Readable and efficient HEP data analysis with bamboo

using python, PyROOT, cling, and RDataFrame

Pieter David

Université catholique de Louvain

PyHEP 2019 Workshop
16–18 October 2019
Abingdon, United Kingdom

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

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: python is a great fit

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: python is a great fit

How much time and effort to:

- plot one more distribution
- change a selection

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: python is a great fit

How much time and effort to:

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

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: python is a great fit

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

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: python is a great fit

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

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: python is a great fit

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

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

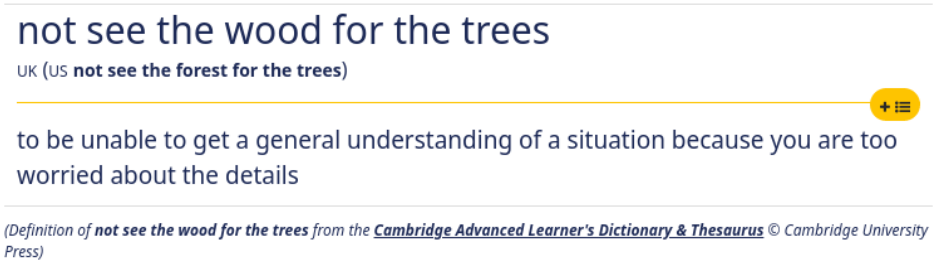
How much time and effort to:

- 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

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: python is a great fit
- 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



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: python is a great fit
- 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)*

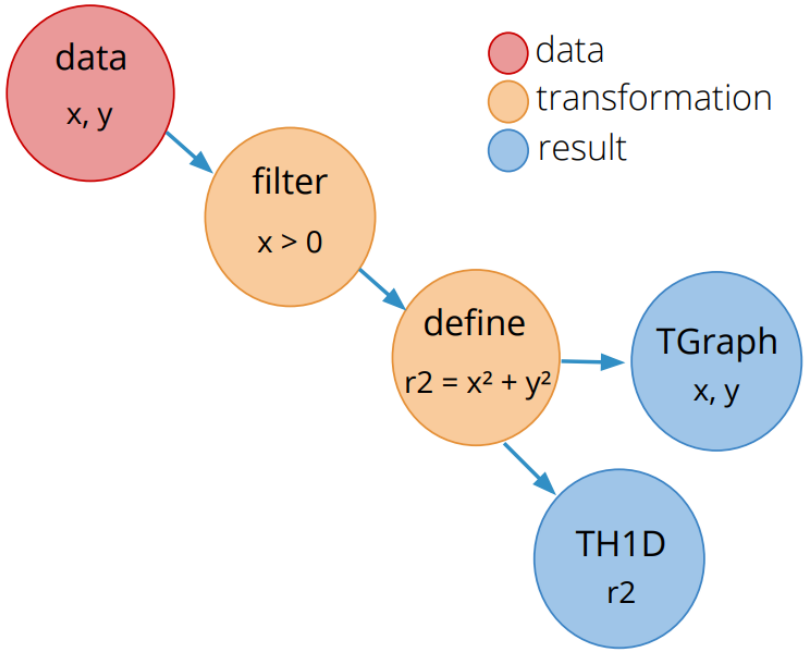
image credit: Claudio Caputo

there are many ideas to address this problem — this is only one attempt, based on what seemed promising for my use-case

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")
```



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

18

from this talk, see also today's CERN EP Software Seminar

Cling (C++ interpreter / JIT compiler) and PyROOT

- The ROOT interpreter is based on Clang/LLVM: correctly handles almost any valid modern C++ code (templates, lambda functions...)
- PyROOT exposes almost all of ROOT's functionality in python (and that of any C++ you add)
- Experimental PyROOT is bringing python callables to RDataFrame (and much more)
- Current bamboo compromise: pass expressions to RDataFrame as strings (generated with python)

bamboo ingredients — outline for the rest of the talk

- High-level python analysis code to define plots and selections (using loops, functions that take function arguments etc.)

bamboo ingredients — outline for the rest of the talk

- High-level python analysis code to define plots and selections (using loops, functions that take function arguments 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

bamboo ingredients — outline for the rest of the talk

- High-level python analysis code to define plots and selections (using loops, functions that take function arguments 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)

bamboo ingredients — outline for the rest of the talk

- High-level python analysis code to define plots and selections (using loops, functions that take function arguments 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

bamboo ingredients — outline for the rest of the talk

- High-level python analysis code to define plots and selections (using loops, functions that take function arguments 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

Why decorate TTrees?

Example: plot a dimuon invariant mass distribution. The momenta are in the branches `Muon_pt[nMuon]`, `Muon_eta[nMuon]`, `Muon_phi[nMuon]`, and `Muon_mass[nMuon]` (and there may be more than two muons in the event)

Why decorate TTrees?

Example: plot a dimuon invariant mass distribution. The momenta are in the branches `Muon_pt[nMuon]`, `Muon_eta[nMuon]`, `Muon_phi[nMuon]`, and `Muon_mass[nMuon]` (and there may be more than two muons in the event)

