Report from the Projection Matching Algorithm code analysis

Code authors: Dorota Stefan, Robert Sulej
Analysis team: Chris Jones, Jim Kowalkowski, Rob Kutschke,
Marc Paterno, Gianluca Petrillo, Erica Snider

Version 1
May 18, 2016

1. Goals of the analysis

As part of developing a process for systematic, formal process for analyzing code contributed to the LArSoft suite, the LArSoft Core Team assembled a group of software design experts to assist with analyzing the Projection Matching Algorithm (PMA) code written by Dorota Stefan and Robert Sulej. The general objectives of the analysis process include:

- Evaluate the compliance of the code with high-level LArSoft design principles, those recommended by the `art` Team regarding interactions with the framework, and more general software engineering best practices gleaned from the experience of the analysis team, and make recommendations as necessary for improved compliance.
- Evaluate the computing and memory performance of the code and identify optimizations that can improve the performance.
    - The metrics to be used are CPU time and memory footprint for a selected test job
- Evaluate the low-level coding practices and make recommendations that improve the overall performance, clarity, maintainability, error handling, etc.
    - CPU and memory usage might serve as suitable metrics in some of these cases as well.

It is important to note that in principle, the target of the analysis and the resulting recommendations include not only the PMA code itself, but also any relevant LArSoft and `art` framework infrastructure or policies that might be relevant within the context of the PMA code. In addition to these general objectives, this particular analysis had the additional goal of exploring how LArSoft might proceed with such analyses in the future.

As will be discussed below, time constraints limited which of these general areas could be investigated.

In the next section we describe the analysis process used for this review. Section 3 then presents the recommendations from the analysis, providing descriptions of the problem and suggested remedies. This is followed in Sect. 4 by issues noted by the analysis team that require more thought or investigation before any recommendation for changes can be made, and by a list of findings that do not require follow-up in Sect. 5. We then conclude with comments on the process from the code authors in Sect. 6, and some "lessons learned" from the code analysis in Sect. 7.

2. The analysis process

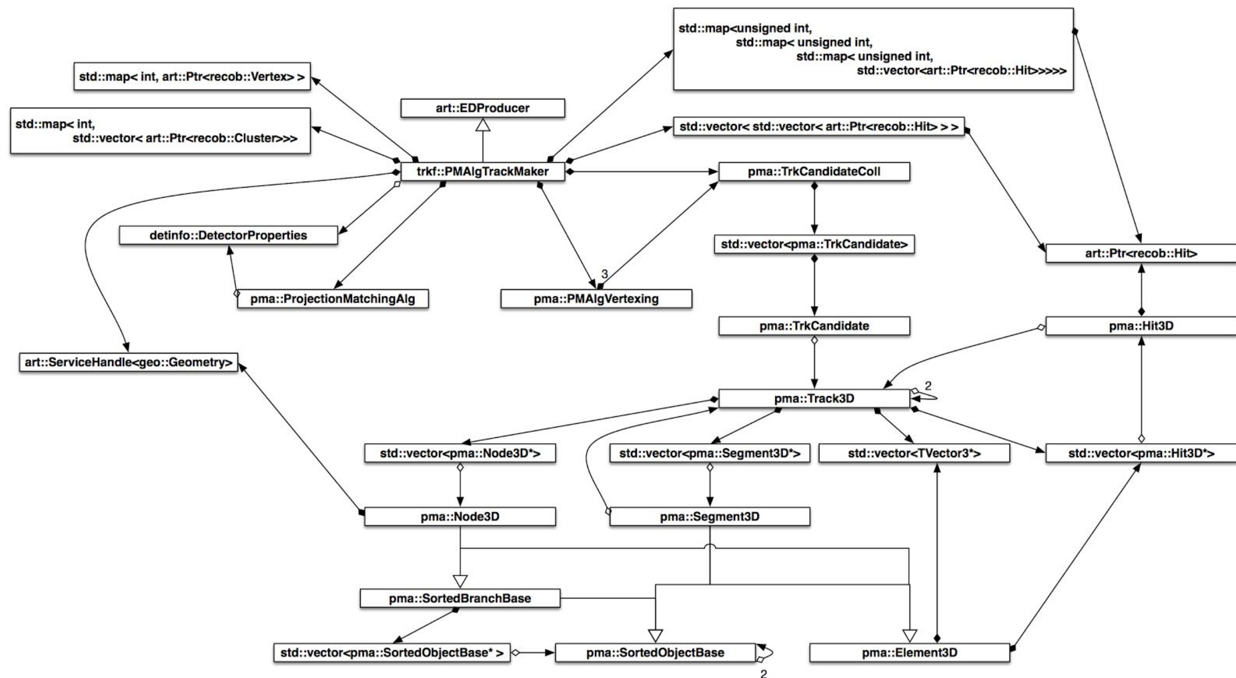
The analysis was carried out in one 3-hour meeting on May 10, and a second 2-hour meeting on May 11, 2016. Due to the limited time available, the technical scope of the analysis was limited to two areas, as determined by the review team during initial meetings:

