carma

Release 0.6.0

Aug 16, 2022

Contents

CARMA is a header only library providing conversions between Numpy arrays and Armadillo matrices.

CHAPTER 1

Introduction

CARMA provides fast bidirectional conversions between Numpy arrays and Armadillo matrices, vectors and cubes, much like RcppArmadillo does for R and Armadillo.

The library extends the impressive Pybind11 library with support for Armadillo. For details on Pybind11 and Armadillo refer to their respective documentation [1], [2].

1.1 Installation

CARMA is a header only library that relies on two other header only libraries, Armadillo and Pybind11.

CARMA can be installed using:

```
mkdir build
cd build
# optionally with -DCMAKE_INSTALL_PREFIX:PATH=
cmake -DCARMA_INSTALL_LIB=ON ..
cmake --build . --config Release --target install
```

You can than include it in a project using:

```
FIND_PACKAGE(carma CONFIG REQUIRED)
TARGET_LINK_LIBRARIES(<your_target> PRIVATE carma::carma)
```

The `REQUIRED` state is propagated to the dependencies of CARMA.

CARMA and Armadillo can then be included using:

```
#include <carma>
#include <armadillo>
```

CARMA can also be used without installation see *Build Configuration*. CARMA provides a number of configurations that can be set in the `carma_config.cmake` file at the root of the directory or passed to CMake, see *Configuration* for details.

1.2 Requirements

CARMA >= v0.5 requires a compiler with support for C++14 and supports:

- Python 3.6 – 3.9
- Numpy >= 1.14
- Pybind11 >= v2.6.0
- Armadillo >= 10.5.2

CARMA makes use of Armadillo's `ARMA_ALIEN_MEM_ALLOC` and `ARMA_ALIEN_MEM_FREE` functionality introduced in version 10.5.2 to use Numpy's (de)allocator.

1.3 OLS example

A brief example on how conversion can be achieved. We convert the Numpy arrays X and y in using automatic casting (type-caster) and using manual conversion on the way out.

```
#include <carma>
#include <armadillo>
#include <pybind11/pybind11.h>
#include <pybind11/numpy.h>
#include <pybind11/pytypes.h>

py::tuple ols(arma::mat& X, arma::colvec& y) {
    // We borrow the data underlying the numpy arrays
    int n = X.n_rows, k = X.n_cols;

    arma::colvec coeffs = arma::solve(X, y);
    arma::colvec resid = y - X * coeffs;

    double sig2 = arma::as_scalar(arma::trans(resid) * resid / (n-k));
    arma::colvec std_errs = arma::sqrt(sig2 * arma::diagvec(␣
↪arma::inv(arma::trans(X)*X)) );

    // We take ownership of the memory from the armadillo objects and
    // return to python as a tuple containing two Numpy arrays.
    return py::make_tuple(
        carma::col_to_arr(coeffs),
        carma::col_to_arr(std_errs)
    );
}

// adapted from https://gallery.rcpp.org/articles/fast-linear-model-with-armadillo/
```

1.4 Considerations

In order to achieve fast conversions the default behaviour is avoid copying both from and to Numpy whenever possible and reasonable. This allows very low overhead conversions but it impacts memory safety and requires user vigilance.

If you intend to return the memory of the input array back as another array, you must make sure to either copy or steal the memory on the conversion in or copy the memory out. If you don't the memory will be aliased by the two Numpy arrays and bad things will happen.

A second consideration is memory layout. Armadillo is optimised for column-major (Fortran order) memory whereas Numpy defaults to row-major (C order). The default behaviour is to automatically convert, read copy, C-order arrays to F-order arrays upon conversion to Armadillo. Users should note that the library will not convert back to C-order when returning.

For details see the *Memory Management* section.

CHAPTER 2

About

This project was created by Ralph Urlus. Significant improvements to the project have been contributed by Pascal H.

2.1 License

carma is provided under a Apache 2.0 license that can be found in the LICENSE file. By using, distributing, or contributing to this project, you agree to the terms and conditions of this license.

CHAPTER 3

First steps

CARMA relies on Pybind11 for the generation of the bindings and casting of the arguments from Python to C++. Make sure you are familiar with Pybind11 before continuing on.

You can embed CARMA in a Pybind11 project using the CMake commands:

```
ADD_SUBDIRECTORY(extern/carma)
TARGET_LINK_LIBRARIES(<your_target> PRIVATE carma)
```

It is advised to use `ADD_SUBDIRECTORY` to incorporate CARMA as this provides an interface target, `carma`, that can be linked to your target. Using this target prevents an include order dependency, see CARMA target for details.

Warning: if you are not using CARMA's cmake target you have to ensure that you include CARMA before Armadillo. Not doing so results in a compile error.

CARMA can provide Armadillo and or Pybind11 at compile time if desired, see *Armadillo* and *Pybind11* details.

See Pybind11's CMake build system documentation or CARMA's examples for a start.

