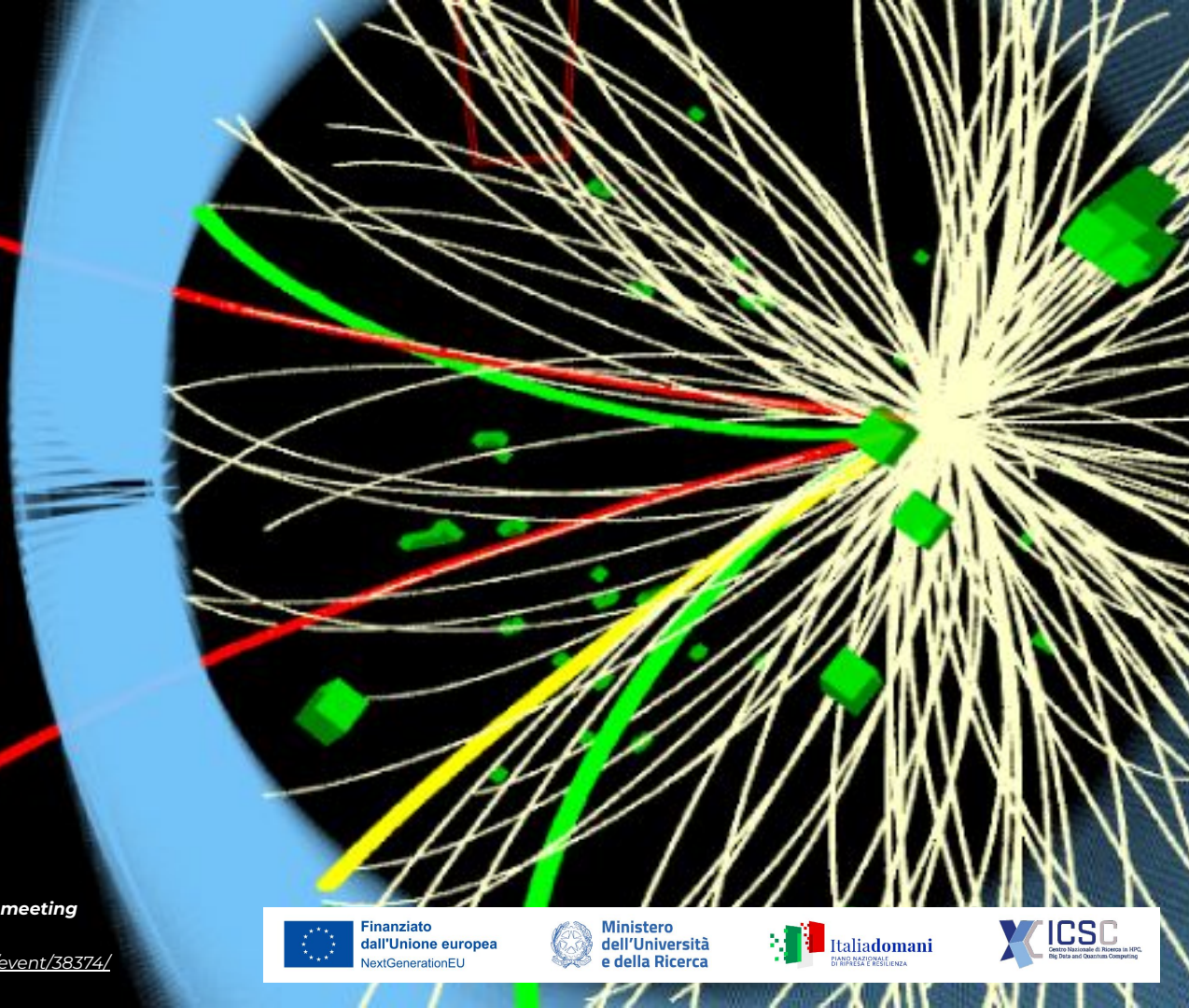
Declarative paradigms for analysis description and implementation

Alberto Annovi, Tommaso Boccali,
Paolo Mastrandrea, Andrea Rizzi



ICSC Spoke2 Annual meeting
CINECA - 20/12/2023
https://agenda.infn.it/event/38374/

Finanziato dall'Unione europea NextGenerationEU
Ministero dell'Università e della Ricerca
Italiadomani PIANO NAZIONALE DI RIPRESA E RESILIENZA
ICSC Centro Nazionale di Ricerca in HPC, Big Data and Quantum Computing

Declarative paradigm for analysis description & implementation

Main paradigm approaches

(from Wikipedia)

There are two main approaches to programming:[1]

- Imperative programming – focuses on how to execute, defines control flow as statements that change a program state.
- Declarative programming – focuses on what to execute, defines program logic, but not detailed control flow.

