xtensor-io

Wolf Vollprecht, Johan Mabille and Sylvain Corlay

Dec 01, 2021

INSTALLATION

Input/Output routines for reading images, audio, and NumPy `npz` files for the xtensor C++ multi-dimensional array library.

xtensor-io offers bindings to popular open source libraries for reading and writing

- Images with OpenImageIO (many supported formats, among others: jpg, png, gif, tiff, bmp ...)
- Sound files with libsndfile (supports wav, ogg files)
- NumPy compressed file format (`npz`). Note: support for uncompressed NumPy files (`npy`) is available in core xtensor.
- Geospatial rasters with GDAL (many supported formats)

CHAPTER

ONE

ENABLING XTENSOR-IO IN YOUR C++ LIBRARIES

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

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

CHAPTER

TWO

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.

2.1 Installation

xtensor-io is a header-only library but depends on some traditional libraries that need to be installed. On Linux, installation of the dependencies can be done through the package manager, anaconda or manual compilation.

2.1.1 Using the conda-forge package

A package for xtensor-io is available for the mamba (or conda) package manager.

The package will also pull all the dependencies (OpenImageIO, libsndfile and zlib).

```
mamba install xtensor-io -c conda-forge
```

The easiest way to make use of xtensor-io in your code is by using cmake for your project. In order for cmake to pick up the xtensor-io dependency, just utilize the interface target and link the xtensor-io library to your target.

```
add_executable(my_exec my_exec.cpp)
target_link_libraries(my_exec
  PUBLIC
    xtensor-io
)
```

This should be enough to add the correct directories to the include_directories and link the required libraries. However, depending on your system setup there might be problmes upon executing, as the dynamic library is not picked up correctly. So if you run into errors that read something like "Library could not be opened ... ", then set the `RPATH`, the runtime library search path, to the `conda` library path of your environment. We utilize the `CMAKE_INSTALL_PREFIX` for this purpose, so if you call cmake like this `cmake .. -DCMAKE_INSTALL_PREFIX=$CONDA_PREFIX` and add the following to your `CMakeLists.txt`.

```
set_target_properties(my_exec
  PROPERTIES
    INSTALL_RPATH "${CMAKE_INSTALL_PREFIX}/lib;${CMAKE_INSTALL_PREFIX}/${CMAKE_INSTALL_
↪LIBDIR}"
```

(continues on next page)

(continued from previous page)

```
    BUILD_WITH_INSTALL_RPATH ON
)
```

2.1.2 From source with cmake

You can also install `xtensor-io` from source with cmake. On Unix platforms, from the source directory: However, you need to make sure to have the required libraries available on your machine. Note: you don't need all libraries if you only use parts of `xtensor-io`. `libsndfile` is required for xaudio, `OpenImageIO` for ximage and `zlib` for xnpz.

Installation of the dependencies on Linux:

```
# Ubuntu / Debian
sudo apt-get install libsndfile-dev libopenimageio-dev zlib1g-dev
# Fedora
sudo dnf install libsndfile-devel OpenImageIO-devel zlib-devel
```

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

On Windows platforms, from the source directory:

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

2.2 Basic Usage

2.2.1 Example : Reading images, audio and NPZ files

```
#include <iostream>
#include <xtensor/xbuilder.hpp>
#include <xtensor/xmath.hpp>
#include <xtensor-io/xnpz.hpp>
#include <xtensor-io/ximage.hpp>
#include <xtensor-io/xaudio.hpp>

int main()
{
    // loads png image into xarray with shape HEIGHT x WIDTH x CHANNELS
    auto arr = xt::load_image("test.png");

    // write xarray out to JPEG image
    xt::dump_image("dumptest.jpg", arr + 5);
```

(continues on next page)

(continued from previous page)

```
    // load npz file containing multiple arrays
    auto npz_map = xt::load_npz("test.npz");

    auto arr_0 = npz_map["arr_0"].cast<double>();
    auto arr_1 = npz_map["arr_1"].cast<unsigned long>();

    // open a wav file
    auto audio = xt::load_audio("files/xtensor.wav");
    std::cout << "Sampling Frequency: " << std::get<0>(audio) << std::endl;
    auto& arr = std::get<1>(audio);  // audio contents (like scipy.io.wavfile results)

    // save a sine wave sound
    int freq = 2000;
    int sampling_freq = 44100;
    double duration = 1.0;

    auto t = xt::arange(0.0, duration, 1.0 / sampling_freq);
    auto y = xt::sin(2.0 * numeric_constants<double>::PI * freq * t);

    xt::dump_audio("files/sine.wav", y, sampling_freq);

    return 0;
}
```

