Findings and Recommendations re the GEANT4 Field Propagation Module

W. E. Brown and K. Genser
Fermi National Accelerator Laboratory
2009-09-16

Contents

1 Introduction

1.1 Scope of assessment

Our charge was to "Identify points for improvement in coding and design, concentrating on improving performance, robustness and maintainability. [T]he initial concentration should be on lower-level technical issues and move to higher-level design and algorithmic issues time permitting." We were to focus on the contents of directories `geometry/navigation` and `geometry/magneticfield`, and specifically to include classes:

- `G4PropagatorInField`,
- `G4VIntersectionLocator`,
- `G4ChordFinder`,
- `G4ClassicalRK4`, and
- `G4CashKarpRKF45`,

and classes that inherit from these.

We performed measurements of the `cmsRun` application, both with and without caching of the magnetic field information, tracking muons from Z decays and particles from Z' decays, in order to be guided in our review by the time spent in each of the functions called. We were similarly guided by GEANT4 novice example `N02` in order to determine which classes and functions are in heaviest use.

Our findings (via `valgrind`) revealed the following to be the most heavily-used of the Field Propagation functions in the novice example; percentages approximate the time spent in the named function:

- `G4ClassicalRK4::DumbStepper` (9%),
- `G4Mag_UsualEqRhs::EvaluateRhsGivenB` (4%),
- `G4MagErrorStepper::Stepper` (2.3%),
- `G4EquationOfMotion::RightHandSide` (1.8%), and
- `G4UniformMagField::GetFieldValue` (0.9%).

In the `cmsRun` applications, the most heavily-used functions (with circa 10% uncertainty) were:

- `G4Mag_UsualEqRhs::EvaluateRhsGivenB` ($2.2 - 3.5\%$),
- `G4Clas::DumbStepper` ($1.8 - 2.9\%$),
- `G4PropagatorInField::ComputeStep` ($1.2 - 2.0\%$),
- `G4MagInt_Driver::QuickAdvance` ($0.8 - 1.1\%$), and
- `G4ChordFinder::FindNextChord` ($0.7 - 1.1\%$).

Thus, in addition to the specific classes named in our charge, we concentrated on the above-named heavily-used classes, with occasional (brief) excursions into other classes.

This review focuses on technical issues regarding the classes' implementation in C++ code. It may be appropriate, once the issues raised below have been addressed, to task another review team with the charge of assessing the module's class design and any other high-level areas of concern.

1.2 Outline

We present, first, some general observations. Along with each item, we indicate our assessment (1) of the item's relative importance and (2) of the relative difficulty of remedying the issue. In each case, we use codes H = high, M = medium, or L = low at the end of each item. The order of the observations within each section provides a first approximation to the order in which we recommend that they be considered for implementation: H/L, M/L, H/M, L/L, M/M, H/H, M/H, L/M, L/H.

In the subsequent section, we present remarks specific to individual classes, functions, *etc.* The order of these observations and remarks does not necessarily correspond to their order of importance.

We then describe a small experiment we undertook to assess the impact of modifying a field propagation module class' implementation. We conclude the paper with a few acknowledgments.

2 General observations

2.1 Code documentation

Inline documentation. In browsing the code, we noted some internal documentation for maintainers. In some cases, however, the information was split among three files, the `.hh`, the `.icc`, and the `.cc`. [M/L]

2.2 Provisions for testing

We were initially unable to locate a framework (*e.g.*, a test suite) for comprehensively testing the types and functions comprising the Field Propagation module. In the absence of such a test suite, it would have been difficult to verify the correctness of any non-trivial modifications that may be made to the module's code.

However, we subsequently learned that the GEANT4 software repository does contain tests, but that these tests are not routinely made part of the tar distribution. We recommend that consideration be given to distributing the tests universally, and/or to granting universal anonymous read access to the repository. Given access by either means, an end user could verify to his own satisfaction the fundamental correctness of the library's behavior. We also recommend that documentation be provided and distributed regarding these tests' availability and operation. [H/M]

A very brief inspection of the tests suggests that they might be best categorized as *integration tests*. That is, each test appears to involve multiple classes' functionality combined, rather than to probe the behavior of a single class. Tests of the latter kind are often known as *unit tests*. We believe there is value in having tests of both kinds, and recommend the routine inclusion of unit tests to provide a firm basis for the integration tests. Unit tests are also valuable when code improvement (such as refactoring) is undertaken, for they expose issues in isolation, which makes them generally easier to isolate and remedy. We also recommend that such unit tests be self-contained, that is, that each test report success or failure without resorting to external data. [H/H]

2.3 Single responsibility per class.

We noted that some classes (*e.g.*, `G4MagInt_Driver`) have certain calculations as their primary responsibilities, yet also have report-printing responsibilities. It is a principle of program design that each class and each function have exactly one responsibility. For classes that gather statistics, for example, we recommend that they be designed in conjunction with statistics types that keep the statistics and can make them available to report-printing functions on request. [H/H]

2.4 Class design

Alan Lenton points out[1] that "One of the things that you are told when you first start learning about classes in C++ is to put the data in the private/protected part of the class and write functions to access and set the data. This is all well and good as far as it goes.

[1] http://www.ibgames.net/alan/technical/classdesign.html.

"Unfortunately, it often goes too far, and you get horrendous classes with (say) 10 items of data and then 20 functions of the form `ReadXX()`/`SetXX()` to read and set those data items. Really, you might as well have made those data items public for all the good the data 'hiding' did.

"In general the data should be set during the construction of the object. It should really only be changed by the object itself as a result of a high level operation."

We observed that the majority of the classes we reviewed followed the "too far" approach described above, as they are replete with a proliferation of getter and setter functions, and have minimal additional functionality. We recommend discontinuing such design practices in favor of adhering to principles of known good object-oriented design. [H/H]

2.5 Level of abstraction

The use of arrays rather than higher-level abstractions is embedded throughout the Field Propagation module. An array is a rather low-level data structure with relatively few operations and even less semantic content. We recommend instead always using the highest-level abstraction consistent with the intent, *i.e.*, one whose `public` interface corresponds to its clients' semantic needs.