3.1 Build examples

The tests and examples can be compiled using CMake. CMake can be installed with `pip install cmake`, your package manager or directly from cmake.

```
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=. -DCARMA_BUILD_EXAMPLES=true␣
↪ -DCARMA_BUILD_TESTS=true .. && make install
```

To run the tests you need to install *pytest*:

```
pip install pytest
```

and run:

```
ctest
```

To install *carma*, you have to define

```
-DCMAKE_INSTALL_PREFIX=/installation/path/directory
```

(default value is `/usr/local`)

The installation directory contains

```
include                    # carma headers
tests                      # carma tests with python module (if enabled using -DBUILD_
↪TESTS=on)
examples                   # carma python examples with python module (if enabled using␣
↪-DBUILD_EXAMPLES=on)
```

See section *Examples* for an overview of the conversion approaches.

CHAPTER 4

Design Patterns

CARMA was designed with three patterns in mind: borrow, transfer ownership and view.

4.1 Borrow

You can borrow the underlying memory of a Numpy array using the `arr_to_*(py::array_t<T>, copy=false)`. The Armadillo object should not be returned without a copy out. Use this when you want to modify the memory in-place. If the array is not well behaved, see *Well behaved*, the data is copied to well-behaved memory and swapped in place of the input array. If `copy=true` this is equivalent to the copy approach below.

Note: the size of the Armadillo object is not allowed change when you borrow, i.e. `strict=true`.

4.2 Transfer ownership

You can transfer ownership to Armadillo using steal or copy. After transferring ownership of the memory, Armadillo behaves as if it has allocated the memory itself, hence it will also free the memory upon destruction using Numpy's deallocator.

4.2.1 Steal

If you want to take ownership of the underlying memory but don't want to copy the data you can steal the array. The Armadillo object can be safely returned out without a copy. There are multiple compile time definitions on how the memory is stolen, see *Configuration* for details. If the memory of the array is not well-behaved a copy of the memory is stolen.

Note: the size of the Armadillo object is allowed change after stealing, `strict=false`.

4.2.2 Copy

If you want to give Armadillo full control of underlying memory but also want to keep Numpy as owner you should copy. The Armadillo object can be safely returned out without a copy. If the memory of the array is not well-behaved a copy of the memory is used instead.

Note: the size of the Armadillo object is allowed change after copying, `strict=false`.

4.3 View

If you want to have a read-only view on the underlying memory you can use `arr_to_*_view`. If the underlying memory is not well-behaved, excluding writeable, it will be copied.

CHAPTER 5

Examples

On a high level CARMA provides two ways to work with Numpy arrays and Armadillo, see the *Function specifications* section for details about the available functions and the examples directory for runnable examples.

5.1 Manual conversion

The easiest way to use CARMA is manual conversion, it gives you the most control over when to copy or not. You pass a Numpy array as an input and/or as the return type and call the respective conversion function.

Warning: Carma will avoid copying by default so make sure not to return the memory of the input array without copying. If you don't copy out, the memory is aliased by both the input and output arrays which will cause a segfault.

5.1.1 Borrow

```
#include <carma>
#include <armadillo>
#include <pybind11/pybind11.h>
#include <pybind11/numpy.h>

py::array_t<double> manual_example(py::array_t<double> & arr) {
    // convert to armadillo matrix without copying.
    // Note the size of the matrix cannot be changed when borrowing
    arma::Mat<double> mat = carma::arr_to_mat<double>(arr);

    // normally you do something useful here ...
    arma::Mat<double> result = arma::Mat<double>(arr.shape(0), arr.shape(1),␣
↪arma::fill::randu);
```

(continues on next page)

(continued from previous page)

```
    // convert to Numpy array and copy out
    return carma::mat_to_arr(result, true);
}
```

5.1.2 Transfer ownership

If you want to transfer ownership to the C++ side you can use:

```
#include <carma>
#include <armadillo>
#include <pybind11/pybind11.h>
#include <pybind11/numpy.h>

py::array_t<double> steal_array(py::array_t<double> & arr) {
    // convert to armadillo matrix
    arma::Mat<double> mat = carma::arr_to_mat<double>(std::move(arr));
    // armadillo now owns and manages the lifetime of the memory
    // if you want to return you don't need to copy out
    return mat_to_arr(mat);
}

py::array_t<double> copy_array(py::array_t<double> & arr) {
    // convert to armadillo matrix
    arma::Mat<double> mat = carma::arr_to_mat<double>(arr, true);
    // armadillo now owns and manages the lifetime of the memory
    // if you want to return you don't need to copy out
    return mat_to_arr(mat);
}

py::array_t<double> copy_const_array(const py::array_t<double> & arr) {
    // convert to armadillo matrix
    arma::Mat<double> mat = carma::arr_to_mat<double>(arr);
    // armadillo now owns and manages the lifetime of the memory
    // if you want to return you don't need to copy out
    return mat_to_arr(mat);
}
```

5.2 Automatic conversion

For automatic conversion you specify the desired Armadillo type for either or both the return type and the function parameter. When calling the function from Python, Pybind11 will call CARMA's type caster when a Numpy array is passed or returned, see *Return policies* for details.

> **Warning:** Make sure to include *carma* in every compilation unit that makes use of the type caster, not including it results in undefined behaviour.

```
#include <carma>
#include <armadillo>
#include <pybind11/pybind11.h>
#include <pybind11/numpy.h>
```

(continues on next page)

(continued from previous page)

```
arma::Mat<double> automatic_example(arma::Mat<double> & mat) {
    // normally you do something useful here with mat ...
    arma::Mat<double> rand = arma::Mat<double>(mat.n_rows, mat.n_cols,␣
↪arma::fill::randu);

    arma::Mat<double> result = mat + rand;
    // type caster will take care of casting `result` to a Numpy array.
    return result;
}
```

Warning: The automatic conversion will **not** copy the Numpy array's memory when converting to Armadillo objects. When converting back to Numpy arrays the memory will **not** be copied out by default. You shoud specify `return_value_policy::copy` if you want to return the input array.