- Low-level data structures and CPU / memory optimization across the PMA code
- High-level design and interfaces, and the structuring of framework and algorithm interactions

The bulk of the first meeting was devoted to the first topic, while that of the second was aimed at the second.

In preparation for the meeting, the analysis team copied the entire `larreco` repository (which contains the PMA code) into the `larsoft-test` repository on github one week in advance. The choice of github was driven in part by the set of collaborative tools provided that allow team members to make line-by-line comments and initiate discussion threads. A day before the meeting, the authors provided reference `fhicl` and input data files that could be used for testing. Team members then ran `valgrind` on a sample job that used PMA. The output of this

analysis would subsequently be used as a reference during the optimization discussion. The team also created a UML-ish diagram of the code being reviewed (see below) prior to the meetings.



We began the first meeting by reviewing objectives, and having the authors review the basic operation of the algorithm, introducing along the way some of the important abstractions used by the algorithm. The team then began the formal analysis work by stepping through elements of the code, discussing issues as they arose or as discovered during preparatory reading of the code. We also ran `igprof` on the sample jobs. We found the `igprof` output more useful than the valgrind output. `igprof` was fast enough to allow us to run it several times during the review.

Between sessions, the authors applied some of the CPU performance suggestions, and re-ran `igprof`.

The second day began with a review from the authors on the overall structure of the code, followed by discussion of use cases, points of design, and interactions between classes.

Follow-up work after the meeting includes writing a report, prioritizing implementation of recommended changes, tracking work on those changes, and discussing issues raised for which there were not yet recommended changes, or for which there was insufficient effort available to address within the desired time frame.

We should note that the entire analysis was greatly facilitated by the clear structure and comments in the existing PMA code. Many of the large scale changes to be suggested will involve primarily moving code between classes rather than wholesale rewrites.

3. Recommendations

The recommendations in this section are organized as follows:

- A short title and number that can be used as identifiers.
- A "type" that tags the general problem area, and that speaks to the scale of the issue or the level at which it affects the code. For this review, it will be one of the following:
  - "Design / architecture"
  - "LArSoft coding guidelines"
  - "*art* coding guidelines"
  - "C++ coding practices"
  - "Code management"
- An "issue" that describes the problem noted. It can be thought of as a sort of "finding".
- An "owner" identifying the person / group / entity (as estimated by the analysis team) nominally responsible for any follow-up on the recommendation. For the current review, this will be one of the following:
  - "Code authors" (which in general implies DUNE responsibility in this case)
  - "LArSoft team"
  - "*art* team"
- A "recommendation" that specifies what actions to take to correct a problem or improve upon the existing code.

It is understood that prioritizing the implementation will occur as part of the follow-up work, although the analysis team may note issues deemed to be particularly important to correct in a timely manner.

3.1. PMA module design

Type: Design / architecture

Issue: The current design has a single module

Owner: authors

Recommendation: We suggest having two or three modules that use a common toolkit. Each module would need only those parameters meaningful to that module, and each would produce and consume data products meaningful for that module.

It seems the toolkit can be built by changing member functions to free functions or helper classes, and member data to arguments of functions. Configuration of the module should remain as module state.

Temporary state of the algorithm should not be part of the state of the module. This will automatically avoid a problem seen in other LArSoft code, in which excessive memory use results from temporary state being held longer than needed.