For example, `G4Field::GetFieldValue()`'s `Point` parameter, currently an array of extent 4, could have been declared and used instead as a `G4LorentzVector`. Such a choice would:

- provide more type safety than is now possible,
- convey additional semantic information than currently provided, and
- reduce the current litter of dealing with arrays, leading to code simplification.

If a purely linear organization (*e.g.*, an array or a `std::vector`) absolutely must be used to hold diverse quantities, we recommend that names be used in place of numerical subscripts, as the following code snippet illustrates:

```
1  enum {X=0, Y=1, Z=2, PX=3, PY=4, PZ=5, T=6, E=7};
2  G4double y[12];
3  ... y[T] ...  // instead of y[6]
```

[H/H]

Naming inconsistencies. We observed inconsistent naming patterns. For example, we found among the `magneticfield` names the following forms: `Magnetic`, `Mag`, `Mag_`, and `M`. [M/H]

Invariants. It is another recognized best practice to provide each class with an invariant (known in this context as a *class invariant*). The purpose of invariants in general is to assist in reasoning about the code, especially in assuring the code's correctness. Unfortunately, we found only few such invariants. [M/H]

2.6 Coding practices

Initialization issues. We discovered an unfortunate pattern in a number of the constructors (and one of the assignment operators) that we examined: some fraction of the data members are left uninitialized or uncopied. This is contrary to recommended practice, as it makes it unnecessarily difficult to reason about code that uses such constructors. It also introduces unnecessary fragility and coupling in the code, as one must be careful to avoid calling a function that uses an uninitialized data member before calling a function that initializes them.

Incomplete (or occasionally even nonexistent) initializer lists represent a significant contributing factor to this problem, and may even lead to reduced performance due to doubly-initialized data

members. We recommend that all constructors always initialize all member data via the initializer list, and that the initializer list be ordered so as to match the order in which the data members are declared. We further recommend that the body of a constructor be reserved for additional processing that can't be accommodated via an initializer list. Finally, in a number of cases (especially where there are no held resources, *e.g.*, in abstract base classes), we recommend that copy constructors (and similar special member functions, where appropriate) be neither declared nor defined and thus left to the compiler to generate. [H/L]

Implementation of copy assignment. We noted in a few classes incomplete or incorrect (*e.g.*, infinitely recursive) copy assignment operator implementations. These functions largely parallel their respective classes' copy constructors, generally augmented by a special test to avoid self-assignment (which rarely, if ever, is used in practice). Modern C++ recommends, for any class `T`, the following approach instead: First, ensure that `T` has (a) a correct and faithful copy constructor, and (b) a non-throwing destructor that correctly disposes of any resources held by the class. Second, provide `T` with a non-throwing `swap` function to exchange the values of two variables of type `T`.[2] Finally, write `T`'s copy assignment operator according to the following model which, by construction, is correct as well as exception-safe in all cases (including the rarely-occurring self-assignment).

```
1 T &  operator = ( T const & other )  {
2   T  tmp( other );
3   swap( tmp );  // assumes swap is a (non-throwing) member function
4   return *this;
5 }
```

However, in a number of cases (especially where there are no held resources, *e.g.*, in abstract base classes), we recommend these functions be neither declared nor defined and thus left to the compiler to generate. [H/L]

Early declarations. In some programming languages, variables must be defined at or near the beginning of a function. In C++, it is possible to define variables at any point within a function, and it is recognized best practice to avoid defining a variable until its use is required. In the Field Propagation module, we have encountered quite a number of cases in which variables are defined early in a function, before they are needed. This practice is in general unnecessarily expensive especially when the variables' types have default constructors which are thus unnecessarily invoked. As an additional cost, the initial values of such variables must be adjusted before they are actually used. By delaying such variables' definitions, they could be correctly initialized and ready for immediate use. [H/L]

Policy re `inline`. We observed that classes employ inconsistent approaches in determining which member functions ought be implemented `inline`, as well as in declaring and placing such functions. For example, some classes declare each such function as `inline` in the class definition and again in each function's implementation, while others only do the latter. Still others don't use the `inline` keyword at all when defining the function within the class body. We recommend a coherent policy be formulated (preferably in accordance with the DRY [Don't Repeat Yourself] principle) so that all classes can have a consistent approach to this important decision and to its realization.

[2] It is always possible to implement such a `swap` function for any class `T` by invoking an appropriate `swap` function for each of `T`'s data members: For each data member of `T` whose type is a native (built-in) type, invoke `std::swap`, and for each data member of `T` whose type is either a library type (*e.g.*, `std::vector`) or a user-defined type, invoke its own `swap` member. Stateless members, if any, do not need to be swapped since all their instances must be equivalent.

As part of such a policy, we recommend that strong consideration be given to the possibility of eliminating all `.icc` files. Such files' presence adds to the complexity of maintaining code as each occurrence forces an additional file to be opened for inspection. Further, each occurrence adds to compilation effort for the same reason.

