NAIA

NTUPLES FOR AMS-ITALY ANALYSIS

INTRODUCTION TO THE FRAMEWORK AND DATA-FORMAT

MOTIVATIONS

1. **Resource optimization**
    - With multiple groups producing each their own set of ntuples, lots of data is replicated on disk, which results in a waste of resources.
    - Also, groups are competing for computing resources for ntuple production.
2. **Code exchange**
    - Using the same data makes it easier to exchange selections and algorithms/procedures.
3. **Reproducibility / readability**
    - Most often custom data formats are produced in a custom way, with a custom processing.
    - Additionally, in many cases, only the code owner can easily understand what's going in the analysis code.

DRIVING PRINCIPLES

- Don't throw anything out
  - This means processing and saving all the events from the original AMS-Root files
- Don't require network access
  - All the needed data should be inside NAIA files (e.g. no online access to RTI csv files on cvmfs)
- Try to cover at least 90% of use-cases
  - Initial variable list comes from an internal survey including every analysis group
  - For special kind of analyses needing specialized variables, we plan to support user-defined TTree-friending

DRIVING PRINCIPLES

- Don't read what you don't need
  - Only perform I/O reads when variables are accessed. Allow to skip uninteresting events before branch reading even occurs.
- Easy to understand
  - Code should be readable and expressive.
  - Variable name and usage should make clear what the intention of the programmer is, at least to an intuitive level.
  - Function names should be descriptive and hint at what the result of the function is.
- Easy to use
  - Automatic installation for local development. CVMFS binary releases for usage on clusters.

GETTING STARTED

Requirements:

- A C++ compiler with full C++17 support
  (currently GCC 12.1.0)
- CMake version 3.13 or higher
- ROOT version 6.28 or higher compiled with C++17 support
  (currently 6.28/04)

This mainly applies if you want to install NAIA on your personal machine. For distributed use (CNAF / CERN) all requirements and NAIA binaries are distributed via CVMFS

```
/cvmfs/ams.cern.ch/Offline/amsitaly/public/install/x86_64-el9-gcc12.1/naia
```

and the correct environment can be setup with a dedicated script

```
/cvmfs/ams.cern.ch/Offline/amsitaly/public/install/x86_64-el9-gcc12.1/naia/naia/v1.1.0/setenvs/setenv_gcc6.28_el9
```

GETTING STARTED

If you are building NAIA on your machine the installation is quite easy

```
# clone NAIA code
git clone ssh://git@gitlab.cern.ch:7999/ams-italy/naia.git -b v1.0.1 # (clone via SSH)
# setup build and final install directories
mkdir naia.build naia.install
# build NAIA
cd naia.build
cmake ../naia -DCMAKE_INSTALL_PREFIX=../naia.install
make all install
```

To use the NAIA ntuples your project will need:

- the headers in `naia.install/include`
- the `naia.install/lib/libNAIAUtility.so` library
- the `naia.install/lib/libNAIAContainers.so` library
- the `naia.install/lib/libNAIAChain.so` library

THE NAIA DATA MODEL

Our data model starts with the `NAIAChain` object

This is the main way to open a NAIA rootfile, it will take care of loading all the relevant TTrees and setting up what we call the "read-on-demand" mechanism (*more on this later*)

Example:

```
// ...
#include "Chain/NAIAChain.h"

int main(int argc, char const *argv[]) {
  // Create a chain object
  NAIA::NAIAChain chain;
  // add one (or more) file to it
  chain.Add("somefile.root");
  // setup the read-on-demand mechanism // N.B: important and mandatory!
  chain.SetupBranches();
}
```

THE NAIA DATA MODEL

Once your chain is created and ready to use, you can easily loop over all the events in the chain, with the help of the `Event` class

```cpp
// ...
#include "Chain/NAIAChain.h"

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

  NAIA::NAIAChain chain;
  chain.Add("somefile.root");
  chain.SetupBranches();

  // Event loop!
  for (Event& event : chain){
    // your analysis here :)
  }
}
```

(you can use the `NAIAChain::GetEvent()` method for index-based looping, if needed)

THE NAIA DATA MODEL

NAIA also provide a simple way of skimming a chain and only save interesting events in the output file

```cpp
// ...
#include "Chain/NAIAChain.h"

int main(int argc, char const *argv[]) {
  NAIA::NAIAChain chain;
  chain.Add("somefile.root");
  chain.SetupBranches();

  auto handle = chain.CreateSkimTree("skimmed.root", "");

  // Event loop!
  for (Event& event : chain){
    if( is_interesting(event) ){
      handle.Fill();
    }
  }

  handle.Write();
}
```

THE NAIA DATA MODEL

The `Event` class is probably the most important one, but also the most boring since it's basically a proxy class containing a collection of *Containers*

Containers are the real building blocks of the NAIA datamodel.

the main TTree and allows for reading the corresponding branch data only when first accessed.

