FERMILAB-CONF-15-224-CD

ROOT 6 and beyond: TObject, C++14 and many cores.

B Bellenot[2], Ph Canal[1], O Couet[2], G Ganis[2], P Mato[2], L Moneta[2], A. Naumann[2], D. Piparo[2].

[1] Fermilab, Batavia, IL, United States of America

[2] CERN, Geneva, Switzerland

E-mail: pcanal@fnal.gov

Abstract. Following the release of version 6, ROOT has entered a new area of development. It will leverage the industrial strength compiler library shipping in ROOT 6 and its support of the C++11/14 standard, to significantly simplify and harden ROOT's interfaces and to clarify and substantially improve ROOT's support for multi-threaded environments. This talk will also recap the most important new features and enhancements in ROOT in general, focusing on those allowed by the improved interpreter and better compiler support, including I/O for smart pointers, easier type safe access to the content of TTrees and enhanced multi processor support.

1. Introduction

After almost 20 years of use and development, ROOT [1] continues to be an essential component of HEP experiment's software stack. As the hardware and software environment continually evolves and improves outside of HEP, our software stacks needs to also adapt to take advantage of new opportunities and address challenges posed by the changing tradeoffs and architectures. In particular the rate of change in HEP's main computing language, C++, has recently increased significantly, bringing substantial improvements and simplifications that can help make user and library code much more readable, robust and efficient.

Since 2005, we no longer benefit from the automatic gains made thanks to the increase in processor clock speed. Instead, the clock speed has reached a plateau. The increase in the number of transistors on a chip now translates in an increase in the number of cores rather than an increase in performance of each core. In conjunction with a slower decline in memory price, this means that in order to benefit from the processing power of the latest chips, we need to review our software architecture and increase the amount of work that can be done in parallel within a similar memory budget.

To tackle these two challenges, ROOT has undergone two main transformations, one is the evolution of the code and infrastructure to be able to take advantage of C++11 and later, the second is the adaptation of the core libraries to be run in multiple concurrent threads. Beyond these fundamental changes, the ROOT packages have continued to improve and extend to offer more features and more

Operated by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the United States Department of Energy

robustness. In this paper, we focus on some of the most noteworthy improvements but also detail the overall direction of upcoming enhancements. A much more comprehensive list is available in the release notes of each ROOT version [1].

2. Release 6.02

2.1. Core Library

2.1.1. Cling

The most fundamental change in the release version 6 of ROOT was the replacement of the CINT [2] C/C++ interpreter by a new C++ interpreter named Cling [3][4]. Cling is based on the production grade LLVM [5] open source compiler infrastructure and the Clang compiler [6]. Switching to a full fledged C++ compiler infrastructure presents several advantages and a few disadvantages compared to the existing situation. CINT was developed originally as a C interpreter and was later extended to partially support C++. Since it was developed solely as interpreter, it benefited from the support of only a small team, making the completion of support for the C++98 standard difficult and following the changes in the latest C++ standards almost impossible. However, being originally developed in the context of embedded systems its memory footprint and run-time performance are unmatched.

Switching to Cling allows relying on the large eco-system and support around LLVM and Clang. In particular, Cling already has full support for the C++11 and C++14 standards and will support the C++17 standard soon after its release. Cling adds to Clang the notion of incremental compilation and brings in a few of the extensions that made CINT so useful as a scripting language. Its syntax however is much stricter (and correct) than CINT was and a few adaptations are needed in existing scripts to move from CINT to Cling; for example Cling no longer supports the use of the dot notation after a pointer but will still automatically determine the type of new variable on the command line.

Significant effort was expended to insure that the files written by ROOT version 5 were readable by version 6 and vice-et-versa. One of the main issues was the naming of C++ classes, in particular template instances. Adapting to the more precise class names in version 6, required several adaptations including a revision of the ROOT checksum scheme to no longer include the spelling of typedefs, always replacing them with the underlying type, except for the few typedefs (Double32_t, Float16_t) that have a special semantic and the few (Long64_t, ULong64_t and string) that are kept to improve platform independence.

2.1.2. Infrastructure.

By migrating to Cling, we also open up the ability to support more platforms for a smaller cost. For example ARM64 is already supported in version v6.04/00 and PowerPC64, both little and big endian flavors, will be supported in v6.06/00. Supporting these platforms with the CINT infrastructure had been too expensive to tackle.

