, CT > >*

Constructors

xt::*xfunctor_view*__xfunctor_view__(CT *e*)

Constructs an *xfunctor_view* expression wrappering the specified *xexpression*.

Parameters

- `e`: the underlying expression

template <class Func, **class** E>
xt::*xfunctor_view*__xfunctor_view__(Func &&*func*, E &&*e*)

Constructs an *xfunctor_view* 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 xt::*xfunctor_view*__operator=__(**const** *xexpression*<E> &*e*)

The extended assignment operator.

Size and shape

auto xt::*xfunctor_view*__size__() **const**

Returns the size of the expression.

auto xt::*xfunctor_view*__dimension__() **const**

Returns the number of dimensions of the expression.

auto xt::*xfunctor_view*__shape__() **const**

Returns the shape of the expression.

layout_type xt::*xfunctor_view*`layout() const`
Returns the layout_type of the expression.

Data

`template <class... Args>`
auto xt::*xfunctor_view*`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 xt::*xfunctor_view*`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 S>`
auto xt::*xfunctor_view*`operator[](const S &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.

`template <class IT>`
auto xt::*xfunctor_view*`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 xt::*xfunctor_view*`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 xt::*xfunctor_view*__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** S>
auto xt::*xfunctor_view*__operator[]__(**const** S &*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** IT>
auto xt::*xfunctor_view*__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 xt::*xfunctor_view*__broadcast_shape__(S &*shape*, bool *reuse_cache* = false) **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
- `reuse_cache`: boolean for reusing a previously computed shape

template <**class** S>
bool xt::*xfunctor_view*__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 = DEFAULT_LAYOUT>
auto xt::*xfunctor_view*__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 = DEFAULT_LAYOUT>
auto xt::*xfunctor_view*__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 = DEFAULT_LAYOUT>
auto xt::*xfunctor_view*__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 = DEFAULT_LAYOUT>
auto xt::*xfunctor_view*__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 = DEFAULT_LAYOUT>
auto xt::*xfunctor_view*__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 = DEFAULT_LAYOUT>
auto xt::*xfunctor_view*__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 xt::*xfunctor_view*__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 xt::*xfunctor_view*__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 xt::*xfunctor_view*__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 xt::*xfunctor_view*__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 xt::*xfunctor_view*__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 xt::*xfunctor_view*__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* = DEFAULT_LAYOUT>
auto xt::*xfunctor_view*__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* = DEFAULT_LAYOUT>
auto xt::*xfunctor_view*__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* = DEFAULT_LAYOUT>
auto xt::*xfunctor_view*__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* = DEFAULT_LAYOUT>
auto xt::*xfunctor_view*__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* = DEFAULT_LAYOUT>
auto xt::*xfunctor_view*__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 = DEFAULT_LAYOUT>
auto xt::*xfunctor_view*__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 xt::*xfunctor_view*__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 xt::*xfunctor_view*__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 xt::*xfunctor_view*__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 xt::*xfunctor_view*__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 xt::*xfunctor_view*__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 xt::*xfunctor_view*__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*

1.13 Functions and generators

1.13.1 xfunction_base

Defined in `xtensor/xfunction.hpp`
template <class *F*, class *R*, class... *CT*>
class xt::**xfunction_base**

Base class for multidimensional function operating on xexpression.

The *xfunction_base* class implements a multidimensional function operating on xexpression. Inheriting classes specify which kind of xexpression the *xfunction_base* operates on.

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::xconst_iterable< xfunction_base< F, R, CT... > >*

Size and shape

auto xt::*xfunction_base*`size() const`
Returns the size of the expression.

auto xt::*xfunction_base*`dimension() const`
Returns the number of dimensions of the function.

auto xt::*xfunction_base*`shape() const`
Returns the shape of the xfunction.

layout_type xt::*xfunction_base*`layout() const`
Returns the layout_type of the xfunction.

Data

template <class... *Args*>
auto xt::*xfunction_base*`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 xt::*xfunction_base*`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 xt::*xfunction_base*`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 xt::*xfunction_base*`broadcast_shape`(S &*shape*, bool *reuse_cache* = false) **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
- `reuse_cache`: boolean for reusing a previously computed shape

template <class S>
bool xt::*xfunction_base*`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

1.13.2 xfunction

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

The xfunction class implements a multidimensional function operating on xtensor expressions.

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::xfunction_base< F, R, CT... >*, *xt::xexpression< xfunction< F, R, CT... > >*

Public Functions

template <**class** Func, **class** U = std::enable_if<!std::is_base_of<Func, *self_type*>::value>>
xt::*xfunction*<u>**xfunction**</u>(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

1.13.3 xoptional_function

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

The *xoptional_function* class implements a multidimensional function operating on xoptional expressions.

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::xfunction_base< F, R, CT... >*, *xt::xexpression< xoptional_function< F, R, CT... > >*

Public Functions

template <**class** Func, **class** U = std::enable_if<!std::is_base_of<Func, *self_type*>::value>>
xt::*xoptional_function*<u>**xoptional_function**</u>(Func &&*func*, CT... *e*)
Constructs an *xoptional_function* applying the specified function to the given arguments.

Parameters

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

1.13.4 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>
xt::*xreducer***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 xt::*xreducer***size**() **const**

Returns the size of the expression.

auto xt::*xreducer***dimension**() **const**

Returns the number of dimensions of the expression.

auto xt::*xreducer***shape**() **const**

Returns the shape of the expression.

layout_type xt::*xreducer***layout**() **const**

Returns the shape of the expression.

Data

template <class... *Args*>
auto xt::*xreducer***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 xt::*xreducer***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 S>
auto xt::*xreducer*`operator[]`(**const** S &*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 It>
auto xt::*xreducer*`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 xt::*xreducer*`broadcast_shape`(S &*shape*, bool *reuse_cache* = false) **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 xt::*xreducer*`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, **class** ES = evaluation_strategy::lazy, class = std::enable_if_t<!std::is_base_of<evalua
auto xt`::reduce`(F &&*f*, E &&*e*, X &&*axes*, ES *es* = ES())

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.
- `evaluation_strategy`: evaluation strategy to use (lazy (default), or immediate)

1.13.5 xaccumulator

Defined in `xtensor/xaccumulator.hpp`

template <**class** F, **class** E, **class** ES = evaluation_strategy::immediate, **typename** std::enable_if_t<!std::is_integral< ES >:
auto xt::**accumulate**(F &&*f*, E &&*e*, ES *evaluation_strategy* = ES())

Accumulate and flatten array **NOTE** This function is not lazy!

