CMS data analysis from run 2 towards HL-LHC

Challenges, lessons learnt, workflows, and tools

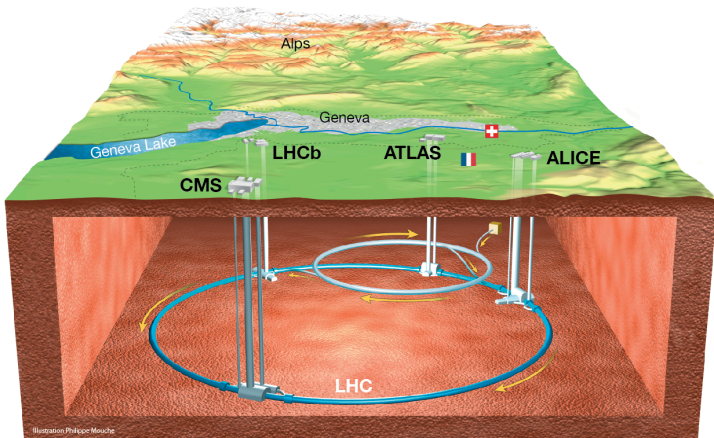Pieter David

Université catholique de Louvain

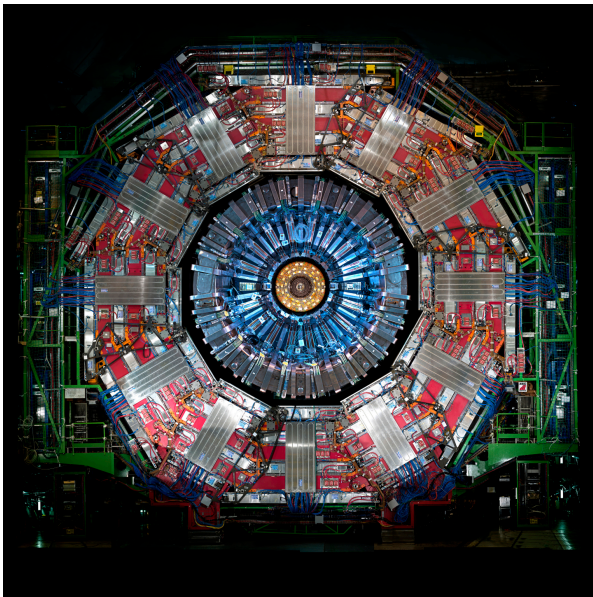CP3 seminar

28 September 2021

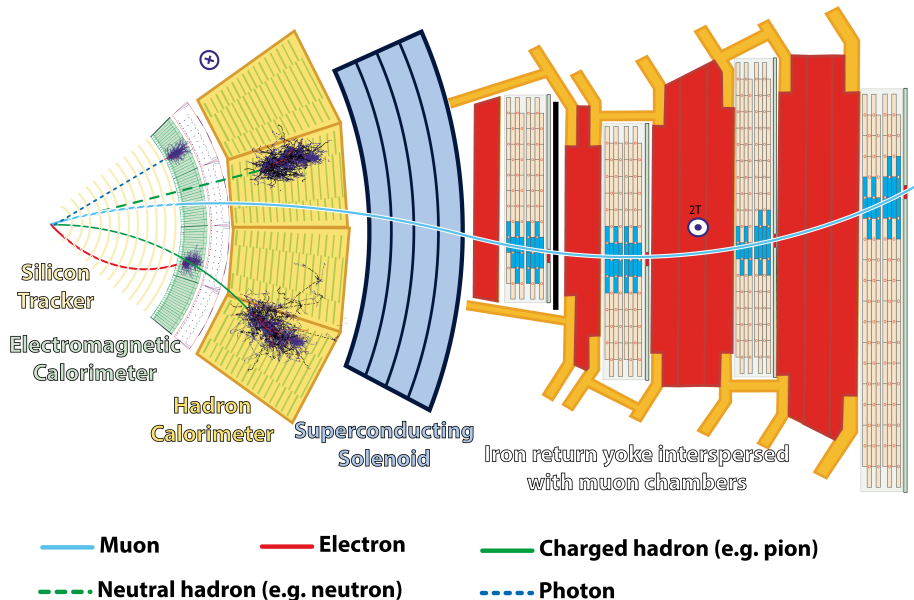# In reality more like this, at 40MHz

CMS Experiment at the LHC, CERN
Data recorded: 2016-Sep-08 08:30:28.497920 GMT
Run / Event / LS: 280327 / 55711771 / 67
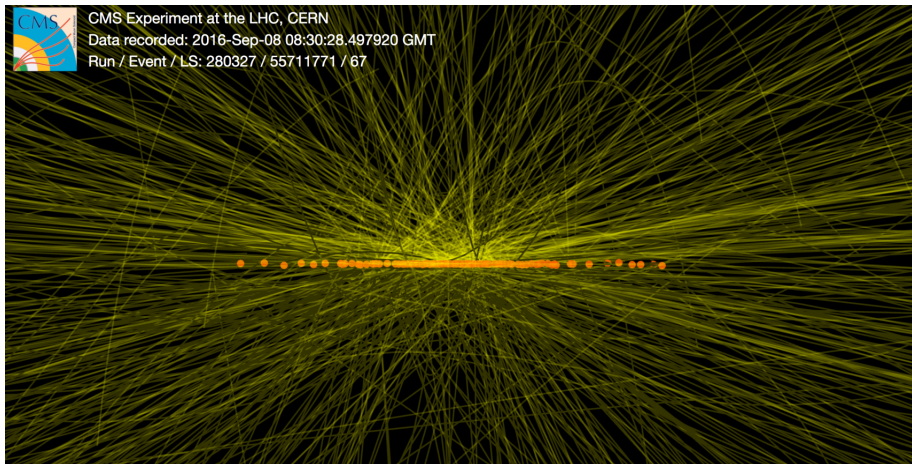
From raw data back to the analysis picture of a collision

