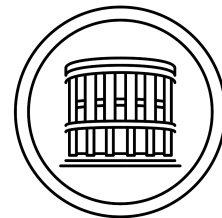


FastFrames - multithreaded physics analysis software at ATLAS

Michal Dubovsky
Comenius University in Bratislava

FastFrames development team:

Tomas Dado, Michal Dubovsky, Diego Baron, Ryan Quinn, Denys Timoshyn, Juan Minguez Laguna



COMENIUS
UNIVERSITY
BRATISLAVA

FastFrames

What it is:

- Framework for ntuple (and RNtuple) processing at analysis level
- Allows users to easily parallelize their analysis code
- Automatically takes care of systematic uncertainties

Useful links:

- Code is publicly available:
<https://gitlab.cern.ch/atlas-amglab/fastframes>
- Documentation:
<https://atlas-project-topreconstruction.web.cern.ch/fastframesdocumentation/latest/>

Disclaimer

- This is not going to be FastFrames tutorial or overview of its documentation.
- My goal is to show you how we designed code, what decision we made, what technologies we used and how we solved some challenging technical problems.
- The goal is to show people how to build a similar software.

FastFrames - why we need it

- With Run 3 of LHC we have a lot of data and Monte Carlo events. This takes a lot of storage space, and takes long to process.
- All modern computers have multiple CPU cores. If you efficiently split your workload among N cores, you will get the result N -times faster. We need efficient way to do this for analysis-level code.
- In order to save space, new format of ntuples have been adapted by ATLAS. There is only one detector-level tree with all the systematics in it.

Run 3 format of ntuples at ATLAS

- We have one detector-level TTree, with nominal and all systematic variations.
- Nominal branches have suffix “_NOSYS”.
- If systematic **s1** affects branch “xyz_NOSYS”, there is a branch “xyz_**s1**”
- Let's have two systematics:
 - “**JET_BJES_Response__1up**” affecting jets
 - “**EG_SCALE_ALL__1up**” affecting electrons
- If we want to calculate sum of pT of first jet and first electron:
 - $\text{sum_NOSYS} = \text{jet_pt_NOSYS}[0] + \text{el_pt_NOSYS}[0]$
 - $\text{sum_JET_BJES_Response_1up} = \text{jet_pt_JET_BJES_Response_1up}[0] + \text{el_pt_NOSYS}[0]$
 - $\text{sum_EG_SCALE_ALL_1up} = \text{jet_pt_NOSYS}[0] + \text{el_pt_EG_SCALE_ALL_1up}[0]$

Run 3 format of ntuples at ATLAS

- In practice, it's even more complicated, because 0th jet (or electron) does not have to pass selection for given systematics or nominal
- We need a way to automatically figure out which systematic affects which branches.
- The philosophy of FastFrames: You tell FastFrames how to define your variable for nominal (NOSYS), and FastFrames will figure out for you how to do it for all systematic uncertainties. You need to define just the list of systematic uncertainties you need (even that can usually be automated by running over all available systematics)

Key design requirements: input/output format

- Usual analysis framework on ATLAS:
 - Creating ntuples (**TopCPToolkit/Easyjet**)
 - Running your analysis code, producing histograms (**FastFrames**)
 - This is where most of the analysis-specific code lives
 - This is what FastFrames can be used for
 - In Run II, there was no common framework for this analysis stage.
 - Running fit or unfolding on your histograms (**TRExFitter**).

How to define analysis-specific things

- FastFrames uses **yaml config file**, which defines samples, regions, systematic uncertainties, variables and overall settings
- Event reconstruction and definition of variables is usually too complicated for config-based definition: We need **C++** for this.

Key design requirements: language and technologies

- Support for analysis specific code, which is supposed to be CPU intensive (event reconstruction). We need **C++** here for performance.
- **Python** is far more suitable to parse the config, check possible user's errors in it and resolve everything on the config side.
- We use Python for config reading, then it is interfaced with C++ via **boost** library.
- Actual input/output and multithreading is done with ROOT's **RDataFrame** class.
- Independent from another ATLAS software. The only dependencies:
 - ROOT
 - boost library
 - ONNX runtime (optional, for evaluating NN/BDT output)

RDataFrame: the heart of FastFrames

- FastFrames relies on **RDataFrame** for multithreading and input reading.
- Functional-like programming:
 - Define what are your inputs (vector<string> of branch names)
 - Define how your output should be calculated (std::function object, taking the branches as input arguments)
 - Define the name of the output.
- You can easily set number of CPU cores to be used.
- You do not need to care about the input reading (and thread-safety there). This is handled by RDataFrame

