Fast and readable analysis with Bamboo

ROOT user workshop

Sébastien Wertz

May 9th, 2022

History & Motivation

- Original author: Pieter David (left CMS)
- Development started in 2018, after recognizing increasing complexity of multi-year analyses
- Goal: find a way to write analyses that was
  - Easy to write, modify, share
  - Fast

    $\rightarrow$ usually one or the other (or neither), rarely both...
- CMS's NanoAOD + RDataFrame: a match made in heaven?
- RDataFrame reduces boilerplate, declarative, but: writing full analysis (with all systematics, bookkeeping...) can still be daunting

RDataFrame: use it directly? ...

Typical example: dimuon invariant mass (*):

Using C++ lambdas:

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

How about:

- Additional selections
- Adding collection cross-cleaning
- Sorting all quantities associated with an object
- For jets: repeat for all systematic variations
- ...

(*): not the only way to do it, but conclusion remains

RDataFrame: use it directly? ...

Typical example: dimuon invariant mass (*):

Or, using JITing:

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

How about:

- Additional selections
- Adding collection cross-cleaning
- Sorting all quantities associated with an object
- For jets: repeat for all systematic variations
- ...

(*): not the only way to do it, but conclusion remains

Enter bamboo: decorated trees

In bamboo, this reduces to:

```python
from bamboo import treefunctions as op
from bamboo.plots import Plot

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

- Idea: decorate tree → provide a view of the event content as a set of (collections of) physics objects in the form of “proxies” (python objects)
- User builds expressions (cuts, variables, …) from these proxies
- When done: Bamboo converts expressions to appropriate (C++) strings, builds RDataFrame, runs event loop
- Same user-facing proxy (e.g. `tree.Jet`) can represent different collections/branches: systematic variations automatically handled (different columns in generated RDF graph)

Under the hood: proxies and operations

- Operations (backend):
  - Can be directly converted to C++ strings for compiling
  - Simple python objects, immutable $\rightarrow$ can be modified through a clone, e.g. for systematic variations
- Proxies (user-facing):
  - Represent objects in the tree, and quantities derived from those
  - Behave like the value they represent (list, float, LorentzVector, ...)
  - Wrap operations (can be several, e.g. for systematics)
- Fairly complete list of implementations to work with proxies
- Tree proxies automatically generated based on the branches found
  - "Groups" (`tree.pdf.x1`), collections (`tree.Muon[0].pt`), objects with methods (`tree.Muon[0].p4.E()`), refs. to other collections (`tree.Muon[0].Jet.btagDeepB`), indices (`SortedJets[0].idx`)
- Proxy mechanism not tied to NanoAOD: TTree decoration can be adapted to $\sim$ any tree format, see e.g. Snowmass [1][2], Delphes

Proxies for more complex tasks

- Select muon and jets
- Clean jets from selected muons, sort jet collection
- Build all unique combinations of 3 jets
- Find 3-jet combination with total invariant mass closest to given value
- Get b-tag value of leading jet of among those 3 jets

```
muons = op.select(tree.Muon, lambda mu:
                    op.AND(mu.pt > 30., op.abs(mu.eta) < 2.4))

jets = op.select(tree.Jet, lambda j: op.AND(j.pt > 30., op.abs(j.eta) < 2.4))

cleanedJets = op.select(jets, lambda j: op.NOT(
        op.rng_any(muons, lambda mu: op.deltaR(mu.p4, j.p4) < 0.4)))
# tree.Jet is not guaranteed to be sorted (jet smearing)
sortedJets = op.sort(cleanedJets, lambda j: -j.pt)

triJets = op.combine(sortedJets, N=3)

XjjjCand = op.rng_min(triJets, lambda jjj:
            op.abs(op.invariant_mass(jjj[0].p4 + jjj[1].p4 + jjj[2].p4) - mX))