(This means that if you never use a particular container in your analysis, you'll never read the corresponding data from file)

Public Attributes

Header	header
EventSummary	evSummary
DAQ	daq
TofBase	tofBase
TofPlus	tofPlus
TofBaseStandalone	tofBaseSt
TofPlusStandalone	tofPlusSt
EcalBase	ecalBase
EcalPlus	ecalPlus
TrTrackBase	trTrackBase
TrTrackPlus	trTrackPlus
SecondTrTrackBase	secondTrTrackBase
TrTrackBaseStandalone	trTrackBaseSt
TrdKBase	trdKBase
TrdKBaseStandalone	trdKBaseSt
RichBase	richBase
RichPlus	richPlus
UnbExtHitBase	extHitBase
MCTruthBase	mcTruthBase
MCTruthPlus	mcTruthPlus

THE NAIA DATA MODEL

Container is the general term to define a class in the NAIA data model that groups several variables, according to specific criteria (e.g. all the variables related to the TOF).

Most containers come in two variants: the *Base* and the *Plus* variant

The *Base* variant contains variables that are accessed by almost every analysis or that are accessed most often

The *Plus* variant contains variables that won't be needed by everyone, or may be needed less frequently

> The important thing is that you always access variables using the `->` operator on the container. This is how the read-on-demand is implemented and leads to wrong results otherwise.

THE NAIA DATA MODEL

Variables in NAIA are a bit more complex, for valid reasons:

We want our data model to be **as light as possible** (especially since we're processing and saving every single event)

This implies that if something's missing we don't want to write anything to disk

- e.g. if there's no hit on Tracker L1 we don't want to write 0, or -9999 or whatever sentinel value to keep track of this. We don't want to write anything at all.

We achieve this by using associative containers (mostly `std::map`) and realizing that in many cases there are patterns we can exploit.

Example: several variables come in "flavors" or are computed by different reconstructions

THE NAIA DATA MODEL

Doing AMS analysis means constantly dealing with "one value for each X type", where X could be a charge reconstruction method, track fitting algorithm, ECAL BDT estimator, and so on…

Example: For tracker hits, there are three available charge reconstruction methods: STD, Hu Liu, Yi Jia.

In general, these reconstructions are not guaranteed always to succeed

We handle this by defining the following:

```cpp
// one number per charge reconstruction type
template <class T> using TrackChargeVariable = std::map<TrTrack::ChargeRecoType, T>;
```

Where `TrTrack::ChargeRecoType` is an enum with three values

```cpp
enum ChargeRecoType {
  STD, ///< Standard tracker charge reconstruction
  HL,  ///< Hu Liu reconstruction
  YJ,  ///< Yi Jia reconstruction
};
```

THE NAIA DATA MODEL

Why an enum?!?

Well... what is more readable and understandable

```
float inner_charge = event.trTrackBase->InnerCharge[2];
```

or

```
float inner_charge = event.trTrackBase->InnerCharge[TrTrack::ChargeRecoType::YJ];
```

Readability debates aside, this avoids the confusion usually brought by magic numbers (you might not remember after a few weeks that Yi Jia reconstruction is at index 2, for example)

In addition, if this ever changes in the future, and it is moved to index 3 you won't have to modify your code in the second case.

For this reason almost every variable structure in NAIA is accessed using specific enums.

THE NAIA DATA MODEL

Some variables have nested structures, for example in `TrTrackPlus` we have:

```
///< Track hit charge (X and Y-side) for each layer, for each charge reconstruction.
LayerVariable<TrackChargeVariable<TrackSideVariable<float>>> LayerCharge;
```

where for each layer, then for each reconstruction, then for each side we store a number.

But it is not guaranteed that the track will have a hit on, say, Layer 1. Or that the underlying cluster is correctly identified on the X side. How do we check for this?

For this, there is a dedicated `ContainsKeys` function, which checks if the desired elements (identified by some keys, i.e. the aforementioned enums) exist in the structure

```
if (ContainsKeys(event.trTrackPlus->LayerCharge, layer_idx, Track::ChargeRecoType::YJ, TrTrack::Side::X)) {
  // do stuff...
}
```

you can find the full list of variable structures here

THE NAIA DATA MODEL

One quick way of discarding uninteresting events without reading almost anything is by using the event mask.

The mask is simply a bitmask where every bit represents a particular `Category`. If the event satisfies a given `Category`, the corresponding bit in the mask will be set.

```cpp
// ...
#include "Chain/NAIAChain.h"

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

  NAIA::NAIAChain chain;
  chain.Add("somefile.root");
  chain.SetupBranches();

  auto handle = chain.CreateSkimTree("skimmed.root", "");

  // Event loop!
  for (Event& event : chain){
    if( event.CheckMask(NAIA::Category::Charge1_Tof | NAIA::Category::Charge1_Trk) ){
      handle.Fill();
    }
  }

  handle.Write();
}
```

THE NAIA DATA MODEL

To do analysis you don't need only Events, but also information about livetime, or amount of generated MC events…

Each NAIA file contains two additional trees just for that

One contains data about the ISS position, its orientation, and physical quantities connected to them, as well as some time-averaged data about the run itself. This kind of data is usually retrieved in AMS analysis from the RTI (Real Time Information) database. This database stores data with a time granularity of one second, and it can be accessed using the gbatch library.