Return returns xarray<T> filled with accumulated values

Parameters

- `f`: functor to use for accumulation
- `e`: xexpression to be accumulated
- `evaluation_strategy`: evaluation strategy of the accumulation

template <**class** F, **class** E, **class** ES = evaluation_strategy::immediate>
auto xt::**accumulate**(F &&*f*, E &&*e*, std::size_t *axis*, ES *evaluation_strategy* = ES())

Accumulate over axis **NOTE** This function is not lazy!

Return returns xarray<T> filled with accumulated values

Parameters

- `f`: Functor to use for accumulation
- `e`: xexpression to accumulate
- `axis`: Axis to perform accumulation over
- `evaluation_strategy`: evaluation strategy of the accumulation

1.13.6 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>
xt::*xgenerator***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 xt::*xgenerator***size**() **const**
Returns the size of the expression.

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

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

Data

template <class... *Args*>
auto xt::*xgenerator***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 xt::*xgenerator***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 xt::*xgenerator***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 xt::*xgenerator***broadcast_shape**(O &*shape*, bool *reuse_cache* = false) **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 xt::*xgenerator***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

1.13.7 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... *CT*>
auto xt::**concatenate**(std::tuple<CT...> &&*t*, std::size_t *axis* = 0)

Concatenates xexpressions along *axis*.

```
xt::xarray<double> a = {{1, 2, 3}};
xt::xarray<double> b = {{2, 3, 4}};
xt::xarray<double> c = xt::concatenate(xt::xtuple(a, b)); // => {{1, 2, 3},
                                                                {2, 3, 4}}
xt::xarray<double> d = xt::concatenate(xt::xtuple(a, b), 1); // => {{1, 2, 3, 2,␣
↪3, 4}}
```

Return xgenerator evaluating to concatenated elements

Parameters

- `t`: xtuple of xexpressions to concatenate
- `axis`: axis along which elements are concatenated

template <class... *CT*>
auto xt::**stack**(std::tuple<CT...> &&*t*, std::size_t *axis* = 0)

Stack xexpressions along *axis*.

Stacking always creates a new dimension along which elements are stacked.

```
xt::xarray<double> a = {1, 2, 3};
xt::xarray<double> b = {5, 6, 7};
xt::xarray<double> s = xt::stack(xt::xtuple(a, b)); // => {{1, 2, 3},
                                                          {5, 6, 7}}
xt::xarray<double> t = xt::stack(xt::xtuple(a, b), 1); // => {{1, 5},
                                                             {2, 6},
                                                             {3, 7}}
```

Return xgenerator evaluating to stacked elements

Parameters

- `t`: xtuple of xexpressions to concatenate
- `axis`: axis along which elements are stacked

template <class... *E*>
auto xt**::meshgrid**(E&&... *e*)

Return coordinate tensors from coordinate vectors.

Make N-D coordinate tensor expressions for vectorized evaluations of N-D scalar/vector fields over N-D grids, given one-dimensional coordinate arrays x1, x2,..., xn.

Return tuple of xgenerator expressions.

Parameters

- `e`: xexpressions to concatenate

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

1.13.8 xsort

Defined in `xtensor/xsort.hpp`
template <class E>
auto xt::**sort**(**const** *xexpression*<E> &*e*)
template <class E>
auto xt::**sort**(**const** *xexpression*<E> &*e*, placeholders::xtuph)
template <class E>
auto xt::**sort**(**const** *xexpression*<E> &*e*, std::size_t *axis*)

Sort xexpression (optionally along axis) The sort is performed using the `std::sort` functions.

A copy of the xexpression is created and returned.

Return sorted array (copy)

Parameters

- `e`: xexpression to sort

- `axis`: axis along which sort is performed

template <class E>
auto xt**::argmin**(**const** *xexpression*<E> &*e*)
template <class E>
auto xt**::argmin**(**const** *xexpression*<E> &*e*, std::size_t *axis*)
Find position of minimal value in xexpression.

Return returns xarray with positions of minimal value

Parameters

- `a`: xexpression to compute argmin on
- `axis`: select axis (or none)

template <class E>
auto xt**::argmax**(**const** *xexpression*<E> &*e*)
template <class E>
auto xt**::argmax**(**const** *xexpression*<E> &*e*, std::size_t *axis*)
Find position of maximal value in xexpression.

Return returns xarray with positions of minimal value

Parameters

- `a`: xexpression to compute argmin on
- `axis`: select axis (or none)

1.13.9 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

template <class T, **class** E = random::default_engine_type>
xtensor<**typename** T::value_type, 1> xt::random**::choice**(**const** *xexpression*<T> &*e*, std::size_t *n*, E &*engine* = random::*get_default_random_engine*())

Randomly select n unique elements from xexpression e.

Note: this function makes a copy of your data, and only 1D data is accepted.

Return xtensor containing 1D container of sampled elements

Parameters

- `e`: expression to sample from
- `n`: number of elements to sample
- `engine`: random number engine

1.14 Mathematical functions

1.14.1 Operators and related functions

Defined in `xtensor/xmath.hpp` and `xtensor/xoperation.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>
std::enable_if_t<xoptional_comparable<E1, E2>::value, 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

template <class R, **class** E>
auto xt**::cast**(E &&*e*)

Element-wise `static_cast`.

Returns an *xfunction* for the element-wise static_cast of *e* to type R.

Return an *xfunction*

Parameters

- `e`: 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
cast	element-wise *static_cast*

1.14.2 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

1.14.3 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

1.14.4 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

1.14.5 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

1.14.6 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

1.14.7 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

1.14.8 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

1.14.9 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_
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

1.14.10 Reducing functions

xtensor provides the following reducing functions for xexpressions:

Defined in xtensor/xmath.hpp
template <class E, **class** X, **class** ES = evaluation_strategy::lazy, class = std::enable_if_t<!std::is_base_of<evaluation_strate
auto xt**::sum**(E &&*e*, X &&*axes*, ES *es* = ES())
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)
- `es`: evaluation strategy of the reducer

template <class E, **class** X, **class** ES = evaluation_strategy::lazy, class = std::enable_if_t<!std::is_base_of<evaluation_strate
auto xt**::prod**(E &&*e*, X &&*axes*, ES *es* = ES())

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)
- `es`: evaluation strategy of the reducer

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, typename = concept_check <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, typename = concept_check <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

1.14.11 Accumulating functions

xtensor provides the following accumulating functions for xexpressions:

Defined in `xtensor/xmath.hpp`
template <class E>
auto xt**::cumsum**(E &&*e*)
template <class E>
auto xt**::cumsum**(E &&*e*, std::size_t *axis*)

Cumulative sum.

Returns the accumulated sum for the elements over given *axis* (or flattened).

Return an *xarray<T>*

Parameters

- `e`: an *xexpression*

- `axis`: the axes along which the cumulative sum is computed (optional)

template <class E>
auto xt::**cumprod**(E &&*e*)
template <class E>
auto xt::**cumprod**(E &&*e*, std::size_t *axis*)

Cumulative product.

