Readable and efficient HEP data analysis with bamboo

Pieter David[1,*]

[1]Université catholique de Louvain, Louvain-la-Neuve, Belgium

Abstract. With the LHC continuing to collect more data and experimental analyses becoming increasingly complex, tools to efficiently develop and execute these analyses are essential. The bamboo framework defines a domain-specific language, embedded in python, that allows to concisely express the analysis logic in a functional style. The implementation based on ROOT's RDataFrame and cling C++ JIT compiler approaches the performance of dedicated native code. Bamboo is currently being used for several CMS Run 2 analyses that rely on the NanoAOD data format, which will become more common in Run 3 and beyond, and for which many reusable components are included, but it provides many possibilities for customisation, which allow for straightforward adaptation to other formats and workflows.

1 Introduction

As the LHC moves to a third run, and then a high-luminosity phase, where the gain in physics reach mostly comes from the increase in the size of the data samples than from an increase in centre-of-mass energy, the focus of experimental measurements shifts towards improving the precision. Data collected over longer periods of time are jointly analyzed, and calibrations, corrections, and data analysis techniques become increasingly fine-grained and sophisticated. This also has an impact on data analysis workflows, where there is a trend towards more compact and standardized data formats, e.g. the introduction of the NanoAOD format in the CMS collaboration [1], which stores about 1 kB of information per collision event, and is foreseen to cover the needs of a major fraction of the measurements and searches.

With the increasing complexity of the experimental analyses, the computer code to perform them easily becomes inefficient, if simplicity and readability is given priority, or inflexible and error-prone due to the need to efficiently process large amounts of data. The bamboo[1] package avoids this trade off by providing a flexible python framework with a high-level interface to construct computation graphs that can efficiently be executed by RDataFrame [2] (RDF), a declarative columnar interface for data analysis in the ROOT framework [3, 4]. The core of the solution is an embedded domain-specific language, an analysis description language [5] with elements of functional-style programming. The ability of ROOT and RDF to include, compile, and call arbitrary C++ code through cling [6], with automatic python bindings through PyROOT [7], is used to provide a set of features that is currently complete enough to perform several CMS analyses in the context of top quark production measurements, searches for rare production modes of the Brout-Englert-Higgs scalar boson, and

---

*e-mail: pieter.david@cern.ch

[1]see also https://gitlab.cern.ch/cp3-cms/bamboo and https://cp3.irmp.ucl.ac.be/~pdavid/bamboo/ for the source code repository and documentation, respectively

searches for additional electroweak-scale scalar bosons, on the full data sample collected by the CMS experiment during the second run of the LHC (2015–2018).

The following sections provide a description of the embedded domain-specific language to construct derived per-event quantities and its implementation (Section 2), the construction of execution graphs through RDF (Section 3), the interface for dedicated tools to implement on-demand corrections (Section 4), and the structure of the framework to maximise the reuse of code that is not part of the analysis logic (Section 5). Finally ideas for future developments (Section 6) and a conclusion (Section 7) are presented.

2 Event view and constructing derived quantities

The defining characteristic of a declarative columnar interface like RDF is that instead of writing code that is executed for every event, or entry in the dataset, a set of transformations and operations is defined based on the columns, before reading any of the data, which are then efficiently executed by the backend. Since RDF allows to pass C++ strings that are then just-in-time compiled, filling some quantities in histograms only takes a few lines of code, in simple cases, and can be done with a single iteration over the input data. For this task, which is very common and therefore used as an example here, the main building blocks are the *Filter* node, which only considers a subset of the input data for which certain requirements are fulfilled, the *Histo1D* node, which fills the values for a column into a histogram, and the *Define* node. The latter allows to add a new column to the dataset, with its values calculated from those of other columns, and can be constructed from a C++ code string that may contain variables with the same names as existing columns, or a C++ callable and a list of input columns this takes as arguments.

While in principle any derived column could be written as a single function of the original inputs, the *Define* nodes calculate their value once per entry, and thus provide a form of caching that is essential for runtime performance. It is clear that in more complex analyses these *Define* columns will contain a significant part of the logic, and often relatively complex code. When using RDF directly, the necessary methods are typically kept in the same source file when implementing an analysis in C++, or in a separate C++ source file when building the RDF graph from python; they can also be implemented in python and JIT-compiled with numba. With formats like NanoAOD, where collections are stored as a set of array branches, keeping the book-keeping of indices, and arrays and other data structures with indices, readable for code that is maximally performant, is challenging.