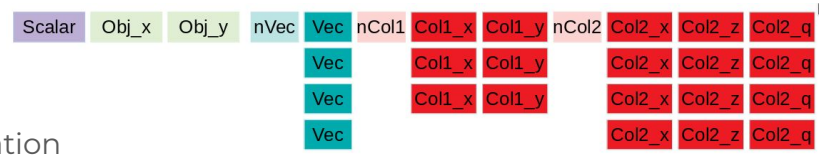
- So far mainly imperative paradigms have been used for analysis description and implementation
  - More straightforward application for “simple” tasks and linear/serial tools
- What has changed in the last decade?
  - HW: parallelism/multithreading
  - SW: more expressive programming languages (Python, C++ 17/20/23)
  - Tasks: increased complexity, increased data size (analyses, combinations)

- Benefits of a (***more***) **declarative paradigm**:
  - **Deeper decoupling** between algorithm and implementation
    - Faster analysis development
    - Wider portability of an analysis (different datasets/experiments)
    - Stronger preservation of the results
  - **Better scaling** of development and preservation for increasing complexity of the algorithms and size of the data
  - Better support for automatic (technical) optimisation
  - Simpler parallelization of the tasks
  - More flexibility: e.g. different backend processors

Activity and plans

- The development of analysis frameworks based on (more) declarative paradigms is growing momentum in the last years across the whole HEP community (e.g. Analysis Description Languages for the LHC)
- Activity in **ICSC-S2-WP2** - use case "*Quasi interactive analysis of big data with high throughput*" (more info in the talk by F. Gravili e T. Diotalevi)
  - Person power: 1+ from INFN-Pisa - new energies/ideas/feedbacks are always welcome
- Plan:
  - Build on modern and recent developments: ***NAIL*** (Natural Analysis Implementation Language - more details in backup)
    - Developed in the CMS collaboration, main developer Andrea Rizzi
    - Based on CMS' **nanoAOD** (columnar) data format, written in Python, heavy lift in C++ (RDataFrame)
  - Extend the **data-format interface**
    - Development, integration and validation (e.g. ATLAS' PHYSLITE)
  - Extend the framework to a **full analysis chain**
  - Test and optimisation phases will benefit from cutting-edge HW resources and community feedback

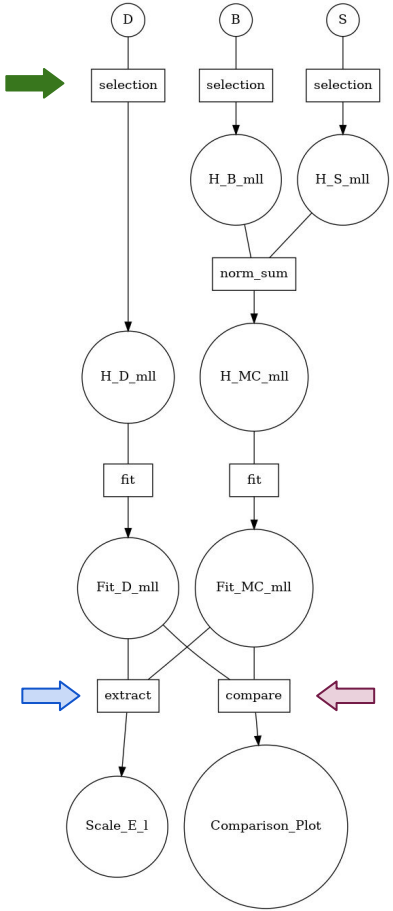
Data-format interface



- In principle 3 data-formats are in the game:
  - **a.** data-format used inside NAIL for variables manipulation
  - **b.** data-format used in the description of the algorithm by the user
  - **c.** data-format used in the ntuple to be processed
- In the first (“CMS-localized”) implementation **a** = **b** = **c**
- **c** is experiment dependent: a translation is needed **a** ↔ **c**
- Strategy:
  - Translation via a Python tool integrated in NAIL
  - Encode all the data-format specific information (and configurations needed) in a dictionary (JSON file)
- Status:
  - Demonstrator integration ongoing
  - Next step - extension to a simple columnar format (e.g. ATLAS’ open data)