Returns the accumulated product for the elements over given *axis* (or flattened).

Return an *xarray<T>*

Parameters

- `e`: an *xexpression*
- `axis`: the axes along which the cumulative product is computed (optional)

cumsum	cumulative sum of elements over a given axis
cumprod	cumulative product of elements over given axes

1.15 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.

1.15.1 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.

Visual Studio 2017 and alias templates with non-class template parameters and multiple aliasing levels

Alias template with non-class parameters only, and multiple levels of aliasing are not properly considered as types by Visual Studio 2017. The base `xcontainer` template class underlying xtensor container types has such alias templates defined. We avoid the multiple levels of aliasing in the case of Visual Studio.

1.15.2 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.

1.15.3 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.

1.15.4 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.

1.15.5 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.

1.16 Build and configuration

1.16.1 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
```

1.16.2 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.

1.17 xtensor internals

This section provides information about *xtensor*'s internals and its architecture. It is intended for developers who want to contribute to *xtensor* or simply understand how it works under the hood. *xtensor* makes heavy use of the CRTP pattern, template meta-programming, universal references and perfect forwarding. One should be familiar with these notions before going any further.

1.17.1 Concepts

xtensor's core is built upon key concepts captured in interfaces that are put together in derived classes through CRTP and multiple inheritance. Interfaces and classes that model expressions implement *value semantic*. CRTP and value semantic achieve static polymorphism and avoids performance overhead of virtual methods and dynamic dispatching.

xexpression

`xexpression` is the base class for all expression classes. It is a CRTP base whose template parameter must be the most derived class in the hierarchy. For instance, if `A` inherits from `B` which in turn inherits from `xexpression`, then `B` should be a template class whose template parameter is `A` and should forward this parameter to `xexpression`:

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

template <class T>
class B : public xexpression<T>
{
    // ...
};

class A : public B<A>
{
    // ...
};
```

`xexpression` only provides three overloads of a same function, that cast an `xexpression` object to the most inheriting type, depending on the nature of the object (lvalue, const lvalue or rvalue):

```
derived_type& derived_cast() & noexcept;
const derived_type& derived_cast() & noexcept;
derived_type derived_cast() && noexcept;
```

xiterable

The iterable concept is modeled by two classes, `xconst_iterable` and `xiterable`, defined in `xtensor/xiterable.hpp`. `xconst_iterable` provides types and methods for iterating on constant expressions, similar to the ones provided by the STL containers. Unlike the STL, the methods of `xconst_iterable` and `xiterable` are templated by a layout parameter that allows you to iterate over a N-dimensional expression in row-major order or column-major order. Row-major layout means that elements that only differ by their last index are contiguous in memory. Column-major layout means that elements that only differ by their first index are contiguous in memory.

```
template <class L>
const_iterator begin() const noexcept;
template <class L>
const_iterator end() const noexcept;
template <class L>
const_iterator cbegin() const noexcept;
template <class L>
const_iterator cend() const noexcept;

template <class L>
const_reverse_iterator rbegin() const noexcept;
template <class L>
const_reverse_iterator rend() const noexcept;
template <class L>
const_reverse_iterator crbegin() const noexcept;
template <class L>
const_reverse_iterator crend() const noexcept;
```

This template parameter is defaulted to `DEFAULT_LAYOUT` (see *Configuration*), so that *xtensor* expressions can be used in generic code such as:

```
std::copy(a.cbegin(), a.cend(), b.begin());
```

where `a` and `b` can be arbitrary types (from *xtensor*, the STL or any external library) supporting standard iteration.

`xiterable` inherits from `xconst_iterable` and provides non-const counterpart of methods defined in `xconst_iterable`. Like `xexpression`, both are CRTP classes whose template parameter must be the most derived type.

Besides traditional methods for iterating, `xconst_iterable` and `xiterable` provide overloads taking a shape parameter. This allows to iterate over an expression as if it was broadcast to the given shape:

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

int main(int argc, char* argv[])
{
    xt::xarray<int> a = { 1, 2, 3 };
    std::vector<std::size_t> shape = { 2, 3 };
    std::copy(a.cbegin(shape), a.cend(shape), std::output_iterator(std::cout, " "));
    // output: 1 2 3 1 2 3
}
```

Iterators returned by methods defined in `xconst_iterable` and `xiterable` are random access iterators.

xsemantic

The `xsemantic_base` interface provides methods for assigning an expression:

```
template <class E>
disable_xexpression<E, derived_type&> operator+=(const E&);

template <class E>
derived_type& operator+=(const xexpression<E>&);
```

and similar methods for `operator-=`, `operator*=`, `operator/=`, `operator%=`, `operator&=`, `operator|=` and `operator^=`.

The first overload is meant for computed assignment involving a scalar; it allows to write code like

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

int main(int argc, char* argv)
{
    xarray<int> a = { 1, 2, 3 };
    a += 4;
    std::cout << a << std::endl;
    // outputs { 5, 6, 7 }
}
```

We rely on SFINAE to remove this overload from the overload resolution set when the parameter that we want to assign is not a scalar, avoiding ambiguity.

Operator-based method taking a general `xexpression` parameter don't perform a direct assignment. Instead, the result is assigned to a temporary variable first, in order to prevent issues with aliasing. Thus, if `a` and `b` are expressions, the following

```
a += b
```

is equivalent to

```
temporary_type tmp = a + b;
a.assign(tmp);
```

Temporaries can be avoided with the assign-based methods:

```
template <class E>
derived_type& plus_assign(const xexpression<E>&);
template <class E>
derived_type&> minus_assign(const xexpression<E>&);
template <class E>
derived_type& multiplies_assign(const xexpression<E>&);
template <class E>
derived_type& divides_assign(const xexpression<E>&);
template <class E>
derived_type& modulus_assign(const xexpression<E>&);
```

`xsemantic_base` is a CRTP class whose parameter must be the most derived type in the hierarchy. It inherits from `xexpression` and forwards its template parameter to this latter one.

`xsemantic_base` also provides a assignment operator that takes an `xexpression` in its protected section:

```
template <class E>
derived_type& operator=(const xexpression<E>&);
```

Like computed assignment operators, it evaluates the expression inside a temporary before calling the `assign` method. Classes inheriting from `xsemantic_base` must redeclare this method either in their protected section (if they are not final classes) or in their public section. In both cases, they should forward the call to their base class.