We don't want any dependency on gbatch so the entire RTI database is converted to a TTree that has only one branch, which contains objects of the `RTIInfo` class, one for each second of the current run.

```
// inside the event loop
// Get the RTI info for the current event
NAIA::RTIInfo &rti_info = chain.GetEventRTIInfo();
```

THE NAIA DATA MODEL

We don't want any dependency on gbatch so the entire RTI database is converted to a TTree that has only one branch, which contains objects of the RTIInfo class, one for each second of the current run.

The tree can be accessed from outside the event loop as well

```
TChain* rti_chain = chain.GetRTITree();
NAIA::RTIInfo* rti_info = new RTIInfo();
rti_chain->SetBranchAddress("RTIInfo", &rti_info);

for (unsigned long long isec=0; isec < rti_chain->GetEntries(); ++isec){
  rti_chain->GetEntry(isec);

  // your analysis here :)
}
```

Clearly useful if you only have to just recompute livetime or analyse only RTI data

THE NAIA DATA MODEL

The second tree contains useful information about the original AMSRoot file from which the current NAIA file was derived.

This information is stored in the `FileInfo` TTree, which usually has only a single entry for each NAIA root-file.

(Having this data in a TTree allows us to chain multiple NAIA root-files and still be able to retrieve the FileInfo data for the current run we're processing)

This tree has one branch, which contains objects of the `FileInfo` class and, if the NAIA root-file is a Montecarlo file, an additional branch containing objects of the `MCFileInfo` class.

```
// inside the event loop
// Get the File infos for the current event
NAIA::FileInfo &file_info = chain.GetEventFileInfo();
// if this is a MC file
NAIA::MCFileInfo &mcfile_info = chain.GetEventMCFileInfo();
```

THE NAIA DATA MODEL

Also in this case the tree can be accessed from outside the event loop as well

```
TChain* file_chain = chain.GetFileInfoTree();
NAIA::FileInfo* file_info = new NAIA::FileInfo();
NAIA::MCFileInfo* mcfile_info = new NAIA::MCFileInfo();

file_chain->SetBranchAddress("FileInfo", &file_info);
if(chain.IsMC()){
  file_chain->SetBranchAddress("MCFileInfo", &mcfile_info);
}

for(unsigned long long i=0; i < file_chain->GetEntries(); ++i){
  file_chain->GetEntry(i);

  // do stuff with file_info

  if(chain.IsMC()){
    // do stuff with mcfile_info
  }
}
```

USING NAIA IN YOUR ANALYSIS

There are some examples in NAIA that should guide you in building your analysis with NAIA. These are divided by usage type:

CMake: *(recommended)*

It is a powerful cross-platform build system that is used to specify in a generic way how programs should be compiled and generate the corresponding Makefiles. It is especially useful when your project makes use of other packages / libraries that need to be imported

```
# CMakeLists.txt:
project(testNAIA)

find_package(NAIA 1.1.1 REQUIRED)

add_executable(main src/main.cpp)
target_link_libraries(main PUBLIC NAIA::NAIAChain)
```

it becomes quite effective when the size of the project increases (many executables/libraries)

USING NAIA IN YOUR ANALYSIS

```
# CMakeLists.txt:
project(testNAIA)
set(CMAKE_CXX_STANDARD 14)

find_package(NAIA 1.0.1 REQUIRED)

add_executable(main src/main.cpp)
target_link_libraries(main PUBLIC NAIA::NAIAChain)
```

NB: following good CMake practices, NAIA internally defines everything that is needed in terms of targets.

The `NAIA::NAIAChain` target internally knows all the include paths, preprocessor macros, library paths, libraries that it needs so that CMake can propagate these requirements to all targets linking against `NAIA::NAIAChain`, such as your own library targets or executables.

To compile just run

```
mkdir build
cd build
cmake .. -DNAIA_DIR=${path_to_your_naia_install}/cmake
make
```

USING NAIA IN YOUR ANALYSIS

Makefile:

If you want to write your own Makefile you can take a look at the existing examples and update the NAIA_DIR variable inside. Remember to add include paths/libraries if needed or if something changes between NAIA versions.

```makefile
NAIA_DIR=/path/to/your/naia/install

CC = g++
CFLAGS = $(shell root-config --cflags) -DSPDLOG_FMT_EXTERNAL
INCLUDES = -I$(NAIA_DIR)/include -I./include
LFLAGS = $(shell root-config --libs) -L $(NAIA_DIR)/lib -Wl,-rpath=$(NAIA_DIR)/lib
LIBS = -lNAIAUtility -lNAIAChain -lNAIAContainers

SRCS = src/main.cpp
OBJS = $(SRCS:.cpp=.o)

MAIN = main

.PHONY: depend clean

all:    $(MAIN)
	@echo  main has been compiled

$(MAIN): $(OBJS)
	$(CC) $(CFLAGS) $(INCLUDES) -o $(MAIN) $(OBJS) $(LFLAGS) $(LIBS)
```

USING NAIA IN YOUR ANALYSIS

ROOT macros:

In this case the libraries and include paths are setup by a dedicated macro

```
// load.C:
{
  TString naia_dir = "/path/to/your/naia/install/dir";
  gROOT->ProcessLine(".include" + naia_dir + "/include");
  gSystem->SetDynamicPath(naia_dir + "/lib:" + gSystem->GetDynamicPath());
  gSystem->Load("libNAIAUtility");
  gSystem->Load("libNAIAContainers");
  gSystem->Load("libNAIAChain");

  gROOT->ProcessLine("#define SPDLOG_FMT_EXTERNAL");
  gROOT->ProcessLine("#include \"Chain/NAIAChain.h\"");
}
```

and then run as

```
root load.C main.cpp
```

USING NAIA IN YOUR ANALYSIS

Bonus: RDataFrame

Sometimes you do need to make a quick plot and macros are just for that. One very cool option could be to use RDataFrame. You won't use a `NAIAChain`, in this case, you're working directly with the naked branches (and you still need `load.C`)

```
void plot_innercharge() {
  // enable multi-threading
  ROOT::EnableImplicitMT();

  TFile *infile = TFile::Open("/storage/gpfs_ams/ams/groups/AMS-Italy/ntuples/v1.1.0/ISS.B1236/pass8/15913618
  TTree *tree = infile->Get<TTree>("NAIAChain");
  ROOT::RDataFrame rdf{*tree};

  // apply two cuts:
  // - Track chisquare Y < 10 (inner tracker only fit)
  // - At least 5 XY hits
  // define the variable to be plotted
  auto augmented_d =
      rdf.Filter(
             [](NAIA::TrTrackBaseData &trtrack) {
               return trtrack.TrChiSq[NAIA::TrTrack::Fit::Kalman][NAIA::TrTrack::Span::InnerOnly][NAIA::TrTra
             },
             {"TrTrackBaseData"})
          .Filter([](NAIA::TrTrackBaseData &trtrack) { return trtrack.LayerChargeXY.size() > 4; }, {"TrTrackB
          .Define("TrInnerCharge",
```

USING NAIA IN YOUR ANALYSIS

```
void plot_innercharge() {
  // enable multi-threading
  ROOT::EnableImplicitMT();

  TFile *infile = TFile::Open("/storage/gpfs_ams/ams/groups/AMS-Italy/ntuples/v1.1.0/ISS.B1236/pass8/15913618
  TTree *tree = infile->Get<TTree>("NAIAChain");
  ROOT::RDataFrame rdf{*tree};

  // apply two cuts:
  // - Track chisquare Y < 10 (inner tracker only fit)
  // - At least 5 XY hits
  // define the variable to be plotted
  auto augmented_d =
      rdf.Filter(
              [](NAIA::TrTrackBaseData &trtrack) {
                return trtrack.TrChiSq[NAIA::TrTrack::Fit::Kalman][NAIA::TrTrack::Span::InnerOnly][NAIA::TrTra
              },
              {"TrTrackBaseData"})
          .Filter([](NAIA::TrTrackBaseData &trtrack) { return trtrack.LayerChargeXY.size() > 4; }, {"TrTrackB
          .Define("TrInnerCharge",
```

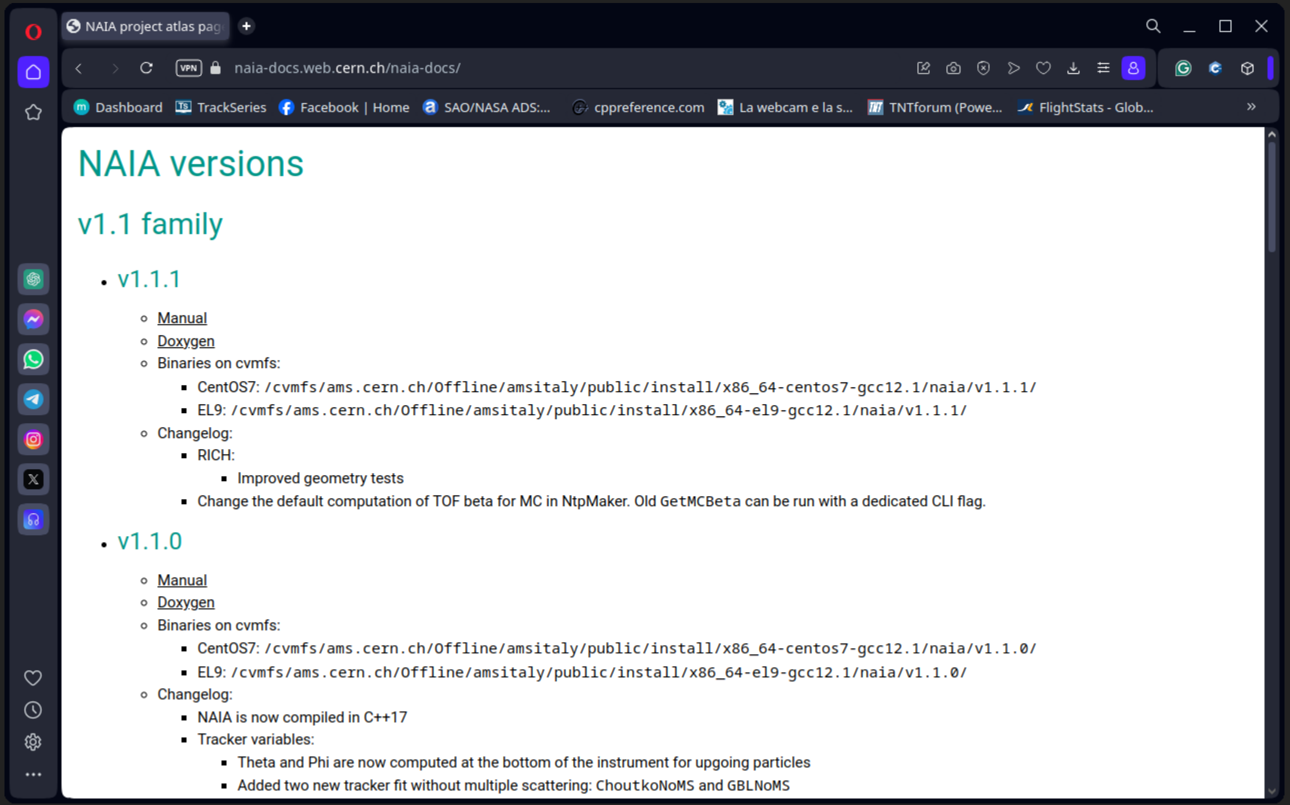
Note the usage of "TrTrackBaseData" rather than "TrTrackBase" (it's the name of the actual tree branch) and no read-on-demand (RDataFrame does this by default)