Two major changes should be noted in the version 6 infrastructure. Using ROOT version 6 and above strictly requires the use of a C++ compiler configured in a mode enabling support for the C++11 standard or later. CMake is the default build system in version 6 and the legacy configure/make system will only be supported for the next few releases.

2.1.3. Support for multithreading

Thanks to a tight collaboration with the CMS [7] experiment's framework team, the support for multi-threaded use of the ROOT libraries, in particular the Core and I/O libraries, has been greatly enhanced. After all the necessary updates have been put in use, the error rate of CMS concurrent production jobs is negligible. The focus of these improvements has been so far the use case where only one user thread, or stream of operation, or task, is accessing a given TFile object and its TTree. This resulted in the addition of the proper protections and code improvements in the meta-data library in ROOT, including the TClass and TStreamerInfo classes and all the related core functionality. The main set of libraries updated in this round have been the Core (including Meta), I/O, Tree, Hist and Fit libraries.

One the main tools used to make progress on the multi-thread support has been the Helgrind [8] tool provided within Valgrind [9]. Helgrind allows discovering variables and object data members that were written and read by multiple threads potentially concurrently. One of the constructs supported by C++11, atomically accessible variable, is not yet fully recognized by Helgrind and thus issues false positive warnings about the access to these variables. To hide these false positives, and all the others cases discovered during this work, we provide two Valgrind suppressions files: *$ROOTSYS/etc/valgrind-root.supp* and *$ROOTSYS/etc/helgrind-root.supp*. Another tool that was essential to the discovery of potential issues is a tool extending the Clang static code analyzer that can point out all the possible code paths leading to the call of a specific function or the access to a specific variable.

2.2. Math Library

Beyond the interpreter itself, the first practical use of Cling as a just-in-time compiler is a complete reimplementation of the TFormula and TF1 functionalities. The implementation in version 5 relies on a hand coded reverse polish notation parser and executor. The new implementation in version 6.04 first takes the expression in TFormula syntax and writes the equivalent code in standard C++, then it compiles it using the Cling just-in-time compiler and subsequently uses the compiled code. This technique insures much better correctness of results, where the limitation is now only in the trivial translation from TFormula syntax to C++ and in the Clang compiler itself. Adding new functionality to extend the TFormula syntax is now much easier as it only involves upgrading the translator instead of having to update both a parser and an execution engine. In particular, we will be able to update much more easily the code to take advantage of the vector instructions provided by modern CPUs. The version 6.04 implementation of TFormula has the same performance as C++ compiled functions, which is three times faster than the version 5 implementation for typical expressions.

Table 1. TFormula performance comparison.

Expression type	v6.04	v5.34
Predefined functions	60 ns	65 ns
Interpreted expression	65 ns	400 ns
Formula functions	60 ns	200 ns
Compiled functions	50 ns	50 ns

The version v5 implementation of TFormula is still available as ROOT::v5::TFormula mostly for support of I/O backward compatibility.

2.3. I/O library

The main updates, besides the thread safety improvements, to the I/O packages were focused on extending and modifying the metadata to support Cling. One particular improvement is the clarification of the spelling of a class name, nicknamed *normalized name*, which is used throughout to refer to a given class, in particular when it is recorded in a ROOT file meta-data. The normalized name is the name of the class where all the typedefs (except for Double32_t, Float16_t, Long64_t, ULong64_t and string) have been resolved, where template parameters that are set to their default value have been removed for STL collections and added for any other class template instances and where all class names are expressed fully qualified.

In addition, the I/O package's support for schema evolution was enhanced, for example by clarifying the order in which the I/O customization rules are executed. The I/O was also extended to support the containers introduced in the C++11 standard: forward_list, unordered_set, unordered_map, unordered_multiset and unordered_multimap.

The TTreeCache mechanism has been proven to significantly reduce the latency required to read file across wide area network or even local area network and, to a lesser but still noticeable extent, from a local disk. Given its general usefulness it is enabled by default starting from version 6.04/00. The TTreeCache prefill mechanism is also turned on by default. This mechanism reduces the latencies during the training phase of the TTreeCache by reading the data for the first entries for all branches in a single large read rather than many small reads, hence extending the advantage of the TTreeCache to the reading of the whole TTree.

