
xtensor

Jan 16, 2019

INSTALLATION

1 Licensing

3

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 the expression system can easily be written.

In fact, *xtensor* can be used to **process numpy data structures in-place** using Python's [buffer protocol](#). For more details on the numpy bindings, check out the [xtensor-python](#) project. Language bindings for R and Julia are also available.

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

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
```

`path_to_prefix` is the absolute path to the folder where `cmake` searches for dependencies and installs libraries. `xtensor` installation from `cmake` assumes this folder contains `include` and `lib` subfolders.

See the [Build and configuration](#) section for more details about `cmake` options.

1.1.5 Including `xtensor` in your project

The different packages of `xtensor` are built with `cmake`, so whatever the installation mode you choose, you can add `xtensor` to your project using `cmake`:

```
find_package(xtensor REQUIRED)
target_include_directories(your_target PUBLIC ${xtensor_INCLUDE_DIRS})
target_link_libraries(your_target PUBLIC xtensor)
```

1.2 Changelog

1.2.1 0.19.1

- Add string specialization to `lexical_cast` #1281.
- Added HDF5 reference for `xtensor-io` #1284.
- Fixed view index remap issue #1288.
- Fixed gcc 8.2 deleted functions #1289.
- Fixed reducer for 0d input #1292.

- Fixed `check_element_index` #1295.
- Added comparison functions #1297.
- Add some tests to ensure chrono works with xexpressions #1272.
- Refactor `functor_view` #1276.
- Documentation improved #1302.
- Implementation of shift operators #1304.
- Make functor adaptor stepper work for proxy specializations #1305.
- Replaced `auto&` with `auto&&` in `assign_to` #1306.
- Fix namespace in `xview_utils.hpp` #1308.
- Introducing `flatten_indices` and `unravel_indices` #1300.
- Default layout parameter for `ravel` #1311.
- Fixed `xvie_stepper` #1317.
- Fixed assignment of view on view #1314.
- Documented indices #1318.
- Fixed shift operators return type #1319.

1.2.2 0.19.0

Breaking changes

- Upgraded to `xtl 0.5` #1275.

Other changes

- Removed type-o in docs, minor code style consistency update #1255.
- Removed most of the warnings #1261.
- Optional bitwise fixed #1263.
- Prevent macro expansion in `std::max` #1265.
- Update `numpy.rst` #1267.
- Update `getting_started.rst` #1268.
- keep and drop `step_size` fixed #1270.
- Fixed typo in `xadapt` #1277.
- Fixed typo #1278.

1.2.3 0.18.3

- Exporting optional dependencies #1253.
- 0-D HTML rendering #1252.
- Include `nlohmann_json` in `xio` for mime bundle repr #1251.

- Fixup xview scalar assignment #1250.
- Implemented *from_indices* #1240.
- *xtensor_forward.hpp* cleanup #1243.
- default layout-type for *unravel_from_strides* and *unravel_index* #1239.
- xfunction iterator fix #1241.
- xstepper fixes #1237.
- *print_options* io manipulators #1231.
- Add syntactic sugar for reducer on single axis #1228.
- Added view vs. adapt benchmark #1229.
- added precisions to the installation instructions #1226.
- removed data interface from dynamic view #1225.
- add xio docs #1223.
- Fixup xview assignment #1216.
- documentation updated to be consistent with last changes #1214.
- prevents macro expansion of `std::max` #1213.
- Fix minor typos #1212.
- Added missing assign operator in *xstrided_view* #1210.
- *argmax* on axis with single element fixed #1209.

1.2.4 0.18.2

- expression tag system fixed #1207.
- optional extension for generator #1206.
- optional extension for *xview* #1205.
- optional extension for *xstrided_view* #1204.
- optional extension for reducer #1203.
- optional extension for *xindex_view* #1202.
- optional extension for *xfunctor_view* #1201.
- optional extension for broadcast #1198.
- extension API and code cleanup #1197.
- *xscalar* optional refactoring #1196.
- Extension mechanism #1192.
- Many small fixes #1191.
- Slight refactoring in *step_size* logic #1188.
- Fixup call of const overload in assembly storage #1187.

1.2.5 0.18.1

- Fixup xio forward declaration #1185.

1.2.6 0.18.0

Breaking changes

- Assign and trivial_broadcast refactoring #1150.
- Moved array manipulation functions (transpose, ravel, flatten, trim_zeros, squeeze, expand_dims, split, atleast_Nd, atleast_1d, atleast_2d, atleast_3d, flip) from xstrided_view.hpp to xmanipulation.hpp #1153.
- iterator API improved #1155.
- Fixed where and nonzero function behavior to mimic the behavior from NumPy #1157.
- xsimd and functor refactoring #1173.

New features

- Implement rot90 #1153.
- Implement argwhere and flatnonzero #1157.
- Implemented xexpression_holder #1164.

Other changes

- Warnings removed #1159.
- Added missing include #1162.
- Removed unused type alias in xmath/average #1163.
- Slices improved #1168.
- Fixed xdrop_slice #1181.

1.2.7 0.17.4

- perfect forwarding in xoptional_function constructor #1101.
- fix issue with base_simd #1103.
- XTENSOR_ASSERT fixed on Windows #1104.
- Implement xmasked_value #1032.
- Added setdiff1d using stl interface #1109.
- Added test case for setdiff1d #1110.
- Added missing reference to diff in From numpy to xtensor section #1116.
- Add amax and amin to the documentation #1121.
- histogram and histogram_bin_edges implementation #1108.

- Added numpy comparison for `interp` #1111.
- Allow multiple return type reducer functions #1113.
- Fixes average bug + adds Numpy based tests #1118.
- Static `xfunction` cache for fixed sizes #1105.
- Add negative reshaping axis #1120.
- Updated `xmasked_view` using `xmasked_value` #1074.
- Clean documentation for views #1131.
- Build with `xsimd` on Windows fixed #1127.
- Implement `mime_bundle_repr` for `xmasked_view` #1132.
- Modify shuffle to use identical algorithms for any number of dimensions #1135.
- Warnings removal on windows #1139.
- Add permutation function to `random` #1141.
- `xfunction_iterator` permutation #933.
- Add `bincount` to `xhistogram` #1140.
- Add contiguous iterable base class and remove layout param from storage iterator #1057.
- Add `storage_iterator` to view and strided view #1045.
- Removes `data_element` from `xoptional` #1137.
- `xtensor` default constructor and scalar assign fixed #1148.
- Add `resize` / `reshape` to `xfixed_container` #1147.
- Iterable refactoring #1149.
- `inner_strides_type` imported in `xstrided_view` #1151.

1.2.8 0.17.3

- `xslice` fix #1099.
- added missing `static_layout` in `xmasked_view` #1100.

1.2.9 0.17.2

- Add experimental TBB support for parallelized multicore assign #948.
- Add inline statement to all functions in `xnpy` #1097.
- Fix strided assign for certain assignments #1095.
- CMake, remove `gtest` warnings #1085.
- Add conversion operators to slices #1093.
- Add optimization to unchecked accessors when contiguous layout is known #1060.
- Speedup assign by computing any layout on vectors #1063.
- Skip resizing for fixed shapes #1072.
- Add `xsimd` apply to `xcomplex` functors (`conj`, `norm`, `arg`) #1086.

- Propagate contiguous layout through views #1039.
- Fix C++17 ambiguity for GCC 7 #1081.
- Correct shape type in argmin, fix svector growth #1079.
- Add `interp` function to `xmath` #1071.
- Fix valgrind warnings + memory leak in `xadapt` #1078.
- Remove more clang warnings & errors on OS X #1077.
- Add move constructor from `xtensor` <-> `xarray` #1051.
- Add global support for negative axes in reducers/accumulators allow multiple axes in `average` #1010.
- Fix reference usage in `xio` #1076.
- Remove occurrences of `std::size_t` and `double` #1073.
- Add missing parantheses around min/max for MSVC #1061.

1.2.10 0.17.1

- Add `std` namespace to `size_t` everywhere, remove `std::copysign` for MSVC #1053.
- Fix (wrong) bracket warnings for older clang versions (e.g. clang 5 on OS X) #1050.
- Fix strided view on view by using `std::addressof` #1049.
- Add more `adapt` functions and shorthands #1043.
- Improve CRTP base class detection #1041.
- Fix `rebind` container ambiguous template for C++17 / GCC 8 regression #1038.
- Fix functor return value #1035.

1.2.11 0.17.0

Breaking changes

- Changed strides to `std::ptrdiff_t` #925.
- Renamed `count_nonzeros` in `count_nonzero` #974.
- homogenize `xfixed` constructors #970.
- Improve `random::choice` #1011.

New features

- add `signed char` to `numpy` deserialization format #1017.
- `simd` assignment now requires convertible types instead of same type #1000.
- shared expression and automatic `xclosure` detection #992.
- `average` function #987.
- added `simd` support for complex #985.
- `argsort` function #977.

- propagate fixed shape #922.
- added `xdrop_slice` #972.
- added doc for `xmasked_view` #971.
- added `xmasked_view` #969.
- added `dynamic_view` #966.
- added ability to use negative indices in keep slice #964.
- added an easy way to create lambda expressions, square and cube #961.
- noalias on rvalue #965.

Other changes

- `xshared_expression` fixed #1025.
- fix `make_xshared` #1024.
- add tests to evaluate shared expressions #1019.
- fix `where` on `xview` #1012.
- basic usage replaced with getting started #1004.
- avoided installation failure in absence of `nlohmann_json` #1001.
- code and documentation clean up #998.
- removed g++ “pedantic” compiler warnings #997.
- added missing header in `basic_usage.rst` #996.
- warning pass #990.
- added missing include in `xview` #989.
- added missing `<map>` include #983.
- `xislice` refactoring #962.
- added missing operators to noalias #932.
- cmake fix for Intel compiler on Windows #951.
- fixed `xsimd` abs deduction #946.
- added `islice` example to view doc #940.

1.2.12 0.16.4

- removed usage of `std::transform` in `assign` #868.
- add strided assignment #901.
- `xsimd` activated for conditional ternary functor #903.
- `xstrided_view` `split` #905.
- assigning an expression to a view throws if it has more dimensions #910.
- faster random #913.
- `xoptional_assembly_base` storage type #915.

- new tests and warning pass #916.
- norm immediate reducer #924.
- add `reshape_view` #927.
- fix immediate reducers with 0 strides #935.

1.2.13 0.16.3

- simd on mathematical functions fixed #886.
- `fill` method added to containers #887.
- access with more arguments than dimensions #889.
- unchecked method implemented #890.
- `fill` method implemented in view #893.
- documentation fixed and warnings removed #894.
- negative slices and new range syntax #895.
- `xview_stepper` with implicit `xt::all` bug fix #899.

1.2.14 0.16.2

- Add include of `xview.hpp` in example #884.
- Remove FS identifier #885.

1.2.15 0.16.1

- Workaround for Visual Studio Bug #858.
- Fixup example notebook #861.
- Prevent expansion of `min` and `max` macros on Windows #863.
- Renamed `m_data` to `m_storage` #864.
- Fix regression with respect to random access stepping with views #865.
- Remove use of `CS`, `DS` and `ES` qualifiers for Solaris builds #866.
- Removal of precision type #870.
- Make json tests optional, bump `xtl/xsimd` versions #871.
- Add more benchmarks #876.
- Forbid simd fixed #877.
- Add more asserts #879.
- Add missing `batch_bool` typedef #881.
- `simd_return_type` hack removed #882.
- Removed test guard and fixed dimension check in `xscalar` #883.

1.2.16 0.16.0

Breaking changes

- data renamed in storage, raw_data renamed in data #792.
- Added layout template parameter to xstrided_view #796.
- Remove equality operator from stepper #824.
- dynamic_view renamed in strided_view #832.
- xtensorf renamed in xtensor_fixed #846.

New features

- Added strided view selector #765.
- Added count_nonzeros #781.
- Added implicit conversion to scalar in xview #788.
- Added tracking allocators to xutils.hpp #789.
- xindexslice and shuffle function #804.
- Allow xadapt with dynamic layout #816.
- Added xtensorf initialization from C array #819.
- Added policy to allocation tracking for throw option #820.
- Free function empty for construction from shape #827.
- Support for JSON serialization and deserialization of xtensor expressions #830.
- Add trapz function #837.
- Add diff and trapz (y, x) functions #841.

Other changes

- Added fast path for specific assigns #767.
- Renamed internal macros to prevent collisions #772.
- dynamic_view unwrapping #775.
- xreducer_stepper copy semantic fixed #785.
- xfunction copy constructor fixed #787.
- warnings removed #791.
- xscalar_stepper fixed #802.
- Fixup xadapt on const pointers #809.
- Fix in owning buffer adaptors #810.
- Macros fixup #812.
- More fixes in xadapt #813.
- Mute unused variable warning #815.

- Remove comparison of steppers in assign loop #823.
- Fix reverse iterators #825.
- gcc-8 fix for template method calls #833.
- refactor benchmarks for upcoming release #842.
- `flip` now returns a view #843.
- initial warning pass #850.
- Fix warning on diff function #851.
- `xsimd` assignment fixed #852.

1.2.17 0.15.9

- missing layout method in `xfixed` #777.
- fixed uninitialized backstrides #774.
- update `xtensor-blas` in binder #773.

1.2.18 0.15.8

- comparison operators for slices #770.
- use default-assignable layout for strided views. #769.

1.2.19 0.15.7

- nan related functions #718.
- return types fixed in dynamic view helper #722.
- `xview` on constant expressions #723.
- added decays to make `const value_type` compile #727.
- iterator for constant `strided_view` fixed #729.
- `strided_view` on `xfunction` fixed #732.
- Fixes in `xstrided_view` #736.
- View semantic (broadcast on assign) fixed #742.
- Compilation prevented when using ellipsis with `xview` #743.
- Index of `xiterator` set to shape when reaching the end #744.
- `xscalar` fixed #748.
- Updated README and related projects #749.
- Perfect forwarding in `xfunction` and views #750.
- Missing include in `xassign.hpp` #752.
- More related projects in the README #754.
- Fixed stride computation for `xtensorf` #755.

- Added tests for backstrides #758.
- Clean up `has_raw_data` ins strided view #759.
- Switch to `ptrdiff_t` for slices #760.
- Fixed `xview` strides computation #762.
- Additional methods in slices, required for `xframe` #764.

1.2.20 0.15.6

- `zeros`, `ones`, `full` and `empty_like` functions #686.
- `squeeze` view #687.
- bitwise shift left and shift right #688.
- `ellipsis`, `unique` and `trim` functions #689.
- `xview` iterator benchmark #696.
- optimize stepper increment #697.
- `minmax` reducers #698.
- `where` fix with SIMD #704.
- additional doc for scalars and views #705.
- mixed arithmetic with SIMD #713.
- broadcast fixed #717.

1.2.21 0.15.5

- `assign` functions optimized #650.
- transposed view fixed #652.
- exceptions refactoring #654.
- performances improved #655.
- view data accessor fixed #660.
- new dynamic view using variant #656.
- alignment added to fixed `xtensor` #659.
- code cleanup #664.
- `xtensorf` and new dynamic view documentation #667.
- qualify namespace for `compute_size` #665.
- make `xio` use `dynamic_view` instead of `view` #662.
- transposed view on any expression #671.
- docs typos and grammar plus formatting #676.
- index view test assertion fixed #680.
- `flatten` view #678.
- handle the case of pointers to const element in `xadapt` #679.

- use quotes in `#include` statements for `xtl` #681.
- additional constructors for `svector` #682.
- removed `test_xsemantics.hpp` from test CMakeLists #684.

1.2.22 0.15.4

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

1.2.23 0.15.3

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

1.2.24 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 optimization #645.

1.2.25 0.15.1

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

1.2.26 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.

1.2.27 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.

1.2.28 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.

1.2.29 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.

1.2.30 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.

1.2.31 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 `npz` 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.

1.2.32 0.12.1

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

1.2.33 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.

1.2.34 0.11.3

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

1.2.35 0.11.2

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

1.2.36 0.11.1

- Fixes some warnings in julia and python bindings

1.2.37 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 Getting started

This short guide explains how to get started with *xtensor* once you have installed it with one of the methods described in the installation section.

1.3.1 First example

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

int main(int argc, char* argv[])
{
    xt::xarray<double> arr1
        {{1.0, 2.0, 3.0},
```

(continues on next page)

(continued from previous page)

```
        {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;

    return 0;
}
```

This example simply adds the second row of a 2-dimensional array with a 1-dimensional array.

1.3.2 Compiling the first example

xtensor is a header-only library, so there is no library to link with. The only constraint is that the compiler must be able to find the headers of *xtensor*, this is usually done by having the directory containing the headers in the include path. With GCC, use the `-I` option to achieve this. Assuming the first example code is located in `example.cpp`, the compilation command is:

```
gcc -I /path/to/xtensor/ example.cpp -o example
```

When you run the program, it produces the following output:

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

1.3.3 Building with cmake

A better alternative for building programs using *xtensor* is to use *cmake*, especially if you are developing for several platforms. Assuming the following folder structure:

```
first_example
|- src
|  |- example.cpp
|- CMakeLists.txt
```

The following minimal `CMakeLists.txt` is enough to build the first example:

```
cmake_minimum_required(VERSION 3.1)
project(first_example)

find_package(xtl REQUIRED)
find_package(xtensor REQUIRED)

add_executable(first_example src/example.cpp)
target_link_libraries(first_example xtensor)
```

cmake has to know where to find the headers, this is done through the `CMAKE_INSTALL_PREFIX` variable. Note that `CMAKE_INSTALL_PREFIX` is usually the path to a folder containing the following subfolders: `include`, `lib` and `bin`, so you don't have to pass any additional option for linking. Examples of valid values for `CMAKE_INSTALL_PREFIX` on Unix platforms are `/usr/local`, `/opt`.

The following commands create a directory for building (avoid building in the source folder), builds the first example with cmake and then runs the program:

```
mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=your_prefix ..
make
./first_program
```

1.3.4 Second example: reshape

This second example initializes a 1-dimensional array and reshapes it in-place:

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

int main(int argc, char* argv[])
{
    xt::xarray<int> arr
        {1, 2, 3, 4, 5, 6, 7, 8, 9};

    arr.reshape({3, 3});

    std::cout << arr;
    return 0;
}
```

When compiled and run, this produces the following output:

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

1.3.5 Third example: index access

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

int main(int argc, char* argv[])
{
    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);
    return 0;
}
```

Outputs:

```
1.0
1
```

1.3.6 Fourth example: broadcasting

This last example shows how to broadcast the `xt::pow` universal function:

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

int main(int argc, char* argv[])
{
    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;
    return 0;
}
```

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 an 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"

using array_type = xt::xarray<double>;
using shape_type = array_type::shape_type;
shape_type shape = {3, 2, 4};
array_type a(shape);
size_t d = a.dimension();
const shape_type& 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 checking. 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 a 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>
#include "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, `XTENSOR_DEFAULT_TRAVERSAL` 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 `XTENSOR_DEFAULT_TRAVERSAL` 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 indexing scheme. In such a scheme, the index (i_0, \dots, i_n) corresponds to the offset $\sum(i_k * s_k)$ from the beginning of the one-dimensional buffer, where (s_0, \dots, s_n) 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 specifying any strided index scheme at runtime (including row-major and column-major schemes), while `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 `XTENSOR_DEFAULT_LAYOUT`.
- at runtime if the previous template parameter was set to `layout_type::dynamic`. In that case, `resize` and constructor overloads allow specifying 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 `XTENSOR_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, xt::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 following 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, xt::layout_type::dynamic> a(shape, xt::layout::row_major);
```

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

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

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

However, in the 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

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

- `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.

- `xtensor_fixed` has a shape fixed at compile time. This allows even more optimizations, such as allocating the storage for the container on the stack, as well as computing strides and backstrides at compile time, making the allocation of this container extremely cheap.

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> a(shape);
// this is equivalent to
// xt::xtensor<double, 3, xt::layout_type::row_major> a(shape);
```

Or when using `xtensor_fixed`:

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

xt::xtensor_fixed<double, xt::xshape<3, 2, 4>> a();
// or xt::xtensor_fixed<double, xt::xshape<3, 2, 4>, xt::layout_type::row_major>()
```

`xarray`, `xtensor` and `xtensor_fixed` containers are all `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 `xarray`, `xtensor` and `xtensor_fixed`.
- `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.
- `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 expensive 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 the 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 $\text{new_b}(0, i, j) == \text{old_b}(i, j)$ for (i, j) in $[0, 1] \times [0, 3]$. After the resize of b , $a(0, i, j) + b(0, i, j)$ 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 Scalars and 0-D expressions

1.6.1 Assignment

In *xtensor*, scalars are handled as if they were 0-dimensional expressions. This means that when assigning a scalar value to an *xarray*, this last one is **not filled** with that value, but resized to become a 0-D array containing the scalar value:

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

xt::xarray<double> a = {{0., 1., 2.}, {3., 4., 5.}};
double s = 1.2;
a = s;
std::cout << a << std::endl;
// prints 1.2
```

While this may look weird and counter-intuitive, this actually ensures full consistency of the expression system. The easiest way to illustrate this is to assume that we have the intuitive scalar assignment (i.e. a broadcasting assignment) and see how it breaks consistency.

1.6.2 Copy semantic consistency

Assuming that the scalar assignment does not resize the array, we have the following behavior:

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

xt::xarray<double> a = {{0., 1., 2.}, {3., 4., 5.}};
double s = 1.2;
a = 1.2;
std::cout << a << std::endl;
// prints {{1.2, 1.2, 1.2}, {1.2, 1.2, 1.2}}
```

This is not consistent with the behavior of the copy constructor from a scalar:

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

xt::xarray<double> a(1.2);
std::cout << a << std::endl;
// prints 1.2 (a is a 0-D array)
```

A way to fix this is to disable copy construction from scalar, and provide a constructor taking a shape and a scalar:

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

xt::xarray<double> a = {{0., 1., 2.}, {3., 4., 5.}};
a = 1.2;
xt::xarray<double> b({2, 3}, 1.2);
```

Although this looks like an acceptable solution, it actually breaks consistency between scalars and 0-dimensional expressions. This may lead to vicious bugs as explained in the next section.

1.6.3 Scalar and 0-D expressions

Assume that you need a function that computes the mean of the elements of an expression and stores it in another expression. A possible implementation is:

```
template <class E1, class E2>
void eval_mean(const E1& e1, E2& e2)
{
    e2 = sum(e1) / e1.size();
}
```

Then, somewhere in your program:

```
// somewhere in the code
xarray<double> a = {{1., 2., 3.}, {4., 5., 6.}},
xarray<double> b = a;
// ...
// later
eval_mean(a, b);
// Now b is a 0-D container holding 21.
```

After that, `b` is a 0-dimensional array containing the mean of the elements of `a`. Indeed, `sum(a) / e1.size()` is a 0-D expression, thus when assigned to `b`, this latter is resized. Later, you realize that you also need the sum of the elements of `a`. Since the `eval_mean` function already computes it, you decide to return it from that function:

```
template <class E1, class E2>
double eval_mean(const E1& e1, E2& e2)
{
    double s = sum(e1)();
    e2 = s / e1.size();
    return s;
}
```

And then you change the client code:

```
// somewhere in the code
xarray<double> a = {{1., 2., 3.}, {4., 5., 6.}},
xarray<double> b = a;
// ...
// later
double s = eval_mean(a, b);
// Now b is a 2-D container!
```

After that, `b` has become a 2-dimensional array! Indeed, since assigning a scalar to an expression does not resize it, the change in `eval_mean` implementation now assigns the mean of `a` to each elements of `b`.

This simple example shows that without consistency between scalars and 0-D expressions, refactoring the code to cache the result of some 0-D computation actually *silently* changes the shape of the expressions that this result is assigned to.

The only way to avoid that behavior and the bugs it leads to is to handle scalars as if they were 0-dimensional expressions.

1.7 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.

1.7.1 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.}};

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. }
```

1.7.2 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 }
}

int main()
{
    std::size_t size = 2;
```

(continues on next page)

```

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 <cstdint>
#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
}

```

1.8 Operators and functions

1.8.1 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.

```
#include "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}}
```

1.8.2 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 elements of e are truthy, false otherwise.

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

- where(E&& b, E1&& e1, 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*.

1.8.3 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 }
```

1.8.4 Bitwise operators

xtensor also contains the following bitwise operators:

- Bitwise and: `operator&`
- Bitwise or: `operator|`
- Bitwise xor: `operator^`
- Bitwise not: `operator~`
- Bitwise left/right shift: `left_shift`, `right_shift`

1.8.5 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`, `cbprt`, ...
- 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.

1.8.6 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 }
```

1.8.7 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});
```

1.8.8 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);
// => res.shape() = {5, 8, 3};
// => res(0, 0, 0) = 1
// => res(0, 7, 0) = 8
```

You can also call the `accumulate` 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);
```

1.8.9 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:

```
#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.

1.8.10 Universal functions and vectorization

xtensor provides utilities to **vectorize any scalar function** (taking multiple scalar arguments) into a function that will perform on *xexpressions*, applying the lazy broadcasting rules which we described in a previous section. These functions are called *xfunctions*. 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 *xfunctions*.