If these recommendations are adopted, we further recommend that each `inline` function now defined in an `.icc` file be defined instead where it is declared in the `*.hh` file (particularly as such functions tend to have very short definitions). Such a practice would avoid any need to say `inline` anywhere. [M/L]

Increment operator selection. Best practices regard the unnecessary use of postincrement in place of preincrement operations as a code "pessimization." In loop control and also occasionally elsewhere in the code we saw consistent use of the postincrement operator, and saw only very few uses of the preincrement operator. [M/L]

Noncopyability. If there are no reasons to copy instances of a class that has a bare pointer as a data member in its state, the class should be non-copyable (*i.e.*, that its copy functions be declared private and remain unimplemented) in order to minimize the risk of pointer proliferation and the attendant problems of coordinating the lifetimes of the pointers and of the (sole, shared) pointee. Indeed, many classes are already noncopyable for this and similar reasons; we recommend stricter application of this policy. [H/M]

Loop predicates. Modern C++ programming techniques recommend the use of loop predicates involving the `!=` operator rather than, say, the `<` operator. There are several reasons for this preference: For one, such use permits a stronger post-loop assertion; for another, loops controlled by iterators must already use `!=` in their predicates since iterators are generally not required to support `<`. The bulk of the predicates we encountered in the Field Propagation module use the `<` operator. [L/L]

Code appearance. The indentation used by the text of the Field Propagation module is inconsistent at times. The code is also occasionally difficult to read, *e.g.*, devoid of conventional spacing between tokens. Many lines end with unnecessary whitespace (which we can see them because our text editor highlights them); this extra whitespace contributes to small amounts of overhead in many places:

- compilation time during each compilation,
- space in the code repository, and
- time and space during each repository check-out and check-in.

We recommend a script be run automatically during each repository check-in and check-out to strip such whitespace from each file. [L/L]

Non-`explicit` conversion constructors. We observed no use of the `explicit` keyword. It is generally recommended that this keyword be applied to all conversion constructors (constructors callable with a single argument) to prevent the compiler from performing the conversion unless explicitly requested to do so. The absence of this keyword can lead to unexpected and unintended conversions, making it more difficult to understand and reason about code. [M/M]

Pointers in containers. We found in the Field Propagation module an occasional container of pointers to objects, rather than a container of the objects themselves. There are a few good reasons to use containers of pointers; for example (1) when the pointees are to be shared and thus avoid the overhead of multiple copies of them, or (2) when the exact type of the pointees is of

an unknown derived type and the `virtual` mechanism is to be used by pointing to instances of the base type. However, neither of these reasons seemed to apply here, and in their absence the chosen approach is a source of potentially significant performance degradation, as each traversal of the container introduces an extra dereferencing operation. Further, each time such a container were to be created, copied, or destroyed, it incurs the overhead of expensive dynamic memory management, together with its attendant added complexity. [M/M]

Pointers' frequency. The Field Propagation module was observed to make significant use of native C++ pointers. Such pointers are widely considered to be a very low-level data structure that, while effective, are sources of complexity in design and of bugs in implementations. Modern C++ recommends instead the use of smart pointers and similar handle types. The Standard Library, for example, today provides `std::auto_ptr` and will in the near future provide additional smart pointers (*e.g.*, `shared_ptr` and `unique_ptr`) that have for circa ten years been available from other sources. In the absence of such a pointer discipline, it would for example not be feasible to explore performance improvements based on multi-threading. [H/H]

Choice of loop. We observed that several algorithms make use of `do ... while` constructs, known as *post-test* loops because their predicates are evaluated after, rather than before, each iteration. Such constructs are rarely found in modern C++ programs, and are often artifacts of code originally written in Fortran (whose loops do follow such a pattern) and later translated to C or to C++. The use of *pre-test* loops, as realized by `while` and `for` constructs, is overwhelmingly prevalent in modern code, and should be given strong consideration. [H/H]

Under-utilization of the Standard Library. We observed significant use of the mathematical functionality of the C++ Standard Library, but very little use of the other abstractions (containers, algorithms, *etc.*) the Library offers. For example, we saw explicit (hand-coded) loops to initialize or to copy arrays that could more simply have been coded as simple calls to such Standard Library algorithms as `std::copy`, `std::uninitialized_fill`, *etc.* Using even such simple algorithms as `std::min` and `std::max` improves the clarity of the program text, and often provides performance benefits as well. Additionally, we observed explicit looping over arrays where the entire loop could be avoided via the use of such `std::vector` operators as copy assignment. [H/H]

Function size. The Field Propagation module exhibits a few examples of functions whose implementation involves several hundred lines of code. Good code hygiene considers such function sizes to be well above the recommended upper limit of roughly one page of code. It is more difficult to reason effectively when functions are large, and it becomes riskier to make updates. [M/H]

Code structure. We noted code that seems suboptimally structured or unnecessarily deeply nested, leading to additional compilation expenses as well as to impediments to understanding. (We point out selected examples in the next section, together with recommended adjustments.) [M/H]

Literals. It has long been recommended practice, in C++ as well as in other programming languages, to provide meaningful names for constants used within the code. This enables readers to understand the purpose of a constant, with no need to infer its intent from its context. The code we inspected does not always follow this practice. [M/H]

Provisions for debugging. The presence of some debugging statements (a few of which seem to introduce side effects!) makes the logic of the underlying code significantly more difficult to read due to the extra bulk. [L/M]

3 Code-specific observations

The following remarks encompass only classes and functions that we encountered during our review. It is not intended as a comprehensive list for the entirety of the Field Propagation module, nor are these remarks necessarily comprehensive even within the specific portions we call out below.

3.1 The `G4Field` hierarchy

This section reports on the hierarchy of classes rooted at `G4Field`. We note this is a four-level hierarchy, and wonder whether such depth is warranted. For example, `G4Field` and `G4ElectroMagneticField` are both abstract classes, the latter directly inheriting from the former, and (outside its own implementation) each is only used in two additional classes. Since each of the fields is an electromagnetic field, we wonder whether these could not be profitably consolidated (by eliminating `G4Field` and retargeting its few uses to `G4ElectroMagneticField`).

`G4Field`

- The copy assignment operator is disastrously wrong; it embeds an infinitely recursive call.
- It is distinctly unusual to have a copy assignment operator but no copy constructor.
- In this class, there is no need to declare or define either the default constructor or the copy assignment operator as there are no data members to be initialized or copied, so the compiler-generated versions of such functions would be trivially fine; we therefore recommend that these functions' declarations and definitions be removed.
- The signature for `GetFieldValue` uses `double` and not `G4Double` as is done elsewhere (*e.g.*, in its derived classes). (But see related comments elsewhere herein.)

