The future of HEP software

René Brun
CERN, Geneva, Switzerland
E-mail: rene.brun@cern.ch

Abstract.
Any attempt to predict any future subject includes in general two components: extrapolations of the current trends and discussions about possible surprises. The extrapolation of the trends based on the latest function derivatives seems to be the easiest one. However, one has to be careful with fashionable trends and associated hype peaks. When one looks at the HEP software in the past 40 years, one sees regions of stability, often coinciding with the development and exploitation of a large accelerator, but also big changes when the hardware or/and software technology permits large steps. These large steps are often associated with surprises unthinkable a few years before. These two observations will likely continue to be true for the coming few decades, if not more. This paper is an attempt to analyze the current function derivatives based on similar analysis many years ago, and also propose some directions and developments that may become key components for our future software.

1. Introduction: How did we reach the current situation?

The very beginning of software in HEP was driven by the main detectors at that time, i.e. bubble chambers. Pictures of events were scanned and converted into disk or tape files. The bubble chamber community was well organized and succeeded in producing popular software packages like Hydra, Thresh, Grind. Around 1970, the first electronic experiments appeared with the development of the Multi-Wire proportional Chambers by G. Charpak. These experiments were small, between 10 and 100 physicists, typically working on the detector developments. Simulation did not exist and the software was in general developed after the data taking and was in the hands of very few people. Things started to change rapidly with the SPS experiments at CERN or SLAC, LBL and BNL. Software developers from some experiments started to join forces to develop a detector simulation kernel as well as common frameworks. The main simulation tool EGS [2] had been developed at SLAC for electromagnetic showers and a few hadronic shower simulators were also appearing. The online software was running on 16 bits machines, the most popular being the PDP11. The offline software was implemented on mainframes like the CDC6600. The software was implemented in assembler for the time consuming tasks or Fortran. Programs were limited by the amount of memory on the machines, typically less than 1 MByte and the data were stored on tapes. An analysis program was in general less than 2000 lines of code or punched cards. The simulation system **GEANT1** appeared in 1974, followed by **GEANT2** in 1975. The **HBOOK** system was developed in 1973 and was rapidly adopted by the software community. In 1980, a major step in the simulation software came with **GEANT3** [1]. A general detector description and tracking system was implemented in a few weeks in the early days of the OPAL experiment, as well as a more powerful interface with **EGS3** and hadronic packages like **Tatina** or **Gheisha**. An interactive version of HBOOK

called **HTV** appeared with the first work stations and popular interactive operating systems like VMS on the VAX780. The development of two important systems was started in 1984: the data structure management system **ZEBRA** [10] (a coalition between the ZBOOK/HBOOK and Hydra developers) and the Physics Analysis Workstation (PAW). ZEBRA was implemented in just a few weeks. With ZEBRA it was possible to create complex tree/graph data structures in memory, a big progress compared to the very primitive Fortran common blocks. These data structures could be saved with the ZEBRA **FZ** package on sequential files or direct access files with **RZ** in a machine independent format. Machine independence was a very important factor because of the variety of machines with different machine word lengths or byte swapping conventions. The first version of **PAW** [6] was also implemented in a few weeks, based on HBOOK, HTV and ZEBRA. GEANT3 was also converted to ZEBRA and was rapidly adopted by a growing number of experiments, in particular the LEP experiments. With the birth of ZEBRA, GEANT3 and PAW, the software support group was substantially extended in order to support a rapidly growing users base. In 1988 an important step appeared in PAW with the support for the storage and interactive analysis of the *column-wise-ntuples*, a key feature for the success of the project. Between 1990 and 1994, many committees appeared to discuss the possible move to Object-Oriented languages and commercial data bases. This interesting period saw also many discussions about the type of computers to be supported in the future. Following the successful use of farms of workstations, an attempt to use massively parallel systems such as the CM5 from Thinking Machines proved to be more difficult than originally thought and the revolution came in 1994 with the advent of the **Intel** Pentium-pro based machines. In 1995 the first version of **ROOT** [9] was implemented in just a few months and publicly demonstrated in September 1995. ROOT was strongly based on the same and successful concepts of PAW (histograms, graphics and data structures on files). The **CINT** C++ interpreter was added in 1996. ROOT files became self descriptive with the classes information saved on the files in 1998 and support for automatic schema evolution. CINT was upgraded to **CLING** in 2012 with full support for modern C++. In 1995 the GEANT4 [5] project was launched. GEANT4 was essentially a translation in C++ of the successful GEANT3 algorithms and in continuous development. Several alternatives to the non-official ROOT were proposed in 1996 or later in C++ or Java. They all failed for many reasons described in several articles or books like "**From the Web to the Grid**" [12]. As an active participant of this long software development period starting in 1970, I have analyzed the evolution of all systems described above, the reasons of their success today in most HEP experiments and also the many problems associated with their development:

- The proof of principle, first version and real use in experiments were implemented in just a few months. This short term principle is important because it gives self confidence to the project members, but also for the potential users and the management. During the celebration of the 30 years of the Web, I was reading again the original proposal by **Tim Berners Lee** stating that the initial phase of the project will take about 3 months, and if this phase was successful, a second phase with a duration of 3 months could be scheduled.
- The authors had, in general, direct experience with the experiment software: simulation, reconstruction or/and data analysis. This requirement is more and more difficult nowadays because of the hiring process and also, as we will see later, that only a small fraction of physicists working for their PhD will stay in HEP.
- The support was given in nearly real time: phone or instantaneous answer to emails. This point is crucial for users. During about 15 years the main support for ROOT was given via the mailing list **roottalk** with about 10000 people connected. A mail sent to roottalk was immediately visible by all, including the developers. The fact that the developers were answering nearly in real time was given confidence to users that support was really provided. Of course, a large mailing list like roottalk is not fashionable today. It has been replaced

by a forum divided into several sub chapters. The forum statistics show that less than one per cent of the registered users read a posted subject, and in general answers appear after one day or more. When considering the success of the social networks, one could imagine a new form of support exploiting this more interactive technology.

2. Target young students

During the past century most physicists were continuing their career in the Academic world or national research organizations. The situation has substantially evolved over the past 15 years. The theme is now "from Academia to Big Data". Physicists with experience in data analysis using the latest machine learning techniques and tools have no problem in finding attractive jobs outside HEP. As a result less than ten per cent of young physicists continue their career in HEP nowadays. Of course this has many consequences for the field. Some consequences are positive because young students may be attracted by the HEP domain as a very valuable training phase. But this trend may also generate more conservatism within the HEP community and this could generate negative consequences for the medium and long term. There is however a possible solution to this problem. Since a few years we witness the growing success of the **MasterClasses** [11] projects in many countries. The idea is to organize a one or two days visit to a HEP laboratory or national institute for pupils in High Schools or undergraduate students. Each student has a laptop where he/she can visualize detectors and events and make queries and selections on a data base of events. Most participants do not have deep knowledge about the HEP field and have only a rudimentary experience with programming languages. Masterclasses have attracted about 40,000 participants in 2018 and this number is expected to grow significantly in the coming years. Large experiments welcome about 3000 physicists. Large projects like ROOT have a users base estimated at about 30,000 people. But students are millions! It could be a fantastic opportunity to develop simpler interfaces, interactive languages, data exploration with graphics tools, intuitive navigation in large and structured data bases to familiarize students with these tools. This new approach will have many positive consequences:

- Familiarize future developers and scientists with the HEP domain.
- Give more feedback to our community of software developers and improve these products for everybody.
- Enlarge the community of users and developers, even if students choose later very different fields.

With multi-millions lines of code software systems, structured in tens or hundreds of libraries, our HEP systems include gradually more and more sophisticated techniques and tools, like **cmake** and **python**, interface with language introspection systems of compilers or OS specific tools for their distribution. These techniques require highly qualified experts to deal with the increasing size and complexity. This is fine for skilled software developers or experts supporting the experiment's frameworks. But it looks already pretty complex for the vast majority of physicists. If one looks at the trends in many fields, it is one of the main requirements to simplify the installation procedure. If I want to download an application on my mobile phone, I do not have to know about cmake or Python and the one-button-click procedure is executed in seconds. Systems line ROOT, GEANT, used by several thousand people are used on a few operating systems (linux, macOS and Windows) with possibly a few versions of the compilers. This means that with only 1000 people installing ROOT from source there are about 100 identical libraries versions. Of course, not everybody installs the full set of the libraries, but given this large number it should be possible to consider a publish/access facility with a certification system (security, anti-virus) such that any new user can immediately download a binary from the closest site. And it must be possible and easy to go back to an older version if a problem is suspected with the latest version.