3.2. Unwarranted use of TObject

Type: C++ coding practices

Issue: The unwarranted use of TObject sub-classes can in some circumstances result in considerable and unnecessary memory and CPU overheads.

Owner: Code authors + LArSoft

Recommendation: The use of TObject sub-classes should be carefully considered and avoided in cases where there might be an adverse impact on performance. The replacement of `TVector2` and `TVector3` by `CLHEP::Hep2Vector` (or `Hep3Vector`) or `ROOT::Math::DisplacementVector2D<...>`, for instance, should be implemented whenever the functionality of the TObject base class of TVector* is not actively utilized. Making this single change yielded a 2-fold increase in speed for the PMA code.

LArSoft should adopt and promulgate this recommendation as a policy.

In some cases, the underlying calculations should be written and optimized by hand. Another factor of 1.5 in speed was obtained in the present case by careful attention to numerics in the most time-intensive part of the code.

3.3. Vector class and linear algebra standardization

Type: architecture

Issue: There is no guidance on the use of 2- and 3-vector classes or linear algebra libraries

Owner: LArSoft

Recommendation: LArSoft should adopt a policy for what 2-, 3- and 4-vector class(es), and what linear algebra libraries to use.

The ATLAS collaboration studied alternatives for the linear algebra classes. Their results were presented at CHEP-2013:
https://cds.cern.ch/record/1608611/files/ATL-SOFT-SLIDE-2013-822.pdf

3.4. Bare pointer usage for Node3D and Segment3D

Type: C++ coding practices

Issue: The handling of `pma::Node3D` and objects is managed via bare pointers, with `new` and `delete` statements used to create the objects.

Owner: Code authors

Recommendation: We recommend use of smart pointers (`std::shared_ptr`, `std::unique_ptr`, or `cet::value_ptr`) and their accompanying helper functions (`std::make_shared` and `std::make_unique`) in the design of this code. This will provide exception safety and improved ease of maintenance. In general, we should strive for

classes in which the compiler-generated destructor does the right thing, which is not possible when `new` is and `delete` is required to manage object lifetime.

3.5. Error handling in LArSoft modules

Type: LArSoft coding guidelines

Issue: The definition of error conditions, and the actions taken in response (including whether to throw an exception) is completely defined by code authors.

Owner: LArSoft

Recommendation: LArSoft needs an error handling policy, saying how module code should respond to error conditions, what exceptions should be thrown, what should be reported to the framework, and how to configure the framework to respond appropriately. The policy should also prescribe what common conditions constitute an “error” versus a “warning”, etc. An education campaign will then be needed to disseminate this information.

3.6. The unnecessary use of std::pair

Type: C++ coding practices

Issue: There are cases where std::pair is used when not necessary (i.e., not using a standard library class that requires it), and at least two cases where a `std::pair` is used to store two identical object types: `std::pair<unsigned, unsigned>` and `std::pair<TVector2, TVector2>`.

Owner: Code authors

Recommendation: Use of `std::pair` objects should be avoided unless required. We recommend instead using a meaningfully-named two-element `struct` with meaningfully-named member data, a solution which is particularly helpful when the type of the two data members are the same. Adhering to this guidance is important for later maintenance.

3.7. Use of literal numeric constants

Type: C++ coding practices

Issue: There are places in the code where literal numeric constants are used.

Owner: Code authors

Recommendation: Replace literal numeric constants with symbolic constants, enumerations, or `constexpr` objects or functions. For example, the literal "3" used in some places to denote the maximum number of views should be replaced by a named constant.

Please refer to the notes added to the pull request in the larsoft-test repository in GitHub for specific instances.

3.8. Creation of TTree objects in reconstruction algorithm code

Type: LArSoft coding guideline

Issue: The PMA module creates a `TTree`.

Owner: Code authors + LArSoft (for the policy)

Recommendation: Creation and filling of `TTree`'s should in most cases be moved to a separate analyzer module. We note this is especially important for multi-threading reasons: interactions with `TFileService` or ROOT facilities require locking, and thus yield poor multi-threaded performance. While embedded diagnostic code is valuable when developing new code, it should generally be removed before using modules in production settings. Transient data products can be used to transmit transient data from the PMA module into an analysis module.