- Trigger: 40MHz collision rate to 2kHz stored, by selecting signatures of interest (high-p_T leptons, jets, missing energy *etc.*)

- Track and primary vertex reconstruction

- Particle flow jet reconstruction, with pileup suppression

- Quality requirements, lepton isolation, jet tagging (initial parton or substructure)

CMS Experiment at the LHC, CERN
Data recorded: 2016-Oct-11 10:44:24.059904 GMT
Run / Event / LS: 282842 / 47118579 / 25
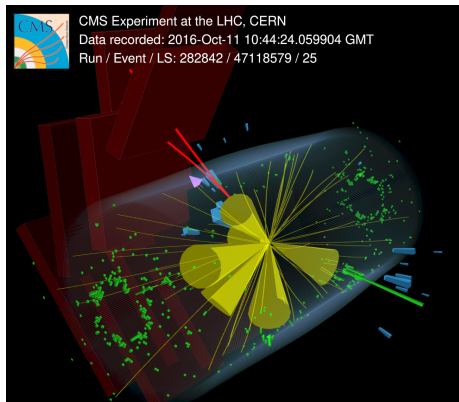
interactive version

After all of these steps (each a major challenge and research area by itself) a recorded event looks more like the common picture of a high-energy collision again: a final state with leptons, jets, photons, missing energy *etc.*

Final physics analysis in a nutshell

Typical goals: search for a new or measure a known process, or measure the distribution of one or more processes in specific kinematic variables (absolute or relative to a better known process)

1. Starting point: well-understood parts of the full data sample (*e.g.* passing specific triggers, with signatures where the main backgrounds are well-described)

2. Enhance the signal and sensitivity with additional requirements, categorisation, higher-level variables (*e.g.* machine learning, kinematic reconstruction)

3. Model the response of the signal and expected backgrounds to this procedure (often through Monte Carlo simulations), including systematic uncertainties

4. Fit a statistical model that parameterizes the yield(s) (or their ratios) as a function of the parameter(s) of interest to extract the results, test hypotheses

Optimisation of choices made at each step to obtain the best sensitivity

Practical challenges for data analysis

- Large data samples: billions of events. Data format used for analysis in CMS: from 3 MB (2010–2011), 400 kB (2012–2015), 50 kB (2016–2019) to 1 kB/event

- Simulation is frozen before all calibrations are done, so needs a number of corrections for the best description of the data

- Systematic variations: change input X and compare the output

- Time pressure, need to adapt to upstream changes and improvements, internal review

LHC / HL-LHC Plan

No more big jumps in energy, so enhanced sensitivity must come from more data, precision, combinations, additional final states, and new ideas
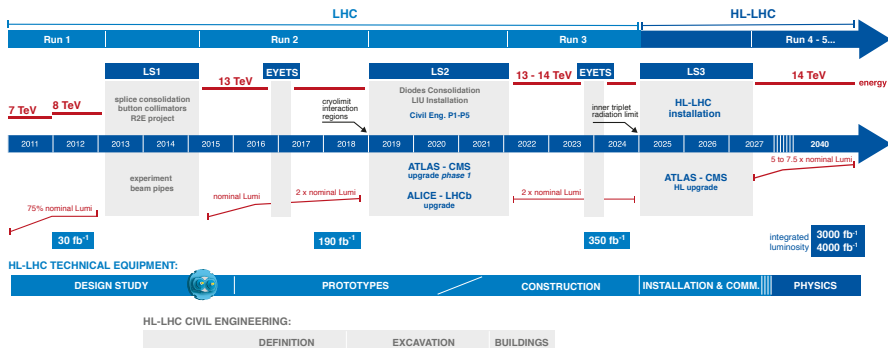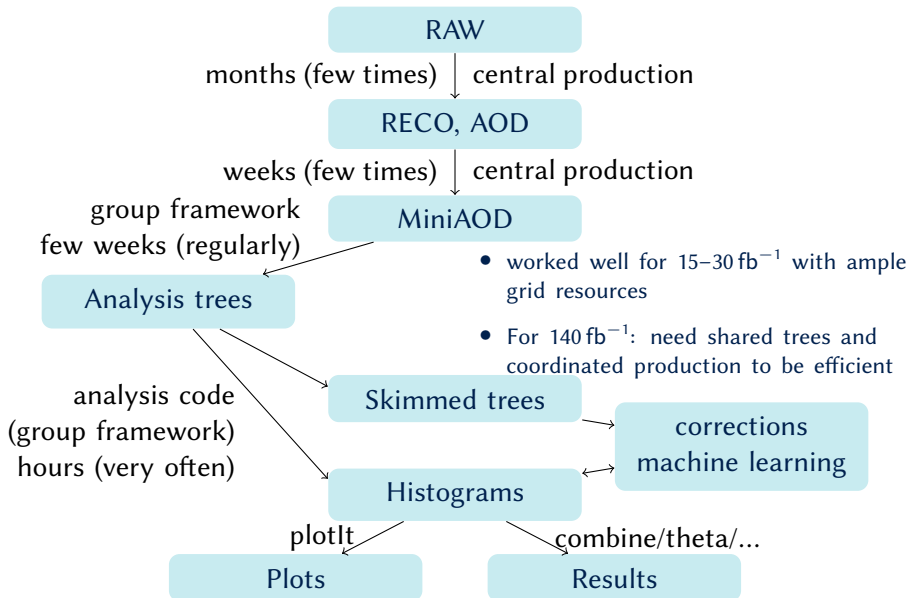
Practical challenges for data analysis