2.2.2 Example : Reading and writing HDF5 files

```
#include <xtensor/xio.hpp>
#include <xtensor-io/xhighfive.hpp>

int main()
{
    xt::xtensor<double,2> A = xt::ones<double>({10,5});

    xt::dump_hdf5("example.h5", "/path/to/data", A);

    A = xt::load_hdf5<xt::xtensor<double,2>>("example.h5", "/path/to/data");

    std::cout << A << std::endl;

    return 0;
}
```

2.2.3 Example : Reading and writing a file with GDAL

```
#include <xtensor/xio.hpp>
#include <xtensor-io/xgdal.hpp>

int main()
{
    // Load every band within an example image.
    // The returned order is band sequential (or [band, row, column]).
    xt::xtensor<int, 3> image = xt::load_gdal<int>("/path/to/data.ext");
    std::cout << image << std::endl;

    // Write the data to disk.
    xt::dump_gdal(image, "/path/to/output.ext");

    // Fancy options exist for both reading and writing.
    // Just as an example, we'll write a GeoTiff with deflate compression.
    xt::dump_gdal_options opt;
    opt.creation_options.emplace_back("COMPRESS=DEFLATE");
    xt::dump_gdal(image, "/path/to/output.ext", opt);
}
```

2.3 HDF5 interface

2.3.1 Basic interface

The basic interface consists of two functions:

- `xt::dump_hdf5(filename, path, data[, xt::file_mode, xt::dump_mode])`

  Dump data (e.g. a matrix) to a DataSet in a HDF5 file. The data can be: `scalar` (incl. `std::string`), `std::vector<scalar>` (incl. `std::string`), `xt::xarray<scalar>`, and `xt::xtensor<scalar,dim>`.

- `data = xt::load_hdf5<...>(filename, path)`

  Read data (e.g. a matrix) from a DataSet in a HDF5 file. The same overloads as for dump are available.

For example:

```
#include <xtensor/xio.hpp>
#include <xtensor-io/xhighfive.hpp>

int main()
{
    xt::xtensor<double,2> A = xt::ones<double>({10,5});

    xt::dump_hdf5("example.h5", "/path/to/data", A);

    A = xt::load_hdf5<xt::xtensor<double,2>>("example.h5", "/path/to/data");

    std::cout << A << std::endl;
```

(continues on next page)

(continued from previous page)

```
    return 0;
}
```

2.3.2 Advanced interface

The advanced interface provides simple free-functions that write to or read from an open `HighFive::File`. It consists of the following functions:

- `xt::dump(file, path, data[, xt::dump_mode])`

  Dump data (e.g. a matrix) to a DataSet in a HDF5 file. The data can be: `scalar` (incl. `std::string`), `std::vector<scalar>` (incl. `std::string`), `xt::xarray<scalar>`, and `xt::xtensor<scalar,dim>`.

- `xt::dump(file, path, data, {i,...})`

  Dump scalar to the index `{i,...}` of a DataSet. If the DataSet does not yet exist, an extendible DataSet of the appropriate rank and shape is created. If it does exists, the DataSet is resized if necessary.

- `data = xt::load<...>(file, path)`

  Read data (e.g. a matrix) from a DataSet in a HDF5 file. The same overloads as for dump are available.

- `data = xt::load<...>(file, path, {i,...})`

  Read scalar as index `{i,...}` of a DataSet.

For example:

```
#include <xtensor/xio.hpp>
#include <xtensor-io/xhighfive.hpp>

int main()
{
    HighFive::File file("example.h5", HighFive::File::Overwrite);

    xt::xtensor<double,2> A = xt::ones<double>({10,5});

    xt::dump(file, "/path/to/data", A);

    xt::dump(file, "/path/to/data", A, xt::dump_mode::overwrite);

    A = xt::load<xt::xtensor<double,2>>(file, "/path/to/data");

    std::cout << A << std::endl;

    return 0;
}
```

2.3.3 Compiling & dependencies

This library uses header only HighFive library and the HDF5 library. Both should be available upon compiling and linking respectively.

