Output formats & cnpy++

M. Reininghaus

2023-05-04

My parquet experience

- recently had to read some showers, 20 .. 40 GB particle file
- `corsika.Library(directory)` is greedy, loads **everything** into memory
  - can’t even access yaml metadata without reading everything
- `pandas.read_parquet(file)` similar

.npy format

- described in *NEP 1 — A simple file format for NumPy arrays*
- platform-independent, binary
- header mostly ASCII (python dict)
- handles any NumPy data type
  - plain integer/floating-point numbers (arrays)
  - tuples with named elements (tabular data, *structured arrays*)
- memory-mappable (access content as if it were in memory; read files not fitting in memory)
- very easy to use in Python:

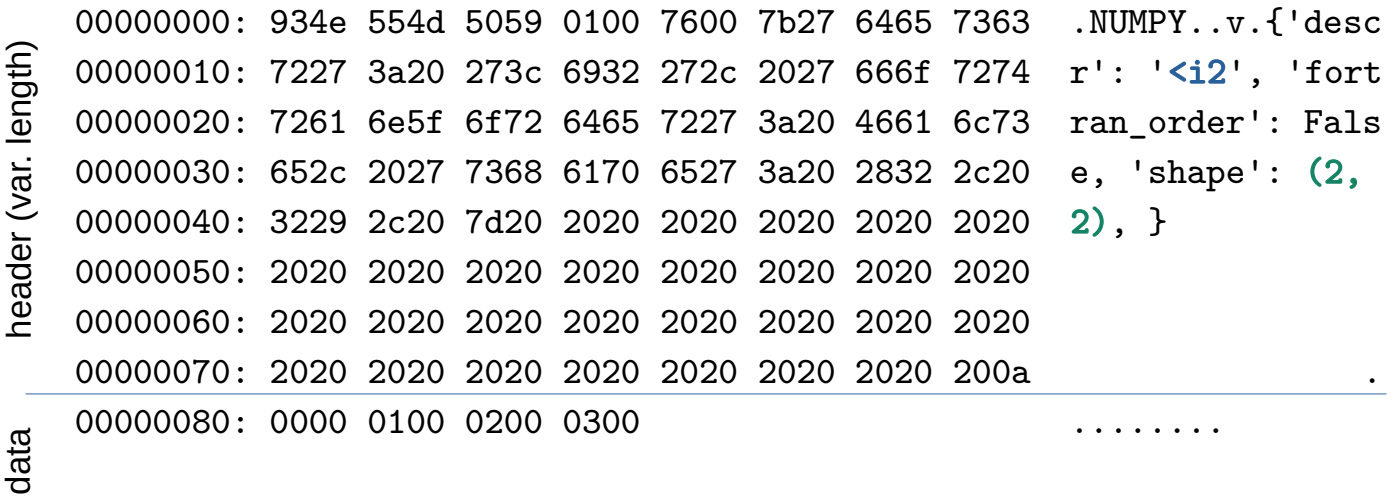
```
np.save(data, "file.npy")
data = np.load("file.npy")
```

- multiple datasets can be zipped together in .npz, optionally compressed

Examples

- multidim. arrays / tensors

$$\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}$$



Examples

- structured array ("a" → int32, "b" → float, "c" → float, "d" → float)

header (var. length)

```
00000000: 934e 554d 5059 0100 7600 7b27 6465 7363  .NUMPY..v.{'desc
00000010: 7227 3a20 5b28 2761 272c 2027 3c69 3427  r': [('a', '<i4'
00000020: 292c 2028 2762 272c 2027 3c66 3427 292c  ), ('b', '<f4'),
00000030: 2028 2763 272c 2027 3c66 3427 292c 2028   ('c', '<f4'), (
00000040: 2764 272c 2027 3c66 3427 295d 2c20 2766  'd', '<f4')], 'f
00000050: 6f72 7472 616e 5f6f 7264 6572 273a 2046  ortran_order': F
00000060: 616c 7365 2c20 2773 6861 7065 273a 2028  alse, 'shape': (
00000070: 342c 292c 207d 2020 2020 2020 2020 200a  4,), }          .
```