5.3 ArrayStore

There are use-cases where you would want to keep the data in C++ and only return when requested. For example, you write an Ordinary Least Squares (OLS) class and you want to store the residuals, covariance matrix, ... in C++ for when additional tests need to be run on the values without converting back and forth.

ArrayStore is a convenience class that provides conversion methods back and forth. It is intended to be used as an attribute as below:

Warning: The ArrayStore owns the data, the returned numpy arrays are views that are tied to the lifetime of ArrayStore.

```
#include <armadillo>
#include <carma>
#include <pybind11/pybind11.h>
#include <pybind11/numpy.h>

class ExampleClass {
    private:
        carma::ArrayStore<arma::Mat<double>> _x;
        carma::ArrayStore<arma::Mat<double>> _y;

    public:
        ExampleClass(py::array_t<double> & x, py::array_t<double> & y) :
        // copy the array and store it as an Armadillo matrix
        _x{carma::ArrayStore<arma::Mat<double>>(x, true)},
        // steal the array and store it as an Armadillo matrix
        _y{carma::ArrayStore<arma::Mat<double>>(y, false)},

        py::array_t<double> member_func() {
            // normallly you would something useful here
            _x.mat += _y.mat;
            // return mutable view off arma matrix
            return _x.get_view(true);
        }
```

(continues on next page)

(continued from previous page)

```
};

void bind_exampleclass(py::module &m) {
    py::class_<ExampleClass>(m, "ExampleClass")
        .def(py::init<py::array_t<double> &, py::array_t<double> &>(), R"pbdoc(
            Initialise ExampleClass.

            Parameters
            ----------
            arr1: np.ndarray
                array to be stored in armadillo matrix
            arr2: np.ndarray
                array to be stored in armadillo matrix
        )pbdoc")
        .def("member_func", &ExampleClass::member_func, R"pbdoc(
            Compute ....
        )pbdoc");
}
```

5.4 Ordinary Least Squares

Combining the above approaches to compute the Ordinary Least Squares:

```
#include <carma>
#include <armadillo>
#include <pybind11/pybind11.h>
#include <pybind11/numpy.h>
#include <pybind11/pytypes.h>

py::tuple ols(arma::mat& X, arma::colvec& y) {
    // We borrow the data underlying the numpy arrays
    int n = X.n_rows, k = X.n_cols;

    arma::colvec coeffs = arma::solve(X, y);
    arma::colvec resid = y - X * coeffs;

    double sig2 = arma::as_scalar(arma::trans(resid) * resid / (n-k));
    arma::colvec std_errs = arma::sqrt(sig2 * arma::diagvec(␣
↪arma::inv(arma::trans(X)*X)) );

    // We take ownership of the memory from the armadillo objects and
    // return to python as a tuple containing two Numpy arrays.
    return py::make_tuple(
        carma::col_to_arr(coeffs),
        carma::col_to_arr(std_errs)
    );
}

// adapted from https://gallery.rcpp.org/articles/fast-linear-model-with-armadillo/
```

Which can be called using:

```
y = np.linspace(1, 100, num=100) + np.random.normal(0, 0.5, 100)
X = np.hstack((np.ones(100)[:, None], np.arange(1, 101)[:, None]))
```

(continues on next page)

(continued from previous page)

```
coeff, std_err = carma.ols(X, y)
```

The repository contains tests, examples and CMake build instructions that can be used as an reference.

CHAPTER 6

Memory Order

A significant difference between Numpy arrays and Armadillo matrices is the memory order. Armadillo is optimised for column-major (Fortran order) memory whereas Numpy defaults to row-major (C order).

The default behaviour is to automatically convert matrices and cubes, read copy, C-order arrays to F-order arrays upon conversion to Armadillo. This behaviour can be disabled at compile time, see *Configuration*.

This conversion can be avoided by ensuring all data generated by Numpy is done with *order='F'* or converted using:

```
np.asarray(cdata, type=np.float64, order='F')
```

Note: Carma will not convert F-order memory back to C-order when returning, this has consequences for matrices and cubes.

CHAPTER 7

Memory (de)allocator

Armadillo will not always hold the same memory for the entirety of a object's lifetime, e.g. you add columns to a matrix. To be able to transfer ownership we have to be sure that the memory will be deallocated by the function matching the allocator. In the case of CARMA this means Armadillo must use the Numpy allocator (`PyDataMem_NEW`) and deallocator (`PyDataMem_FREE`). Numpy's allocator is compatible with Armadillo as both call/wrap `malloc/calloc`.

CHAPTER 8

Memory Safety

In order to provide fast conversions the default behaviour of CARMA is to avoid copying where possible. However, this requires users to be vigilant to avoid aliasing of memory. This section describes when copies are made by default.

Note when borrowing the memory of an array the Armadillo object cannot be changed in size, this will trigger an exception. If the Armadillo object will change in size you should use a copy or steal.

