MAUS Work Specification Document

Change History

Version	Changes	Author	Date
1	First draft	C.Rogers	01/10/2018

Project Summary

Project Title	Cuts structure update
Main Issue	
Subtask Issues	
Project Lead	
Project Supervisor(s)	C. Rogers
Associated Manpower	

Table of Contents

Overview of Work

Motivation and Overview

The analysis group seeks to reject certain recon events based on characteristics of the data. For example, we may reject recon events because there appears to be more than one particle passing through the apparatus, because one of the detectors did not operate correctly, etc. The analysis group needs to understand the effect of using or not using the Cut; so the relevant recon events should be kept in the data structure, but flagged as accepted or rejected.

This has been implemented by Misha in the data structure as a *Cuts* object. The Cuts object is a wrapper for *std::vector<std::bool> _cut_store* containing flags indicating whether a particular cut has passed or failed; and a static map naming each cut in the vector.

Additionally a mapper has been developed to apply a few cuts, *MapCppCuts*. The code is stored in launchpad branch *lp:~mfedorov/maus/clean_branch*. A few different cuts have been implemented.

As the analysis has developed, it has become clear that this nice piece of work needs to be developed further. In particular

- The Cuts object additionally needs to store the value that was cut against; e.g. if the cut is on TOF01 time, then the Cuts object should store a boolean indicating whether the cut was passed or failed, and the TOF01 time. This would enable analysts to estimate the effect of varying the cut value without reimplementing the full cut logic.
- Actually quite a few different Cuts need to be implemented in MapCppCuts. A more systematic way (e.g. some abstraction layer) needs to be implemented to more clearly systematise and separate the logic.

Task Breakdown

Merge code

Merge the existing Cuts code into the trunk.

WillCut Class

A new class should be implemented, the WillCut object. This is a templated class. The class has two members, a boolean, indicating that the object should cut, and a templated member storing the value of the cut. The usual Getters and Setters should be implemented.

Question: Should we overload the logic operators so WillCut looks like a boolean? i.e. is it possible to seemlessly implement something so statements like

```
if (my_will_cut) {
}
if (my_will_cut && foo) {
}
```

work seamlessly? Is this even possible?

Cuts class

The _cut_store member of the Cuts class should be of type *std::vector<WillCut>* in place of the existing *std::vector<bool>*. Those functions that enable getting or setting data in the _cut_store should be updated to take a *WillCut* object instead.

Cutter object

Looking at the *MapCppCuts* class, we basically have a whole set of member functions like

bool CheckSomeCut(ReconEvent event) const;*

Then we run through the member functions, getting relevant cuts at birth time and applying them at process time. This structure can be formalised by making a Cutter base class. The Cutter base class looks like this:

```
class Cutter {
    public:
        void birth(Json::Value cuts) = 0;
        WillCut process (ReconEvent* event) = 0;
        void death() = 0;
};
```

MapCppCuts should store a *std::vector<Cutter*>* and iterates over each member of the vector at birth time, calling the birth function;, process time calling the process function; and death time calling the death function.

The data cards contain a “cuts” dictionary containing a mapping of string name to [low, high] bounds of the cut values. This dictionary is handed to the Cutter class at birth time. The idea is to make things quite systematic and keep all of the data pertaining to *MapCppCuts* in one place in the data cards (they are rather sprawling currently).

Implement existing cuts as Cutter objects

The existing cuts should be reimplemented as Cutter objects. Hopefully this is mostly just cut'n'paste.

- Exactly one space point in TOF0
- Exactly one space point in TOF1
- TOF01 time
- Exactly one track in TKU
- Momentum in TKU station 1
- Change in Momentum between TKU and TKD
- P value in TKU
- “Mass cut”

The code should be merged after this job is done.

Implement new cuts

Some additional cuts will need to be implemented. *Question: should it be responsibility of analysts to implement these cuts (And any others they need for their analysis)?*