Also, you get speedups for free, since it's automatically multi-threaded (very useful for final-stage plots or quicklooks!)

SUPPORT AND COMMUNITY

NAIA has a simple landing page that provides quick and easy access to a manual and class documentation for each version

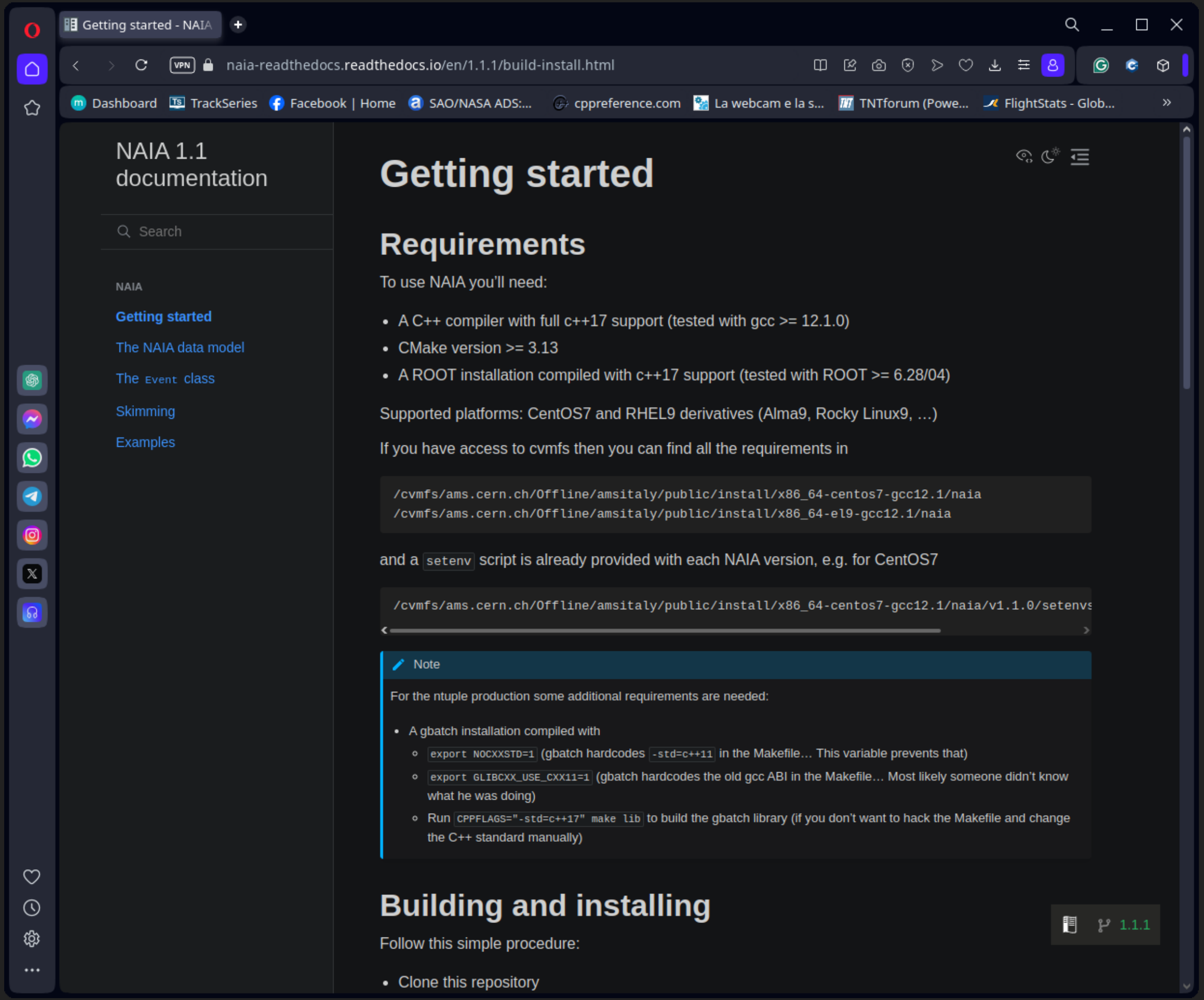
(including changelogs and a reference on where the data is stored at CNAF)



SUPPORT AND COMMUNITY

A simple user manual should guide you on the steps needed to install and use NAIA, and provides a quick reference on the datamodel and ideas behind NAIA.