Extension to full analysis chain

- Application of (more) declarative paradigms to the full analysis chain:
  - Sample preparation
  - **Event Loop (NAIL)**
  - Snapshot/data reduction
  - Combination / comparison of distributions
  - Statistical analysis / Extraction of results
  - Selective / incremental execution
- Risks assessment:
  - *"It's just another framework"* : Too specific / not general enough
  - *"It would be great if it worked"* : Too general / not customizable enough
- Status: development of a base structure, check for completeness
- Plan for demonstrator:
  - Single task prototype (e.g. Event Loop)
  - Incremental extension to other tasks

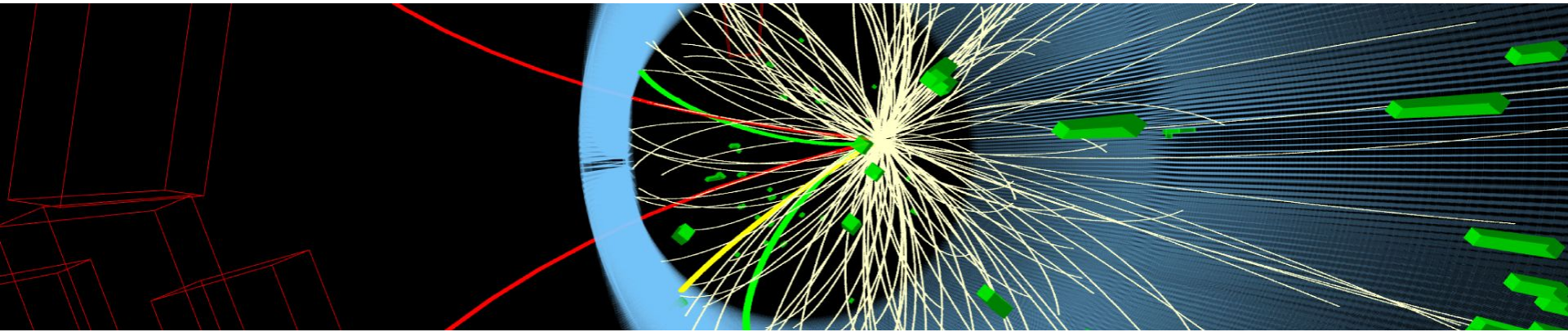


Example: (over-) simplified energy calibration scheme

Summary

- The application of (more) declarative paradigms in analysis description and implementation
  - Can boost analysis' speed (development and execution), preservation and portability
  - Is growing interest through the whole HEP community
- Plan:
  - Build on modern and recent developments (e.g. CMS' NAIL with nanoAOD data-format)
  - Extend the **data-format interface** to
    - Development, integration and validation (e.g. ATLAS' PHYSLITE)
  - Extend the framework to a **full analysis chain**
  - Test and optimisation phases will benefit from cutting-edge HW resources and community feedback
- Activity in **ICSC-S2-WP2** - use case "*Quasi interactive analysis of big data with high throughput*"
  - Person power: 1+ from INFN-Pisa - new energies/ideas/feedbacks are always welcome