leadCandBtag = XjjjCand[0].bTagDeepB
```

Selecting and plotting events: basic building blocks

Selection object

- Holds cuts and weights
- Start from inclusive selection (all events), unit weight
- Gradually refine selection: add cuts and/or weights
- RDF: *`Filter`* nodes

Declaring a plot

- Requires only selection object, and plotted quantity(ies)
- Fill single or multiple entries (collection) (per-entry weight supported)
- RDF: *`HistoND`* nodes (only $N \leq 3$ supported atm)

Selecting and plotting events: basic building blocks

Selection object

- Holds cuts and weights
- Start from inclusive selection (all events), unit weight
- Gradually refine selection: add cuts and/or weights
- RDF: `Filter` nodes

Declaring a plot

- Requires only selection object, and plotted quantity(ies)
- Fill single or multiple entries (collection) (per-entry weight supported)
- RDF: `HistoND` nodes (only $N \leq 3$ supported atm)

More advanced functionalities follow $\sim$ same interface:

- Selections for data-driven estimations
- Categorized selections (e.g. concisely handle multiple lepton flavours)
- ...

Selecting and plotting events: basic building blocks

Selection object

- Holds cuts and weights
- Start from inclusive selection (all events), unit weight
- Gradually refine selection: add cuts and/or weights
- RDF: `Filter` nodes

Declaring a plot

- Requires only selection object, and plotted quantity(ies)
- Fill single or multiple entries (collection) (per-entry weight supported)
- RDF: `HistoND` nodes (only $N \leq 3$ supported atm)

Notes:

- Skims also supported: `Snapshot` (more later)
- `Define` nodes also inserted in the RDF graph (typically before first `Filter` node that uses them, to avoid recomputing quantities)

Alternative “backends”

Different methods of constructing the RDataFrame:

1. “Lazy” (default):
   - First register all selections, plots, …in Bamboo
   - Then create the RDataFrame
   - Advantage: ordering of *`Define`*/*`Filter`* nodes handled by Bamboo
2. “Debug”:
   - Create RDataFrame nodes eagerly as user builds expressions in Bamboo
   - Useful to detect problems with RDF building earlier
   - User has to think about ordering of definitions for efficiency
3. “Compiled”:
   - As “lazy”, but no JITing: generate full C++ code for standalone executable, call external compiler
   - Advantage: can use compiler optimizations inject debugging symbols, …
   - In practice, compilation of realistic analysis with optimizations is too slow
   - Considering to discontinue (optimizations now usable in cling, debugging hopefully soon)

Selecting and plotting events

```python
from bamboo.plots import Plot, EquidistantBinning as EqBin
from bamboo import treefunctions as op

def definePlots(self, t, noSel, sample=None, sampleCfg=None):
    plots = []

    muons = op.select(t.Muon, lambda mu:
        op.AND(mu.pt > 30., op.abs(mu.eta) < 2.4))

    muSel = noSel.refine("1mu", cut=(op.rng_len(muons) == 1))

    plots.append(Plot.make1D("mu_pt", muons[0].pt, muSel,
        EqBin(100, 30., 130.), title="Muon pt"))

    jets = op.select(t.Jet, lambda j: op.AND(j.pt > 30., os.abs(j.eta) < 2.4))

    mu4JetSel = muSel.refine("1mu_4j", cut=(op.rng_len(jets) >= 4))

    plots.append(Plot.make1D("jet_pt", op.map(jets, lambda j: j.pt),
        mu4JetSel, EqBin(100, 30., 130.), title="All jets pt"))

    return plots
```

Caveat: merging selections is not possible (limitation of RDF); helpers provided to add histograms from distinct selections in a postprocessing step

Systematic uncertainties

- If an expression is marked as having systematic variations, Bamboo will automatically branch the RDF graph, create histogram variations only when needed
- Single event loop, all systematics computed on-the-fly
- Variations are solely identified by their full name, not limited to up/down
  E.g. *psISRup*, *psISRdown*, *pdf1*, *pdf2* ...
- Different expressions with the same variation name are varied together
  → correlate e.g. effect of JES on jet kinematics and b-tagging SFs
- Configuring simple weight-based systematic uncertainties:

```
psISRSyst = op.systematic(1., name="psISR",
    up=tree.PSWeight[2], down=tree.PSWeight[0])

pdfSysts = op.systematic(1.,
    **{ f"pdf{i}": tree.LHEPdfWeight[i] for i in range(1, 101) })

selWithSysts = noSel.refine("withSysts", weight=[psISRSyst, pdfSysts])
```

- 1 variation = 1 histogram: shows its limits with many variations
  → use Josh Bendavid's narf? (systematic index as extra dimension)

Running an analysis

Running an analysis in Bamboo requires:

1. An analysis module deriving from a base class $\rightarrow$ reuse Bamboo's facilities for sample bookkeeping, job submissions, etc.

