# 5

# End User Analysis for Particle Physics Computation

**G. S. Davies, P. Onyisi, A. Roberts**

(contributors from the community)

## 5.1 Introduction

What do we talk about when we talk about analysis?

## 5.2 People do software work

### 5.2.1 Problems

- Lack of long-term support for software efforts (grant cycle is three years)
- Software efforts are siloed by collaboration - it's rare for people to be funded for cross-experiment software efforts. E.g. if Peter writes a package that another collaboration wants to use, it is not a common pattern for that collaboration to fund him for that support. Need mechanisms for people to get partial support to work beyond their immediate use cases. And also for a field to say, "yes, this software is important!"
- Recognition that supports stable careers for sustainable software work
- Misalignment between what we need for good software and what the field recognizes as valid work
- What else?

### 5.2.2 Personnel support case studies

**PDG** The PDG is recognized as important despite not being innovative. We all expect the PDG to be there.

**ROOT** They don't have funding cycles?

**XSEDE ECSS**

**SGCI**

## 5.3 Analysis Ecosystems: Libraries, Languages, and Data Formats - Peter

No analysis software functions entirely on its own; any package is situated in the context of the input data it consumes, the output data it produces, the other software it depends on, and the way it is configured or embedded in other code. Because of this we talk about "software ecosystems," groups of packages which are typically used together.

Packages in an ecosystem typically have common data interchange formats and similar programming language interfaces. In some cases they may be distributed together as a single metapackage with overall versioning. There are two major ecosystems in HEP:

- **ROOT**: hosted by CERN, the ROOT suite is a tightly-integrated set of libraries that cover a broad range of HEP analysis needs, including I/O, event loop execution (including a parallel distributed mode), histogramming, fitting and statistical analysis, and visualization. The libraries are written in C++ and that is still considered the primary language for its use, although the Python bindings (PyROOT) are very well supported and by construction expose essentially the full API. Bindings for the R language are also currently supported. The ROOT libraries were developed to meet the specific needs of HEP experiments and as such provide solutions that are well-matched to HEP analysis problems, although this also means that use outside of HEP is extremely limited. The tightly-bound nature of ROOT means that using alternative software for any particular functionality can be very difficult. ROOT is undergoing a redesign ("ROOT 7") which aims to improve interfaces with the benefit of modern C++ and the benefit of over 25 years of practical experience.

- **Python**: this is a somewhat loose term for a set of tools, with Python as the primary language interface, introduced with the primary goal of enabling the use of software developed outside of HEP, in particular for machine learning. This ecosystem is still in development and has no single governance team. It tends to emphasize independent packages for different aspects of the analysis pipeline (so, for example, I/O is handled with a different package from histogramming). Development teams in this ecosystem are typically small and feature junior personnel.

### 5.3.1 Programming Languages

Users interact with software libraries and packages through programming languages. These can be separated into general-purpose languages (GPLs) which can be used for any task, and domain-specific languages (DSLs) (ROOT TCut, PAW kumac, etc.) which provide a restricted set of higher-level primitives which simplify certain operations. The two most commonly-used general-purpose languages in HEP are C++ and Python.

Examples of domain-specific languages are the TCut syntax used for applying selections and constructing new variables in ROOT, and the kumac language used to control the old FORTRAN-based PAW suite.

General-purpose languages, by definition, are extremely capable and are used to solve problems outside of HEP. Exposure to these languages is one of the major technical skills that is transferable outside the field. It is considered necessary for HEP students to develop familiarity with, and preferably proficiency in, at least one general-purpose language. It is not necessarily the case that the languages that are used in HEP are the ones prevalent in the industries that particle physicists transition to (for example, R is widely-used in data science and virtually unknown in HEP). Because of their complexity, the time it takes to train personnel in them, and the need to transfer responsibilities for maintaining code from one person to another, the diversity of general-purpose languages used in the field is strictly limited. By contrast, domain-specific languages historically have been easier to master due to the limited range of constructs available.