- Large data samples: billions of events. Data format used for analysis in CMS: from 3 MB (2010–2011), 400 kB (2012–2015), 50 kB (2016–2019) to 1 kB/event

- Simulation is frozen before all calibrations are done, so needs a number of corrections for the best description of the data

- Systematic variations: change input X and compare the output

- Time pressure, need to adapt to upstream changes and improvements, internal review

- Sample sizes $\times$ 2–3 in Run 3, $\times$ 20 in high-lumi, modest increase in energy (13 to 14 TeV), and must combine data-taking years

- Calibrations and corrections become increasingly accurate — and complex, *e.g.* time-dependent

- More and more systematic variations (50–100 is typical nowadays)

- Analyses become more complex, and more prior knowledge is required, but PhD and postdoc contracts stay the same duration

Typical analysis workflow 2016–2018

RAW

months (few times) | central production

RECO, AOD

weeks (few times) | central production

MiniAOD

group framework
few weeks (regularly)

Analysis trees

- worked well for 15–30 fb^{-1} with ample grid resources

- For 140 fb^{-1}: need shared trees and coordinated production to be efficient

Skimmed trees

analysis code
(group framework)
hours (very often)

Histograms

corrections
machine learning

plotIt

combine/theta/...

Plots

Results

Typical analysis workflow after run 2

RAW

months (few times) | central production

RECO, AOD

weeks (few times) | central production

group framework
few weeks (regularly)

MiniAOD

central production
1 month (regularly)

Analysis trees

NanoAOD

analysis code
(group framework)
hours (very often)

Skimmed trees

corrections
machine learning

Histograms

plotIt

combine/theta/...

Plots

Results

Typical analysis workflow after run 2

RAW

months (few times) | central production

RECO, AOD

weeks (few times) | central production

group framework
few weeks (regularly)

MiniAOD

central production
1 month (regularly)

Analysis trees

NanoAOD

bamboo

analysis code
(group framework)
hours (very often)

Skimmed trees

corrections
machine learning

Histograms

plotIt

combine/theta/...

Plots

Results

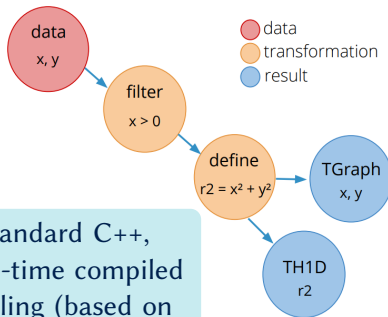# RDataFrame: ROOT's declarative data analysis approach

Analyses as computation graphs

```
ROOT::RDataFrame df(dataset);

auto df2 = df.Filter("x > 0")
            .Define("r2", "x*x + y*y");

auto rHist = df2.Histo1D("r2");

auto g = df2.Graph("x","y")
```

data
x, y

filter
x > 0

define
r2 = x² + y²

TGraph
x, y

TH1D
r2

○ data
○ transformation
○ result

any standard C++,
just-in-time compiled
with cling (based on
LLVM and clang)

E. Guiraud, "RDataFrame", ROOT users' workshop 2018

18

from this talk

An analysis framework that lets the user write compact high-level code, and takes care of executing it efficienctly in different scenarios

Depending on the perspective:

- a set of tools to efficiently build RDF graphs (JIT-compiled)
- an embedded domain-specific language for producing plots, skims *etc.* (compact code, similar to analysis description languages)

Design principles and goals:

1. avoid the black box effect: the user makes all the choices that affect the results (but there are helpers and shared tools for many common things)
2. analysis code should be as simple and compact as possible (given 1)
3. be as fast as possible (main target: turnaround time for plots using the local T2 batch system)

Feedback from the early adopters (Sébastien, Khawla, Florian, Gourab...) has been very important to reach the current state, thanks a lot!