Warning: Armadillo objects are **not** copied out by default. You should not return a object without copying when the Armadillo object points to borrowed memory as this will cause double frees and segfaults.

CHAPTER 9

Well behaved

Armadillo expects the memory to be aligned and Fortran order (column contiguous). To ensure this we check if the array is **well-behaved** which is **not** true when one of the following is true:

1. memory is not aligned
2. memory is not writable
3. array does not own the memory (OWNDATA=false)
4. array has `ndim >= 2` and memory is not F-contiguous

Note: Conditions 3 and 4 can be disabled, independently, with compile time definitions, see the section *Configuration*.

Based on the type of conversion, steal, borrow or view, the effect on the input array differs.

9.1 Borrow

The memory of the array is copied to well-behaved memory and swapped in the place of the original memory. This is done to ensure the design that pattern that borrowed arrays don't require a return to reflect the changes.

9.2 Stealing

If an array will be stolen but it's not well-behaved the memory is copied to well-behaved memory and ownership of the new memory is transferred to Armadillo. Note this steals a reference to the input array, which will be destructed by Numpy/Python.

9.3 Copy

If an array will be copied it is automatically copied to well-behaved memory and ownership of the new memory is transferred to Armadillo. This has no effect on the input array.

9.4 View

For view's the definition of well-behaved ignores the writeable flag as the Armadillo object will be read-only.

CHAPTER 10

Configuration

CARMA offers a number of compile-time settings that determine when the input array's memory is not considered well-behaved and how to handle stealing the memory of an incoming array

You can find the configuration file, `carma_config.cmake`, in the root of the repository where you can enable or disable the various settings.

10.1 Array conditions

As detailed in the section *Well behaved* input arrays are copied when they don't meet the criteria. Two of these criteria can be turned off.

```
OPTION(ENABLE_CARMA_DONT_REQUIRE_OWNDATA "Enable CARMA_DONT_REQUIRE_OWNDATA" ON)
```

Warning: When using this option you have make sure that the memory of the input array can be safely freed using Numpy's deallocator (`PyDataMem_FREE`) as this is used to manage the memory after transfering ownership to Armadillo.

Do **not** copy arrays if the data is not owned by Numpy, default behaviour is to copy when OWNDATA is False. Note this will lead to `segfaults` when the array's data is stolen or otherwise aliased by Armadillo. Is useful when you want to pass back and forth arrays previously converted by CARMA.

```
OPTION(ENABLE_CARMA_DONT_REQUIRE_F_CONTIGUOUS "Enable CARMA_DONT_REQUIRE_F_CONTIGUOUS
↪" ON)
```

Do **not** copy C-contiguous arrays, default behaviour is to copy C-contiguous arrays to Fortran order as this is what Armadillo is optimised for. Note that on the conversion back, it is assumed that the memory of the Armadillo object has Fortran order layout.

10.2 Stealing

The default behaviour is to only set `OWNDATA=false` when stealing the data of an array. This is fast and leaves the array usable. However, two additional options exists that make it clearer when an array has been stolen.

```
OPTION(ENABLE_CARMA_SOFT_STEAL "Enable CARMA_SOFT_STEAL" ON)
```

When stealing the data of an array replace it with an array containing a single NaN. This is a safer but slower option compared to HARD_STEAL but easier to notice than the default.

```
OPTION(ENABLE_CARMA_HARD_STEAL "Enable CARMA_HARD_STEAL" ON)
```

When stealing the data of an array CARMA sets a `nullptr` in place of the pointer to the memory. Note this will cause a `segfault` when accessing the original array's data. However, this ensures stolen arrays are not accidentally used later on.

10.3 Debugging

```
OPTION(ENABLE_CARMA_EXTRA_DEBUG "Enable CARMA_EXTRA_DEBUG" ON)
OPTION(ENABLE_ARMA_EXTRA_DEBUG "Enable ARMA_EXTRA_DEBUG" ON)
```

Turn this setting on if you want to debug conversions. Debug prints are generated that specify when arrays are not well-behaved, stolen or swapped in place. Note that the additional debugging information from Armadillo can be enabled using the second setting.

10.4 Release

```
OPTION(ENABLE_ARMA_NO_DEBUG "Enable ENABLE_ARMA_NO_DEBUG" OFF)
```

This option sets ARMA_NO_DEBUG as part of the release flags.

"Disable all run-time checks, such as bounds checking. This will result in faster code, but you first need to make sure that your code runs correctly! We strongly recommend to have the run-time checks enabled during development, as this greatly aids in finding mistakes in your code, and hence speeds up development. We recommend that run-time checks be disabled only for the shipped version of your program (i.e. final release build)." – Armadillo documentation

10.5 Developer settings

Two settings exists to faciliate development of CARMA:

```
-DCARMA_DEV_MODE=ON
```

This enables:

- `CARMA_BUILD_TESTS=ON`
- `CARMA_DEV_TARGET=ON`
- `CMAKE_EXPORT_COMPILE_COMMANDS=1`
- `CMAKE_INSTALL_PREFIX ${PROJECT_SOURCE_DIR}/build)`

```
-DCARMA_DEV_DEBUG_MODE=ON
```

Turns on CARMA_DEV_MODE and ENABLE_CARMA_EXTRA_DEBUG.

