The Muon Ionization Cooling Experiment User Software

A Dobbs\textsuperscript{1} and D Rajaram\textsuperscript{2} for the MICE Collaboration

\textsuperscript{1} Physics Department, Blackett Laboratory, Imperial College London, Exhibition Road, London, SW7 2AZ, UK
\textsuperscript{2} Physics Department, Illinois Institute of Technology, Chicago, IL 60616, USA
E-mail: durga@fnal.gov

Abstract. The Muon Ionization Cooling Experiment (MICE) is a proof-of-principle experiment designed to demonstrate muon ionization cooling for the first time. MICE is currently on Step IV of its data taking programme, where transverse emittance reduction will be demonstrated. The MICE Analysis User Software (MAUS) is the reconstruction, simulation and analysis framework for the MICE experiment. MAUS is used for both offline data analysis and fast online data reconstruction and visualization to serve MICE data taking. This paper provides an introduction to MAUS, describing the central Python and C++ based framework, the data structure and and the code management and testing procedures.

1. Introduction
1.1. The MICE Experiment
The Muon Ionization Cooling Experiment (MICE) sited at the Rutherford Appleton Laboratory (RAL) will be the first demonstration of muon ionization cooling – the reduction of the phase-space of muon beams. Muon beam cooling is essential for future facilities based such as the Neutrino Factory of Muon Collider [1]. The experiment is designed to be built and operated in a staged manner. In the first stage (Step I), the muon beamline was commissioned [2] and characterized [3]. The present step – Step IV – will study the change in normalized transverse emittance using lithium hydride and liquid hydrogen absorbers under various optical configurations.

1.2. Software Requirements
The MICE software must serve both the accelerator physics and particle physics needs of the experiment. Traditional particle physics functionality includes reconstructing tracks, identifying particles, and simulating the response from various detectors, while the accelerator physics aspect includes computing transfer matrices and Twiss parameters and propagating beam envelopes. All of these require knowledge of the beamline, geometries of the detectors, knowledge of the magnetic fields, and functionality to reconstruct or simulate various detectors, thus necessitating a single software scope. Given the complexity and the inherent time-scales of experiments, the need also arises to ensure long-term correctness and maintainability of the software used for the experiment.
2. MAUS

The MICE Analysis User Software (MAUS) has been the experiment’s simulation, reconstruction, and analysis software framework since 2010 and provides capabilities to 1) perform a Monte Carlo (MC) simulation of the experiment, 2) reconstruct tracks and identify particles from simulations and real data, and 3) provide monitoring and diagnostics tools while running the experiment.

Installation is by a set of shell scripts with SCons [4] as the build tool. The codebase is maintained with the GNU Bazaar revision control system [5] and is hosted on Launchpad [6]. MAUS has a number of dependencies on standard packages such as Python, ROOT [7] and GEANT4 [8] which are built as “third party” external libraries during the installation process. The officially supported platform is Scientific Linux 6 though developers successfully build on CentOS, Ubuntu, and Fedora distributions.

2.1. Architecture

The architecture is inspired by the MapReduce model in order to simplify the interfaces that developers have to follow and aid running the code in parallel and it was felt that MapReduce parallelizes particle physics data flow problems in a useful fashion. The basic building block of the MAUS framework is a module. There are four types of modules in MAUS:

(i) Inputters generate input data either by reading data from files or sockets, or by generating an input beam.

(ii) Mappers modify the input data, for example by reconstructing signals from detectors, or tracking particles to generate MC hits.

(iii) Reducers collate the mapped data and allow functionality that requires access to the entire data set.

(iv) Outputters save the data either by streaming over a socket or writing data to disk.

There are some objects that sit outside the scope of this modular framework but are nevertheless required by several of the modules. For instance, knowledge of the detector geometries, magnetic fields, and calibrations are required by the reconstruction and simulation modules, and objects like electronics cabling maps are required to unpack data from the data acquisition (DAQ) source, and error handling functionality is required by all of the modules. All these objects are accessed through a static singleton globals class.

The principal event type is the spill. A single spill corresponds to data from the particle burst associated with a dip of the MICE target. A spill typically lasts 3ms and contains several DAQ triggers. Data from a given trigger correspond to a single MICE event. In the language of the Input-Map-Reduce-Output framework above, an Input module creates an instance of spill data, a Map module processes the spill (reconstructing, simulating, etc), a Reduce module acts on a collection of spills when all the mappers finish, and finally an Output module stores the MAUS data structure to the output.