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

```

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

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

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

```

1.9 Histogram

1.9.1 Basic usage

Note:

```

xt::histogram(a, bins[, weights][, density])
xt::histogram_bin_edges(a[, weights][, left, right][, bins][, mode])

```

Any of the options [...] can be omitted (though the order must be preserved). The defaults are:

- `weights = xt::ones(data.shape())`
- `density = false`
- `left = xt::amin(data) (0)`
- `right = xt::amax(data) (0)`
- `bins = 10`
- `mode = xt::histogram::automatic`

The behavior, in-, and output of `histogram` is similar to that of `numpy.histogram` with that difference that the bin-edges are obtained by a separate function call:

```

#include <xtensor/xtensor.hpp>
#include <xtensor/xhistogram.hpp>
#include <xtensor/xio.hpp>

int main()
{
    xt::xtensor<double,1> data = {1., 1., 2., 2., 3.};

    xt::xtensor<double,1> count = xt::histogram(data, std::size_t(2));

    xt::xtensor<double,1> bin_edges = xt::histogram_bin_edges(data, std::size_t(2));

    return 0;
}

```

1.9.2 Bin-edges algorithm

To customize the algorithm to be used to construct the histogram, one needs to make use of the latter `histogram_bin_edges`. For example:

```
#include <xtensor/xtensor.hpp>
#include <xtensor/xhistogram.hpp>
#include <xtensor/xio.hpp>

int main()
{
    xt::xtensor<double,1> data = {1., 1., 2., 2., 3.};

    xt::xtensor<double,1> bin_edges = xt::histogram_bin_edges(data, std::size_t(2),
↳xt::histogram_algorithm::uniform);

    xt::xtensor<double,1> prob = xt::histogram(data, bin_edges, true);

    std::cout << bin_edges << std::endl;
    std::cout << prob << std::endl;

    return 0;
}
```

The following algorithms are available:

- `automatic`: equivalent to `linspace`.
- `linspace`: linearly spaced bin-edges.
- `logspace`: bins that logarithmically increase in size.
- `uniform`: bin-edges such that the number of data-points is the same in all bins (as much as possible).

1.10 Views

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

1.10.1 Sliced views

Sliced views consist of the combination of the `xexpression` to adapt, and a list of `slice` 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 through 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
- `keep(i0, i1, i2, ...)` a slice selecting non-contiguous indices to keep on the underlying expression

- `drop(i0, i1, i2, ...)` a slice selecting non-contiguous indices to drop on 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);

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

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

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

// View with non contiguous slices
auto v4 = xt::view(a, xt::drop(0), xt::all(), xt::keep(0, 3));
// => v4.shape() = { 2, 2, 2 }
// => v4(0, 0, 0) = a(1, 0, 0)
// => v4(1, 1, 1) = a(2, 1, 3)
```

The range function supports the placeholder `_` syntax:

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

using namespace xt::placeholders; // required for `_` to work

auto a = xt::xarray<int>::from_shape({3, 2, 4});
auto v1 = xt::view(a, xt::range(_, 2), xt::all(), xt::range(1, _));
// The previous line is equivalent to
auto v2 = xt::view(a, xt::range(0, 2), xt::all(), xt::range(1, 4));
```

`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
```

1.10.2 Strided views

While the `xt::view` is a compile-time static expression, `xtensor` also contains a dynamic strided view in `xstrided_view.hpp`. The strided 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. Note that the slice vector is actually a type-alias for a `std::vector` of a variant for all the slice types. The strided view does not support the slices returned by the `keep` and `drop` functions.

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

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

xt::xstrided_slice_vector sv({xt::range(0, 1), xt::newaxis()});
sv.push_back(1);
sv.push_back(xt::all());

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

// Equivalent but shorter
auto v2 = xt::strided_view(a, { xt::range(0, 1), xt::newaxis(), 1, xt::all() });
// v2 == v1

// ILLEGAL:
auto v2 = xt::strided_view(a, { xt::all(), xt::all(), xt::all(), xt::keep(0, 3),
↪xt::drop(1, 4) });
// xt::drop and xt::keep are not supported with strided views
```

Since `xtensor 0.16.3`, a new range syntax can be used with strided views:

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

using namespace xt::placeholders;

auto a = xt::xarray<int>::from_shape({3, 2, 3, 4, 5});
auto v1 = xt::strided_view(a, {_r|0|1, 1, _r|_|2, _r|_|-1});
// The previous line is equivalent to
auto v2 = xt::strided_view(a, {xt::range(0, 1), 1, xt::range(_, 2), xt::range(_, _, -
↪1)});
```

The `xstrided_view` is very efficient on contiguous memory (e.g. `xtensor` or `xarray`) but less efficient on `xexpressions`.

1.10.3 Transposed views

`xtensor` provides a lazy transposed view on any expression, whose layout is either row major order or column major order. Trying to build a transposed view on a expression with a dynamic layout throws an exception.

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

xt::xarray<int> a = { {0, 1, 2}, {3, 4, 5} };
auto tr = xt::transpose(a);
// tr == { {0, 3}, {1, 4}, {2, 5} }
```

(continues on next page)

(continued from previous page)

```
xt::xarray<int, layout_type::dynamic> b = { {0, 1, 2}, {3, 4, 5} };
auto tr2 = xt::transpose(b);
// => throw transpose_error
```

Like the strided view, the transposed view is built upon the `xstrided_view`.

1.10.4 Flatten views

It is sometimes useful to have a one-dimensional view of all the elements of an expression. `xtensor` provides two functions for that, `ravel` and `flatten`. The former one let you specify the order used to read the elements while the latter one uses the layout of the expression.

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

xt::xarray<int> a = { {0, 1, 2}, {3, 4, 5} };
auto flc = xt::ravel<layout_type::column_major>(a);
std::cout << flc << std::endl;
// => prints { 0, 3, 1, 4, 2, 5 }

auto fl = xt::flatten(a);
std::cout << fl << std::endl;
// => prints { 0, 1, 2, 3, 4, 5 }
```

Like the strided view and the transposed view, the flatten view is built upon the `xstrided_view`.

1.10.5 Reshape views

The reshape view allows to handle an expression as if it was given a new shape, however no additional memory allocation occurs, the original expression keeps its shape. Like any view, the underlying expression is not copied, thus assigning a value through the view modifies the underlying expression.

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

auto a = xt::xarray<int>::from_shape({3, 2, 4});
auto v = xt::reshape_view(a, { 4, 2, 3 });
// a(0, 0, 3) == v(0, 1, 0)
// a(0, 1, 0) == v(0, 1, 1)

v(0, 2, 0) = 4;
// a(0, 1, 2) == 4
```

Like the strided view and the transposed view, the reshape view is built upon the `xstrided_view`.

1.10.6 Dynamic views

The dynamic view is like the strided view, but with support of the slices returned by the `keep` and `drop` functions. However, this support has a cost and the dynamic view is slower than the strided view, even when no keeping or dropping slice is involved.

```

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

auto a = xt::xarray<int>::from_shape({3, 2, 3, 4, 5});
xt::xdynamic_slice_vector sv({xt::range(0, 1), xt::newaxis()});
sv.push_back(1);
sv.push_back(xt::all());
sv.push_back(xt::keep(0, 2, 3));
sv.push_back(xt::drop(1, 2, 4));

auto v1 = xt::dynamic_view(a, sv);

// Equivalent but shorter
auto v2 = xt::dynamic_view(a, { xt::range(0, 1), xt::newaxis(), 1, xt::all(),
↪xt::keep(0, 2, 3), xt::drop(1, 2, 4) });
// v2 == v1

```

1.10.7 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 }
b += 100;
// => a = {{101, 5, 3}, {104, 105, 6}}

```

1.10.8 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}}

```

1.10.9 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}}
```

1.10.10 Masked view

Masked views are multidimensional views that apply a mask on an `xexpression`.

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

xt::xarray<double> a = {{1, 5, 3}, {4, 5, 6}};
xt::xarray<bool> mask = {{true, false, false}, {false, true, false}};

auto m = xt::masked_view(a, mask);
// => m = {{1, masked, masked}, {masked, 5, masked}}

m += 100;
// => a = {{101, 5, 3}, {4, 105, 6}}
```

1.10.11 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)
```

1.10.12 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 a 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}};
```

1.10.13 Assigning to a view

When assigning an expression `rhs` to a container such as `xarray`, this last one is resized so its shape is the same as the one of `RHS`. However, since views *cannot be resized*, when assigning an expression to a view, broadcasting rules are applied:

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

xarray<double> a = {{0., 1., 2.}, {3., 4., 5.}};
double b = 1.2;
auto tr = view(a, 0, all());
tr = b;
// => a = {{1.2, 1.2, 1.2}, {3., 4., 5.}}
```

1.11 Indices

1.11.1 Definition

There are two types of indices: *array indices* and *flat indices*. Consider this example (stored in row-major):

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

int main()
{
    xt::xarray<size_t> a = xt::arange<size_t>(3 * 4);

    a.reshape({3,4});

    std::cout << a << std::endl;
}
```

Which prints

```
{{ 0, 1, 2, 3},
 { 4, 5, 6, 7},
 { 8, 9, 10, 11}}
```

The array index {1, 2} corresponds to the flat index 6.

1.11.2 Array indices

Functions like `xt::argwhere(a < 5)` return a `std::vector` of array indices. Using the same matrix as above, we can do

```
int main()
{
    xt::xarray<size_t> a = xt::arange<size_t>(3 * 4);

    a.reshape({3,4});

    auto idx = xt::from_indices(xt::argwhere(a >= 6));

    std::cout << idx << std::endl;
}
```

which prints

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

Here we observe that to work print we need to convert the `std::vector` to a `xt::xtensor<size_t, 2>` array, which is done using `xt::from_indices`.

1.11.3 From array indices to flat indices

To convert the array indices to a `xt::xtensor<size_t, 1>` of flat indices, `xt::ravel_indices` can be used. For to same example:

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

int main()
{
    xt::xarray<size_t> a = xt::arange<size_t>(3 * 4);

    a.reshape({3,4});

    auto idx = xt::ravel_indices(xt::argwhere(a >= 6), a.shape());

    std::cout << idx << std::endl;
}
```

which prints

```
{ 6, 7, 8, 9, 10, 11}
```

Note: To convert to a `std::vector` use

```
auto idx = xt::ravel_indices<xt::ravel_vector_tag>(xt::argwhere(a >= 6), a.shape());
```

1.11.4 1-D arrays: array indices == flat indices

For 1-D arrays the array indices and flat indices coincide. One can use the generic functions `xt::flatten_indices` to get a `xt::xtensor<size_t, 1>` of (array/flat) indices. For example:

```
#include <xtensor/xtensor.hpp>
#include <xtensor/xview.hpp>
#include <xtensor/xio.hpp>

int main()
{
    xt::xtensor<size_t, 1> a = xt::arange<size_t>(16);

    auto idx = xt::flatten_indices(xt::argwhere(a >= 6));

    std::cout << idx << std::endl;

    std::cout << xt::view(a, xt::keep(idx)) << std::endl;
}
```

which print the indices and the selection (which are in this case identical):

```
{ 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}
{ 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}
```

1.11.5 From flat indices to array indices

To convert *flat indices* to *array indices* the function `xt::ravel_indices` can be used. For example

```
#include <xtensor/xarray.hpp>
#include <xtensor/xtensor.hpp>
#include <xtensor/xstrides.hpp>
#include <xtensor/xio.hpp>

int main()
{
    xt::xarray<size_t> a = xt::arange<size_t>(3 * 4);

    a.reshape({3,4});

    auto flat_indices = xt::ravel_indices(xt::argwhere(a >= 6), a.shape());

    auto array_indices = xt::from_indices(xt::unravel_indices(flat_indices, a.
↪shape()));

    std::cout << "flat_indices = " << std::endl << flat_indices << std::endl;
    std::cout << "array_indices = " << std::endl << array_indices << std::endl;
}
```

which prints

```

flat_indices =
{ 6, 7, 8, 9, 10, 11}
array_indices =
{{1, 2},
 {1, 3},
 {2, 0},
 {2, 1},
 {2, 2},
 {2, 3}}

```

Notice that once again the function `xt::from_indices` has been used to convert a `std::vector` of indices to a `xt::xtensor` array for printing.

1.12 Expression builders

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

1.12.1 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 an expression with ones on the k-th diagonal.

1.12.2 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

1.12.3 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.

1.12.4 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.

1.12.5 Meshes

- `meshgrid(x1, x2, ...)`: generates N-D coordinate expressions given one-dimensional coordinate arrays `x1, x2, ...`. If specified vectors have lengths $N_i = \text{len}(x_i)$, `meshgrid` returns $(N_1, N_2, N_3, \dots, N_n)$ -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.

1.13 Missing values

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

1.13.1 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.

1.13.2 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>() }};
```

(continues on next page)

(continued from previous page)

```
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}}
```

1.13.3 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, hv);
std::cout << assembly << std::endl;
```

outputs:

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

1.13.4 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 } }
```

1.14 File input and output

xtensor has some built-in mechanisms to make loading and saving data easy. The base xtensor package allows to save and load data in the .csv, .json and .npy format. Please note that many more input and output formats are available in the xtensor-io package. xtensor-io offers functions to load and store from image files (jpg, gif, png...), sound files (wav, ogg...), HDF5 files (h5, hdf5, ...), and compressed numpy format (npz).

1.14.1 Loading CSV data into xtensor

The following example code demonstrates how to use load_csv and dump_csv to load and save data in the Comma-separated value format. The reference documentation is [xcsv: read/write CSV files](#).

```
#include <istream>  
#include <fstream>  
#include <iostream>  
  
#include "xtensor/xarray.hpp"  
#include "xtensor/xcsv.hpp"  
  
int main()  
{  
    ifstream in_file;  
    in_file.open("in.csv");  
    auto data = xt::load_csv<double>(in_file);  
  
    ofstream out_file;  
    out_file("out.csv");  
  
    xt::xarray<double> a = {{1,2,3,4}, {5,6,7,8}};  
    xt::dump_csv(out_file, a);  
  
    return 0;  
}
```

1.14.2 Loading NPY data into xtensor

The following example demonstrates how to load and store xtensor data in the npy “NumPy” format, using the load_npy and dump_npy functions. Reference documentation for the functions used is found here [xnpz: read/write NPY files](#).

```
#include <istream>  
#include <iostream>  
#include <fstream>  
  
#include "xtensor/xarray.hpp"  
#include "xtensor/xnpz.hpp"  
  
int main()  
{
```

(continues on next page)

(continued from previous page)

```

{
    // Note: you need to supply the data type you are loading
    //       in this case "double".
    auto data = xt::load_npy<double>("in.npy");

    xt::xarray<double> a = {{1,2,3,4}, {5,6,7,8}};
    xt::dump_npy("out.npy", a);

    return 0;
}

```

1.14.3 Loading JSON data into xtensor

It's possible to load and dump data to json, using the json library written by nlohmann (<https://nlohmann.github.io/json/>) which offers a convenient way to handle json data in C++. Note that the library needs to be separately installed. The reference documentation is found *xjson: serialize to/from JSON*.

```

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

int main()
{
    xt::xarray<double> t = {{{1, 2},
                           {3, 4}},
                          {{1, 2},
                           {3, 4}}};

    nlohmann::json jl = t;
    // To obtain the json serialized string
    std::string s = jl.dump();

    xt::xarray<double> res;
    auto j = "[[10.0,10.0],[10.0,10.0]]"_json;
    from_json(j, res);
}

```

1.15 Expressions and semantic

1.15.1 xexpression

Defined in `xtensor/xexpression.hpp`

```

template <class D>
class xexpression

```

Base class for xexpressions.

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

Template Parameters

- E: The derived type.

Subclassed by `xt::xsemantic_base< D >`

Downcast functions

auto **derived_cast** ()

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

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

auto **derived_cast** () **const**

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

template <class E>

class **xshared_expression** : **public** xt::xexpression<xshared_expression<E>>

Shared xexpressions.

Due to C++ lifetime constraints it's sometimes necessary to create shared expressions (akin to a shared pointer).

For example, when a temporary expression needs to be used twice in another expression, shared expressions can come to the rescue:

```
template <class E>
auto cos_plus_sin(xexpression<E>&& expr)
{
    // THIS IS WRONG: forwarding rvalue twice not permitted!
    // return xt::sin(std::forward<E>(expr)) + xt::cos(std::forward<E>(expr));
    // THIS IS WRONG TOO: because second `expr` is taken as reference (which will
    ↳be invalid)
    // return xt::sin(std::forward<E>(expr)) + xt::cos(expr)
    auto shared_expr = xt::make_xshared(std::forward<E>(expr));
    auto result = xt::sin(shared_expr) + xt::cos(shared_expr);
    std::cout << shared_expr.use_count() << std::endl; // Will print 3 because
    ↳used twice in expression
    return result; // all valid because expr lifetime managed by xshared_
    ↳expression / shared_ptr.
}
```

Public Functions

xshared_expression (std::shared_ptr<E> &&ptr)

Constructor for xshared expression (note: usually the free function `make_xshared` is recommended).

See `make_xshared`

Parameters

- ptr: shared ptr that contains the expression

long **use_count** () **const**

Return the number of times this expression is referenced.

Internally calls the `use_count()` function of the `std::shared_ptr`.

template <class E>

auto `xt::make_xshared` (*xexpression*<E> &&*expr*)
 Helper function to create shared expression from any *xexpression*.

Return *xshared* expression

Parameters

- *expr*: rvalue expression that will be shared

1.15.2 *xsemantic_base*

Defined in `xtensor/xsemantic.hpp`

```
template <class D>
class xsemantic_base : public xt::xexpression<D>
    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.

Subclassed by `xt::xcontainer_semantic< D >`, `xt::xview_semantic< D >`

Computed assignement

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

Return a reference to **this*.

Parameters

- *e*: the scalar to add.

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

Return a reference to **this*.

Parameters

- *e*: the scalar to subtract.

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

Return a reference to **this*.

Parameters

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

```
template <class E>
```

auto **operator/=** (**const** E &e)
Divides *this by the scalar e.

Return a reference to *this.

Parameters

- e: the scalar involved in the operation.

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

Return a reference to *this.

Parameters

- e: the xexpression to add.

template <**class** E>

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

Return a reference to **this*.

Parameters

- *e*: the *xexpression* to subtract.

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

Return a reference to **this*.

Parameters

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

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

Return a reference to **this*.

Parameters

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

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

Ensures no temporary will be used to perform the assignment.

Return a reference to *this.

Parameters

- e: the xexpression to assign.

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

Ensures no temporary will be used to perform the assignment.

Return a reference to *this.

Parameters

- e: the xexpression to add.

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

Ensures no temporary will be used to perform the assignment.

Return a reference to *this.

Parameters

- e: the xexpression to subtract.

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

Ensures no temporary will be used to perform the assignment.

Return a reference to *this.

Parameters

- e: the xexpression involved in the operation.

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

Ensures no temporary will be used to perform the assignment.

Return a reference to `*this`.

Parameters

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

```
template <class E>
```

```
auto 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.

```
template <class E>
```

```
auto bit_and_assign (const xexpression<E> &e)
```

Computes the bitwise and of `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 bit_or_assign (const xexpression<E> &e)
```

Computes the bitwise or of `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 bit_xor_assign (const xexpression<E> &e)
```

Computes the bitwise xor of `e` to `*this`.

Ensures no temporary will be used to perform the assignment.

Return a reference to `*this`.

Parameters

- `e`: the xexpression to add.

1.15.3 xcontainer_semantic

Defined in `xtensor/xsemantic.hpp`

```
template <class D>
```

```
class xcontainer_semantic : public xt::xsemantic_base<D>
```

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

Assign functions

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

Assigns the temporary tmp to *this.

Return a reference to *this.

Parameters

- tmp: the temporary to assign.

1.15.4 xview_semantic

Defined in `xtensor/xsemantic.hpp`

template <class D>

class **xview_semantic** : public xt::xsemantic_base<D>

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

Assign functions

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

Assigns the temporary tmp to *this.

Return a reference to *this.

Parameters

- tmp: the temporary to assign.

1.15.5 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

1.16 Containers and views

1.16.1 layout

Defined in `xtensor/xlayout.hpp`

enum `xt::layout_type`

layout_type enum for xcontainer based xexpressions

Values:

dynamic = 0x00

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

any = 0xFF

layout_type compatible with all others

row_major = 0x01

row major layout_type

column_major = 0x02

column major layout_type

template `<class... Args>`

constexpr `layout_type xt::compute_layout(Args... args)`

Implementation of the following logical table:

```

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

```

Using bitmasks to avoid nested if-else statements.

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

Parameters

- `args`: the input layouts.

1.16.2 xcontainer

Defined in `xtensor/xcontainer.hpp`

template `<class D>`

class `xcontainer` : **public** `xt::xcontiguous_iterable<D>`

Base class for dense multidimensional containers.

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

Template Parameters

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

Subclassed by `xt::xstrided_container< D >`

Size and shape

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

Returns the number of element in the container.

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

Returns the number of dimensions of the container.

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

Returns the shape of the container.

constexpr auto **strides** () **const**

Returns the strides of the container.

constexpr auto **backstrides** () **const**

Returns the backstrides of the container.

Data

template <class T>

void **fill** (const T &value)

Fills the container with the given value.

Parameters

- `value`: the value to fill the container with.

auto **storage** ()

Returns a reference to the buffer containing the elements of the container.

auto **storage** () **const**

Returns a constant reference to the buffer containing the elements of the container.

auto **data** ()

Returns a pointer to the underlying array serving as element storage.

The pointer is such that range [`data()`; `data() + size()`] is always a valid range, even if the container is empty (`data()` is not is not dereferenceable in that case)

auto **data** () **const**

Returns a constant pointer to the underlying array serving as element storage.

The pointer is such that range [`data()`; `data() + size()`] is always a valid range, even if the container is empty (`data()` is not is not dereferenceable in that case)

auto **data_offset** () **const**

Returns the offset to the first element in the container.

template <class... Args>

auto **operator** () (Args... args)

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

Parameters

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

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

Parameters

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

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

Warning This method is meant for performance, for expressions with a dynamic number of dimensions (i.e. not known at compile time). Since it may have undefined behavior (see parameters), `operator()` should be preferred whenever it is possible.

Warning This method is NOT compatible with broadcasting, meaning the following code has undefined behavior:

```
xt::xarray<double> a = {{0, 1}, {2, 3}};
xt::xarray<double> b = {0, 1};
auto fd = a + b;
double res = fd.unchecked(0, 1);
```

Parameters

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

template <class... *Args*>

auto **unchecked** (*Args... args*) **const**

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

Warning This method is meant for performance, for expressions with a dynamic number of dimensions (i.e. not known at compile time). Since it may have undefined behavior (see parameters), `operator()` should be preferred whenever it is possible.

Warning This method is NOT compatible with broadcasting, meaning the following code has undefined behavior:

```
xt::xarray<double> a = {{0, 1}, {2, 3}};
xt::xarray<double> b = {0, 1};
auto fd = a + b;
double res = fd.unchekek(0, 1);
```

Parameters

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

template <class *S*>

auto **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 **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 **element** (*It first*, *It last*)

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

Parameters

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

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

Parameters

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

Broadcasting

```
template <class S>
bool broadcast_shape (S &shape, 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
- `reuse_cache`: parameter for internal optimization

```
template <class S>
bool has_linear_assign (const S &strides) const
    Checks whether the xcontainer can be linearly assigned to an expression with the specified strides.
```

Return a boolean indicating whether a linear assign is possible

1.16.3 xstrided_container

Defined in `xtensor/xcontainer.hpp`

```
template <class D>
class xstrided_container : public xt::xcontainer<D>
    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 implementation of `xcontainer`.

Public Functions

```
template <class S = shape_type>
void 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 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 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 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 XTENSOR_DEFAULT_LAYOUT)

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

Return `layout_type` of the container

1.16.4 xiterable

Defined in `xtensor/xiterable.hpp`

```
template <class D>
class xconst_iterable
    Base class for multidimensional iterable constant expressions.
```

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

Template Parameters

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

Subclassed by `xt::xiterable< D >`

Constant iterators

template <layout_type L>

auto **begin** () **const**

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

Template Parameters

- L: order used for the traversal. Default value is XTENSOR_DEFAULT_TRAVERSAL.

template <layout_type L>

auto **end** () **const**

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

Template Parameters

- L: order used for the traversal. Default value is XTENSOR_DEFAULT_TRAVERSAL.

template <layout_type L>

auto **cbegin** () **const**

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

Template Parameters

- L: order used for the traversal. Default value is XTENSOR_DEFAULT_TRAVERSAL.

template <layout_type L>

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

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

Template Parameters

- L: order used for the traversal. Default value is XTENSOR_DEFAULT_TRAVERSAL.

Constant reverse iterators

template <layout_type L>

auto **rbegin** () **const**

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

Template Parameters

- L: order used for the traversal. Default value is XTENSOR_DEFAULT_TRAVERSAL.

template <layout_type L>

auto **rend** () **const**

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

Template Parameters

- L: order used for the traversal. Default value is XTENSOR_DEFAULT_TRAVERSAL.

template <layout_type L>

auto **crbegin** () **const**

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

Template Parameters

- L: order used for the traversal. Default value is XTENSOR_DEFAULT_TRAVERSAL.

template <layout_type L>

auto **crend** () **const**

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

Template Parameters

- L: order used for the traversal. Default value is XTENSOR_DEFAULT_TRAVERSAL.

Constant broadcast iterators

template <layout_type L, class S>

auto **begin** (const S &shape) **const**

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

The iteration is broadcasted to the specified shape.

Parameters

- shape: the shape used for broadcasting

Template Parameters

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

template <layout_type L, class S>

auto **end** (const S &shape) **const**

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

The iteration is broadcasted to the specified shape.

Parameters

- shape: the shape used for broadcasting

Template Parameters

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

template <layout_type L, class S>

auto **cbegin** (const S &shape) **const**

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

The iteration is broadcasted to the specified shape.

Parameters

- shape: the shape used for broadcasting

Template Parameters

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

template <layout_type L, class S>

auto **cend** (**const** S &shape) **const**

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

The iteration is broadcasted to the specified shape.

Parameters

- shape: the shape used for broadcasting

Template Parameters

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

Constant reverse broadcast iterators

template <layout_type L, class S>

auto **rbegin** (**const** S &shape) **const**

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

The iteration is broadcasted to the specified shape.

Parameters

- shape: the shape used for broadcasting

Template Parameters

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

template <layout_type L, class S>

auto **rend** (**const** S &shape) **const**

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

The iteration is broadcasted to the specified shape.

Parameters

- shape: the shape used for broadcasting

Template Parameters

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

template <layout_type L, class S>

auto **crbegin** (**const** S &shape) **const**

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

The iteration is broadcasted to the specified shape.

Parameters

- shape: the shape used for broadcasting

Template Parameters

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

template <layout_type L, class S>

auto **crend** (**const** S &shape) **const**
 Returns a constant iterator to the element following the last element of the reversed expression.
 The iteration is broadcasted to the specified shape.

Parameters

- shape: the shape used for broadcasting

Template Parameters

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

```
template <class D>
class xiterable : public xt::xconst_iterable<D>
  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.