```
using ROOT::Math::VectorUtil::InvariantMass;
using LorentzVector =
    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(LorentzVector(pt[0], eta[0], phi[0], m[0]),
                         LorentzVector(pt[1], eta[1], phi[1], m[1]));
  }, {"Muon_pt", "Muon_eta", "Muon_phi", "Muon_mass"}
  ).Histo1D(..., "Dimuon_mass", ...);
```

Why decorate TTrees?

Example: plot a dimuon invariant mass distribution. The momenta are in the branches `Muon_pt[nMuon]`, `Muon_eta[nMuon]`, `Muon_phi[nMuon]`, and `Muon_mass[nMuon]` (and there may be more than two muons in the event)

```
using ROOT::Math::VectorUtil::InvariantMass;
using LorentzVector =
    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(LorentzVector(pt[0], eta[0], phi[0], m[0]),
                         LorentzVector(pt[1], eta[1], phi[1], m[1]));
  }, {"Muon_pt", "Muon_eta", "Muon_phi", "Muon_mass"}
  ).Histo1D(..., "Dimuon_mass", ...);
```

Alternative, using the JIT compiler instead of a lambda function

```
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", ...);
```

Why decorate TTrees?

bamboo equivalent:

```
from bamboo import treefunctions as op
Plot.make1D(...,
    op.invariant_mass(t.Muon[0].p4, t.Muon[1].p4), ...)
```

Why decorate TTrees?

bamboo equivalent:

```
from bamboo import treefunctions as op
Plot.make1D(...,
    op.invariant_mass(t.Muon[0].p4, t.Muon[1].p4), ...)
```

- The example on the previous slide assumes no selection or sorting is needed (could e.g. keep a list of indices with sorted good muons)
- For jets: need to take different branches for systematic variations
- In practice: need to compose, e.g. invariant mass of the two highest-p_T b-tagged jets that are not within $\Delta R < 0.3$ from any selected muon

Why decorate TTrees?

bamboo equivalent:

```
from bamboo import treefunctions as op
Plot.make1D(...,
    op.invariant_mass(t.Muon[0].p4, t.Muon[1].p4), ...)
```

- The example on the previous slide assumes no selection or sorting is needed (could e.g. keep a list of indices with sorted good muons)
- For jets: need to take different branches for systematic variations
- In practice: need to compose, e.g. invariant mass of the two highest-p_T b-tagged jets that are not within $\Delta R < 0.3$ from any selected muon

Solution to #3:

```
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))
Plot.make1D(...,
    op.invariant_mass(cleanedBJets[0].p4, cleanedBJets[1].p4), ...)
```

Not the fairest comparison — just to show how this can make code simpler

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

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()`)

image credit

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()`

image credit

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

Selections and plots

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

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)

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)

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

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

Selections and plots: an 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
```

Implementation: interface to RDataFrame and Cling

- `Plot` and `Selection` interact with a wrapper class around the RDataFrame
- A tree of "selection nodes" is built up, grouping a `Filter` node and an attached set of `Define` nodes

image credit

Implementation: interface to RDataFrame and Cling

- `Plot` and `Selection` interact with a wrapper class around the RDataFrame
- A tree of "selection nodes" is built up, grouping a `Filter` node and 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)

image credit

Implementation: interface to RDataFrame and Cling

- `Plot` and `Selection` interact with a wrapper class around the RDataFrame
- A tree of "selection nodes" is built up, grouping a `Filter` node and 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)
- Much of this needs fast searches through expression trees for dependencies etc., which is achieved by caching the value-based hash of every expression (possible because they are immutable)