RDataFrame: minimal working example

- Minimal example on how to use RDataFrame in C++:
- You can find the example input file here: [link](#)
- The event loop is triggered when the output histogram is first used (lazy execution)

```
#include <TFile.h>
#include <TTree.h>
#include "ROOT/RDataFrame.hxx"

void run_example() {
    TFile* file = TFile::Open("data/data.root");
    TTree* reco_tree = (TTree*)file->Get("reco");

    ROOT::EnableImplicitMT(); // Enable multi-threading with 4 threads
    ROOT::RDataFrame df(*reco_tree);
    ROOT::RDF::SNode mainNode(df);
    mainNode = mainNode.Define(
        "pt_sum_leading_2_jets", // will add this branch to the dataframe
        [] (const std::vector<float>& jet_pt) -> float { // lambda function to compute the new branch
            if (jet_pt.size() < 2) return 0.0f;
            return 0.001 * (jet_pt[0] + jet_pt[1]);
        },
        {"jet_pt_NOSYS"} // list of branches it depends on
    );

    // define histogram
    auto h_pt_sum_leading_2_jets = mainNode.Hist1D(
        {"h_pt_sum_leading_2_jets", "Sum of leading 2 jets pT; pT [GeV]; Events", 10, 0, 1000},
        "pt_sum_leading_2_jets"
    );

    // save histogram to file
    TFile outputFile("output.root", "RECREATE");
    h_pt_sum_leading_2_jets->Write();
    outputFile.Close();
}
```

Key design requirements: analysis-specific C++ code

- For any real analysis, you will need also your own C++ code:
 - Reconstructing non-trivial observables
 - Running NN inference
 - Really, doing anything that cannot be expressed via simple formula in config
- How to solve this in C++:
 - Have a base class ([MainFrame](#)) in FastFrames, make some of its methods virtual, so that user can override them, defining his own physics variables.
 - Read and parse the config using the actual FastFrames code, then instantiate the user's derived class and use it.

Key design requirements: C++ inheritance

- **We don't want users having to change FastFrames.** FastFrames cannot depend on analysis specific code.
- Typical C++ factory pattern would not work:

```
#include "MyBaseClass.h"
#include "MyDerivedClassA.h" // <- It would not work without including this header
std::unique_ptr<MyBaseClass> myAwesomeFactoryFunction(const std::string& type) {
    if (type == "BaseClass")      { return std::make_unique<MyBaseClass>(); }
    if (type == "DerivedClassA") { return std::make_unique<MyDerivedClassA>(); }
    throw std::runtime_error("Type not supported");
}
```

Key design requirements: C++ inheritance

What we do instead:

- Compile custom class (analysis specific code) and add path to its library to `LD_LIBRARY_PATH` . The compiled shared object has a name `"lib<your custom frame name>.so"`

Adding path to analysis-specific library to system path:

```
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/home/dubovsky/Analysis/ttZ_boosted/customfastframesclasstz/install/lib
```

- When running FF, user specifies his class name in config. FF will then load the shared object at the runtime, instantiating user's class instead of the default FastFrames one

The actual C++ code ([link](#)):

```
std::unique_ptr<MainFrame> baseFrame(nullptr);
const std::string& name = m_config->customFrameName();

if (!name.empty()) {
    // read the custom code for adding variables
    LOG(INFO) << "Using custom class: " << name << "\n";
    gSystem->Load(("lib" + name).c_str());

    TClass* c = ::TClass::GetClass(name.c_str());
    if (!c) {
        LOG(ERROR) << "Cannot get class " << name << "\n";
        throw std::invalid_argument("");
    }
    baseFrame.reset(static_cast<MainFrame*>(c->New()));
    if (!baseFrame) {
        LOG(ERROR) << "Something went wrong with creating the custom class\n";
        throw std::invalid_argument("");
    }
} else {
    baseFrame = std::make_unique<MainFrame>();
}
```

Interfacing Python and C++

- We use boost library for the interface. We need objects (classes) in the whole interface: for example **Sample** object contains list of **Systemtic** objects. We need to configure/access this from python.
- How we do it:
 - For each class there is a wrapper class in the C++ part of interface
 - The wrapper contains **std::shared_ptr** to the actual instance. Each wrapper defines method for getting pointer to its **std::shared_ptr** as **uint64**
 - For example: when adding **Systemtic** object to **Sample** object:
 - We get pointer to Systematic's instance **std::shared_ptr** as **uint64**, and pass it to **SampleWrapper::addSystematic** method [[link](#)]
 - **SampeWrapper** object reinterprets it as ***shared_ptr<Systemtic>** [[link](#)]
 - Dereferencing the pointer we get **shared_ptr<Systemtic>** to the actual systematics that we can use in C++ part [[link](#)]

Conclusions:

- We managed to develop efficient and easy to use multithreaded analysis framework
- The development started in late 2023, we have 6 people in the development team.
- Exact number of users is hard to estimate, but it is at least few hundreds:
 - 325 people in Mattermost support channel
 - 113 forks on gitlab
- ROOT's **RDataFrame** makes multithreaded analysis code easy to implement.
- Combination of C++ (CPU intensive stuff) and python (config parser, scripting) has proven to be a great choice of languages for such software.

Unfolding and memory issues

Key design requirements

- FastFrames must be able to work with common ntuple formats used by ATLAS (produced by EasyJet and TopCPToolkit)
- It must produce a format which can be directly used in fit or unfolding step by TRexFitter. There is even a script for producing TRexFitter config file for your fit or unfolding.
- It must be customizable for analysis-specific needs. This includes:
 - Config file
 - Your own C++ code (class inheriting from base class defined in FastFrames)

Key design requirements: config

- Most of the things can be set from config file
 - This includes both FastFrames general settings, as well as your own settings that you may need in your own C++ code (let's say you want to set a b-tagging working point from config)
 - Config file is a standard yaml file.
 - You can define here binning and names of your histograms, list and names of samples, systematic uncertainties, regions etc.