xframe

Johan Mabille, Sylvain Corlay, Wolf Vollprecht and Martin Renou

Apr 13, 2021

INSTALLATION

Multi-dimensional labeled arrays and data frame based on xtensor.

xframe is a C++ library meant for numerical analysis with multi-dimensional labeled array expressions (also referred as variable expressions) and data frame expressions. It is built upon xtensor and provides similar features:

- an extensible expression system enabling **lazy broadcasting** based on dimension names.
- an API following the idioms of the C++ standard library.
- tools to manipulate variable expressions and build upon *xframe*.

The API of *xframe* is inspired by the ones of pandas and xarray.

xframe requires a modern C++ compiler supporting C++14. The following C++ compilers are supported:

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

CHAPTER

ONE

LICENSING

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

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

1.1 Installation

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

Besides the xframe 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 xframe headers.

1.1.1 Using the conda-forge package

A package for xframe is available for the mamba (or conda) package manager.

```
mamba install -c conda-forge xframe
```

1.1.2 From source with cmake

You can also install `xframe` from source with cmake. This requires that you have the xtensor 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. `xframe` installation from cmake assumes this folder contains `include` and `lib` subfolders.

1.1.3 Including xframe in your project

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

```
find_package(xframe REQUIRED)
target_include_directories(your_target PUBLIC ${xframe_INCLUDE_DIRS})
target_link_libraries(your_target PUBLIC xframe)
```

1.2 Getting started

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

1.2.1 First example

```
#include <iostream>
#include "xtensor/xrandom.hpp"
#include "xframe/xio.hpp"
#include "xframe/xvariable.hpp"

int main(int argc, char* argv[])
{
    using coordinate_type = xf::xcoordinate<xf::fstring>;
    using variable_type = xf::xvariable<double, coordinate_type>;
    using data_type = variable_type::data_type;

    // Creation of the data
    data_type data = xt::eval(xt::random::rand({6, 3}, 15., 25.));
    data(0, 0).has_value() = false;
    data(2, 1).has_value() = false;

    // Creation of coordinates and dimensions
    auto time_axis = xf::axis({"2018-01-01", "2018-01-02", "2018-01-03", "2018-01-04",
↪ "2018-01-05", "2018-01-06"});
    auto city_axis = xf::axis({"London", "Paris", "Brussels"});
    auto coord = xf::coordinate({{"date", time_axis}, {"city", city_axis}});
    auto dim = xf::dimension({"date", "city"});

    // Creation of the variable
    auto var = variable_type(data, coord, dim);
    std::cout << var << std::endl;

    return 0;
}
```

This example creates a variable, that is, a tensor data (here random) with labels and dimension names.

1.2.2 Compiling the first example

xframe 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 *xframe* and those of its dependencies, that is, *xtensor* and *xtl*; 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/headers/ example.cpp -o example
```

When you run the program, it produces the following output (data should be different since it is randomly generated):

```
{{    N/A, 23.3501, 24.6887},
 {17.2103, 18.0817, 20.4722},
 {16.8838,     N/A, 24.9646},
 {24.6769, 22.2584, 24.8111},
 {16.0986, 22.9811, 17.9703},
 {15.0478, 16.1246, 21.3976}}
Coordinates:
date: (2018-01-01, 2018-01-02, 2018-01-03, 2018-01-04, 2018-01-05, 2018-01-06, )
city: (London, Paris, Brussels)
```

1.2.3 Building with cmake

A better alternative for building programs using *xframe* 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(xframeREQUIRED)

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
```

1.2.4 Second example: simplified variable creation

xframe provides many shortcuts so coordinates and variables can be created with a concise syntax. The following example creates the same variable as the previous one:

```
#include <iostream>
#include "xtensor/xrandom.hpp"
#include "xframe/xio.hpp"
#include "xframe/xvariable.hpp"

int main(int argc, char* argv[])
{
    using coordinate_type = xf::xcoordinate<xf::fstring>;
    using variable_type = xf::xvariable<double, coordinate_type>;
    using data_type = variable_type::data_type;

    // Creation of the data
    data_type data = xt::eval(xt::random::rand({6, 3}, 15., 25.));
    data(0, 0).has_value() = false;
    data(2, 1).has_value() = false;

    // Creation of the variable
    auto var = variable_type(
        data,
        {
            {"date", xf::axis({"2018-01-01", "2018-01-02", "2018-01-03", "2018-01-04",
↪ "2018-01-05", "2018-01-06"})},
            {"city", xf::axis({"London", "Paris", "Brussels"})}
        }
    );
    std::cout << var << std::endl;

    return 0;
}
```

When compiled and run, this produces output similar to the one of the previous example (same coordinate system but different data due to random generation).

1.2.5 Third example: data access

xframe provides different ways to access data in a variable.

```
#include <iostream>
#include "xtensor/xrandom.hpp"
#include "xframe/xio.hpp"
#include "xframe/xvariable.hpp"

int main(int argc, char* argv[])
{

    using coordinate_type = xf::xcoordinate<xf::fstring>;
    using variable_type = xf::xvariable<double, coordinate_type>;
    using data_type = variable_type::data_type;

    // Creation of the data
    data_type data = xt::eval(xt::random::rand({6, 3}, 15., 25.));
```

(continues on next page)

(continued from previous page)

```
    data(0, 0).has_value() = false;
    data(2, 1).has_value() = false;

    // Creation of the variable
    auto var = variable_type(
        data,
        {
            {"date", xf::axis({"2018-01-01", "2018-01-02", "2018-01-03", "2018-01-04",
↪ "2018-01-05", "2018-01-06"})},
            {"city", xf::axis({"London", "Paris", "Brussels"})}
        }
    );

    // Data access
    std::cout << "operator() - " << var(3, 0) << std::endl;
    std::cout << "locate     - " << var.locate("2018-01-04", "London") << std::endl;
    std::cout << "iselect    - " << var.iselect({{"date", 3}, {"city", 0}}) <<␣
↪std::endl;
    std::cout << "select     - " << var.select({{"date", "2018-01-04"}, {"city",
↪"London"}}) << std::endl;

    return 0;
}
```

Outputs:

```
operator() - 24.6769
locate     - 24.6769
iselect    - 24.6769
select     - 24.6769
```

1.3 Data structures

1.3.1 Axes

An axis is a mapping of labels to positions in a given dimension. It is the equivalent of the `index` object from pandas. *xframe* supports many types of labels, the most common are strings, char, integers and dates. An axis is created from a list of labels, a builder function is provided so the type of the axis can be inferred. The following example illustrates the two main ways of creating an axis:

```
using saxis_type = xf::xaxis<xf::fstring, std::size_t>;

saxis_type s1({ "a", "b", "d", "e" });
auto s2 = xf::axis({ "a", "b", "d", "e" });
// s1 and s2 are similar axes
```

It is also possible to create an axis given the size of the axis or the start, stop and step:

```
auto s3 = xf::axis(5);         // == xf::axis({ 0, 1, 2, 3, 4 });
auto s4 = xf::axis(2, 7);      // == xf::axis({ 2, 3, 4, 5, 6 });
auto s5 = xf::axis(0, 10, 2); // == xf::axis({ 0, 2, 4, 6, 8 });
auto s6 = xf::axis("a", "d"); // == xf::axis({ "a", "b", "c" });
```

The axis API is similar to the one of a constant `std::map` that throws an exception when asked a missing key:

```
std::size_t i0 = s1["a"];
try
{
    std::size_t i1 = s1["c"];
}
catch(std::exception& e)
{
    // The exception will be catch since "c" is not a label of s1
    std::cout << e.what() << std::endl;
}
```

xaxis also provides iterators and methods to compute the union and the intersection of axes. However a user rarely needs to manipulate the axes directly, the most common operation is to create them and then store them in a coordinate system.

1.3.2 Coordinates

Coordinates are mappings of dimension names to axes. *xframe* provides different methods to easily create them:

```
using coordinate_type = xf::xcoordinate<xf::fstring>;

coordinate_type c1({{"group", xf::axis({"a", "b", "d", "e"})},
                    {"city",  xf::axis({"London", "Paris", "Brussels"})}});
auto c2 = xf::coordinate({{"group", xf::axis({"a", "b", "d", "e"})},
                          {"city",  xf::axis({"London", "Paris", "Brussels"})}});
// c1 and c2 are similar coordinates
```

Note: The builder function xf::coordinate converts the const char* arguments to fstring and returns a xcoordinate<fstring> object. You can modify this behavior by specifying the key type of the coordinate as the first template parameter of the coordinate function: auto c2 = xf::coordinate<std::string>({{"group", xf::axis({"a", "b", "d", "e"})}, ...});

xnamed_axis allows to store a dimension name - axis pair that you can reuse in different coordinates objects; if you want to create a coordinate object from a named axis, all the arguments must be named axes; fortunately, a xnamed_axis can be created in place, as shown below:

```
// This object will be used in different coordinates objects
auto a1 = xf::named_axis("igroup", xf::axis({1, 2, 4, 5}));

auto c1 = xf::coordinate<xf::fstring>(a1, xf::named_axis("city", xf::axis({"London",
↪"Parid", "Brussels"})));
auto c2 = xf::coordinate<xf::fstring>(a1, xf::named_axis("country", xf::axis({"USA",
↪"Japan"})));
```

As you can notice, coordinates objects can store axes with different label types. By default, these types are int, std::size_t, char and xf::fstring, you can specify a different type list:

```
using coordinate_type = xf::xcoordinate<xf::fstring, xtl::mpl::vector<int,␣
↪std::string>>;

coordinate_type c({{"group", xf::axis({"a", "b", "d", "e"})},
                   {"city",  xf::axis({"London", "Paris", "Brussels"})}});
```

1.3.3 Dimension

A dimension object is the mapping of the dimension names to the dimension positions in the data tensor. Creating a `xdimension` is as simple as creating an `xcoordinate` or an `xaxis`:

```
using dimension_type = xf::xdimension<xf::fstring>;

dimension_type dim1({"city", "group"});
auto dim2 = xf::dimension({"city", "group"});
// dim1 and dim2 are similar dimensions
```

`xdimension` provides an API similar to `xaxis` and therefore can be considered as a special axis. Together a dimension object and a coordinate object form a coordinate system which maps labels and dimension names to indexes in the data tensor.