`G4ElectroMagneticField`

- The (virtual) destructor is trivial and so should be defined `inline` (as was done in the `G4Field` base class).[3]
- This class has the same issues re default constructor and copy assignment operator pointed out re the `G4Field` base class.
- The copy assignment operator is wrong because, as a derived class, it ought (but fails to) invoke the copy assignment operator of its base class.
- The copy constructor is correct, but the compiler-generated one would have been equally correct and likely at least slightly more efficient.

For the above reasons, we recommend:

[3] We recognize that, historically, there have been prevalent beliefs in the C++ programming community to the contrary, among them:
- No member function can be declared both `inline` and `virtual`.
- No destructor can be declared both `inline` and `virtual`.
- It is necessary to have a non-`inline` destructor if one is building a shared library.

However, none of these is true and, to the best of our knowledge, modern C++ compilers should have no trouble coping with the recommended practice, especially when the destructor in question is trivial and there are no data members that themselves have expensive destructors.

As the Google C++ Style Guide [http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml] puts it, "A decent rule of thumb is to not inline a function if it is more than 10 lines long. Beware of destructors, which are often longer than they appear because of implicit member- and base-destructor calls!"

- omitting the declarations and definitions of the default constructor, copy constructor, and copy assignment operator;
- defining the virtual destructor `inline`; and
- deleting the entire `G4ElectroMagneticField.cc` file.

G4MagneticField

This class has the same issues as described in the analysis of its base class, `G4ElectroMagneticField`, and the same recommendations apply.

G4UniformMagField

- It is unclear why the `fFieldComponents` data member has array type, as it seems that `G4ThreeVector` would be a more suitable choice that would lead to considerable code simplification and some small performance benefits, *e.g.*,:
  - the compiler-generated versions of the copy constructor and the copy assignment operator would suffice;
  - `GetConstantFieldValue()` becomes an inlineable one-line function; and
  - there would be no need for 3-iteration loops over assignment (or, equivalently for any sequence of three assignments)[4].
- The destructor should be `inline`d as was recommended for the base class.

G4QuadrupoleMagField

- The default constructor appears to double-initialize its data member (once via the implicit initializer list, a second time via assignment); this is a small space and time inefficiency; we recommend using only an explicit initializer list wherever possible.
- The first parameter to `GetFieldValue()` is declared to have type `G4double[7]`. However, the body only uses the leading two items of the array, and the base class function being overridden declares the comparable parameter as having type `G4double[4]`. This seems inconsistent; a named type in lieu of an array might have been preferable.

3.2 The G4EquationOfMotion hierarchy

This section reports on the hierarchy of classes rooted at `G4EquationOfMotion`. We note in passing that the classes in this hierarchy do not follow as consistent a naming scheme as do other hierarchies.

G4EquationOfMotion

- The comments in the `.hh` file are seriously out of date and wrong (*e.g.*, "This is the _only_-function a subclass must define.", followed immediately by a declaration for another pure virtual function).
- The documentation (here and elsewhere where appropriate) should state that this type does not take ownership of any pointer passed to it.
- If this class (and its subclasses) are intended not to make any changes to the field, perhaps the `itsField` member should have type `const G4Field*`.
- `EvaluateRhsReturnB()` has functionality identical to that of `RightHandSide()`; they should perhaps share implementation (*e.g.*, by having the latter call the former).

[4] This recommendation is general, and could be repeated in many places.

G4EqMagElectricField

- The class name is somewhat misleading; one would expect this class to inherit (directly or indirectly) from `G4Field`.
- The header need not `#include "G4ElectroMagneticField.hh"` as there are no uses of that type (only of a pointer to that type, so a forward declaration would suffice).

G4Mag_EqRhs

- The constructor incompletely initializes the object being created, as it fails to define the initial value of the member `fCof_val`. As described in general earlier, such a practice is dangerous, because a user can (in this case) call `FCof()` and so obtain an uninitialized and hence arbitrary and unpredictable value.
- At least some of the comments in the `.hh` file are misleading or even wrong. Further, comments ought never include such literals as "0.299792458" or "two".
- We recommend all functions in the `.cc` file be `inline`d.
- The `SetChargeMomentumMass` function scales and then memorizes the first of its parameters, but ignores its other two parameters. The class then ignores that memorized value except when explicitly reqested (via `FCof()`) to deliver it to a caller. All this seems an unusual design.

G4Mag_UsualEqRhs

- The data member `fInvCurrentMomentumXc` is uninitialized by the constructor, is set by `SetChargeMomentumMass`, is ignored by the rest of the class, and has no possible way of being used by any other entity. We recommend that this data member go away, along with its sole use.
- `EvaluateRhsGivenB()` could employ the `sqr()` algorithm defined in `templates.hh` in lieu of explicit self-multiplication.

G4ErrorMag_UsualEqRhs

- All functions could easily be declared `inline`.
- The origin of function `sqr()` (called by `EvaluateRhsGivenB()`) is non-obvious from inspecting the source code. After some effort, we located its origin in header `templates.hh`, but this header is included only (very!) indirectly. We recommend review and possible rescission of the policy that relies on transitivity of `#include`s. Modern C++ compilers have become very efficient at recognizing headers that are `#include`d more than once, and the effort expended by maintainers to locate a header now easily outweighs the effort expended by compilers in avoiding duplications among `#include`d headers.

3.3 The `G4MagIntegratorStepper` hierarchy

This section reports on the hierarchy of classes rooted at `G4MagIntegratorStepper`.

G4MagIntegratorStepper

- This class' interface uses low-level arrays instead of higher-level abstractions; there seems little inherent reason to avoid specifying the intent of the parameters instead of specifying their implementation.
- It is surprising that `SetEquationOfMotion()` is careful to check its parameter before saving it, but that the constructor is not at all careful about this; because of this, the user of `GetEquationOfMotion()` must be careful always to check the result that is provided, as it may be null.