Manual

Compiling can then proceed through

```
g++ -std=c++14 -lhdf5 main.cpp

# manually set paths
g++ -I... -L... -std=c++14 -lhdf5 main.cpp
```

Alternatively, HDF5 provides a wrapper command that sets the paths to the HDF5 library (not to HighFive):

```
h5c++ -std=c++14 main.cpp
```

Using CMake

The following basic structure of `CMakeLists.txt` can be used:

```
cmake_minimum_required(VERSION 2.8.12)

# define a project name
project(example)

# define empty list of libraries to link
set(PROJECT_LIBS "")

# enforce the C++ standard
set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# set optimization level
set(CMAKE_BUILD_TYPE Release)

# ...

# find HighFive
find_package(HighFive REQUIRED)

# find HDF5
find_package(HDF5 REQUIRED)
include_directories(${HDF5_INCLUDE_DIRS})
set(PROJECT_LIBS ${HDF5_C_LIBRARIES})

# create executable
add_executable(${PROJECT_NAME} main.cpp)

# link libraries
target_link_libraries(${PROJECT_NAME} ${PROJECT_LIBS})
```

2.4 Stored Arrays

2.4.1 File arrays

Arrays can be stored on a file system using `xfile_array`, enabling persistence of data. This type of array is a file-backed cached `xarray`, meaning that you can use it as a normal array, and it will be flushed to the file when it is destroyed or when `flush()` is explicitly called (provided that its content has changed). Various file systems can be used, e.g. the local file system or Google Cloud Storage, and data can be stored in various formats, e.g. GZip or Blosc.

File Mode

A file array can be created using one of the three following file modes:

- `load`: the array is loaded from the file, meaning that the file must already exist, otherwise an exception is thrown.
- `init`: the array will initialize the file, meaning that its content will be flushed regardless of any pre-existing file.
- `init_on_fail`: the array is loaded from the file if it exists, otherwise the array will initialize the file. An initialization value can be used to fill the array.

The default mode is `load`.

IO handler

Stored arrays can be read and written to various file systems using an IO handler, which is a template parameter of `xfile_array`. The following IO handlers are currently supported:

- `xio_disk_handler`: for accessing the local file system.
- `xio_gcs_handler`: for accessing Google Cloud Storage.

The IO handler is itself templated by a file format.

File format

An array is stored in a file system using a file format, which usually performs some kind of compression. A file format has the ability to store the data, and optionally the shape of the array. It can also optionally be configured. The following file formats are currently supported:

- `xio_binary_config`: raw binary format.
- `xio_gzip_config`: GZip format.
- `xio_blosc_config`: Blosc format.

These formats currently only store the data, not the shape. GZip and Blosc formats are configurable, but not the binary format.

Example : on-disk file array

```
#include <xtensor-io/xfile_array.hpp>
#include <xtensor-io/xio_binary.hpp>
#include <xtensor-io/xio_disk_handler.hpp>

int main()
{
    // an on-disk file array stored in binary format
    using file_array = xt::xfile_array<double, xt::xio_disk_handler<xt::xio_binary_
↪config>>;
    // since the file doesn't alreay exist, we use the "init" file mode
    file_array a1("a1.bin", xt::xfile_mode::init);

    std::vector<size_t> shape = {2, 2};
    a1.resize(shape);

    a1(0, 1) = 1.;
    // the in-memory value is changed, but not the on-disk file yet.
    // the on-disk file will change when the array is explicitly flushed,
    // or when it is destroyed (e.g. when going out of scope)

    a1.flush();
    // now the on-disk file has changed

    // a2 points to a1's file, we use the "load" file mode
    file_array a2("a1.bin", xt::xfile_mode::load);
    // the binary format doesn't store the shape
    a2.resize(shape);

    // a1 and a2 are equal
    assert(xt::all(xt::equal(a1, a2)));

    return 0;
}
```

2.4.2 Chunked File Arrays

As for a "normal" array, a chunked array can be stored on a file system. Under the hood, it will use `xfile_array` to store the chunks. But rather than having one file array for each chunk (which could have a huge memory footprint), only a limited number of file arrays are used at the same time in a chunk pool. The container which is responsible for managing the chunk pool (i.e. map logical chunks in the array to physical chunks in the pool) is the `xchunk_store_manager`, but you should not use it directly. Instead, we provide factory functions to create a chunked file array, as shown below:

Example : on-disk chunked file array

```
#include "xtensor-io/xchunk_store_manager.hpp"
#include "xtensor-io/xio_binary.hpp"
#include "xtensor-io/xio_disk_handler.hpp"

int main()
{
    namespace fs = ghc::filesystem;

    std::vector<size_t> shape = {4, 4};
    std::vector<size_t> chunk_shape = {2, 2};
    std::string chunk_dir = "chunks1";
    fs::create_directory(chunk_dir);
    double init_value = 5.5;
    std::size_t pool_size = 2;  // a maximum of 2 chunks will be hold in memory

    auto a1 = xt::chunked_file_array<double, xt::xio_disk_handler<xt::xio_binary_config>>
↪(shape, chunk_shape, chunk_dir, init_value, pool_size);

    a1(2, 1) = 1.2;  // this assigns to chunk (1, 0) in memory
    a1(1, 2) = 3.4;  // this assigns to chunk (0, 1) in memory
    a1(0, 0) = 5.6;  // because the pool is full, this saves chunk (1, 0) to disk
                     // and assigns to chunk (0, 0) in memory
    // when a1 is destroyed, all the modified chunks are saved to disk
    // here, only chunks (0, 1) and (0, 0) are saved, since chunk (1, 0) was not changed
    // flushing can be triggered manually by calling a1.chunks().flush()
}
```

2.5 API Reference

2.5.1 NPZ files

Defined in `xtensor-io/xnpz.hpp`

inline auto **xt::load_npz**(std::string filename)
Loads a npz file.

This function returns a map. The individual arrays are not casted to a specific file format. This has to be done before they can be used as xarrays (e.g. `auto arr_0 = npz_map["arr_0"].cast<double>();`)

Parameters **filename** – The filename of the npz file

Returns returns a map with the stored arrays. Array names are the keys.

template<class **T**>
xarray<*T*> **xt::load_npz**(std::string filename, std::string search_varname)
Loads a specific array indicated by search_varname from npz file.

All other data in the npz file is ignored.

Parameters

- **filename** – The npz filename
- **search_varname** – The array name to be loaded

Returns xarray with the contents of the loaded array

template<class **E**>
void **xt::dump_npz**(std::string filename, std::string varname, const xt::xexpression<*E*> &e, bool compression = false, bool append_to_existing_file = true)

Save a xarray or xtensor to a NPZ file.

If a npz file with `filename` exists already, the new data is appended to the existing file by default. Note: currently no checking of name-collision is performed!

Parameters

- **filename** – filename to save to
- **varname** – desired name of the variable
- **e** – xexpression to save
- **compression** – true enables compression, otherwise store uncompressed (default false)
- **append_to_existing_file** – If true, appends new data to existing file

2.5.2 Images

Defined in `xtensor-io/ximage.hpp`

template<class **T** = unsigned char>
xarray<*T*> **xt::load_image**(std::string filename)

Load an image from file at filename.

Storage format is deduced from file ending.

Parameters **filename** – The path of the file to load

Returns xarray with image contents. The shape of the xarray is `HEIGHT x WIDTH x CHANNELS` of the loaded image, where `CHANNELS` are ordered in standard `R G B (A)`.

template<class **E**>
void **xt::dump_image**(std::string filename, const xexpression<*E*> &data, dump_image_options const &options = dump_image_options())

Save image to disk.

The desired image format is deduced from `filename`. Supported formats are those supported by OpenImageIO. Most common formats are supported (jpg, png, gif, bmp, tiff). The shape of the array must be `HEIGHT x WIDTH` or `HEIGHT x WIDTH x CHANNELS`.

Parameters

- **filename** – The path to the desired file
- **data** – Image data
- **options** – Pass a dump_image_options object to fine-tune image export

2.5.3 Audio files

Defined in `xtensor-io/xaudio.hpp`

template<class **T** = short>
auto **xt::load_audio**(std::string filename)

Read a WAV or OGG file This function reads a WAV file at `filename`.

Parameters **filename** – File to load.

Template Parameters **T** – select type (default: short, 16bit)

Returns tuple with (samplerate, xarray holding the data). The shape of the xarray is FRAMES x CHANNELS.

template<class **E**>
void **xt::dump_audio**(std::string filename, const xexpression<*E*> &data, int samplerate, int format = SF_FORMAT_WAV | SF_FORMAT_PCM_16)

Save an xarray in WAV or OGG sound file format Please consult the libsndfile documentation for more format flags.