Two refinements of this concept are provided, `xcontainer_semantic` and `xview_semantic`. Refer to the *Assignment* section for more details about semantic classes and how they're involved in expression assignment.

xsemantic classes hierarchy:

xcontainer

The `xcontainer` class provides methods for container-based expressions. It does not hold any data, this is delegated to inheriting classes. It assumes the data are stored using a strided-index scheme. `xcontainer` defines the following methods:

Shape, strides and size

```
size_type size() const noexcept;
size_type dimension() const noexcept;

const inner_shpae_type& shape() const noexcept;
const inner_strides_type& strides() const noexcept;
const inner_backstrides_type& backstrides() const noexcept;
```

Data access methods

```
template <class... Args>
const_reference operator()(Args... args) const;

template <class... Args>
const_reference at(Args... args) const;

template <class S>
disable_integral_t<S, const_reference> operator[](const S& index) const;

template <class I>
const_reference operator[](std::initialize_list<I> index) const;

template <class It>
const_reference element(It first, It last) const;

const container_type& data() const;
```

(and their non-const counterpart)

Broadcasting methods

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

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

Lower-level methods are also provided, meant for optimized assignment and BLAS bindings. They are covered in the *Assignment* section.

If you read the entire code of `xcontainer`, you'll notice that two types are defined for shape, strides and backstrides: `shape_type` and `inner_shape_type`, `strides_type` and `inner_strides_type`, and `backstrides_type` and `inner_backstrides_type`. The distinction between `inner_shape_type` and `shape_type` was motivated by the xtensor-python wrapper around numpy data structures, where the inner shape type is a proxy on the shape section of the numpy arrayobject. It cannot have a value semantics on its own as it is bound to the entire numpy array.

`xstrided_container` inherits from `xcontainer`; it represents a container that holds its shape and strides. It provides methods for reshaping the container:

```
template <class S = shape_type>
void resize(D&& shape, bool force = false);

template <class S = shape_type>
void resize(S&& shape, layout_type l);

template <class S = shape_type>
void resize(S&& shape, const strides_type& strides);

template <class S = shape_type>
void reshape(S&& shape, layout_type l);
```

Both `xstrided_container` and `xcontainer` are CRTP classes whose template parameter must be the most derived type in the hierarchy. Besides, `xcontainer` inherits from `xiterable`, thus providing iteration methods.

xfunction_base

The `xfunction_base` is used to model mathematical operations and functions. It provides similar methods to the ones defined in `xcontainer`, and embeds the functor describing the operation and its operands.

Like other interfaces, it is a CRTP class whose template parameter must be the most derived type of the hierarchy. It inherits from `xconst_iterable`, thus providing iteration methods.

The fact that `xfunction_base` is not a final class and needs to be inherited from allows to define function classes that provide a richer API and have them working with already existing code.

1.17.2 Implementation classes

Requirements

An implementation class in *xtensor* is a final class that models a specific kind of expression. It must inherit (either directly or indirectly) from `xexpression` and define (or inherit from classes that define) the following types:

container types

```
value_type;
reference;
const_reference;
pointer;
const_pointer;
size_type;
difference_type;
shape_type;
```

iterator types

```
iterator;
const_iterator;
reverse_iterator;
const_reverse_iterator;

template <class S, layout_type L>
broadcast_iterator<S, L>;
template <class S, layout_type L>
const_broadcast_iterator<S, L>;
template <class S, layout_type L>
reverse_broadcast_iterator<S, L>;
template <class S, layout_type L>
const_reverse_broadcast_iterator<S, L>;

storage_iterator;
const_storage_iterator;
reverse_storage_iterator;
const_reverse_storage_iterator;
```

layout data

```
static layout_type static_layout;
static bool contiguous_layout;
```

It must also provide the following methods, either by defining them itself, or by inheriting from classes that define them, partially or totally:

shape methods

```
size_type size() const noexcept;
size_type dimension() const noexcept;
const inner_shpae_type& shape() const noexcept;
```

broadcasting methods

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

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

data access methods

```
template <class... Args>
const_reference operator()(Args... args) const;

template <class... Args>
const_reference at(Args... args) const;

template <class S>
disable_integral_t<S, const_reference> operator[](const S& index) const;

template <class I>
const_reference operator[](std::initialize_list<I> index) const;

template <class It>
const_reference element(It first, It last) const;

const container_type& data() const;
```

iteration methods

These methods are usually provided by inheriting from `xconst_iterable` or `xiterable`. See *Iterating over expressions* for more details.

If the expression is mutable, it must also define the non-const counterparts of the data access methods, and inherits from a semantic class to provide assignment operators.

xarray and xtensor

Although they represent different concepts, `xarray` and `xtensor` have really similar implementations so only `xarray` will be covered.

`xarray` is a strided array expression that can be assigned to. Everything `xarray` needs is already defined in classes modeling *Concepts*, so `xarray` only has to inherit from these classes and define constructors and assignment operators:

Besides implementing the methods that define value semantic, `xarray` and `xtensor` hold the data container. Since the `xcontainer` base class implements all the logic for accessing the data, it must me able to access the data container. This is achieved by requiring that every class inheriting from `xcontainer` provides the following methods:

```
container_type& data_impl() noexcept;
const container_type& data_impl() const noexcept;
```

These are the implementation methods of the `data()` interface methods defined in `xcontainer`, and thus are defined in the private section of `xarray` and `xtensor`. In order to grant access to `xcontainer`, this last one is declared as `friend`:

```
template <class EC, layout_type L, class SC, class Tag>
class xarray : public xstrided_container<xarray<EC, L, SC, Tag>,
               public xcontainer_semantic<xarray<EC, L, SC, Tag>>
{
public:

    // ....

private:

    container_type m_data;
    container_type& data() noexcept;
    const container_type& data() const noexcept;

    friend class xcontainer<xarray<EC, L, SC, Tag>>;
};
```

This pattern is similar to the template method pattern used in hierarchy of classes with entity semantic (see virtuality).

Inner types definition

Although the base classes use the types defined in the Requirement section, they cannot define them; first because different base classes may need the same types and we want to avoid duplication of type definitions. The second reason is that most of the types may rely on other types specific to the implementation classes. For instance, `value_type`, `reference`, etc, of `xarray` are simply the types defined in the container type hold by `xarray`:

```
using value_type = typename container_type::value_type;
using reference = typename container_type::reference;
using const_reference = typename container_type::const_reference;
...
```

Moreover, CRTP base classes cannot access inner types defined in CRTP leaf classes, because a CRTP leaf class is only declared, not defined, when the CRTP base class is being defined.