CHAPTER 11

Build Configuration

CARMA v0.5 requires a compiler with support for C++14 and supports:

- Python 3.6 – 3.9
- Numpy >= 1.14
- Pybind11 >= v2.6.0
- Armadillo >= 10.5.2

11.1 CMake build

CARMA provides a CMake configuration that can be used to integrate into existing builds. You can either use it directly or install it first. To edit the configuration please see

It is advised to use the `carma::carma` interface target that can be linked to your target. This target pre-compiles the `cnalloc.h` header containing wrappers around Numpy's (de)allocator that are then picked up by Armadillo. By pre-compiling the header we can ensure that the `ARMA_ALIEN_MEM_ALLOC` and `ARMA_ALIEN_MEM_FREE` definitions exist when including Armadillo regardless of the include order.

Warning: if you are not using CARMA's cmake target you have to ensure that you include CARMA before Armadillo. Not doing so results in a compile error.

11.1.1 Installation

Make sure to change the configuration if desired before installing CARMA.

CARMA can be installed using:

```
mkdir build
cd build
# optionally with -DCMAKE_INSTALL_PREFIX:PATH=
cmake -DCARMA_INSTALL_LIB=ON ..
cmake --build . --config Release --target install
```

You can than include it in a project using:

```
FIND_PACKAGE(carma CONFIG REQUIRED)
TARGET_LINK_LIBRARIES(<your_target> PRIVATE carma::carma)
```

The `REQUIRED` state is propagated to the dependencies of CARMA.

Variables

The `FIND_PACKAGE` call sets the following variables

- `carma_FOUND` – true if CARMA was found
- `carma_INCLUDE_DIR` – the path to CARMA's include directory
- `carma_INCLUDE_DIRS` – the paths to CARMA's include directory and the paths to the include directories of the dependencies.

Components

When including carma using `FIND_PACKAGE` two targets are created:

- `carma::carma`

`carma::carma` has been linked with Python, Numpy, Pybind11 and Armadillo and pre-compiles the `cnalloc.h` header which means that there is no required order to includes or carma and armadillo. If you only want this component you can use `FIND_PACKAGE(carma CONFIG REQUIRED COMPONENTS carma)`

- `carma::headers`

If you want to have a header-only target that is not linked with the dependencies and does not pre-compile `cnalloc.h`. You must than make sure to link it to it's dependencies and make sure to always include carma before armadillo. If you only want this component you can use `FIND_PACKAGE(carma CONFIG REQUIRED COMPONENTS headers)`

11.1.2 Subdirectory

Alternatively, you can use CARMA without installing it by using:

```
ADD_SUBDIRECTORY(extern/carma)
TARGET_LINK_LIBRARIES(<your_target> PRIVATE carma::carma)
```

The same targets and conditions as for the installation hold, however, this build will obtain Armadillo and Pybind11 if they have not been provided.

11.2 Armadillo

Users can provide a specific Armadillo version by making sure the target `armadillo` is set before including CARMA or by setting:

```
-DARMADILLO_ROOT_DIR=/path/to/armadillo/root/directory
```

When using the subdirectory build, if neither is set, CARMA will provide the `armadillo` target at build time and store a clone of armadillo in `carma/extern/armadillo-code`. The Armadillo version, by default `10.5.2`, can be set using:

```
-DUSE_ARMA_VERSION=10.5.x
```

11.3 Pybind11

Users can provide a specific Pybind11 version by making sure the target `pybind11` is set before including CARMA or by setting:

```
-DPYBIND11_ROOT_DIR=/path/to/pybind11/root/directory
```

When using the subdirectory build, if neither is set, CARMA will provide the `pybind11` target at build time and store a clone in `carma/extern/pybind11`. The Pybind11 version, by default `v2.6.2` can be set using:

```
-DUSE_PYBIND11_VERSION=v2.6.2
```

11.4 Python

CARMA needs to link against Python's and Numpy's headers and uses CMake's `FIND_PYTHON` to locate them. `FIND_PYTHON` doesn't always find the right Python, e.g. when using `pyenv`. When this happens you can set `Python3_EXECUTABLE`, which is then also passed on to Pybind11 to ensure the same Python versions are found.

```
-DPython3_EXECUTABLE=$(which python3)
# or
-DPython3_EXECUTABLE=$(python3 -c 'import sys; print(sys.executable)')
```

CHAPTER 12

Manual conversion

CARMA provides a set of functions for manual conversion of Numpy arrays and Armadillo matrices. Manual conversion should be used when fine grained control of memory is required.

12.1 Numpy to Armadillo

Functions to convert Numpy arrays to Armadillo matrices, vectors or cubes

12.1.1 Matrix

`arma::Mat<T> arr_to_mat(py::array<T>& arr, bool copy=false)`

Convert Numpy array to Armadillo matrix. When borrowing the the array, copy=false, if it is not well-behaved we perform a in-place swap to a well behaved array

If the array is 1D we create a column oriented matrix (N, 1)

Parameters

- **`arr`** – numpy array to be converted
- **`copy`** – copy the memory of the array, default is false

Raises

- **`runtime_error`** – if *n_dims* > 2 or memory is not initialised (nullptr)
- **`runtime_error`** – if copy is false and the array is not well-behaved and not writable as this prevents an inplace-swap.

`arma::Mat<T> arr_to_mat(const py::array<T>& arr)`

Copy the memory of the Numpy array in the conversion to Armadillo matrix.