Parameters

- **filename** – save under filename
- **data** – xarray/xexpression data to save
- **samplerate** – The samplerate of the data
- **format** – select format (see sndfile documentation). Default is 16 bit PCM WAV format.

2.5.4 HDF5 files

Defined in `xtensor-io/xhighfive.hpp`

template<class **T**>
inline void **xt::dump_hdf5**(const std::string &fname, const std::string &path, const *T* &data, xt::file_mode fmode = xt::file_mode::create, xt::dump_mode dmode = xt::dump_mode::create)

Write field to a new DataSet in an HDF5 file.

Parameters

- **file** – opened HighFive::File (has to be writeable)
- **path** – path of the DataSet
- **data** – the data to write
- **dmode** – DataSet-write mode (xt::dump_mode::create | xt::dump_mode::overwrite)

Returns dataset the newly created HighFive::DataSet (e.g. to add an attribute)

template<class **T**>
inline auto **xt::load_hdf5**(const std::string &fname, const std::string &path)

Load a DataSet in an open HDF5 file to an object (templated).

Parameters

- **file** – opened HighFive::File (has to be writeable)
- **path** – path of the DataSet

Returns data the read data

Warning: doxygenfunction: Unable to resolve function "xt::dump" with arguments (HighFive::File&, const std::string&, const xt::xarray<T>&, xt::dump_mode) in doxygen xml output for project "xtensor-io" from directory: ../xml. Potential matches:

```
- template<class T> HighFive::DataSet dump(HighFive::File &file, const std::string &
↪path, const T &data, const std::vector<std::size_t> &idx)
- template<class T> HighFive::DataSet dump(HighFive::File &file, const std::string &
↪path, const T &data, xt::dump_mode dmode = xt::dump_mode::create)
```

Warning: doxygenfunction: Unable to resolve function "xt::dump" with arguments (HighFive::File&, const std::string&, const xt::xtensor<T, rank>&, xt::dump_mode) in doxygen xml output for project "xtensor-io" from directory: ../xml. Potential matches:

```
- template<class T> HighFive::DataSet dump(HighFive::File &file, const std::string &
↪path, const T &data, const std::vector<std::size_t> &idx)
- template<class T> HighFive::DataSet dump(HighFive::File &file, const std::string &
↪path, const T &data, xt::dump_mode dmode = xt::dump_mode::create)
```

Warning: doxygenfunction: Unable to resolve function "xt::dump" with arguments (HighFive::File&, const std::string&, const std::vector<T>&, xt::dump_mode) in doxygen xml output for project "xtensor-io" from directory: ../xml. Potential matches:

```
- template<class T> HighFive::DataSet dump(HighFive::File &file, const std::string &
↪path, const T &data, const std::vector<std::size_t> &idx)
- template<class T> HighFive::DataSet dump(HighFive::File &file, const std::string &
↪path, const T &data, xt::dump_mode dmode = xt::dump_mode::create)
```

template<class **T**>
inline HighFive::DataSet **xt::dump**(HighFive::File &file, const std::string &path, const *T* &data, xt::dump_mode dmode = xt::dump_mode::create)

Write scalar/string to a new DataSet in an open HDF5 file.

Parameters

- **file** – opened HighFive::File (has to be writeable)
- **path** – path of the DataSet
- **data** – the data to write
- **dmode** – DataSet-write mode (xt::dump_mode::create | xt::dump_mode::overwrite)

Returns dataset the newly created HighFive::DataSet (e.g. to add an attribute)

template<class **T**>
inline HighFive::DataSet **xt::dump**(HighFive::File &file, const std::string &path, const *T* &data, const std::vector<std::size_t> &idx)

Write a scalar to a (new, extendible) DataSet in an open HDF5 file.

Parameters

- **file** – opened HighFive::File (has to be writeable)
- **path** – path of the DataSet

- **data** – the data to write
- **idx** – the indices to which to write

Returns dataset the (newly created) HighFive::DataSet (e.g. to add an attribute)

template<class **T**>
inline auto **xt::load**(const HighFive::File &file, const std::string &path, const std::vector<std::size_t> &idx)
Load entry "{i,...}" from a DataSet in an open HDF5 file to a scalar.

Parameters

- **file** – opened HighFive::File (has to be writeable)
- **idx** – the indices to load
- **path** – path of the DataSet

Returns data the read data

template<class **T**>
inline auto **xt::load**(const HighFive::File &file, const std::string &path)
Load a DataSet in an open HDF5 file to an object (templated).