The solution is to define those types in an external structure that is specialized for each CRTP leaf class:

```
// Declaration only, no generic definition
template <class C>
struct xcontainer_inner_types;
```

In `xarray.hpp`

```
template <class EC, layout_type L, class SC, class Tag>
struct xcontainer_inner_types<xarray<EC, L,SC, Tag>>
{
    // Definition of types required by CRTP bases
};
```

In order to avoid a lot of boilerplate, the CRTP base classes expect only a few types to be defined in this structure, and then compute the other types, based on these former definitions. The requirements on types definition regarding the base classes is detailed below.

xsemantic

The semantic classes only expect the following type: `temporary_type`.

xcontainer

`xcontainer` and `xstrided_container` expect the following types to be defined:

```
container_type;
shape_type;
strides_type;
backstrides_type;
inner_shape_type;
inner_strides_type;
inner_backstrides_type;
layout_type;
```

xiterable

Since many expressions are not containers, the definition of types required by the iterable concept is done in a dedicated structure following the same pattern as `xcontainer_inner_types`, i.e. a sturcture declared and specialized for each final class:

```
template <class C>
struct xiterable_inner_types;
```

The following types must be defined in each specialization:

```
inner_shape_type;
const_stepper;
stepper;
```

More detail about the stepper types is given in *Iterating over expressions*.

1.17.3 Expression tree

Most of the expressions in *xtensor* are lazy-evaluated, they do not hold any value, the values are computed upon access or when the expression is assigned to a container. This means that *xtensor* needs somehow to keep track of the expression tree.

xfunction_base / xfunction

A node in the expression tree may be represented by different classes in *xtensor*; here we focus on basic arithmetic operations and mathematical functions, which are represented by an instance of `xfunction_base`. This is a template class whose parameters are:

- a functor describing the operation of the mathematical function
- the return type, computing from the types of the child expressions involved in the operation
- the closures of the child expressions, i.e. the most optimal way to store each child expression

Although `xfunction_base` provides the API required for an `expression`, it is not the final class representing expression tree nodes, it is meant to be inherited. This allow to define function classes with a richer API and to extend *xtensor*'s expression system. The implementation class used to represent operations and mathematical functions in *xtensor* is `xfunction`, it accepts the same template parameters as its base class. It contains nothing more than constructors and assign operators, all the magic being done in `xfunction_base`.

Consider the following code:

```
xarray<double> a = xt::ones({2, 2});
xarray<double> b = xt::ones({2, 2});

auto f = (a + b);
```

Here the type of f is xfunction<plus, double, const xarray<double>&, const xarray<double>&>, and f stores constant references on the arrays involved in the operation. This can be illustrated by the figure below:

The implementation of xfunction_base methods is quite easy: they forward the call to the nodes and apply the operation when this makes sense. For instance, assuming that the operands are stored as m_first and m_second, and the functor describing the operation as m_functor, the implementation of operator() and broadcast_shape looks like:

```
template <class F, class R, class... CT>
template <class... Args>
inline auto xfunction_base<F, R, CT...>::operator()(Args... args) const -> const_
↪reference
{
    return m_functor(m_first(args...), m_second(args...));
}

template <class F, class R, class... CT>
template <class S>
inline bool xfunction_base<F, R, CT...>::broadcast_shape(S& shape) const
{
    return m_first.broadcast_shape(shape) && m_second.broadcast_shape(shape);
}
```

In fact, xfunction_base can handle an arbitrary number of arguments. The practical implementation is slightly more complicated than the code snippet above, however the principle remains the same.

Holding expressions

Each node of an expression tree holds const references to its child nodes, or the child nodes themselves, depending on their nature. When building a complex expression, if a part of this expression is an rvalue, it is moved inside its parent, else a constant reference is used:

```
xarray<double> some_function();

xarray<double> a = xt::ones({2, 2});
auto f = a + some_function();
```

Here f holds a constant reference on a, while the array returned by some_function is moved into f. The actual types held by the expression are the **closure types**, more details can be found in *Closure semantics*.

Building the expression tree

As previously stated, each mathematical function in xtensor returns an instance of xfunction. This section explains in details how the template parameters of xfunction are computed according to the type of the function, the number and the types of its arguments. Let's consider the definition of operator+:

```
template <class E1, class E2>
inline auto operator+(E1&& e1, E2&& e2) -> detail::xfunction_type<detail::plus, E1,␣
↪E2>
{
    return detail::make_xfunction<detail::plus>(std::forward<E1>(e1), std::forward<E2>
↪(e2));
}
```

This top-level function selects the appropriate functor and forwards its arguments to the `make_xfunction` generator. This latter is responsible for setting the remaining template parameters of `xfunction`:

```
template <template <class...> class F, class... E>
inline auto make_xfunction(E&&... e) noexcept
{
    using expression_tag = xexpression_tag_t<E...>;
    using functor_type = build_functor_type_t<expression_tag, F, E...>;
    using type = select_xfunction_expression_t<expression_tag, functor_type, const_
↪xclosure_t<E>...>;
    return type(functor_type(), std::forward<E>(e)...);
}
```

The first line computes the `expression_tag` of the expression. This tag is used for selecting the right implementation class inheriting from `xfunction_base`. In *xtensor*, three tags are provided, with the following mapping:

- `xscalar_expression_tag` -> `xfunction`
- `xtensor_expression_tag` -> `xfunction`
- `xoptional_expression_tag` -> `xoptional_function`

Any expression may define a tag as its `expression_tag` inner type. If not, `xtensor_expression_tag` is used by default. Tags have different priorities so that a resulting tag can be computed for expressions involving different tag types. As we will see in the next section, this system of tags and mapping make it easy to plug new functions types in *xtensor* and have them working with all the mathematical functions already implemented.

The function class mapped to the expression tag is retrieved in the third line of `make_xfunction`, that is:

```
using type = select_xfunction_expression_t<expression_tag, functor_type, const_
↪xclosure_t<E>...>;
```

`const_closure_t` computes the closure type (see *Closure semantics*) of each argument and passes it to the function class to instantiate.

The exact type of the functor is computed thanks to the `build_functor_type_t` generator. It computes the return type of the function according to the `value_type` of the arguments (most of the times, a simple type promotion is enough).

Once all the types are known, `make_xfunction` can instantiate the right function type and returns it:

```
return type(functor_type(), std::forward<E>(e)...);
```

Plugging new function types

As mentioned in the section above, one can define a new function class and have it used by *xtensor*'s expression system. Let's illustrate this with the `xoptional_function` class. The first thing to do is to define a new tag:

```
struct xoptional_expression_tag
{
};
```

Then the tag selection rules must be updated if we want to be able to mix `xtensor_expression_tag` and `xoptional_expression_tag`. This is done by specializing the `expression_tag_and` metafunction available in the namespace `xt::detail`:

```
namespace xt
{
    namespace detail
    {
        template <>
        struct expression_tag_and<xtensor_expression_tag, xoptional_expression_tag>
        {
            using type = xoptional_expression_tag;
        };

        template <>
        struct expression_tag_and<xoptional_expression_tag, xtensor_expression_tag>
            : expression_tag_and<xtensor_expression_tag, xoptional_expression_tag>
        {
        };
    }
}
```

The second specialization simply forwards to the first one so we don't duplicate code. Note that when plugging your own function class, these specializations can be skipped if the new function class (and its corresponding tag) is not compatible, and thus not supposed to be mixed, with the function classes provided by *xtensor*.