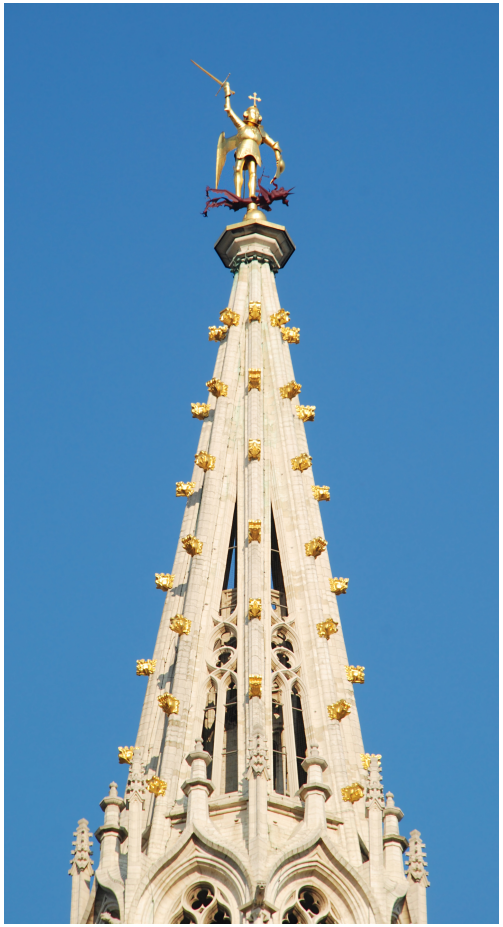


image credit

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...

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

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)

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

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

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

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:

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

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

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

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...)

Pushing the limits: automatic systematics

- 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)

Pushing the limits: automatic systematics

- 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

Pushing the limits: automatic systematics

- 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

Pushing the limits: automatic systematics

- 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 are cheap, but anything used in cuts may change the events passing a filter, so need to branch off, and duplicate subsequent defines)

Quite some bookkeeping, but fully generic (changes to analysis code are minimal), and the code for this is localised in a handful of places

Future developments and links

- Development status: beta; usable, but moving from a narrow proof-of-concept to actual research, so some interfaces may still change
- Performance: memory usage increases to about 5GB for 5000 histograms (more than expected, under investigation), CPU time acceptable for now ($\mathcal{O}(1\text{min})$ for defining plots — compiled code speed for the event loop)
- Being used or evaluated for three CMS analyses — early adopter feedback has been extremely valuable; many thanks, especially to Khawla Jaffel (UCLouvain) and Sébastien Wertz (Universität Zürich)

- Technical requirements: python3.6+, a recent ROOT (6.14/06, 6.16/00 or 6.18/04), and a few python packages (typical installation: with pip in a virtualenv). plotIt (a ROOT-based C++ tool) is used for turning histograms into pdf/png stack plots
- The code is public in this repository, and there is HTML documentation

Conclusion

- RDataFrame is a very powerful tool (especially when paired with Cling for piece-by-piece compilation)
- For actual analysis use: still a lot of bookkeeping to do, especially for flat trees (e.g. indices)
- bamboo is an attempt to bridge this gap, such that the user code is little more than a compact description of the analysis

- Hopefully some (not too) creative uses of python, and interesting ideas
- Started from a simple problem and idea, ended up writing a more general tool... usable, but much room for improvement and additionss

Thanks for your attention, I am looking forward to hearing your thoughts/suggestions/criticism and exchange ideas

Additional material

Installation

Suggested way to get a recent ROOT release: from the LCG software distribution on CVMFS

Installation (to be improved):

```
source /cvmfs/sft.cern.ch/lcg/views/LCG_95apython3/x86_64-centos7-gcc8-opt/setup.sh
python -m venv myvenv
source myvenv/bin/activate
git clone -o upstream git+https://gitlab.cern.ch/cp3-cms/bamboo.git
cd bamboo/ext
./getjetclasses.sh ## copy some source files from CMSSW
cd -
pip install ./bamboo
```

plotIt (for pdf/png output):

```
git clone -o upstream https://github.com/cp3-llbb/plotIt.git
cd plotIt/external
./build-external.sh
cd ..
BOOST_ROOT=$CMAKE_PREFIX_PATH make -j4
cd ..
cp plotIt/plotIt myvenv/bin
```

Upload to pip/condaforge is planned
(the bamboo name is taken, so probably bamboo-hep)

CP3SlurmUtils on test.pypi.org, will be uploaded to production PyPI soon