Parameters

- **file** – opened HighFive::File (has to be writeable)
- **path** – path of the DataSet

Returns data the read data

inline bool **xt::extensions::exist**(const HighFive::File &file, const std::string &path)
Check if a path exists (is a Group or DataSet) in an open HDF5 file.

Parameters

- **file** – opened HighFive::File
- **path** – path of the Group/DataSet

Warning: doxygenfunction: Unable to resolve function "xt::extensions::create_group" with arguments (const HighFive::File&, const std::string&) in doxygen xml output for project "xtensor-io" from directory: ../xml. Potential matches:

```
- void create_group(HighFive::File &file, const std::string &path)
```

inline std::size_t **xt::extensions::size**(const HighFive::File &file, const std::string &path)
Get the size of an existing DataSet in an open HDF5 file.

Parameters

- **file** – opened HighFive::File
- **path** – path of the DataSet

Returns size the size of the HighFive::DataSet

inline std::vector<std::size_t> **xt::extensions::shape**(const HighFive::File &file, const std::string &path)
Get the shape of an existing DataSet in an open HDF5 file.

Parameters

- **file** – opened HighFive::File

- **path** – path of the DataSet

Returns shape the shape of the HighFive::DataSet

2.5.5 GDAL files

Defined in `xtensor-io/xgdal.hpp`

template<typename **T**>
xtensor<*T*, 3> **xt::load_gdal**(const std::string &file_path, *load_gdal_options* options = {})
Load pixels from a GDAL dataset.

Template Parameters **T** – the type of pixels to return. If it doesn't match the content of the raster, GDAL will silently cast to this data type (which may cause loss of precision).

Parameters

- **file_path** – the file to open.
- **options** – options to use when loading.

Returns pixels; band sequential by default, but index order is controlled by options.

template<typename **T**>
xtensor<*T*, 3> **xt::load_gdal**(GDALDatasetH dataset, *load_gdal_options* options = {})
Load pixels from an opened GDAL dataset.

Template Parameters **T** – in type of pixels to return. If this doesn't match the content of the dataset, GDAL will silently cast to this data type (which may cause loss of precision).

Parameters

- **dataset** – an opened GDAL dataset.
- **options** – options to used when loading.

Returns pixels; band sequential by default, but index order is controlled by options.

struct **xt::load_gdal_options**
Options for loading a GDAL dataset.

Public Functions

inline **load_gdal_options**()
Load every band and return band-sequential memory (index order = [band, row, column]).

Public Members

layout **interleave**
The desired layout of the returned tensor - defaults to band sequential([band, row, col]).

Arbitrary index order is supported, like band interleaved by pixel ([row, col, band]).

std::vector<int> **bands_to_load**
The indices of bands to read from the dataset.

All bands are read if empty.

Remark Per GDAL convention, these indices start at *one* (not zero).

std::function<void(const std::string&)> **error_handler**
The error handler used to report errors (like file missing or a read fails).

By default, a std::runtime_error is thrown when an error is encountered

template<typename **T**>
GDALDatasetH **xt::dump_gdal**(const xexpression<*T*> &e, const std::string &path, *dump_gdal_options* options = {})
Dump a 2D or 3D xexpression to a GDALDataset.

Parameters

- **e** – data to dump; must evaluate to a 2D or 3D shape.
- **path** – where to save the data.
- **options** – options to use when dumping.

Returns nullptr by default, or, an open dataset if dump_gdal_options.return_opened_dataset (user must close).

struct **dump_gdal_options**
Options for dumping a GDAL dataset.

2.5.6 xfile_array

Defined in `xtensor-io/xfile_array.hpp`

template<class **E**, class **IOH**>

class **xfile_array_container** : public xaccessible<*xfile_array_container*<*E*, *IOH*>>, public xiterable<*xfile_array_container*<*E*, *IOH*>>, public xcontainer_semantic<*xfile_array_container*<*E*, *IOH*>>
Dense multidimensional file-backed cached container with tensor semantic.

The *xfile_array_container* class implements a dense multidimensional container with tensor semantic, stored on a file system. It acts as a file-backed cached container, allowing for data persistence.