We introduced a new class, *TTreeReader*, which greatly simplifies the reading of data from a TTree by removing the need for any explicit calls to the setup functions (SetBranchAddress, GetEntry, etc.). (1) shows a simple use to extract a value from a series of objects of type MyPart stored in a TTree.

```
TFile *f = TFile::Open("tr.root");
TTreeReader tr("T");
   TTreeReaderValuePtr<MyPart> p(tr, "p");                      (1)
   while (tr.GetNextEntry()) {
      printf("Momentum: %g\n", p->GetP());
   }
 }
```

2.4. Graphics

The set of available visual representations has been extended to include the candle plot (Figure 1.) which graphically describes a data distribution with five values per bin, the violin plot (Figure 2.) which encodes the information about the probability density function at each point, and the ability to use color gradients and transparencies in all plots was added (Figure 3.). A significant extension of the graphic engine was introduced to display mathematical equations and non-Latin characters exactly as TeX does (Figure 4.)

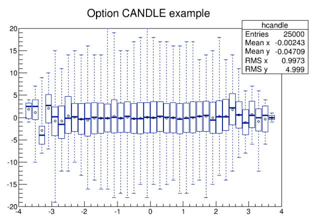


Figure 1. Histogram plotted with the CANDLE option.

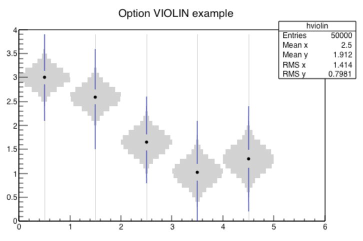


Figure 2. Histogram plotted with the VIOLIN option.

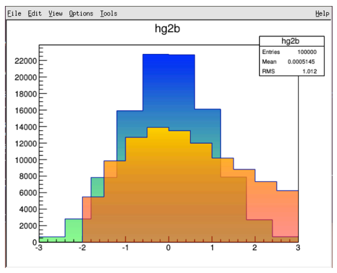
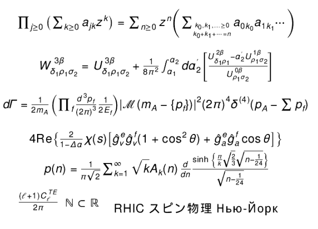


Figure 3. Histogram with linear gradients and semi transparent colors.



$$\prod_{j\geq0}\left(\sum_{k\geq0}a_{jk}z^k\right)=\sum_{n\geq0}z^n\left(\sum_{\substack{k_0,k_1,\ldots\geq0\\k_0+k_1+\cdots=n}}a_{0k_0}a_{1k_1}\cdots\right)$$

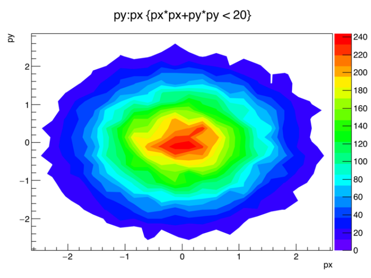
$$W^{3\beta}_{\delta_1\rho_1\sigma_2}=U^{3\beta}_{\delta_1\rho_1\sigma_2}+\frac{1}{8\pi^2}\int_{\alpha_1}^{\alpha_2}d\alpha_2'\left[\frac{U^{2\beta}_{\delta_1\rho_1}-\alpha_2'U^{1\beta}_{\rho_1\sigma_2}}{U^{0\beta}_{\rho_1\sigma_2}}\right]$$

$$d\Gamma=\frac{1}{2m_A}\left(\prod_f\frac{d^3p_f}{(2\pi)^3}\frac{1}{2E_f}\right)|\mathscr{M}(m_A-\{p_f\})|^2(2\pi)^4\delta^{(4)}(p_A-\sum p_f)$$

$$4\mathrm{Re}\left\{\frac{2}{1-\Delta\alpha}\chi(s)\left[\hat{g}_\nu^e\hat{g}_\nu^f(1+\cos^2\theta)+\hat{g}_a^e\hat{g}_a^f\cos\theta\right]\right\}$$

$$p(n)=\frac{1}{\pi\sqrt{2}}\sum_{k=1}^{\infty}\sqrt{k}A_k(n)\frac{d}{dn}\frac{\sinh\left\{\frac{\pi}{k}\sqrt{\frac{2}{3}}\sqrt{n-\frac{1}{24}}\right\}}{\sqrt{n-\frac{1}{24}}}$$