What is bamboo?   <code> code   📦 docs   2103.01889

An analysis framework that lets the user write compact high-level code, and takes care of executing it efficienctly in different scenarios

Depending on the perspective:

- a set of tools to efficiently build RDF graphs (JIT-compiled)
- an embedded domain-specific language for producing plots, skims *etc.* (compact code, similar to analysis description languages)

Building blocks:

- A data-based *event* representation (for NanoAOD and other tree formats)
- Functional-style helpers to build complex expressions (embedded domain-specific language approach to analysis description languages)
- RDataFrame+cling+PyROOT
- some python glue to keep it all together, and implement common recipes

Feedback from the early adopters (Sébastien, Khawla, Florian, Gourab...) has been very important to reach the current state, thanks a lot!

Building an analysis with bamboo: code

- build expressions from the decorated NanoAOD and helper functions

```
muons = op.select(t.Muon, lambda mu : op.AND(mu.mediumId,
    mu.pfRelIso03_all < 0.4, mu.pt > 15., op.abs(mu.eta) < 2.4))
leadMuPt = muons[0].pt
dimu = op.combine(muons, N=2, lambda m1,m2 : m1.charge != m2.charge)
ll = dimu[0]
m_ll = (ll[0].p4 + ll[1].p4).M()
cleanedBJets = op.select(t.Jet, lambda j : op.AND(
    op.NOT(op.rng_any(muons, lambda m : op.deltaR(m.p4, j.p4) <0.3)),
    j.bTag > 0.6))
```

Building an analysis with bamboo: code

- build expressions from the decorated NanoAOD and helper functions

```python
muons = op.select(t.Muon, lambda mu : op.AND(mu.mediumId,
    mu.pfRelIso03_all < 0.4, mu.pt > 15., op.abs(mu.eta) < 2.4))
dimu = op.combine(muons, N=2, lambda m1,m2 : m1.charge != m2.charge)
ll = dimu[0]
m_ll = (ll[0].p4 + ll[1].p4).M()
```

- cut flow: define `Selection` objects by adding cuts and weights to the parent selection (starting from all events in the input, weight 1)

```python
if isMC(sample):
    baseSel = baseSel.refine("mcWeight", weight=t.genWeight)
diMuSel = baseSel.refine("hasDimu", cut=( op.rng_len(dimu) >= 1 ))
```

Building an analysis with bamboo: code

- build expressions from the decorated NanoAOD and helper functions

```python
muons = op.select(t.Muon, lambda mu : op.AND(mu.mediumId,
    mu.pfRelIso03_all < 0.4, mu.pt > 15., op.abs(mu.eta) < 2.4))
dimu = op.combine(muons, N=2, lambda m1,m2 : m1.charge != m2.charge)
ll = dimu[0]
m_ll = (ll[0].p4 + ll[1].p4).M()
```

- cut flow: define `Selection` objects by adding cuts and weights to the parent selection (starting from all events in the input, weight 1)

```python
if isMC(sample):
    baseSel = baseSel.refine("mcWeight", weight=t.genWeight)
diMuSel = baseSel.refine("hasDimu", cut=( op.rng_len(dimu) >= 1 ))
```

- plots: construct `Plot` objects from axis variable(s) and a `Selection`

```python
plot = Plot.make1D("dimuM", m_ll, diMuSel, EqB(100, 20., 120.))
```

Building an analysis with bamboo: code

- build expressions from the decorated NanoAOD and helper functions

```python
muons = op.select(t.Muon, lambda mu : op.AND(mu.mediumId,
    mu.pfRelIso03_all < 0.4, mu.pt > 15., op.abs(mu.eta) < 2.4))
dimu = op.combine(muons, N=2, lambda m1,m2 : m1.charge != m2.charge)
ll = dimu[0]
m_ll = (ll[0].p4 + ll[1].p4).M()
```

- cut flow: define `Selection` objects by adding cuts and weights to the parent selection (starting from all events in the input, weight 1)

```python
if isMC(sample):
    baseSel = baseSel.refine("mcWeight", weight=t.genWeight)
diMuSel = baseSel.refine("hasDimu", cut=( op.rng_len(dimu) >= 1 ))
```

- plots: construct `Plot` objects from axis variable(s) and a `Selection`

```python
plot = Plot.make1D("dimuM", m_ll, diMuSel, EqB(100, 20., 120.))
```

- wrap this up in a module

```python
class DimuonPlots(NanoAODHistoModule):
    def definePlots(self, t, noSel, sample=None, sampleCfg=None):
        ## analysis code
        return plots ## list of Plot objects
```

From this code to stack plots

```
bambooRun -m dimu.py:DimuonPlots dimu_example.yml -o test_dimu1
```

From this code to stack plots

```
bambooRun -m dimu.py:DimuonPlots dimu_example.yml -o test_dimu1
```

- Samples (input files, name, scaling) are defined in a YAML file, e.g.

```
samples:
  DY_M10to50_2017:
    group: DY
    era: "2017"
    db: "das:/DYJetsToLL_M-10to50_TuneCP5_13TeV-madgraphMLM-pythia8/..."
    cross-section: 18610
    generated-events: 'genEventSumw'
    split: 2
```

From this code to stack plots

```
bambooRun -m dimu.py:DimuonPlots dimu_example.yml -o test_dimu1
```