LArSoft should develop a policy for how and when `TTree`'s should be used within algorithm code.

3.9. Configuration parameter validation

Type: *art* coding guidelines

Issue: The existing code doesn't validate the configuration parameters

Owner: Code authors + LArSoft (for the policy)

Recommendation: Parameter set validation as provided by the facilities in fhicl-cpp should be utilized whenever feasible. This will be easier to do when the recommended module split-up is done.

LArSoft should adopt a policy regarding parameter set validation.

3.10. Use of enumeration values in parameter sets

Type: *art* coding guideline

Issue: There are places where the values from an enumeration are used as the values in parameter sets.

Owner: Code authors

Recommendation: When using parameter sets to determine an enumeration value to use at run-time, one should use names that represent the value in the parameter set, then convert to the appropriate enumeration value in the code. The code can then use the enumeration values directly. We can recommend some code examples showing how this can be done.

3.11. Use of non-const class-level statics

Type: C++ coding practices

Issue: There are places where non-`const` class-level statics are used.

Owner: Code authors

Recommendation: Use of non-const class-level or function level statics and globals should be avoided. The existing ones in the code appear to be candidates for being declared `const`.

Consider use of `constexpr` whenever possible.

Some of the static data should become non-static member data, for example `pma::Element3D::fOptFactors` should be eliminated, and replaced where necessary by function arguments that pass the required values. This change would allow different instances of `Element3D` to be used simultaneously without interference.

3.12. Use of strings for input data product selection

Type: `art` coding guidelines

Issue: The PMA module uses `std::string` values to select input data products.

Owner: Code authors + LArSoft (for the policy)

Recommendation: Use `art::InputTag` rather than strings for input data product selection. This choice makes the intent manifest, and configuration files more explicit in what products are being selected. Using input tags will also help in parameter set validation.

LArSoft should develop a policy regarding the use of `art::InputTag` values to select input data products.

3.13. gprof compiler flag set during default build

Type: Code management

Issue: The default build for larreco has the `-pg` flag set enabling generation of `gprof` code, but the use of `gprof` is deprecated.

Owner: LArSoft

Recommendation: This switch should not be used. The “profile” build does not require it, and profilers such as igProf or Allinea MAP do not require it. (This was discovered during profiling where a call to `_init` was observed.)

3.14. Construction of std::unique_ptr

Type: C++ coding practices

Issue: Instances of std::unique_ptr are being explicitly constructed.

Owner: Code authors

Recommendation: Explicit construction of `unique_ptr` should be replace with the use of `std::make_unique` and using `auto` for the declaration of the `unique_ptr` object.

3.15. Use of braces in for and if statements

Type: C++ coding practices

Issue: Some one line `for` and `if` blocks are written without explicit braces around the statement.

Owner: Code authors + LArSoft

Recommendation: All `for` and `if` statements should be followed by explicit braces. Failure to do so can cause trouble with maintenance.

LArSoft should have a policy on this issue.

3.16. Improved support for use of IgProf

Type: Code management

Issue: `IgProf` is not available within the default LArSoft environment

Owner: LArSoft

Recommendation: `IgProf` should be made more widely available to the LArSoft community, and delivered by default along with other supported software tools by whatever means is appropriate. Guidance and support for using `IgProf` should also be provided as part of this policy.

3.17. Unnecessary use of for loop counters and iterators

Type: C++ coding practices

Issue: There are cases where for loops are constructed with loop counters and iterators when an implicit range is available.

Owner: Code authors

Recommendation: Use range-for loops instead of explicit iteration whenever implicit ranges are available.

4. Items for further investigation and discussion

The following items were noted during the review as issues that require additional investigation or thought before a recommendation can be made. The code authors are strongly advised to follow up on these items, seeking expert input as needed, in order to determine whether further actions are required.

The format of these items is the same as that of the recommendations, except that a final “comment” is listed instead of a “recommendation”.

4.1. Tree structure of pma::Node3D and pma::Segment3D

Type: C++ coding practices

Issue: The data structure representing a tree of `pma::Node3D` and `pma::Segment3D` objects has more generality than required, which leads to overly complicated code to deal with it, and is not type safe since it requires the use of `dynamic_cast`.