HEP has had relatively few general-purpose languages in recent history. Until the early 2000s FORTRAN was commonplace. A desire to move to more modern and commonly-used languages drove a transition across the field to C++ (although there was competition, notably from Java); this was generally a top-down move imposed as new experiments were built or experimental upgrades were implemented. Often experiments found it valuable to use a second general-purpose language such as Python or tcl as a high-level scripting system for their data processing code. Python in particular came to be adopted by users as a convenient language and its simultaneous adoption as the standard language in machine learning has driven massive bottom-up adoption of the language. Availability of library bindings in various languages is extremely important for their use; both C++ and Python raise significant barriers to using libraries written in those languages elsewhere, although thanks to a lot of work the border between those two specific languages is relatively low.

Python is not necessarily an optimal language for scientific computing. In its reference implementation, it is a fully interpreted language, making it much slower than C++ for many tasks. The speed issue creates a programming paradigm in which users express operations via intermediate libraries (such as numpy for data manipulation or TensorFlow for neural network construction), introducing what amount to mini-languages embedded in Python. For this reason there is interest in exploring languages that can combine the expressiveness and ease-of-use of Python with compilation to machine code; the most commonly-explored option is Julia. However it is clear that introducing another general-purpose language in HEP will require a very compelling case and it appears that the status quo regarding Python will continue for the foreseeable future, perhaps including the adoption of acceleration technologies such as numba.

Visualizations are increasingly being done via web browsers (such as Jupyter notebooks or JSROOT). By far the dominant language in that environment is JavaScript, which is a language which particle physicists generally have very little experience in, and one where best practices have evolved very rapidly. If critial parts of the analysis ecosystem are written in this language, expertise will need to be maintained at some level.

Domain-specific languages in HEP span a range of applicability, from specifying variable construction for specific histograms to the construction of statistical analysis to a complete event selection workflow [1]. In a meaningful sense this includes the expressions needed to control numpy/AwkwardArray/TensorFlow from Python, or to express operations using the ROOT RDataFrame; although formally these are library operations, analysts are expected to learn how to express their intent in terms of high-level operations while the details of execution are kept intentionally opaque.

Domain-specific languages have the advantage of providing high-level primitives which in principle permit optimization of the actual execution of the code. In particular this includes parallelization and acceleration, operations which analysts may be uncomfortable with or which require significant investment to implement

properly but which can provide speed and capability improvements when available. The major disadvantage is the (usually) restricted scope of operations that can be expressed in DSLs, which are typically designed with specific tasks in mind and which may make it unnatural or impossible for users to do other things. This is particularly dangerous for DSLs that operate at the "analysis description" level; it is not clear that such languages can generalize reasonably between frontiers.

Users can come to regard DSLs as primary parts of the analysis interface and the learning curve for them can be high. In particular, if there is more than one DSL relevant to a specific task, users may prefer to learn only one. If DSLs are linked to specific libraries or ecosystems, the synergies are liable to tie users to those technologies.

### 5.3.2 Data Formats

Analysis data come in many forms:

- **Event data:** these consist of information, typically with a fixed but complex schema, describing individual events.
- **Histograms:** These summarize features extracted from event data.
- **Other summary data:** There are forms of summary data that cannot reasonably be expressed via histograms: sometimes tabular data is a better fit and more space-efficient, and sometimes the schema for the data is sufficiently complex that it makes sense to store a sui generis kind of object (such as for the results of fits).
- **Configuration data:** The configuration for running some software may be stored in a human-readable and -editable format, or in a binary format — the latter is especially common when one package is configuring the operation of another. In either case, interacting with the stored configuration requires data access, and it may be possible to alter the configuration like any other kind of data.
- **Metadata:** For end-user analysis, this tends to primarily be provenance tracking information.

File formats can describe both the overall container for data and the specific types of objects that can be stored. ROOT separates these fairly strictly, in that the ROOT file format can store essentially any C++ object, and ROOT objects can be serialized to formats other than ROOT files (such as JSON or XML). ROOT provides a number of pre-defined data objects, such as tables (known as TTrees) and histograms, and multiple objects can be present in the same file. Other data formats allow less freedom; for example, Apache Parquet merges the container and the data object and is only suitable for tabular data, while HDF5 files allow for a specific set of contained structures.

It is important to note that analysis end-users very rarely interact directly with underlying file formats — they are interested in the in-memory transient representation of data, rather than the persistent format, and the translation between the two is handled by libraries. The capabilities of specific formats may limit what users can do, and certain formats may provide more optimized storage, but otherwise the details are generally hidden from users. Therefore transitions in data format are easier to handle than those in libraries or languages. Newer versions of ROOT include the capability to read in data in CSV text format, sqlite files, or Apache Arrow.