- Samples (input files, name, scaling) are defined in a YAML file, e.g.

```yaml
samples:
  DY_M10to50_2017:
    group: DY
    era: "2017"
    db: "das:/DYJetsToLL_M-10to50_TuneCP5_13TeV-madgraphMLM-pythia8/..."
    cross-section: 18610
    generated-events: 'genEventSumw'
    split: 2
```

- `definePlots` gets all sample metadata, so can adjust the graph (scalefactors, corrections *etc.*); it is only called once per sample (or batch job) to construct the RDF graph, which then runs at compiled speed

From this code to stack plots

```
bambooRun -m dimu.py:DimuonPlots dimu_example.yml -o test_dimu1
```

- Samples (input files, name, scaling) are defined in a YAML file, e.g.

```yaml
samples:
  DY_M10to50_2017:
    group: DY
    era: "2017"
    db: "das:/DYJetsToLL_M-10to50_TuneCP5_13TeV-madgraphMLM-pythia8/..."
    cross-section: 18610
    generated-events: 'genEventSumw'
    split: 2
```

- `definePlots` gets all sample metadata, so can adjust the graph (scalefactors, corrections *etc.*); it is only called once per sample (or batch job) to construct the RDF graph, which then runs at compiled speed
- by default combined in a stack plot with plotIt, but easy to customize

From this code to stack plots

```
bambooRun -m dimu.py:DimuonPlots dimu_example.yml -o test_dimu1
```

- Samples (input files, name, scaling) are defined in a YAML file, e.g.

```
samples:
  DY_M10to50_2017:
    group: DY
    era: "2017"
    db: "das:/DYJetsToLL_M-10to50_TuneCP5_13TeV-madgraphMLM-pythia8/..."
    cross-section: 18610
    generated-events: 'genEventSumw'
    split: 2
```

- `definePlots` gets all sample metadata, so can adjust the graph (scalefactors, corrections *etc.*); it is only called once per sample (or batch job) to construct the RDF graph, which then runs at compiled speed
- by default combined in a stack plot with plotIt, but easy to customize
- Running on a batch system, implicit multi-threading, verbose logging, only the plotting step... configured through command-line arguments

What goes on behind the scenes

- Decorated tree attributes (collections, groups), collection items *etc.* are in fact proxy objects that behave like the value they refer to, but instead build up the expression graph. The intermediate results (e.g. a filtered list, physics object, combination, which are conceptually close to how we talk and think about analysis) can be used freely in the analysis code.
- When adding selections and plots, the cuts, weights, and axis variables (and expensive intermediate results) are defined as columns in the RDF

This may sound a bit heavy, but it works well so far, and has advantages:

- exploit C++ JITting from python analysis code
- automation by transforming expressions: <u>systematics uncertainties without code duplication</u>. Technically, <u>inputs</u> (weight, scalefactor, kinematics...) are marked as having systematic variations, and producing the histogram with the alternative value of this in any of the cuts/weights/axis variables it needs, can be done automatically.
- in principle this allows for optimisations of the graph, and conversion to other formats or backends

Other available ingredients

- Data-driven background estimates (recipe)

- Printout or LaTeX table of cutflow (recipe)

- rerunning the combination step on previously produced histograms

- running on slurm or HTCondor (and recovering failed jobs)

- MVA inference with Tensorflow, TMVA, torchscript, lwtnn, and ONNX-Runtime, including easy production of dataframes for training

- on-demand calculation of various corrections (lepton and jet energy scale, propagated to missing transverse momentum)

- many customisation points: analysis-specific command-line switches, other types of plots *etc.*; loading C++ extensions from user code is also possible (as in ROOT)

- Git integration (optional): keep track of versions used, remind to commit

Analysis workflows with bamboo

- Not the only option, but most users run bamboo on centrally produced NanoAOD, with on-demand calculation of corrections (fast standalone C++ implementation of what we need): this eliminates the grid jobs to obtain a format for analysis, and gives a lot more flexibility (also interesting for deriving corrections and calibrations)

- Analysis preservation: a bamboo analysis is a collection of python classes, a few YAML files (samples), and commands to chain the steps. bamboo is <u>installed</u> with pip (so also easy in conda or <u>docker</u>), and requires only a recent ROOT and python3 with a few standard packages. The main challenge to running on CI is fast enough data access (we use locally available files, xrootd is the fallback option, but may be slow)

- (biased opinion) bamboo encourages compact, simple and, therefore, readable analysis code

- Performance: small overhead from python code in typical use cases, acceptable turnaround times (see <u>this talk</u> for some more details)

Performance and phenomenology studies, open data etc.