See *xchunk_store_manager*

tparam E The type of the container holding the elements

tparam IOH The type of the IO handler (e.g. xio_disk_handler)

using **xt::xfile_array** = *xfile_array_container*<xarray<T, L, A, SA>, IOH>

2.5.7 xchunk_store_manager

Defined in `xtensor-io/xchunk_store_manager.hpp`

template<class **EC**, class **IP** = xindex_path>

class **xchunk_store_manager** : public xaccessible<*xchunk_store_manager*<*EC*, *IP*>>, public xiterable<*xchunk_store_manager*<*EC*, *IP*>>
Multidimensional chunk container and manager.

The *xchunk_store_manager* class implements a multidimensional chunk container. Chunks are managed in a pool, allowing for a limited number of chunks that can simultaneously be hold in memory, by swapping chunks when the pool is full. Should not be used directly, instead use chunked_file_array factory functions.

tparam EC The type of a chunk (e.g. xfile_array)

tparam IP The type of the index-to-path transformer (default: xindex_path)

2.5.8 chunked_file_array

Defined in `xtensor-io/xchunk_store_manager.hpp`

Warning: doxygenfunction: Unable to resolve function "xt::chunked_file_array" with arguments None in doxygen xml output for project "xtensor-io" from directory: ../xml. Potential matches:

```
- template<class IOH, layout_type L = XTENSOR_DEFAULT_LAYOUT, class IP = xindex_path,␣
↪class E, class S> xchunked_array<xchunk_store_manager<xfile_array<typename E::value_
↪type, IOH, L>, IP>> chunked_file_array(const xexpression<E> &e, S &&chunk_shape,␣
↪const std::string &path, std::size_t pool_size = 1, layout_type chunk_memory_layout␣
↪= XTENSOR_DEFAULT_LAYOUT)
- template<class IOH, layout_type L = XTENSOR_DEFAULT_LAYOUT, class IP = xindex_path,␣
↪class E> xchunked_array<xchunk_store_manager<xfile_array<typename E::value_type,␣
↪IOH, L>, IP>> chunked_file_array(const xexpression<E> &e, const std::string &path,␣
↪std::size_t pool_size = 1, layout_type chunk_memory_layout = XTENSOR_DEFAULT_LAYOUT)
- template<class T, class IOH, layout_type L = XTENSOR_DEFAULT_LAYOUT, class IP =␣
↪xindex_path, class S> xchunked_array<xchunk_store_manager<xfile_array<T, IOH, L>,␣
↪IP>> chunked_file_array(S &&shape, S &&chunk_shape, const std::string &path, const␣
↪T &init_value, std::size_t pool_size = 1, layout_type chunk_memory_layout = XTENSOR_
↪DEFAULT_LAYOUT)
- template<class T, class IOH, layout_type L = XTENSOR_DEFAULT_LAYOUT, class IP =␣
↪xindex_path, class S> xchunked_array<xchunk_store_manager<xfile_array<T, IOH, L>,␣
↪IP>> chunked_file_array(S &&shape, S &&chunk_shape, const std::string &path,␣
↪std::size_t pool_size = 1, layout_type chunk_memory_layout = XTENSOR_DEFAULT_LAYOUT)
- template<class T, class IOH, layout_type L = XTENSOR_DEFAULT_LAYOUT, class IP =␣
↪xindex_path, class S> xchunked_array<xchunk_store_manager<xfile_array<T, IOH, L>,␣
↪IP>> chunked_file_array(std::initializer_list<S> shape, std::initializer_list<S>␣
↪chunk_shape, const std::string &path, const T &init_value, std::size_t pool_size =␣
↪1, layout_type chunk_memory_layout = XTENSOR_DEFAULT_LAYOUT)
- template<class T, class IOH, layout_type L = XTENSOR_DEFAULT_LAYOUT, class IP =␣
↪xindex_path, class S> xchunked_array<xchunk_store_manager<xfile_array<T, IOH, L>,␣
↪IP>> chunked_file_array(std::initializer_list<S> shape, std::initializer_list<S>␣
↪chunk_shape, const std::string &path, std::size_t pool_size = 1, layout_type chunk_
↪memory_layout = XTENSOR_DEFAULT_LAYOUT)
```

2.6 Releasing xtensor-io

2.6.1 Releasing a new version

From the master branch of xtensor-io

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

2.6.2 Updating the conda-forge recipe

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

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

2.6.3 Updating the stable branch

Once the conda-forge package has been updated, update the `stable` branch to the newly added tag.

INDEX

X