The las thing required is to specialize the `select_xfunction_expression` metafunction, as it is shown below:

```
namespace xt
{
    namespace detail
    {
        template <class F, class... E>
        struct select_xfunction_expression<xoptional_expression_tag, F, E...>
        {
            using type = xoptional_function<F, typename F::result_type, E...>;
        };
    }
}
```

In this example, `xoptional_function` inherits from `xfunction_base` and define some additional methods, so it provides a richer API the `xfunction`. However it is possible to define a function class with a different API, thus not inheriting from `xfunction_base`. In that case, the assignment mechanics need to be customized too, this is detailed in *Assignment*.

1.17.4 Iterating over expressions

xiterable and inner types

xtensor provides two base classes for making expressions iterable: `xconst_iterable` and `xiterable`. They define the API for iterating as described in *Concepts*. For an expression to be iterable, it must inherit directly or indirectly from one of these classes. For instance, the `xbroadcast` class is defined as following:

```
template <class CT, class X>
class xbroadcast : public xexpression<xbroadcast<CT, X>>,
                   public xconst_iterable<xbroadcast<CT, X>>
{
    // ...
};
```

Some of the methods provided by `xconst_iterable` or `xiterable` may need to be refined in the inheriting class. In that case, a common pattern is to make the inheritance private, import the methods we need with using declaration and redefine the methods whose behavior differ from the one provided in the base class. This is what is done in `xfunction_base`:

```
template <class F, class R, class... CT>
class xfunction_base : private xconst_iterable<xfunction_base<F, R, CT...>>
{
public:

    using self_type = xfunction_base<F, R, CT...>;
    using iterable_base = xconst_iterable<self_type>;

    using iterable_base::begin;
    using iterable_base::end;
    using iterable_base::cbegin;
    using iterable_base::cend;
    using iterable_base::rbegin;
    using iterable_base::rend;
    using iterable_base::crbegin;
    using iterable_base::crend;

    template <layout_type L = DL>
    const_storage_iterator storage_begin() const noexcept;
    template <layout_type L = DL>
    const_storage_iterator storage_end() const noexcept;
    template <layout_type L = DL>
    const_storage_iterator storage_cbegin() const noexcept;
    template <layout_type L = DL>
    const_storage_iterator storage_cend() const noexcept;

    template <layout_type L = DL>
    const_reverse_storage_iterator storage_rbegin() const noexcept;
    template <layout_type L = DL>
    const_reverse_storage_iterator storage_rend() const noexcept;
    template <layout_type L = DL>
    const_reverse_storage_iterator storage_crbegin() const noexcept;
    template <layout_type L = DL>
    const_reverse_storage_iterator storage_crend() const noexcept;
};
```

The implementation of the iterator methods defined in `xconst_iterable` and `xiterable` rely on a few types and methods that must be defined in the inheriting class.

First, as stated in the *xiterable section*, the `xiterable_inner_types` structure must be specialized as illustrated below:

```
template <class F, class R, class... CT>
struct xiterable_inner_types<xfunction_base<F, R, CT...>>
{
```

(continues on next page)

(continued from previous page)

```
    using inner_shape_type = promote_shape_t<typename std::decay_t<CT>::shape_type...>
↪;
    using const_stepper = xfunction_stepper<F, R, CT...>;
    using stepper = const_stepper;
};
```

Then the inheriting class must define the following methods:

```
template <class S>
const_stepper stepper_begin(const S& shape) const noexcept;
template <class S>
const_stepper stepper_end(const S& shape, layout_type l) const noexcept;
```

If the expression class inherits from `xiterable` instead of `xconst_iterable`, the non-const counterparts of the previous methods must also be defined. Every method implemented in one of the base class eventually calls one of these stepper methods, whose mechanics is explained hereafter.

Steppers

Steppers are the low-level tools for iterating over expressions. They provide a raw API for "stepping" of a given amount in a given dimension, dereferencing the stepper, and moving it to the beginning or the end of the expression:

```
reference operator*() const;

void step(size_type dim, size_type n = 1);
void step_back(size_type dim, size_type n = 1);
void reset(size_type dim);
void reset_back(size_type dim);

void to_begin();
void to_end(layout_type l);
```

The `reset` and `reset_back` methods are shortcut to `step_back` and `step` called with `dim` and `shape[dim] - 1`. The steppers are initialized with a "position" (that may be an index, a pointer to the underlying buffer of an container-based expression, etc...) in the expression, and can then be used to browse the expression in any direction:

In this diagram, the data is stored in row-major order, and we step in the second dimension (dimension index starts at 0). The positions of the stepper are represented by the red dots.

The `to_end` method takes a layout parameter, because the ending positions of a stepper depend on the layout used to iterate. Indeed, if we call `step_back` after a call to `to_end`, we want the stepper to point to the last element. To ensure this for both row-major order and column-major order iterations, the ending positions must be set as shown below:

The red dots are the position of a stepper iterating in column-major while the green ones are the positions of a stepper iterating in row-major order. Thus, if we assume that `p` is a pointer to the last element (the square containing 11), the ending positions of the stepper are `p + 1` in row-major, and `p + 3` in column-major order.

A stepper is specific to an expression type, therefore implementing a new kind of expression usually requires to implement a new kind of stepper. However *xtensor* provides a generic `xindexed_stepper` class, that can be used with any kind of expressions. Even though it is generally not optimal, authors of new expression types can make use of the generic index stepper in a first implementation.

Broadcasting

The steppers of container-based expressions rely on strides and backstrides for stepping. A naive implementation of the `step` method would be:

```
template <class C>
inline void xstepper<C>::step(size_type dim, size_type n)
{
    m_it += n * p_c->strides()[dim];
}
```

where `m_it` is an itrator on the underlying buffer, and `p_c` a pointer to the container_based expression.

However, this implementation fails when broadcasting is involved. Consider the following expression:

```
xarray<int> a = {{0, 1,  2,  3},
                 {4, 5,  6,  7},
                 {8, 9, 10, 11}};
xarray<int> b = {0, 1, 2, 3};
auto r = a + b;
```

`r` is an `xfunction` representing the sum of `a` and `b`. The stepper specific to this expression holds the steppers of the arguments of the function; calling `step` or `step_back` results in calling `step` or `step_back` of the steppers of `a` and `b`.