Bamboo provides a higher-level interface that almost fully automates the construction of *Define* nodes in the RDF graph: instead of using the individual columns directly, it constructs an event view object, with attributes that correspond to the collections and objects stored as separate columns. All of these are placeholder objects that behave like the value type of the column or object they represent, e.g. for collections a list interface with random access and a length placeholder, and for objects an attribute for each column. This allows the user code to build up expressions for derived columns piece by piece, by composing basic operations on the event view, similarly as could be done in an object-oriented framework that processes one record at a time. The use of decorated placeholder objects to build up execution graphs is analogous to machine learning frameworks like Theano [8] and Tensorflow [9], but for dataset columns instead of multi-dimensional arrays, and based on RDF and cling.

The main difference with writing the event loop code directly is that intra-event control flow in the execution graph is not introduced by control flow in the python code written by the user, but by calling helper functions that insert basic operations. Many of these are commonly used higher-order functions from functional-style programming, for instance *select*, which takes a list and predicate, and returns a new list with the elements for which the predicate

```
muons = op.select(t.Muon, lambda mu : op.AND(mu.pt > 10, op.abs(mu.eta) < 2.4,
                                              mu.mediumId, mu.relIso < 0.4))
dimu = op.combine(muons, N=2, lambda m1,m2 : m1.charge != m2.charge)
cand = dimu[0]
m_mumu = (cand[0].p4+cand[1].p4).M()
```

Figure 1. Example bamboo code to select reconstructed muons that pass some selection criteria, construct a list of dimuon candidates with opposite charge, and calculate the mass of the first candidate.

is true; see Figure 1 for an illustration of the typical code style. Most of the information is then contained in the function arguments, which represent predicates or transformations on one or more collection elements. Inspiration for the higher-level functions was drawn from LoKi [10].

The example in Figure 1 also highlights another feature of the event view and placeholder classes: the `p4` attribute is not stored in the input dataset, but defined to automatically construct a four-dimensional momentum vector from the stored components. References between objects in different collections, which are stored as indices in the NanoAOD file, are added to the proxy classes in a similar way.

Together with the data-derived event view, the set of helper methods define an embedded domain-specific language for constructing column expressions. This approach has a number of advantages: the user-defined expressions are restricted to the set of supported operations by the python interpreter, the main analysis logic can easily be kept together in the same source file, it is very compact, and it can use the full power of python, with list comprehension, loops, and helper functions and classes, to construct similar expressions without repeated code.

The implementation defines an interface implemented by about twenty concrete classes to represent all such expressions, and about fifteen base placeholder or proxy types that wrap an expression, out of which five are only needed for automatic systematic variations, which are discussed in detail below. The concrete event and collection view classes are dynamically generated as subclasses with attributes based on the columns found in the tree, using the `type` builtin. Instances of the classes that represent operations are considered immutable once they are fully constructed. For each of them a hash is calculated that is unique for the expression they represent, which allows for fast comparisons and the identification of common subexpressions. When a derived column is used in a *Filter* or *Histo1D* node, all subexpressions that should be precalculated for reuse, based on heuristics of which operations are expensive and annotations provided by the user, are automatically added as JIT-compiled *Define* nodes in the RDF graph.

3 Selections, plots, and systematic uncertainties

For the construction of *Filter* and *Histo1D* nodes in the RDF graph a thin wrapper to insert *Define* nodes would be sufficient, but given the central role played by the different stages of selection on the input data, bamboo opts for using its *Filter* wrapper, the *Selection* class, to include some additional information, most importantly a per-event weight column. Since many corrections are applied to simulated data as weights that depend on per-event quantities, to reshape the corresponding distributions to match those observed in data, it is natural to define a subset of the input data and the corresponding weight by adding selection criteria (cuts) and weight factors in steps, as more information is exploited to obtain a smaller subset. The main method to construct such *Selection* instances is by calling the *refine* method on an existing instance, with optional lists of cuts and weights to add.

```
from bamboo.plots import EquidistantBinning as EqB
plots = []
if isMC:
    noSel = noSel.refine("mcWeight", weight=evt.genWeight)
twoMuSel = noSel.refine("dimu", cut=(op.rng_len(muons) > 1))
plots.append(Plot.make1D("dimu_M", (muons[0].p4 + muons[1].p4).M(), twoMuSel,
    EqB(100, 20., 120.), title="Dimuon invariant mass"))
twoMuTwoJetSel = twoMuSel.refine("dimudijet", cut=(op.rng_len(jets) > 1))
plots.append(Plot.make1D("dimudijet_leadJetPT", cleanedJets[0].pt,
    twoMuTwoJetSel, EqB(50, 0., 250.), title="Leading jet PT"))
```