Subclassed by `xt::xcontiguous_iterable< D >`, `xt::xmasked_view< CTD, CTM >`,
`xt::xoptional_assembly_base< D >`

Iterators

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

Template Parameters

- L: order used for the traversal. Default value is XTENSOR_DEFAULT_TRAVERSAL.

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

Template Parameters

- L: order used for the traversal. Default value is XTENSOR_DEFAULT_TRAVERSAL.

Broadcast iterators

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

  The iteration is broadcasted to the specified shape.
```

Parameters

- shape: the shape used for broadcasting

Template Parameters

- S: type of the shape parameter.

- L: order used for the traversal. Default value is XTENSOR_DEFAULT_TRAVERSAL.

template <layout_type L, class S>

auto **end** (const S &shape)

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

The iteration is broadcasted to the specified shape.

Parameters

- shape: the shape used for broadcasting

Template Parameters

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

Reverse iterators

template <layout_type L>

auto **rbegin** ()

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

Template Parameters

- L: order used for the traversal. Default value is XTENSOR_DEFAULT_TRAVERSAL.

template <layout_type L>

auto **rend** ()

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

Template Parameters

- L: order used for the traversal. Default value is XTENSOR_DEFAULT_TRAVERSAL.

Reverse broadcast iterators

template <layout_type L, class S>

auto **rbegin** (const S &shape)

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

The iteration is broadcasted to the specified shape.

Parameters

- shape: the shape used for broadcasting

Template Parameters

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

template <layout_type L, class S>

auto **rend** (const S &shape)

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

The iteration is broadcasted to the specified shape.

Parameters

- `shape`: the shape used for broadcasting

Template Parameters

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

```
template <class D>
class xcontiguous_iterable : xt::xiterable<D>
    Base class for multidimensional iterable expressions with contiguous storage.
```

The `xcontiguous_iterable` class defines the interface for multidimensional expressions with contiguous that can be iterated.

Template Parameters

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

Subclassed by `xt::xcontainer< xarray_adaptor< EC, L, SC, Tag > >`, `xt::xcontainer< xarray_container< EC, L, SC, Tag > >`, `xt::xcontainer< xfixed_adaptor< EC, S, L, Tag > >`, `xt::xcontainer< xfixed_container< ET, S, L, Tag > >`, `xt::xcontainer< xtensor_adaptor< EC, N, L, Tag > >`, `xt::xcontainer< xtensor_container< EC, N, L, Tag > >`, `xt::xcontainer< D >`

Iterators

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

Template Parameters

- `L`: order used for the traversal. Default value is `XTENSOR_DEFAULT_TRAVERSAL`.

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

Template Parameters

- `L`: order used for the traversal. Default value is `XTENSOR_DEFAULT_TRAVERSAL`.

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

Template Parameters

- `L`: order used for the traversal. Default value is `XTENSOR_DEFAULT_TRAVERSAL`.

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

Template Parameters

- `L`: order used for the traversal. Default value is `XTENSOR_DEFAULT_TRAVERSAL`.

template <layout_type L>

auto **cbegin** () **const**

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

Template Parameters

- L: order used for the traversal. Default value is XTENSOR_DEFAULT_TRAVERSAL.

template <layout_type L>

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

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

Template Parameters

- L: order used for the traversal. Default value is XTENSOR_DEFAULT_TRAVERSAL.

Broadcast iterators

template <layout_type L, class S>

auto **begin** (const S &shape)

Returns an iterator to the first element of the expression.

The iteration is broadcasted to the specified shape.

Parameters

- shape: the shape used for broadcasting

Template Parameters

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

template <layout_type L, class S>

auto **end** (const S &shape)

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

The iteration is broadcasted to the specified shape.

Parameters

- shape: the shape used for broadcasting

Template Parameters

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

template <layout_type L, class S>

auto **begin** (const S &shape) **const**

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

The iteration is broadcasted to the specified shape.

Parameters

- shape: the shape used for broadcasting

Template Parameters

- S: type of the shape parameter.

- L: order used for the traversal. Default value is XTENSOR_DEFAULT_TRAVERSAL.

```
template <layout_type L, class S>
```

```
auto end(const S &shape) const
```

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

The iteration is broadcasted to the specified shape.

Parameters

- shape: the shape used for broadcasting

Template Parameters

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

```
template <layout_type L, class S>
```

```
auto cbegin(const S &shape) const
```

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

The iteration is broadcasted to the specified shape.

Parameters

- shape: the shape used for broadcasting

Template Parameters

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

```
template <layout_type L, class S>
```

```
auto cend(const S &shape) const
```

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

The iteration is broadcasted to the specified shape.

Parameters

- shape: the shape used for broadcasting

Template Parameters

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

Reverse iterators

```
template <layout_type L>
```

```
auto rbegin()
```

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

Template Parameters

- L: order used for the traversal. Default value is XTENSOR_DEFAULT_TRAVERSAL.

```
template <layout_type L>
```

```
auto rend()
```

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

Template Parameters

- L: order used for the traversal. Default value is XTENSOR_DEFAULT_TRAVERSAL.

template <layout_type L>

auto **rbegin** () **const**

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

Template Parameters

- L: order used for the traversal. Default value is XTENSOR_DEFAULT_TRAVERSAL.

template <layout_type L>

auto **rend** () **const**

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

Template Parameters

- L: order used for the traversal. Default value is XTENSOR_DEFAULT_TRAVERSAL.

template <layout_type L>

auto **crbegin** () **const**

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

Template Parameters

- L: order used for the traversal. Default value is XTENSOR_DEFAULT_TRAVERSAL.

template <layout_type L>

auto **crend** () **const**

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

Template Parameters

- L: order used for the traversal. Default value is XTENSOR_DEFAULT_TRAVERSAL.

Reverse broadcast iterators

template <layout_type L, class S>

auto **rbegin** (const S &shape)

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

The iteration is broadcasted to the specified shape.

Parameters

- shape: the shape used for broadcasting

Template Parameters

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

template <layout_type L, class S>

auto **rend** (const S &shape)

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

The iteration is broadcasted to the specified shape.

Parameters

- `shape`: the shape used for broadcasting

Template Parameters

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

```
template <layout_type L, class S>
```

```
auto rbegin(const S &shape) const
```

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

The iteration is broadcasted to the specified shape.

Parameters

- `shape`: the shape used for broadcasting

Template Parameters

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

```
template <layout_type L, class S>
```

```
auto rend(const S &shape) const
```

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

The iteration is broadcasted to the specified shape.

Parameters

- `shape`: the shape used for broadcasting

Template Parameters

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

```
template <layout_type L, class S>
```

```
auto crbegin(const S &shape) const
```

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

The iteration is broadcasted to the specified shape.

Parameters

- `shape`: the shape used for broadcasting

Template Parameters

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

```
template <layout_type L, class S>
```

```
auto crend(const S &shape) const
```

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

The iteration is broadcasted to the specified shape.

Parameters

- `shape`: the shape used for broadcasting

Template Parameters

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

1.16.5 xarray

Defined in `xtensor/xarray.hpp`

```
template <class EC, layout_type L, class SC, class Tag>
class xarray_container: public xt::xstrided_container<xarray_container<EC, L, SC, Tag>>, public xt::xcontainer_sem
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.

Constructors

```
xarray_container ()
```

Allocates an uninitialized `xarray_container` that holds 0 element.

```
xarray_container (const shape_type &shape, layout_type l = L)
```

Allocates an uninitialized `xarray_container` with the specified shape and `layout_type`.

Parameters

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

```
xarray_container (const shape_type &shape, const_reference value, layout_type l = L)
```

Allocates an `xarray_container` with the specified shape and `layout_type`.

Elements are initialized to the specified value.

Parameters

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

```
xarray_container (const shape_type &shape, const strides_type &strides)
```

Allocates an uninitialized `xarray_container` with the specified shape and strides.

Parameters

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

xarray_container(const shape_type &shape, const strides_type &strides, const_reference value)

Allocates an uninitialized *xarray_container* with the specified shape and strides.

Elements are initialized to the specified value.

Parameters

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

xarray_container(storage_type &&storage, inner_shape_type &&shape, inner_strides_type &&strides)

Allocates an *xarray_container* by moving specified data, shape and strides.

Parameters

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

xarray_container(const value_type &t)

Allocates an *xarray_container* that holds a single element initialized to the specified value.

Parameters

- t: the value of the element

Constructors from initializer list

xarray_container(nested_initializer_list_t<value_type, 1> t)

Allocates a one-dimensional *xarray_container*.

Parameters

- t: the elements of the *xarray_container*

xarray_container(nested_initializer_list_t<value_type, 2> t)

Allocates a two-dimensional *xarray_container*.

Parameters

- t: the elements of the *xarray_container*

xarray_container(nested_initializer_list_t<value_type, 3> t)

Allocates a three-dimensional *xarray_container*.

Parameters

- t: the elements of the *xarray_container*

xarray_container(nested_initializer_list_t<value_type, 4> t)

Allocates a four-dimensional *xarray_container*.

Parameters

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

xarray_container (nested_initializer_list_t<value_type, 5> *t*)

Allocates a five-dimensional *xarray_container*.

Parameters

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

Extended copy semantic

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

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

Public Functions

```
template <class S>
xarray_container<EC, L, SC, Tag> 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: `XTENSOR_DEFAULT_LAYOUT`).
- `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: `XTENSOR_DEFAULT_LAYOUT`).
- 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.

1.16.6 `xarray_adaptor`

Defined in `xtensor/xarray.hpp`

```
template <class EC, layout_type L, class SC, class Tag>
```

```
class xarray_adaptor : public xt::xstrided_container<xarray_adaptor<EC, L, SC, Tag>>, public xt::xcontainer_semantic
```

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.

Constructors

```
xarray_adaptor (storage_type &&storage)
```

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

Parameters

- `storage`: the container to adapt

```
xarray_adaptor (const storage_type &storage)
```

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

Parameters

- `storage`: the container to adapt

```
template <class D>
```

```
xarray_adaptor (D &&storage, 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

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

```
template <class D>
xarray_adaptor (D &&storage, 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

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

Extended copy semantic

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

1.16.7 adapt (xarray_adaptor)

Defined in `xtensor/xadapt.hpp`

```
template <layout_type L = ::xt::layout_type::row_major, class C, class SC, xtl::check_concept< detail::not_an_array< std::decay_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, xtl::check_concept< detail::not_an_array< std::decay_t< SC >>, detail::not_a_layout_type< C, SC, SS >>> 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 <layout_type L = ::xt::layout_type::row_major, class P, class O, class SC, class A = detail::default_allocator< P, O, SC >> xt::adapt (P &&container, O &&shape, A &&strides)
```

```
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,  lay-
                                     out_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_ownership()` 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 = detail::default_allocator_for_ptr_t<P>, xtl::check_concept<d
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
own-
er-
ship,
SC
&&shape,
SS
&&strides,
const
A
&al-
loc
=
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_ownership()` 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

1.16.8 xtensor

Defined in `xtensor/xtensor.hpp`

template <class *EC*, `std::size_t` *N*, *layout_type* *L*, class *Tag*>

class `xtensor_container` : **public** `xt::xstrided_container`<`xtensor_container`<*EC*, *N*, *L*, *Tag*>>, **public** `xt::xcontainer`_sen

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.

Constructors

`xtensor_container` ()

Allocates an uninitialized `xtensor_container` that holds 0 element.

`xtensor_container` (`nested_initializer_list_t`<`value_type`, *N*> *t*)

Allocates an `xtensor_container` with nested initializer lists.

`xtensor_container` (`const` `shape_type` &*shape*, *layout_type* *l* = *L*)

Allocates an uninitialized `xtensor_container` with the specified shape and `layout_type`.

Parameters

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

`xtensor_container` (`const` `shape_type` &*shape*, `const_reference` *value*, *layout_type* *l* = *L*)

Allocates an `xtensor_container` with the specified shape and `layout_type`.

Elements are initialized to the specified value.

Parameters

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

`xtensor_container` (`const` `shape_type` &*shape*, `const` `strides_type` &*strides*)

Allocates an uninitialized `xtensor_container` with the specified shape and strides.

Parameters

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

xtensor_container(`const` `shape_type` `&shape`, `const` `strides_type` `&strides`, `const_reference` `value`)

Allocates an uninitialized `xtensor_container` with the specified shape and strides.

Elements are initialized to the specified value.

Parameters

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

xtensor_container(`storage_type` `&&storage`, `inner_shape_type` `&&shape`, `inner_strides_type` `&&strides`)

Allocates an `xtensor_container` by moving specified data, shape and strides.

Parameters

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

Extended copy semantic

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

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

typedef `xt::xtensor`

Alias template on `xtensor_container` with default parameters for data container type.

This allows to write

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

instead of the heavier syntax

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

Template Parameters

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

template <class T>

auto xt::from_indices (const std::vector<T> &idx)

Converts std::vector<index_type> (returned e.g.

from *xt::argwhere*) to xtensor.

Return *xt::xtensor*<typename index_type::value_type, 2> (e.g.
xt::xtensor<size_t, 2>)

Parameters

- idx: vector of indices

template <class T>

auto xt::flatten_indices (const std::vector<T> &idx)

Converts std::vector<index_type> (returned e.g.

from *xt::argwhere*) to a flattened xtensor.

Return *xt::xtensor*<typename index_type::value_type, 1> (e.g.
xt::xtensor<size_t, 1>)

Parameters

- vector: of indices

template <class Tag = ravel_tensor_tag, class C, class S>

ravel_return_type_t<C, Tag> xt::ravel_indices (const C &idx, const S &shape, layout_type l = layout_type::row_major)

Converts std::vector<index_type> (returned e.g.

from *xt::argwhere*) to xtensor whereby the indices are ravelled. For 1-d input there is no conversion.

Return *xt::xtensor*<typename index_type::value_type, 1> (e.g.
xt::xtensor<size_t, 1>)

Parameters

- idx: vector of indices
- shape: the shape of the original array
- l: the layout type (row-major or column-major)

1.16.9 xtensor_adaptor

Defined in `xtensor/xtensor.hpp`

```
template <class EC, std::size_t N, layout_type L, class Tag>
```

```
class xtensor_adaptor : public xt::xstrided_container<xtensor_adaptor<EC, N, L, Tag>>, public xt::xcontainer_semanti
```

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.

Constructors

```
xtensor_adaptor (storage_type &&storage)
```

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

Parameters

- storage: the container to adapt

```
xtensor_adaptor (const storage_type &storage)
```

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

Parameters

- storage: the container to adapt

```
template <class D>
```

```
xtensor_adaptor (D &&storage, 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

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

```
template <class D>
```

```
xtensor_adaptor (D &&storage, 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

- storage: the container to adapt
- shape: the shape of the *xtensor_adaptor*

- `strides`: the strides of the `xtensor_adaptor`

Extended copy semantic

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

1.16.10 adapt (xtensor_adaptor)

Defined in `xtensor/xadapt.hpp`

```
template <layout_type L = ::xt::layout_type::row_major, class C>
xtensor_adaptor<C, 1, L> xt::adapt (C &&container, layout_type l = L)
    Constructs a 1-D 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 <layout_type L = ::xt::layout_type::row_major, class C, class SC, xtl::check_concept< detail::not_an_array< std::decay_t<C>>, detail::not_a_layout_type< SC >>>
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, xtl::check_concept< detail::not_an_array< std::decay_t< SC >>, detail::not_a_layout_type< SC >>>,
xarray_adaptor<xtl::closure_type_t<C>, layout_type::dynamic, std::decay_t<SC>> xt::adapt (C &&con-
    tainer, 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 <layout_type L = ::xt::layout_type::row_major, class P, class O, class A = detail::default_allocator_for_ptr_t<P>>>
```

```

xtensor_adaptor<xbuffer_adaptor<xtl::closure_type_t<P>, O, A>, l, L> xt : : adapt (P          &&pointer,
                                     typename
                                     A::size_type    size,
                                     O  ownership,    lay-
                                     out_type l = L, const
                                     A &alloc = A())

```

Constructs a 1-D *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_ownership()` or `acquire_ownership()`
- `l`: the `layout_type` of the *xtensor_adaptor*
- `alloc`: the allocator used for allocating / deallocating the dynamic array

```

template <layout_type L = ::xt::layout_type::row_major, class P, class O, class SC, class A = detail::default_allocator_for_
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,    lay-
                                     out_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_ownership()` 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 = detail::default_allocator_for_ptr_t<P>, xtl::check_concept<

```

```

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
own-
er-
ship,
SC
&&shape,
SS
&&strides,
const
A
&al-
loc
=
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_ownership()` 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

1.16.11 xtensor_fixed

Defined in `xtensor/xfixed.hpp`

```

template <class ET, class S, layout_type L, class Tag>
class xfixed_container : public xt::xcontainer<xfixed_container<ET, S, L, Tag>>, public xt::xcontainer_semantic<xfixe
Dense multidimensional container with tensor semantic and fixed dimension.

```

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

See *xtensor_fixed*

Template Parameters

- *ET*: The type of the elements.
- *S*: The *xshape* template parameter of the container.
- *L*: The *layout_type* of the tensor.
- *Tag*: The expression tag.

Constructors

xfixed_container (**const** inner_shape_type &shape, layout_type l = L)

Create an uninitialized *xfixed_container*.

Note this function is only provided for homogeneity, and the shape & layout argument is disregarded (the template shape is always used).

Parameters

- shape: the shape of the *xfixed_container* (unused!)
- l: the layout_type of the *xfixed_container* (unused!)

xfixed_container (**const** inner_shape_type &shape, value_type v, layout_type l = L)

Create an *xfixed_container*, and initialize with the value of v.

Note, the shape argument to this function is only provided for homogeneity, and the shape argument is disregarded (the template shape is always used).

Parameters

- shape: the shape of the *xfixed_container* (unused!)
- v: the fill value
- l: the layout_type of the *xfixed_container* (unused!)

template <**class** IX = std::integral_constant<std::size_t, N>, **class** EN = std::enable_if_t<IX::value != 0, int>>

xfixed_container (nested_initializer_list_t<value_type, N> t)

Allocates an *xfixed_container* with shape S with values from a C array.

The type returned by get_init_type_t is raw C array value_type[X][Y][Z] for xt::xshape<X, Y, Z>. C arrays can be initialized with the initializer list syntax, but the size is checked at compile time to prevent errors. Note: for clang < 3.8 this is an initializer_list and the size is not checked at compile-or runtime.

Extended copy semantic

template <**class** E>

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

The extended copy constructor.

template <**class** E>

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

The extended assignment operator.

Public Functions

template <**class** ST = std::array<std::size_t, N>>

void **resize** (ST &&shape, bool force = false) **const**

Note that the *xfixed_container* **cannot** be resized.

Attempting to resize with a different size throws an assert in debug mode.

template <**class** ST = shape_type>

```
void resize (ST &&shape, layout_type l) const
    Note that the xfixed_container cannot be resized.
```

Attempting to resize with a different size throws an assert in debug mode.

```
template <class ST = shape_type>
void resize (ST &&shape, const strides_type &strides) const
    Note that the xfixed_container cannot be resized.
```

Attempting to resize with a different size throws an assert in debug mode.

```
template <class ST = std::array<std::size_t, N>>
void reshape (ST &&shape, layout_type layout = L) const
    Note that the xfixed_container cannot be reshaped to a shape different from S.
```

```
typedef xt::xtensor_fixed
```

Alias template on *xfixed_container* with default parameters for layout type.

This allows to write

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

instead of the syntax

```
xt::xfixed_container<double, xt::xshape<2, 2>, xt::layout_type::row_major> a = ...
```

Template Parameters

- T: The value type of the elements.
- FSH: A xshape template shape.
- L: The layout_type of the tensor (default: XTENSOR_DEFAULT_LAYOUT).
- A: The allocator of the containers holding the elements.

1.16.12 xoptional_assembly_base

Defined in `xtensor/xoptional_assembly_base.hpp`

```
template <class D>
class xoptional_assembly_base : xt::xiterable<D>
    Base class for dense multidimensional optional assemblies.
```

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 *xoptional_assembly_base* provides the interface.

Size and shape

```
auto size () const
    Returns the number of element in the optional assembly.
```

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

auto **shape () const**
Returns the shape of the optional assembly.

auto **strides () const**
Returns the strides of the optional assembly.

auto **backstrides () const**
Returns the backstrides of the optional assembly.

Data

template <class... *Args*>
auto **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 **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 **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 **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... Args>
auto unchecked (Args... args)
```

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

Warning This method is meant for performance, for expressions with a dynamic number of dimensions (i.e. not known at compile time). Since it may have undefined behavior (see parameters), `operator()` should be preferred whenever it is possible.

Warning This method is NOT compatible with broadcasting, meaning the following code has undefined behavior:

```
xt::xarray<double> a = {{0, 1}, {2, 3}};
xt::xarray<double> b = {0, 1};
auto fd = a + b;
double res = fd.uncheked(0, 1);
```

Parameters

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

```
template <class... Args>
auto unchecked (Args... args) const
```

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

Warning This method is meant for performance, for expressions with a dynamic number of dimensions (i.e. not known at compile time). Since it may have undefined behavior (see parameters), `operator()` should be preferred whenever it is possible.

Warning This method is NOT compatible with broadcasting, meaning the following code has undefined behavior:

```
xt::xarray<double> a = {{0, 1}, {2, 3}};
xt::xarray<double> b = {0, 1};
auto fd = a + b;
double res = fd.uncheked(0, 1);
```

Parameters

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

```
template <class S>
auto 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 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 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 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 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
- `reuse_cache`: parameter for internal optimization

```
template <class S>
```

```
bool has_linear_assign (const S &strides) const
```

Checks whether the *optional_assembly_base* can be linearly assigned to an expression with the specified strides.

Return a boolean indicating whether a linear assign is possible

Public Functions

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

Parameters

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

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

Return layout_type of the container

template <class T>
 void **fill** (const T &value)
 Fills the data with the given value.

Parameters

- value: the value to fill the data with.

auto **value** ()
 Return an expression for the values of the optional assembly.

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

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

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

1.16.13 xoptional_assembly

Defined in `xtensor/xoptional_assembly.hpp`

template <class *VE*, class *FE*>

class xoptional_assembly: **public** `xt::optional_assembly_base<xoptional_assembly<VE, FE>>`, **public** `xt::xcontainer`

Dense multidimensional container holding optional values, optimized for tensor operations.

The `xoptional_assembly` class implements a dense multidimensional container holding optional values. This container is optimized for 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.

Constructors

xoptional_assembly ()

Allocates an uninitialized `xoptional_assembly` that holds 0 element.

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`

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`

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*

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

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

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 = std::enable_if_t<is_xexpression<OVE>::value && is_xexpression<OFE>>
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

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

xoptional_assembly(nested_initializer_list_t<value_type, 1> t)

Allocates a one-dimensional *xoptional_assembly*.

Parameters

- `t`: the elements of the *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*

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

Parameters

- t: the elements of the *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*

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>
xoptional_assembly (const *xexpression*<E> &e)
The extended copy constructor.

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

Public Functions

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

Parameters

- s: the shape of the *xoptional_assembly*

1.16.14 xoptional_assembly_adaptor

Defined in `xtensor/optional_assembly.hpp`

template <class VEC, class FEC>

```
class xoptional_assembly_adaptor : public xt::xoptional_assembly_base<xoptional_assembly_adaptor<VEC, FEC>>,
    Dense multidimensional adaptor holding optional values, optimized for tensor operations.
```

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

Constructors

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

1.16.15 xmasked_view

Defined in `xtensor/xmasked_view.hpp`

```
template <class CTD, class CTM>
class xmasked_view : public xt::xview_semantic<xmasked_view<CTD, CTM>>, xt::xiterable<xmasked_view<CTD, CTM>>
    View on an xoptional_assembly or xoptional_assembly_adaptor hiding values depending on a given mask.
```

The *xmasked_view* class implements a view on an *xoptional_assembly* or *xoptional_assembly_adaptor*, it takes this *xoptional_assembly* and a mask as input. The mask is an *xexpression* containing boolean values, whenever the value of the mask is false, the optional value of *xmasked_view* is considered missing, otherwise it depends on the underlying *xoptional_assembly*.

Template Parameters

- CTD: The type of expression holding the values.
- CTM: The type of expression holding the mask.

Constructors

```
template <class D, class M>
xmasked_view (D &&data, M &&mask)
    Creates an xmasked_view, given the xoptional_assembly or xoptional_assembly_adaptor and the mask.
```

Parameters

- `data`: the underlying *xoptional_assembly* or *xoptional_assembly_adaptor*
- `mask`: the mask.

Size and shape

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

Returns the number of elements in the *xmasked_view*.

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

Returns the number of dimensions of the *xmasked_view*.

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

Returns the shape of the *xmasked_view*.

auto **strides** () **const**

Returns the strides of the *xmasked_view*.

auto **backstrides** () **const**

Returns the backstrides of the *xmasked_view*.

Data

template <class... *Args*>

auto **operator** () (*Args*... *args*)

Returns a reference to the element at the specified position in the *xmasked_view*.

Parameters

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

template <class... *Args*>

auto **operator** () (*Args*... *args*) **const**

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

Parameters

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

template <class... *Args*>

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

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

Parameters

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

Exceptions

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

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

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

Parameters

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

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 unchecked (Args... args)
```

Returns a reference to the element at the specified position in the *xmasked_view*.

Warning This method is meant for performance, for expressions with a dynamic number of dimensions (i.e. not known at compile time). Since it may have undefined behavior (see parameters), `operator()` should be preferred whenever it is possible.