According to the broadcasting rules, the shape of `r` is `{ 3, 4}`. Thus, calling `r.stepper_begin().step(1, 1)` will eventually call `b.stepper_begin().step(1, 1)`, leading to undefined behavior since the shape of `b` is `{4}`. To avoid that, a broadcasting offset is added to the stepper:

```
template <class C>
inline void xstepper<C>::step(size_type dim, size_type n)
{
    if (dim >= m_offset)
    {
        m_it += difference_type(n * p_c->strides()[dim - m_offset]);
    }
}
```

This implementation takes into account that the broadcasting is done on the last dimension and dimensions are stored in ascending order; here dimension 1 of `a` correpsonds to dimension 0 of `b`.

This implementation ensures that a step in dimension 0 of the function updates the stepper of `a` while the stepper of `b` remains unchanged; on the other hand, stepping in dimension 1 will update both steppers, as illustrated below:

The red dots are initial stepper positions, the green dots and blue dots are the positions of the steppers after calling `step` with different dimension arguments.

Iterators

xtensor iterator is implemented in the `xiterator` class. This latter provides a STL compliant iterator interface, and is built upon the steppers. Whereas the steppers are tied to the expression they refer to, `xiterator` is generic enough to work with any kind of stepper.

An iterator holds a stepper an multi-dimensional index. A call to `operator++` increases the index and calls the `step` method of the stepper accordingly. The way the index is increased depends on the layout used for iterating. For a row-major order iteration over a container with shape `{3, 4}`, the index iterating sequence is:

```
{0, 0}
{0, 1}
{0, 2}
{0, 3}
{1, 0}
{1, 1}
{1, 2}
{1, 3}
{2, 0}
{2, 1}
{2, 2}
{2, 3}
```

When a member of an index reaches its maximum value, it is reset to 0 and the member in the next dimension is increased. This translates in the calls of two methods of the stepper, first `reset` and then `step`. This is illustrated by the following picture:

The green arrows represent the iteration from `{0, 0}` to `{0, 3}`. The blue arrows illustrate what happens when the index is increased from `{0, 3}` to `{1, 0}`: first the stepper is reset to `{0, 0}`, then `step(0, 1)` is called, setting the stepper to the position `{1, 0}`.

`xiterator` implements a random access iterator, providing `operator--` and `operator[]` methods. The implementation of these methods is similar to the one of `operator++`.

1.17.5 Assignment

In this section, we consider the class `xarray` and its semantic bases (`xcontainer_semantic` and `xsemantic_base`) to illustrate how the assignment works. *xtensor* provides different mechanics of assignment depending on the type of expression.

Extended copy semantic

`xarray` provides an extended copy constructor and an extended assignment operator:

```
template <class E>
xarray(const xexpression<E>&);

template <class E>
self_type& operator=(const xexpression<E>& e);
```

The assignment operator forwards to `xsemantic_base::operator=` whose implementation is given below:

```
template <class E>
derived_type& operator=(const xexpression<E>& e)
{
    temporary_type tmp(e);
    return this->derived_cast().assign_temporary(std::move(tmp));
}
```

Here `temporary_type` is `xarray`, the assignment operator computes the result of the expression in a temporary variable and then assigns it to the `xarray` instance. This temporary variable avoids aliasing when the array is involved in the rhs expression where broadcasting happens:

```
xarray<double> a = {1, 2, 3, 4};
xarray<double> b = {{1, 2, 3, 4},
                    {5, 6, 7, 8}};
a = a + b;
```

The extended copy constructor calls `xsemantic_base::assign` which calls `xcontainer::assign_xexpression`. This two-steps invocation allows to provide an uniform API (assign, plus_assign, minus_assign, etc) in the top base class while specializing the implementations in inheriting classes (`xcontainer_semantic` and `xview_semantic`). `xcontainer::assign_xexpression` eventually calls the free function `xt::assign_xexpression` which will be discussed in details later.

The behavior of the extended copy semantic can be summarized with the following diagram:

Computed assignment

Computed assignment can be achieved either with traditional operators (`operator+=`, `operator-=`) or with the corresponding assign functions (`plus_assign`, `minus_assign`, etc). The computed assignment operators forwards to the extended assignment operator as illustrated below:

```
template <class D>
template <class E>
inline auto xsemantic_base<D>::operator+=(const xexpression<E>& e) -> derived_type&
{
    return operator=(this->derived_cast() + e.derived_cast());
}
```

The computed assign functions, like `assign` itself, avoid the instantiation of a temporary variable. They call the overload of `computed_assign` which, in the case of `xcontainer_semantic`, simply forwards to the free function `xt::computed_assign`:

```
template <class D>
template <class E>
inline auto xsemantic_base<D>::plus_assign(const xexpression<E>& e) -> derived_type&
{
    return this->derived_cast().computed_assign(this->derived_cast() + e.derived_
↪cast());
}

template <class D>
template <class E>
inline auto xcontainer_semantic<D>::computed_assign(const xexpression<E>& e) ->␣
↪derived_type&
{
    xt::computed_assign(*this, e);
    return this->derived_cast();
}
```

Again this two-steps invocation allows to provide a uniform API in `xsemantic_base` and specializations in the inheriting semantic classes. Besides this allows some code factorization since the assignment logic is implemented only once in `xt::computed_assign`.

Scalar computed assignment

Computed assignment operators involving a scalar are similar to computed assign methods:

```
template <class D>
template <class E>
inline auto xsemantic_base<D>::operator+=(const E& e) -> disable_xexpression<E,␣
↪derived_type&>
{
    return this->derived_cast().scalar_computed_assign(e, std::plus<>());
}

template <class D>
template <class E, class F>
inline auto xcontainer_semantic<D>::scalar_computed_assign(const E& e, F&& f) ->␣
↪derived_type&
{
    xt::scalar_computed_assign(*this, e, std::forward<F>(f));
    return this->derived_cast();
}
```

The free function `xt::scalar_computed_assign` contains optimizations specific to scalars.

Expression assigners

The three main functions for assigning expressions (`assign_xexpression`, `computed_assign` and `scalar_computed_assign`) have a similar implementation: they forward the call to the `xexpression_assigner`, a template class that can be specialized according to the expression tag:

```
template <class E1, class E2>
inline void assign_xexpression(xexpression<E1>& e1, const xexpression<E2>& e2)
{
    using tag = xexpression_tag_t<E1, E2>;
    xexpression_assigner<tag>::assign_xexpression(e1, e2);
}

template <class Tag>
class xexpression_assigner : public xexpression_assigner_base<Tag>
{
public:

    using base_type = xexpression_assigner_base<Tag>;

    template <class E1, class E2>
    static void assign_xexpression(xexpression<E1>& e1, const xexpression<E2>& e2);

    template <class E1, class E2>
    static void computed_assign(xexpression<E1>& e1, const xexpression<E2>& e2);

    template <class E1, class E2, class F>
    static void scalar_computed_assign(xexpression<E1>& e1, const E2& e2, F&& f);

    // ...
};
```

xtensor provides specializations for `xtensor_expression_tag` and `xoptional_expression_tag`. When implementing a new function type whose API is unrelated to the one of `xfunction_base`, the `xexpression_assigner` should be specialized so that the assignment relies on this specific API.

assign_xexpression

The `assign_xexpression` methods first resizes the lhs expression, it chooses an assignment method depending on many properties of both lhs and rhs expressions. One of these properties, computed during the resize phase, is the nature of the assignment: trivial or not. The assignment is said to be trivial when the memory layout of the lhs and rhs are such that assignment can be done by iterating over a 1-D sequence on both sides. In that case, two options are possible:

- if `xtensor` is compiled with the optional `xsimd` dependency, and if the layout and the `value_type` of each expression allows it, the assignment is a vectorized index-based loop operating on the expression buffers.
- if the `xsimd` assignment is not possible (for any reason), an iterator-based loop operating on the expresion buffers is used instead.