(mostly what you have seen on these slides, but a bit more descriptive)

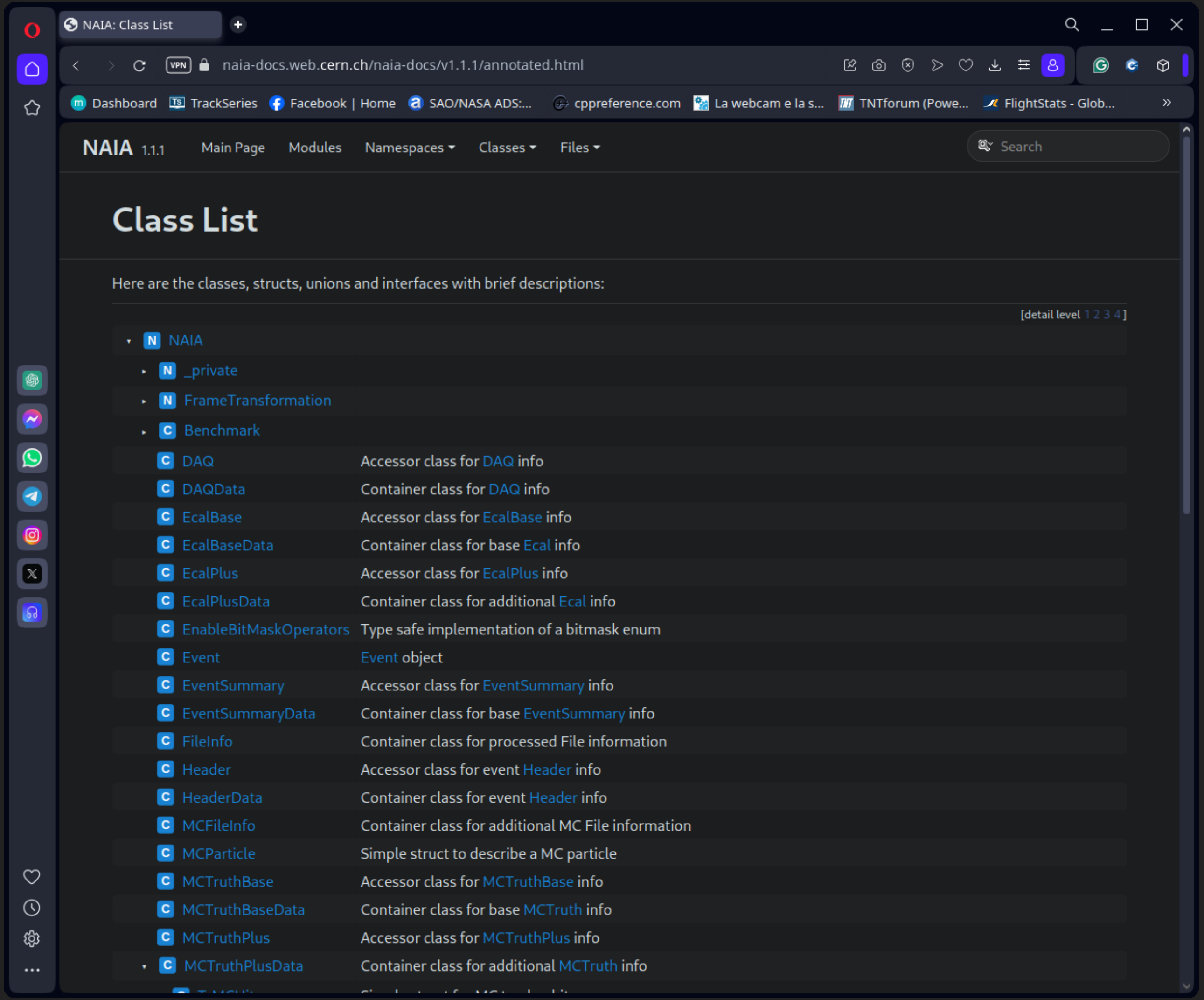


SUPPORT AND COMMUNITY

A simple user manual should guide you on the steps needed to install and use NAIA, and provides a quick reference on the datamodel and ideas behind NAIA.

(mostly what you have seen on these slides, but a bit more descriptive)