Warning This method is NOT compatible with broadcasting, meaning the following code has undefined behavior:

```
xt::xarray<double> a = {{0, 1}, {2, 3}};
xt::xarray<double> b = {0, 1};
auto fd = a + b;
double res = fd.uncheked(0, 1);
```

Parameters

- `args`: a list of indices specifying the position in the *xmasked_view*. Indices must be unsigned integers, the number of indices must be equal to the number of dimensions of the *xmasked_view*, else the behavior is undefined.

```
template <class... Args>
auto unchecked (Args... args) const
```

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

Warning This method is meant for performance, for expressions with a dynamic number of dimensions (i.e. not known at compile time). Since it may have undefined behavior (see parameters), `operator()` should be preferred whenever it is possible.

Warning This method is NOT compatible with broadcasting, meaning the following code has undefined behavior:

```
xt::xarray<double> a = {{0, 1}, {2, 3}};
xt::xarray<double> b = {0, 1};
auto fd = a + b;
double res = fd.uncheked(0, 1);
```

Parameters

- `args`: a list of indices specifying the position in the `xmasked_view`. Indices must be unsigned integers, the number of indices must be equal to the number of dimensions of the `xmasked_view`, else the behavior is undefined.

```
template <class S>
```

```
auto operator[] (const S &index)
```

Returns a reference to the element at the specified position in the `xmasked_view`.

Parameters

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

```
template <class S>
```

```
auto operator[] (const S &index) const
```

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

Parameters

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

```
template <class It>
```

```
auto element (It first, It last)
```

Returns a reference to the element at the specified position in the `xmasked_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 number of dimensions of the `xmasked_view`.

```
template <class It>
```

```
auto element (It first, It last) const
```

Returns a constant reference to the element at the specified position in the `xmasked_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 number of dimensions of the `xmasked_view`.

Public Functions

```
layout_type layout () const
```

Return the `layout_type` of the `xmasked_view`.

Return `layout_type` of the `xmasked_view`

```
template <class T>
```

```
void fill (const T &value)
```

Fills the data with the given value.

Parameters

- `value`: the value to fill the data with.

auto **value** ()

Return an expression for the values of the *xmasked_view*.

auto **value** () **const**

Return a constant expression for the values of the *xmasked_view*.

auto **visible** ()

Return an expression for the mask of the *xmasked_view*.

auto **visible** () **const**

Return a constant expression for the mask of the *xmasked_view*.

1.16.16 xview

Defined in `xtensor/xview.hpp`

template <class *CT*, class... *S*>

class **xview**: **public** `xt::xview_semantic<xview<CT, S...>>`, **public** `xt::xiterable<xview<CT, S...>>`, **public** `extension::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*, *keep*, *drop*

Template Parameters

- *CT*: the closure type of the *xexpression* to adapt
- *S*: the slices type describing the shape adaptation

Extended copy semantic

template <class *E*>

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

The extended assignment operator.

Constructor

template <class *CTA*, class *FSL*, class... *SL*>

xview (*CTA* &&*e*, *FSL* &&*first_slice*, *SL*&&... *slices*)

Constructs a view on the specified xexpression.

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

See *view*

Parameters

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

Size and shape

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

Returns the number of dimensions of the view.

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

Returns the size of the expression.

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

Returns the shape of the view.

auto **slices** () **const**

Returns the slices of the view.

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

Returns the slices of the view.

Data

template <class T>

void **fill** (const T &value)

Fills the view with the given value.

Parameters

- `value`: the value to fill the view with.

auto **expression** ()

Returns a reference to the underlying expression of the view.

auto **expression** () **const**

Returns a const reference to the underlying expression of the view.

template <class... Args>

auto **operator** () (Args... args)

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

Parameters

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

template <class... Args>

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

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

Parameters

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

Exceptions

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

```
template <class... Args>
auto unchecked(Args... args)
```

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

Warning This method is meant for performance, for expressions with a dynamic number of dimensions (i.e. not known at compile time). Since it may have undefined behavior (see parameters), operator() should be preferred whenever it is possible.

Warning This method is NOT compatible with broadcasting, meaning the following code has undefined behavior:

```
xt::xarray<double> a = {{0, 1}, {2, 3}};
xt::xarray<double> b = {0, 1};
auto fd = a + b;
double res = fd.unchecked(0, 1);
```

Parameters

- *args*: a list of indices specifying the position in the view. Indices must be unsigned integers, the number of indices must be equal to the number of dimensions of the view, else the behavior is undefined.

```
template <class OS>
auto operator[](const OS &index)
```

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

Parameters

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

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

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

Parameters

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

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

Returns a constant reference to the element at the specified position in the view, 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 **unchecked** (Args... *args*) **const**

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

Warning This method is meant for performance, for expressions with a dynamic number of dimensions (i.e. not known at compile time). Since it may have undefined behavior (see parameters), `operator()` should be preferred whenever it is possible.

Warning This method is NOT compatible with broadcasting, meaning the following code has undefined behavior:

```
xt::xarray<double> a = {{0, 1}, {2, 3}};
xt::xarray<double> b = {0, 1};
auto fd = a + b;
double res = fd.unchecked(0, 1);
```

Parameters

- *args*: a list of indices specifying the position in the view. Indices must be unsigned integers, the number of indices must be equal to the number of dimensions of the view, else the behavior is undefined.

template <class OS>

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

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

Parameters

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

template <class T>

auto **storage** ()

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

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

template <class T>

auto **strides** () **const**

Return the strides for the underlying container of the view.

template <class T>

auto **data** () **const**

Return the pointer to the underlying buffer.

template <class T>

auto **data_offset** () **const**

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

Broadcasting

template <class ST>

bool **broadcast_shape** (ST &*shape*, 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
- `reuse_cache`: parameter for internal optimization

template <class ST>

bool **has_linear_assign** (const ST &strides) const

Checks whether the xview can be linearly assigned to an expression with the specified strides.

Return a boolean indicating whether a linear assign is possible

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 start_val, B stop_val)

Select a range from `start_val` to `stop_val`.

You can use the shorthand `_` syntax to select from the start or until the end.

```
using namespace xt::placeholders; // to enable _ syntax

range(3, _) // select from index 3 to the end
range(_, 5) // select from index 0 to 5
range(_, _) // equivalent to `all()`
```

See *view*, *strided_view*

template <class A, class B, class C>

auto xt::range (A start_val, B stop_val, C step)

Select a range from `start_val` to `stop_val` with step `step`. You can use the shorthand `_` syntax to select from the start or until the end.

```
using namespace xt::placeholders; // to enable _ syntax
range(3, _, 5) // select from index 3 to the end with stepsize 5
```

See *view*, *strided_view*

auto xt::all ()

Returns a slice representing a full dimension, to be used as an argument of view function.

See *view*, *strided_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](#), [strided_view](#)

auto xt::ellipsis()

Returns a slice representing all remaining dimensions, and selecting all in these dimensions.

Ellipsis will expand to a series of `all()` slices, until the number of slices is equal to the number of dimensions of the source array.

Note: ellipsis can only be used in `strided_view`!

```
xarray<double> a = xarray<double>::from_shape({5, 5, 1, 1, 5});
auto v = xt::strided_view(a, {2, xt::ellipsis(), 2});
// equivalent to using {2, xt::all(), xt::all(), xt::all(), 2};
```

See [strided_view](#)

template <class T>

detail::disable_integral_keep<T> xt::keep(T &&indices)

Create a non-contiguous slice from a container of indices to keep.

Note: this slice cannot be used in the `xstrided_view`!

```
xt::xarray<double> a = xt::arange(9);
a.reshape({3, 3});
xt::view(a, xt::keep(0, 2); // => {{0, 1, 2}, {6, 7, 8}}
xt::view(a, xt::keep(1, 1, 1); // => {{3, 4, 5}, {3, 4, 5}, {3, 4, 5}}
```

Return instance of `xkeep_slice`

Parameters

- `indices`: The indices container

template <class T>

detail::disable_integral_drop<T> xt::drop(T &&indices)

Create a non-contiguous slice from a container of indices to drop.

Note: this slice cannot be used in the `xstrided_view`!

```
xt::xarray<double> a = xt::arange(9);
a.reshape({3, 3});
xt::view(a, xt::drop(0, 2); // => {{3, 4, 5}}
```

Return instance of `xdrop_slice`

Parameters

- `indices`: The container of indices to drop

1.16.17 xstrided_view

Defined in `xtensor/xstrided_view.hpp`

template <class CT, class S, layout_type L = layout_type::dynamic, class FST = typename detail::flat_storage_type<CT>::type>

class xstrided_view: public xt::xview_semantic<xstrided_view<CT, S, L, FST>>, public xt::xiterable<xstrided_view<CT, S, L, FST>>

View of an xexpression using strides.

The `xstrided_view` class implements a view utilizing an initial offset and strides.

See *strided_view*, *transpose*

Template Parameters

- CT: the closure type of the *xexpression* type underlying this view
- L: the layout of the strided view
- S: the strides type of the strided view
- FST: the flat storage type used for the strided view

Extended copy semantic

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

Constructor

```
template <class CTA>
xstrided_view (CTA &&e, S &&shape, strides_type &&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

Data

```
template <class T>
void fill (const T &value)
    Fills the view with the given value.
```

Parameters

- value: the value to fill the view with.

```
typedef xt::xstrided_slice_vector
    vector of slices used to build a xstrided_view
```

```
template <layout_type L = layout_type::dynamic, class E, class S, class X>
auto xt::strided_view (E &&e, S &&shape, X &&stride, std::size_t offset = 0, layout_type layout = L)
    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

- `L`: the static layout type of the view (default: `dynamic`)
- `E`: type of xexpression
- `S`: strides type
- `X`: strides type

template <class E>

auto `xt::strided_view` (E &&e, const *xstrided_slice_vector* &lices)

Function to create a dynamic view from an xexpression and an `xstrided_slice_vector`.

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

Return initialized `strided_view` according to slices

Parameters

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

You can also achieve the same with the following short-hand syntax:

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

template <class E, class S>

auto `xt::reshape_view` (E &&e, S &&shape, *layout_type* layout)

Return a view on a container with a new shape.

Note: if you resize the underlying container, this view becomes invalidated.

Return view on xexpression with new shape

Parameters

- `e`: xexpression to reshape
- `shape`: new shape
- `layout`: new layout (optional)

1.16.18 `xbroadcast`

Defined in `xtensor/xbroadcast.hpp`

```
template <class CT, class X>
class xbroadcast : public xt::xexpression<xbroadcast<CT, X>>, public xt::xconst_iterable<xbroadcast<CT, X>>, public
```

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.

Constructor

```
template <class CTA, class S>
xbroadcast (CTA &&e, const 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

```
template <class CTA>
xbroadcast (CTA &&e, shape_type &&s)
```

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

Parameters

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

Size and shape

```
auto size () const
```

Returns the size of the expression.

```
auto dimension () const
```

Returns the number of dimensions of the expression.

```
auto shape () const
```

Returns the shape of the expression.

```
layout_type layout () const
```

Returns the `layout_type` of the expression.

Data

```
auto expression () const
```

Returns a constant reference to the underlying expression of the broadcast expression.

```
template <class... Args>
```

auto **operator ()** (Args... *args*) **const**

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

Parameters

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

template <class... *Args*>

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

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

Parameters

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

Exceptions

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

template <class... *Args*>

auto **unchecked** (Args... *args*) **const**

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

Warning This method is meant for performance, for expressions with a dynamic number of dimensions (i.e. not known at compile time). Since it may have undefined behavior (see parameters), `operator()` should be preferred whenever it is possible.

Warning This method is NOT compatible with broadcasting, meaning the following code has undefined behavior:

```
xt::xarray<double> a = {{0, 1}, {2, 3}};
xt::xarray<double> b = {0, 1};
auto fd = a + b;
double res = fd.uncheked(0, 1);
```

Parameters

- *args*: a list of indices specifying the position in the expression. Indices must be unsigned integers, the number of indices must be equal to the number of dimensions of the expression, else the behavior is undefined.

template <class *S*>

auto **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 **element** (*It*, *It last*) **const**

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

Parameters

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

Broadcasting

```
template <class S>
```

```
bool broadcast_shape (S &shape, 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`: parameter for internal optimization

```
template <class S>
```

```
bool has_linear_assign (const S &strides) const
```

Checks whether the `xbroadcast` can be linearly assigned to an expression with the specified strides.

Return a boolean indicating whether a linear assign is possible

```
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.

1.16.19 `xindex_view`

Defined in `xtensor/xindex_view.hpp`

```
template <class CT, class I>
```

```
class xindex_view: public xt::xview_semantic<xindex_view<CT, I>>, public xt::xiterable<xindex_view<CT, I>>, public
```

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

Extended copy semantic

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

Constructor

```
template <class CTA, class I2>
xindex_view (CTA &&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 size () const
    Returns the size of the xindex_view.

auto dimension () const
    Returns the number of dimensions of the xindex_view.

auto shape () const
    Returns the shape of the xindex_view.
```

Data

```
template <class T>
void fill (const T &value)
    Fills the view with the given value.
```

Parameters

- *value*: the value to fill the view with.

```
auto operator () (size_type idx = size_type(0))
    Returns a reference to the element at the specified position in the xindex_view.
```

Parameters

- *idx*: index specifying the position in the *index_view*. More indices may be provided, only the last one will be used.

```
auto unchecked (size_type idx)
    Returns a reference to the element at the specified position in the xindex_view.
```

Parameters

- *idx*: index specifying the position in the *index_view*.

auto **operator ()** (size_type *idx* = size_type(0)) **const**

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

Parameters

- *idx*: index specifying the position in the *index_view*. More indices may be provided, only the last one will be used.

auto **unchecked** (size_type *idx*) **const**

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

Parameters

- *idx*: index specifying the position in the *index_view*.

auto **expression** ()

Returns a reference to the underlying expression of the view.

auto **expression** () **const**

Returns a constant reference to the underlying expression of the view.

template <class *S*>

auto **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 **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 **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.

template <class *It*>

auto **element** (*It first*, *It*) **const**

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 **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
- *reuse_cache*: parameter for internal optimization

template <class O>

bool **has_linear_assign** (const O&) **const**

Checks whether the *xindex_view* can be linearly assigned to an expression with the specified strides.

Return a boolean indicating whether a linear assign is possible

template <class *ECT*, class *CCT*>

class **xfiltration**

Filter of a xexpression for fast scalar assign.

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

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

See *filtration*

Template Parameters

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

Extended copy semantic

template <class E>

auto **operator=** (const E &*e*)

Assigns the scalar *e* to `*this`.

Return a reference to `\ *this`.

Parameters

- *e*: the scalar to assign.

Constructor

```
template <class ECTA, class CCTA>
xfiltration(ECTA &&e, CCTA &&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 assignment

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

Return a reference to `*this`.

Parameters

- `e`: the scalar to add.

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

Return a reference to `*this`.

Parameters

- `e`: the scalar to subtract.

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

Return a reference to `*this`.

Parameters

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

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

Return a reference to `*this`.

Parameters

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

```
template <class E>
auto operator%=(const 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, argwhere(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`: expression 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*

1.16.20 xfunctor_view

Defined in `xtensor/xfunctor_view.hpp`

```
template <class F, class CT>
```

```
class xfunctor_view : public xt::xfunctor_applier_base<F, CT>, public xt::xview_semantic<xfunctor_view<F, CT>>, pub
```

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

Extended copy semantic

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

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*

1.17 Functions and generators

1.17.1 xfunction

Defined in `xtensor/xfunction.hpp`

template <class *F*, class... *CT*>

class **xfunction**: `xt::xconst_iterable<xfunction<F, CT...>>`, **public** `xt::xexpression<xfunction<F, CT...>>`, **public** `extension::`

Multidimensional function operating on xtensor expressions.

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

Template Parameters

- *F*: the function type
- *CT*: the closure types for arguments of the function

Constructor

template <**class** *Func*, class... *CTA*, **class** *U* = `std::enable_if_t<!std::is_base_of<std::decay_t<Func>, self_type>::value>`>

xfunction (*Func* &&*f*, *CTA*&&... *e*)

Constructs an xfunction applying the specified function to the given arguments.

Parameters

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

Size and shape

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

Returns the size of the expression.

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

Returns the number of dimensions of the function.

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

Returns the shape of the xfunction.

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

Returns the *layout_type* of the xfunction.

Data

template <class... *Args*>

auto **operator** () (*Args*... *args*) **const**

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

Parameters

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

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

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

Parameters

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

Exceptions

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

```
template <class... Args>
auto unchecked (Args... args) const
```

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

Warning This method is meant for performance, for expressions with a dynamic number of dimensions (i.e. not known at compile time). Since it may have undefined behavior (see parameters), `operator()` should be preferred whenever it is possible.

Warning This method is NOT compatible with broadcasting, meaning the following code has undefined behavior:

```
xt::xarray<double> a = {{0, 1}, {2, 3}};
xt::xarray<double> b = {0, 1};
auto fd = a + b;
double res = fd.unchecked(0, 1);
```

Parameters

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

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

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

Parameters

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

Broadcasting

```
template <class S>
bool broadcast_shape (S &shape, 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 has_linear_assign(const S &strides) const
    Checks whether the xfunction can be linearly assigned to an expression with the specified strides.
```

Return a boolean indicating whether a linear assign is possible

Defined in `xtensor/xmath.hpp`

```
template <class F, class... E>
auto xt::make_lambda_xfunction(F &&lambda, E&&... args)
```

Create a xfunction from a lambda.

This function can be used to easily create performant xfunctions from lambdas:

```
template <class E1>
inline auto square(E1&& e1) noexcept
{
    auto fnct = [] (auto x) -> decltype(x * x) {
        return x * x;
    };
    return make_lambda_xfunction(std::move(fnct), std::forward<E1>(e1));
}
```

Lambda function allow the reusal of a single arguments in multiple places (otherwise only correctly possible when using `xshared_expressions`). `auto` lambda functions are automatically vectorized with `xsimd` if possible (note that the trailing `-> decltype(...)` is mandatory for the feature detection to work).

Return lazy xfunction

Parameters

- `lambda`: the lambda to be vectorized
- `args`: forwarded arguments

1.17.2 xreducer

Defined in `xtensor/xreducer.hpp`

```
template <class F, class CT, class X>
class xreducer : public xt::xexpression<xreducer<F, CT, X>>, public xt::xconst_iterable<xreducer<F, CT, X>>, public e
    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.:

Constructor

template <class Func, class CTA, class AX>

xreducer (Func &&func, CTA &&e, AX &&axes)

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

Parameters

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

Size and shape

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

Returns the size of the expression.

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

Returns the number of dimensions of the expression.

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

Returns the shape of the expression.

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

Returns the shape of the expression.

Data

auto **expression** () **const**

Returns a constant reference to the underlying expression of the reducer.

template <class... Args>

auto **operator** () (Args... args) **const**

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

Parameters

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

template <class... Args>

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

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

Parameters

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

Exceptions

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

template <class... *Args*>

auto **unchecked** (*Args... args*) **const**

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

Warning This method is meant for performance, for expressions with a dynamic number of dimensions (i.e. not known at compile time). Since it may have undefined behavior (see parameters), `operator()` should be preferred whenever it is possible.

Warning This method is NOT compatible with broadcasting, meaning the following code has undefined behavior:

```
xt::xarray<double> a = {{0, 1}, {2, 3}};
xt::xarray<double> b = {0, 1};
auto fd = a + b;
double res = fd.uncheked(0, 1);
```

Parameters

- `args`: a list of indices specifying the position in the reducer. Indices must be unsigned integers, the number of indices must be equal to the number of dimensions of the reducer, else the behavior is undefined.

template <class *S*>

auto **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 **element** (*It first*, *It last*) **const**

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

Parameters

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

Broadcasting

template <class *S*>

bool **broadcast_shape** (*S* &*shape*, 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
- `reuse_cache`: parameter for internal optimization

```
template <class S>
```

```
bool has_linear_assign(const S &strides) const
```

Checks whether the xreducer can be linearly assigned to an expression with the specified strides.

Return a boolean indicating whether a linear assign is possible

```
template <class F, class E, class X, class EVS = evaluation_strategy::lazy, xtl::check_concept< xtl::negation< is_evaluation_strategy< E, F, X, EVS >>>
auto xt::reduce(F &&f, E &&e, X &&axes, EVS evaluation_strategy = EVS())
```

Returns an *xexpression* applying the specified reducing function to an expression 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)

1.17.3 xaccumulator

Defined in `xtensor/xaccumulator.hpp`

```
template <class F, class E, class EVS = evaluation_strategy::immediate, xtl::check_concept< is_evaluation_strategy< EVS, F, E >>>
auto xt::accumulate(F &&f, E &&e, EVS evaluation_strategy = EVS())
```

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 EVS = evaluation_strategy::immediate>
```

```
auto xt::accumulate(F &&f, E &&e, std::ptrdiff_t axis, EVS evaluation_strategy = EVS())
```

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

1.17.4 xgenerator

Defined in `xtensor/xgenerator.hpp`

```
template <class F, class R, class S>
```

```
class xgenerator : public xt::xexpression<xgenerator<F, R, S>>, public xt::xconst_iterable<xgenerator<F, R, S>>, public
```

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

Constructor

```
template <class Func>
```

```
xgenerator (Func &&f, const S &shape)
```

Constructs an `xgenerator` applying the specified function over the given shape.

Parameters

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

Size and shape

```
auto size () const
```

Returns the size of the expression.

```
auto dimension () const
```

Returns the number of dimensions of the function.

```
auto shape () const
```

Returns the shape of the `xgenerator`.

Data

```
template <class... Args>
```

```
auto operator () (Args... args) const
```

Returns the evaluated element at the specified position in the function.

Parameters

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

```
template <class... Args>
```

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

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

Parameters

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

Exceptions

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

template <class... *Args*>

auto **unchecked** (Args... *args*) **const**

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

Warning This method is meant for performance, for expressions with a dynamic number of dimensions (i.e. not known at compile time). Since it may have undefined behavior (see parameters), `operator()` should be preferred whenever it is possible.

Warning This method is NOT compatible with broadcasting, meaning the following code has undefined behavior:

```
xt::xarray<double> a = {{0, 1}, {2, 3}};
xt::xarray<double> b = {0, 1};
auto fd = a + b;
double res = fd.uncheked(0, 1);
```

Parameters

- *args*: a list of indices specifying the position in the expression. Indices must be unsigned integers, the number of indices must be equal to the number of dimensions of the expression, else the behavior is undefined.

template <class *It*>

auto **element** (*It first*, *It last*) **const**

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

Parameters

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

Broadcasting

template <class *O*>

bool **broadcast_shape** (*O &shape*, 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`: parameter for internal optimization

```
template <class O>
bool has_linear_assign(const O&) const
    Checks whether the xgenerator can be linearly assigned to an expression with the specified strides.
```

Return a boolean indicating whether a linear assign is possible

1.17.5 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, layout_type L = ::xt::layout_type::row_major, class S>
xarray<T, L> xt::empty(const S &shape)
    Create a xcontainer (xarray, xtensor or xtensor_fixed) with uninitialized values of with value_type T and shape.
    Selects the best container match automatically from the supplied shape.
```

- `std::vector` → `xarray<T>`
- `std::array` or `initializer_list` → `xtensor<T, N>`
- `xshape<N...>` → `xtensor_fixed<T, xshape<N...>>`

Parameters

- `shape`: shape of the new xcontainer

```
template <class E>
auto xt::full_like(const xexpression<E> &e, typename E::value_type fill_value)
    Create a xcontainer (xarray, xtensor or xtensor_fixed), filled with fill_value and of the same shape, value type and layout as the input xexpression e.
```

Parameters

- `e`: the *xexpression* from which to extract shape, value type and layout.

- `fill_value`: the value used to set each element of the returned xcontainer.

```
template <class E>
```

```
auto xt::empty_like (const xexpression<E> &e)
```

Create a xcontainer (xarray, xtensor or xtensor_fixed) with uninitialized values of the same shape, value type and layout as the input xexpression *e*.

Parameters

- *e*: the xexpression from which to extract shape, value type and layout.

```
template <class E>
```

```
auto xt::zeros_like (const xexpression<E> &e)
```

Create a xcontainer (xarray, xtensor or xtensor_fixed), filled with zeros and of the same shape, value type and layout as the input xexpression *e*.

Note: contrary to `zeros(shape)`, this function returns a non-lazy, allocated container! Use `'xt::zeros<double>(e.shape())'` for a lazy version.

Parameters

- *e*: the xexpression from which to extract shape, value type and layout.

```
template <class E>
```

```
auto xt::ones_like (const xexpression<E> &e)
```

Create a xcontainer (xarray, xtensor or xtensor_fixed), filled with ones and of the same shape, value type and layout as the input xexpression *e*.

Note: contrary to `ones(shape)`, this function returns a non-lazy, evaluated container! Use `'xt::ones<double>(e.shape())'` for a lazy version.

Parameters

- *e*: the xexpression from which to extract shape, value type and layout.

```
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, class S = T>
```

```
auto xt::arange (T start, T stop, S 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: expressions 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.

1.17.6 xmanipulation

Defined in `xtensor/xmanipulation.hpp`

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.

template <layout_type L = ::xt::layout_type::row_major, class E>

auto xt::ravel (E &&e)

Returns a flatten view of the given expression.

No copy is made.

Parameters

- e: the input expression

Template Parameters

- *L*: the layout used to read the elements of *e*. If no parameter is specified, XTENSOR_DEFAULT_TRAVERSAL is used.
- *E*: the type of the expression

```
template <class E>
```

```
auto xt::flatten (E &&e)
```

Returns a flatten view of the given expression.

No copy is made. The layout used to read the elements is the one of *e*.

Parameters

- *e*: the input expression

Template Parameters

- *E*: the type of the expression

```
template <class E>
```

```
auto xt::trim_zeros (E &&e, const std::string &direction = "fb")
```

Trim zeros at beginning, end or both of 1D sequence.

Return returns a view without zeros at the beginning and end

Parameters

- *e*: input xexpression
- *direction*: string of either 'f' for trim from beginning, 'b' for trim from end or 'fb' (default) for both.