If the array is 1D we create a column oriented matrix (N, 1)

Parameters **`arr`** – numpy array to be converted

Raises **runtime_error** – if *n_dims* > 2 or memory is not initialised (nullptr)

arma::Mat<T> arr_to_mat(py::array<T>&& arr)
Steal the memory of the Numpy array in the conversion to Armadillo matrix. If the memory is not well-behaved we steal a copy of the array

If the array is 1D we create a column oriented matrix (N, 1)

Parameters **arr** – numpy array to be converted

Raises **runtime_error** – if *n_dims* > 2 or memory is not initialised (nullptr)

const arma::Mat<T> arr_to_mat_view(const py::array<T>& arr)
Copy the memory of the Numpy array in the conversion to Armadillo matrix if not well behaved otherwise borrow.

If the array is 1D we create a column oriented matrix (N, 1)

Parameters **arr** – numpy array to be converted

Raises **runtime_error** – if *n_dims* > 2 or memory is not initialised (nullptr)

12.1.2 Vector

arma::Col<T> arr_to_col(py::array<T>& arr, bool copy=false)
Convert Numpy array to Armadillo column. When borrowing the the array, copy=false, if it is not well-behaved we perform a in-place swap to a well behaved array

Parameters

- **arr** – numpy array to be converted
- **copy** – copy the memory of the array, default is false

Raises

- **runtime_error** – if *n_cols* > 1 or memory is not initialised (nullptr)
- **runtime_error** – if copy is false and the array is not well-behaved and not writable as this prevents an inplace-swap.

arma::Col<T> arr_to_col(const py::array<T>& arr)
Copy the memory of the Numpy array in the conversion to Armadillo column.

Parameters **arr** – numpy array to be converted

Raises **runtime_error** – if *n_cols* > 1 or memory is not initialised (nullptr)

arma::Col<T> arr_to_col(py::array<T>&& arr)
Steal the memory of the Numpy array in the conversion to Armadillo column. If the memory is not well-behaved we steal a copy of the array

Parameters **arr** – numpy array to be converted

Raises **runtime_error** – if *n_cols* > 1 or memory is not initialised (nullptr)

const arma::Col<T> arr_to_col_view(const py::array<T>& arr)
Create a read-only view on the array as a Armadillo Col. Copy the memory of the Numpy array in the conversion to Armadillo column if not well_behaved otherwise borrow the array.

Parameters **arr** – numpy array to be converted

Raises **runtime_error** – if *n_cols* > 1 or memory is not initialised (nullptr)

arma::Row<T> arr_to_row(py::array<T>& arr, bool copy=false)
Convert Numpy array to Armadillo row. When borrowing the the array, copy=false, if it is not well-behaved we perform a in-place swap to a well behaved array

Parameters

- **arr** – numpy array to be converted
- **copy** – copy the memory of the array, default is false

Raises

- **runtime_error** – if *n_rows* > 1 or memory is not initialised (nullptr)
- **runtime_error** – if copy is false and the array is not well-behaved and not writable as this prevents an inplace-swap.

arma::Row<T> arr_to_col(const py::array<T>& arr)
Copy the memory of the Numpy array in the conversion to Armadillo row.

Parameters **arr** – numpy array to be converted

Raises **runtime_error** – if *n_rows* > 1 or memory is not initialised (nullptr)

arma::Row<T> arr_to_col(py::array<T>&& arr)
Steal the memory of the Numpy array in the conversion to Armadillo row. If the memory is not well-behaved we steal a copy of the array

Parameters **arr** – numpy array to be converted

Raises **runtime_error** – if *n_cols* > 1 or memory is not initialised (nullptr)

const arma::Row<T> arr_to_col_view(const py::array<T>& arr)
Create a read-only view on the array as a Armadillo Col. Copy the memory of the Numpy array if not well_behaved otherwise borrow the array.

Parameters **arr** – numpy array to be converted

Raises **runtime_error** – if *n_rows* > 1 or memory is not initialised (nullptr)

12.1.3 Cube

arma::Cube<T> arr_to_cube(py::array<T>& arr, bool copy=false)
Convert Numpy array to Armadillo Cube. When borrowing the the array, copy=false, if it is not well-behaved we perform a in-place swap to a well behaved array

Parameters

- **arr** – numpy array to be converted
- **copy** – copy the memory of the array, default is false

Raises

- **runtime_error** – if *n_dims* < 3 or memory is not initialised (nullptr)
- **runtime_error** – if copy is false and the array is not well-behaved and not writable as this prevents an inplace-swap.

arma::Cube<T> arr_to_cube(const py::array<T>& arr)
Copy the memory of the Numpy array in the conversion to Armadillo Cube.

Parameters **arr** – numpy array to be converted

Raises **runtime_error** – if *n_dims* < 3 or memory is not initialised (nullptr)

arma::Cube<T> arr_to_cube(py::array<T>&& arr)
Steal the memory of the Numpy array in the conversion to Armadillo Cube. If the memory is not well-behaved we steal a copy of the array

Parameters **arr** – numpy array to be converted

Raises **runtime_error** – if *n_dims* < 3 or memory is not initialised (nullptr)

const arma::Cube<T> arr_to_cube_view(const py::array<T>& arr)
Create a read-only view on the array as a Armadillo Cube. Copy the memory of the Numpy array if not well_behaved otherwise borrow the array.