3. Evolution of existing large HEP software projects

The simplification of the installation procedures described above is a pre-requisite for a wider distribution of our general scientific packages GEANT and ROOT. These two systems have been continuously developed over the past 25 years, with more and more functionality and adaptation to the latest OS or WEB technologies. I am very impressed by the quality, enthusiasm and the new ideas of the developers in the ROOT project. They are continuously testing and adapting the system to new techniques (I/O, graphics, math libraries) and providing support for a growing users community. The main implementation language is C++ and ROOT is taking advantage of the constant developments of this language. However, C++ has several drawbacks. In this short note I will concentrate on 4 problems only:

- The language does not provide a reflection system describing the class structure at run time. As a result, special software like **rootcint**, **rootcling** had to be developed to parse header files or special compiler specific interfaces, generating the so-called class dictionaries with the corresponding structures saved in shared libraries. There is, however, some hope that a standard facility could be implemented in the next version of C++.
- Before C++11 pointers were reference pointers only. There was no concept of structural pointers as we had in our old systems like ZEBRA. With structural pointers, it was easy to navigate in a complex tree/graph structure of objects. Memory leaks did not exist (or double/multiple deletes) because when deleting an object, it was possible to delete at the same time all structure pointers contained in this object. With C++11 the *unique_ptr* type was introduced, solving partially this problem for volatile data structures in memory. However, without a reflection system, it is quite impossible to use this new type when doing I/O.
- C++ pointers are absolute. When copying blocks of memory from a main processor to an associate processor like a GPU and back, one must navigate in the data structure to add/subtract offsets to the pointers. Having relocatable pointers would simplify the code and be far much faster.
- When creating a $vector < T >$ structure, for instance a vector of particles, all data members of one particle object are stored together. There is no possibility to create the Px, Py, Pz, etc. components as contiguous vectors allowing a straightforward support for vectorization. It is again the same problem that we spotted 40 years ago in implementing *column-wise* ntuples in addition/replacement of the *row-wise* ntuples. As a result, most attempts to vectorize code have failed because, in general, the gain obtained by vectorization is lost in creating structures of arrays from arrays of structures, then deleting them. This problem is quite serious because a lot of time is wasted in vectorizing subsets of large codes, giving the illusion that a typical factor 2 is gained, but lost again when the full code is run.

All these 3 problems (and others too) have been reported more than 20 years ago to the C++ gurus. A considerable amount of time is wasted by skilled developers, and will continue to be wasted, until these problems are fixed in the C++ language. However, C++ is a great and efficient language at the base of the vast majority of our HEP software.

From time to time, we see new languages appearing which are supposed to be better than C++, but they are mostly used in modest time consuming applications. **Python** is an interesting language, now used in many HEP environments. **Python** came in our field about 15 years ago as a scripting language replacing the shell script languages like **Perl**. Since a few years **Python** is the main user interface in the machine learning ecosystem with successful applications like **Scikit-learn**, **Keras**, **TensorFlow** or **Pytorch**. A similar language **GO**, claimed to be faster than **Python**, but does not seem to penetrate our field.

The success of **Python** is a clear demonstration that a quick learning language is important to facilitate the insertion of young people on one hand, but also make easier the use of more and

more sophisticated data analysis techniques. In view of the MasterClasses evolution described above, it seems important to develop (e.g. for ROOT) a Google-like style interface, familiar to everybody, to ask simple questions like "*what is the content of this file*", "*show the jets Pt distribution when at least one muon is produced*", etc.. The user could then ask for the generation of the code corresponding to the query in C++, Python, Go or any new appearing language. This interface will facilitate the introduction to our applications and also generate a more critical approach when a new language is proposed. This is somehow similar to the possibility, when having a file with a canvas object with histograms, functions, labels, etc., to generate a C++ script that can regenerate the same picture again, or a **pdf**, **gif**, **latex** file. I am convinced that the development of this proposal will be supported by the web technology companies.

4. Detector Simulation at a cross-road

When the GEANT1 or GEANT2 simulation systems were created at the end of the 70s it was unthinkable to simulate in great detail events in the small detectors of that time. A program running on big machines like the CDC7600 could not use more than 500 Kilobytes of memory and the detailed simulation was restricted to sub-detectors like calorimeters or a small set of tracking chambers. With machines with 1 Megabyte of memory it became possible de develop systems like GEANT3 in 1980. GEANT3 was a fundamental step in detector simulation, including a detector description package as well as interfaces to shower simulation packages like EGS3, Tatina or Gheisha. GEANT3 became rapidly the main simulation tool for the LEP experiments and later for the design of the LHC detectors. In 1990 it was possible to run programs using 16 MBytes of memory. GEANT4 came in 1995, including more and more packages for particle interactions during the transport. However today we continue to run our simulation systems with the same principle despite the fact that machines have now at least 10,000 times more memory than when GEANT3 was created. Many techniques have been tried to gain time using vectorization (1983, 1986, 2018) [3], [4] or parallel processing when multi-process programs became possible in 1988, then again in 1992 with networks of workstations or more recently with multi-threading or acellerators like GPUs.