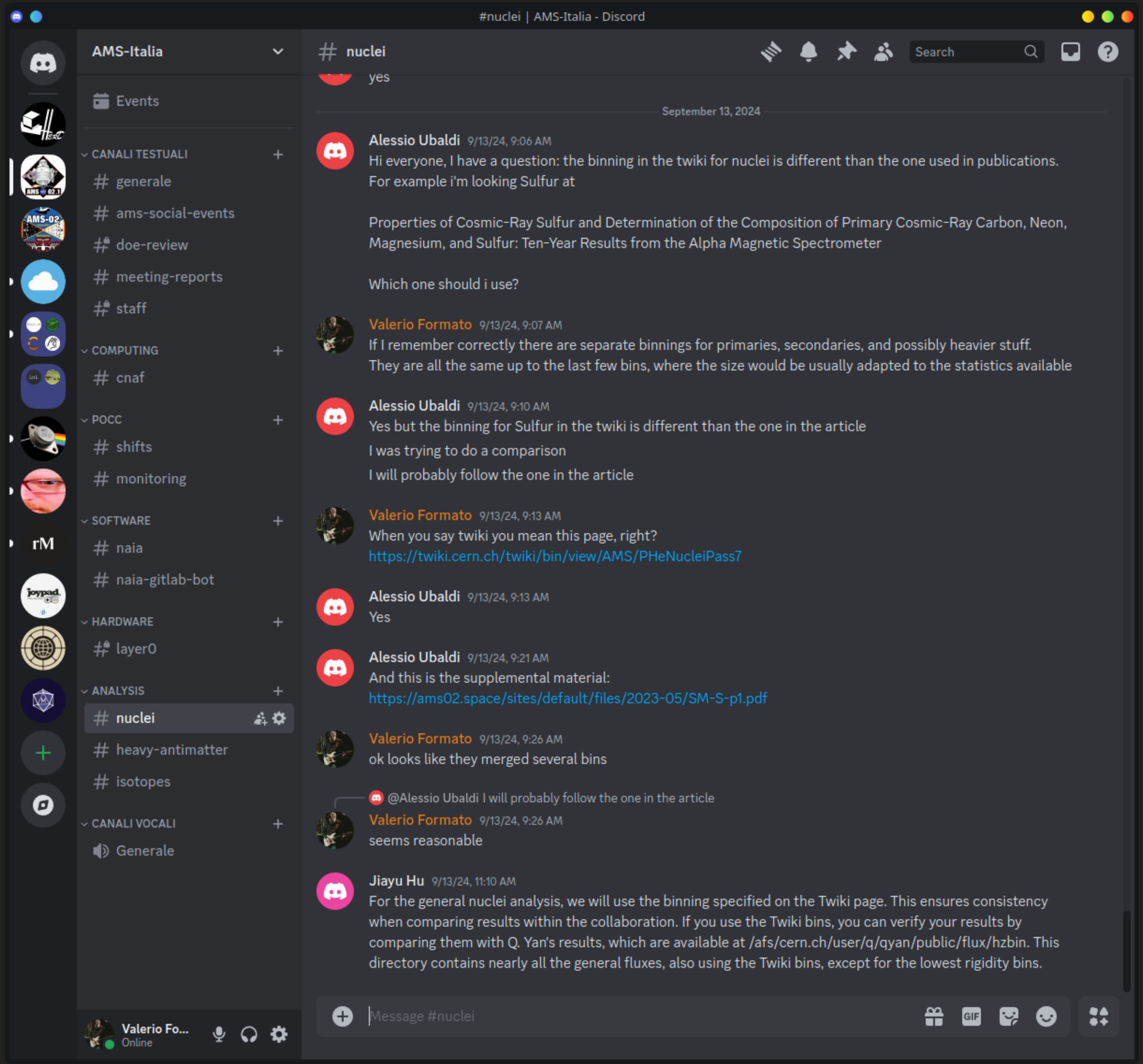
And of course, for all the details on classes and methods and so on, there is doxygen page for every version.



SUPPORT AND COMMUNITY

In addition you are strongly encouraged to join the AMS-Italy discord server.