### 5.3.3 Visualization

End-user analysis relies critically on visualization to give feedback to the physicist. Plots allow the user to quickly understand characteristics of the data but also to debug code and workflow problems.

Previous generations of analysis libraries supported interactive visualization through native graphics libraries; remote use involved the use of technologies such as remote X Windows. Over high latency links this can be extremely difficult to use and requires specific software to be installed on the user's machine. Recent trends exploit the near-universal availability of web browsers following common standards to offload rendering and interaction to the user's browser. This results in a more uniform experience across platforms and reduces external dependencies. This is the standard mode for code in Jupyter notebook environments (in particular the Python ecosystem) and is the baseline for future ROOT graphics.

The ROOT ecosystem's visualization libraries are naturally matched to the specific HEP use case. Visualization in the Python ecosystem is typically handled through the Python matplotlib package, a standard for scientific plot creation; matplotlib does not directly support a number of common HEP plot forms or histogram structure formats, so additional libraries have been written to help bridge these boundaries.

Although event displays are not typically used as part of an analysis workflow, they are still of interest to end users, and the same issues apply. Work is being done on experiment-agnostic event displays that render in browsers using JavaScript.

### 5.3.4 Requirements for a Sustainable End-User Software Ecosystem

A choice of software stacks is becoming available for HEP analysis. It is generally thought that the ROOT and Python stacks have different strengths and will competitively coexist into the future.

In order for the ecosystems to achieve long-term, sustainable success, we note the following requirements:

- **Support of Personnel.** Unsupported software projects undergo "code rot" over time, a process where changes external to the package itself cause it to lose functionality. In the Python ecosystem, for example, the migration from Python 2 to Python 3 rendered old versions of packages unusable. For this reason alone, it is mandatory that any software that forms a key part of a HEP analysis ecosystem must have a maintainer who is able to provide the necessary level of support for the package. The community must understand that maintenance is a task of equivalent importance to developing new code, and recognize people's work appropriately. There must also be a mechanism for transferring responsibility for a package as necessary.

- **Documentation and Training.** Analysis software ecosystems are used as a gestalt — not as a disconnected set of packages.

- **Interoperability.** Ecosystem lock-in can be a problem for multiple reasons. Packages that can only be used in a single ecosystem will not provide benefits to users not in that ecosystem.

## 5.4 Analysis Models

Once analysis code is written, it must be run on data. An analysis pipeline may involve multiple stages of data reduction and different codes, executing on very different platforms.

Analysis users value fast turn around — being able to quickly answer physics questions. Because the rapidity with which analysis workflows can complete is paramount, data processing architectures which are well-suited to managed production workflows may not match well on to analysis tasks. Users often desire to run analyses on hardware that is under their control, and in fact such capability may be essential to allow users to develop, test, and debug their code.

### 5.4.1 Scale

Users need to be able to scale code execution from a few events (for testing) to an experiment's full dataset (for actual analysis). The former requires interactive response, while the latter may require distributed execution (on a single cluster or across multiple sites). Interactive use historically has occurred via terminal sessions and visualization software native to the particular operating system and environment. Distributed execution has occurred on batch systems, either single-site or multi-site; in the latter case, especially if a federated computing model is adopted, complex issues of data locality and access, bookkeeping, job brokering, fair access, and so on arise.

Particle physics problems are usually *high-throughput*, not *high-performance*, problems: that is, they consist of a very large number of fairly lightweight computations which are essentially independent of each other, and so do not require computing resources to appear as a single, very powerful image (as on a traditional supercomputer). Traditional tightly-coupled supercomputer execution environments such as OpenMPI are therefore not typically needed for HEP analysis applications and in fact may be detrimental as they do not exploit the fine granularity of HEP problems. However, the increasing exploitation of coprocessors and accelerators (such as GPUs) in HEP code requires libraries that couple CPU and GPU execution on a single node.

Solutions that smoothly scale from small-scale interactive tests to full-data processing are desirable. In the absence of such solutions users face a barrier during the development of their analyses which can be quite substantial. Such solutions need to abstract away the execution of the event loop so that it can be executed on whatever resources are available, transparently to the user. Therefore by necessity they restrict the form of the user code to some extent.