```
template <class E>
```

```
auto xt::squeeze (E &&e)
```

Returns a squeeze view of the given expression.

No copy is made. Squeezing an expression removes dimensions of extent 1.

Parameters

- *e*: the input expression

Template Parameters

- *E*: the type of the expression

```
template <class E, class S, class Tag = check_policy::none, std::enable_if_t<!std::is_integral< S >::value, int > = 0>
```

```
auto xt::squeeze (E &&e, S &&axis, Tag check_policy = Tag())
```

Remove single-dimensional entries from the shape of an xexpression.

Parameters

- *e*: input xexpression
- *axis*: integer or container of integers, select a subset of single-dimensional entries of the shape.
- *check_policy*: select *check_policy*. With *check_policy::full()*, selecting an axis which is greater than one will throw a *runtime_error*.

```
template <class E>
```

auto xt::expand_dims (E &&e, std::size_t axis)

Expand the shape of an xexpression.

Insert a new axis that will appear at the axis position in the expanded array shape. This will return a `strided_view` with a `xt::newaxis()` at the indicated axis.

Return returns a `strided_view` with expanded dimension

Parameters

- e: input xexpression
- axis: axis to expand

template <class E>

auto xt::split (E &e, std::size_t n, std::size_t axis = 0)

Split xexpression along axis into subexpressions.

This splits an xexpression along the axis in n equal parts and returns a vector of `strided_view`. Calling `split` with `axis >` dimension of e or a n that does not result in an equal division of the xexpression will throw a `runtime_error`.

Parameters

- e: input xexpression
- n: number of elements to return
- axis: axis along which to split the expression

template <std::size_t N, class E>

auto xt::atleast_Nd (E &&e)

Expand dimensions of xexpression to at least N

This adds `newaxis()` slices to a `strided_view` until the dimension of the view reaches at least N. Note: dimensions are added equally at the beginning and the end. For example, a 1-D array of shape (N,) becomes a view of shape (1, N, 1).

Return `strided_view` with expanded dimensions

Parameters

- e: input xexpression

Template Parameters

- N: the number of requested dimensions

template <class E>

auto xt::atleast_1d (E &&e)

Expand to at least 1D.

See [atleast_Nd](#)

template <class E>

auto xt::atleast_2d (E &&e)

Expand to at least 2D.

See [atleast_Nd](#)

```
template <class E>
auto xt::atleast_3d(E &&e)
    Expand to at least 3D.
```

See *atleast_Nd*

```
template <std::ptrdiff_t N = 1, class E>
auto xt::rot90(E && e, const std::array< std::ptrdiff_t, 2 > & axes = {0, 1})
    Rotate an array by 90 degrees in the plane specified by axes.

    Rotation direction is from the first towards the second axis.
```

Return returns a view with the result of the rotation

Parameters

- *e*: the input xexpression
- *axes*: the array is rotated in the plane defined by the axes. Axes must be different.

Template Parameters

- *N*: number of times the array is rotated by 90 degrees. Default is 1.

```
template <class E>
auto xt::flip(E &&e, 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 returns a view with the result of the flip

Parameters

- *e*: the input xexpression
- *axis*: the axis along which elements should be reversed

1.17.7 xsort

Defined in `xtensor/xsort.hpp`

```
template <class E>
auto xt::sort(const xexpression<E> &e, placeholders::xtuph)
```

```
template <class E>
auto xt::sort(const xexpression<E> &e, std::ptrdiff_t axis = -1)
    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::argsort(const xexpression<E> &e, placeholders::xtuph)
```

```
template <class E>
```

```
auto xt::argsort (const xexpression<E> &e, std::ptrdiff_t axis = -1)
```

Argsort xexpression (optionally along axis) Performs an indirect sort along the given axis.

Returns an xarray of indices of the same shape as e that index data along the given axis in sorted order.

Return argsorted index array

Parameters

- e: xexpression to argsort
- axis: axis along which argsort is performed

```
template <layout_type L = ::xt::layout_type::row_major, class E>
```

```
auto xt::argmin (const xexpression<E> &e)
```

```
template <layout_type L = ::xt::layout_type::row_major, 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

- e: input xexpression
- axis: select axis (or none)

```
template <layout_type L = ::xt::layout_type::row_major, class E>
```

```
auto xt::argmax (const xexpression<E> &e)
```

```
template <layout_type L = ::xt::layout_type::row_major, 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 maximal value

Parameters

- e: input xexpression
- axis: select axis (or none)

```
template <class E>
```

```
auto xt::unique (const xexpression<E> &e)
```

Find unique elements of a xexpression.

This returns a flattened xtensor with sorted, unique elements from the original expression.

Parameters

- e: input xexpression (will be flattened)

1.17.8 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,
                                                    bool replace = true, 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
- `replace`: whether to sample with or without replacement
- `engine`: random number engine

```
template <class T, class E = random::default_engine_type>
void xt::random::shuffle (xexpression<T> &e, E &engine = random::get_default_random_engine())
```

Randomly shuffle elements inplace in `xcontainer` along first axis.

The order of sub-arrays is changed but their contents remain the same.

Parameters

- `e`: `xcontainer` to shuffle inplace
- `engine`: random number engine

```
template <class T, class E = random::default_engine_type>
std::enable_if_t<std::is_integral<T>::value, xtensor<T, 1>> xt::random::permutation (T e, E &engine = random::get_default_random_engine())
```

Randomly permute a sequence, or return a permuted range.

If the first parameter is an integer, this function creates a new `arange(e)` and returns it randomly permuted. Otherwise, this function creates a copy of the input, passes it to

See `shuffle` and returns the result.

Return randomly permuted copy of container or `arange`.

Parameters

- `e`: input `xexpression` or integer
- `engine`: random number engine to use (optional)

1.17.9 xhistogram

Defined in `xtensor/xhistogram.hpp`

enum `xt::histogram_algorithm`

Defines different algorithms to be used in “`histogram_bin_edges`”.

Values:

automatic

linspace

logspace

uniform

template `<class R = double, class E1, class E2, class E3>`

`auto xt::histogram` (`E1 &&data`, `E2 &&bin_edges`, `E3 &&weights`, `bool density = false`)

Compute the histogram of a set of data.

Return An one-dimensional `xarray<double>`, length: `bin_edges.size()-1`.

Parameters

- `data`: The data.
- `bin_edges`: The bin-edges.
- `weights`: Weight factors corresponding to each data-point.
- `density`: If true the resulting integral is normalized to 1. [default: false]

template `<class E1, class E2, xtl::check_concept< is_xexpression< std::decay_t< E2 >> > = 0>`

`auto xt::bincount` (`E1 &&data`, `E2 &&weights`, `std::size_t minlength = 0`)

Count number of occurrences of each value in array of non-negative ints.

The number of bins (of size 1) is one larger than the largest value in `x`. If `minlength` is specified, there will be at least this number of bins in the output array (though it will be longer if necessary, depending on the contents of `x`). Each bin gives the number of occurrences of its index value in `x`. If `weights` is specified the input array is weighted by it, i.e. if a value `n` is found at position `i`, `out[n] += weight[i]` instead of `out[n] += 1`.

Return 1D container with the bincount

Parameters

- `data`: the 1D container with integers to count into bins
- `weights`: a 1D container with the same number of elements as `data`
- `minlength`: The minlength

template `<class E1, class E2, class E3>`

`auto xt::histogram_bin_edges` (`E1 &&data`, `E2 &&weights`, `E3 left`, `E3 right`, `std::size_t bins = 10`,
`histogram_algorithm mode = histogram_algorithm::automatic`)

Compute the bin-edges of a histogram of a set of data using different algorithms.

Return An one-dimensional `xarray<double>`, length: `bins+1`.

Parameters

- `data`: The data.
- `weights`: Weight factors corresponding to each data-point.

- `left`: The lower-most edge.
- `right`: The upper-most edge.
- `bins`: The number of bins. [default: 10]
- `mode`: The type of algorithm to use. [default: “auto”]

Further overloads

template <class E1, class E2>

auto xt::**histogram**(E1 &&data, E2 &&bin_edges, bool density = false)
 Compute the histogram of a set of data.

Return An one-dimensional xarray<double>, length: bin_edges.size()-1.

Parameters

- `data`: The data.
- `bin_edges`: The bin-edges.
- `density`: If true the resulting integral is normalized to 1. [default: false]

template <class E1>

auto xt::**histogram**(E1 &&data, std::size_t bins = 10, bool density = false)
 Compute the histogram of a set of data.

Return An one-dimensional xarray<double>, length: bin_edges.size()-1.

Parameters

- `data`: The data.
- `bins`: The number of bins. [default: 10]
- `density`: If true the resulting integral is normalized to 1. [default: false]

template <class E1, class E2>

auto xt::**histogram**(E1 &&data, std::size_t bins, E2 &&weights, bool density = false)
 Compute the histogram of a set of data.

Return An one-dimensional xarray<double>, length: bin_edges.size()-1.

Parameters

- `data`: The data.
- `bins`: The number of bins.
- `weights`: Weight factors corresponding to each data-point.
- `density`: If true the resulting integral is normalized to 1. [default: false]

template <class E1, class E2>

auto xt::**histogram_bin_edges**(E1 &&data, E2 left, E2 right, std::size_t bins = 10, *his-*
togram_algorithm mode = histogram_algorithm::automatic)
 Compute the bin-edges of a histogram of a set of data using different algorithms.

Return An one-dimensional xarray<double>, length: bins+1.

Parameters

- `data`: The data.
- `left`: The lower-most edge.
- `right`: The upper-most edge.
- `bins`: The number of bins. [default: 10]
- `mode`: The type of algorithm to use. [default: “auto”]

template <class E1, class E2>

auto xt::histogram_bin_edges (E1 &&data, E2 &&weights, std::size_t bins = 10, histogram_algorithm mode = histogram_algorithm::automatic)

Compute the bin-edges of a histogram of a set of data using different algorithms.

Return An one-dimensional xarray<double>, length: bins+1.

Parameters

- `data`: The data.
- `weights`: Weight factors corresponding to each data-point.
- `bins`: The number of bins. [default: 10]
- `mode`: The type of algorithm to use. [default: “auto”]

template <class E1>

auto xt::histogram_bin_edges (E1 &&data, std::size_t bins = 10, histogram_algorithm mode = histogram_algorithm::automatic)

Compute the bin-edges of a histogram of a set of data using different algorithms.

Return An one-dimensional xarray<double>, length: bins+1.

Parameters

- `data`: The data.
- `bins`: The number of bins. [default: 10]
- `mode`: The type of algorithm to use. [default: “auto”]

1.18 IO Operations

1.18.1 xio: pretty printing

Defined in `xtensor/xio.hpp`

This file defines functions for pretty printing xexpressions. It defines appropriate overloads for the << operator for `std::ostream`s and xexpressions.

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

int main()
{
    xt::xarray<double> a = {{1,2,3}, {4,5,6}};
    std::cout << a << std::endl;
    return 0;
}
```

Will print

```
{ { 1., 2., 3.},
  { 4., 5., 6.}}
```

With the following functions, the global print options can be set:

void `xt::print_options::set_line_width` (int *line_width*)
Sets the line width.

After `line_width` chars, a new line is added.

Parameters

- `line_width`: The line width

void `xt::print_options::set_threshold` (int *threshold*)
Sets the threshold after which summarization is triggered (default: 1000).

Parameters

- `threshold`: The number of elements in the xexpression that triggers summarization in the output

void `xt::print_options::set_edge_items` (int *edge_items*)
Sets the number of edge items.

If the summarization is triggered, this value defines how many items of each dimension are printed.

Parameters

- `edge_items`: The number of edge items

void `xt::print_options::set_precision` (int *precision*)
Sets the precision for printing floating point values.

Parameters

- `precision`: The number of digits for floating point output

One can also locally overwrite the print options with io manipulators:

`class line_width`

io manipulator used to set the width of the lines when printing an expression.

```
using po = xt::print_options;
xt::xarray<double> a = {{1, 2, 3}, {4, 5, 6}};
std::cout << po::line_width(100) << a << std::endl;
```

`class threshold`

io manipulator used to set the threshold after which summarization is triggered.

```
using po = xt::print_options;
xt::xarray<double> a = xt::rand::randn<double>({2000, 500});
std::cout << po::threshold(50) << a << std::endl;
```

`class edge_items`

io manipulator used to set the number of edge items if the summarization is triggered.

```
using po = xt::print_options;
xt::xarray<double> a = xt::rand::randn<double>({2000, 500});
std::cout << po::edge_items(5) << a << std::endl;
```

class precision

io manipulator used to set the precision of the floating point values when printing an expression.

```
using po = xt::print_options;
xt::xarray<double> a = xt::rand::randn<double>({2000, 500});
std::cout << po::precision(5) << a << std::endl;
```

1.18.2 xnpv: read/write NPY files

Defined in `xtensor/xnpv.hpp`

```
template <typename T, layout_type L = layout_type::dynamic>
```

```
auto xt::load_npy(const std::string &filename)
```

Loads a npy file (the numpy storage format)

Return xarray with contents from npy file

Parameters

- `filename`: The filename or path to the file

Template Parameters

- `T`: select the type of the npy file (note: currently there is no dynamic casting if types do not match)
- `L`: select `layout_type::column_major` if you stored data in Fortran format

```
template <typename E>
```

```
void xt::dump_npy(const std::string &filename, const xexpression<E> &e)
```

Save xexpression to NumPy npy format.

Parameters

- `filename`: The filename or path to dump the data
- `e`: the xexpression

1.18.3 xcsv: read/write CSV files

Defined in `xtensor/xcsv.hpp`

```
template <class T, class A = std::allocator<T>>
```

```
xcsv_tensor<T, A> xt::load_csv(std::istream &stream)
```

Load tensor from CSV.

Returns an *xexpression* for the parsed CSV

Parameters

- `stream`: the input stream containing the CSV encoded values

```
template <class E>
```

```
void xt::dump_csv(std::ostream &stream, const xexpression<E> &e)
```

Dump tensor to CSV.

Parameters

- `stream`: the output stream to write the CSV encoded values
- `e`: the tensor expression to serialize

1.18.4 xjson: serialize to/from JSON

Defined in `xtensor/xjson.hpp`

template <class E>

`enable_xexpression<E> xt::to_json(nlohmann::json &j, const E &e)`

JSON serialization of an xtensor expression.

The `to_json` method is used by the `nlohmann_json` package for automatic serialization of user-defined types. The method is picked up by argument-dependent lookup.

Parameters

- `j`: a JSON object
- `e`: a const *xexpression*

template <class E>

`enable_xcontainer_semantics<E> xt::from_json(const nlohmann::json &j, E &e)`

JSON deserialization of a xtensor expression with a container or a view semantics.

The `from_json` method is used by the `nlohmann_json` library for automatic serialization of user-defined types. The method is picked up by argument-dependent lookup.

Note: for converting a JSON object to a value, `nlohmann_json` requires the value type to be default constructible, which is typically not the case for expressions with a view semantics. In this case, `from_json` can be called directly.

Parameters

- `j`: a const JSON object
- `e`: an *xexpression*

1.19 Mathematical functions**1.19.1 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 subtraction 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::operator? (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 E>
```

```
bool xt::operator& (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 E1, class E2>
```

```
auto xt::less (E1 &&e1, E2 &&e2)
```

Lesser than.

Returns an *xfunction* for the element-wise lesser than comparison of *e1* and *e2*. This function is equivalent to *operator<(E1&&, E2&&)*.

Return an *xfunction*

Parameters

- $e1$: an *xexpression* or a scalar
- $e2$: an *xexpression* or a scalar

```
template <class E1, class E2>
auto xt::less_equal (E1 &&e1, E2 &&e2)
Lesser or equal.
```

Returns an *xfunction* for the element-wise lesser or equal comparison of $e1$ and $e2$. This function is equivalent to *operator<=(E1&&, E2&&)*.

Return an *xfunction*

Parameters

- $e1$: an *xexpression* or a scalar
- $e2$: an *xexpression* or a scalar

```
template <class E1, class E2>
auto xt::greater (E1 &&e1, E2 &&e2)
Greater than.
```

Returns an *xfunction* for the element-wise greater than comparison of $e1$ and $e2$. This function is equivalent to *operator>(E1&&, E2&&)*.

Return an *xfunction*

Parameters

- $e1$: an *xexpression* or a scalar
- $e2$: an *xexpression* or a scalar

```
template <class E1, class E2>
auto xt::greater_equal (E1 &&e1, E2 &&e2)
Greater or equal.
```

Returns an *xfunction* for the element-wise greater or equal comparison of $e1$ and $e2$. This function is equivalent to *operator>=(E1&&, 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)
Bitwise and.
```

Returns an *xfunction* for the element-wise bitwise and 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)
```

Bitwise or.

Returns an *xfunction* for the element-wise bitwise 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)
```

Bitwise xor.

Returns an *xfunction* for the element-wise bitwise xor 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)
```

Bitwise not.

Returns an *xfunction* for the element-wise bitwise not of *e*.

Return an *xfunction*

Parameters

- *e*: an *xexpression*

```
template <class E1, class E2>
```

```
auto xt::left_shift (E1 &&e1, E2 &&e2)
```

Bitwise left shift.

Returns an *xfunction* for the element-wise bitwise left shift of *e1* by *e2*.

Return an *xfunction*

Parameters

- *e1*: an *xexpression*
- *e2*: an *xexpression*

```
template <class E1, class E2>
```

```
auto xt::right_shift (E1 &&e1, E2 &&e2)
```

Bitwise right shift.

Returns an *xfunction* for the element-wise bitwise right shift of *e1* by *e2*.

Return an *xfunction*

Parameters

- *e1*: an *xexpression*
- *e2*: an *xexpression*

```
template <class E1, class E2>
```

auto xt : :operator<< (E1 &&e1, E2 &&e2)

Bitwise left shift.

Returns an *xfunction* for the element-wise bitwise left shift of e1 by e2.

Return an *xfunction*

See *left_shift*

Parameters

- e1: an *xexpression*
- e2: an *xexpression*

template <class E1, class E2>

auto xt : :operator>> (E1 &&e1, E2 &&e2)

Bitwise right shift.

Returns an *xfunction* for the element-wise bitwise right shift of e1 by e2.

Return an *xfunction*

See *right_shift*

Parameters

- e1: an *xexpression*
- e2: an *xexpression*

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

<i>operator+</i>	identity
<i>operator-</i>	opposite
<i>operator+</i>	addition
<i>operator-</i>	subtraction
<i>operator*</i>	multiplication
<i>operator/</i>	division
<i>operator </i>	logical or
<i>operator&&</i>	logical and
<i>operator!</i>	logical not
<i>where</i>	ternary selection
<i>any</i>	return true if any value is truthy
<i>all</i>	return true if all the values are truthy
<i>operator<</i>	element-wise lesser than
<i>operator<=</i>	element-wise less or equal
<i>operator></i>	element-wise greater than
<i>operator>=</i>	element-wise greater or equal
<i>operator==</i>	expression equality

Continued on next page

Table 1 – continued from previous page

<i>operator!=</i>	expression inequality
<i>equal</i>	element-wise equality
<i>not_equal</i>	element-wise inequality
<i>less</i>	element-wise lesser than
<i>less_equal</i>	element-wise less or equal
<i>greater</i>	element-wise greater than
<i>greater_equal</i>	element-wise greater or equal
<i>cast</i>	element-wise <i>static_cast</i>
<i>operator&</i>	bitwise and
<i>operator </i>	bitwise or
<i>operator^</i>	bitwise xor
<i>operator~</i>	bitwise not
<i>left_shift</i>	bitwise shift left
<i>right_shift</i>	bitwise shift right
<i>operator<<</i>	bitwise shift left
<i>operator>></i>	bitwise shift right

1.19.2 Index related functions

Defined in `xtensor/xoperation.hpp`

```
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 T>
```

```
auto xt::nonzero (const T &arr)
```

return vector of indices where T is not zero

Return vector of vectors, one for each dimension of `arr`, containing the indices of the non-zero elements in that dimension

Parameters

- `arr`: input array

```
template <class T>
```

```
auto xt::argwhere (const T &arr)
```

return vector of indices where `arr` is not zero

Return vector of `index_types` where `arr` is not equal to zero (use `xt::from_indices` to convert)

See `xt::from_indices`

Parameters

- `arr`: input array

```
template <class T>
```

auto xt : : **from_indices** (const std::vector<T> &idx)
 Converts std::vector<index_type> (returned e.g.
 from xt::argwhere) to xtensor.

Return xt::xtensor<typename index_type::value_type, 2> (e.g.
 xt::xtensor<size_t, 2>)

Parameters

- idx: vector of indices

<i>where</i>	indices selection
<i>nonzero</i>	indices selection
<i>argwhere</i>	indices selection
<i>from_indices</i>	builder from indices

1.19.3 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*

<i>abs</i>	absolute value
<i>fabs</i>	absolute value
<i>fmod</i>	remainder of the floating point division operation
<i>remainder</i>	signed remainder of the division operation
<i>fma</i>	fused multiply-add operation
<i>minimum</i>	element-wise minimum
<i>maximum</i>	element-wise maximum
<i>fmin</i>	element-wise minimum for floating point values
<i>fmax</i>	element-wise maximum for floating point values
<i>fdim</i>	element-wise positive difference
<i>clip</i>	element-wise clipping operation
<i>sign</i>	element-wise indication of the sign

1.19.4 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*

<i>exp</i>	natural exponential function
<i>exp2</i>	base 2 exponential function
<i>expm1</i>	natural exponential function, minus one
<i>log</i>	natural logarithm function
<i>log2</i>	base 2 logarithm function
<i>log10</i>	base 10 logarithm function
<i>log1p</i>	natural logarithm of one plus function

1.19.5 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 $e1$ 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 <std::size_t N, class E>
```

```
auto xt::pow(E &&e)
```

Integer power function.

Returns an *xfunction* for the element-wise power of $e1$ to an integral constant.

Instead of computing the power by using the (expensive) logarithm, this function computes the power in a number of straight-forward multiplication steps. This function is therefore much faster (even for high N) than the generic pow-function.

For example, $e1^{20}$ can be expressed as $((e1^2)^2)^2 * (e1^2)^2$, which is just 5 multiplications.

Return an *xfunction*

Parameters

- e : an *xexpression*

Template Parameters

- N : the exponent (has to be positive integer)

```
template <class E1>
```

```
auto xt::square(E1 &&e1)
```

Square power function, equivalent to $e1 * e1$.

Returns an *xfunction* for the element-wise value of $e1 * e1$.

Return an *xfunction*

Parameters

- $e1$: an *xexpression* or a scalar

```
template <class E1>
```

```
auto xt::cube(E1 &&e1)
```

Cube power function, equivalent to $e1 * e1 * e1$.

Returns an *xfunction* for the element-wise value of $e1 * e1$.

Return an *xfunction*

Parameters

- $e1$: 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

<i>pow</i>	power function
<i>sqrt</i>	square root function
<i>cbrt</i>	cubic root function
<i>hypot</i>	hypotenuse function

1.19.6 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)
```

Arctangent 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

- e_2 : an *xexpression* or a scalar

<i>sin</i>	sine function
<i>cos</i>	cosine function
<i>tan</i>	tangent function
<i>asin</i>	arc sine function
<i>acos</i>	arc cosine function
<i>atan</i>	arc tangent function
<i>atan2</i>	arc tangent function, determining quadrants

1.19.7 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
```

<i>sinh</i>	hyperbolic sine function
<i>cosh</i>	hyperbolic cosine function
<i>tanh</i>	hyperbolic tangent function
<i>asinh</i>	inverse hyperbolic sine function
<i>acosh</i>	inverse hyperbolic cosine function
<i>atanh</i>	inverse hyperbolic tangent function

1.19.8 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  $e$ , without 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 absolute value of the gamma function of e .

Return an *xfunction*

Parameters

- e : an *xexpression*

<i>erf</i>	error function
<i>erfc</i>	complementary error function
<i>tgamma</i>	gamma function
<i>lgamma</i>	natural logarithm of the gamma function

1.19.9 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*

<i>ceil</i>	nearest integers not less
<i>floor</i>	nearest integers not greater
<i>trunc</i>	nearest integers not greater in magnitude
<i>round</i>	nearest integers, rounding away from zero
<i>nearbyint</i>	nearest integers using current rounding mode
<i>rint</i>	nearest integers using current rounding mode

1.19.10 Classification functions

`xtensor` provides the following classification functions for *xexpressions* and scalars:

Defined in `xtensor/xmath.hpp`

```
template <class E>
```

```
auto xt::isfinite (E &&e)
    finite value check
```

Returns an *xfunction* for the element-wise finite value check tangent of *e*.

Return an *xfunction*

Parameters

- *e*: an *xexpression*

```
template <class E>
auto xt::isinf (E &&e)
    infinity check
```

Returns an *xfunction* for the element-wise infinity check tangent of *e*.

Return an *xfunction*

Parameters

- *e*: an *xexpression*

```
template <class E>
auto xt::isnan (E &&e)
    NaN check.
```

Returns an *xfunction* for the element-wise NaN check tangent of *e*.

Return an *xfunction*

Parameters

- *e*: an *xexpression*

```
template <class E1, class E2>
auto xt::isclose (E1 &&e1, E2 &&e2, double rtol = 1e-05, double atol = 1e-08, bool equal_nan = false)
    Element-wise closeness detection.
```

Returns an *xfunction* that evaluates to true if the elements in *e1* and *e2* are close to each other according to parameters *atol* and *rtol*. The equation is: $\text{std::abs}(a - b) \leq (\text{m_atol} + \text{m_rtol} * \text{std::abs}(b))$.

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)

<i>isfinite</i>	checks for finite values
<i>isinf</i>	checks for infinite values
<i>isnan</i>	checks for NaN values
<i>isclose</i>	element-wise closeness detection
<i>allclose</i>	closeness reduction

1.19.11 Reducing functions

xtensor provides the following reducing functions for xexpressions:

Defined in `xtensor/xmath.hpp`

```
template <class T = void, class E, class X, class EVS = evaluation_strategy::lazy, xtl::check_concept< xtl::negation< is_c
auto xt : :sum (E &&e, X &&axes, EVS es = EVS())
```

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 T = void, class E, class X, class EVS = evaluation_strategy::lazy, xtl::check_concept< xtl::negation< is_c
auto xt : :prod (E &&e, X &&axes, EVS es = EVS())
```