- Side-effect of building a tool that makes the code for complex analyses manageable: technically less complicated analyses can become really compact

- Example Run1 open data analyses (CMS H $\rightarrow 4\ell$, ATLAS W production) in this repository

- Also used for several Snowmass studies using Delphes simulation (HH production, by Suat, Sandhya, Paola, and Angela) — these are fast-moving analyses, so a good opportunity to streamline the process and improve the documentation

$\rightarrow$ Demo (time permitting)

Conclusions

- Many challenges ahead to exploit the increasingly large data samples collected at the LHC

- Innovation in analysis tools is required to be efficient: many ideas and final states to explored in limited time

- Several steps towards that in recent years: NanoAOD, RDataFrame in ROOT. At CP3: developed bamboo, a python framework to take advantage of these, building on experience from the past

- Declarative style (no *event loop*) is a big change, but allows to rethink and naturally define an "analysis description language" that is easy to write, with performance close to compiled C++

- Complete enough for many CMS analysis use cases, with more developments planned. Extensive documentation, but feel free to reach out when considering a new use case

- More details in 2103.01889 (vCHEP2021 conference paper)

Additional material

Decorated trees and expressions versus plain RDataFrame

With split collections (`Muon_pt[nMuon]` *etc.*) a lot of bookkeeping of indices can be done automatically. As an example, without the builtin helper method, the RDataFrame code to calculate a dimuon invariant mass (without any selection) on NanoAOD would be something like

```cpp
using ROOT::Math::VectorUtil::InvariantMass;
using LV = ROOT::Math::LorentzVector<ROOT::Math::PtEtaPhiM4D<float>>;
df.Define("Dimuon_mass",
  [] (const auto& pt, const auto& eta, const auto& phi, const auto& m) {
    return InvariantMass(LV(pt[0], eta[0], phi[0], m[0]),
                         LV(pt[1], eta[1], phi[1], m[1]));
  }, {"Muon_pt", "Muon_eta", "Muon_phi", "Muon_mass"}
  ).Histo1D(..., "Dimuon_mass", ...);
```

or, using the JIT instead of fully compiled code,

```cpp
df.Define("Dimuon_mass_v2",
  "InvariantMass("
    "LorentzVector(Muon_pt[0], Muon_eta[0], Muon_phi[0], Muon_mass[0]),"
    "LorentzVector(Muon_pt[1], Muon_eta[1], Muon_phi[1], Muon_mass[1]))"
  ).Histo1D(..., "Dimuon_mass_v2", ...);
```

Performance

- bamboo uses mostly JITted code, so some overhead is expected — so far acceptable, in return for simpler analysis code

- No detailed benchmarking done so far, but speed is in the target range: turnaround of a few hours for $\mathcal{O}(100)$ plots (thousands of histograms) of the CMS Run2 data on a batch system

- Memory usage has been a bigger worry, but ROOT 6.22/00 brought a huge improvement (factor 3–5), details in this forum thread

- Implicit multi-threading mostly "just works", can be useful in case of large graphs or a lot of calculation (otherwise I/O and decompression dominate)

- Filling a list of histograms with the same value but different weights is a common pattern, some repetition (bin lookup) could be avoided there

- Is it possible to evaluate an MVA on inputs from a batch of events? More generally: how is, or could, vectorisation be used?

bamboo ingredients

- High-level python analysis code to define plots and selections (using loops, higher-order functions etc.)

- Decorated version of the input `TTree`: an event looks like a set of containers of physics objects (jets, leptons, tracks etc.) and (groups of) per-event quantities

- Expressions (selection, weight, variable) are composed of simple python objects, built from decorators, and decorated to behave as a value (to construct derived expressions)

- When the analysis is complete: convert expressions to strings for RDataFrame, run over all samples, and make plots

- Every analysis derives from a base class, such that e.g. splitting in batch jobs, and plotting code can be reused

A python layer on top of RDataFrame: motivation

- Typical LHC analysis: a number of selection and slimming steps to go from fully reconstructed triggered events to reduced `TTrees`, then: lots of histograms at different selection stages, MVAs, some statistical analysis, and results

- Code needs to be *very flexible*, and allow to keep a good overview: strong case for using python

- Data sets are large (run 2: several TB in reduced formats), so code becomes slow... try to make python faster? Use C++ instead?

How much time and effort to:

- plot one more distribution

- change a selection

- change a selection, and compare N plots between the two cases

- add a correction that a) is a per-event weight, or b) changes object kinematics, and/or c) tracks changing detector conditions — only for simulation, not for data

- add a higher-statistics sample that covers part of the phasespace of an already included one

- include systematic variations

A python layer on top of RDataFrame: motivation

- Typical LHC analysis: a number of selection and slimming steps to go from fully reconstructed triggered events to reduced `TTrees`, then: lots of histograms at different selection stages, MVAs, some statistical analysis, and results

- Code needs to be *very flexible*, and allow to keep a good overview: strong case for using python

- Data sets are large (run 2: several TB in reduced formats), so code becomes slow... try to make python faster? Use C++ instead?

Personal experience: need for speed makes analysis code messy (hard to find bugs), inflexible, or both

not see the wood for the trees
UK (US **not see the forest for the trees**)

to be unable to get a general understanding of a situation because you are too worried about the details

*(Definition of **not see the wood for the trees** from the **Cambridge Advanced Learner's Dictionary & Thesaurus** © Cambridge University Press)*

but with modern ROOT (RDataFrame + cling + PyROOT), an event loop can be built declaratively, with compiled code, from python — so this performance versus readability and flexibility tradeoff can be avoided

A python layer on top of RDataFrame: motivation

- Typical LHC analysis: a number of selection and slimming steps to go from fully reconstructed triggered events to reduced `TTrees`, then: lots of histograms at different selection stages, MVAs, some statistical analysis, and results

- Code needs to be *very flexible*, and allow to keep a good overview: strong case for using python