$\frac{(\ell+1)C_\ell^{TE}}{2\pi}$ $\mathbb{N}\subset\mathbb{R}$ RHIC スピン物理 Нью-Йорк



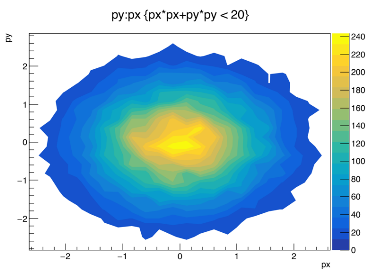


Figure 4. Formulae and non-Latin alphabet drawn with TMahText.

Figure 5. Using the Rainbow palette leads to spurious structure.

Figure 6. Same as Figure 1 with a better palette (57)

A significant issue with graphical representation of scientific data is the ubiquitous use of the rainbow color palette (Figure 5). This palette has the advantage of being visually appealing, however it also easily introduces visual structures that do not exist in the data [10]. For example in Figure 5, the plot seems to indicate three very distinct regions in the data (center, middle, and outer edge) while, as seen when using the palette number 57 in ROOT (Figure 6), the real shape of the data is a very smooth increase from the outer edge to the center of the figure. We recommend with the highest urgency that usage of the rainbow palette be discontinued in all plots (except for those with discrete data where no spurious structure could appear).

2.5. Remote access

The default plugin used to access remote files via the HTTP protocol has been switched to the plugin based on ***Davix*** [11], a project which provides optimized remote I/O, data management and management of large collections of files over the WebDav [12], Amazon S3 [13] and HTTP protocols.

The development of three significant new features is enabling the easy creation of custom live web displays of ROOT based contents. ROOT I/O was extended to support the JSON file format as an output format. The ROOT JavaScript library JSRootIO has been released last year and is under continued development, adding several extensions: drag and drop, context menus, and more. Finally we added a new class, HttpServer that provides remote HTTP access to a running ROOT application and enables the creation of HTML/JavaScript user interfaces showing live data. There are many benefits to this approach: standard HTTP interfaces to ROOT application, no need for temporary ROOT files to access data, user interfaces running in all browsers.

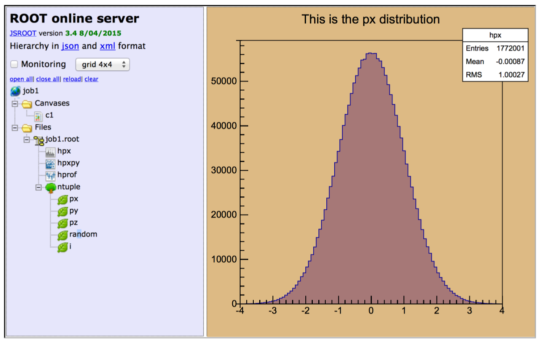


Figure 7. Web display using THttpServer and JSRootIO

3. Version 6 and beyond

With the production release v6.04, ROOT's own libraries can start leveraging the features of both Cling, in particular its ability to produce just-in-time production quality compiled C++ code, and the new features of the C++ language introduced in its newest standards (C++11, C++14 and the upcoming C++17 standards). ROOT was started 20 years ago and can profit from the hindsight provided by the range of applications and users that have benefited and tried its various features over time. These factors can be leveraged to provide the ROOT libraries with:

- Increased overall performance
- Increased user friendliness
- Clarification and leveraging of parallelism
- Standardization

3.1. Increase overall performance

Modern CPUs and coprocessors, for example GPGPUs and the Xeon Phis, relies more on increasing the width of the assembly instructions than on speeding-up individual instructions to increase performance. To take advantage of this trend, the ROOT code will be improved to use code vectorization wherever applicable. This will include reviewing which functions should be inlined and which should stay outlined as well as rearranging some of the data structures' layout to increase the compiler's vectorization opportunities. The primary targets for these improvements are the common math functions and the Histogram, Fit, and RooFit libraries as well as the I/O subsystem.