Note: Like `xf::coordinate`, the builder function `xf::dimension` converts the `const char*` arguments to `fstring` and returns a `xdimension<fstring>` object. You can modify this behavior by specifying the label type of the dimension as the first template parameter of the `dimension` function: `auto d = xf::dimension<std::string>({"city", "group"});`

1.3.4 Variables

A variable is a data tensor with a coordinate system, that is an `xcoordinate` object and an `xdimension` object. It is the C++ equivalent of the `xarray.DataArray` Python class. `xvariable` provides many constructors:

```
using coordinate_type = xf::xcoordinate<xf::fstring>;
using dimension_type = xf::xdimension<xf::fstring>;
using variable_type = xvariable<double, coordinate_type>;

data_type d = xt::eval(xt::random::rand({3, 4}));
auto c = xf::coordinate({{"group", xf::axis({"a", "b", "d", "e"})},
                         {"city",  xf::axis({"London", "Paris", "Brussels"})}});
auto dim = xf::dimension({"city", "group"});

variable_type v1(d, c, dim);

// Coordinates and dimension can be built in place
variable_type v2(d, xf::coordinate({{"group", xf::axis({"a", "b", "d", "e"})},
                                    {"city",  xf::axis({"London", "Paris", "Brussels"}
↪)}}),
                    xf::dimension({"city", "group"}));
```

The data parameter can be omitted, in that case the variable creates an uninitialized data tensor:

```
variable_type v3(c, dim);

variable_type v4(xf::coordinate({{"group", xf::axis({"a", "b", "d", "e"})},
                                 {"city",  xf::axis({"London", "Paris", "Brussels"})}}
↪),
                 xf::dimension({"city", "group"}));
```

A variable can also be created from a map of axes and a list of dimension names:

```
variable_type::coordinate_map coord_map;
coord_map["group"] = xf::axis({"a", "b", "d", "e"});
coord_map["city"]  = xf::axis({"London", "Paris", "Brussels"});
dimension_type::label_list dim_list = {"group", "city"};

variable_type v5(d, coord_map, dim_list);
variable_type v6(coord_map, dim_list);
```

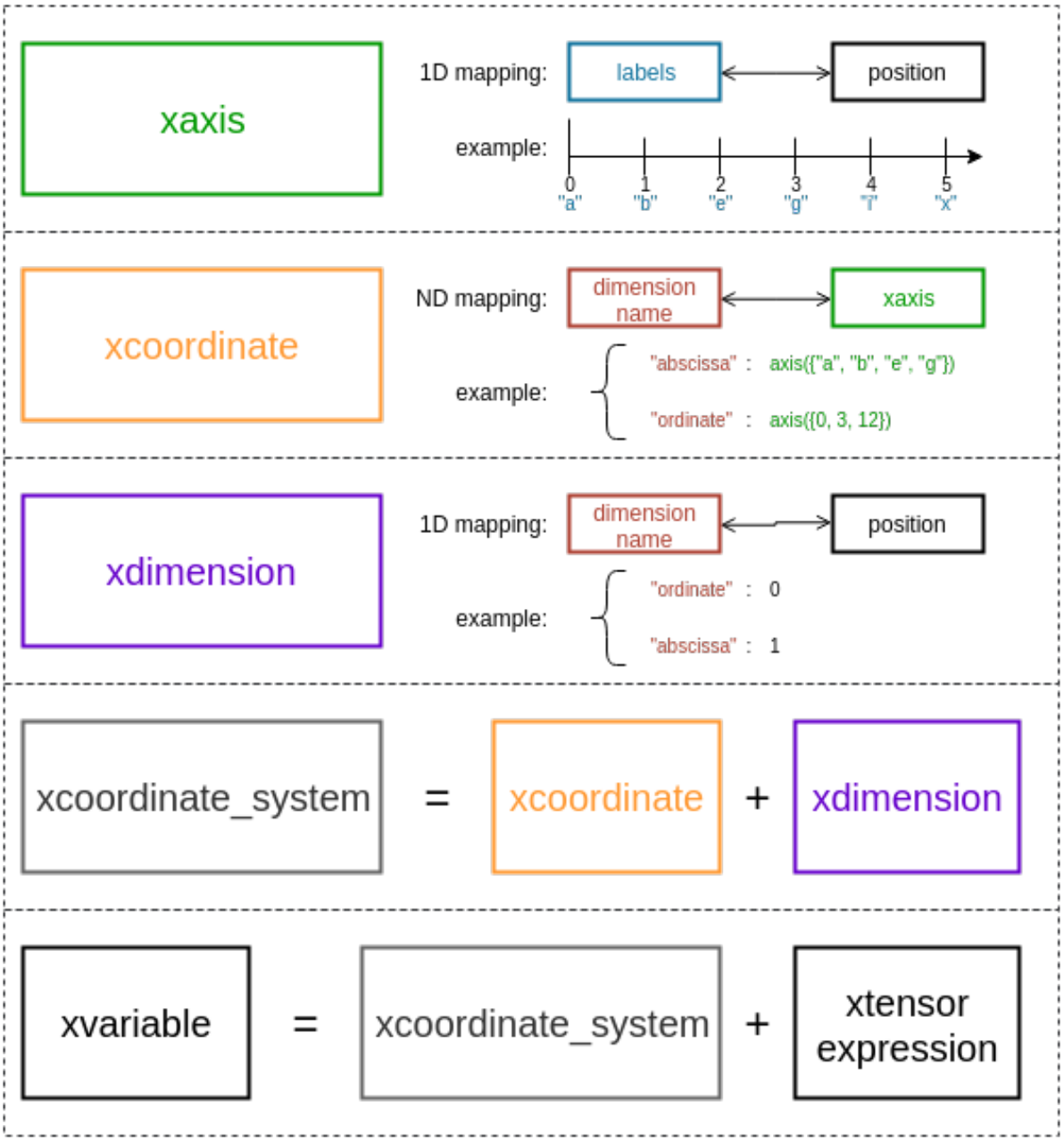
If the dimension object is omitted, the dimension mapping is inferred from the coordinate object. In the code below, the mapping is different from the previous defined variables, `group` is the name of the first dimension and `city` is the name of the second one:

```
variable_type v7(d, {{"group", xf::axis({"a", "b", "d", "e"})},
                     {"city",  xf::axis({"london", "Paris", "Brussels"})}});

// variable with same coordinate system but uninitialized data
variable_type v8({{"group", xf::axis({"a", "b", "d", "e"})},
                  {"city",  xf::axis({"london", "Paris", "Brussels"})}});
```

xframe also provides builder functions, so that the type of the variable can be inferred:

```
auto v10 = variable(d, c, dim);
auto v11 = variable(d, xf::coordinate({{"group", xf::axis({"a", "b", "d", "e"})},
                                       {"city",  xf::axis({"London", "Paris",
↪"Brussels"})}}),
                       xf::dimension({"city", "group"}));

auto v12 = variable(c, dim);
auto v13 = variable(xf::coordinate({{"group", xf::axis({"a", "b", "d", "e"})},
                                    {"city",  xf::axis({"London", "Paris", "Brussels"}
↪)}}),
                    xf::dimension({"city", "group"}));
```

1.3.5 Summary


xaxis
1D mapping:
labels
position
example:
0 "a"
1 "b"
2 "e"
3 "g"
4 "l"
5 "x"
xcoordinate
ND mapping:
dimension name
xaxis
example:
"abscissa" : axis({"a", "b", "e", "g"})
"ordinate" : axis({0, 3, 12})
xdimension
1D mapping:
dimension name
position
example:
"ordinate" : 0
"abscissa" : 1
xcoordinate_system
=
xcoordinate
+
xdimension
xvariable
=
xcoordinate_system
+
xtensor expression

1.4 Indexing and selecting data

In this section, we consider the following variable:

```
using coordinate_type = xf::xcoordinate<xf::fstring>;
using dimension_type = xf::xdimension<xf::fstring>;
using variable_type = xf::xvariable<double, coordinate_type>;

data_type d = xt::eval(xt::random::rand({6, 3}, 15., 25.));
variable_type v(std::move(d),
                {
                    {"group", xf::axis({"a", "b", "d", "e", "g", "h"})},
                    {"city",  xf::axis({"London", "Paris", "Brussels"})}
                });
```

Printing this variable in a Jupyter Notebook gives:

	London	Paris	Brussels
a	16.3548	23.3501	24.6887
b	17.2103	18.0817	20.4722
d	16.8838	24.9288	24.9646
e	24.6769	22.2584	24.8111
g	16.0986	22.9811	17.9703
h	15.0478	16.1246	21.3976

xframe provides flexible indexing methods for data selection, similar to the ones of xarray. These methods are summarized in the following table:

Dimension lookup	Index lookup	`xvariable` syntax
Positional	By integer	`v(2, 1)`
Positional	By label	`v.locate("d", "Paris")`
By name	By integer	`v.iselect({{"group", 2}, {"city", 1}})`
By name	By label	`v.select({{"group", "d"}, {"city", "Paris"}})`

1.4.1 Positional indexing

The most basic way to access elements of an `xvariable` is to use `operator()`, like you would do with an `xtensor`:

```
std::cout << v(2, 1) << std::endl;
```

Contrary to Python, it is not possible to have different return types for a same method in C++. Multi selection is done with free functions that return views on the variable:,

```
#include "xvariable_view.hpp"

auto view1 = xf::ilocate(v, xf::irange(0, 5, 2), xf::irange(1, 3));
std::cout << view1 << std::endl;
```

	Paris	Brussels
a	23.3501	24.6887
d	24.9288	24.9646
g	22.9811	17.9703

Therefore a change in the view will reflect in the underlying variable:

```
view1(0, 1) = 0.;
std::cout << v(2, 2) << std::endl;
// Outputs 0.
```

In the code creating the view, `irange` returns a range slice from *xtensor*, so any multi selection in *xtensor* is also supported in *xframe*.

`xvariable` also supports label-based indexing, with the `locate` method for single point selection, and `locate` free function for multi selection:

```
std::cout << v.locate("d", "Paris") << std::endl;
auto view2 = xf::locate(v, xf::range("a", "h", 2), xf::range("Paris", "Brussels"));
std::cout << view2 << std::endl;
// Same output as previous code
```

Be aware of the difference between `range` and `irange` parameters: for the former one, accepting labels, the last value is *included* while for the latter one, accepting integral indices, the las value is *excluded*.

xframe provides label-based slices similar to those of *xtensor*, so label-based multi selection is really similar to positional multi selection.

1.4.2 Indxing with dimension names

With the dimension names, we do not have to rely on the dimension order. We can use them explicitely to select data; Like positional indexing, *xframe* provides methods and free functions depending on the kind of selection you want to do:

```
// Dimension by name, index by position
std::cout << v.iselect({{"city", 1}, {"group", 2}}) << std::endl;
auto view3 = xf::iselect(v, {{"city", xf::irange(1, 3)}, {"group", xf::irange(0, 5,␣
↪2)}});

// Dimension by name, index by label
std::cout << v.select({{"city", "Paris"}, {"group", "d"}}) << std::endl;
auto view4 = xf::select(v, {{"city", xf::range("Paris", "Brussels")}, {"group",␣
↪xf::range("a", "h", 2)}});
// view3 and view4 gives the same output as view2 and view1
```

Contrary to xarray, *xframe* does not provide a selection operator accepting a map argument.