Owner: Code authors

Comment: The design of this tree-like structure could likely be simplified by having a structure that enforces more of the necessary constraints on types and removes the need for dynamic and static-casting. We also recommend the use of smart pointers (`std::shared_ptr`, `std::unique_ptr`, or `cet::value_ptr`) in the design of this code, as noted in Recommendation 3.4.

4.2. Design of object corresponding to cryo_tpc_view_hitmap

Type: C++ coding practices

Issue: Use of nested maps to represent a multi-dimensional index

Owner: Code authors + LArSoft

Comment: The typedef `cryo_tpc_view_hitmap` should be reviewed in more detail. Consider turning the three separate integer indices into a dense index. The one case in which only the first two indices are used to look up a map of `view->vector<Hit>` needs careful consideration.

A general solution supported by the LArSoft geometry service should be considered and adopted if possible.

4.3. The use of art::Ptr<> templates

Type: *art* coding guidelines

Issue: `art::Ptr<>` templates may be used in cases where a less complex object may suffice (e.g., persistency is not required)

Owner: Code authors

Comment: The review team did not have time to look into the details of how `art::Ptr<>` templates are used in the code. Consider removing them from inner loops, and from temporary data structures. Making objects smaller reduces memory use and can improve speed. Also, the validity check in art::Prt<> can be avoided, which again improves execution speed. It is possible to generate the `art::Ptr<>` objects needed in persistent objects at the end of the algorithm's work. The tradeoffs in speed, size and complexity needs to be evaluated.

4.4. Data object design

Type: *art* coding guidelines + LArSoft coding guidelines

Issue: The team did not have time to analyze the use and design of data products

Owner: Code authors + LArSoft

Comment: Follow-up work should include evaluation of how well the data products meet the needs of the algorithms, how well they capture the output of the algorithms, and how well they meet the needs of later stages of processing (including analysis).

4.5. Clearing transient data at the beginning of events

Type: art coding guidelines + LArSoft coding guidelines

Issue: PMA module (and some other modules) clear member data representing transient, non-configuration data at the beginning of events.

Owner: LArSoft

Comment: Data structures that contain transient data and that remain in scope at the end of the event should be cleared at the end of the event rather than at the beginning of the next event. The LArSoft team should investigate how often this pattern occurs, and educate users in the perils of using it.

A better solution is to have transient data go out of scope at the end of each event.

4.6. The use of statics

Type: *art* coding guidelines + LArSoft coding guidelines

Issue: Do we need a policy on statics, particularly non-const statics?

Owner: LArSoft ??

Comment:

4.7. Vector sqrt instruction

Type: Code management

Issue: Noted that the IEEE `sqrt` function was removed from the system in favor of the vector `sqrt` instruction

Owner: LArSoft ??

Comment: This was a surprise. The compiler switches that permit it should be identified.

4.8. Replacing Assns creation with new utilities

Type: LArSoft coding guidelines

Issue: New utilities for creating Assns will soon be available

Owner: Code authors

Comment: When available, the authors should consider replacing the current Assns creation with the use of the new utilities.

5. Findings and comments requiring no follow-up

There were no findings other than those listed above.

6. Code author comments on the process

We find the process instructive and helpful for the further PMA development. The limited time did not allow to reach many details of the implementation, but on the other hand we may consider this was the first round of a larger process. The amount of recommendations resulting from this review seems for us reasonable to be addressed before we are ready for analysis of the PMA code parts not looked at during this review.

7. Lessons learned

1. Keeping transient data as module state between events can significantly and unnecessarily increase the memory footprint of modules.
2. TObject memory and CPU overheads can have a significant impact on overall performance in some cases. Care should be exercised when choosing to use TObjects sub-classes.
3. IgProf was much more convenient that Callgrind for profiling work.
4. The GitHub repository was very convenient for commenting.
5. Cloning the repository and using local tools was faster for looking at large amounts of code.
    a. For structural reviews, the GitHub commenting was not very useful.
    b. For code conformance and best practices comments, the GitHub commenting was very useful.
6. The search facilities in git itself (e.g. git grep) are useful in looking at code.
7. The git history facilities are also useful.