I am strongly convinced that this conservatism must change substantially. On one side we see more and more experts in the Machine Learning field who are already exploiting new ML techniques in the data analysis (including simulation) fields, and also the ultra-urgent requirement to gain very large factors (10 to 100 or more) in detector simulation, reconstruction and analysis to cope with the luminosity increases of our colliders.

When looking at this evolution in the past 40 years and also based on my current work on physics models, where I use very similar techniques when colliding particles, I give below an example of algorithms that could be further developed to reach a huge gain in time and still preserving (possibly optimizing) the physics precision during detector simulation. Let's imagine a multi-million cells detector in which we transport NP particle types (about 20, all particles with a lifetime greater than one picosecond). For each of the NP particles we simulate $NEvents = NTheta * NPhi * NlogP$. $NTheta$ (say 1000), $NPhi$ (say 1000) are the number of divisions in spherical coordinates of our detector. $NlogP$ (say 100) is the number of divisions in the logarithm of the momentum in the physics cross-sections for the respective particles. We now simulate once (or few times) a full detector simulation for the $NP*NTheta*Nphi*NlogP$, ie 20*1000*1000*100 = 2 billion particles. For each particle and its corresponding sub-particles we store the result of the simulation, say about 1 KByte. As a result we build a data base of about 2 Terabytes. One can also add more complexity by adding NZ (say 10) Z colliding positions at the intersection point. Of course, this data base is structured such that for a given particle type the data are in adjacent buffers. This data base, easily portable on any laptop or computer system, is now becoming the core of our fast simulation engine. If we take an event produced by our event generator with NG particles with $theta_i$, phi_i, P_i, Z_i we have now the

choice between several algorithms, e.g.:

- A super fast simulation picks the results for each particle in the closest *theta*, *phi*, P, Z bin buffers.
- A fast simulation takes the result and adjacent results (likely based on a machine learning strategy) to combine them with interpolations or more sophisticated algorithms.

This strategy would be perfect for the recently announced persistent memories, and also portable on the HPC (High Performance Computers) combining mainframes with associated processors like GPU's or FPGA's. It requires discussions and rapid prototyping for many different simulation cases. It must be a team strategy and not just a domain in the hands of a young student.

5. Summary

When looking back at the past 40 years of computing in High Energy Physics, it is interesting to see the solution to common problems resolved with the adaptation or reincarnation of popular software systems to the latest hardware and software commodities, even when it implies changes in the computing languages. Software represents now a major investment in manpower and total cost. Common tools and libraries are used by most HEP experiments, adapting these tools to their particular needs. Continuity and long term support are vital for experiments running for 20 years or more. The future software is driven by the current trends, but also by a vision of the expected needs many years from now. The success requires enthusiastic drivers and also dynamic and enthusiastic participants. HEP software is at a crossroad. On one side young physicists or software engineers who will spend the rest of their career outside the HEP field. On the other side people with long term positions, possibly tempted by conservative attitudes, but hopefully boosted by younger generations. A growing fraction of the software will not be HEP specific anymore, e.g. graphics, user interfaces or math libraries. The software for the future is also driven by the hardware evolution. The concept of mainframe has evolved from one processor running multiple single-process jobs, to machines with many cores and jobs running on many cores in parallel, to combinations of large mainframes with associated acellerators. Quantum computers will come progressively in niche areas or algorithms where important performance gains are expected.

References

[1] Brun R et al GEANT3 – Detector Description and Simulation Tool *CERN Program Library Long Writeup* **W5013**

[2] EGS: R.L. Ford, W.R. Nelson, SLAC-210, UC-32 (1978).

[3] J.-L. Dekeyser, "Architectures et Algorithmes paralleles pour les methodes MonteCarlo en Physique des Particules" These: Universite des Sciences de Lille, U.F.R. d'I.E.E.A, 1986

[4] Dekeyser J-L, Georgiopoulos C 1988 Vectorization of the Geant3 geometrical routines on a Cyber 205, *Nuclear Instruments and Methods in Physics Research* **A264** 291-296

[5] http://geant4.web.cern.ch/geant4/index.shtml

[6] Brun R et al 1987 PAW – Physics Analysis Workstation. *The Complete CERN Program Library Version 1.07*

[7] Baroncelli T, TATINA *University of Rome*

[8] Fesefelt H, GHEISHA *Aachen III*

[9] Brun R et al 2009 ROOT – A C++ framework for petabyte data storage, statistical analysis and visualization *Computer Physics Communications* **180 12**, 2499-2512

[10] Brun R and Zoll J 1989 ZEBRA – Data Structure Management System *CERN Program Library* **Q100**

[11] https://www.physicsmasterclasses.org/

[12] Brun R, Carminati F *From the WEB to the GRID and beyond* Springer