1.4.3 Keeping and dropping labels

`drop` and `keep` functions return slices that can be used to create a view with the listed labels along the specified dimensions dropped or kept:

```
auto view5 = xf::select(v, {{"city", xf::drop("London")}, {"group", xf::keep("a", "d",
↪ "g")}});
// view5 is equivalent to view4
```

This is different form xarray where the `xarray.DataArray.drop` method returns a new object. To achieve the same with *xframe*, simply assign the view to a new `xvariable` object:

```
variable_type v2 = view5;
```

1.4.4 Masking views

Masking views allow to select data points based on conditions expresses on labels. These conditions can be arbitrary complicated boolean expressions. Contrary to other views which are generally a subset of the original data, a masking view has the same shape as its underlying `xvariable`.

Masking views are created with the `where` function:

```
data_type d2 = {{ 1.,  2., 3. },
                { 4.,  5., 6. },
                { 7.,  8., 9. }};

auto v3 = variable_type(
    d2,
    {
        {"x", xf::axis(3)},
        {"y", xf::axis(3)},
    }
);

auto view6 = xf::where(
    v3,
    not_equal(v3.axis<int>("x"), 2) && v3.axis<int>("y") < 2
);
std::cout << view6 << std::endl;
```

In a Jupyter Notebookn, this outputs the following:

	0	1	2
0	1	2	masked
1	4	5	masked
2	masked	masked	masked

When assigning to a masked view, masked values are not changed. Like other views, a masking view is a proxy on its underlying variable, no copy is made, so changing an unmasked value actually changes the corresponding value in the undnerlying variable.

1.4.5 Assigning values with indexing

Data selection in variables return either references or views; therefore, contrary to xarray, it is *possible* to assign values to a subset of a variable with any of the indexing method:

```
// The next four lines are equivalent, they change a single value of v:
v(2, 1) = 2.5;
v.locate("d", "Paris") = 2.5;
v.iselect({{"city", 1}, {"group", 2}}) = 2.5;
v.select({{"city", "Paris"}, {"group", "d"}}) = 2.5;

data_type d3 = {{0.,  1.},
                {2.,  3.},
                {4.,  5.}};

auto v4 = variable_type(
    d3,
    {
```

(continues on next page)

(continued from previous page)

```
        {"group", xf::axis({"a", "d", "g"})},
        {"city", xf::axis({"Paris", "Brussels"})}
    }
);

// The next four lines are equivalent, they change a subset of v
xf::ilocate(v, xf::irange(0, 5, 2), xf::irange(1, 3)) = v4;
xf::locate(v, xf::range("a", "h", 2), xf::range("Paris", "Brussels")) = v4;
xf::iselect(v, {{"city", xf::irange(1, 3)}, {"group", xf::irange(0, 5, 2)}}) = v4;
xf::select(v, {{"city", xr::range("Paris", "Brussels")}, {"group", xf::range("a", "h",
↪ 2)}}) = v4;
```

Printing `v` after the assign gives

	London	Paris	Brussels
a	16.3548	0	1
b	17.2103	18.0817	20.4722
d	16.8838	2	3
e	24.6769	22.2584	24.8111
g	16.0986	4	5
h	15.0478	16.1246	21.3976

1.4.6 Reindexing views

Reindexing views give variables new set of coordinates to corresponding dimensions. Like other views, no copy is involved. Asking for values corresponding to new labels not found in the original set of coordinates returns missing values. In the next example, we reindex the `city` dimension:

```
auto view7 = xf::reindex(v, {{"city", xf::axis({"London", "New York", "Brussels"})}});
```

	London	New York	Brussels
a	16.3548	N/A	24.6887
b	17.2103	N/A	20.4722
d	16.8838	N/A	24.9646
e	24.6769	N/A	24.8111
g	16.0986	N/A	17.9703
h	15.0478	N/A	21.3976

Like xarray, *xframe* provides the useful `reindex_like` shortcut which allows to reindex a variable given the set of coordinates of another variable:

```
auto v5 = variable_type(
    d,
    {
        {"group", xf::axis({"a", "b", "d", "e", "g", "h"})},
        {"city", xf::axis({"London", "New York", "Brussels"})}
    }
);

auto view8 = xf::reindex_like(v, v5);
// view8 is equivalent to view7
```

A reindexing view is a read-only view, it is not possible to change its value with indexing. This allows memory optimizations, the view does not have to store the missing values, it can return a proxy to a static-allocated missing value.

The `align` function allows to reindex many variables with more flexible options:

```
auto t1 = xf::align<join::inner>(v, v5);
std::cout << std::get<0>(t1) << std::endl;
std::cout << std::get<1>(t1) << std::endl;
```

The last lines print the same output:

	London	Brussels
a	16.3548	24.6887
b	17.2103	20.4722
d	16.8838	24.9646
e	24.6769	24.8111
g	16.0986	17.9703
h	15.0478	21.3976

In the following, the variables are aligned w.r.t the union of the coordinates instead of their intersection:

```
auto t2 = xf::align<join::outer>(v, v5);
std::cout << std::get<0>(t2) << std::endl;
std::cout << std::get<1>(t2) << std::endl;
```

The first outuput is

	London	Paris	Brussels	New York
a	16.3548	23.3501	24.6887	N/A
b	17.2103	18.0817	20.4722	N/A
d	16.8838	24.9288	24.9646	N/A
e	24.6769	22.2584	24.8111	N/A
g	16.0986	22.9811	17.9703	N/A
h	15.0478	16.1246	21.3976	N/A

While the second have `N/A` in the `Paris` column.

1.5 Computation

xframe is actually more than a multi-dimensional data frame library; like xtensor, it is an expression engine that allows numerical computation on any object implementing the variable or the frame interfaces.

1.5.1 Expressions

Assume `x`, `y` and `z` are variables with *compatible coordinates* (we'll come back to that later), the return type of an expression such as `x + exp(y) * sin(z)` is **not a variable**. The result is a variable expression which offers the same interface as `xvariable` but does not hold any value. Such expressions can be plugged into others to build more cmoplex expressions:

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

The expression engines avoids the evaluation of intermediate results and their storage in temporary variables, so you can achieve the same performance as if you had written a loop. Such a loop is quite more complicated than an array loop since labels and dimension names are involved in the assignment mechanism.

Since a variable expression provides the same API as `xvariable`, all the indexing and selection operations are available:

```
auto v = (x + exp(y) * sin(z)).select({{"city", "Paris"}, {"group", "a"}});
auto view = select(x + exp(y) * sin(z), {{"city", xf::keep("Paris")}, {"group",␣
↪xf::drop({"b", "d", "h"})}});
```

1.5.2 Lazy evaluation

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

```
// Assuming x and y are variables each holding ~1M values
auto f = cos(x) + sin(y);

double first_res = f.locate("a", "London");
double second_res = f.locate("b", "Pekin");
// 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 cmoputation and the size of the data, it might be convenient to store the result of the expression in a temporary variable:

```
variable_type tmp = cos(x) + sin(y);
variable_type res1 = tmp + 2 * exp(z);
variable_type res2 = tmp - 3 * exp(w);
```

1.5.3 Broadcasting by dimension names

Like xarray, broadcasting in *xframe* is done according to the dimension names rather than their positions. This way, you do not need to transpose variable or insert dimensions of length 1 to to get array operations to work, as commonly done in xtensor with `xt::reshape` or `xt::newaxis`.

This can be illustrated with the following examples. First, consider two one-dimensional variable aligned along different dimensions:

```
auto v1 = variable_type(data_type({1., 2.}), {{"x", xf::axis(1, 3)}});
auto v2 = variable_type(data_type({3., 7.}), {{"y", xf::axis(2, 5)}});
```

We can apply mathematical operations to these variables, their dimension are expanded automatically:

```
variable_type res = v1 + v2;
std::cout << res << std::endl;
// Output:
// {{ 4.  8.}
//  { 5.  9.}}
// Coordinates:
// x: (1, 3, )
// y: (2, 5, )
```

Contrary to xarray , dimensions are not reordered to the order in which they first appeared:

```
auto v3 = variable_type(data_type({{1., 2.}, {3., 4.}}),
                        {{"y", xf::axis({2, 5})}, {"x", xf::axis({1, 4})}});
variable_type res2 = v1 + v3;
std::cout << res2 << std::endl;
// Output:
// {{ 2.  4. }
//  { 4.  6. }}
// Coordinates:
// y: (2, 5, )
// x: (1, 3, )
```

This allows many optimizations in the assignment mechanism.

1.5.4 Automatic alignment

xframe enforces alignment between coordinate labels on objects involved in operations. The default result of an operation is by the *intersection* of the coordinate labels:

```
auto v4 = variable_type(data_type({1., 2., 3.}), {{"x", xf::axis({1, 3, 5})}});
auto v5 = variable_type(data_type({4., 7., 12.}), {{"x", xf::axis({1, 5, 7})}});

variable_type res3 = v4 + v5;
std::cout << res3 << std::endl;
// Output:
// { 5.  10. }
// Coordinates:
// x: (1, 5,)
```

Operations are slower when variables are not aligned, so it might be useful to explicitly align variables involved in loops of performance critical code.

1.5.5 Operators and functions

xframe provides all the basic operators and mathematical functions:

- arithmetic operators: `+`, `-`, `*`, `/`
- logical operators: `&&`, `||`, `!`
- comparison operators: `==`, `!=`, `<`, `<=`, `>`, `>=`
- basic functions: `abs`, `remainder`, `fma`, ...
- exponential functions: `exp`, `expm1`, `log`, `log1p`, ...
- power functions: `pow`, `sqrt`, `cbrt`, ...
- trigonometric functions: `sin`, `cos`, `tan`, ...
- hyperbolic functions: `sinh`, `cosh`, `tanh`, ...
- error and gamma functions: `erf`, `erfc`, `tgamma`, `lgamma`, ...

Actually, any function operating on `xtensor` expressions can work with `xvariable` expressions without any additional work; the only constraint is to accept and return expressions:

```
template <class E1, class E2>
inline auto distance(const xexpression<E1>& e1, const xexpression<E2>& e2)
{
    const E1& de1 = e1.derived_cast();
    const E2& de2 = e2.derived_cast();
    //
    return sqrt(de1 * de1 + de2 * de2);
}
```

This function can work with both tnesor and variable expressions, performing broadcasting according to the rules of *xtensor* or *xframe* depending on its argument type:

```
xt::xarray<double> a1 = {1., 2. };
xt::xarray<double> a2 = {{1., 3.}, {4., 7.}};