- Data sets are large (run 2: several TB in reduced formats), so code becomes slow... try to make python faster? Use C++ instead?

not see the wood for the trees
UK (US **not see the forest for the trees**)

to be unable to get a general understanding of a situation because you are too worried about the details

*(Definition of **not see the wood for the trees** from the **Cambridge Advanced Learner's Dictionary & Thesaurus** © Cambridge University Press)*

image credit: Claudio Caputo

bamboo is an attempt to turn this idea into a framework usable for analysis of CMS NanoAODs (and similar formats)

Implementation: tree decorations

- Tree proxy class generated on the fly, based on the branches that are found
- By default, each branch is an attribute of the tree proxy (the class is generated with `type()`)
- Groups of non-array branches: "group proxy" in between: `t.HLT.MuXX`, `t.pdf.x1`
- Groups of array branches: container proxy, and a proxy for the elements: `t.Muon[0].IDLoose`
- Can also add references and arbitrary functions: `t.Jet[0].Mu1.pt`, `t.Muon[0].p4.E()`

- Needs to be adapted to recognize different tree formats, but for *flat trees* (most common) this is fairly straightforward (examples are from CMS NanoAOD, one other format is implemented)

image credit

Implementation: expressions and proxies

Expressions

- are composed of simple python objects, e.g. `t.Muon[0].pt` (`Muon_pt[0]`) becomes `GetItem(GetArrayLeaf("Muon_pt"),0)`

- can be converted to a string for RDataFrame/JIT

- are considered immutable as soon as they are fully constructed and passed around (but a fresh clone can be modified by the owner)

Proxies

- Wrap an expression

- Emulate the value type of expression's result (through python operator overloading and other magic methods)

- float-like, integer-like, object-like, and a few list-like classes — but no complete type system (yet), so limited checks at construction

Currently each of these interfaces has about 25 implementations – the user should only need the decorated tree and the `bamboo.treefunctions` module

Selections and plots

Zooming in on the currently main use case of different selections and histograms now (skims also work)

- This only needs two fundamental RDataFrame actions: `Filter` and `Histo{1D,2D}` (and `Define`, to calculate intermediate values)

- Important distinction: `Filter` changes control flow, whereas the others do not — so there is some freedom in ordering the `Define` nodes (in between the `Filter` that makes sure the expression is valid and the first use)

Current solution (`bamboo.plots`):

- `Selection` class, with each instance (optionally) holding a set of selection requirements (cuts) and weight factors

- Selections are defined by adding cuts or weights to a more inclusive selection (starting point: all events in the input, unit weight)

- `Plot` instances are defined by a `Selection`, variable(s), binning(s), and layout options

- RDataFrame nodes are created when `Selection` and `Plot` objects are constructed

Extending the basic functionality

Written in python, and tried to keep things loosely coupled (interfaces), so many things are straightforward to customise and extend:

- Loading additional C++ headers and libraries in the interpreter
  Examples: good runs/events filter and scale factors from JSON files,
  jet and muon energy scale corrections calculated on the fly

- Alternative analysis (base) classes, e.g. for different tree formats, to customise plotting, or to calculate efficiencies in addition

- There is a hook to specify additional command-line arguments from the analysis module

- The sample definition (YAML) is open-ended, the base class only looks at the attributes it needs (e.g. input files, to do the job splitting), and the plotting library at a few more (normalisation for MC, grouping and ordering, colors...)

Structure of a bamboo analysis module

```python
from bamboo.analysismodules import NanoAODHistoModule
class DimuonPlots(NanoAODHistoModule):
    def definePlots(self, t, noSel, sample=None, sampleCfg=None):
        from bamboo.plots import Plot
        from bamboo.plots import EquidistantBinning as EqB
        from bamboo import treefunctions as op
        if self.isMC(sample):
            noSel = noSel.refine("mcWeight", weight=t.genWeight)
        plots = []
        muons = op.select(t.Muon, lambda mu : op.AND(mu.mediumId,
                            mu.pfRelIso03_all < 0.4, mu.pt > 15.))
        twoMuSel = noSel.refine("has2mu", cut=( op.rng_len(muons) > 1 ))
        plots.append(Plot.make1D("dimuM", (muons[0].p4+muons[1].p4).M(),
            twoMuSel, EqB(100, 20., 120.), title="Invariant mass"))
        return plots
```

RResultPtrs to all histograms collected in the base class, then filled
Selection represents a subsample (Filter node), with a weight column;
constructed with parent.refine(name, cut=..., weight=...)
Plot (Histo1D node): name, variable, binning, Selection, and options

Building expressions from decorated flat `TTrees`

- Decorations group related branches. As an example: how to plot the invariant mass of the two highest-p_T b-tagged jets that are not within $\Delta R < 0.3$ from any isolated muon with $p_T > 15\,\text{GeV}$?

```python
muons = op.select(t.Muon, lambda mu : op.AND(mu.pt > 15., mu.iso < 0.4))
cleanedBJets_unsorted = op.select(t.Jet, lambda j : op.AND(
    op.NOT(op.rng_any(muons, lambda mu : op.deltaR(mu.p4, j.p4) < 0.3)),
    j.bTag > 0.6))
cleanedBJets = op.sort(cleanedBJets_unsorted, lambda j : -j.pt)
has2b = noSel.refine("2b", cut=(op.rng_len(cleanedBJets) >= 2))
Plot.make1D("mbb", (cleanedBJets[0].p4+cleanedBJets[1].p4).M(), has2b,
        EqB(100, 0., 500.))
```