Parameters **arr** – numpy array to be converted

Raises **runtime_error** – if *n_dims* < 3 or memory is not initialised (nullptr)

12.1.4 to_arma

`to_arma` is a convenience wrapper around the `arr_to_*` functions and has the same behaviour and rules. For example,

```
arma::Mat<double> mat = to_arma::from<arma::Mat<double>>(arr, copy=false);
```

template <typename armaT> armaT to_arma::from(const py::array_t<eT>& arr)

template <typename armaT> armaT to_arma::from(py::array_t<eT>& arr, bool copy)

template <typename armaT> armaT to_arma::from(py::array_t<eT>&& arr)

12.2 Armadillo to Numpy

This section documents the functions to convert Armadillo matrices, vectors or cubes to Numpy arrays.

12.2.1 Matrix

py::array_t<T> mat_to_arr(arma::Mat<T>& src, bool copy=false)
Convert Armadillo matrix to Numpy array, note the returned array will have column contiguous memory (F-order)

Note the returned array will have F order memory.

Parameters

- **src** – armadillo object to be converted.
- **copy** – copy the memory of the array, default is false which steals the memory

py::array_t<T> mat_to_arr(const arma::Mat<T>& src)
Copy the memory of the Armadillo matrix in the conversion to Numpy array.

Note the returned array will have F order memory.

Parameters **src** – armadillo object to be converted.

py::array_t<T> mat_to_arr(arma::Mat<T>&& src)
Steal the memory of the Armadillo matrix in the conversion to Numpy array.

Note the returned array will have F order memory.

Parameters **src** – armadillo object to be converted.

py::array_t<T> mat_to_arr(arma::Mat<T>* src, bool copy=false)
Convert Armadillo matrix to Numpy array.

Note the returned array will have F order memory.

Parameters

- **src** – armadillo object to be converted.
- **copy** – copy the memory of the array, default is false

12.2.2 Vector

py::array_t<T> col_to_arr(arma::Col<T>& src, bool copy=false)
Convert Armadillo col to Numpy array, note the returned array will have column contiguous memory (F-order)

Note the returned array will have F order memory.

Parameters

- **src** – armadillo object to be converted.
- **copy** – copy the memory of the array, default is false which steals the memory

py::array_t<T> col_to_arr(const arma::Col<T>& src)
Copy the memory of the Armadillo Col in the conversion to Numpy array.

Note the returned array will have F order memory.

Parameters **src** – armadillo object to be converted.

py::array_t<T> col_to_arr(arma::Col<T>&& src)
Steal the memory of the Armadillo Col in the conversion to Numpy array.

Note the returned array will have F order memory.

Parameters **src** – armadillo object to be converted.

py::array_t<T> col_to_arr(arma::Col<T>* src, bool copy=false)
Convert Armadillo Col to Numpy array.

Note the returned array will have F order memory.

Parameters

- **src** – armadillo object to be converted.
- **copy** – copy the memory of the array, default is false

py::array_t<T> row_to_arr(arma::Row<T>& src, bool copy=false)
Convert Armadillo Row to Numpy array, note the returned array will have column contiguous memory (F-order)

Note the returned array will have F order memory.

Parameters

- **src** – armadillo object to be converted.
- **copy** – copy the memory of the array, default is false which steals the memory

py::array_t<T> row_to_arr(const arma::Row<T>& src)
Copy the memory of the Armadillo Row in the conversion to Numpy array.

Note the returned array will have F order memory.

Parameters **src** – armadillo object to be converted.

py::array_t<T> row_to_arr(arma::Row<T>&& src)
Steal the memory of the Armadillo Row in the conversion to Numpy array.

Note the returned array will have F order memory.

Parameters **src** – armadillo object to be converted.

py::array_t<T> row_to_arr(arma::Row<T>* src, bool copy=false)
Convert Armadillo Row to Numpy array.

Note the returned array will have F order memory.

Parameters

- **src** – armadillo object to be converted.
- **copy** – copy the memory of the array, default is false

12.2.3 Cube

py::array_t<T> cube_to_arr(arma::Cube<T>& src, bool copy=false)
Convert Armadillo Cube to Numpy array, note the returned array will have column contiguous memory (F-order)

Note the returned array will have F order memory.

Parameters

- **src** – armadillo object to be converted.
- **copy** – copy the memory of the array, default is false which steals the memory

py::array_t<T> Cube_to_arr(const arma::Cube<T>& src)
Copy the memory of the Armadillo Cube in the conversion to Numpy array.

Note the returned array will have F order memory.

Parameters **src** – armadillo object to be converted.

py::array_t<T> cube_to_arr(arma::Cube<T>&& src)
Steal the memory of the Armadillo cube in the conversion to Numpy array.

Note the returned array will have F order memory.

Parameters **src** – armadillo object to be converted.

py::array_t<T> cube_to_arr(arma::Cube<T>* src, bool copy=false)
Convert Armadillo matrix to Numpy array.

Note the returned array will have F order memory.