### 5.4.2 Interfaces

The "traditional" HEP analysis execution environment is a terminal. GUI applications add significantly more coding complexity and generally limited benefits, although they have been used (for example, in the ROOT PROOF suite, which still generally is invoked via a terminal). The use of a terminal allows terminal scripting languages, such as bash, to be used to orchestrate a workflow.

There is a recent trend towards using *notebooks*, particularly those provided by the Jupyter environment, as the main interface for user code execution. Jupyter notebooks, which are rendered in a browser but with a connection to a backend kernel at the actual code execution site, function essentially as recorded

interactive terminal sessions with additional documentation and output visualization capabilities. Jupyter notebooks are ill-suited for execution as actual code and are instead primarily used to script libraries which (for example) spawn worker tasks to actually perform the requested computations.

## 5.5 Dataset Bookkeeping and Formats - Amy

- Raw data: overwhelmingly, custom binary formats that are specific to an experminent
- Processed and/or skimmed data: ROOT, always and forever, and we all know it
- Intermediate files that facilitate tool use: Parquet, Zarr, etc. HDF5 is rarely used, its libraries are a bit painful and all the tools people want to use work with file formats that have friendlier IO libraries
- Metadata (file metadata, run information, processing states): Databases, stand-alone text files, wikis. Metadata is also often included in the data file.

## 5.6 Collaborative Software - Gavin

T. Aarrestad et al. [HEP Software Foundation], "HL-LHC Computing Review: Common Tools and Community Software", arXiv:2008.13636 [physics.comp-ph ]] (pdf).

Simone Campana, Alessandro Di Girolamo, Paul Laycock, Zach Marshall, Heidi Schellman, Graeme A Stewart. "HEP computing collaborations for the challenges of the next decade", arXiv:2203.07237 [physics.comp-ph] (pdf).

Dave Casper, Maria Elena Monzani, Benjamin Nachman, Costas Andreopoulos, Stephen Bailey, Deborah Bard, et al. "Software and Computing for Small HEP Experiments", arXiv:2203.07645 [hep-ex] (pdf). (also under EF0, NF0, RF0, CF0)

Code management, version control, Messaging, telework, Q&A platforms, forums ,wikis.

Slack, Discord, Zulip

Containers are an important part of job submission. Some pieces of software are so platform dependent that it's sometimes nearly impossible to replicate the framework on another machine.

Tools for collaboration

git Many people love git Some people feel that more git training is needed

communication Almost all use some kind of instant messaging (slack, skype, etc.) Fewer use bug tracking/project management

Person-to-person & group communications Video & voice - Skype, Zoom, Vidyo, ... Synchronous chat - Skype, Slack, Zulip, Mattermost, Discord, ... Asynchronous text - email, Discourse, ... Do these platforms meet our needs? Should we seek better integration? Should we try to standardize across HEP? Should we prefer centralized (e.g. lab-hosted) or decentralized (user-controlled) models? Accessibility issues Overlap with training & documentation how much of "how to do things" lives in private or hard-to-search channels? How much do these tools enable "chatting with colleagues" and can we improve that?

Should this include a discussion on software citations, such as via Zenodo or such?

## 5.7 Training - Amy

We do not have the expertise we need for the computational work of upcoming physics.

Question: where can this work come from?

- Collaborations with industry are rare (are they, though, for GPU work?).
- Collaborations with computer scientists are tougher than they appear. Computer Scientists have a higher pay scale. Also, much of our work is not interesting from a computer science perspective.
- We can grow our own. There are serious barriers to this. (1) Curriculum change - or at least asking students to double major - is a big ask. (2) Stable career paths don't currently exist. Research software engineers are often supported on soft money and computing work is not typically valued in the promotion or tenure process.

Training within standard physics curriculum (none) Training within groups (this burdens groups without access to computing expertise) Training within collaborations (a good argument for analysis reproducibility is its future usefulness for training) Training within fields (just getting started, Software Carpentry, Data Carpentry, FIRST-HEP, DANCE-Edu) Industry training opportunities

## 5.8 Thoughts

Must have support mechanisms for both bottom-up and top-down development & long-term support

# Bibliography

[1] H.B. Prosper, S. Sekmen and G. Unel, *Analysis Description Language: A DSL for HEP Analysis*, in *2022 Snowmass Summer Study*, 3, 2022 [2203.09886].