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 T = void, class E, class X, class EVS = evaluation_strategy::lazy, xtl::check_concept< xtl::negation< is_c
auto xt : :mean (E &&e, X &&axes, EVS es = EVS())
```

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)

template <class E, class X, class EVS = evaluation_strategy::lazy, xtl::check_concept< xtl::negation< is_evaluation_strategy>>>
 auto xt : : **variance** (E &&e, X &&axes, EVS es = EVS())

Compute the variance along the specified axes.

Returns the variance of the array elements, a measure of the spread of a distribution. The variance is computed for the flattened array by default, otherwise over the specified axes.

Note: this function is not yet specialized for complex numbers.

Return an *xexpression*

See `stddev`, *mean*

Parameters

- *e*: an *xexpression*
- *axes*: the axes along which the mean is computed (optional)
- *es*: evaluation strategy to use (lazy (default), or immediate)

template <class E, class X, class EVS = evaluation_strategy::lazy, xtl::check_concept< xtl::negation< is_evaluation_strategy>>>
 auto xt : : **stddev** (E &&e, X &&axes, EVS es = EVS())

Compute the standard deviation along the specified axis.

Returns the standard deviation, a measure of the spread of a distribution, of the array elements. The standard deviation is computed for the flattened array by default, otherwise over the specified axis.

Note: this function is not yet specialized for complex numbers.

Return an *xexpression*

See `variance`, *mean*

Parameters

- *e*: an *xexpression*
- *axes*: the axes along which the mean is computed (optional)
- *es*: evaluation strategy to use (lazy (default), or immediate)

template <class T>

auto xt : : **diff** (const *xexpression*<T> &a, std::size_t n = 1, std::ptrdiff_t axis = -1)

Calculate the n-th discrete difference along the given axis.

Calculate the n-th discrete difference along the given axis. This function is not lazy (might change in the future).

Return an xarray

Parameters

- *a*: an *xexpression*
- *n*: The number of times values are differenced. If zero, the input is returned as-is. (optional)
- *axis*: The axis along which the difference is taken, default is the last axis.

template <class T = void, class E, class X, class EVS = evaluation_strategy::lazy, xtl::check_concept< xtl::negation< is_evaluation_strategy>>>
 auto xt : : **amax** (E &&e, X &&axes, EVS es = EVS())

Maximum element along given axis.

Returns an *xreducer* for the maximum of elements over given *axes*.

Return an *xreducer*

Parameters

- *e*: an *xexpression*
- *axes*: the axes along which the maximum is found (optional)
- *es*: evaluation strategy of the reducer

template <class T = void, class E, class X, class EVS = evaluation_strategy::lazy, xtl::check_concept< xtl::negation< is_c
 auto xt:::amin (E &&e, X &&axes, EVS es = EVS())

Minimum element along given axis.

Returns an *xreducer* for the minimum of elements over given *axes*.

Return an *xreducer*

Parameters

- *e*: an *xexpression*
- *axes*: the axes along which the minimum is found (optional)
- *es*: evaluation strategy of the reducer

template <class T>

auto xt:::trapz (const *xexpression*<T> &y, double dx = 1.0, std::ptrdiff_t axis = -1)

Integrate along the given axis using the composite trapezoidal rule.

Returns definite integral as approximated by trapezoidal rule. This function is not lazy (might change in the future).

Return an xarray

Parameters

- *y*: an *xexpression*
- *dx*: the spacing between sample points (optional)
- *axis*: the axis along which to integrate.

template <class T, class E>

auto xt:::trapz (const *xexpression*<T> &y, const *xexpression*<E> &x, std::ptrdiff_t axis = -1)

Integrate along the given axis using the composite trapezoidal rule.

Returns definite integral as approximated by trapezoidal rule. This function is not lazy (might change in the future).

Return an xarray

Parameters

- *y*: an *xexpression*
- *x*: an *xexpression* representing the sample points corresponding to the *y* values.
- *axis*: the axis along which to integrate.

Defined in `xtensor/xnorm.hpp`

template <class E, class X, class EVS, class>

auto xt:::norm_l0 (E &&e, X &&axes, EVS es)

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* (or xcontainer, depending on evaluation strategy) 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)
- *es*: evaluation strategy to use (lazy (default), or immediate)

```
template <class E, class X, class EVS, class>
```

```
auto xt : :norm_l1 (E &&e, X &&axes, EVS es)
```

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* (or xcontainer, depending on evaluation strategy) 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)
- *es*: evaluation strategy to use (lazy (default), or immediate)

```
template <class E, class X, class EVS, class>
```

```
auto xt : :norm_sq (E &&e, X &&axes, EVS es)
```

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* (or xcontainer, depending on evaluation strategy) 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)
- *es*: evaluation strategy to use (lazy (default), or immediate)

```
template <class E, class X, class EVS = evaluation_strategy::lazy, xtl::check_concept< is_xexpression< E >, xtl::negation<
```

```
auto xt : :norm_l2 (E &&e, X &&axes, EVS es = EVS())
```

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))`) (or xcontainer, depending on evaluation strategy)

Parameters

- *e*: an *xexpression*
- *es*: evaluation strategy to use (lazy (default), or immediate)
- *axes*: the axes along which the norm is computed

```
template <class E, class X, class EVS, class>
```

auto xt : :norm_linf (E &&e, X &&axes, EVS es)

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* (or xcontainer, depending on evaluation strategy) 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)
- es: evaluation strategy to use (lazy (default), or immediate)

template <class E, class X, class EVS = evaluation_strategy::lazy, xtl::check_concept< xtl::negation< is_evaluation_strategy

auto xt : :norm_lp_to_p (E &&e, double p, X &&axes, EVS es = EVS())

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* (or xcontainer, depending on evaluation strategy) 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)
- es: evaluation strategy to use (lazy (default), or immediate)

template <class E, class X, class EVS = evaluation_strategy::lazy, xtl::check_concept< xtl::negation< is_evaluation_strategy

auto xt : :norm_lp (E &&e, double p, X &&axes, EVS es = EVS())

Lp norm of an array-like argument over given axes.

Returns an *xreducer* for the Lp norm ($p \neq 0$) of the elements across given *axes*.

Return an *xreducer* (or xcontainer, depending on evaluation strategy) 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)
- es: evaluation strategy to use (lazy (default), or immediate)

template <class E, class EVS = evaluation_strategy::lazy, xtl::check_concept< is_xexpression< E > > = 0>

auto xt : :norm_induced_l1 (E &&e, EVS es = EVS())

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* (or xcontainer, depending on evaluation strategy)

Parameters

- *e*: a 2D *xexpression*
- *es*: evaluation strategy to use (lazy (default), or immediate)

```
template <class E, class EVS = evaluation_strategy::lazy, xtl::check_concept< is_xexpression< E > = 0>
auto xt::norm_induced_linf (E &&e, EVS es = EVS())
```

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* (or *xcontainer*, depending on evaluation strategy)

Parameters

- *e*: a 2D *xexpression*
- *es*: evaluation strategy to use (lazy (default), or immediate)

<i>sum</i>	sum of elements over given axes
<i>prod</i>	product of elements over given axes
<i>mean</i>	mean of elements over given axes
<i>variance</i>	variance of elements over given axes
<i>stddev</i>	standard deviation of elements over given axes
<i>diff</i>	Calculate the n-th discrete difference along the given axis
<i>amax</i>	amax of elements over given axes
<i>amin</i>	amin of elements over given axes
<i>trapz</i>	Integrate along the given axis using the composite trapezoidal rule
<i>norm_l0</i>	L0 pseudo-norm over given axes
<i>norm_l1</i>	L1 norm over given axes
<i>norm_sq</i>	Squared L2 norm over given axes
<i>norm_l2</i>	L2 norm over given axes
<i>norm_linf</i>	Infinity norm over given axes
<i>norm_lp_to_p</i>	p _{th} power of L _p norm over given axes
<i>norm_lp</i>	L _p norm over given axes
<i>norm_induced_l1</i>	Induced L1 norm of a matrix
<i>norm_induced_linf</i>	Induced L-infinity norm of a matrix

1.19.12 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::ptrdiff_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::ptrdiff_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)

<i>cumsum</i>	cumulative sum of elements over a given axis
<i>cumprod</i>	cumulative product of elements over given axes

1.19.13 NaN functions

xtensor provides the following functions that deal with NaNs in *xexpressions*:

Defined in `xtensor/xmath.hpp`

```
template <class E>
auto xt::nan_to_num (E &&e)
    Convert nan or +/- inf to numbers.
```

This functions converts nan to 0, and +inf to the highest, -inf to the lowest floating point value of the same type.

Return an *xexpression*

Parameters

- *e*: input *xexpression*

```
template <class E, class X, class EVS = evaluation_strategy::lazy, xtl::check_concept< xtl::negation< is_evaluation_strategy> > >
auto xt::nansum (E &&e, X &&axes, EVS es = EVS())
    Sum of elements over given axes, replacing nan with 0.
```

Returns an *xreducer* for the sum of elements over given *axes*, replacing nan with 0.

Return an *xreducer*

Parameters

- *e*: an *xexpression*
- *axes*: the axes along which the sum is performed (optional)
- *es*: evaluation strategy of the reducer (optional)

```
template <class E, class X, class EVS = evaluation_strategy::lazy, xtl::check_concept< xtl::negation< is_evaluation_strategy> > >
auto xt::nanprod (E &&e, X &&axes, EVS es = EVS())
    Product of elements over given axes, replacing nan with 1.
```

Returns an *xreducer* for the sum of elements over given *axes*, replacing nan with 1.

Return an *xreducer*

Parameters

- *e*: an *xexpression*
- *axes*: the axes along which the sum is performed (optional)
- *es*: evaluation strategy of the reducer (optional)

```
template <class E>
auto xt::nancumsum(E &&e)
```

```
template <class E>
auto xt::nancumsum(E &&e, std::ptrdiff_t axis)
    Cumulative sum, replacing nan with 0.
```

Returns an xaccumulator for the sum of elements over given *axis*, replacing nan with 0.

Return an xaccumulator

Parameters

- *e*: an *xexpression*
- *axis*: the axis along which the elements are accumulated (optional)

```
template <class E>
auto xt::nancumprod(E &&e)
```

```
template <class E>
auto xt::nancumprod(E &&e, std::ptrdiff_t axis)
    Cumulative product, replacing nan with 1.
```

Returns an xaccumulator for the product of elements over given *axis*, replacing nan with 1.

Return an xaccumulator

Parameters

- *e*: an *xexpression*
- *axis*: the axis along which the elements are accumulated (optional)

<i>nan_to_num</i>	convert NaN and +/- inf to finite numbers
<i>nansum</i>	sum of elements over a given axis, replacing NaN with 0
<i>nanprod</i>	product of elements over given axes, replacing NaN with 1
<i>nancumsum</i>	cumsum of elements over a given axis, replacing NaN with 0
<i>nancumprod</i>	cumprod of elements over given axes, replacing NaN with 1

1.20 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.

1.20.1 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.

Visual Studio and `min` and `max` macros

Visual Studio defines `min` and `max` macros causing calls to e.g. `std::min` and `std::max` to be interpreted as syntax errors. The `NOMINMAX` definition may be used to disable these macros.

In `xtensor`, to prevent macro replacements of `min` and `max` functions, we wrap them with parentheses, so that client code does not need the `NOMINMAX` definition.

Visual Studio 2017 (15.7.1) seeing declarations as extra overloads

In `xvectorize.hpp`, Visual Studio 15.7.1 sees the forward declaration of `vectorize(E&&)` as a separate overload.

1.20.2 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.

1.20.3 Clang < 3.8 not 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 prevents 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.

1.20.4 GCC < 5.1 and `std::is_trivially_default_constructible`

The versions 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.

1.20.5 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 header `<math.h>` 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.

1.20.6 GCC-8 and deleted functions

GCC-8 (8.2 specifically) doesn't seem to SFINAE deleted functions correctly. A strided view on a `dynamic_view` errors with a message: `use of deleted function`. It should pick the *other* implementation by SFINAE on the function signature, because our `has_strides<dynamic_view>` meta-function should return `false`. Instantiating the `has_strides<dynamic_view>` in the `inner_types` fixes the issue. Original issue here: <https://github.com/QuantStack/xtensor/issues/1273>

1.21 Build and configuration

1.21.1 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 dimension checks 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 and runs 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
```

1.21.2 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.
- `XTENSOR_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`.
- `XTENSOR_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.
- `XTENSOR_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.
- `XTENSOR_DEFAULT_TRAVERSAL`: defines the default traversal order (`row_major`, `column_major`) for algorithms and iterators on tensors and arrays. We *strongly* discourage using this macro, which is provided for testing purpose.

1.22 Internals of `xtensor`

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.

1.22.1 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 rcbegin() const noexcept;
template <class L>
const_reverse_iterator rcend() const noexcept;
```

This template parameter is defaulted to `XTENSOR_DEFAULT_TRAVERSAL` (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

xtensor

```
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_shape_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::initializer_list<I> index) const;

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

const storage_type& storage() 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

The `xfunction` class 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. It inherits from `xconst_iterable`, thus providing iteration methods.

1.22.2 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_shape_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... Args>
const_reference unchecked(Args... args) const;

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

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

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

const storage_type& storage() 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.

List of available expression classes

xtensor provides the following expression classes:

Containers

- `xarray_container`: N-dimensional array with dynamic shape
- `xarray_adaptor`: N-dimensional array adaptor for STL-like containers or C arrays
- `xtensor_container`: N-dimensional array with static number of dimensions
- `xtensor_adaptor`: N-dimensional tensor adaptor for STL-like containers or C arrays
- `xfixed_container`: N-dimensional array with static shape
- `xfixed_adaptor`: N-dimensional fixed tensor adaptor for STL-like containers or C arrays

Most of the methods of these classes are implemented in the base class `xcontainer`, the inheriting classes only provide constructors and assignment operators for the value semantic.

The container classes are generally used through type aliases which set many of the template arguments:

- `xarray`
- `xtensor`
- `xfixed_tensor`

The classes for adaptors can be instantiated through the many overloads of `xt::adapt` function, so that their templates parameters are deduced.

Scalar

xtensor provides the `xscalar` class to adapt scalar values and give them the required API.

Optional containers

- `xoptional_assembly`: N-dimensional array holding optional values.
- `xoptional_assembly_adaptor`: N-dimensional adaptor holding optional values.

Most of the methods of these classes are defined in their base class `xoptional_assembly_base`.

Views

- `xview`: N-dimensional view with static number of slices, supporting all kind of slices
- `xstrided_view`: N-dimensional view with dynamic number of slices, supporting strided slices only (see below)
- `xdynamic_view`: N-dimensional view with dynamic number of slices, supporting all kind of slices
- `xfunctor_view`: N-dimensional view applying a functor to its underlying elements (e.g. `imag`, `real`)
- `xindex_view`: Flat (1D) view yielding the values at the indices of its index array
- `xmasked_view`: View on optional expression hiding values depending on a mask

When the index of an element in the underlying expression of a view can be computed thanks to a strided scheme, the slice used in this view is said to be a strided slice. *xtensor* provides the following strided slices:

- `xrange`
- `xstepped_range`
- `xall`
- `xnewaxis`

The following slices are not strided, and thus incompatible with `xstrided_view`:

- `xkeep_slice`
- `xdrop_slice`

Functional expressions

Contrary to containers and views, the functional expressions are immutable.

- `xbroadcast`: Broadcasts an expression to a specific shape
- `xfunction`: N-dimensional function operating on tensor expressions
- `xgenerator`: N-dimensional function operating on indices
- `xreducer`: Reducing function operating over specified axes

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 be able to access the data container. This is achieved by requiring that every class inheriting from `xcontainer` provides the following methods:

```
storage_type& storage_impl() noexcept;
const storage_type& storage_impl() const noexcept;
```

These are the implementation methods of the `storage()` 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:
    storage_type m_storage;
    storage_type& storage() noexcept;
    const storage_type& storage() 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 storage_type::value_type;
using reference = typename storage_type::reference;
using const_reference = typename storage_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:

```
storage_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 structure 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*.

1.22.3 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

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`. This is a template class whose parameters are:

- a functor describing the operation of the mathematical function
- the closures of the child expressions, i.e. the most optimal way to store each child expression

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, 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` 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... CT>
template <class... Args>
inline auto xfunction<F, CT...>::operator() (Args... args) const -> const_reference
{
    return m_functor(m_first(args...), m_second(args...));
}

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

In fact, `xfunction` 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 = F;
    using type = select_xfunction_expression_t<expression_tag, F, 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 class modeling a function. In `xtensor`, two tags are provided, with the following mapping:

- `xtensor_expression_tag` -> `xfunction`
- `xoptional_expression_tag` -> `xfunction`

In the case of `xfunction`, the tag is also used to select a mixin base class that will extend its API.

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, F, 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.

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 an hypothetical `xmapped_function` class, which provides additional mapping access operators. The first thing to do is to define a new tag:

```

struct xmapped_expression_tag
{
};

```

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

```

namespace xt
{
    namespace extension
    {
        template <>
        struct expression_tag_and<xtensor_expression_tag, xmapped_expression_tag>
        {
            using type = xmapped_expression_tag;
        };

        template <>
        struct expression_tag_and<xmapped_expression_tag, xtensor_expression_tag>
            : expression_tag_and<xtensor_expression_tag, xmapped_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 last requirement 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<xmapped_expression_tag, F, E...>
        {
            using type = xmapped_function<F, typename F::result_type, E...>;
        };
    }
}

```

In this example, `xmapped_function` may provide the same API as `xfunction` and define some additional methods unrelated to the assignment mechanics. However it is possible to define a function class with an API totally different from the one of `xfunction`. In that case, the assignment mechanics need to be customized too, this is detailed in [Assignment](#).

1.22.4 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 iterator 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` corresponds 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 and a 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++`.

1.22.5 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 resize 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 expression 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); });
}
```

1.23 Extending xtensor

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

1.23.1 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.

1.23.2 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;
};

// This is the adaptor we need to define to plug raw_tensor in xtensor
template <class T>
class raw_tensor_adaptor;
```

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_adaptor<T>>
{
    using container_type = typename raw_tensor<T>::container_type;
    using inner_shape_type = typename raw_tensor<T>::shape_type;
    using inner_strides_type = inner_shape_type;
    using inner_backstrides_type = inner_shape_type;
    using shape_type = inner_shape_type;
    using strides_type = inner_shape_type;
    using backstrides_type = inner_shape_type;
    static constexpr layout_type layout = raw_tensor<T>::layout;
};

```

The `inner_XXX_type` are the types used to store and read the shape, strides and 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_adaptor<T>>
    : xcontainer_iterable_types<raw_tensor_adaptor<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 definition of the previous structures, 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>;
};

```

(continues on next page)

(continued from previous page)

```

using base_type = xcontainer<self_type>;
using semantic_base = xcontainer_semantic<self_type>;

// ... specific constructors here

raw_tensor_adaptor(const raw_tensor_adaptor&) = default;
raw_tensor_adaptor& operator=(const raw_tensor_adaptor&) = default;

raw_tensor_adaptor(raw_tensor_adaptor&&) = default;
raw_tensor_adaptor& operator=(raw_tensor_adaptor&&) = default;

template <class E>
raw_tensor_type(const xexpression<E>& e)
    : base_type()
    {
        semantic_base::assign(e);
    }

template <class E>
self_type& operator=(const xexpression<E>& e)
    {
        return semantic_base::operator=(e);
    }
};

```

The last two methods are extended copy constructor and assign operator. They allow writing things like

```

using tensor_type = raw_tensor_adaptor<double>;
tensor_type a, b, c;
// .... init a, b and c
tnensor_type d = a + b - c;

```

Implement the resize methods

The next methods to define are the overloads of `resize`. `xtensor` provides utility 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 utility 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);
}

```

(continues on next page)

```

template <class T>
void resize(const shape_type& shape, const strides_type& strides)
{
    m_raw.m_shape = shape;
    m_raw.m_strides = strides;
    m_raw.m_backstrides.resize(shape.size());
    adapt_strides(m_raw.m_shape, m_raw.m_strides, m_raw.m_backstrides);
    m_raw.m_data.resize(compute_size(m_shape));
}

```

Implement private accessors

xcontainer assume the following methods are implemented in its inheriting class:

```

inner_shape_type& shape_impl();
const inner_shape_type& shape_impl() const;

inner_strides_type& strides_impl();
const inner_strides_type& strides_impl() const;

inner_backstrides_type& backstrides_impl();
const inner_backstrides_type& backstrides_impl() const;

```

However, since xcontainer provides a public API for getting the shape and the strides, these methods should be declared protected or private and xcontainer should be declared as a friend class so that it can access them.

1.23.3 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>

```

(continues on next page)

(continued from previous page)

```

    const T& element(It first, It last) const;
};

// This is the adaptor we need to define to plug table in xtensor
template <class T>
class table_adaptor;

```

Define inner types

The following definitions are required:

```

template <class T>
struct xcontainer_inner_type<table_adaptor<T>>
{
    using temporary_type = xarray<T>;
};

template <class T>
struct xiterable_inner_types<table_adaptor<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 type-defs.

```

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_type = inner_shape_type;
    using shape_type = inner_shape_type;
    using strides_type = inner_strides_type;

    using iterable_base = xexpression_iterable<self_type>;

```

(continues on next page)

(continued from previous page)

```

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 took 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 constructors and assignment operators. You also must define the 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...)
}

```

(continues on next page)

(continued from previous page)

```

template <class... Args>
const_reference operator() (Args... args) const
{
    // Should forward to table<T>::operator() (args...)
}

reference operator[](const xindex& index)
{
    return element(index.cbegin(), index.cend());
}

const_reference operator[](const xindex& index) const
{
    return element(index.cbegin(), index.cend());
}

reference operator[](size_type i)
{
    return operator() (i);
}

const_reference operator[](size_type i) const
{
    return operator() (i);
}

template <class It>
reference element(It first, It last)
{
    // Should forward to table<T>::element(first, last)
}

template <class It>
const_reference element(It first, It last)
{
    // Should forward to table<T>::element(first, last)
}

```

Implement broadcast mechanic

This part is relatively straightforward:

```

size_type dimension() const
{
    return shape().size();
}

const shape_type& shape() const
{
    // Should forward to table<T>::shape()
}

template <class S>
bool broadcast_shape(const S& s) const

```

(continues on next page)

```
{
    // 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);
}
```

1.24 Releasing xtensor

1.24.1 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 `CMakeLists.txt`, update the version of the dependencies and the corresponding variables, e.g. `xtl_REQUIRED_VERSION`.
- In file `environment.yml`, update the version of the dependencies including `xtensor`.
- 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`)

1.24.2 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.

1.24.3 Updating the stable branch

Once the conda-forge package has been updated, update the `stable` branch to the newly added tag.

1.25 From numpy to xtensor

1.25.1 Containers

Two container types are provided. `xarray` (dynamic number of dimensions) and `xtensor` (static number of dimensions).

Python 3 - numpy	C++ 14 - xtensor
<code>np.array([[3, 4], [5, 6]])</code>	<code>xt::xarray<double>({{3, 4}, {5, 6}})</code> <code>xt::xtensor<double, 2>({{3, 4}, {5, 6}})</code>
<code>arr.reshape([3, 4])</code>	<code>arr.reshape({3, 4})</code>
<code>arr.astype(np.float64)</code>	<code>xt::cast<double>(arr)</code>

1.25.2 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
<code>np.linspace(1.0, 10.0, 100)</code>	<code>xt::linspace<double>(1.0, 10.0, 100)</code>
<code>np.logspace(2.0, 3.0, 4)</code>	<code>xt::logspace<double>(2.0, 3.0, 4)</code>
<code>np.arange(3, 7)</code>	<code>xt::arange(3, 7)</code>
<code>np.eye(4)</code>	<code>xt::eye(4)</code>
<code>np.zeros([3, 4])</code>	<code>xt::zeros<double>({3, 4})</code>
<code>np.ones([3, 4])</code>	<code>xt::ones<double>({3, 4})</code>
<code>np.empty([3, 4])</code>	<code>xt::empty<double>({3, 4})</code>
<code>np.meshgrid(x0, x1, x2, indexing='ij')</code>	<code>xt::meshgrid(x0, x1, x2)</code>

xtensor's meshgrid implementation corresponds to numpy's 'ij' indexing order.

1.25.3 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
<code>a[:, np.newaxis]</code> <code>a[:5, 1:]</code> <code>a[5:1:-1, :]</code> <code>a[..., 3]</code>	<code>xt::view(a, xt::all(), xt::newaxis())</code> <code>xt::view(a, xt::range(_, 5),</code> <code>xt::range(1, _))</code> <code>xt::view(a, xt::range(5, 1, -1),</code> <code>xt::all())</code> <code>xt::strided_view(a, {xt::ellipsis,</code> <code>3})</code>
<code>np.broadcast(a, [4, 5, 7])</code>	<code>xt::broadcast(a, {4, 5, 7})</code>
<code>np.vectorize(f)</code>	<code>xt::vectorize(f)</code>
<code>a[a > 5]</code>	<code>xt::filter(a, a > 5)</code>
<code>a[[0, 1], [0, 0]]</code>	<code>xt::index_view(a, {{0, 0}, {1, 0}})</code>

1.25.4 Random

The random module provides simple ways to create random tensor expressions, lazily.

Python 3 - numpy	C++ 14 - xtensor
<code>np.random.seed(0)</code>	<code>xt::random::seed(0)</code>
<code>np.random.randn(10, 10)</code>	<code>xt::random::randn<double>({10, 10})</code>
<code>np.random.randint(10, 10)</code>	<code>xt::random::randint<int>({10, 10})</code>
<code>np.random.rand(3, 4)</code>	<code>xt::random::rand<double>({3, 4})</code>
<code>np.random.choice(arr, 5)</code>	<code>xt::random::choice(arr, 5)</code>
<code>np.random.shuffle(arr)</code>	<code>xt::random::shuffle(arr)</code>
<code>np.random.permutation(30)</code>	<code>xt::random::permutation(30)</code>

1.25.5 Concatenation, splitting, squeezing

Concatenating expressions does not allocate memory, it returns a tensor or view expression holding closures on the specified arguments.