- `GetEquationOfMotion()` is an accessor function, and so should likely be declared `const`.
- `G4MagIntegratorStepper()` could make use of the `sqr()` template defined in `templates.hh`.
- `G4MagIntegratorStepper()` declares its intermediate variables as type `double` instead of type `G4double`, although it parameter is `G4double`.
- `NormaliseTangentVector()`'s loop seems unnecessary; a call to `std::transform()` might be a candidate for its replacement, or (perhaps even better) just a sequence of three assignments. If the explicit loop must for some reason remain, its control should be written as: `for( int i=0; i!= 3; ++i)`.
- `G4MagIntegratorStepper()` has a calculation that depends on a value of `1e-14`; the intent of this value is nowhere documented, nor is the rationale for its selection.
- The class constructor and destructor could be trivially `inline`d.
- It is not clear why this class (and therefore the entire hierarchy rooted here) was designed to be non-copyable.

G4CashKarpRKF45

- All pointers to dynamically-allocated arrays should be replaced by `std::vector<>`.
- `Stepper()` unnecessarily recalculates needed constants each time it is called; these constants should be declared with `static` lifetime.
- `Stepper()` declares a single variable, `i`, to control all the counted loops; there is no savings of time or space in doing so as opposed to declaring a fresh variable as part of each loop; we recommend this single `i` be eliminated and each loop control be rewritten as: `for( int i = 0; i != numberOfVariables; ++i )`. (The same or similar observation applies elsewhere, as well.)
- The five local variables in `DistChord()` seem poorly conceived and used:
  - Variables `distLine` and `distChord` are entirely unnecessary; they can be entirely removed and the 11 lines of code in which they are participate ought be replaced by the following single statement (spanning 3 lines for extra clarity):

```
1 return initialPoint == finalPoint
2      ? (midPoint-initialPoint).mag()
3      : G4LineSection::Distline(midPoint, initialPoint, finalPoint);
```

  - Each of the remaining 3 variables should be defined no sooner than they can be properly initialized (in order to avoid the overhead of an unnecessary default initialization followed by an assignment), *e.g.*,:

```
1 G4ThreeVector midPoint(fMidVector[0], fMidVector[1], fMidVector[2]);
```

G4MagErrorStepper

- Some but not all of the `virtual` functions are declared `virtual`; this inconsistency is confusing. We recommend all `virtual` functions be thusly declared (even if `virtual` is implied via a base class).
- All pointers to dynamically-allocated arrays should be replaced by `std::vector<>`.
- `Stepper()` and `DistChord()` have a hidden dependency via their use of the non-local variables `fInitialPoint`, *etc.* Such dependencies have been widely recognized as constituting poor programming practice, and so we recommend that the two functions be consolidated into one (here and everywhere else in this hierarchy, too, for consistency). Remarks from `G4CashKarpRKF45` above are also applicable here.
- We recommend that all literals be named, including the use of 7 in this class.

G4ClassicalRK4

- This class exhibits many of the same issues already pointed out in the review of its base class `G4MagErrorStepper`: use of literals, single variable controlling many loops, use of dynamically-allocated arrays instead of `std::vector<>`s, *etc.*

G4SimpleHeum

As a sibling to class `G4ClassicalRK4`, this class shares most of the same comments.

G4MagHelicalStepper

- Because the interfaces are array-based, it is rarely clear as to what the arrays' extents are, and what each represents (this comment applies almost everywhere, at least so far), and so code assumes certain sizes that the code can't verify.
- The constructor leaves most data unitialized (allowing possible consequences as pointed out elsewhere), and initializes the pointer data member via an assignment rather than via the initializer list.
- `DistChord()`'s body is mostly a nested `if` whose structure realizes a 1-of-3 selection. We recommend it be rephrased in the following conventional style:

```
if(Ang<=pi)
  return GetRadHelix() * (1.0 - std::cos(0.5*Ang));
else if(Ang<twopi)
  return GetRadHelix() * (1.0 + std::cos(0.5*(twopi-Ang)));
else  // return Diameter of projected circle
  return 2.0 * GetRadHelix();
```

- Some of the indentation in the `.icc` file is inconsistent.
- Integer literals are mixed with doubles, even within the same line:

```
CosT     = 1 - 0.5 * Theta2 + 1.0/24.0 * Theta4;
```

- `AdvanceHelix()` contains a `FIXME` comment; this function may need attention.
- `AdvanceHelix()` does not always use the smallest number of operations:

```
Bnorm = (1.0/Bmag)*Bfld;
```

  which could be recoded:

```
Bnorm = Bfld / Bmag;
```

  and similarly in other functions.

G4ExactHelixStepper

- The constructor does a better job than most in this hierarchy by way of initializing data members, but a few are still left uninitialized.
- The constructor's loop might be replaced by calling `std::fill_n()`.
- A design issue: if the base class tracks `EqRhs` (done in construction), this class may not need its own copy.
- The const value `nvar` is defined in several places; if a property of the class, we recommend that it become a class-level value.
- The `DistChord()` calculation is essentially identical to the same-named function in the `G4MagHelicalStepper` base class. There may be no need to have both.
- Some of the indentation in the `.cc` file is inconsistent.

G4HelixHeum

- `DumbStepper` defines `nvar`, but then fails to use it two lines later in defining arrays of that size.
- It is suspicious that the introductory comment states this is a third order solver, yet `IntegratorOrder()` returns 2 instead of 3 (another reason for avoiding literals [even in text form] inside comments).

