bamboo: an RDataFrame-based analysis framework

Feedback from the implementation experience

Pieter David

Université catholique de Louvain

84rd ROOT Parallelism, Performance and Programming Model Meeting
8 October 2020

bamboo: a bit of context

- started (early 2019) as a rewrite of my histogram-filling code for NanoAOD, which required more flexibility than analysis-specific trees
- NanoAOD samples are centrally produced flat trees, directly — or with some small additions — usable for final analysis, introduced in 2018–2019 in CMS to increase the efficiency of analyses on large datasets (aiming to cover half of all CMS analysis use cases), more details here
- based on RDataFrame, cling, and PyROOT
- Currently used in several CMS analyses (mostly Higgs and top quark physics, also some performance measurements), close to a 1.0 release

A few links: repository, HTML documentation, some examples

I wrote most of the code of bamboo, but it could not have reached the current state without the feedback and help from a few early users, in particular Sébastien Wertz, Khawla Jaffel, and Florian Bury

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

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

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

How much time and effort to:

- plot one more distribution
- change a selection

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

How much time and effort to:

- plot one more distribution
- change a selection
- change a selection, and compare N plots between the two cases

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

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

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

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

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

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?

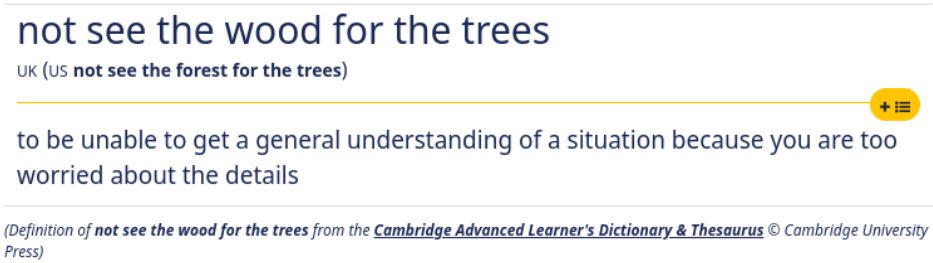
How much time and effort to:

- 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



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)

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 $p_T > 15$ GeV?

```
muons = op.select(t.Muon, lambda mu : op.AND(mu.pt > 15., mu.iso < 0.4))
cleanedBJets = 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))
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

Structure of a bamboo analysis module

```
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
```

`RResultPtr`s 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

The corresponding generated code