```
class BasicPlots(NanoAODHistoModule):
    def definePlots(self, tree, noSel, sample=None, sampleCfg=None):
        ...
        return plots
```

Running an analysis

Running an analysis in Bamboo requires:

1. An analysis module deriving from a base class → reuse Bamboo's facilities for sample bookkeeping, job submissions, etc.
2. A configuration file (YAML): mostly for specifying input samples

```yaml
tree: Events
eras:
    2018UL:
        luminosity: 59830.
samples:
    TTToSemiLeptonic__2018UL:
        era: 2018UL
        db: das:/TTToSemiLeptonic_TuneCP5_13TeV-powheg-pythia8/.../NANOAODSIM
        cross-section: 365.35
        generated-events: genEventSumw
```

Running an analysis

Running an analysis in Bamboo requires:

1. An analysis module deriving from a base class $\rightarrow$ reuse Bamboo's facilities for sample bookkeeping, job submissions, etc.
2. A configuration file (YAML): mostly for specifying input samples

Then, just run it: `% bambooRun -m myAnalysis.py:BasicPlots myConfig.yml`

Running an analysis

Running an analysis in Bamboo requires:

1. An analysis module deriving from a base class $\rightarrow$ reuse Bamboo's facilities for sample bookkeeping, job submissions, etc.
2. A configuration file (YAML): mostly for specifying input samples

Then, just run it: `% bambooRun -m myAnalysis.py:BasicPlots myConfig.yml`

Result: one file per sample, containing all histograms/skims

Running an analysis

Running an analysis in Bamboo requires:

1. An analysis module deriving from a base class $\rightarrow$ reuse Bamboo's facilities for sample bookkeeping, job submissions, etc.
2. A configuration file (YAML): mostly for specifying input samples

Then, just run it: `% bambooRun -m myAnalysis.py:BasicPlots myConfig.yml`

Result: one file per sample, containing all histograms/skims

- Analysis description is contained in user module + config file
- Independent of how the events are processed
- Single entry point: `bambooRun` $\rightarrow$ choose processing mode through command-line arguments, no changes to analysis code necessary

Running an analysis

Running an analysis in Bamboo requires:

1. An analysis module deriving from a base class → reuse Bamboo's facilities for sample bookkeeping, job submissions, etc.
2. A configuration file (YAML): mostly for specifying input samples

Then, just run it: `% bambooRun -m myAnalysis.py:BasicPlots myConfig.yml`

Result: one file per sample, containing all histograms/skims

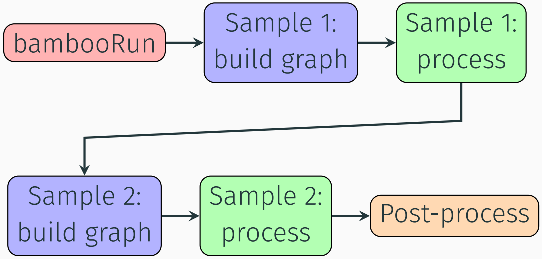
Need one RDF graph/sample:

- Different data-taking eras → different cuts, scale factors, systematics
- Differences in data vs. MC (background) vs. MC (signal)
- Specific selections for data-driven estimations
- MC: sample-dependent uncertainties

Processing modes: sequential

```
% bambooRun ... [--distributed sequential] [--threads 4]
```

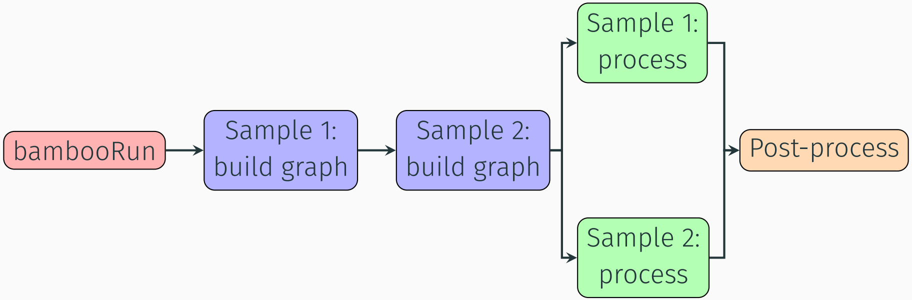
- Default mode, mostly useful for quick tests
- Small memory overhead from every RDF
- Advantage: JITted symbols can be reused across graphs
- Can use implicit multithreading or distributed RDF



Processing modes: parallel