// Broadcasting is applied according to xtensor rules,
// that is by dimension order
xt::xarray<double> ares = distance(a1, a2);

// Broadcasting is applied according to xframe rules,
// that is by dimension name
variable_type vres = distance(v1, v3);
```

1.5.6 Missing values

Contrary to pandas or xarray, *xframe* does not use particular values for representing missing values. Instead, it makes use of the dedicated type `xtl::xoptional` which gathers the value and a flag to specify whether the value is missing or not:

```
data_type d = {1., 2., 3. };
d(1).has_value() = false;

auto v = variable_type(d, {{"x", xf::axis({1, 3, 4})}});
std::cout << v << std::endl;
```

(continues on next page)

(continued from previous page)

```
// Output:
// { 1.  N/A  3. }
// Coordinates:
// x: (1, 3, 4,)
```

1.6 Axes

1.6.1 xaxis_base

Defined in `xframe/xaxis_base.hpp`

template<class **D**>
class xaxis_base

Base class for axes.

The *xaxis_base* class defines the common interface for axes, which define the mapping of labels to positions in a given dimension. The axis_base class embeds the list of labels only, the mapping is hold by the inheriting classes.

Template Parameters

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

Labels

auto **labels**() **const**

Returns the list of labels contained in the axis.

auto **label**(size_type *i*) **const**

Return the i-th label of the axis.

Parameters

- `i`: the position of the label.

bool **empty**() **const**

Checks if the axis has no labels.

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

Returns the number of labels in the axis.

Iterators

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

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

This element is a pair label - position.

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

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

auto **rbegin**() **const**
Returns a constant iterator to the first element of the reverse axis.

This element is a pair labal - position.

auto **rend**() **const**
Returns a constant iterator to the element following the last element of the reversed axis.

auto **crbegin**() **const**
Returns a constant iterator to the first element of the reverse axis.

This element is a pair labal - position.

auto **crend**() **const**
Returns a constant iterator to the element following the last element of the reversed axis.

template<class **D1**, class **D2**>
bool xf::**operator==**(**const** xaxis_base<*D1*> &*lhs*, **const** xaxis_base<*D2*> &*rhs*)
Returns true is lhs and rhs are equivalent axes, i.e.

they contain the same label - position pairs.

Parameters

- lhs: an axis.
- rhs: an axis.

template<class **D1**, class **D2**>
bool xf::**operator!=**(**const** xaxis_base<*D1*> &*lhs*, **const** xaxis_base<*D2*> &*rhs*)
Returns true is lhs and rhs are not equivalent axes, i.e.

they contain different label - position pairs.

Parameters

- lhs: an axis.
- rhs: an axis.

1.6.2 xaxis

Defined in xframe/xaxis.hpp

template<class **L**, class **T** = std::size_t, class **MT** = hash_map_tag>
class xaxis : **public** xf::xaxis_base<*xaxis*<*L*, *T*, *MT*>>
Class modeling an axis in a coordinate system.

The xaxis class is used for modeling general axes; an axis is a mapping of labels to positions in a given dimension. It is the equivalent of the Index object from pandas.

Template Parameters

- L: the type of labels.
- T: the integer type used to represent positions. Default value is std::size_t.
- MT: the tag used for choosing the map type which holds the label- position pairs. Possible values are map_tag and hash_map_tag. Default value is hash_map_tag.

Constructors

xaxis()
Constructs an empty axis.

xaxis(**const** label_list &*labels*)
Constructs an axis with the given list of labels.

This list is copied and the constructor internally checks whether it is sorted.

Parameters

- `labels`: the list of labels.

xaxis(label_list &&*labels*)
Constructs an axis with the given list of labels.

The list is moved and therefore it is invalid after the axis has been constructed. The constructor internally checks whether the list is sorted.

Parameters

- `labels`: the list of labels.

xaxis(std::initializer_list<key_type> *init*)
Constructs an axis from the given initializer list of labels.

The constructor internally checks whether the list is sorted.

template<class **L1**>
xaxis(xaxis_default<*L1*, T> *axis*)
Constructs an axis from a `default_axis`.

See default_axis

template<class **InputIt**>
xaxis(*InputIt first*, *InputIt last*)
Constructs an axis from the content of the range [first, last)

Parameters

- `first`: An iterator to the first label.
- `last`: An iterator the the element following the last label.

Data

bool **contains**(**const** key_type &*key*) **const**
Returns true if the axis contains the speficied label.

Parameters

- `key`: the label to search for.

auto **operator[]**(**const** key_type &*key*) **const**
Returns the position of the specified label.

If this last one is not found, an exception is thrown.

Parameters

- `key`: the label to search for.

Filters

template<class **F**>
auto **filter**(**const** *F* &*f*) **const**
Builds an return a new axis by applying the given filter to the axis.

Parameters

- `f`: the filter used to select the labels to keep in the new axis.

template<class **F**>
auto **filter**(**const** *F* &*f*, size_type *size*) **const**
Builds an return a new axis by applying the given filter to the axis.

When the size of the new list of labels is known, this method allows some optimizations compared to the previous one.

Parameters

- `f`: the filter used to select the labels to keep in the new axis.
- `size`: the size of the new label list.

Iterator

auto **find**(**const** key_type &*key*) **const**
Returns a constant iterator to the element with label equivalent to `key`.

If no such element is found, past-the-end iterator is returned.

Parameters

- `key`: the label to search for.

auto **cbegin**() **const**
Returns a constant iterator to the first element of the axis.

This element is a pair label - position.

auto **cend**() **const**
Returns a constant iterator to the element following the last element of the axis.

Set operations

template<class ...**Args**>
bool **merge**(**const** *Args*&... *axes*)
Merges all the axes arguments into this ones.

After this function call, the axis contains all the labels from all the arguments.

Return true is the axis already contained all the labels.

Parameters

- `axes`: the axes to merge.

template<class ...**Args**>
bool **intersect**(**const** *Args*&... *axes*)
Replaces the labels with the intersection of the labels of the axes arguments and the labels of this axis.

Return true if the intersection is equivalent to this axis.

Parameters

- `axes`: the axes to intersect.

Public Functions

bool **is_sorted**() **const**
Returns true if the labels list is sorted.

template<class **T** = std::size_t, class **L**>
xaxis<*L*, *T*> `xf::axis`(*L start*, *L stop*, *L step* = 1)
Returns an axis containing a range of integral labels.

Parameters

- `start`: the first value of the range.
- `stop`: the end of the range. The range doe snot contain this value.
- `step`: Spacing between values. Default step is `1`.

Template Parameters

- `T`: the integral type used for positions. Default value
- `L`: the type of the labels.

template<class **T** = std::size_t, class **L**>
xaxis<*L*, *T*> `xf::axis`(std::initializer_list<*L*> *init*)
Builds an returns an axis from the specified list of labels.

Parameters

- `init`: the list of labels.

Template Parameters

- `T`: the integral type used for positions. Default value is `std::size_t`.
- `L`: the type of the labels.

1.6.3 xaxis_default

Defined in `xframe/xaxis_default.hpp`

template<class **L**, class **T** = std::size_t>
class xaxis_default : **public** xf::xaxis_base<*xaxis_default*<*L*, *T*>>
Default axis with integral labels.

The *xaxis_default* class is used for modeling a default axis that holds a contiguous sequence of integral labels starting at 0.

Template Parameters

- `L`: the type of labels. This must be an integral type.
- `T`: the integer type used to represent positions. Default value is `std::size_t`.

Public Functions

xaxis_default(size_type *size* = 0)
Constructs a default axis holding `size` integral elements.

The labels sequence is [0, 1, ... size - 1)

bool **is_sorted**() **const**
Returns true if the labels list is sorted.

bool **contains**(**const** key_type &*key*) **const**
Returns true if the axis contains the speficied label.

Parameters

- `key`: the label to search for.

auto **operator[]**(**const** key_type &*key*) **const**
Returns the position of the specified label.

If this last one is not found, an exception is thrown.

Parameters

- `key`: the label to search for.

auto **find**(**const** key_type &*key*) **const**
Returns a constant iterator to the element with label equivalent to `key`.

If no such element is found, past-the-end iterator is returned.

Parameters

- `key`: the label to search for.

auto **cbegin**() **const**
Returns a constant iterator to the first element of the axis.

This element is a pair label - position.

auto **cend**() **const**
Returns a constant iterator to the element following the last element of the axis.

template<class **F**>
auto **filter**(**const** *F* &*f*) **const**
Builds an return a new axis by applying the given filter to the axis.

Parameters

- `f`: the filter used to select the labels to keep in the new axis.

template<class **F**>

auto **filter**(**const** *F* &*f*, size_type *size*) **const**

Builds an return a new axis by applying the given filter to the axis.

When the size of the new list of labels is known, this method allows some optimizations compared to the previous one.

Parameters

- `f`: the filter used to select the labels to keep in the new axis.
- `size`: the size of the new label list.

template<class **T** = std::size_t, class **L**>
xaxis_default<*L*, *T*> `xf::axis`(*L size*)

Returns a default axis that holds `size` integral labels.

Parameters

- `size`: the number of labels.

Template Parameters

- `T`: the integral type used for positions. Default value is `std::size_t`.
- `L`: the type of the labels. This must be an integral type.

1.6.4 xaxis_function

Defined in `xframe/xaxis_function.hpp`

template<class **F**, class **R**, class ...**CT**>
class xaxis_function : **public** xt::xexpression<*xaxis_function*<*F*, *R*, *CT*...>>

An expression of xaxis.

The *xaxis_function* class is used for creating an expression on named axis, e.g. `auto expr = not_equal(axis1, 2) && axis2 < 2`.

See *xaxis_expression_leaf*

See *xnamed_axis*

Template Parameters

- `F`: the function type.
- `R`: the result type.
- `CT`: the function argument types.

Public Functions

template<class **Func**, class **U** = std::enable_if<!std::is_base_of<*Func*, self_type>::value>>
xaxis_function(*Func* &&*f*, CT... *e*)

Builds an axis function.

Parameters

- `f`: the function to apply.
- `e`: the function arguments.

template<std::size_t **N**>
auto **operator()**(**const** selector_sequence_type<*N*> &*selector*) **const**

Returns an evaluation of the *xaxis_function*.

Example:

```
auto axis1 = named_axis("abs", axis(16));
auto axis2 = named_axis("ord", axis({'a', 'c', 'i'}));

auto func1 = axis1 < 5 && not_equal(axis2, 'i');