Parameters

- **src** – armadillo object to be converted.
- **copy** – copy the memory of the array, default is false

12.2.4 to_numpy

`to_numpy` has overloads for `Mat<T>`, `Row<T>`, `Col<T>` and `Cube<T>`. It should be called with e.g. `to_numpy<arma::Mat<double>>(m)`

template <typename armaT> py::array_t<eT> to_numpy_view(const armaT<eT>& src)
Create 'view' on Armadillo object as non-writeable Numpy array. :note: the returned array will have F order memory. :param src: armadillo object to be converted.

template <typename armaT> py::array_t<eT> to_numpy(armaT<eT>& src, bool copy=false)
Convert Armadillo object to Numpy array.

Note the returned array will have F order memory.

Parameters

- **src** – armadillo object to be converted.
- **copy** – copy the memory of the array, default is false which steals the memory

template <typename armaT> py::array_t<eT> to_numpy(const armaT<eT>& src)
Copy the memory of the Armadillo object in the conversion to Numpy array.

Note the returned array will have F order memory.

Parameters **src** – armadillo object to be converted.

template <typename armaT> py::array_t<eT> to_numpy(armaT<eT>&& src)
Steal the memory of the Armadillo object in the conversion to Numpy array.

Note the returned array will have F order memory.

Parameters **src** – armadillo object to be converted.

template <typename armaT> py::array_t<eT> to_numpy(armaT<eT>* src)
Convert Armadillo object to Numpy array.

Note the returned array will have F order memory.

Parameters

- **src** – armadillo object to be converted.
- **copy** – copy the memory of the array, default is false

CHAPTER 13

Automatic conversion – Type caster

CARMA provides a type caster which enables automatic conversion using pybind11.

Warning: *carma* should included in every compilation unit where automated type casting occurs, otherwise undefined behaviour will occur.

Note: The underlying casting function has overloads for `{const lvalue, lvalue, rvalue, pointer}` Armadillo objects of type `{Mat, Col, Row, Cube}`.

13.1 Return policies

Pybind11 provides a number of return value policies of which a subset is supported:

To pass the return value policy set it in the binding function:

```
m.def("example_function", &example_function, return_value_policy::copy);
```

return_value_policy::move
: move/steal the memory from the armadillo object

return_value_policy::automatic
: move/steal the memory from the armadillo object

return_value_policy::take_ownership
: move/steal the memory from the armadillo object

return_value_policy::copy
: copy the memory from the armadillo object

CHAPTER 14

ArrayStore

ArrayStore is a convenience class for storing the memory in a C++ class.

Warning: The ArrayStore owns the data, the returned numpy arrays are views that are tied to the lifetime of ArrayStore.

template <typename armaT> ArrayStore

mat armaT
Matrix containing the memory of the array.

ArrayStore(*py::array_t<T>& arr*, *bool copy*)
Class constructor

Parameters

- **arr** – Numpy array to be stored as Armadillo matrix
- **steal** – Take ownership of the array if not copy

template <typename armaT> ArrayStore(const arma & src)
Class constructor, object is copied

Parameters **src** – Armadillo object to be stored

template <typename armaT> ArrayStore(armaT& src, copy)
Class constructor, object is copied or moved/stolen

Parameters **src** – Armadillo object to be stored

template <typename armaT> ArrayStore(armaT && src)
Class constructor, object is moved

Parameters **mat** – Armadillo object to be stored

get_view(*bool writeable*)
Obtain a view of the memory as Numpy array.

Parameters **writeable** – Mark array as writeable

set_array(*py::array_t<T> & arr*, *bool copy*)
Store new array in the ArrayStore.

Parameters

- **arr** – Numpy array to be stored as Armadillo matrix
- **copy** – Take ownership of the array or copy

template <typename T> set_data(const armaT& src)
Store new matrix in the ArrayStore, object is copied.

Parameters **src** – Armadillo object to be stored

template <typename T> set_data(armaT& src, bool copy)
Store new object in the ArrayStore, copied or moved.

Parameters **src** – Armadillo object to be stored, matrix is copied

template <typename T> set_data(arma::Mat<T> && src)
Store new matrix in the ArrayStore, object is moved.

Parameters **src** – Armadillo matrix to be stored

CHAPTER 15

NdArray flags

Utility functions to check flags of numpy arrays.

bool is_f_contiguous(const py::array_t<T> & arr)
Check if Numpy array's memory is Fotran contiguous.

Parameters **arr** – numpy array to be checked

bool is_c_contiguous(const py::array_t<T> & arr)
Check if Numpy array's memory is C contiguous.

Parameters **arr** – numpy array to be checked

bool is_writable(const py::array_t<T> & arr)
Check if Numpy array's memory is mutable.

Parameters **arr** – numpy array to be checked

bool is_owndata(const py::array_t<T> & arr)
Check if Numpy array's memory is owned by numpy.

Parameters **arr** – numpy array to be checked

bool is_aligned(const py::array_t<T> & arr)
Check if Numpy array's memory is aligned.

Parameters **arr** – numpy array to be checked

bool requires_copy(const py::array_t<T> & arr)
Check if Numpy array memory needs to be copied out, is true when either not writable, owndata or is not aligned.

Parameters **arr** – numpy array to be checked

void set_not_owndata(py::array_t<T> & arr)
Set Numpy array's flag OWNDATA to false.

Parameters **arr** – numpy array to be changed

void set_not_writeable(py::array_t<T> & arr)
Set Numpy array's flag WRITEABLE to false.

Parameters `arr` – numpy array to be changed

Index