Figure 2. Example bamboo code to define *Plot* and *Selection* instances, see Section 3 for more details

Filling a histogram with a distribution requires the columns with the quantity of interest, a binning specification, the subsample to consider, and optionally a per-event weight. The latter two define a *Selection*, so a *Plot* instance — the *Histo1D* wrapper — is constructed from a (derived) column expression, a binning specification, and a *Selection* instance. Both *Selection* and *Plot* instances need to have a unique name, which is necessary in the second case as a name for the histogram, and in both cases useful to define columns with descriptive names, and for debugging printout. The *Plot* class can also hold a set of additional options to influence the graphical display of the histogram, e.g. colours, styles, and labels, such that all code to add the histograms for a stack plot and display it are kept together. Figure 2 shows a typical example of user code to define *Plot* and *Selection* instances.

The hierarchy of *Selection* objects that accumulates selection criteria and weights not only maps very naturally on the RDF graph, it also allows for a simple generic implementation for what are often called data-driven background estimates, where a disjoint data subsample is used to model a background contribution, with a global or per-event transfer factor measured in yet another subsample or simulation, as a weight, corresponding to the probability ratio between the two selections. This is implemented through a *Selection* subclass that keeps not only the main *Selection*, but also one or more *Selection* instances for such backgrounds, with inverted selections and different weights: since the *refine* method of this subclass adapts all of the selection, no changes elsewhere in the analysis code are required to include such a background estimate. In this way it is also easily possible to fill the histograms for different background estimation strategies all at once to compare them.

Automatic systematic uncertainty variations

In almost all realistic analysis use cases, the propagation of the uncertainty on some measured quantities, or on the agreement between data and simulation, is an important task, and a major source of complexity of the framework and analysis code. For the case of histograms that was considered above, the implication is that for every histogram produced for a sample, $1 + N$ histograms now need to be produced, each of the alternative versions with some changes in the weights, selection criteria, binning variables, or several of these. A major advantage of the domain-specific language used by bamboo is that this can be done without changes to the analysis code: all that is required is marking some inputs as having alternative values. When creating *Selection* and *Plot* objects, the framework searches for such marked inputs, and generates the RDF graph to produce the necessary additional histograms. In case an input with variations is used in a selection criterion, a new branch of the RDF graph is created, where most of the nodes that are later added to the main branch will need to be duplicated, so

the size of the RDF graph may increase significantly. If a variation only affects the weight and binning variable, only a few nodes need to be added. While this is foreseen to become directly supported in RDF, the efficient implementation of systematic variations, both code-wise and in runtime performance, is currently one of the most compelling features of bamboo.

4 C++ extensions

In addition to the specific code described above, typical analyses also need some building blocks that can more efficiently be implemented directly in C++. Typical examples include loading additional weights or efficiency corrections from a file, inference of multivariate classifier using a framework for machine learning, the calculation of variations of some branch values, additional reconstructed based on stored event quantities etc. Since many of these can be widely shared across analyses, and most of the current bamboo users are analysing CMS NanoAOD, C++ implementations for several commonly used tasks are bundled.

Currently most efficiency corrections and weights are stored in a common JSON format that was developed for an earlier framework, for which a C++ reader class is included. A module to reapply jet energy corrections, and to calculate variations of the reconstructed jet energy that correspond to the uncertainty on the jet energy correction and resolution, based on the values stored in the NanoAOD, is also provided, including the propagation of these variations to the reconstructed missing transverse momentum. This is a nontrivial and crucial ingredient for producing results that include all relevant systematic uncertainties based on the centrally produced samples — in many other frameworks for NanoAOD analysis an additional processing step is needed, which stores the variations as columns in the file. The possibility to apply, and easily change, corrections to the centrally produced samples makes bamboo particularly suited to provide feedback about new corrections and recipes, and to derive these, provided that the necessary inputs are stored.

For the evaluation of multivariate classifiers using external machine learning frameworks, a set of thin wrappers are provided with a very similar interface: a class that is constructed with the path of the weights file and additional information, if needed, with a method that takes the values for the features for one example, and returns the value or values of the output classifier. Such classes are currently available for TMVA [11] (through the `TMVA::Experimental::RReader` class included in ROOT), lwtnn [12], Tensorflow [9] (through the C API), PyTorch [13] (using TorchScript), and ONNX-Runtime [14]. Most of these support multiple input nodes, with potentially different types, and are thread-safe, such that implicit multithreading [15] can be used. The performance may in the future be improved by evaluating on a batch instead of a single example at a time.