// This will evaluate the xaxis_function for `axis1.label(10) == 9` and
↪`axis2.label(1) == 'c'``
func1({{"abs", 10}, {"ord", 1}});
```

Return the evaluation of the *xaxis_function*

Parameters

- `selector`: a selector_sequence_type for selecting the position where you want to evaluate the function.

1.6.5 xaxis_expression_leaf

Defined in `xframe/xaxis_expression_leaf.hpp`

template<class **CTA**>
class xaxis_expression_leaf : **public** xt::xexpression<*xaxis_expression_leaf*<*CTA*>>

An subset of an expression on axis.

The *xaxis_expression_leaf* class is used with *xaxis_function* for creating an expression on *xnamed_axis*, e.g. `auto expr = not_equal(axis1, 2) && axis2 < 3`. In the example above, `axis1`, `axis2` are *xnamed_axis* which will be converted to xaxis expression leaves before stored in the *xaxis_function* object.

See *xaxis_function*

Template Parameters

- `CTA`: the named axis type.

Public Functions

template<class **AX**>
xaxis_expression_leaf(*AX* &&*n_axis*)

Builds an *xaxis_expression_leaf*.

Parameters

- `n_axis`: the *xnamed_axis*.

template<std::size_t **N**>
auto **operator()**(**const** selector_sequence_type<*N*> &*selector*) **const**

Returns the label from the *xnamed_axis*, given a selector.

Parameters

- `selector`: a selector_sequence_type.

1.6.6 xaxis_view

Defined in `xframe/xaxis_view.hpp`

template<class **L**, class **T**, class **MT** = hash_map_tag>
class xaxis_view

View of an axis.

The *xaxis_view* class is used for modeling a view on an existing axis, i.e. a subset of this axis. This is done by filtering the labels of the axis. No copy is involved. This class is used as a building block for view on coordinates. This class is intended to be used with *xaxis_variant*, so the label template parameter is a type list rather than a simple type.

See *xaxis_variant*

Template Parameters

- `L`: the type list of labels.
- `T`: the integer type used to represent positions.
- `MT`: the tag used for choosing the map type which holds the label- position pairs. Possible values are `map_tag` and `hash_map_tag`. Default value is `hash_map_tag`.

Public Functions

template<class **S**>
xaxis_view(**const** axis_type &*axis*, *S* &&*slice*)

Builds a sliced view of the specified axis.

Parameters

- `axis`: the axis on which the view is built.
- `slice`: the slice used for filtering labels.

operator axis_type() **const**

Converts this view into a real axis.

The view itself is not modified, a new axis is created from the filtered labels. This conversion operator allows to pass a view to methods that accept regular axes, however it might not be convenient for explicit conversion. Prefer as_xaxis in this case.

See *as_xaxis*

auto **labels**() **const**

Returns the list of labels in the view.

Since the view does not hold any data, this list is created upon demand. The filtered labels are copied into it.

auto **label**(size_type *i*) **const**

Return the `i-th` label of the view.

bool **empty**() **const**

Checks if the view has no labels.

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

Returns the number of labels in the axis.

bool **contains**(**const** key_type &*key*) **const**
Returns true if the view contains the speficied label.

Parameters

- `key`: the label to search for.

auto **operator[]**(**const** key_type &*key*) **const**
Returns the position of the specified label in the underlying axis.

If this last one is not found, an exception is thrown.

Parameters

- `key`: the label to search for.

auto **index**(size_type *label_index*) **const**
Get the label mapped to the specified position in the view, and returns it position in the underlying axis.

Parameters

- `label_index`: the index of the label in the view.

auto **find**(**const** key_type &*key*) **const**
Returns a constant iterator to the element with label equivalent to `key`.

If no such element is found, past-the-end iterator is returned.

Parameters

- `key`: the label to search for.

auto **begin**() **const**
Returns a constant iterator to the first element of the view.

This element is a pair label - position.

auto **end**() **const**
Returns a constant iterator to the element following the last element of the view.

auto **cbegin**() **const**
Returns a constant iterator to the first element of the view.

This element is a pair label - position.

auto **cend**() **const**
Returns a constant iterator to the element following the last element of the view.

auto **rbegin**() **const**
Returns a constant iterator to the first element of the reverse view.

This element is a pair labal - position.

auto **rend**() **const**
Returns a constant iterator to the element following the last element of the reversed view.

auto **crbegin**() **const**
Returns a constant iterator to the first element of the reverse view.

This element is a pair labal - position.

auto **crend**() **const**
Returns a constant iterator to the element following the last element of the reversed view.

auto **as_xaxis**() **const**
Converts this view into a real axis.

The view itself is not modified, a new axis is created from the filtered labels.

template<class **F**>
auto **filter**(**const** *F* &*f*) **const**
Builds an return a new axis by applying the given filter to the view.

Parameters

- `f`: the filter used to select the labels to keep in the new axis.

template<class **F**>
auto **filter**(**const** *F* &*f*, size_type *size*) **const**
Builds an return a new axis by applying the given filter to the view.

When the size of the new list of labels is known, this method allows some optimizations compared to the previous one.

Parameters

- `f`: the filter used to select the labels to keep in the new axis.
- `size`: the size of the new label list.

template<class **L**, class **T**, class **MT**>
bool `xf::operator==`(**const** xaxis_view<*L*, *T*, *MT*> &*lhs*, **const** xaxis_view<*L*, *T*, *MT*> &*rhs*)
Returns true is `lhs` and `d` rhs are equivalent axes, i.e.

they contain the same label - position pairs.

Parameters

- `lhs`: an axis.
- `rhs`: an axis.

template<class **L**, class **T**, class **MT**>
bool `xf::operator!=`(**const** xaxis_view<*L*, *T*, *MT*> &*lhs*, **const** xaxis_view<*L*, *T*, *MT*> &*rhs*)
Returns true is `lhs` and `rhs` are not equivalent axes, i.e.

they contain different label - position pairs.

Parameters

- `lhs`: an axis.
- `rhs`: an axis.

template<class **L**, class **T**, class **MT**>
bool `xf::operator==`(**const** xaxis_view<*L*, *T*, *MT*> &*lhs*, **const** xaxis_variant<*L*, *T*, *MT*> &*rhs*)
Returns true is `lhs` and `rhs` are equivalent axes, i.e.

they contain the same label - position pairs.

Parameters

- `lhs`: an axis.
- `rhs`: an axis.

template<class **L**, class **T**, class **MT**>
bool `xf::operator!=`(**const** xaxis_view<*L*, *T*, *MT*> &*lhs*, **const** xaxis_variant<*L*, *T*, *MT*> &*rhs*)
Returns true is `lhs` and `rhs` are not equivalent axes, i.e.

they contain different label - position pairs.

Parameters

- `lhs`: an axis.
- `rhs`: an axis.

template<class **L**, class **T**, class **MT**>
bool `xf::operator==`(**const** xaxis_variant<*L*, *T*, *MT*> &*lhs*, **const** xaxis_view<*L*, *T*, *MT*> &*rhs*)
Returns true is `lhs` and `rhs` are equivalent axes, i.e.

they contain the same label - position pairs.

Parameters

- `lhs`: an axis.
- `rhs`: an axis.

template<class **L**, class **T**, class **MT**>
bool `xf::operator!=`(**const** xaxis_variant<*L*, *T*, *MT*> &*lhs*, **const** xaxis_view<*L*, *T*, *MT*> &*rhs*)
Returns true is `lhs` and `rhs` are not equivalent axes, i.e.

they contain different label - position pairs.

Parameters

- `lhs`: an axis.
- `rhs`: an axis.

1.6.7 xaxis_variant

Defined in `xframe/xaxis_variant.hpp`

template<class **L**, class **T**, class **MT** = hash_map_tag>
class xaxis_variant
Axis whose label type is a variant.

The *xaxis_variant* holds a variant of axes with different label types. It provides the same API as a regular axis, wrapping the visitor mechanism required to access the underlying axis. This allows to store axes with different label types in a coordinate system.

Template Parameters

- `L`: the type list of labels
- `T`: the integer type used to represent positions.
- `MT`: the tag used for choosing the map type which holds the label- position pairs. Possible values are `map_tag` and `hash_map_tag`. Default value is `hash_map_tag`.

Constructors

template<class **LB**>
xaxis_variant(**const** xaxis<*LB*, T, MT> &*axis*)
Constructs an *xaxis_variant* from the specified xaxis.

This latter is copied in the variant.

Template Parameters

- `LB`: the label type of the axis argument.

Parameters

- `axis`: the axis to copy in the variant.

template<class **LB**>
xaxis_variant(xaxis<*LB*, T, MT> &&*axis*)
Constructs an *xaxis_variant* from the specified xaxis.

This latter is moved in the variant.

Template Parameters

- `LB`: the label type of the axis argument.

Parameters

- `axis`: the axis to move in the variant.

template<class **LB**>
xaxis_variant(**const** xaxis_default<*LB*, T> &*axis*)
Constructs an *xaxis_variant* from the specified *xaxis_default*.

This latter is copied in the variant.

Template Parameters

- `LB`: the label type of the axis argument.

Parameters

- `axis`: the axis to copy in the variant.

template<class **LB**>
xaxis_variant(xaxis_default<*LB*, T> &&*axis*)
Constructs an *xaxis_variant* from the specified *xaxis_default*.

This latter is moved in the variant.

Template Parameters

- `LB`: the label type of the axis argument.

Parameters

- `axis`: the axis to move in the variant.

Data

bool **contains**(**const** key_type &*key*) **const**
Returns true if the axis contains the speficied label.

Parameters

- `key`: the label to search for.

auto **operator[]**(**const** key_type &*key*) **const**
Returns the position of the specified label.

If this last one is not found, an exception is thrown.

Parameters

- `key`: the label to search for.

Filters

template<class **F**>
auto **filter**(**const** *F* &*f*) **const**
Builds an return a new axis by applying the given filter to the axis.

Parameters

- `f`: the filter used to select the labels to keep in the new axis.

template<class **F**>
auto **filter**(**const** *F* &*f*, size_type *size*) **const**
Builds an return a new axis by applying the given filter to the axis.

When the size of the new list of labels is known, this method allows some optimizations compared to the previous one.

Parameters

- `f`: the filter used to select the labels to keep in the new axis.
- `size`: the size of the new label list.

Set operations

template<class ...**Args**>
bool **merge**(**const** *Args*&... *axes*)
Merges all the axes arguments into this ones.

After this function call, the axis contains all the labels from all the arguments.

Return true is the axis already contained all the labels.

Parameters

- `axes`: the axes to merge.

template<class ...**Args**>
bool **intersect**(**const** *Args*&... *axes*)
Replaces the labels with the intersection of the labels of the axes arguments and the labels of this axis.

Return true if the intersection is equivalent to this axis.

Parameters

- `axes`: the axes to intersect.

Labels

auto **`labels`**`()` **`const`**
Returns the list of labels contained in the axis.

auto **`label`**(size_type *i*) **`const`**
Return the i-th label of the axis.

Parameters

- `i`: the position of the label.

bool **`empty`**`()` **`const`**
Checks if the axis has no labels.

auto **`size`**`()` **`const`**
Returns the number of labels in the axis.

bool **`is_sorted`**`()` **`const`**
Returns true if the labels list is sorted.

Iterators

auto **`find`**(**`const`** key_type *&key*) **`const`**
Returns a constant iterator to the element with label equivalent to `key`.

If no such element is found, past-the-end iterator is returned.

Parameters

- `key`: the label to search for.

auto **`begin`**`()` **`const`**
Returns a constant iterator to the first element of the axis.

This element is a pair label - position.

auto **`end`**`()` **`const`**
Returns a constant iterator to the element following the last element of the axis.

auto **`cbegin`**`()` **`const`**
Returns a constant iterator to the first element of the axis.

This element is a pair label - position.

auto **`cend`**`()` **`const`**
Returns a constant iterator to the element following the last element of the axis.

auto **`rbegin`**`()` **`const`**
Returns a constant iterator to the first element of the reverse axis.

This element is a pair labal - position.

auto **rend**() **const**
Returns a constant iterator to the element following the last element of the reversed axis.

auto **crbegin**() **const**
Returns a constant iterator to the first element of the reverse axis.

This element is a pair labal - position.

auto **crend**() **const**
Returns a constant iterator to the element following the last element of the reversed axis.

Public Functions

bool **operator==**(**const** self_type &*rhs*) **const**