Python 3 - numpy	C++ 14 - xtensor
<code>np.stack([a, b, c], axis=1)</code>	<code>xt::stack(xtuple(a, b, c), 1)</code>
<code>np.concatenate([a, b, c], axis=1)</code>	<code>xt::concatenate(xtuple(a, b, c), 1)</code>
<code>np.squeeze(a)</code>	<code>xt::squeeze(a)</code>
<code>np.expand_dims(a, 1)</code>	<code>xt::expand_dims(a, 1)</code>
<code>np.atleast_3d(a)</code>	<code>xt::atleast_3d(a)</code>
<code>np.split(a, 4, axis=0)</code>	<code>xt::split(a, 4, 0)</code>

1.25.6 Rearrange elements

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
<code>np.diag(a)</code>	<code>xt::diag(a)</code>
<code>np.diagonal(a)</code>	<code>xt::diagonal(a)</code>
<code>np.triu(a)</code>	<code>xt::triu(a)</code>
<code>np.tril(a, k=1)</code>	<code>xt::tril(a, 1)</code>
<code>np.flip(a, axis=3)</code>	<code>xt::flip(a, 3)</code>
<code>np.flipud(a)</code>	<code>xt::flip(a, 0)</code>
<code>np.fliplr(a)</code>	<code>xt::flip(a, 1)</code>
<code>np.transpose(a, (1, 0, 2))</code>	<code>xt::transpose(a, {1, 0, 2})</code>
<code>np.rot90(a)</code>	<code>xt::rot90(a)</code>
<code>np.rot90(a, 2, (1, 2))</code>	<code>xt::rot90<2>(a, {1, 2})</code>

1.25.7 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
<code>for x in np.nditer(a):</code>	<code>for(auto it=a.begin(); it!=a.end(); ++it)</code>
Iterating over a with a prescribed broadcasting shape	<code>a.begin({3, 4})</code> <code>a.end({3, 4})</code>
Iterating over a in a row-major fashion	<code>a.begin<xt::layout_type::row_major>()</code> <code>a.begin<xt::layout_type::row_major>()</code>
Iterating over a in a column-major fashion	<code>a.</code> <code>begin<xt::layout_type::column_major>()</code> <code>a.</code> <code>end<xt::layout_type::column_major>()</code>

1.25.8 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
<code>np.where(a > 5, a, b)</code>	<code>xt::where(a > 5, a, b)</code>
<code>np.where(a > 5)</code>	<code>xt::where(a > 5)</code>
<code>np.argwhere(a > 5)</code>	<code>xt::argwhere(a > 5)</code>
<code>np.any(a)</code>	<code>xt::any(a)</code>
<code>np.all(a)</code>	<code>xt::all(a)</code>
<code>np.logical_and(a, b)</code>	<code>a && b</code>
<code>np.logical_or(a, b)</code>	<code>a b</code>
<code>np.isclose(a, b)</code>	<code>xt::isclose(a, b)</code>
<code>np.allclose(a, b)</code>	<code>xt::allclose(a, b)</code>

1.25.9 Indices

Python 3 - numpy	C++ 14 - xtensor
<code>np.ravel_multi_index(indices, a.shape)</code>	<code>xt::ravel_indices(indices, a.shape())</code>

1.25.10 Comparisons

Python 3 - numpy	C++ 14 - xtensor
<code>np.equal(a, b)</code>	<code>xt::equal(a, b)</code>
<code>np.not_equal(a, b)</code>	<code>xt::not_equal(a, b)</code>
<code>np.less(a, b)</code>	<code>xt::less(a, b)</code> <code>a < b</code>
<code>np.less_equal(a, b)</code>	<code>xt::less_equal(a, b)</code> <code>a <= b</code>
<code>np.greater(a, b)</code>	<code>xt::greater(a, b)</code> <code>a > b</code>
<code>np.greater_equal(a, b)</code>	<code>xt::greater_equal(a, b)</code> <code>a >= b</code>
<code>np.nonzero(a)</code>	<code>xt::nonzero(a)</code>
<code>np.flatnonzero(a)</code>	<code>xt::flatnonzero(a)</code>

1.25.11 Minimum, Maximum, Sorting

Python 3 - numpy	C++ 14 - xtensor
<code>np.amin(a)</code>	<code>xt::amin(a)</code>
<code>np.amax(a)</code>	<code>xt::amax(a)</code>
<code>np.argmin(a)</code>	<code>xt::argmin(a)</code>
<code>np.argmax(a, axis=1)</code>	<code>xt::argmax(a, 1)</code>
<code>np.sort(a, axis=1)</code>	<code>xt::sort(a, 1)</code>
<code>np.argsort(a, axis=1)</code>	<code>xt::argsort(a, 1)</code>
<code>np.unique(a)</code>	<code>xt::unique(a)</code>
<code>np.setdiff1d(ar1, ar2)</code>	<code>xt::setdiff1d(ar1, ar2)</code>
<code>np.diff(a[, n, axis])</code>	<code>xt::diff(a[, n, axis])</code>

1.25.12 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
<code>np.real(a)</code>	<code>xt::real(a)</code>
<code>np.imag(a)</code>	<code>xt::imag(a)</code>
<code>np.conj(a)</code>	<code>xt::conj(a)</code>

- 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 a 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())`.

1.25.13 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 expressions don't hold any values and are computed upon access or assignment.

Python 3 - numpy	C++ 14 - xtensor
<code>np.sum(a, axis=[0, 1])</code>	<code>xt::sum(a, {0, 1})</code>
<code>np.sum(a, axis=1)</code>	<code>xt::sum(a, 1)</code>
<code>np.sum(a)</code>	<code>xt::sum(a)</code>
<code>np.prod(a, axis=[0, 1])</code>	<code>xt::prod(a, {0, 1})</code>
<code>np.prod(a, axis=1)</code>	<code>xt::prod(a, 1)</code>
<code>np.prod(a)</code>	<code>xt::prod(a)</code>
<code>np.mean(a, axis=[0, 1])</code>	<code>xt::mean(a, {0, 1})</code>
<code>np.mean(a, axis=1)</code>	<code>xt::mean(a, 1)</code>
<code>np.mean(a)</code>	<code>xt::mean(a)</code>
<code>np.std(a, [axis])</code>	<code>xt::stddev(a, [axis])</code>
<code>np.var(a, [axis])</code>	<code>xt::variance(a, [axis])</code>
<code>np.trapz(a, dx=2.0, axis=-1)</code> <code>np.trapz(a, x=b, axis=-1)</code>	<code>xt::trapz(a, 2.0, -1)</code> <code>xt::trapz(a, b, -1)</code>
<code>np.count_nonzero(a, axis=[0, 1])</code>	<code>xt::count_nonzero(a, {0, 1})</code>
<code>np.count_nonzero(a, axis=1)</code>	<code>xt::count_nonzero(a, 1)</code>
<code>np.count_nonzero(a)</code>	<code>xt::count_nonzero(a)</code>

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 commutative and associative up to rounding errors.

1.25.14 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
<code>np.set_printoptions(precision=4)</code>	<code>xt::print_options::set_precision(4)</code>
<code>np.set_printoptions(threshold=5)</code>	<code>xt::print_options::set_threshold(5)</code>
<code>np.set_printoptions(edgeitems=3)</code>	<code>xt::print_options::set_edgeitems(3)</code>
<code>np.set_printoptions(linewidth=100)</code>	<code>xt::print_options::set_line_width(100)</code>

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
<code>np.load(filename)</code>	<code>xt::load_npy<double>(filename)</code>
<code>np.save(filename, arr)</code>	<code>xt::dump_npy(filename, arr)</code>
<code>np.load_txt(filename, delimiter=',')</code>	<code>xt::load_csv<double>(stream)</code>

1.25.15 Mathematical functions

xtensor universal functions are provided for a large set number of mathematical functions.

Basic functions:

Python 3 - numpy	C++ 14 - xtensor
<code>np.absolute(a)</code>	<code>xt::abs(a)</code>
<code>np.sign(a)</code>	<code>xt::sign(a)</code>
<code>np.remainder(a, b)</code>	<code>xt::remainder(a, b)</code>
<code>np.clip(a, min, max)</code>	<code>xt::clip(a, min, max)</code>
	<code>xt::fma(a, b, c)</code>
<code>np.interp(x, xp, fp, [,left, right])</code>	<code>xt::interp(x, xp, fp, [,left, right])</code>

Exponential functions:

Python 3 - numpy	C++ 14 - xtensor
<code>np.exp(a)</code>	<code>xt::exp(a)</code>
<code>np.expm1(a)</code>	<code>xt::expm1(a)</code>
<code>np.log(a)</code>	<code>xt::log(a)</code>
<code>np.log1p(a)</code>	<code>xt::log1p(a)</code>

Power functions:

Python 3 - numpy	C++ 14 - xtensor
<code>np.power(a, p)</code>	<code>xt::pow(a, b)</code>
<code>np.sqrt(a)</code>	<code>xt::sqrt(a)</code>
<code>np.square(a)</code>	<code>xt::square(a) xt::cube(a)</code>
<code>np.cbrt(a)</code>	<code>xt::cbrt(a)</code>

Trigonometric functions:

Python 3 - numpy	C++ 14 - xtensor
<code>np.sin(a)</code>	<code>xt::sin(a)</code>
<code>np.cos(a)</code>	<code>xt::cos(a)</code>
<code>np.tan(a)</code>	<code>xt::tan(a)</code>

Hyperbolic functions:

Python 3 - numpy	C++ 14 - xtensor
<code>np.sinh(a)</code>	<code>xt::sinh(a)</code>
<code>np.cosh(a)</code>	<code>xt::cosh(a)</code>
<code>np.tanh(a)</code>	<code>xt::tanh(a)</code>

Error and gamma functions:

Python 3 - numpy	C++ 14 - xtensor
<code>scipy.special.erf(a)</code>	<code>xt::erf(a)</code>
<code>scipy.special.gamma(a)</code>	<code>xt::tgamma(a)</code>
<code>scipy.special.gammaln(a)</code>	<code>xt::lgamma(a)</code>

Classification functions:

Python 3 - numpy	C++ 14 - xtensor
<code>np.isnan(a)</code>	<code>xt::isnan(a)</code>
<code>np.isinf(a)</code>	<code>xt::isinf(a)</code>
<code>np.isfinite(a)</code>	<code>xt::isfinite(a)</code>

Histogram:

Python 3 - numpy	C++ 14 - xtensor
<code>np.histogram(a, bins[, weights][, density])</code>	<code>xt::histogram(a, bins[, weights][, density])</code>
<code>np.histogram_bin_edges(a, bins[, weights][, left, right][, bins][, mode])</code>	<code>xt::histogram_bin_edges(a, bins[, weights][, left, right][, bins][, mode])</code>
<code>np.bincount(arr)</code>	<code>xt::bincount(arr)</code>

1.25.16 Linear algebra

Many functions found in the `numpy.linalg` module are implemented in `xtensor-blas`, a separate 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, vector and tensor products

Python 3 - numpy	C++ 14 - xtensor
<code>np.dot(a, b)</code>	<code>xt::linalg::dot(a, b)</code>
<code>np.vdot(a, b)</code>	<code>xt::linalg::vdot(a, b)</code>
<code>np.outer(a, b)</code>	<code>xt::linalg::outer(a, b)</code>
<code>np.matrix_power(a, 123)</code>	<code>xt::linalg::matrix_power(a, 123)</code>
<code>np.kron(a, b)</code>	<code>xt::linalg::kron(a, b)</code>
<code>np.tensordot(a, b, axes=3)</code>	<code>xt::linalg::tensordot(a, b, 3)</code>
<code>np.tensordot(a, b, axes=((0,2), (1, 3)))</code>	<code>xt::linalg::tensordot(a, b, {0, 2}, {1, 3})</code>

Decompositions

Python 3 - numpy	C++ 14 - xtensor
<code>np.linalg.cholesky(a)</code>	<code>xt::linalg::cholesky(a)</code>
<code>np.linalg.qr(a)</code>	<code>xt::linalg::qr(a)</code>
<code>np.linalg.svd(a)</code>	<code>xt::linalg::svd(a)</code>

Matrix eigenvalues

Python 3 - numpy	C++ 14 - xtensor
<code>np.linalg.eig(a)</code>	<code>xt::linalg::eig(a)</code>
<code>np.linalg.eigvals(a)</code>	<code>xt::linalg::eigvals(a)</code>
<code>np.linalg.eigh(a)</code>	<code>xt::linalg::eigh(a)</code>
<code>np.linalg.eigvalsh(a)</code>	<code>xt::linalg::eigvalsh(a)</code>

Norms and other numbers

Python 3 - numpy	C++ 14 - xtensor
<code>np.linalg.norm(a, order=2)</code>	<code>xt::linalg::norm(a, 2)</code>
<code>np.linalg.cond(a)</code>	<code>xt::linalg::cond(a)</code>
<code>np.linalg.det(a)</code>	<code>xt::linalg::det(a)</code>
<code>np.linalg.matrix_rank(a)</code>	<code>xt::linalg::matrix_rank(a)</code>
<code>np.linalg.slogdet(a)</code>	<code>xt::linalg::slogdet(a)</code>
<code>np.trace(a)</code>	<code>xt::linalg::trace(a)</code>

Solving equations and inverting matrices

Python 3 - numpy	C++ 14 - xtensor
<code>np.linalg.inv(a)</code>	<code>xt::linalg::inv(a)</code>
<code>np.linalg.pinv(a)</code>	<code>xt::linalg::pinv(a)</code>
<code>np.linalg.solve(A, b)</code>	<code>xt::linalg::solve(A, b)</code>
<code>np.linalg.lstsq(A, b)</code>	<code>xt::linalg::lstsq(A, b)</code>

1.26 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 between numpy and xtensor.

1.26.1 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 implicitly 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`.

Accumulators (`cumsum`, `cumprod`) throw an exception if an axis argument is passed and the array argument is a 0-D argument:

1.26.2 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.

1.26.3 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.

1.26.4 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*.

1.26.5 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.

1.26.6 Array indices

Array indices are in xtensor stored as a `std::vector` of array indices, whereby each entry corresponds to the array indices of one item. This results in a slightly different usage of `xt::ravel_indices` than of `np.ravel_multi_index`.

1.27 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 deferred 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 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 metafunctions complement the tools of the standard library for the move semantics.

1.27.1 Basic rules for determining 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_type`, and `const_closure_type`. The latter adds the `const` qualifier to the reference even when the provided argument is not `const`.

```
template <class S>
struct closure_type
{
    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_type_t = typename closure_type<S>::type;
```

The implementation for `const_closure_type` is slightly shorter.

```
template <class S>
struct const_closure_type
{
    using underlying_type = std::decay_t<S>;
```

(continues on next page)

```

using type = typename std::conditional<
    std::is_lvalue_reference<S>::value,
    std::add_const_t<underlying_type>&,
    underlying_type>::type;
};

template <class S>
using const_closure_type_t = typename const_closure_type<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.

1.27.2 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, it actually holds a closure type on the scalar value.

The logic 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_type_t<E>;
};

template <class E>
struct const_xclosure<E, disable_xexpression<std::decay_t<E>>>
{
    using type = xscalar<std::decay_t<E>>;
};

```

(continues on next page)

(continued from previous page)

```
template <class E>
using const_xclosure_t = typename const_xclosure<E>::type;
```

1.27.3 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 use case, 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.
- If you would need to *move* more than once, take a look at the [Reusing expressions / sharing expressions](#).

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 assignment.

- 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.

Reusing expressions / sharing expressions

Sometimes it is necessary to use a xexpression in two separate places in another xexpression. For example, when computing something like `sin(A) + cos(A)` we can see `A` being referenced twice. This works fine if we can guarantee that `A` has a long enough lifetime. However, when writing generic interfaces that accept rvalues we cannot always guarantee that `A` will live long enough. Another scenario is the creation of a temporary, which needs to be used at more than one place in the resulting expression. We can only `std::move(...)` the temporary once into the expression to hand lifetime management to the expression.

In order to solve this problem, xtensor offers two solutions: the first involves ad-hoc lambda construction and the second utilizes shared pointers wrapped in a `xshared_expression`.

We can rewrite the `sin(A) + cos(A)` function as a lambda that we use to create a vectorized xfunction, and xtensor has a simple utility to achieve this:

```
template <class E>
inline auto sin_plus_cos(E&& e) noexcept
{
    auto func = [] (auto x) -> decltype(sin(x) + cos(x)) {
        return sin(x) + cos(x);
    };
    return detail::make_lambda_function(std::move(func), std::forward<E>(e));
}
```

Note: writing a lambda is just sugar for writing a functor. Also, using `auto x` as the function argument enables automatic `xsimd` acceleration.

As the data flow through the lambda is entirely transparent to the compiler, using this construct is generally faster than using `xshared_expressions`. The usage of `xshared_expression` also requires the creation of a `shared_ptr` which dynamically allocates some memory and is therefore slow(ish). But under certain circumstances it might be required, e.g. to implement a fully lazy average:

```
template <class E, class W>
inline auto average(E&& e, W&& weights, std::ptrdiff_t axis) noexcept
{
    auto shared_weights = xt::make_xshared(std::move(weights));
    auto expr = xt::sum(e * shared_weights, {axis}) / xt::sum(shared_weights);
    // the following line prints how often shared_weights is used
    std::cout << shared_weights.use_count() << std::endl; // ==> 3
    return expr;
}
```

We can see that, before returning from the function, three copies of `shared_weights` exist: two in the two `xt::sum` functions, and one is the temporary. After returning from the function, only two copies of the `xshared_expression` will exist. As discussed before, `xt::make_xshared` has the same overhead as creating a `std::shared_ptr` which is used internally by the shared expression.

1.28 Related projects

1.28.1 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 in-place.

Example 1: Use an algorithm of the C++ library on a `numpy` array in-place

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>

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]]

```

1.28.2 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

1.28.3 xtensor-julia

The `xtensor-julia` project provides the implementation of container types compatible with `xtensor`'s expression system, `jarray` and `jltensor` which effectively wrap Julia arrays, allowing operating on Julia arrays in-place.

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)
```

(continues on next page)

```
{
    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]]
```

1.28.4 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

1.28.5 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 in-place.

Example 1: Use an algorithm of the C++ library on a R array in-place

C++ code

```

#include <numeric> // Standard library import for std::accumulate
#include "xtensor/xmath.hpp" // xtensor import for the C++ universal_
↪functions
#include "xtensor-r/rarray.hpp" // R bindings
#include <Rcpp.h>

using namespace Rcpp;

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

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

```

R code

```

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

```

Outputs

```
1.2853996391883833
```

1.28.6 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.

1.28.7 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.

Example 1: Calculate a derivative in Fourier space

Calculate the derivative of a (discretized) field in Fourier space, e.g. a sine shaped field `sin`:

C++ code

```

#include <xtensor-fftw/basic.hpp> // rfft, irfft
#include <xtensor-fftw/helper.hpp> // rfftscale
#include <xtensor/xarray.hpp>
#include <xtensor/xbuilder.hpp> // xt::arange
#include <xtensor/xmath.hpp> // xt::sin, cos
#include <complex>

```

(continues on next page)

```

#include <xtensor/xio.hpp>

// generate a sinusoid field
double dx = M_PI / 100;
xt::xarray<double> x = xt::arange(0., 2 * M_PI, dx);
xt::xarray<double> sin = xt::sin(x);

// transform to Fourier space
auto sin_fs = xt::fftw::rfft(sin);

// multiply by i*k
std::complex<double> i {0, 1};
auto k = xt::fftw::rfftscale<double>(sin.shape()[0], dx);
xt::xarray<std::complex<double>> sin_derivative_fs = xt::eval(i * k * sin_fs);

// transform back to normal space
auto sin_derivative = xt::fftw::irfft(sin_derivative_fs);

std::cout << "x:          " << x << std::endl;
std::cout << "sin:         " << sin << std::endl;
std::cout << "cos:         " << xt::cos(x) << std::endl;
std::cout << "sin_derivative: " << sin_derivative << std::endl;

```

Outputs

```

x:          { 0.          , 0.031416, 0.062832, 0.094248, ..., 6.251769}
sin:       { 0.000000e+00, 3.141076e-02, 6.279052e-02, 9.410831e-02, ..., -3.
↳141076e-02}
cos:       { 1.000000e+00, 9.995066e-01, 9.980267e-01, 9.955620e-01, ..., 9.
↳995066e-01}
sin_derivative: { 1.000000e+00, 9.995066e-01, 9.980267e-01, 9.955620e-01, ..., 9.
↳995066e-01}

```

1.28.8 xtensor-io

The `xtensor-io` project is an extension to the `xtensor` library for reading and writing image, sound and npz file formats to and from `xtensor` data structures.

1.28.9 xtensor-ros

The `xtensor-ros` project is an extension to the `xtensor` library providing helper functions to easily send and receive `xtensor` and `xarray` datastructures as ROS messages.

1.28.10 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.

1.28.11 xtl

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

1.29 Designing language bindings with xtensor

`xtensor` and its *Related projects* make it easy to implement a feature once in C++ and expose it to the main languages of data science, such as Python, Julia and R with little extra work. Although, if that sounds simple in principle, difficulties may appear when it comes to define the API of the C++ library. The following illustrates the different options we have with the case of a single function `compute` that must be callable from all the languages.

1.29.1 Generic API

Since the `xtensor` bindings provide different container types for holding tensors (`pytensor`, `rtensor` and `jtensor`), if we want our function to be callable from all the languages, it must accept a generic argument:

```
template <class E>
void compute (E&& e);
```

However, this is a bit too generic and we may want to enforce that this function only accepts `xtensor` arguments. Since all `xtensor` containers inherit from the “xexpression” CRTP base class, we can easily express that constraint with the following signature:

```
template <class E>
void compute (const xexpression<E>& e)
{
    // Now the implementation must use e() instead of e
}
```

Notice that with this change, we lose the ability to call the function with non-constant references or rvalue references. If we want them back, we need to add the following overloads:

```
template <class E>
void compute (xexpression<E>& e);

template <class E>
void compute (xexpression<E>&& e);
```

In the following we assume that the constant reference overload is enough. We can now expose the `compute` function to the other languages, let’s illustrate this with Python bindings:

```
PYBIND11_MODULE (pymod, m)
{
    xt::import_numpy ();

    m.def ("compute", &compute<pytensor<double, 2>>);
}
```

1.29.2 Full qualified API

Accepting any kind of expression can still be too permissive; assume we want to restrict this function to 2-dimensional tensor containers only. In that case, a solution is to provide an API function that forwards the call to a common generic implementation:

```
namespace detail
{
    template <class E>
    void compute_impl(E&&);
}

template <class T>
void compute(const xtensor<T, 2>& t)
{
    detail::compute_impl(t);
}
```

Exposing it to the Python is just as simple:

```
template <class T>
void compute(const pytensor<T, 2>& t)
{
    detail::compute_impl(t);
}

PYBIND11_MODULE(pymod, m)
{
    xt::import_numpy();

    m.def("compute", &compute<double>);
}
```

Although this solution is really simple, it requires writing four additional functions for the API. Besides, if later, you decide to support array containers, you need to add four more functions. Therefore this solution should be considered for libraries with a small number of functions to expose, and whose APIs are unlikely to change in the future.

1.29.3 Container selection

A way to keep the restriction on the parameter type while limiting the required amount of typing in the bindings is to rely on additional structures that will “select” the right type for us.

The idea is to define a structure for selecting the type of containers (tensor, array) and a structure to select the library implementation of that container (xtensor, pytensor in the case of a tensor container):

```
// library container selector
struct xtensor_c
{
};

// container selector, must be specialized for each
// library container selector
template <class C, class T, std::size_t N>
struct tensor_container;

// Specialization for xtensor library (or C++)
```

(continues on next page)

(continued from previous page)

```

template <class T, std::size_t N>
struct tensor_container<xtensor_c, T, N>
{
    using type = xt::xtensor<T, N>;
};

template <class C, class T, std::size_t N>
using tensor_container_t = typename tensor_container<C, T, N>::type;

```

The function signature then becomes

```

template <class T, class C = xtensor_c>
void compute(const tensor_container_t<C, T, 2>& t);

```

The Python bindings only require that we specialize the `tensor_container` structure

```

struct pytensor_c
{
};

template <class T, std::size_t N>
struct tensor_container<pytensor_c, T, N>
{
    using type = pytensor<T, N>;
};

PYBIND11_MODULE(pymod, m)
{
    xt::import_numpy();

    m.def("compute", &compute<double, pytensor_c>);
}

```

Even if we need to specialize the “`tensor_container`” structure for each language, the specialization can be reused for other functions and thus reduce the amount of typing required. This comes at a cost though: we’ve lost type inference on the C++ side.

```

xt::xtensor<double, 2> t {{1., 2., 3.}, {4., 5., 6.}};

compute<double>(t); // works
compute(t);        // error (couldn't infer template argument 'T')

```

Besides, if later we want to support arrays, we need to add an “`array_container`” structure and its specializations, and an overload of the `compute` function:

```

template <class C, class T>
struct array_container;

template <class C, class T>
struct array_container<xtensor_c, T>
{
    using type = xt::xarray<T>;
};

template <class C, class T>
using array_container_t = typename array_container<C, T>::type;

```

(continues on next page)

```
template <class T, class C = xtensor_c>
void compute(const array_container_t<C, T>& t);
```

1.29.4 Type restriction with SFINAE

The major drawback of the previous option is the loss of type inference in C++. The only means to get it back is to reintroduce a generic parameter type. However, we can make the compiler generate an invalid type so the function is removed from the overload resolution set when the actual type of the argument does not satisfy some constraint. This principle is known as SFINAE (Substitution Failure Is Not An Error). Modern C++ provide metafunctions to help us make use of SFINAE:

```
template <class C>
struct is_tensor : std::false_type
{
};

template <class T, std::size_t N, layout_type L, class Tag>
struct is_tensor<xtensor<T, N, L, Tag>> : std::true_type
{
};

template <class T, template <class> class C = is_tensor,
          std::enable_if_t<C<T>::value, bool> = true>
void compute(const T& t);
```

Here when `C<T>::value` is true, the `enable_if_t` invocation generates the `bool` type. Otherwise, it does not generate anything, leading to an invalid function declaration. The compiler removes this declaration from the overload resolution set and no error happens if another “compute” overload is a good match for the call. Otherwise, the compiler emits an error.