Technically, all the above follow a similar scheme: they are compiled and linked as shared libraries, which are dynamically added to the ROOT runtime, together with the corresponding headers. As much configuration as possible is passed directly to the class constructor, but in some cases the automatic python bindings provided by PyROOT are used to pass additional settings by calling member methods on an instance from the python code. The same mechanism is used to implement unit tests that are integrated with the pytest-based bamboo test suite, and PyROOT introspection is essential for finding the return types where C++ methods are called.

The C++ template functions that implement basic functional algorithms are defined in a header, and passed to ROOT for JIT compilation. Many of these are range versions of STL algorithms, e.g. the *select* algorithm used as an example before is essentially `std::copy_if` on a list of indices, so they may be changed to use, or be replaced by, their standard counterparts in C++20.

5 Task management

The bamboo framework is designed around the usually most resource-intensive task of applying selections, calculating derived quantities, and filling histograms and skimmed datasets. Since these are performed by the RDF event loop based on the graph built by the user code, which is presented through the uniform interface of the `definePlots` and `defineSkimSelection` methods that the user module should implement, the framework can provide a shared implementation for processing multiple samples, locally or on a batch system.

The list of samples are specified in a YAML configuration file that is passed to the `bambooRun` command-line script together with the name of the user analysis module. Each sample entry needs to have a unique name, which is used as a name for the output file, and specify how a list of files can be retrieved: this can be a list of paths or URLs in the configuration file, the path of a text file that contains these, or a query string for a sample management database, e.g. the CMS DAS system [16]. In the latter case it is also possible to keep a local disk cache of the query results, to avoid executing them at every run.

The two actions performed by `bambooRun` are then obtaining a ROOT file with histograms or skimmed TTrees for each sample, and postprocessing these to obtain e.g. combined plots. The latter can also be run separately, such that changes that do not require recreating the results files, e.g. layout changes, or different labels, can be applied in little time. How the former is organised, is decided based on the additional arguments: by default, the samples are processed sequentially. The most common option is using a batch system to submit one or more jobs that each process one sample, or part of a sample — the splitting is then defined in the YAML configuration file — with the `--distributed=driver` option. The commands for the individual tasks are almost identical, but with `--distributed=worker` and some additions to specify which sample and files to process. Since batch jobs may fail for various reasons, the driver mode checks periodically for the status of all the jobs, merges the results for split samples, and prints out information about failed jobs, including a command to resubmit those after solving the problem, until all of the tracked jobs have finished. A `--distributed=finalize` mode is also provided, which checks for the per-job results, and performs any remaining merging if some jobs were manually resubmitted. Both the slurm and HTCondor batch submission systems are supported, with the option to specify some site-specific configuration settings. A plugin architecture is used, such that support for other batch systems can be added with limited code changes. A recent addition to RDF is the possibility to trigger the execution of multiple graphs in parallel from the same process. This can be accommodated as an additional distributed mode without requiring changes to the user code.

The postprocessing step by default does not nothing for skims, and makes stack plots for histograms, which is done with the standalone plotIt tool: almost all the required information is already available in the bamboo configuration file. The main additional pieces that are otherwise not needed by bamboo are the scaling of simulated samples according to the cross-section, corresponding data luminosity, and the size of each simulated sample. The former two are specified in the YAML configuration file, while the latter can also be calculated automatically based on the in-file metadata for NanoAOD.

6 Future directions

The bamboo framework already offers many options for customisation beyond the most common workflows. An example that is actively used is custom postprocessing: the default action is to produce stack plots for one-dimensional histograms and ignores other outputs, but it only takes a few lines of code to also make plots for two-dimensional histograms, convert

and rename all outputs, or register produced skims with a database, since all the necessary information is readily available. Some illustrations of processing other data formats than NanoAOD can be found in a repository with examples on CMS and ATLAS open data [17], which are based on the corresponding RDF tutorials. These also show the potential for analyses with less complexity than measurements with the latest datasets and corrections: for studies on open data, sensitivity studies on simulation, and phenomenological analyses, the analysis code may in many cases fit in a few screens.

An interesting side-effect of the embedded domain-specific language and its implementation is that, although it maps very naturally to the RDF execution graphs, it is it avoids most direct calls of user code to ROOT and RDF directly. This has also been observed while implementing a backend that does not build the RDF graph, but generates source code for a standalone RDF-based C++ program that does this: the dynamic features provided by ROOT are in that case only needed for loading libraries and headers, and introspection through PyROOT, and for directly passing configuration options to C++ classes, which is only done in a few places. That opens up possibilities beyond the current paradigm of batch-processing large data samples and producing many thousands of histograms at once, but instead caching results from previous runs, and only rerunning what is needed, or passing queries to a dedicated system that optimizes the caching of columns and datasets — the unique hash of each derived column is currently an implementation detail, but it could become an essential element in such a system.