Returns true is this axis and `rhs` are equivalent axes, i.e.

they contain the same label - position pairs.

Parameters

- `rhs`: an axis.

bool **operator!=**(**const** self_type &*rhs*) **const**
Returns true is this axis and `rhs` are not equivalent axes, i.e.

they contain different label - position pairs.

Parameters

- `rhs`: an axis.

1.6.8 xnamed_axis

Defined in `xframe/xnamed_axis.hpp`

template<class **K**, class **T**, class **MT** = hash_map_tag, class **L** = xtl::mpl::vector<int, std::size_t, char, xf::fstring>, class **LT** = xtl::mpl::ca
class xnamed_axis : **public** xt::xexpression<*xnamed_axis*<*K*, *T*, *MT*, *L*, *LT*>>
Class modeling a dimension name - axis pair in a coordinate system.

The *xnamed_axis* class stores a dimension name and an xaxis as an *xaxis_variant*.

See *xaxis_variant*, *xaxis*

Template Parameters

- `K`: the dimension name type.
- `T`: the integer type used to represent positions in the underlying *xaxis_variant*.

Public Functions

template<class **A**>
xnamed_axis(**const** name_type &*name*, *A* &&*axis*)
Builds an *xnamed_axis*.

Parameters

- `name`: the dimension name.
- `axis`: the underlying xaxis.

template<class **A**>
xnamed_axis(name_type &&*name*, *A* &&*axis*)
Builds an *xnamed_axis*.

Parameters

- `name`: the dimension name.
- `axis`: the underlying xaxis.

auto **name**() **const** &
Returns the name of the *xnamed_axis*.

Return the dimension name.

auto **axis**() **const** &
Returns the underlying *xaxis_variant*.

Return the *xaxis_variant*.

See *xaxis_variant*

auto **label**(size_type *i*) **const** &
Returns the label at the given position of the underlying xaxis.

Return the label at the given position of the underlying xaxis.

Parameters

- `i`: the position.

template<class **K**, class **A**>
auto `xf::`**named_axis**(**const** *K* &*name*, *A* &&*axis*)
Builder function which creates an *xnamed_axis*.

Return the created *xnamed_axis*.

Parameters

- `name`: the dimension name.
- `axis`: the xaxis that you want to store in the *xnamed_axis*.

Template Parameters

- `K`: the dimension name type.

- `A`: the axis type.

1.7 Coordinates and dimensions

1.7.1 xcoordinate_base

Defined in `xframe/xcoordinate_base.hpp`

template<class **K**, class **A**>
class xcoordinate_base

Base class for coordinates.

The *xcoordinate_base* class defines the common interface for coordinates, which define the mapping of dimension names to axes.

Template Parameters

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

Public Functions

bool **empty**() **const**

Returns true if the coordinates is empty, i.e.

it contains no mapping of axes with dimension names.

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

Returns the number of axes in the coordinates.

bool **contains**(**const** key_type &*key*) **const**

Returns true if the coordinates contains the specified dimension name.

Parameters

- `key`: the dimension name to search for.

bool **contains**(**const** key_type &*key*, **const** label_type &*label*) **const**

Returns true if the coordinates contains the specified dimension name and if the axis mapped with this name contains the specified label.

Parameters

- `key`: the dimension name to search for.
- `label`: the label to search for in the mapped axis.

auto **operator[]**(**const** key_type &*key*) **const**

Returns the axis mapped to the specified dimension name.

If this last one is not found, throws an exception.

Parameters

- `key`: the name of the dimension to search for.

auto **data**() **const**
Returns the container of the dimension names to axes mapping.

auto **find**(**const** key_type &*key*) **const**
Returns a constant iterator to the axis mapped to the specified dimension name.

If no such element is found, past-the-end iterator is returned.

Parameters

- `key`: the dimension name to search for.

auto **begin**() **const**
Returns a constant iterator to the first element of the coordinates.

Such an element is a pair dimension name - axis.

auto **end**() **const**
Returns a constant iterator to the element following the last element of the coordinates.

auto **cbegin**() **const**
Returns a constant iterator to the first element of the coordinates.

Such an element is a pair dimension name - axis.

auto **cend**() **const**
Returns a constant iterator to the element following the last element of the coordinates.

auto **key_begin**() **const**
Returns a constant iterator to the first dimension name of the coordinates.

auto **key_end**() **const**
Returns a constant iterator to the element following the last dimension name of the coordinates.

template<class **KB**, class **LB**>
auto **operator[]**(**const** std::pair<*KB*, *LB*> &*key*) **const**
Returns the position of the specified labels of the axis mapped to the specified dimension name.

Throws an exception if either the dimension name or the label is not part of this coordinate.

Parameters

- `key`: the pair dimension name - label to search for.

template<class **K**, class **A1**, class **A2**>
bool `xf::`**operator==**(**const** xcoordinate_base<*K*, *A1*> &*lhs*, **const** xcoordinate_base<*K*, *A2*> &*rhs*)
Returns true if `lhs` and `rhs` are equivalent coordinates, i.e.

they hold the same axes mapped to the same dimension names.

Parameters

- `lhs`: a coordinate object.
- `rhs`: a coordinate object.

template<class **K**, class **A1**, class **A2**>
bool `xf::`**operator!=**(**const** xcoordinate_base<*K*, *A1*> &*lhs*, **const** xcoordinate_base<*K*, *A2*> &*rhs*)
Returns true if `lhs` and `rhs` are not equivalent coordinates, i.e.

they hold different axes or the dimension names to axes mappings are different.

Parameters

- `lhs`: a coordinate object.

- `rhs`: a coordinate object.

1.7.2 xcoordinate

Defined in `xframe/xcoordinate.hpp`

template<class **K**, class **L** = xtl::mpl::vector<int, std::size_t, char, xf::fstring>, class **S** = std::size_t, class **MT** = hash_map_tag>
class xcoordinate : **public** xf::xcoordinate_base<*K*, xaxis_variant<*L*, *S*, *MT*>>

Class modeling coordinates.

The xcoordinate class is used for modeling coordinates. Coordinates are mapping of dimension names to axes. Axes in a xcoordinate object can have different label types.

Template Parameters

- `K`: the type of dimension names.
- `L`: the type list of axes labels.
- `S`: the integer type used to represent positions in axes. Default value is `std::size_t`.
- `MT`: the tag used for choosing the map type which holds the label- position pairs in the axes. Possible values are `map_tag` and `hash_map_tag`. Default value is `hash_map_tag`.

Public Functions

xcoordinate(**const** map_type &*axes*)

Constructs an xcoordinate object with the given mapping of dimension names to axes.

This mapping is copied.

Parameters

- `axes`: the dimension names to axes mapping.

xcoordinate(map_type &&*axes*)

Constructs an xcoordinate object with the given mapping of dimension names to axes.

This mapping is moved and therefore it is invalid after the xcoordinate has been constructed.

Parameters

- `axes`: the dimension names to axes mapping.

xcoordinate(std::initializer_list<value_type> *init*)

Constructs an xcoordinate object from the given initializer list of dimension names - axes pairs.

template<class ...**K1**, class ...**LT**>
xcoordinate(xnamed_axis<*K1*, S, MT, L, *LT*>... *axes*)

Constructs an xcoordinate object from the given named axes.

void **clear**()

Removes all the elements from the xcoordinate.

After this call, *`size()`* returns zero.

template<class **Join**, class ...**Args**>
xtrivial_broadcast **broadcast**(**const** *Args*&... *coordinates*)

Broadcast the specified coordinates to this xcoordinate.

Return an object specifying if the labels and the dimension of the coordinates are the same.

Parameters

- `coordinates`: the coordinates to broadcast.

template<class **K** = fstring, class **L** = xtl::mpl::vector<int, std::size_t, char, xf::fstring>, class **S** = std::size_t, class **MT** = hash_map_tag>
xcoordinate<*K*, *L*, *S*, *MT*> `xf::coordinate`(**const** std::map<*K*, xaxis_variant<*L*, *S*, *MT*>> &*axes*)

Builds and returns an xcoordinate object from the specified mapping of dimension names to axes.

The map is copied.

Parameters

- `axes`: the dimension names to axes mapping.

template<class **K** = fstring, class **L** = xtl::mpl::vector<int, std::size_t, char, xf::fstring>, class **S** = std::size_t, class **MT** = hash_map_tag>
xcoordinate<*K*, *L*, *S*, *MT*> `xf::coordinate`(std::map<*K*, xaxis_variant<*L*, *S*, *MT*>> &&*axes*)

Builds and returns an xcoordinate object from the specified mapping of dimension names to axes.

The map is moved, therefore it is invalid after the xcoordinate object has been built.

Parameters

- `axes`: the dimension names to axes mapping.

template<class **K**, class ...**K1**, class **S**, class **MT**, class **L**, class **LT**, class ...**LT1**>
xcoordinate<*K*, *L*, *S*, *MT*> `xf::coordinate`(xnamed_axis<*K*, *S*, *MT*, *L*, *LT*> *axis*, xnamed_axis<*K1*, *S*, *MT*, *L*, *LT1*>... *axes*)

Builds and returns an xcoordinate object from the specified list of named axes.

Parameters

- `axes`: the list of named axes.

template<class **Join**, class **K**, class **L**, class **S**, class **MT**, class ...**Args**>
xtrivial_broadcast `xf::broadcast_coordinates`(xcoordinate<*K*, *L*, *S*, *MT*> &*output*, **const** *Args*&... *coordinates*)

Broadcast a list of coordinates to the specified output coordinate.

Parameters

- `output`: the xcoordinate result.
- `coordinates`: the list of xcoordinate objects to broadcast.

1.7.3 xcoordinate_view

Defined in `xframe/xcoordinate_view.hpp`

template<class **K**, class **L** = xtl::mpl::vector<int, std::size_t, char, xf::fstring>, class **S** = std::size_t, class **MT** = hash_map_tag>
class xcoordinate_view : **public** xf::xcoordinate_base<*K*, xaxis_view<*L*, *S*, *MT*>>

view of an xcoordinate

The *xcoordinate_view* is used for modeling a view on an existing xcoordinate, i.e. a subset of this xcoordinate. This is done by either ignoring some axes, or by holding views on axes of the underlying xcoordinate.

See *xaxis_view*

Template Parameters

- `K`: the type of dimension names.

- `L`: the type list of axes labels.
- `S`: the integer type used to represent positions in axes. Default value is `std::size_t`.
- `MT`: the tag used for choosing the map type which holds the label- position pairs in the axes. Possible values are `map_tag` and `hash_map_tag`. Default value is `hash_map_tag`.

Public Functions

xcoordinate_view(**const** map_type &*axes*)

Constructs an *xcoordinate_view* with the given mapping of dumension names to views on axes.

The mapping is copied.

Parameters

- `axes`: the dimension names to views on axes mapping.

xcoordinate_view(map_type &&*axes*)

Constructs an *xcoordinate_view* with the given mapping of dumension names to views on axes.

This mapping is moved and therefore it is invalid after the *xcoordinate_view* has been constructed.

Parameters

- `axes`: the dimension names to views on axes mapping.

template<class **K**, class **L**, class **S**, class **MT**>
xcoordinate_view<*K*, *L*, *S*, *MT*> `xf::coordinate_view`(**const** std::map<*K*, xaxis_view<*L*, *S*, *MT*>> &*axes*)

Builds and returns an *xcoordinate_view* from the specified mapping of dimension names to views on axes.

The map is copied.

Parameters

- `axes`: the dimension names to views on axes mapping.

template<class **K**, class **L**, class **S**, class **MT**>
xcoordinate_view<*K*, *L*, *S*, *MT*> `xf::coordinate_view`(std::map<*K*, xaxis_view<*L*, *S*, *MT*>> &&*axes*)

Builds and returns an xcoordinate_viewfrom the specified mapping of dimension names to views on axes.

The map is moved, therefore it is invalid after the xcoordinate object has been built.

Parameters

- `axes`: the dimension names to views on axes mapping.

1.7.4 xcoordinate_chain

Defined in `xframe/xcoordinate_chain.hpp`

template<class **C**>
class xcoordinate_chain

reindexed view of an xcoordinate.

The *xcoordinate_chain* is used for creating a reindexed view on an existing xcoordinate. This is done by holding new axes and keeping a reference to the underlying xcoordinate.

See *xcoordinate*

See *xcoordinate_expanded*

Template Parameters

- `C`: the type of the underlying xcoordinate.

Public Functions

xcoordinate_chain(**const** coordinate_type &*sub_coord*, **const** map_type &*new_coord*)
Constructs an *xcoordinate_chain*, given the reference to an xcoordinate and the reference to the additional coordinates.

Parameters

- `sub_coord`: the reference to the underlying xcoordinate.
- `new_coord`: the `dimension name <-> axis` mapping of the new coordinates.

xcoordinate_chain(**const** coordinate_type &*sub_coord*, map_type &&*new_coord*)
Constructs an *xcoordinate_chain*, given the reference to an xcoordinate and the reference to the additional coordinates.

Parameters

- `sub_coord`: the reference to the underlying xcoordinate.