```
% bambooRun ... --distributed parallel [--threads 4]
```

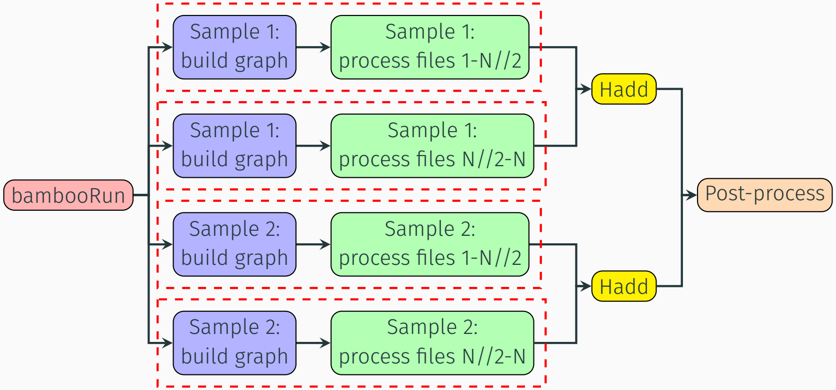
- Use `RDF::RunGraphs`
- Need to build all RDF graphs first
- Small memory overhead from every graph
- Advantage: JITted symbols can be reused across graphs
- Can use implicit multithreading or distributed RDF



Processing modes: batch

```
% bambooRun ... --distributed driver [--threads 4]
```

- Submit jobs on a cluster (HTCondor, Slurm supported)
- Monitoring loop, combines results for one sample as soon as its jobs are done → no overhead
- Some duplication of work: every job builds a graph (→ IMT on nodes)
- Usual limitations of batch processing: manual splitting, job failures...


bambooRun
Sample 1: build graph
Sample 1: process files 1-N//2
Sample 1: build graph
Sample 1: process files N//2-N
Sample 2: build graph
Sample 2: process files 1-N//2
Sample 2: build graph
Sample 2: process files N//2-N
Hadd
Hadd
Post-process

Distributed processing

```
% bambooRun ... --distributed parallel --distrdf-be dask_slurm
```

- Experimental support of distributed RDataFrame with Dask or Spark
- In practice, currently most relevant is Dask with jobqueue
- Initial difficulties in properly propagating environment & dependencies to workers, now solved
- Optimal splitting (number of tasks) not obvious
  Every task needs to re-build graph + JIT? overhead?
- Still missing: Numba support (WIP?)

Dask/jobqueue experience

- Observed scaling issues (large graphs): fixed soon?
- Stability issues (killed workers, timeouts): can error handling be improved in distRDF or should this be understood/solved in Dask?
- “Stuck” clusters: all jobs cancelled, but client keeps running
- Properly configuring & tuning Dask-distributed/jobqueue is delicate...

Customization hooks

Users' modules can also easily:

- Add command-line arguments, passed from *bambooRun*
- Extend the configuration file syntax (e.g. better handling of samples/eras)
- Configure the tree decorations (e.g. jet systematics)
- Further post-process the outputs, profit from available metadata
  Some post-processing typically necessary to use results e.g. in Combine (rescale, move, rename histograms)

```
class BasicPlots(NanoAODHistoModule):
    def addArgs(self, parser):
        ...
    def customizeAnalysisCfg(self, analysisCfg):
        ...
    def prepareTree(self, tree, sample=None, sampleCfg=None):
        ...
    def postProcess(self, taskList, config=None, workdir=None, resultsdir=None):
        ...
```

More features and recipes

Bamboo has been used for a variety of analyses: searches, unfolding, future studies; data-driven or MC-driven; using MVAs/DNNs; …
$\rightarrow$ fairly complete set of features and recipes collected, e.g. for:

More features and recipes

Bamboo has been used for a variety of analyses: searches, unfolding, future studies; data-driven or MC-driven; using MVAs/DNNs; ...
$\rightarrow$ fairly complete set of features and recipes collected, e.g. for:

- Evaluating MVAs: TMVA (RReader), Tensorflow, PyTorch, ONNX Runtime (C/C++ APIs)

```
from bamboo.treefunctions import mvaEvaluator
mu = tree.Muon[0]
dnn = mvaEvaluator("dnn.pt", mvaType="Torch")
dnn_out = dnn(mu.pt, mu.eta, mu.phi)
```