As was demonstrated with the move of TFormula to using just-in-time compilation, we can significantly improve run-time performance by using Cling and this same technique should be applied to the infrastructure (TTreeFormula) used for plotting data stored in a TTree. Similarly we could use just-in-time compilation to improve the run-time performance of ROOT I/O operations by detecting hot spots at run-time then assembling and compiling the relevant I/O actions into a customized action that can be specialized for that exact use case and be more optimized and more compact.

There are many opportunities in the I/O package to improve runtime, disk space and memory usage, these include: switch the on disk format to use a little-endian representation, compress each entry in a TTree individually to improve random access, and reducing the cost of deep class hierarchies by removing duplicated meta data.

Now that the TTreeCache is enabled by default, it is ready for investigation of improvements on some of its core algorithms including the basket size optimization, and the prefetching methodology.

3.2. Increase user friendliness

Thanks to new features in the C++11, C++14 and C++17 standards, many interfaces in ROOT can be significantly improved. At the time the current ROOT interfaces were designed, the compilers' implementation of class templates was in its infancy and not yet viable. In C++11, the template based type safe containers are as efficient and more widely used as their existing counterparts in ROOT; using these containers in the ROOT interfaces would make the interfaces more widely familiar as well as providing more functionality. Another opportunity would be to use the concept of *smart pointers* to indicates the ownership rules of each function; for example by using std::unique_ptr as shown in (2).

```
void OwnOrNot(std::unique_ptr<TWhatever> arg);                (2)
OwnOrNot( & myWhatever ); // Compilation error!
```

We are also planning on using this kind of technique to remove the need for implicit behavior while still retaining the simplicity of use. For example rather than having some classes, e.g. Histograms, automatically register their objects with a current TFile, if any, we could make this association explicit while not adding much complexity to the code as shown in (3).

```
// Current syntax
TFile f(name); TH1F h1(…);                    f.Write();         (3)
// ROOT version 7 syntax, no implicit shared ownership.
TFile f(name); auto h1 = f.Create<TH1F>(…); f.Write();
```

All these improvements would result in a code that is much more readable, almost self-documenting and at the same time improve user productivity by dramatically reducing the likelihood of memory errors or wrong results in their code.

3.3. Clarify and leverage parallelism

We will build on and extend the thread-safely improvements already made. We are planning on clarifying the concurrent access capabilities of each APIs by properly respecting the idiom that all const functions shall be thread safe and adding explicit documentation of the thread capability of all major subsystems, designing new interfaces whenever necessary for correctness and clarity.

A major effort will be to produce and use a proper scheduling interfaces to allow coordination with user's framework's tasks scheduler so that we can start using multiple threads/tasks in Histogram, TTree, I/O, Math packages while not over-subscribing the system when user code is also attempting to use multiple thread of operations.

Furthermore we should provide the necessary API updates and examples to make it easy to write analysis that can take full advantage of computer providing several or even many cores. This will require the development of an efficient merge step for the typical analysis output, e.g. histograms, TTrees. We will be able to build on the strengths of PROOF and PROOF-Lite as well as well-established industry standards.

Ideally we would make it easy for user to transition their existing analyses to leverage many cores. One of the must have would be to be able to provide hints of the required changes, for example by giving easy access to the static analyzer extension developed by the CMS's framework developers when they prepared the migration to a multi-threaded framework.

3.4. Standardization
We will extent support for and make more extensive use of new C++11 constructs including: std::string, std::string_view, std::array, std::shared_ptr, std::unique_ptr and the new STL collections. We will continue the ROOT tradition to explore, standardize and offer HEP common solutions.

4. Interfaces Revolution – v7
This combination of the availability of many new language features and the challenges in optimally exploiting newer hardware presents a unique opportunity to significantly improve ROOT and its interfaces. Large existing code bases relied upon in production across sciences and continents are tightly coupled with ROOT to take full advantage of its features. Consequently we must balance the need for backward compatibility and reuse of the existing code base with the necessity to evolve the current interfaces.

This can be achieved by a gradual introduction of new backward incompatible interfaces along side the existing one by placing the new interfaces in a new namespace. Whereas the current interface are declared in the global namespace (e.g. ::TFile) the new interface will be introduced as previews into the ROOT namespace (e.g. ::ROOT::TFile) with an alias in a version specific namespace (e.g. ::ROOT::v7::TFile). When ROOT version 7 is released, the old (replaced) interfaces will be deprecated and moved into the ::ROOT::v6 namespace.