These methods are implemented in specializations of the `trivial_assigner` class.

When the assignment is not trivial, *Steppers* are used to perform the assignment. Instead of using `xiterator` of each expression, an instance of `data_assigner` holds both steppers and makes them step together.

computed_assign

The `computed_assign` method is slightly different from the `assign_xexpression` method. After resizing the lhs member, it checks if some broadcasting is involved. If so, the rhs expression is evaluated into a temporary and the temporary is assigned to the lhs expression, otherwise rhs is directly evaluated in lhs. This is because a computed assignment always implies aliasing (meaning that the lhs is also involved in the rhs): `a += b;` is equivalent to `a = a + b;`.

scalar_computed_assign

The `scalar_computed_assign` method simply iterates over the expression and applies the scalar operation on each value:

```
template <class Tag>
template <class E1, class E2, class F>
inline void xexpression_assigner<Tag>::scalar_computed_assign(xexpression<E1>& e1,␣
↪const E2& e2, F&& f)
{
    E1& d = e1.derived_cast();
    std::transform(d.cbegin(), d.cend(), d.begin(),
                   [e2, &f](const auto& v) { return f(v, e2); });
}
```

1.18 Extending xtensor

`xtensor` provides means to plug external data structures into its expression engine without copying any data.

1.18.1 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 `reshape`, `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.

1.18.2 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;
```

(continues on next page)

(continued from previous page)

```
    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.

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
```

(continues on next page)

(continued from previous page)

```
    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 resize methods

The next methods to define are the overloads of `resize`. `xtensor` provides utilities functions to compute strides based on the shape and the layout, so the implementation of the `resize` overloads is straightforward:

```
#include "xtensor/xstrides.hpp" // for utitilities functions

template <class T>
void resize(const shape_type& shape)
{
    if(m_shape != shape)
        resize(shape, layout::row_major);
}

template <class T>
void resize(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 resize(const shape_type& shape, const strides_type& strides)
{
```

(continues on next page)

(continued from previous page)

```
    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.

1.18.3 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());
}
```

(continues on next page)

(continued from previous page)

```
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 resize 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);
}
```

1.19 Releasing xtensor

1.19.1 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.
- In file `README.md`, update the dependencies table.
- 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`)

1.19.2 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.

1.20 From numpy to xtensor

1.20.1 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}})` `xt::xtensor<double, 2>({{3, 4}, {5, 6}})`
`arr.reshape([3, 4])`	`arr.reshape({3, 4})`
`arr.astype(np.float64)`	`xt::cast<double>(arr)`

1.20.2 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.

1.20.3 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]`
`a[:5, 1:]`
`a[5:1:-1, :]`
	`xt::view(a, xt::all(), xt::newaxis())`
	`xt::view(a, xt::range(_, 5), xt::range(1, _))`
	`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}})`

1.20.4 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})`
`np.random.choice(arr, 5)`	`xt::random::choice(arr, 5)`

1.20.5 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)`

1.20.6 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)`

1.20.7 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})` `a.end({3, 4})`
Iterating over `a` in a row-major fashion	`a.begin<layout_type::row_major>()` `a.begin<layout_type::row_major>()`
Iterating over `a` in a column-major fashion	`a.begin<layout_type::column_major>()` `a.end<layout_type::column_major>()`

1.20.8 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)`

1.20.9 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)`

1.20.10 Minimum, Maximum, Sorting

Python 3 - numpy	C++ 14 - xtensor
`np.amin(a)`	`xt::amin(a)`
`np.amax(a)`	`xt::amax(a)`
`np.argmin(a)`	`xt::argmin(a)`
`np.argmax(a, axis=1)`	`xt::argmax(a, axis=1)`
`np.sort(a, axis=1)`	`xt::sort(a, axis=1)`

1.20.11 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)`
`np.conj(a)`	`xt::conj(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())`.

1.20.12 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.

1.20.13 I/O

Print options

These options determine the way floating point numbers, tensors and other xtensor expressions are displayed.

Python 3 - numpy	C++ 14 - xtensor
`np.set_printoptions(precision=4)`	`xt::print_options::set_precision(4)`
`np.set_printoptions(threshold=5)`	`xt::print_options::set_threshold(5)`
`np.set_printoptions(edgeitems=3)`	`xt::print_options::set_edgeitems(3)`

Reading npy, csv file formats

Functions `load_csv` and `dump_csv` respectively take input and output streams as arguments.

Python 3 - numpy	C++ 14 - xtensor
`np.load(file)`	`xt::load_npy<double>(filename)`
`np.load_txt(filename, delimiter=',')`	`xt::load_csv<double>(stream)`

1.20.14 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)`

1.20.15 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)`

1.21 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.

1.21.1 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`.

1.21.2 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.

1.21.3 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.

1.21.4 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*.

1.21.5 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.

1.22 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.

1.22.1 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.

1.22.2 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;
```

1.22.3 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.

1.23 Related projects

1.23.1 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"
#include <numeric>
#include <cmath>
```

(continues on next page)

(continued from previous page)

```
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]]
```

1.23.2 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

1.23.3 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]]
```

1.23.4 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

1.23.5 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)]]
```

(continues on next page)

(continued from previous page)

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

R code

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

Outputs

```
1.2853996391883833
```

1.23.6 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.

1.23.7 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.

1.23.8 xsimd

The xsimd project provides a unified API for making use of the SIMD features of modern preprocessors for C++ library authors. It also provides accelerated implementation of common mathematical functions operating on batches.

xsimd is an optional dependency to `xtensor` which enable SIMD vectorization of xtensor operations. This feature is enabled with the `XTENSOR_USE_XSIMD` compilation flag, which is set to *false* by default.

1.23.9 xtl

The xtl project, the only dependency of `xtensor` is a C++ template library holding the implementation of basic tools used accross the libraries in the QuantStack ecosystem.

Index

X