- Derived "collections" only exist in python, at the RDataFrame level they are vectors of indices, and the other columns are used directly
- Also allows to add some extensions for convenience (p4 is not in the NanoAOD, but constructed from `X_pt`, `X_eta`, `X_phi`, and `X_mass`)
- Can convert to code and RDataFrame nodes when constructing plots and selections, or in one go later

Implementation: interface to RDataFrame and Cling

- `Plot` and `Selection` interact with a wrapper (for bookkeeping) around the RDataFrame

- A tree of "selection nodes" is built up, each grouping a `Filter` node with an attached set of `Define` nodes

- When converting an expensive expression to a C++ string, values are defined on-demand by attaching `Define` nodes (and functions declared with the interpreter as needed; global scope, so can be reused everywhere)

- Main python challenge: fast traversal and comparison of (sub)expressions to avoid redefinition. Caching of a value-based hash of every expression (they are effectively immutable) solved this for almost all cases
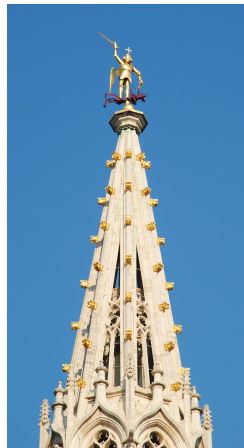

image credit

Pushing the limits: automatic systematic variations

- Many systematic uncertainties are taken into account in very similar ways: as a change in per-event weight, or as a different value for some quantities (e.g. jet energy)

- If expressions are marked as changing under a certain systematic effect (in the decorations, or explicitly when constructing the expression), the correspondingly varied histograms can be automatically produced

- On by default, but can be disabled for a selection (and everything attached to it) or a plot

Implementation: the backend code scans cuts, weights, and variables for marked nodes, and defines the additional RDataFrame nodes as needed: alternative weights only add `Define` and `Histo1D`, but anything used in a `Filter` (e.g. jet p_T) clones the whole attached subgraph

Quite some bookkeeping, but fully generic (changes to analysis code are minimal), and the code for this is localised in a handful of places — killer feature, but also a performance stress-test (much larger graph)

What is in `bamboo.treefunctions`?

The main module with helper methods to construct expressions

- (most importantly) per-event range operations, using array branches (first, min/max, select, combine etc.). These are implemented using a range version of STL algorithms like find_if, copy_if, accumulate... and converting the result of the python lambda to a C++ lambda

- evaluating multivariate classifiers (we are actively using TMVA and tensorflow; torchscript and lwtnn are also implemented)

- indicating if/when expressions should be defined as columns

and also (since everything needs to become an expression)

- basic math, boolean logic, special functions etc.

- more C++-specific: construct an object, call a method

- some kinematic operations (using `ROOT::Math::VectorUtil`)

full list

RDF Development experience

- RDataFrame provides a nice and consistent API to build on
  - One thing "missing" is combining results (e.g. adding up histograms) from different analysis categories (different `Filter` branches of the graph) — the limitation makes sense (and can be worked around in the analysis code)
  - `Count` and `Sum` are nice, but ended up using 1-bin histograms for counters because they collect entries, sum of weights, and the uncertainty together

- JIT C++ support is really complete (templates) and solid (the compiler warnings and errors prevented a few bugs)

- Dynamically adding code, with automatic python bindings, makes it very easy to load extensions (e.g. for reading weights from a file, or evaluating a multivariate classifier)

- Main annoyance so far: logical errors in analysis code (read beyond array end) give a segmentation fault at runtime, and are hard to debug (removing parts of the graph to isolate the problem is time-consuming)

Example: code for saving skims as pandas dataframes

```python
import os.path
from bamboo.plots import Skim
from bamboo.analysisutils import loadPlotIt
from bamboo.root import gbl
import pandas as pd

def postProcess(self, taskList, config=None, workdir=None, resultsdir=None):
    super().postProcess(taskList, config=config, workdir=workdir, resultsdir=results
    skims = [ap for ap in self.plotList if isinstance(ap, Skim)]
    p_config, samples, _, systematics, legend = loadPlotIt(config, [], eras=self.ar
    for skim in skims:
        frames = []
        for smp in samples:
            for cb in (smp.files if hasattr(smp, "files") else [smp]):
                cols = gbl.ROOT.RDataFrame(cb.tFile.Get(skim.treeName)).AsNumpy()
                cols["weight"] *= cb.scale
                cols["process"] = [smp.name]*len(cols["weight"])
                frames.append(pd.DataFrame(cols))
        df = pd.concat(frames)
        cats = pd.unique(df["process"])
        df["process"] = pd.Categorical(df["process"], categories=cats, ordered=Fals
        pqoutname = os.path.join(resultsdir, f"{skim.name}.parquet")
        df.to_parquet(pqoutname)
        logger.info(f"Dataframe for skim {skim.name} saved to {pqoutname}")
```