```
ele_bdt = op.mvaEvaluator("BDT.weights.xml", mvaType="TMVA")
ele_MVA = op.map(tree.Electron, lambda el: ele_bdt(el.dxy, el.sip3d, ...)[0])
# attach MVA outputs to electron proxies
tree.Electron.valueType.mva = treedecorators.itemProxy(ele_MVA)
# can then use as
tree.Electron[0].mva
```

- Limitation of RDF: no batch evaluation of MVAs
  $\rightarrow$ complex DNNs can be slow (improved by SOFIE!?)
- Need to produce skims for MVA training

More features and recipes

Bamboo has been used for a variety of analyses: searches, unfolding, future studies; data-driven or MC-driven; using MVAs/DNNs; ...
$\rightarrow$ fairly complete set of features and recipes collected, e.g. for:

- Producing skims: add new branches, keep input branches, ...skims can then also be reprocessed by Bamboo

```
twoMuSel = noSel.refine("twoMuons", cut=[ op.rng_len(muons) > 1 ])
plots.append(Skim("dimuSkim", {
            # copy from input file
            "run": None, "event": None,
            # add new branches
            "dimu_M": op.invariant_mass(muons[0].p4, muons[1].p4),
        }, twoMuSel,
    # also keep all electron branches
    keepOriginal=Skim.KeepRegex("^(n)?Electron.*$")))
```

More features and recipes

Bamboo has been used for a variety of analyses: searches, unfolding, future studies; data-driven or MC-driven; using MVAs/DNNs; …
$\rightarrow$ fairly complete set of features and recipes collected, e.g. for:

Storage needs & skims

- Typical workflow: 1) request local replica (Rucio) of NanoAOD samples ($\mathcal{O}(10)$ TB at T2/T3); 2) Produce final histograms in one go
- Or, skim with Bamboo (remote xrootd access: slow, but do it once), store only skims locally
  (but variations still computed on-the-fly $\rightarrow$ lightweight skims!)
- Skims follow same (NanoAOD) schema $\rightarrow$ same Bamboo user code can can produce and use skims, see example
- Writing skims as RNtuple could be interesting! (not supported yet in RDF)
- Essentially a caching issue… possibilities to improve site caching, avoid manual skimming step & local replicas?

More features and recipes

Bamboo has been used for a variety of analyses: searches, unfolding, future studies; data-driven or MC-driven; using MVAs/DNNs; ...
$\rightarrow$ fairly complete set of features and recipes collected, e.g. for:

- Data-driven background estimations
- Splitting an MC sample into sub-components
- Using user-defined functions or classes in C++ or python+Numba
- Producing cut flow reports, generate yield tables (Latex)
- Jet & MET variations
- Rochester muon momentum corrections
- ...

(see backup)

Performance, in practice

Example case: 150 plots of $\sim$ 50 bins, 70 variations each (out of which 25 on-the-fly jet variations) $\rightarrow \sim$ 10k `Histo1D`, 3k `Define`

Memory

- Batch mode (single graph): < 1.5 GB
- Sequential/parallel: ~ 1 GB upfront, $\lesssim 10$ MB for each additional RDF

Event throughput

- With systematics, single-threaded, reading from HDD through LAN: $\sim$ kHz
- 2–5x slower with 50–150 variations than without: much more efficient than re-running event loop for every variation (even when restricting to jet variations)
- Time to insight: few hours on batch system for full Run2: could be better? tail of slow jobs, random FS failures spoil the picture…(use intermediate skims?) $\rightarrow$ distributed RDF expected to help

Sore points: from more to less Bamboo-specific

- Entry point = executable, results written to files → no interactive exploration possible (e.g. notebooks)
- Finding efficient patterns for implementing small studies/changes/checks during review can be difficult
- Postprocessing of outputs: available metadata (e.g. in *`postProcess`* method) helps, but manipulating *`TFile`*'s + *`THN`*'s is awkward