5. Collaborations
ROOT has benefited from many collaborative efforts over the years and the tradition is being strengthened for some of the core parts. For instance, contributions for the I/O package have been decentralized in the last few years and plans are underway to expand these contributions even more. The Math package now includes a plugin easing the data and algorithm sharing between ROOT and the R project for statistical computing. New Random generators for concurrent environment are being developed in collaboration with the MinMax project funded by the European Union.

To continue to strengthen and foster this kind of collaborations and contributions, we will refresh the "How To Contribute" page to bring it up to date for the new tools and conventions ROOT is relying on, e.g. git, C++11, etc. and add an explicit list of outstanding projects.

In order to improve the user experience, historically, ROOT has included as part of its source code repository copies of products it depends on. With both the improvement in network connection and the need for more dependent products, this scheme might be improved by replacing the copy by a reference to an external source, for example for the future update of the ROOT geometry package based on the VecGeom library [14]. This will require the development of strong, long lasting procedures to express and resolve the dependencies. One of the challenges involved is the preservation of these connections over time so that older versions of the software are still buildable in the future. The scheme developed for this purpose will allow for better, more flexible integration for products ROOT depends on and could also be reused for products that depend on ROOT itself.

6. Conclusion

ROOT has laid the foundation for a radical modernization of its code base and will be starting to add new clearer and more intuitive APIs that will overtime replace the historical interfaces with the goal of helping to make writing physics analysis code even simpler, more intuitive and more robust. The main driving principles in this re-engineering are simplicity, robustness, performance, embrace of multi-tasking and vectorization, providing even better features and continuation our many collaborations. Some of these improvements will be available as part of a future looking refreshing of the ROOT interfaces leveraging years of experience and the latest tools and techniques.

References

[1] R. Brun and F. Rademakers, 1997, "ROOT – An Object Oriented Data Analysis Framework", *Proceedings AIHENP'96 Workshop, Lausanne, Sep. 1996, Nuclear Instruments and Methods in Physics Research* A **389**, p81-86.
See also http://root.cern.ch/

[2] Goto M 1995 C++ Interpreter - CINT, *CQ publishing, ISBN4-789-3085-3* (Japanese)

[3] Cling citation

[4] Piporo D. 2015 "ROOT6: a quest for performance", Proc. Computing In High Energy And Nuclear Physics, Journal of Physics: Conference Series, **2015**

[5] Lattner C and Adve V 2004 "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation", *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization* (CGO '04). IEEE Computer Society, Washington, DC, USA, p75-.
See also LLVM llvm.org <http://llvm.org>.

[6] Clang. Clang.llvm.org <http://Clang.llvm.org>.

[7] Chatrchyan S and others 2008 "The CMS experiment at the CERN LHC" *Journal of Instrumentation,* **3**, pS08004.

[8] Muehlenfeld A and Wotawa F, 2006 "Fault Detection in Multi-Threaded C++ Server Applications" *Informal Proceedings of the International Workshop on Multithreading in Hardware and Software* **TV06**

[9] Nethercote N and Seward J 2007 "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation" *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*

[10] Borland, D and Taylor, R.M, 2007, "Rainbow Color Map (Still) Considered Harmful" *Computer Graphics and Applications, IEEE,* **27**, 2, p14-17

[11] DAVIX, https://dmc.web.cern.ch/projects/davix/home

[12] E. James Whitehead, Jr. and Yaron Y. Goland. 1999. "WebDAV: a network protocol for remote collaborative authoring on the Web". In *Proceedings of the sixth conference on European Conference on Computer Supported Cooperative Work* **99** p291-310.

[13] Mayur R. Palankar, Adriana Iamnitchi, Matei Ripeanu, and Simson Garfinkel. 2008. "Amazon S3 for science grids: a viable solution?". In Proceedings of the 2008 international workshop on Data-aware distributed computing, **08** p55-64

[14] Apostolakis J *et al* 2014 "Vectorising the detector geometry to optimise particle transport" *J. Phys.: Conf. Ser.* **513** 052038.