G4RKG3_Stepper

- We are concerned about the implementation status of this class; while it is located among the production code, several comments suggest strongly that this class is only incompletely implemented and tested.
- This class has many of the same issues already pointed out in the review of its base classes: use of literals, single variable controlling many loops, mixing integer and double types, use of dynamically-allocated arrays instead of `std::vector<>`s, *etc.*

3.4 The G4ChordFinder hierarchy

G4ChordFinder

- We recommend all literals be named, including `1.0e-2*mm`, the default for the constructor's `stepMinimum` parameter, and the many other literals in this class and elsewhere.
- The data member `fAllocatedStepper` seems unnecessary. We recommend that it be eliminated together with all its uses, which will moot its redundant initialization in one of the constructors. (Note that it is safe in C++ to delete a pointer even if its value is 0.)
- `FindNextChord()` at its core has a loop with the following essential structure:

```
done = false;
n = 0;
do {
  code block A
  done = predicate;
  if( !done )
    code block B
  n++;
} while( !done );
```

We recommended this be reformulated to avoid one of the two tests and slightly reduce the number of variables needed, *e.g.*,:

```
code block A
int n = 1;
for( ;  !(predicate); ++n )
  code block B
  code block A
}
```

and perhaps even rephrasing the predicate so that it needn't be inverted during the (now sole) test.

- `NewStep()` has code such as:

```
else if (stepTrial > 1000.0 * stepTrialOld)
{
   stepTrial= 1000.0 * stepTrialOld;
}
```

which is more clearly expressed (also eliminating a duplicate calculation):

```
else
  stepTrial = std::min( stepTrial, 1000.0 * stepTrialOld );
```

- `AdvanceChordLimited()` has another example of a misplaced (or even unneeded) variable, `good_advance`, now employed as follows (comments elided):

```
G4bool good_advance;
if ( dyErr < epsStep * stepPossible )
{
  yCurrent = yEnd;
  good_advance = true;
}
else
{
  good_advance = fIntgrDriver->AccurateAdvance(yCurrent, stepPossible,
                                               epsStep, nextStep);
  if ( ! good_advance )
  {
    stepPossible= yCurrent.GetCurveLength()-startCurveLen;
  }
}
```

(See also `ApproxCurvePointV()` for a remarkably similar example, and see below for suggested improvement.)

- `InvParabolic()` is declared to take all its parameters by `const` value. This `const`ness is an implementation detail that does not belong in the declaration. Further, it only rarely is significant and likely should not be part of the definition either.

G4ChordFinderSaf

- The constructors and destructor seem worth `inline`ing.
- The destructor's internal comment is incorrect.
- See base class comment re `FindNextChord()` which seems applicable here, too, and which we recommend be simplified as shown:

```
if( dyErr < epsStep * stepPossible )
  yCurrent = yEnd;
else if( ! fIntgrDriver->AccurateAdvance( yCurrent, stepPossible,
                                          epsStep, nextStep ) )
  stepPossible= yCurrent.GetCurveLength()-startCurveLen;
```

3.5 G4MagInt_Driver

- The source file names (`G4MagIntegratorDriver.*`) do not match the name of this class. This seems unusual within the GEANT4 file system (at least so far) and makes it harder to find the correct files.
- Several functions are declared `protected`, yet this class is not designed to be used as a base class.
- The constructor calls (directly and indirectly) `public` member functions. This is in general a dangerous and potentially fragile practice because a class' invariant is generally considered established only after the constructor finishes, and all `public` member functions are entitled to assume that it holds when they are called. The constructor also violates the practice of initializing member data in the order in which the member data is declared in the class.
- The conditionally-defined macro `G4DEBUG_FIELD` seems to be positioned too late within the file; there is an `#ifdef` for it earlier.
- The indentation is not always consistent, *e.g.*, in the `.icc` file.
- The use of "no" or "No" as an abbreviation for "number" is potentially very confusing, as in the identifier `noWarningsIssued`. We recommend use of "n" instead of "no" (*e.g.*, `nWarningsIssued`), with "num" as a 2nd choice (*e.g.*, `numWarningsIssued`). See also `no_warnings`, a variable that is directly tested by conversion to `bool` and so gives a result opposite to that naively expected.
- `AccurateAdvance()` includes code with the following structure:

```
if( hstep <= 0.0 )
{
  if(hstep==0.0)
  {
    ...
    return succeeded;
  }
  else
  {
    ...
    return false;
  }
}
```

We recommend instead the following structure, which is no more expensive to execute and more clearly reflects the tripartite nature of the decisions:

```
if( hstep > 0.0 )
  ;  // nothing to do
else if( hstep == 0.0 )
{
  ...
  return succeeded;
}
else  // ( hstep < 0.0 )
{
  ...
  return false;
}
```

- `AccurateAdvance()` includes the code fragment:

```
if( (hinitial > 0.0)
    && (hinitial < hstep)
    && (hinitial > perMillion * hstep) ){ ...
```

Since we know from earlier code that `hstep` is positive, this can be simplified to:

```
if(  (hinitial < hstep)
  && (hinitial > perMillion * hstep) ){ ...
```

and which more clearly reflects the intended range-checking when expressed:

```
if (hstep * perMillion < hinitial
                       && hinitial < hstep)  { ...
```

- A minor performance gain in `AccurateAdvance()` may be achieved by checking `hstep` (see above) at the beginning of the function, before doing anything with local variables, as they won't be used if the `hstep` check fails. It's also a good idea to validate a function's inputs before the function begins any serious work.
- `AccurateAdvance()` uses a `do...while()` loop as a major part of its body. This is an uncommonly-used construct that is normally reserved for use when a looping algorithm truly requires at least one iteration. Because this algorithm seems not to impose such a requirement, we recommend the loop (and allied code) be reformulated as an ordinary `while()...` loop.
- `AccurateAdvance()` directly uses data member `fMinimumStep` in some places, but calls `Hmin()` (which returns `fMinimumStep`) in others. The reason for this inconsistency is unclear.