7 Conclusion

The bamboo framework provides a convenient way to analyse collision data, especially when using flat TTree formats like CMS NanoAOD. It separates the user code, which is kept as simple, compact, and readable as possible through an embedded domain-specific language, from its efficient execution, which takes advantage of the possibilities provided by RDF and cling. The latter is currently optimised for processing on batch systems, but the architecture allows for the different components to evolve separately and thus adapt to evolutions of the computing environment.

Acknowledgements

References

[1] A. Rizzi, G. Petrucciani, M. Peruzzi (CMS collaboration), *A further reduction in CMS event data for analysis: the NANOAOD format*, in Proceedings of the 23rd International Conference on Computing in High Energy and Nuclear Physics (CHEP 2018), *EPJ Web of Conferences*, **214** (2019), 06021, 10.1051/epjconf/201921406021

[2] E. Guiraud, A. Naumann, D. Piparo, *TDataFrame: functional chains for ROOT data analyses*, v1.0 (2017), 10.5281/zenodo.260230

[3] R. Brun, F. Rademakers, *ROOT: An object oriented data analysis framework*, Nucl. Instrum. Meth. A **389** (1997), 81–86

[4] F. Rademakers et al., *root-project/root*, v6.20/06 (2020), 10.5281/zenodo.3895852

[5] S. Sekmen, P. Gras, L. Gray, B. Krikler, J. Pivarski, H.B. Prosper, A. Rizzi, G. Unel, G. Watts, *Analysis Description Languages for the LHC*, PoS **LHCP2020**, 065 (2021), 10.22323/1.382.0065, arXiv:2011.01950

[6] V. Vassilev, P. Canal, A. Naumann, L. Moneta, P. Russo, *Cling – The New Interactive Interpreter for ROOT 6*, Journal of Physics: Conference Series **396** (IOP, 2021) 5, 052071, 10.1088/1742-6596/396/5/052071,

[7] W.T. Lavrijsen, A. Dutta, *High-performance Python-C++ bindings with PyPy and Cling*, in *6th Workshop on Python for High-Performance and Scientific Computing (PyHPC)* (IEEE, 2016), 27–35, https://cppyy.readthedocs.io/en/latest/

[8] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, Y. Bengio, *Theano: A CPU and GPU math compiler in Python*, in *Proceedings of the 9th Python in Science conference* (2010), 10.25080/Majora-92bf1922-003

[9] M. Abadi et al., *TensorFlow: Large-scale machine learning on heterogeneous systems* (2015), software available from tensorflow.org

[10] I. Belyaev, *LOKI: Smart & Friendly C++ Physics Analysis Toolkit*, http://cern.ch/lhcb-comp/Analysis/Loki

[11] A. Hoecker, P. Speckmayer, J. Stelzer, J. Therhaag, E. von Toerne, H. Voss, *TMVA: Toolkit for Multivariate Data Analysis*, PoS **ACAT**, 040 (2007), CERN-OPEN-2007-007, arXiv:physics/0703039

[12] D.H. Guest, J.W. Smith, M. Paganini, M. Kagan, M. Lanfermann, A. Krasznahorkay, D.E. Marley, A. Ghosh, B. Huth, *lwtnn/lwtnn*: version 2.11.1 (2020), 10.5281/zenodo.4310003

[13] A. Paszke et al., *PyTorch: An Imperative Style, High-Performance Deep Learning Library*, in *Advances in Neural Information Processing Systems 32* (2019), pp. 8024–8035, arXiv:1912.01703

[14] *ONNX Runtime: Optimize and Accelerate Machine Learning Inferencing and Training*, https://www.onnxruntime.ai

[15] D. Piparo, E. Tejedor, E. Guiraud, G. Ganis, P. Mato, L. Moneta, X. Valls Pla, P. Canal, *Expressing Parallelism with ROOT*, J. Phys. Conf. Ser. **898** (2017) 7, 072022, FERMILAB-CONF-16-738-CD, 10.1088/1742-6596/898/7/072022

[16] G. Ball, V. Kuznetsov, D. Evans, S. Metson, *Data Aggregation System: A system for information retrieval on demand over relational and non-relational distributed data sources*, J. Phys. Conf. Ser. **331** (2011), 042029, FERMILAB-CONF-11-874-CMS, 10.1088/1742-6596/331/4/042029

[17] P. David, *bamboo Open Data examples*, pieterdavid/bamboo-opendata-examples