  Get python boost-histograms, put everything in single *`pd.DataFrame`*?
- Default postprocessing not well suited for combining/comparing outputs from different runs
- Abstractions: only interact with proxies, lazy event loop in RDF
  → interactively inspecting data, individual events not possible
- Debugging with jitted RDF is difficult (improvements soon?)
- Batch processing: too many manual inputs (job splitting), actions (managing failed jobs) needed (distRDF to the rescue?)

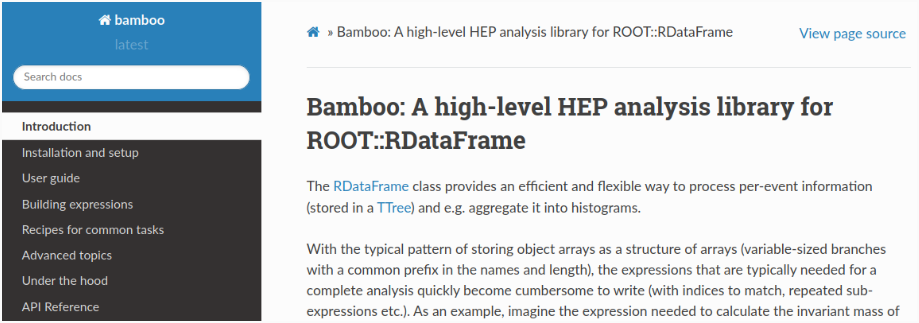
Ongoing, planned developments

- Finalize integration of distributed RDF
- Integrate *`RDF::Vary`* (automatic systematics in RDF)
  $\rightarrow$ simplify graph, lighter+faster!
- MVA evaluation: support SOFIE
- Incremental runs: every expression has a unique hash $\rightarrow$ store them, detect what changed w.r.t. a previous run, only re-process what changed + detect set of unique RDF graphs among all processed samples, re-use existing graph on several samples (if ever possible in RDF)?
- Move beyond ***`bambooRun`*** as single entry point $\rightarrow$ integrate with workflow management tools?
- Easier postprocessing with pyPlotIt

Caveat: only one active maintainer...

Documentation and examples

- Documentation
- Repository (includes examples)
- OpenData examples → run on binder!
- lxplus demo with systematics (requires CMS access)



Bamboo: A high-level HEP analysis library for ROOT::RDataFrame

The RDataFrame class provides an efficient and flexible way to process per-event information (stored in a TTree) and e.g. aggregate it into histograms.

With the typical pattern of storing object arrays as a structure of arrays (variable-sized branches with a common prefix in the names and length), the expressions that are typically needed for a complete analysis quickly become cumbersome to write (with indices to match, repeated sub-expressions etc.). As an example, imagine the expression needed to calculate the invariant mass of

Conclusions

- RDataFrame: write physics, not loops
- Writing a full analysis from scratch using RDF still requires re-inventing a lot of wheels
- RDF is (still quite) low level...Bamboo provides a high-level analysis description language embedded in familiar Python
- Fast and efficient processing of stock NanoAODs, no custom intermediate ntuples needed
- Not tied to CMS or NanoAOD: can be adapted to $\sim$ any format
- In use for 3 years, 6–7 analyses so far, $\sim$ 10–15 active users (AFAIK)
  - Future hinges on finding additional developers...
  - Some features upstreamed to RDF
- Join the discussion on Mattermost! (CMS only)

Conclusions

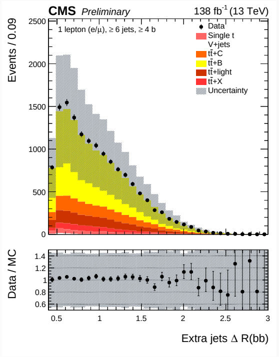
- RDataFrame: write physics, not loops
- Writing a full analysis from scratch using RDF still requires re-inventing a lot of wheels
- RDF is (still quite) low level…Bamboo provides a high-level analysis description language embedded in familiar Python
- Fast and efficient processing of stock NanoAODs, no custom intermediate ntuples needed
- Not tied to CMS or NanoAOD: can be adapted to ∼ any format
- In use for 3 years, 6–7 analyses so far, ∼ 10–15 active users (AFAIK)
  - Future hinges on finding additional developers…
  - Some features upstreamed to RDF
- Join the discussion on Mattermost! (CMS only)

Thank you!

Backup

Default postprocessing

- By default: write YAML config with list of plots and files, and call PlotIt: C++ tool to produce stacked plots using ROOT
- Fairly configurable (long list of options) but too rigid at the same time: good for data vs. MC stack + ratio, not much else
- Plan: move to python-based pyPlotIt
  - Re-use configuration file structure
  - More flexible manipulations, stacks, ratios, ...
  - UHI-compatible, can be used with mplhep