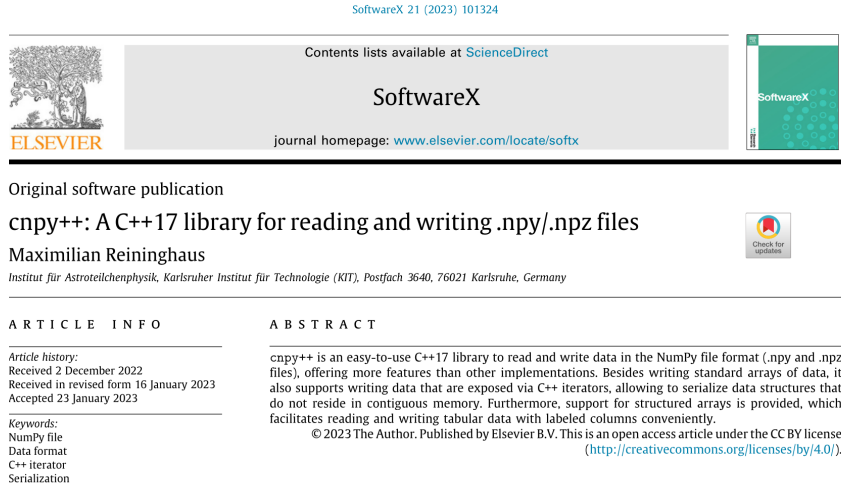
data

```
00000080: 0100 0000 0000 803f 0000 803f 0000 803f  .......?...?...?
00000090: 0200 0000 0000 8040 0000 0041 0000 8041  .......@...A...A
000000a0: 0300 0000 0000 1041 0000 d841 0000 a242  .......A...A...B
000000b0: 0400 0000 0000 8041 0000 8042 0000 8043  .......A...B...C
```

cnpy++

- reading & writing .npy / .npz with C++17
- based on cnpy (unmantained)
- codebase modernized, features added: compressed npz (libzip), **structured arrays**, writing from **iterators**

  ➔ write non-contiguous data (std::list<>), lazy iterators



SoftwareX 21 (2023) 101324

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

cnpy++: A C++17 library for reading and writing .npy/.npz files

Maximilian Reininghaus

Institut für Astroteilchenphysik, Karlsruher Institut für Technologie (KIT), Postfach 3640, 76021 Karlsruhe, Germany

ARTICLE INFO

Article history:
Received 2 December 2022
Received in revised form 16 January 2023
Accepted 23 January 2023

Keywords:
NumPy file
Data format
C++ iterator
Serialization

ABSTRACT

cnpy++ is an easy-to-use C++17 library to read and write data in the NumPy file format (.npy and .npz files), offering more features than other implementations. Besides writing standard arrays of data, it also supports writing data that are exposed via C++ iterators, allowing to serialize data structures that do not reside in contiguous memory. Furthermore, support for structured arrays is provided, which facilitates reading and writing tabular data with labeled columns conveniently.

Example: std::list<> ↔ .npy

```
#include <cstdint>
#include <list>
#include <cnpy++.hpp>

int main () {
     // prepare data in a std::list
     std::list<uint16_t> const l{10, 11, 12, 13};

     // save into "list.npy"
     cnpypp::npy_save("list.npy", l.begin(), l.end());

     // append data from reversed list
     cnpypp::npy_save("list.npy", l.rbegin(), l.rend(), "a" );

     cnpypp::NpyArray const arr = cnpypp::npy_load("list.npy", true);
     auto const* const loaded_data = arr.data<uint32_t>();

     for (uint32_t x: loaded_data) { std::cout << x << ", "; }
     // 10, 11, 12, 13, 13, 12, 11, 10
}
```

Example: structured arrays

```
#include <cstdint>
#include <vector>
#include <cnpy++.hpp>

int main () {
     std::vector<std::tuple<int32_t, int8_t, int16_t>> const tupleVec{
        {0xaaaaaaaa, 0xbb, 0xcccc}, {0xddddddddd, 0xee, 0xffff},
        {0x99999999, 0x88, 0x7777}};

    cnpypp::npy_save("structured.npy", {"a", "b", "c"}, tupleVec.begin(),
                     {tupleVec.size()});

    // load memory-mapped
    cnpypp::NpyArray arr = cnpypp::npy_load("structured.npy", true);
    auto r = arr.tuple_range<int32_t, int8_t, int16_t>();

    for (auto [a, b, c]: r) { /* do something */ }

}
```

Example: structured arrays

```
#include <range/v3/view/zip.hpp>

int main() {
  // prepare two independent vectors to
  // be used as columns
  std::vector<int32_t> const xVec{4, 5, 6};
  std::vector<double> const yVec{2.22, 3.33, 4.44};

  // zip them together into tuple via zip iterator
  auto const zipview = ranges::views::zip(xVec, yVec);

  cnpypp::npy_save("structured.npy", {"x", "y"},
    zipview.begin(), {zipview.size()});

  // read back into readData
  cnpypp::NpyArray const readData =
    cnpypp::npy_load("structured.npy");

  // iterate over tuples
  for (auto const& [x, y]:
       readData.tuple_range<int32_t, double>()) {
    std::cout << x << "  " << y << std::endl;
  }

  // iterate only over "x" column
  for (int32_t const x:
       readData.column_range<int32_t>("x")) {
    std::cout << x << std::endl;
  }
}
```

CORSIKA 8

- cnpy(++) used for `boost::histogram` serialization
- could replace parquet (long. profiles, radio traces, …), possibly with compression
- current version usable for *particle* output only by appending blocks, not 1-by-1 (→ header needs to be updated)