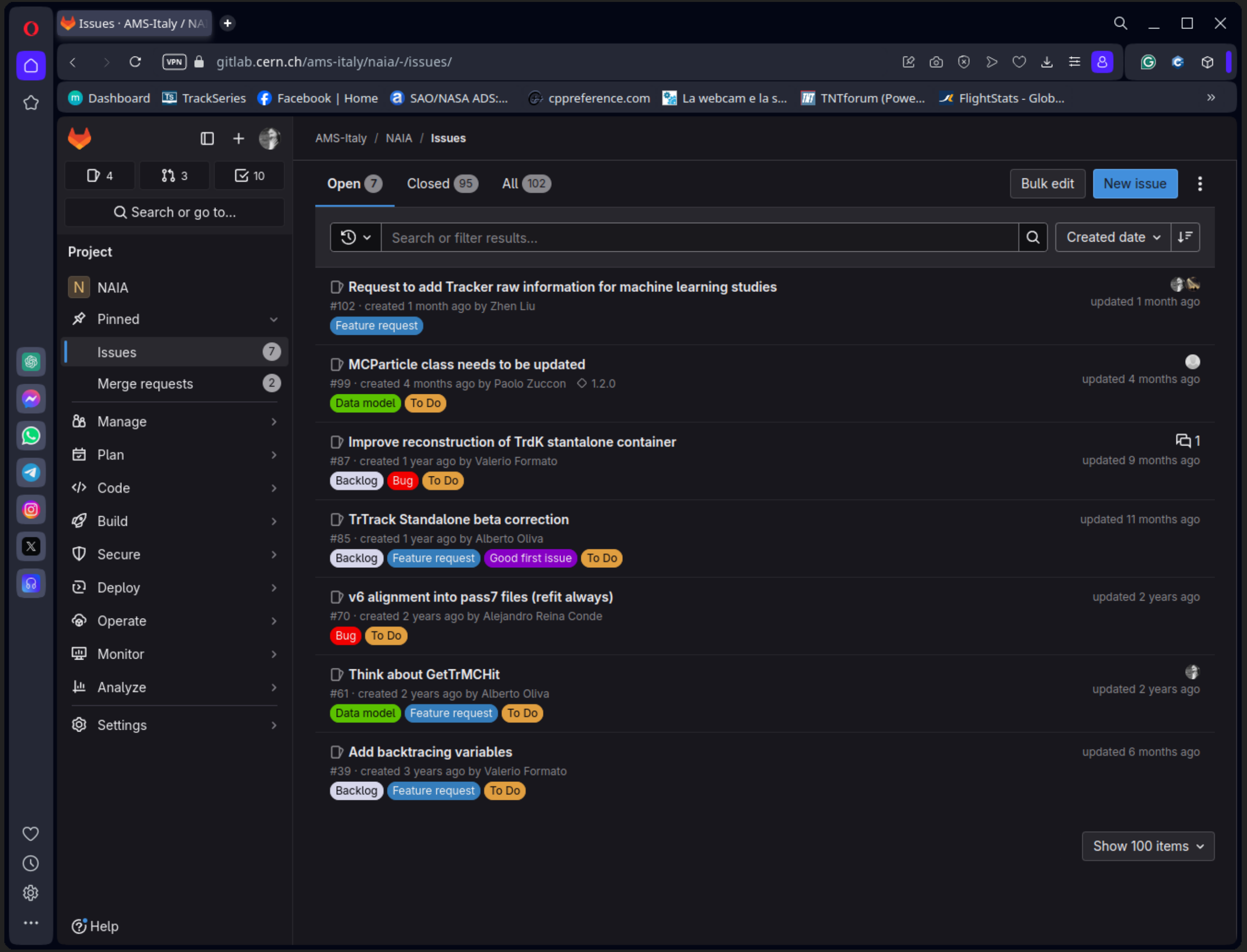
This is particularly useful as a community gathering point to chat and discuss common activities, meetings, analysis and tools.


#nuclei | AMS-Italia - Discord
AMS-Italia
Events
CANALI TESTUALI
generale
ams-social-events
doe-review
meeting-reports
staff
COMPUTING
cnaf
POCC
shifts
monitoring
SOFTWARE
naia
naia-gitlab-bot
HARDWARE
layer0
ANALYSIS
nuclei
heavy-antimatter
isotopes
CANALI VOCALI
Generale
Valerio Fo...
Online
nuclei
Search
yes
September 13, 2024
Alessio Ubaldi 9/13/24, 9:06 AM
Hi everyone, I have a question: the binning in the twiki for nuclei is different than the one used in publications.
For example i'm looking Sulfur at
Properties of Cosmic-Ray Sulfur and Determination of the Composition of Primary Cosmic-Ray Carbon, Neon, Magnesium, and Sulfur: Ten-Year Results from the Alpha Magnetic Spectrometer
Which one should i use?
Valerio Formato 9/13/24, 9:07 AM
If I remember correctly there are separate binnings for primaries, secondaries, and possibly heavier stuff.
They are all the same up to the last few bins, where the size would be usually adapted to the statistics available
Alessio Ubaldi 9/13/24, 9:10 AM
Yes but the binning for Sulfur in the twiki is different than the one in the article
I was trying to do a comparison
I will probably follow the one in the article
Valerio Formato 9/13/24, 9:13 AM
When you say twiki you mean this page, right?
https://twiki.cern.ch/twiki/bin/view/AMS/PHeNucleiPass7
Alessio Ubaldi 9/13/24, 9:13 AM
Yes
Alessio Ubaldi 9/13/24, 9:21 AM
And this is the supplemental material:
https://ams02.space/sites/default/files/2023-05/SM-S-p1.pdf
Valerio Formato 9/13/24, 9:26 AM
ok looks like they merged several bins
@Alessio Ubaldi I will probably follow the one in the article
Valerio Formato 9/13/24, 9:26 AM
seems reasonable
Jiayu Hu 9/13/24, 11:10 AM
For the general nuclei analysis, we will use the binning specified on the Twiki page. This ensures consistency when comparing results within the collaboration. If you use the Twiki bins, you can verify your results by comparing them with Q. Yan's results, which are available at /afs/cern.ch/user/q/qyan/public/flux/hzbin. This directory contains nearly all the general fluxes, also using the Twiki bins, except for the lowest rigidity bins.
Message #nuclei

SUPPORT AND COMMUNITY

Finally, for any bug, issue, or feature request for NAIA you can always go to the main repository on gitlab and open an issue to describe your needs.

There are already two templates: one for bug reporting, the other for feature requests.



TOOLS IN THE WORK

NAIA is just a data model and framework for AMS analysis. We can think of it as the foundational layer, but there is room for creating useful tools to further the data analysis experience.

We currently have in production:

- A NAIA adapter for ROOT's TSelector framework (NaiaTSelector)
- A common selection library (NSL)

And in the works:

- A ROOT-based spline fitting library (RSpline)
- A rewrite of the plugin system initially proposed for the dbar analysis

HAPPY CODING!