```
//   muons = op.select(t.Muon, lambda mu : op.AND(mu.mediumId,
//                                                mu.pfRelIso03_all < 0.4, mu.pt > 15.))
// DEBUG:bamboo.dataframebackend:Defining new symbol with interpreter:
ROOT::VecOps::RVec<std::size_t> myFun1(const ROOT::VecOps::RVec<Bool_t>& myArg0,
    const ROOT::VecOps::RVec<Float_t>& myArg1,
    const ROOT::VecOps::RVec<Float_t>& myArg2, const UInt_t& myArg3)
{ return rdfhelpers::select(rdfhelpers::IndexRange<std::size_t>{myArg3},
    [&myArg0, &myArg1, &myArg2, &myArg3] ( std::size_t i0 ) {
      return ( myArg0[i0] && ( myArg1[i0] <  0.4 ) && ( myArg2[i0] >  15.0 ) );
    });
};
// DEBUG:bamboo.dataframebackend:Defining myCol1 as
myFun1(Muon_mediumId, Muon_pfRelIso03_all, Muon_pt, nMuon)
//   twoMuSel = noSel.refine("has2mu", cut=( op.rng_len(muons) > 1 ))
// DEBUG:bamboo.dataframebackend:Filtering with ( myCol1.size() >  1 )
//   plots.append(Plot.make1D("dimuM", (muons[0].p4+muons[1].p4).M(),
//                twoMuSel, EqB(100, 20., 120.), title="Invariant mass"))
// DEBUG:bamboo.dataframebackend:Defining v0_dimuM as
( ROOT::Math::LorentzVector<ROOT::Math::PtEtaPhiM4D<float> >{
    Muon_pt[myCol1[0]], Muon_eta[myCol1[0]], Muon_phi[myCol1[0]],
      Muon_mass[myCol1[0]]} +
  ROOT::Math::LorentzVector<ROOT::Math::PtEtaPhiM4D<float> >{
    Muon_pt[myCol1[1]], Muon_eta[myCol1[1]], Muon_phi[myCol1[1]],
      Muon_mass[myCol1[1]]} ).M()
```

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

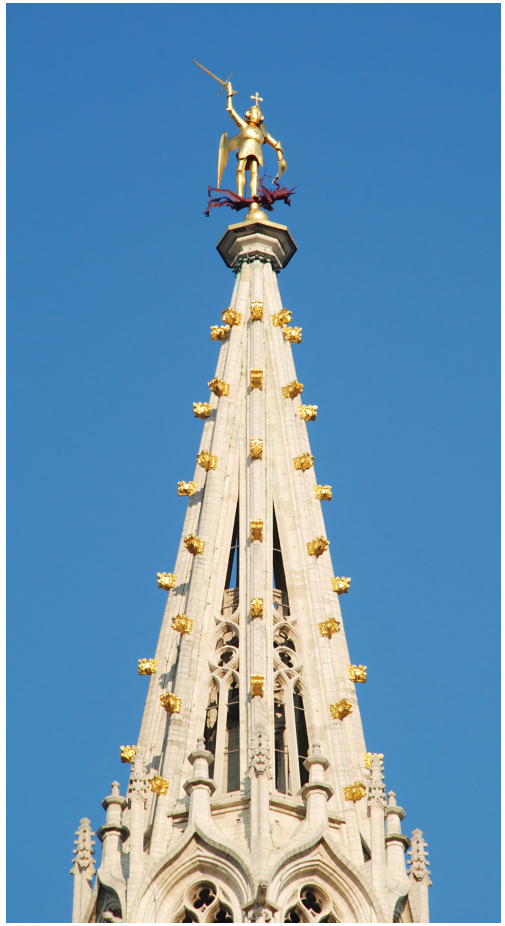


image credit

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

Combining histograms for different samples

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

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

- `definePlots` gets all sample metadata, so can adjust the graph for it
- One RDataFrame graph per sample, or (more commonly) batch job
- By default combined in a stack plot with plotIt, but easy to override

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

options for running on a batch system, enabling IMT, verbose logging...

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?

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)

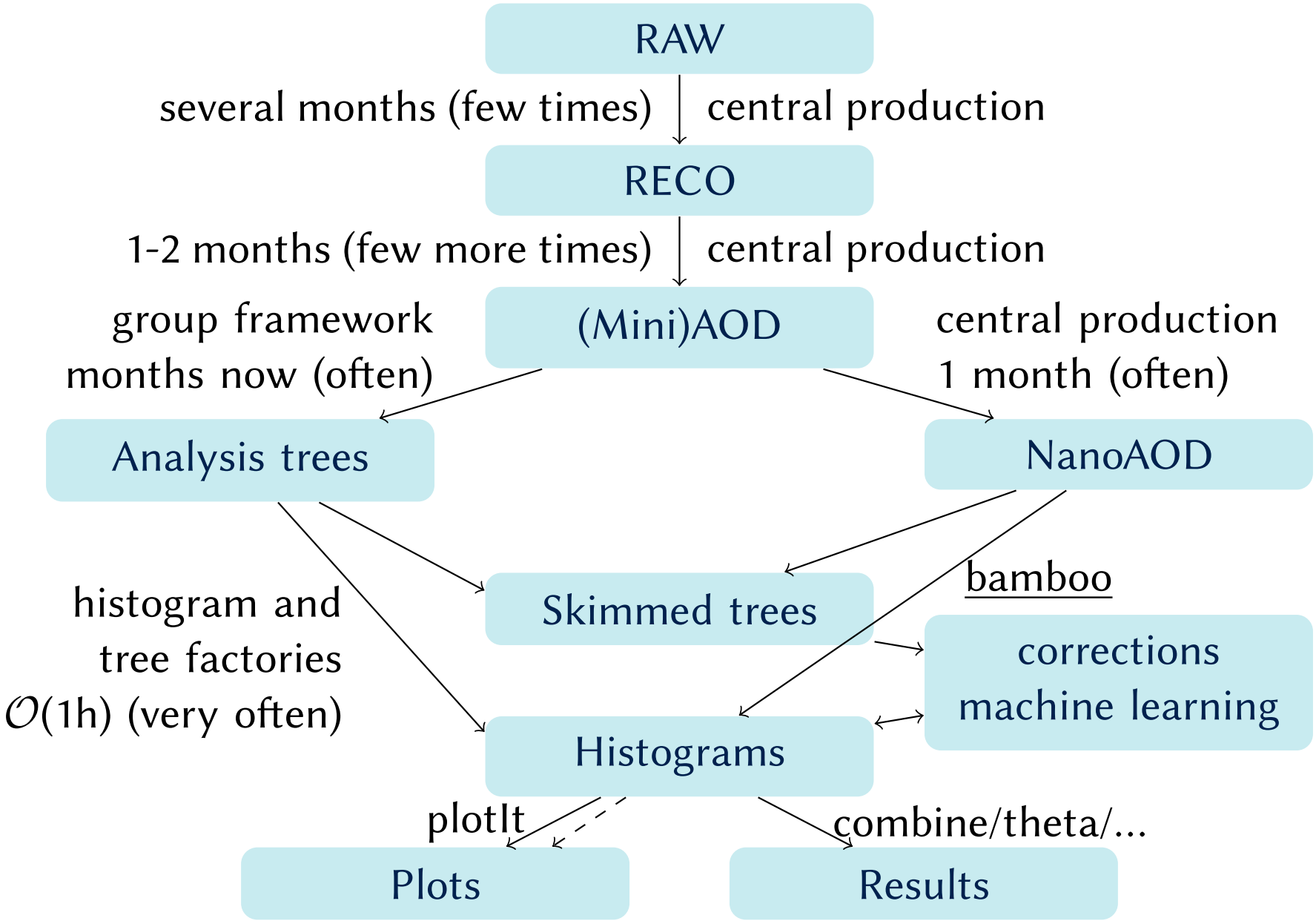
Conclusions

- Described bamboo, with a focus on filling histograms (the most common use case; making skims is also supported, based on `RDF::Snapshot`)
- RDataFrame with JIT scales well to making plots for a CMS Run2 analysis (thousands of histograms, $\mathcal{O}(10\,\mathrm{TB})$ of data available locally, using a batch system to split and process samples in parallel)
- Relatively large graphs, and centralised calls to RDataFrame, so it is relatively easy to try some changes in the latter that could improve performance (and I am happy to test things; our largest graphs assume CMS data locally available, so not as easy to provide runnable code)

Thanks for the opportunity to present here and exchange ideas with you!

Additional material

Analysis workflows: from MiniAOD to NanoAOD



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

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

From code to workflows