- `QuickAdvance()` has a local `static` variable `no_call` that is initialized and incremented, but nowhere used. We recommend this variable's removal, as this may improve performance (unless the compiler has already noticed that this variable is not used elsewhere and therefore has already elided it out of existence).
- Adherence to the style of *Numerical Recipes* is not recommended, as these algorithms originated in Fortran, and all their recodings in C and C++ reflect that history rather than taking best advantage of the strengths of each language.
- The `PrintStatus()` overloads are only invoked when `G4DEBUG_FIELD` is defined, so they ought be declared and defined only under the same circumstance.
- Coupling the destruction of such an object to report-printing seems a suboptimal design choice. Please see our general comments on printing statistics reports.
- If arrays must be copied, a call to `std::copy_n` is a better choice than an explicit loop for the purpose.

3.6 G4FieldTrack

- The name of this class suggests a relationship with such other similarly-named classes as `G4Track`. However, the class definition does not manifest any such relationship. This seeming contradiction suggests that a better name for the class, one that better identifies the underlying abstraction, may be in order.
- It is unclear why no constructor takes an object such as a `G4Particle` or a `G4Track` as a source of information being aggregated by this class.
- The `SixVector` could easily (here and elsewhere) have its own non-array type (*e.g.*, named `PhaseSpacePoint`) with its own operations that would greatly simplify its clients' code; we recommend doing so, along the lines of `G4ThreeVector`, *etc.*
- The comment claiming that the class is only used in connection with RK algorithm seems questionable; the class is used in many places throughout GEANT4.
- As evidenced by an introductory remark, this class seems deliberately designed to lack coherence in that it neither requires nor preserves any relationships among its collection of data. Thus, the class has no invariant that any user can rely on. Such a lack is inconsistent with the usual principles of object-oriented design.
- The class seems inconsistent in its implementation of `inline` functions: a few are defined inside the class, but the rest are in a separate `.icc` file.
- There is a private nested type, but a public accessor for its instance, yielding a pointer to an inaccessible type. This design seems inconsistent.
- There is a most unusual constructor, documented as "Almost default constructor", that has a `char` parameter that it ignores. We recommend this constructor become a true default constructor (taking no parameters) instead.
- One constructor concludes with the following remarkable code sequence:

```
1 G4ThreeVector Spin(0.0, 0.0, 0.0);
2 if( !pSpin ) Spin= G4ThreeVector(0.,0.,0.);
3 else         Spin= *pSpin;
4 InitialiseSpin( Spin );
```

  which can be more simply, efficiently, and directly written:

```
1 InitialiseSpin( pSpin ? *pSpin : G4ThreeVector(0.,0.,0.) );
```

- `SetChargeAndMoments()` (and elsewhere) uses the C++ macro `DBL_MAX`, but this macro is tied to the type `double`, not to `G4double`; if `G4double` were ever to differ from `double`, this code is likely to break.
- There is no need for an explicit copy constructor in the nested class; the compiler would generate one with identical semantics and (likely) better performance.
- It is unclear why we have both `SetSpin()` and `InitialiseSpin()` with identical functionality.

- The class lacks any means of setting itself to an all-zero state (which functionality other classes mimic by explicitly constructing such an entity and copying it); a default constructor might be one approach, as might a `clear()` member, as might both.
- The constructors are not all declared together. This makes it more difficult to know (for example) which one is being called from elsewhere.

3.7 The `G4VIntersectionLocator` hierarchy

`G4VIntersectionLocator`

- The constructor takes no advantage of the initializer list, and (as elsewhere) fails to initialize all data members (only 5 of the 9 are given values).
- Unlike many other GEANT4 classes that have pointers as data members, this class is copyable. However (again unlike most other classes), this class owns a resource (namely a `G4Navigaror` that is acquired in the constructor and discarded via the destructor). If copied, both instances will then own the same `G4Navigator` and therefore that `G4Navigator` will be erroneously discarded by both the source and the target `G4VIntersectionLocator`. We therefore recommend, as a minimal change, that this class become noncopyable.
- `ReEstimateEndpoint()` has a block of code that is enabled (via `#else`) only when debugging is disabled. This seems odd.
- Many remarks made earlier in other contexts also apply in this class.

`G4MultiLevelLocator`

- The class is copyable, but the compiler-generated copy functions will not do the right thing with respect to the array data member.
- `EstimateIntersectionPoint()` is quite a large function (> 600 lines long).
- In `EstimateIntersectionPoint()` the `fin_section_depth` array should be replaced by a `std::vector<bool>` in order to simplify its initialization; further the container should hold objects, not pointers to objects, in order to improve performance; this would be facilitated when `G4FieldTrack` is given a default constructor as recommended elsewhere.
- In `EstimateIntersectionPoint()` we have (comments elided):

```
1  if(depth==0)
2  {
3    CurrentA_PointVelocity = CurrentB_PointVelocity;
4    CurrentB_PointVelocity = CurveEndPointVelocity;
5    SubStart_PointVelocity = CurrentA_PointVelocity;
6    restoredFullEndpoint = true;
7  }
8  else
9  {
10   CurrentA_PointVelocity = CurrentB_PointVelocity;
11   CurrentB_PointVelocity =  *ptrInterMedFT[depth];
12   SubStart_PointVelocity = CurrentA_PointVelocity;
13   restoredFullEndpoint = true;
14 }
```

in which 3 of the 4 assignments in each branch are identical; we recommend factoring for simplicity and clarity, as in:

```
CurrentA_PointVelocity = CurrentB_PointVelocity;
CurrentB_PointVelocity = (depth==0) ? CurveEndPointVelocity
                                    : *ptrInterMedFT[depth];
SubStart_PointVelocity = CurrentA_PointVelocity;
restoredFullEndpoint   = true;
```

- In `EstimateIntersectionPoint()` we have a negatively-named variable `there_is_no_intersection` which is always used as `! there_is_no_intersection`. We recommend the variable be renamed `there_is_an_intersection`, be given the opposite polarity in value, and thereafter used without need for further negation.