```
from matplotlib import pyplot as plt
import mplhep, plotit; from plotit import Stack
config, samples, plots, systematics, legend = plotit.loadFromYAML(cfgName
    )
for p in plots:
expStack = Stack([smp.getHist(p) for smp in samples if smp.cfg.type=="MC"
    ])
obsStack = Stack([smp.getHist(p) for smp in samples if smp.cfg.type=="
    DATA"])
mplhep.histplot(obsStack, histtype="errorbar", color="k")
mplhep.histplot(expStack.entries, stack=True, histtype="fill",
color=[e.style.fill_color for e in expStack.entries])
```

Data-driven background estimations

- Replace contribution of sample A/region SR with contribution from sample B/region CR + applied weights (e.g. fake rate transfer factor)

```
datadriven:
    chargeMisID:
        uses: [ data ]   # sample B
        replaces: [ DY ] # sample A
    nonprompt:
        uses: [ data ]
        replaces: [ TTbar ]
```

```
hasSameSignElEl = SelectionWithDataDriven.create(hasElEl, # common base selection
        "hasSSDiEl", "chargeMisID",
        cut=(diel[0].Charge == diel[1].Charge),   # region SR
        ddCut=(diel[0].Charge != diel[1].Charge), # region CR
        ddWeight=p_chargeMisID(diel[0]) + p_chargeMisID(diel[1]),
        enable=any("chargeMisID" in self.datadrivenContributions and
            self.datadrivenContributions["chargeMisID"].usesSample(sample,
                sampleCfg)))
```

- `SelectionWithDataDriven.create` similar to usual `Selection.refine`
- Resulting object behaves as any selection → refine, make plots etc.

Calling user-defined custom functions or classes

- Declare function, wrap it in a proxy, use it to build expressions:

```
ROOT.gInterpreter.Declare("""
    float computePDFWgtMean(const ROOT::VecOps::RVec<float>& weights) {
        return ROOT::VecOps::Mean(weights)
    }
""")

myFun = op.extMethod("computePDFWgtMean", returnType="float")
newSel = noSel.refine("avgWgt", weight=myFun(tree.LHEPdfWeight))
```

- Or use Numba:

```
import numpy as np
@ROOT.Numba.Declare(['RVec<float>'], 'float')
def computePDFWgtMean(weights):
    return np.mean(weights)

myFun = op.extMethod("computePDFWgtMean", returnType="float")
newSel = noSel.refine("avgWgt", weight=myFun(tree.LHEPdfWeight))
```

Calling user-defined custom functions or classes

- Or use external code: *myHeader.h*

```
class MyCalc {
    public:
        MyCalc(std::string path) { ... }
        evaluate(float pt) { ... }
};
```

- Then load dependencies:

```
bamboo.root.loadDependency(headers=myHeader.h, libraries=...)
```

- Finally, instantiate object and call its method:

```
myCalc = op.define("MyCalc", 'const auto <<name>> = MyCalc("file.root");')
myCorr = myCalc.evaluate(tree.Muon[0].pt)
```

- Note: *<<name>>* automatically replaced by Bamboo, makes sure symbols are unique

Example of extending configuration file

- Include additional information, e.g. tag signal processes
- Single entry for all eras → duplicate entry in *customizeAnalysisCfg()*, add *era* tag to config and *__era* suffix to sample name
- Splitting sample into sub-components
- Handling systematic variations from alternative samples

```
TTTo2L2Nu_hdampUP_TuneCP5_13TeV-powheg-pythia8:
    dbs:
        2017UL: das:/TTTo2L2Nu_hdampUP_TuneCP5_13TeV-powheg-pythia8/
             RunIISummer20UL17NanoAODv9-106X_mc2017_realistic_v9-v1/NANOAODSIM
        2018UL: das:/TTTo2L2Nu_hdampUP_TuneCP5_13TeV-powheg-pythia8/
             RunIISummer20UL18NanoAODv9-106X_upgrade2018_realistic_v16_L1v1-v1/
             NANOAODSIM
    subprocesses: ['ttB', 'ttcc', 'ttjj']
    signal_subprocesses: ['ttB']
    signal_tag: "powheg_5FS"
    cross-section: *xs_tt_2l
    syst: ['hdampup', 'TTTo2L2Nu_TuneCP5_13TeV-powheg-pythia8']
    generated-events: genEventSumw
```

Systematic uncertainties: scale factors

- CMS's correctionlib: JSON schema + reading library, recommended method for reading scale factors & associated variations:

```python
from bamboo.scalefactors import get_correction

elIDSF = get_correction("EGM_POG_SF_UL.json", "UL-Electron-ID-SF",
params={ "pt": lambda el: el.pt, "eta": lambda el: el.eta,
    "year": "2018UL", "WorkingPoint": "Loose" },
systParam="ValType", systNomName="sf",
systName="elID", systVariations=("sfup", "sfdown"))
# resulting variations in bamboo: elIDup, elIDdown

looseEl = op.select(tree.Electron, lambda el: el.looseId)

withDiEl = noSel.refine("withDiEl",
cut=(op.rng_len(looseEl) >= 2),
weight=[ elIDSF(looseEl[0]), elIDSF(looseEl[1]) ])
```

- `CorrectionSet` object declared once to `gInterpreter`, can be reused across samples
- Typically, evaluated SFs are always `Define`-d as a new column
  $\rightarrow$ avoid unnecessary re-evaluations

Systematic uncertainties: jet & MET

- Utility (now available as standalone package) to:
  - Re-apply JECs, smear jets, compute JEC & JER variations (regular & fat)
  - Propagate all those to MET (Type-1 MET)
- C++, RDF-friendly or standalone, python through pyROOT
- Originally based & validated on nanoAOD-tools implementation
- Bamboo: jets/MET kinematic variations are computed on-the-fly, automatically propagated to selections & plots

```
from bamboo.analysisutils import configureJets
configureJets(tree._Jet, "AK4PFchs", jec="Summer19UL18_V5_MC",
smear="Summer19UL18_JRV2_MC",
jesUncertaintySources="Merged", regroupTag="V2",
splitJER=True, addHEM2018Issue=True)
```

- Caching of SF .txt files from JECDB $\rightarrow$ will now move to correctionlib
- Need to centrally maintain these features – in this form or another (out of scope for correctionlib?)
- Note: Bamboo can also read variations from postprocessed nanoAODs

More details on JetMET tool

- Supported corrections:
  - AK4 jets & fat jets: apply JEC (any levels), JER, uncertainties (total/merged/split), JER uncertainty splitting, ad-hoc uncertainty for HEM18
  - In addition, for fat jets: JMS, JMR, GMS, GMR, Puppi corrections
  - Full Type-1 MET recipe
  - EE2017 noise fix recipe for MET
- Seed is passed explicitly $\rightarrow$ full reproducibility
- TODO: better handling of recipe evolution (e.g. EOY $\rightarrow$ UL): new classes? tag new version and deprecate the old?

More details on JetMET tool

- Config helper for instantation:

```
from CMSJMECalculators import config as calcConfigs
config = calcConfigs.JetVariations()
from CMSJMECalculators.jetdatabasecache import JetDatabaseCache
jrDBCache = JetDatabaseCache("JRDatabase", repository="cms-jet/JRDatabase")
config.ptResolutionSF = jrDBCache.getPayload(
                            "Summer16_25nsV1_MC", "SF", "AK4PFchs")
...
calc = config.create()
```

- Or create directly in C++:

```
auto calc = JetVariationsCalculator::create(jecParams, jesUncs, ...);
```

- Can be used:
  - From C++ & from RDataFrame

  ```
  df.Define("ak4JetVars", "calc.produce(Jet_pt, Jet_eta, ...)")
  ```

  - From python through pyROOT

  ```
  from CMSJMECalculators.utils import toRVecFloat, toRVecInt
  jetVars = calc.produce(
              toRVecFloat(tree.Jet_pt), toRVecFloat(tree.Jet_eta), ...)
  ```

Getting a specific variation

- Jet variations: original collection available as *`tree._Jet[``nominal'']`*, other variations directly accessible as *`tree._Jet[``jesTotalUp'']`* etc. *`jet.idx`* always refer to index in original collection
- Get a specific variation for any expression:

```
triJets = op.combine(sortedJets, N=3)
XjjjCand = op.rng_min(triJets, lambda jjj:
    op.abs(op.invariant_mass(jjj[0].p4 + jjj[1].p4 + jjj[2].p4) - mX))
leadCandBtag = XjjjCand[0].bTagDeepB

leadCandBtag_jesTotalUp = op.forSystematicVariation(leadCandBtag, "jesTotalUp")
```

- Useful for debugging, skims...

Analysis preservation

- Bamboo: analysis code should be kept outside of framework itself, in separate Git repository
- *`bambooRun`* output folder → contains *`version.yml`* file with Git commit of analysis code (& Bamboo itself), and full list of command-line arguments to *`bambooRun`* used to produce the results
  → all the information needed to reproduce the results
- Different levels of enforcement policies, chosen by user: “testing” (default: no check, only print warning), “committed”, “tagged”, “pushed”

```
WARNING:bamboo.workflow:Running with commit 8ffc100 for config and module. Please
      tag (and push) for better traceability
```

Other development ideas

- Proper type system for proxies, better operator overloading (e.g. *`RVec`* broadcasting)
- Support indexed friend trees
- Control/restrict systematic variations at the selection or plot level (current approach is “take-all”)