- `new_coord`: the `dimension name <-> axis` mapping of the new coordinates.

bool **empty**() **const**
Returns true if the coordinates is empty, i.e.

it contains no mapping of axes with dimension names.

auto **size**() **const**
Returns the number of axes in the coordinates.

bool **contains**(**const** key_type &*key*) **const**
Returns true if the coordinates contains the specified dimension name.

Parameters

- `key`: the dimension name to search for.

bool **contains**(**const** key_type &*key*, **const** label_type &*label*) **const**
Returns true if the coordinates contains the specified dimension name and if the axis mapped with this name contains the specified label.

Parameters

- `key`: the dimension name to search for.
- `label`: the label to search for in the mapped axis.

auto **operator[]**(**const** key_type &*key*) **const**
Returns the axis mapped to the specified dimension name.

If this last one is not found, throws an exception.

Parameters

- `key`: the name of the dimension to search for.

auto **find**(**const** key_type &*key*) **const**
Returns a constant iterator to the axis mapped to the specified dimension name.

If no such element is found, past-the-end iterator is returned.

Parameters

- `key`: the dimension name to search for.

auto **begin**() **const**
Returns a constant iterator to the first element of the coordinates.

Such an element is a pair dimension name - axis.

auto **end**() **const**
Returns a constant iterator to the element following the last element of the coordinates.

auto **cbegin**() **const**
Returns a constant iterator to the first element of the coordinates.

Such an element is a pair dimension name - axis.

auto **cend**() **const**
Returns a constant iterator to the element following the last element of the coordinates.

auto **key_begin**() **const**
Returns a constant iterator to the first dimension name of the coordinates.

auto **key_end**() **const**
Returns a constant iterator to the element following the last dimension name of the coordinates.

template<class **KB**, class **LB**>
auto **operator[]**(**const** std::pair<*KB*, *LB*> &*key*) **const**
Returns the position of the specified labels of the axis mapped to the specified dimension name.

Throws an exception if either the dimension name or the label is not part of this coordinate.

Parameters

- `key`: the pair dimension name - label to search for.

template<class **C**>
xcoordinate_chain<*C*> `xf::`**reindex**(**const** *C* &*coordinate*, **const** **typename** *C*::map_type &*new_coord*)
Creates a reindexed view on an xcoordinate, given the reference on the xcoordinate and the new coordinates.

Parameters

- `sub_coord`: the reference to the underlying xcoordinate.
- `new_coord`: the `dimension name <-> axis` mapping of the new coordinates.

template<class **C**>
xcoordinate_chain<*C*> `xf::`**reindex**(**const** *C* &*coordinate*, **typename** *C*::map_type &&*new_coord*)
Creates a reindexed view on an xcoordinate, given the reference on the xcoordinate and the new coordinates.

Parameters

- `sub_coord`: the reference to the underlying xcoordinate.
- `new_coord`: the `dimension name <-> axis` mapping of the new coordinates.

1.7.5 xcoordinate_expanded

Defined in `xframe/xcoordinate_expanded.hpp`

template<class **C**>
class xcoordinate_expanded
view of an xcoordinate which expands the xcoordinate dimensions

The *xcoordinate_expanded* is used for modeling an expanded view on an existing xcoordinate, i.e. a superset of this xcoordinate. This is done by holding an extra coordinate map and keeping a reference to the underlying xcoordinate.

See *xcoordinate*

See *xcoordinate_chain*

Template Parameters

- `C`: the type of the underlying xcoordinate.

Public Functions

xcoordinate_expanded(**const** coordinate_type &*sub_coord*, **const** map_type &*new_coord*)
Constructs an *xcoordinate_expanded*, given the reference to an xcoordinate and the reference to the additional dimension mapping.

Parameters

- `sub_coord`: the reference to the underlying xcoordinate.
- `new_coord`: the `dimension name <-> axis` mapping of the additional coordinates.

xcoordinate_expanded(**const** coordinate_type &*sub_coord*, map_type &&*new_coord*)
Constructs an *xcoordinate_expanded*, given the reference to an xcoordinate and the reference to the additional dimension mapping.

Parameters

- `sub_coord`: the reference to the underlying xcoordinate.
- `new_coord`: the `dimension name <-> axis` mapping of the additional coordinates.

bool **empty**() **const**
Returns true if the coordinates is empty, i.e.

it contains no mapping of axes with dimension names.

auto **size**() **const**
Returns the number of axes in the coordinates.

bool **contains**(**const** key_type &*key*) **const**
Returns true if the coordinates contains the specified dimension name.

Parameters

- `key`: the dimension name to search for.

bool **contains**(**const** key_type &*key*, **const** label_type &*label*) **const**
Returns true if the coordinates contains the specified dimension name and if the axis mapped with this name contains the specified label.

Parameters

- `key`: the dimension name to search for.
- `label`: the label to search for in the mapped axis.

auto **operator[]**(**const** key_type &*key*) **const**
Returns the axis mapped to the specified dimension name.

If this last one is not found, throws an exception.

Parameters

- `key`: the name of the dimension to search for.

auto **find**(**const** key_type &*key*) **const**
Returns a constant iterator to the axis mapped to the specified dimension name.

If no such element is found, past-the-end iterator is returned.

Parameters

- `key`: the dimension name to search for.

auto **begin**() **const**
Returns a constant iterator to the first element of the coordinates.

Such an element is a pair dimension name - axis.

auto **end**() **const**
Returns a constant iterator to the element following the last element of the coordinates.

auto **cbegin**() **const**
Returns a constant iterator to the first element of the coordinates.

Such an element is a pair dimension name - axis.

auto **cend**() **const**
Returns a constant iterator to the element following the last element of the coordinates.

auto **key_begin**() **const**
Returns a constant iterator to the first dimension name of the coordinates.

auto **key_end**() **const**
Returns a constant iterator to the element following the last dimension name of the coordinates.

template<class **KB**, class **LB**>
auto **operator[]**(**const** std::pair<*KB*, *LB*> &*key*) **const**
Returns the position of the specified labels of the axis mapped to the specified dimension name.

Throws an exception if either the dimension name or the label is not part of this coordinate.

Parameters

- `key`: the pair dimension name - label to search for.

template<class **C**>

xcoordinate_expanded<*C*> `xf::`**`expand_dims`**`(`**`const`** *C* &*coordinate*, **`const typename`** *C*::map_type &*new_coord*)

Expand the dimensions of an xcoordinate, given the reference to an xcoordinate and the reference to the additional dimension mapping.

Parameters

- `sub_coord`: the reference to the underlying xcoordinate.
- `new_coord`: the `dimension name <-> axis` mapping of the additional coordinates.

template<class **C**>
xcoordinate_expanded<*C*> `xf::`**`expand_dims`**`(`**`const`** *C* &*coordinate*, **`typename`** *C*::map_type &&*new_coord*)

Expand the dimensions of an xcoordinate, given the reference to an xcoordinate and the reference to the additional dimension mapping.

Parameters

- `sub_coord`: the reference to the underlying xcoordinate.
- `new_coord`: the `dimension name <-> axis` mapping of the additional coordinates.

1.7.6 xdimension

Defined in `xframe/xdimension.hpp`

template<class **L**, class **T** = std::size_t>
`class xdimension` : **`private`** xf::xaxis<*L*, *T*, map_tag>

Class modeling dimensions.

The xdimension class is used for modeling the mapping of dimension names to their positions in a data tensor. This class is a special axis with a broadcast method instead of merge and intersect, thus its API is really close the one of `xaxis`.

See *xaxis*

Template Parameters

- `L`: the type of dimension name.
- `T`: the integer type use to represent the positions of the dimensions. Default value is `std::size_t`.

Public Functions

`xdimension``()`

Constructs an empty xdimension object.

`xdimension``(`**`const`** label_list &*labels*)

Constructs an xdimension object with the given list of dimension labels.

This list is copied.

Parameters

- `labels`: the list of dimension names.

xdimension(label_list &&*labels*)
Constructs an xdimension object with the given list of dimension labels.

This list is moved and therefore it is invalid after the xdimension object has been constructed.

Parameters

- `labels`: the list of dimension names.

xdimension(std::initializer_list<key_type> *init*)
Constructs an xdimension object from the given initializer list of dimension names.

Parameters

- `init`: the list of dimension names.

template<class **InputIt**>
xdimension(*InputIt first*, *InputIt last*)
Constructs an xdimension object from the content of the range [first, last)

Parameters

- `first`: An iterator to the first dimension name.
- `last`: An iterator the the element following the last dimension name.

template<class ...**Args**>
bool **broadcast**(**const** *Args*&... *dims*)
Broadcast the specified dimensions to this xdimension object.

Return true if the dimension objects are the same.

Parameters

- `dims`: the xdimension objects to broadcast.

template<class **L**, class **T**>
bool `xf::operator==`(**const** xdimension<*L*, *T*> &*lhs*, **const** xdimension<*L*, *T*> &*rhs*)
Returns true if `lhs` and `rhs` are equivalent dimension mappings, i.e.

they contain the same dimension name - dimension position pairs.

Parameters

- `lhs`: an xdimension object.
- `rhs`: an xdimension obejct.

template<class **L**, class **T**>
bool `xf::operator!=`(**const** xdimension<*L*, *T*> &*lhs*, **const** xdimension<*L*, *T*> &*rhs*)
Returns true if `lhs` and `rhs` are not equivalent dimension mappings, i.e.

they contain different dimension name - dimension position pairs.

Parameters

- `lhs`: an xdimension object.
- `rhs`: an xdimension obejct.

1.8 Variables

1.8.1 xexpand_dims_view

Defined in `xframe/xexpand_dims_view.hpp`

template<class **CT**>
class xexpand_dims_view : **public** xt::xexpression<*xexpand_dims_view*<*CT*>>, **private** xf::xvariable_base<*xexpand_dim*

View on a variable with additional axes.

The *xvariable_masked_view* class is used for creating a view on a variable with additional axes.

Template Parameters

- `CT`: the closure type on the underlying variable.

Public Functions

template<class **E**>
xexpand_dims_view(*E* &&*e*, **const** extra_dimensions_type &*dims*)

Constructs an *xexpand_dims_view* given a map of extra dimensions.

Parameters

- `e`: the variable expression on which to create the view.
- `dims`: the map of extra dimensions, the key being the dimension name, the value the position where to put the new dimension.

template<class **E**, class **K**>
auto `xf::expand_dims`(*E* &&*e*, std::initializer_list<*K*> *dim_names*)

Creates a view on a variable expression with additional axes, given a list of extra dimension names.

Parameters

- `e`: the variable expression on which to create the view.
- `dim_names`: the list of extra dimension names.

template<class **E**>
auto `xf::expand_dims`(*E* &&*e*, std::initializer_list<std::pair<**typename** std::decay_t<*E*>::key_type, std::size_t>> *dims*)

Creates a view on a variable expression with additional axes, given a list of extra dimensions.

Parameters

- `e`: the variable expression on which to create the view.
- `dims`: the extra dimensions as a mapping "dimension name" -> "position".

1.8.2 xvariable_masked_view

Defined in `xframe/xvariable_masked_view.hpp`

template<class **CTV**, class **CTAX**>
class xvariable_masked_view : **public** xt::xview_semantic<*xvariable_masked_view*<*CTV*, *CTAX*>>, **private** xf::xvaria

View on a variable which will apply a mask on the variable, given an expression on the axes.

The *xvariable_masked_view* class is used for applying a mask on a variable, avoiding assignment to masked values when assigning a scalar or an other variable to the view. The mask is created given an expression on the axes.

Template Parameters

- `CTV`: the closure type on the underlying variable.
- `CTAX`: the closure type on the axes function.

Public Functions

template<class **V**, class **AX**>
xvariable_masked_view(*V &&variable_expr*, *AX &&axis_expr*)

Builds an *xvariable_masked_view*.

Parameters

- `variable_expr`: the underlying variable.
- `axis_expr`: the axis expression.

template<class **EV**, class **EAX**>
auto `xf::`**where**(*EV &&variable_expr*, *EAX &&axis_expr*)

Apply a mask on a variable where the axis expression is false.

e.g.