G4BrentLocator

Most of the remarks re `G4MultiLevelLocator` apply to this class as well.

G4SimpleLocator

Many of the remarks re `G4MultiLevelLocator` apply to this class as well.

3.8 G4PropagatorInField

- The general comments are split among the `.hh`, `.icc`, and `.cc` files. As discussed earlier, we recommend the comments be consolidated in a single place.
- Only the last 4 lines of the constructor body belong there. Consistent with our previous general recommendations regarding initialization, we recommend the remaining lines be reformulated as part of the initializer list.
- The body of `FindAndSetFieldManager()` has 10 lines of code, involving nested logic and 2 local variables. This entire body can be simplified as follows, using no nesting and no local variables:

```
if( pCurrentPhysicalVolume )
  fCurrentFieldMgr = pCurrentPhysicalVolume->etc;
if( ! fCurrentFieldMgr )
  fCurrentFieldMgr = fDetectorFieldMgr;
fSetFieldMgr = true;
return fCurrentFieldMgr;
```

- `ComputeStep()` calls `FindAndSetFieldManager()`, and uses its return value to set the same variable to the same value as was just done in the call to `FindAndSetFieldManager()`. This assignment therefore seems to be redundant and can be removed.
- `ClearPropagatorState()` fails to clear all the data members as the name promises; for example, any `fpTrajectoryFilter` pointer is preserved.
- The body of `GimmeTrajectoryVectorAndForgetIt()` (a whimsical name!) can be slightly simplified and clarified:

```
return fpTrajectoryFilter
     ? fpTrajectoryFilter->GimmeThePointsAndForgetThem()
     : 0;
```

- A comment refers to a "METHOD" but technically C++ has no methods, only member functions.
- Many functions (here and elsewhere) use the convention of declaring variables with no initial value, then later (often immediately or very soon) providing the intended value. As pointed out above, such a practice is in general unnecessarily expensive especially when the variables' types have default constructors which are thus unnecessarily invoked.

- `ComputeStep()` has a `do...while()` loop at its core. This is a rarely-needed construct (as pointed out elsewhere) that is likely wrong here, as it is possible (even though highly unlikely here) to set the maximum number of iterations to zero, in which case the loop still would iterate once before discovering that it had done so unnecessarily.
- In `ComputeStep()`, the variable `s_length_taken` may be unnecessary and might be replaced by `fFull_CurveLen_of_LastAttempt`.
- In `ComputeStep()` we have

```
1 intersects = intersects && found_intersection;
```

  in a block not entered unless `intersects` is known to be true. This fragment can therefore be simplified as:

```
1 intersects = found_intersection;
```

  Even better, the variable `found_intersection` can be removed and its uses replaced by `intersects`, thus obviating the above assignment.
- `GetThresholdNoZeroSteps()` might be better written, with no local variable, via a single `switch` statement:

```
1 switch( i )  {
2   case 0:  return 3;
3   case 1:  return fActionThreshold_NoZeroSteps;
4   case 2:  return fSevereActionThreshold_NoZeroSteps;
5   case 3:  return fAbandonThreshold_NoZeroSteps;
6   default: return 0;
7 }
```

3.9 G4ErrorPropagatorData

- This class is intended to be a singleton and should therefore disallow copying.
- The static data member `theErrorPropagatorData`, a pointer, would be better defined inside `GetErrorPropagatorData()`; the data member has no need to be accessible from outside this function.
- Even better, that variable does not need to exist at all, as `GetErrorPropagatorData()` can simply directly return `&GetErrorPropagatorData` and thus be reduced in size by 4 lines of code (by removing the `if` statement).
- A name of the form `GetSomething()` suggests that its purpose is to access member data. The conventionial name, in a singleton, for the instance function is `instance()`.
- All functions can trivially be made `inline`, and should be. This change will produce a small performance benefit on each use of the class with little or no impact on code size.
- The `const_cast` in `SetTarget()` is incredibly dangerous and should be avoided. The simplest correction is to change the type of the data member as shown: `const G4ErrorTarget* theTarget;`.
- No default constructor is provided, leaving data members in inappropriate (likely undefined) states that can be accessed from other parts of the program.
- It is unclear that this class should be designed as a singleton, as it is simply a repository of selected data items that could each be `static` data members. Such a simpler design could yield small performance benefits as there would then be no need to provide and hence call any instance function before accessing the data, especially if all the member functions were `static` as well.
- Once all the members are `static`, and observing that each function is either a setter or a getter, it is unclear that most of these functions need to exist.
- The `enum` definitions could easily be moved inside the (`public` part of the) class, in which case (a) the enumerators' identifiers would no longer need to have their type prepended, and (b) client code would need only replace the existing `_` with `::` and then recompile.

3.10 Header templates.hh

- The definition of abs (or anything else) in namespace std is not permitted by language rules: it evokes undefined behavior.
- The #defines seem to have outlived their usefulness.
- The definition of the sqr template does seem useful, but the name of the header does not at all suggest that this functionality (or any other particular functionality) is here.
- G4SwapPtr<> seems entirely superfluous as it only duplicates functionality already present in std::swap().
- G4SwapObj<> seems marginally useful, but should be implemented as a single call to std::swap(*a, *b).

4 A small experiment in code improvement

Based on the above findings, we did not expect to see significant performance improvements from implementing any single recommendation. We did make a limited modification of the code pursuant to our comments involving several of the functions in the G4ChordFinder class. Indeed, these few transformations produced no measurable effect in the cmsRun context tracking particles from Z' decays with no caching.

5 Acknowledgments

We are grateful for advice and assistance received from Sunanda Banerjee, Daniel Elvira, Mark Fischler, Jim Kowalkowski, Marc Paterno, Liz Sexton-Kennedy, and Julia Yarba.