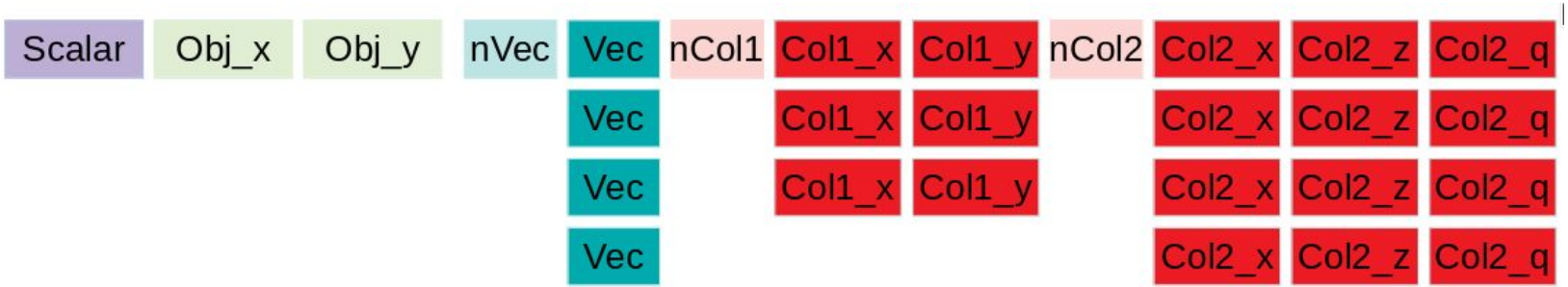
Backup



NAIL (Natural Analysis Implementation Language)

- *“NAIL is an analysis language that should allow to define in an abstract way a data analysis of a typical HEP experiment such as CMS or ATLAS. NAIL assumes an input data model for the event to process (...) and allow to specify the event by event processing actions in a declarative form. Analysis variations for optimizations and systematics do not need to be explicitly coded but are automatically derived from the event processing computational graph. Currently ROOT's RDataFrame is used as backend for a concrete implementation of the event processing as it allows parallelization and lazy evaluation.”* (from the README file of the NAIL package)

- Developed in the CMS collaboration, main developer Andrea Rizzi
- Based on CMS’ **nanoAOD** (*columnar*) data format, written in Python, heavy lift in C++ (RDataFrame)



AoS vs SoA

- From Wikipedia : *“In computing, an **array of structures (AoS)**, **structure of arrays (SoA)** … are contrasting ways to arrange a sequence of records in memory, with regard to interleaving, and are of interest in SIMD and SIMT programming.”*

AoS

```
1  struct point3D {
2      float x;
3      float y;
4      float z;
5  };
6  struct point3D points[N];
7  float get_point_x(int i) { return points[i].x; }
```

SoA

```
1  struct pointlist3D {
2      float x[N];
3      float y[N];
4      float z[N];
5  };
6  struct pointlist3D points;
7  float get_point_x(int i) { return points.x[i]; }
```

- CMS: SoA (e.g. nanoAOD)
- ATLAS: AoS interface with SoA memory storage (e.g. xAOD, PHYSLITE)

Where the increased speed comes from?

- **RVec**
  - *"A "std::vector"-like collection of values implementing handy operation to analyse them."*
  - Documentation
  - Optimised for **speed**
  - Its storage is contiguous in memory
  - **Automatic internal loop**

```
std::vector<float> goodMuons_pt;
const auto size = mu_charge.size();
for (size_t i=0; i < size; ++i) {
   if (mu_pt[i] > 10 && abs(mu_eta[i]) <= 2. &&  mu_charge[i] == -1) {
      goodMuons_pt.emplace_back(mu_pt[i]);
   }
}
```

```
auto goodMuons_pt = mu_pt[ (mu_pt > 10.f && abs(mu_eta) <= 2.f && mu_charge == -1) ]
```

Where the increased speed comes from?

- **RDataFrame**
  - *"ROOT's RDataFrame offers a modern, high-level **interface** for analysis of data stored in TTree , CSV and other data formats, in C++ or Python.*

    In addition, multi-threading and other low-level optimisations allow users to exploit all the resources available on their machines completely transparently."
  - Documentation
  - Optimised for **speed**
  - **Lazy** evaluation and **automatic internal loop**

```
TTreeReader reader("myTree", file);
TTreeReaderValue<A_t> a(reader, "A");
TTreeReaderValue<B_t> b(reader, "B");
TTreeReaderValue<C_t> c(reader, "C");
while(reader.Next()) {
   if(IsGoodEvent(*a, *b, *c))
      DoStuff(*a, *b, *c);
}
```

```
ROOT::RDataFrame d("myTree", file, {"A", "B", "C"});
d.Filter(IsGoodEvent).Foreach(DoStuff);
```