```
$ bambooRun -m myAnalysis.py:BasicPlots mySamples.yml
```

- Also between the command line and the analysis definition, much code can be shared or reused, e.g. for processing different samples and combining the results in one plot, using a batch system...
- Proposed solution: analysis module inherits from a base class, and implements the `definePlots` method (which returns a list of `Plot` objects)
- Input samples, and plot options, are passed through a YAML file

With different options one can:

- interactively explore the decorated tree (in an IPython prompt)
- run over only one file per sample, for testing locally
- run sequentially or on a batch system (slurm or HTCondor are supported), worker jobs use almost the same command
- rerun only the postprocessing (plotting) step
- enable “implicit multi-threading”

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...)

Selections and plots: a longer example

```
def definePlots(self, t, noSel, sample=None, sampleCfg=None):
  from bamboo.plots import Plot, EquidistantBinning
  from bamboo import treefunctions as op
  plots = []
  muons = op.select(t.Muon, lambda mu : mu.pt > 20.)
  twoMuSel = noSel.refine("dimu", cut=(op.rng_len(muons) > 1))
  plots.append(Plot.make1D("dimu_M",
    op.invariant_mass(muons[0].p4, muons[1].p4), twoMuSel,
    EquidistantBinning(100, 20., 120.), title="Dimuon invariant mass"))
  jets = op.select(t.Jet, lambda j : j.pt > 20.)
  plots.append(Plot.make1D("dimu_nAllJets", op.rng_len(jets), twoMuSel,
    EquidistantBinning(10, 0., 10.), title="Number of jets (uncleaned)"))
  cleanedJets = op.select(jets, lambda j : op.NOT(
    op.rng_any(muons, lambda mu : op.deltaR(mu.p4, j.p4) < 0.3)))
  plots.append(Plot.make1D("dimu_nJets", op.rng_len(jets), twoMuSel,
    EquidistantBinning(10, 0., 10.), title="Number of jets (cleaned)"))
  twoMuTwoJetSel = twoMuSel.refine("dimudijet",
    cut=(op.rng_len(cleanedJets) > 1))
  plots.append(Plot.make1D("dimudijet_leadJetPT", cleanedJets[0].pt,
    twoMuTwoJetSel, EquidistantBinning(50,0.,250.),title="Leading jet PT"))
  return plots
```