```
// Will only assign 36 to values where the ordinate label is lower than 6
where(var, var.axis<int>("ordinate") < 6) = 36;

// Will only add 2.4 to values where the abscissa is not equal to 'm' and the␣
↪ordinate is not equal to 1
where(
     var,
     not_equal(var.axis<char>("abscissa"), 'm')) && not_equal(var.axis<int>(
↪"ordinate"), 1)
) += 2.4;
```

Return an *xvariable_masked_view*.

Parameters

- `variable_expr`: the variable.
- `axis_expr`: the axis expression.

1.9 From xarray to xframe

1.9.1 Containers

Python 3 - xarray	C++ 14 - xframe
`xr.DataArray([[1, 2], [3, 4]], [('x', ['a', 'b']), ('y', [1, 4])])`	`xf::variable<double>({{1, 2}, {3, 4}}, {{"x", xf::axis({"a", "b"})}, {"y", xf::axis({1, 4})}})`

1.9.2 Indexing

xframe returns views for multi-selection, no copy is made.

Python 3 - xarray	C++ 14 - xframe
`da[0, 1]`	`v(0, 1)`
`da.loc['a', 'Paris']`	`v.locate("a", "Paris")`
`da.isel(city=1, group=0)`	`v.iselect({{"city", 1}, {"group", 0}})`
`da.sel(city='Paris', group='a')`	`v.select({{"city", "Paris"}, {"group", "a"}})`
`da[:, 1]`	`xf::ilocate(v, xf::iall(), 1)`
`da.loc[:, 'Paris']`	`xf::locate(v, xf::all(), "Paris")`
`da.isel(city=1)`	`xf::iselect(v, {{"city", 1}})`
`da.sel(city='Paris')`	`xf::select(v, {{"city", "Paris"}})`
`da.reindex(city=['NYC', 'Paris'])`	`xf::reindex(v, {{"city", xf::axis({"NYC", "Paris"})}})`
`da.reindex_like(df)`	`xf::reindex_like(v, v2)`

1.9.3 Logical

Logical universal functions are truly lazy. `xf::where(condition, a, b)` does not evaluate `a` where `condition` is falsy, and it does not evaluate `b` where `condition` is truthy. *xarray* relies on *numpy* functions, that can also operate on `xarray.DataArray`.

Python 3 - xarray	C++ 14 - xframe
`xr.where(a > 5, a, b)  xr.where(a > 5, 100, a)`	`xf::where(a > 5, a, b)  xf::where(a > 5, 100, a)`
`np.any(a)`	`xf::any(a)`
`np.all(a)`	`xf::all(a)`
`np.logical_and(a, b)`	`a && b`
`np.logical_or(a, b)`	`a || b`
`np.isclose(a, b)`	`xf::isclose(a, b)`
`np.allclose(a, b)`	`xf::allclose(a, b)`

1.9.4 Comparisons

xarray relies on *numpy* functions, that can also operate on `xarray.DataArray`.

Python 3 - xarray	C++ 14 - xframe
`np.equal(a, b)`	`xf::equal(a, b)`
`np.not_equal(a, b)`	`xf::not_equal(a, b)`
`np.less(a, b)`	`xf::less(a, b)` `a < b`
`np.less_equal(a, b)`	`xf::less_equal(a, b)` `a <= b`
`np.greater(a, b)`	`xf::greater(a, b)` `a > b`
`np.greater_equal(a, b)`	`xf::greater_equal(a, b)` `a >= b`

1.9.5 Mathematical functions

xframe universal functions are provided for a large set number of mathematical functions. *xarray* relies on *numpy* functions, that can also operate on `xarray.DataArray`.

Basic functions:

Python 3 - xarray	C++ 14 - xframe
`np.absolute(a)`	`xf::abs(a)`
`np.sign(a)`	`xf::sign(a)`
`np.remainder(a, b)`	`xf::remainder(a, b)`
`np.clip(a, min, max)`	`xf::clip(a, min, max)`
	`xf::fma(a, b, c)`

Exponential functions:

Python 3 - xarray	C++ 14 - xframe
`np.exp(a)`	`xf::exp(a)`
`np.expm1(a)`	`xf::expm1(a)`
`np.log(a)`	`xf::log(a)`
`np.log1p(a)`	`xf::log1p(a)`

Power functions:

Python 3 - xarray	C++ 14 - xframe
`np.power(a, p)`	`xf::pow(a, b)`
`np.sqrt(a)`	`xf::sqrt(a)`
`np.square(a)`	`xf::square(a) xf::cube(a)`
`np.cbrt(a)`	`xf::cbrt(a)`

Trigonometric functions:

Python 3 - xarray	C++ 14 - xframe
`np.sin(a)`	`xf::sin(a)`
`np.cos(a)`	`xf::cos(a)`
`np.tan(a)`	`xf::tan(a)`

Hyperbolic functions:

Python 3 - xarray	C++ 14 - xframe
`np.sinh(a)`	`xf::sinh(a)`
`np.cosh(a)`	`xf::cosh(a)`
`np.tanh(a)`	`xf::tanh(a)`

Error and gamma functions:

Python 3 - xarray	C++ 14 - xframe
`scipy.special.erf(a)`	`xf::erf(a)`
`scipy.special.gamma(a)`	`xf::tgamma(a)`
`scipy.special.gammaln(a)`	`xf::lgamma(a)`

INDEX

X