- Number of clusters in TKU track; reject event if all TKU tracks have less than *n* clusters
- Number of clusters in TKD track; reject event if all TKU tracks have less than *n* clusters
- TKD momentum
- TKD chi2
- TKU chi2
- Exactly one space point in TOF2
- TOF12 time
- Upstream aperture cuts; reject event if the track radius of the global upstream track extrapolated to the virtual plane with a particular station number is more than some value. Aperture cuts should be implemented for each of the principal apertures, namely:
  - diffuser
  - TKU butterfly
  - Upstream PRY
  - lh2 upstream window
  - absorber centre
  - lh2 downstream window flange
  - SSD aperture
  - TKD He window
  - TKD stations
- Track matching between TKU and TOF1
- Track matching between TKU and TOF0
- Residual position between extrapolated track from TKU to TOF1
- Residual position between extrapolated track from TKU to TOF0
- Residual time between extrapolated track from TKU and TOF1 to TOF0
- Residual time between extrapolated track from TKU and TOF1 to TOF2
- Residual position between extrapolated track from TKU to TKD and any TKD track
- Residual momentum between extrapolated track from TKU to TKD and any TKD track
- Residual position between extrapolated track from TKD to TOF2

MAUS Overview and Requirements

MAUS is an analysis tool designed for the reconstruction and analysis of the Muon Ionisation Cooling Experiment.

The MAUS project uses a number of tools to manage code and the development process. Issue trackers are used to monitor development of new features and log any issues arising such as bugs. Code is stored using Version Control System (VCS). Communication between the development team is achieved using mailing lists, regular phone conferences etc. Code quality is assured by a set of style requirements, testing requirements, documentation of code and regular code reviews.

These tools are documented here for convenience, but additional documentation can be found on the MAUS website http://micewww.pp.rl.ac.uk/projects/maus/wiki/MAUSDevs. Where there is a conflict, the MAUS website takes precedence as this is more likely to be updated.

Issue Tracking

The redmine issue tracking system is used to mointor development of new features and log any issues arising such as bugs. Any information pertaining to the features in this document shall be stored in the relevant issues on that system. Redmine issues can be found by following the *Issues* link on the MAUS website.

Code Version Control

Code is controlled using the *bazaar* Distributed Version Control System. **Before** writing code, developers shall checkout a copy of the maus trunk (*lp:maus*). Developers shall then make their own development branch on launchpad. Code shall be pushed regularly to this development branch (typically nightly). Every release, the MAUS trunk shall be merged with the development branch to ensure that development is performed against the latest MAUS version.

Further guidance on bzr usage can be found on the MAUS wiki.

Communication

Communication is performed using mailing lists and fortnightly phone meetings. All developers should be subscribed to the following mailing lists:

- mice-software
- maus-devs
- maus-users

Where possible, developers shall attend the fortnightly phone meetings, as announced on the mice-software mailing lists.

Code Quality

Code quality is ensured by a number of tools. Code must be commented and conform to certain style requirements. All code must have appropriate regression tests. Documentation must be created or updated as appropriate. Code may be reviewed by one or more developers before submission.

The workflow that ensures code quality is controlled by the workflow field in the issue tracker. As code moves through the workflow, this field should be updated.

Coding Style and Comments

Code shall be written in the appropriate style. For C++, we follow the google style guide; for python we enforce the standard python style guide. Scripts that automatically check for code style are provided and are executed upon execution of the unit tests.

All public functions shall be commented. C++ functions should have a description in the header file. Python functions should be commented using python standard docstrings. Comments should include the purpose of the function, the definition of any input parameters and the values that can be returned. Python comments should specify possible types for all input and output parameters. Comments should be formatted for the Doxygen tool.

Testing

Unit tests shall be provided by the developer for all code they develop at the 90% line coverage level. We aim to provide about 90% line coverage for our unit tests. Tests for modules (maps, reduces, inputs, outputs) should be written in python and kept in the folder for that module. Tests for common code (src/common_cpp, src/common_py) should be written in the same language as the code to be tested and kept in tests/cpp_unit or tests/py_unit respectively. All C++ code should be compatible with the google testing framework. All python code should inherit from the python unittest framework.

The MAUS test server is used to replicate environments that the code is required to be run in such as the MICE control room and standard high energy physics clusters. All tests must pass not only in the developers local environment but also in the test server environment.

Documentation

High level documentation should be written in latex and placed in the doc/doc_src area. All user interfaces should be documented (either input or output) in latex.

Code Review

All code by developers new to the project shall be reviewed by the project supervisors. Additional reviewers, or code reviews, may be requested by the project supervisor. The code should be reviewed by the project supervisor when it is ready to merge and all tests pass on the test server.

Code review is a way of:

- finding bugs
- spreading knowledge of the code to other developers
- ensuring high quality code by:
    - checking that code is adequately tested
    - checking that the code is documented properly
    - checking that the code has the correct style

When code is ready for review, please set the workflow field on the issue tracker to the appropriate value.

Effort and Timescale

Effort Available

Major Milestones