The default value is here to avoid the need to pass a boolean value when invoking the `compute` function; this value is of no use, we only rely on the SFINAE trick.

This declaration has a slight problem: adding `enable_if_t` to the signature of each function we want to expose is cumbersome. Let’s make this part more expressive:

```
template <template<class> class C, class T>
using check_constraints = std::enable_if_t<C<T>::value, bool>;
template <class T, template <class> class C = is_tensor,
          check_constraints<C, T> = true>
void compute(const T& t);
```

All good, we have type inference and an expressive syntax for declaring our function. Besides, if we want to relax the constraint so the function can accept both tensors and arrays, all we have to do is to replace the default value for `C`:

```
// Equivalent to is_tensor<T>::value || is_array<T>::value
template <class T>
struct is_container : xtl::disjunction<is_tensor<T>, is_array<T>>
{
};

template <class T, template <class> class C = is_container,
          check_constraints<C, T> = true>
void compute(const T& t);
```

This is far more flexible than the previous option. This flexibility comes at a minor cost: exposing the function to the Python is slightly more verbose:

```
template <class T, std::size_t N, layout_type L>
struct is_tensor<pytensor<T, N, L>> : std::true_type
{
};

PYBIND11_MODULE(pymod, m)
{
    xt::import_numpy();

    m.def("compute", &compute<pytensor<double, 2>>);
}
```

1.29.5 Conclusion

Each solution has its pros and cons and choosing one of them should be done according to the flexibility you want to impose on your API and the constraints you are imposed by the implementation. For instance, a method that requires a lot of typing in the bindings might not suit for libraries with a huge amount of functions to expose, while a full generic API might be problematic if the implementation expects containers only. Below is a summary of the advantages and drawbacks of the different options:

- Generic API: full genericity, no additional typing required in the bindings, but maybe too permissive.
- Full qualified API: simple, accepts only the specified parameter type, but requires a lot of typing for the bindings.
- Container selection: quite simple, requires less typing than the previous method, but loses type inference on the C++ side and lacks some flexibility.
- Type restriction with SFINAE: more flexible than the previous option, gets type inference back, but slightly more complex to implement.

X

- `xt::abs` (C++ function), 152
- `xt::accumulate` (C++ function), 123
- `xt::acos` (C++ function), 159
- `xt::acosh` (C++ function), 160
- `xt::adapt` (C++ function), 79, 80, 85, 86
- `xt::all` (C++ function), 105, 145
- `xt::allclose` (C++ function), 164
- `xt::amax` (C++ function), 166
- `xt::amin` (C++ function), 167
- `xt::any` (C++ enumerator), 59
- `xt::any` (C++ function), 145
- `xt::arange` (C++ function), 128
- `xt::argmax` (C++ function), 135
- `xt::argmin` (C++ function), 135
- `xt::argsort` (C++ function), 134, 135
- `xt::argwhere` (C++ function), 151
- `xt::asin` (C++ function), 159
- `xt::asinh` (C++ function), 160
- `xt::atan` (C++ function), 159
- `xt::atan2` (C++ function), 159
- `xt::atanh` (C++ function), 161
- `xt::atleast_1d` (C++ function), 133
- `xt::atleast_2d` (C++ function), 133
- `xt::atleast_3d` (C++ function), 133
- `xt::atleast_Nd` (C++ function), 133
- `xt::automatic` (C++ enumerator), 138
- `xt::bincount` (C++ function), 138
- `xt::broadcast` (C++ function), 111
- `xt::cast` (C++ function), 150
- `xt::cbrt` (C++ function), 158
- `xt::ceil` (C++ function), 162
- `xt::clip` (C++ function), 154
- `xt::column_major` (C++ enumerator), 59
- `xt::compute_layout` (C++ function), 59
- `xt::concatenate` (C++ function), 129
- `xt::cos` (C++ function), 158
- `xt::cosh` (C++ function), 160
- `xt::cube` (C++ function), 157
- `xt::cumprod` (C++ function), 170, 171
- `xt::cumsum` (C++ function), 170
- `xt::diag` (C++ function), 130
- `xt::diagonal` (C++ function), 130
- `xt::diff` (C++ function), 166
- `xt::drop` (C++ function), 106
- `xt::dump_csv` (C++ function), 142
- `xt::dump_npy` (C++ function), 142
- `xt::dynamic` (C++ enumerator), 59
- `xt::ellipsis` (C++ function), 106
- `xt::empty` (C++ function), 126
- `xt::empty_like` (C++ function), 127
- `xt::equal` (C++ function), 147
- `xt::erf` (C++ function), 161
- `xt::erfc` (C++ function), 161
- `xt::eval` (C++ function), 58
- `xt::exp` (C++ function), 155
- `xt::exp2` (C++ function), 155
- `xt::expand_dims` (C++ function), 132
- `xt::expm1` (C++ function), 155
- `xt::eye` (C++ function), 127
- `xt::fabs` (C++ function), 152
- `xt::fdim` (C++ function), 154
- `xt::filter` (C++ function), 116
- `xt::filtration` (C++ function), 116
- `xt::flatten` (C++ function), 132
- `xt::flatten_indices` (C++ function), 83
- `xt::flip` (C++ function), 134
- `xt::floor` (C++ function), 162
- `xt::fma` (C++ function), 153
- `xt::fmax` (C++ function), 153
- `xt::fmin` (C++ function), 154
- `xt::fmod` (C++ function), 152
- `xt::from_indices` (C++ function), 83, 151
- `xt::from_json` (C++ function), 143
- `xt::full_like` (C++ function), 126
- `xt::greater` (C++ function), 148
- `xt::greater_equal` (C++ function), 148
- `xt::histogram` (C++ function), 138, 139
- `xt::histogram_algorithm` (C++ type), 138

`xt::histogram_bin_edges` (C++ function), 138–140
`xt::hypot` (C++ function), 158
`xt::imag` (C++ function), 117
`xt::index_view` (C++ function), 116
`xt::isclose` (C++ function), 164
`xt::isfinite` (C++ function), 163
`xt::isinf` (C++ function), 164
`xt::isnan` (C++ function), 164
`xt::keep` (C++ function), 106
`xt::layout_type` (C++ type), 59
`xt::left_shift` (C++ function), 149
`xt::less` (C++ function), 147
`xt::less_equal` (C++ function), 148
`xt::lgamma` (C++ function), 162
`xt::linspace` (C++ enumerator), 138
`xt::linspace` (C++ function), 128
`xt::load_csv` (C++ function), 142
`xt::load_npy` (C++ function), 142
`xt::log` (C++ function), 155
`xt::log10` (C++ function), 156
`xt::log1p` (C++ function), 156
`xt::log2` (C++ function), 156
`xt::logspace` (C++ enumerator), 138
`xt::logspace` (C++ function), 128
`xt::make_lambda_xfunction` (C++ function), 120
`xt::make_xshared` (C++ function), 52
`xt::maximum` (C++ function), 153
`xt::mean` (C++ function), 165
`xt::meshgrid` (C++ function), 129
`xt::minimum` (C++ function), 153
`xt::nan_to_num` (C++ function), 171
`xt::nancumprod` (C++ function), 172
`xt::nancumsum` (C++ function), 172
`xt::nanprod` (C++ function), 171
`xt::nansum` (C++ function), 171
`xt::nearbyint` (C++ function), 163
`xt::newaxis` (C++ function), 105
`xt::nonzero` (C++ function), 151
`xt::norm_induced_l1` (C++ function), 169
`xt::norm_induced_linf` (C++ function), 170
`xt::norm_l0` (C++ function), 167
`xt::norm_l1` (C++ function), 168
`xt::norm_l2` (C++ function), 168
`xt::norm_linf` (C++ function), 168
`xt::norm_lp` (C++ function), 169
`xt::norm_lp_to_p` (C++ function), 169
`xt::norm_sq` (C++ function), 168
`xt::not_equal` (C++ function), 147
`xt::ones` (C++ function), 126
`xt::ones_like` (C++ function), 127
`xt::operator!` (C++ function), 145
`xt::operator!=` (C++ function), 147
`xt::operator*` (C++ function), 144
`xt::operator+` (C++ function), 143, 144
`xt::operator-` (C++ function), 143, 144
`xt::operator/` (C++ function), 144
`xt::operator==` (C++ function), 146
`xt::operator&` (C++ function), 148
`xt::operator&&` (C++ function), 145
`xt::operator^` (C++ function), 149
`xt::operator~` (C++ function), 149
`xt::operator|` (C++ function), 148
`xt::operator||` (C++ function), 144
`xt::operator>` (C++ function), 146
`xt::operator>=` (C++ function), 146
`xt::operator>>` (C++ function), 150
`xt::operator<` (C++ function), 146
`xt::operator<=` (C++ function), 146
`xt::operator<<` (C++ function), 149
`xt::pow` (C++ function), 156, 157
`xt::print_options::edge_items` (C++ class), 141
`xt::print_options::line_width` (C++ class), 141
`xt::print_options::precision` (C++ class), 142
`xt::print_options::set_edge_items` (C++ function), 141
`xt::print_options::set_line_width` (C++ function), 141
`xt::print_options::set_precision` (C++ function), 141
`xt::print_options::set_threshold` (C++ function), 141
`xt::print_options::threshold` (C++ class), 141
`xt::prod` (C++ function), 165
`xt::random::choice` (C++ function), 137
`xt::random::get_default_random_engine` (C++ function), 135
`xt::random::permutation` (C++ function), 137
`xt::random::rand` (C++ function), 136
`xt::random::randint` (C++ function), 136
`xt::random::randn` (C++ function), 136
`xt::random::seed` (C++ function), 135
`xt::random::shuffle` (C++ function), 137
`xt::range` (C++ function), 105
`xt::ravel` (C++ function), 131
`xt::ravel_indices` (C++ function), 83
`xt::real` (C++ function), 117
`xt::reduce` (C++ function), 123
`xt::remainder` (C++ function), 152
`xt::reshape_view` (C++ function), 108
`xt::right_shift` (C++ function), 149
`xt::rint` (C++ function), 163
`xt::round` (C++ function), 163

- `xt::row_major` (C++ *enumerator*), 59
`xt::sign` (C++ *function*), 154
`xt::sin` (C++ *function*), 158
`xt::sinh` (C++ *function*), 160
`xt::sort` (C++ *function*), 134
`xt::split` (C++ *function*), 133
`xt::sqrt` (C++ *function*), 157
`xt::square` (C++ *function*), 157
`xt::squeeze` (C++ *function*), 132
`xt::stack` (C++ *function*), 129
`xt::stddev` (C++ *function*), 166
`xt::strided_view` (C++ *function*), 107, 108
`xt::sum` (C++ *function*), 165
`xt::tan` (C++ *function*), 159
`xt::tanh` (C++ *function*), 160
`xt::tgamma` (C++ *function*), 161
`xt::to_json` (C++ *function*), 143
`xt::transpose` (C++ *function*), 131
`xt::trapz` (C++ *function*), 167
`xt::tril` (C++ *function*), 130
`xt::trim_zeros` (C++ *function*), 132
`xt::triu` (C++ *function*), 131
`xt::trunc` (C++ *function*), 162
`xt::uniform` (C++ *enumerator*), 138
`xt::unique` (C++ *function*), 135
`xt::variance` (C++ *function*), 165
`xt::view` (C++ *function*), 105
`xt::where` (C++ *function*), 145, 151
`xt::xarray` (C++ *type*), 77
`xt::xarray_adaptor` (C++ *class*), 78
`xt::xarray_adaptor::operator=` (C++ *function*), 79
`xt::xarray_adaptor::xarray_adaptor` (C++ *function*), 78
`xt::xarray_container` (C++ *class*), 75
`xt::xarray_container::from_shape` (C++ *function*), 77
`xt::xarray_container::operator=` (C++ *function*), 77
`xt::xarray_container::xarray_container` (C++ *function*), 75–77
`xt::xarray_optional` (C++ *type*), 77
`xt::xbroadcast` (C++ *class*), 108
`xt::xbroadcast::at` (C++ *function*), 110
`xt::xbroadcast::broadcast_shape` (C++ *function*), 111
`xt::xbroadcast::dimension` (C++ *function*), 109
`xt::xbroadcast::element` (C++ *function*), 110
`xt::xbroadcast::expression` (C++ *function*), 109
`xt::xbroadcast::has_linear_assign` (C++ *function*), 111
`xt::xbroadcast::layout` (C++ *function*), 109
`xt::xbroadcast::operator()` (C++ *function*), 109
`xt::xbroadcast::operator[]` (C++ *function*), 110
`xt::xbroadcast::shape` (C++ *function*), 109
`xt::xbroadcast::size` (C++ *function*), 109
`xt::xbroadcast::unchecked` (C++ *function*), 110
`xt::xbroadcast::xbroadcast` (C++ *function*), 109
`xt::xconst_iterable` (C++ *class*), 64
`xt::xconst_iterable::begin` (C++ *function*), 65, 66
`xt::xconst_iterable::cbegin` (C++ *function*), 65, 66
`xt::xconst_iterable::cend` (C++ *function*), 65, 66
`xt::xconst_iterable::crbegin` (C++ *function*), 65, 67
`xt::xconst_iterable::crend` (C++ *function*), 66, 67
`xt::xconst_iterable::end` (C++ *function*), 65, 66
`xt::xconst_iterable::rbegin` (C++ *function*), 65, 67
`xt::xconst_iterable::rend` (C++ *function*), 65, 67
`xt::xcontainer` (C++ *class*), 59
`xt::xcontainer::at` (C++ *function*), 61
`xt::xcontainer::backstrides` (C++ *function*), 60
`xt::xcontainer::broadcast_shape` (C++ *function*), 63
`xt::xcontainer::data` (C++ *function*), 60
`xt::xcontainer::data_offset` (C++ *function*), 60
`xt::xcontainer::dimension` (C++ *function*), 60
`xt::xcontainer::element` (C++ *function*), 62
`xt::xcontainer::fill` (C++ *function*), 60
`xt::xcontainer::has_linear_assign` (C++ *function*), 63
`xt::xcontainer::operator()` (C++ *function*), 60
`xt::xcontainer::operator[]` (C++ *function*), 62
`xt::xcontainer::shape` (C++ *function*), 60
`xt::xcontainer::size` (C++ *function*), 60
`xt::xcontainer::storage` (C++ *function*), 60
`xt::xcontainer::strides` (C++ *function*), 60
`xt::xcontainer::unchecked` (C++ *function*), 61, 62
`xt::xcontainer_semantic` (C++ *class*), 57
`xt::xcontainer_semantic::assign_temporary` (C++ *function*), 58

`xt::xcontiguous_iterable (C++ class), 70`
`xt::xcontiguous_iterable::begin (C++ function), 70, 71`
`xt::xcontiguous_iterable::cbegin (C++ function), 70, 72`
`xt::xcontiguous_iterable::cend (C++ function), 71, 72`
`xt::xcontiguous_iterable::crbegin (C++ function), 73, 74`
`xt::xcontiguous_iterable::crend (C++ function), 73, 74`
`xt::xcontiguous_iterable::end (C++ function), 70–72`
`xt::xcontiguous_iterable::rbegin (C++ function), 72–74`
`xt::xcontiguous_iterable::rend (C++ function), 72–74`
`xt::xexpression (C++ class), 51`
`xt::xexpression::derived_cast (C++ function), 52`
`xt::xfiltration (C++ class), 114`
`xt::xfiltration::operator*=(C++ function), 115`
`xt::xfiltration::operator+=(C++ function), 115`
`xt::xfiltration::operator-=(C++ function), 115`
`xt::xfiltration::operator/=(C++ function), 115`
`xt::xfiltration::operator=(C++ function), 114`
`xt::xfiltration::operator%=(C++ function), 115`
`xt::xfiltration::xfiltration (C++ function), 115`
`xt::xfixed_container (C++ class), 87`
`xt::xfixed_container::operator=(C++ function), 88`
`xt::xfixed_container::reshape (C++ function), 89`
`xt::xfixed_container::resize (C++ function), 88, 89`
`xt::xfixed_container::xfixed_container (C++ function), 88`
`xt::xfunction (C++ class), 118`
`xt::xfunction::at (C++ function), 118`
`xt::xfunction::broadcast_shape (C++ function), 119`
`xt::xfunction::dimension (C++ function), 118`
`xt::xfunction::element (C++ function), 119`
`xt::xfunction::has_linear_assign (C++ function), 120`
`xt::xfunction::layout (C++ function), 118`
`xt::xfunction::operator() (C++ function), 118`
`xt::xfunction::shape (C++ function), 118`
`xt::xfunction::size (C++ function), 118`
`xt::xfunction::unchecked (C++ function), 119`
`xt::xfunction::xfunction (C++ function), 118`
`xt::xfunctor_view (C++ class), 117`
`xt::xfunctor_view::operator=(C++ function), 117`
`xt::xgenerator (C++ class), 124`
`xt::xgenerator::at (C++ function), 124`
`xt::xgenerator::broadcast_shape (C++ function), 125`
`xt::xgenerator::dimension (C++ function), 124`
`xt::xgenerator::element (C++ function), 125`
`xt::xgenerator::has_linear_assign (C++ function), 126`
`xt::xgenerator::operator() (C++ function), 124`
`xt::xgenerator::shape (C++ function), 124`
`xt::xgenerator::size (C++ function), 124`
`xt::xgenerator::unchecked (C++ function), 125`
`xt::xgenerator::xgenerator (C++ function), 124`
`xt::xindex_view (C++ class), 111`
`xt::xindex_view::broadcast_shape (C++ function), 114`
`xt::xindex_view::dimension (C++ function), 112`
`xt::xindex_view::element (C++ function), 113`
`xt::xindex_view::expression (C++ function), 113`
`xt::xindex_view::fill (C++ function), 112`
`xt::xindex_view::has_linear_assign (C++ function), 114`
`xt::xindex_view::operator() (C++ function), 112, 113`
`xt::xindex_view::operator=(C++ function), 112`
`xt::xindex_view::operator[] (C++ function), 113`
`xt::xindex_view::shape (C++ function), 112`
`xt::xindex_view::size (C++ function), 112`
`xt::xindex_view::unchecked (C++ function), 112, 113`
`xt::xindex_view::xindex_view (C++ function), 112`
`xt::xiterable (C++ class), 68`
`xt::xiterable::begin (C++ function), 68`
`xt::xiterable::end (C++ function), 68, 69`
`xt::xiterable::rbegin (C++ function), 69`
`xt::xiterable::rend (C++ function), 69`
`xt::xmasked_view (C++ class), 97`

`xt::xmasked_view::at (C++ function), 98, 99`
`xt::xmasked_view::backstrides (C++ function), 98`
`xt::xmasked_view::dimension (C++ function), 98`
`xt::xmasked_view::element (C++ function), 100`
`xt::xmasked_view::fill (C++ function), 100`
`xt::xmasked_view::layout (C++ function), 100`
`xt::xmasked_view::operator() (C++ function), 98`
`xt::xmasked_view::operator[] (C++ function), 100`
`xt::xmasked_view::shape (C++ function), 98`
`xt::xmasked_view::size (C++ function), 98`
`xt::xmasked_view::strides (C++ function), 98`
`xt::xmasked_view::unchecked (C++ function), 99`
`xt::xmasked_view::value (C++ function), 101`
`xt::xmasked_view::visible (C++ function), 101`
`xt::xmasked_view::xmasked_view (C++ function), 97`
`xt::xoptional_assembly (C++ class), 94`
`xt::xoptional_assembly::from_shape (C++ function), 96`
`xt::xoptional_assembly::operator= (C++ function), 96`
`xt::xoptional_assembly::xoptional_assembly (C++ function), 94–96`
`xt::xoptional_assembly_adaptor (C++ class), 96`
`xt::xoptional_assembly_adaptor::operator= (C++ function), 97`
`xt::xoptional_assembly_adaptor::xoptional_assembly_adaptor (C++ function), 97`
`xt::xoptional_assembly_base (C++ class), 89`
`xt::xoptional_assembly_base::at (C++ function), 90`
`xt::xoptional_assembly_base::backstrides (C++ function), 90`
`xt::xoptional_assembly_base::broadcast_shape (C++ function), 92`
`xt::xoptional_assembly_base::dimension (C++ function), 89`
`xt::xoptional_assembly_base::element (C++ function), 92`
`xt::xoptional_assembly_base::fill (C++ function), 93`
`xt::xoptional_assembly_base::has_linear_assign (C++ function), 92`
`xt::xoptional_assembly_base::has_value (C++ function), 94`
`xt::xoptional_assembly_base::layout (C++ function), 93`
`xt::xoptional_assembly_base::operator() (C++ function), 90`
`xt::xoptional_assembly_base::operator[] (C++ function), 91, 92`
`xt::xoptional_assembly_base::reshape (C++ function), 93`
`xt::xoptional_assembly_base::resize (C++ function), 93`
`xt::xoptional_assembly_base::shape (C++ function), 90`
`xt::xoptional_assembly_base::size (C++ function), 89`
`xt::xoptional_assembly_base::strides (C++ function), 90`
`xt::xoptional_assembly_base::unchecked (C++ function), 91`
`xt::xoptional_assembly_base::value (C++ function), 93`
`xt::xreducer (C++ class), 120`
`xt::xreducer::at (C++ function), 121`
`xt::xreducer::broadcast_shape (C++ function), 122`
`xt::xreducer::dimension (C++ function), 121`
`xt::xreducer::element (C++ function), 122`
`xt::xreducer::expression (C++ function), 121`
`xt::xreducer::has_linear_assign (C++ function), 123`
`xt::xreducer::layout (C++ function), 121`
`xt::xreducer::operator() (C++ function), 121`
`xt::xreducer::operator[] (C++ function), 122`
`xt::xreducer::shape (C++ function), 121`
`xt::xreducer::size (C++ function), 121`
`xt::xreducer::unchecked (C++ function), 122`
`xt::xreducer::xoptional_assembly_adaptor (C++ function), 121`
`xt::xsemantic_base (C++ class), 53`
`xt::xsemantic_base::assign (C++ function), 56`
`xt::xsemantic_base::bit_and_assign (C++ function), 57`
`xt::xsemantic_base::bit_or_assign (C++ function), 57`
`xt::xsemantic_base::bit_xor_assign (C++ function), 57`
`xt::xsemantic_base::divides_assign (C++ function), 56`
`xt::xsemantic_base::minus_assign (C++ function), 56`
`xt::xsemantic_base::modulus_assign (C++ function), 57`
`xt::xsemantic_base::multiplies_assign (C++ function), 56`
`xt::xsemantic_base::operator*= (C++ function), 53, 55`

`xt::xsemantic_base::operator+=` (*C++ function*), 53, 54
`xt::xsemantic_base::operator-=` (*C++ function*), 53, 54
`xt::xsemantic_base::operator/=` (*C++ function*), 53, 55
`xt::xsemantic_base::operator%=` (*C++ function*), 54, 55
`xt::xsemantic_base::operator&=` (*C++ function*), 54, 55
`xt::xsemantic_base::operator^=` (*C++ function*), 54, 55
`xt::xsemantic_base::operator|=` (*C++ function*), 54, 55
`xt::xsemantic_base::plus_assign` (*C++ function*), 56
`xt::xshared_expression` (*C++ class*), 52
`xt::xshared_expression::use_count` (*C++ function*), 52
`xt::xshared_expression::xshared_expression` (*C++ function*), 52
`xt::xstrided_container` (*C++ class*), 63
`xt::xstrided_container::layout` (*C++ function*), 64
`xt::xstrided_container::reshape` (*C++ function*), 64
`xt::xstrided_container::resize` (*C++ function*), 63, 64
`xt::xstrided_slice_vector` (*C++ type*), 107
`xt::xstrided_view` (*C++ class*), 106
`xt::xstrided_view::fill` (*C++ function*), 107
`xt::xstrided_view::operator=` (*C++ function*), 107
`xt::xstrided_view::xstrided_view` (*C++ function*), 107
`xt::xtensor` (*C++ type*), 82
`xt::xtensor_adaptor` (*C++ class*), 84
`xt::xtensor_adaptor::operator=` (*C++ function*), 85
`xt::xtensor_adaptor::xtensor_adaptor` (*C++ function*), 84
`xt::xtensor_container` (*C++ class*), 81
`xt::xtensor_container::operator=` (*C++ function*), 82
`xt::xtensor_container::xtensor_container` (*C++ function*), 81, 82
`xt::xtensor_fixed` (*C++ type*), 89
`xt::xtensor_optional` (*C++ type*), 82
`xt::xview` (*C++ class*), 101
`xt::xview::at` (*C++ function*), 102, 103
`xt::xview::broadcast_shape` (*C++ function*), 104
`xt::xview::data` (*C++ function*), 104
`xt::xview::data_offset` (*C++ function*), 104
`xt::xview::dimension` (*C++ function*), 102
`xt::xview::expression` (*C++ function*), 102
`xt::xview::fill` (*C++ function*), 102
`xt::xview::has_linear_assign` (*C++ function*), 105
`xt::xview::layout` (*C++ function*), 102
`xt::xview::operator()` (*C++ function*), 102, 103
`xt::xview::operator=` (*C++ function*), 101
`xt::xview::operator[]` (*C++ function*), 103, 104
`xt::xview::shape` (*C++ function*), 102
`xt::xview::size` (*C++ function*), 102
`xt::xview::slices` (*C++ function*), 102
`xt::xview::storage` (*C++ function*), 104
`xt::xview::strides` (*C++ function*), 104
`xt::xview::unchecked` (*C++ function*), 102, 103
`xt::xview::xview` (*C++ function*), 101
`xt::xview_semantic` (*C++ class*), 58
`xt::xview_semantic::assign_temporary` (*C++ function*), 58
`xt::zeros` (*C++ function*), 126
`xt::zeros_like` (*C++ function*), 127