MAUS has two execution concepts. A job refers to a single execution of the code, while a run refers to the processing of data for a DAQ run or MC run. Since data are typically accessed from a single source and written to a single destination, Inputters and Outputters are initialized and destroyed at the beginning and end of a job. On the other hand, Mappers and Reducers are initialized at the beginning of a run in order to allow loading run-specific information such as electronic cabling maps, fields, and calibrations.

Developers are allowed to write modules in either Python or C++ and Python bindings to C++ are handled through internal abstractions. C++ is used for complex or low level algorithms where computation time is important while Python is used for simple or high level algorithms where development time is a more important criterion. In practice, all the
reconstruction modules are written in C++ but support is provided for legacy modules written in Python.

MAUS has an Application Programmer Interface (API) that provides developers with a well-defined environment for developing reconstruction code, while allowing independent development of the backend and code-sharing of common elements like error handling and data manipulation.

Data can be represented in two formats. The default data format is a ROOT binary and the secondary format is JSON [9]. This is an ascii data-tree format readable with any text editor. Specific JSON parsers are also available - for example, the Python json module is available and comes prepackaged with MAUS. There is a third special data type that MAUS handles, the Image type used by Reducers to output images of monitoring histograms, efficiency plots, etc and this is available only in JSON format.

Modules can exchange data either as C++ or JSON object types. In Python, the data’s format can be changed by using a converter module and in C++, mappers are templated to a MAUS data type and the API then handles any necessary conversion to that type. During production deployment of the software it was found that there was a significant hit in the performance speed due to inherent slowness in (de)serializing JSON objects. Hence it was decided that modules for official reconstruction and simulations must exchange C++ objects as a default in order to minimize need for conversion between data types. However, data can still be output in JSON format and developers find it extremely useful during debugging.

2.2. Data structure

The principal part of the MAUS data structure is a ROOT Tree each entry of which corresponds to the data associated with a spill. A spill can have three event types associated with it: an MC event (MCEvent) contains an array of data, each member of which represents the MC of a single particle traversing the experiment, a reconstructed event (ReconEvent) contains an array of data, each member of which corresponds to a particle event corresponding to a trigger, and a raw data event (DAQEvent), which corresponds to the raw data readout. There are 5 different detectors in MAUS: 1) time-of-flight (TOF) scintillators, 2) threshold Cherenkov (Ckov) counters, 3) scintillating fiber trackers, 4) "KL" sampling calorimeter, and 5) an electron-muon ranger (EMR). Each of these detectors have several layers of reconstruction which can be broken into individual reconstruction modules.

The MCEvent is subdivided into sensitive detector hits (energy deposited, position, momentum) and information about the parent particle that created the hits in the various detectors. The ReconEvent and DAQEvents are subdivided by detector and contain the raw and reconstructed data for each detector and the trigger.

The user-level scripts to run reconstruction and simulation jobs are written in Python and several examples and utilities are bundled with the software. An example of a driver to read from a DAQ file and output to a ROOT tree is shown below to illustrate the Input-Map-Reduce-Output functionality:

```python
#!/usr/bin/env python

# Reconstruct data from the MICE experiment

Offline analysis to produce reconstructed elements for MICE. TOF is reconstructed through to space points; Ckov is reconstructed through to Digits.

import MAUS
```
def run():
    """
    Analyze data from the MICE experiment
    """

    # Set up the input that reads from DAQ
    my_input = MAUS.InputCppDAQOfflineData()

    # Create an empty array of mappers, then populate it
    # with the functionality you want to use.
    my_map = MAUS.MapPyGroup()

    # Set up the MAUS data structure for reconstructed events
    my_map.append(MAUS.MapCppReconSetup())

    # Add reconstruction modules
    my_map.append(MAUS.MapCppTOFDigits())  # digitize PMT hits from each TOF detector
    my_map.append(MAUS.MapCppTOFSlabHits())  # reconstruct scintillator slab hits
    my_map.append(MAUS.MapCppTOFSpacePoints())  # reconstruct TOF spacepoints

    my_map.append(MAUS.MapCppCkovDigits())  # reconstruct Ckov digits and store light yields
    my_map.append(MAUS.MapCppKLDigits())  # construct emcal digits
    my_map.append(MAUS.MapCppKLCellHits())  # reconstruct emcal cell hits

    my_map.append(MAUS.MapCppTrackerDigits())  # construct scintillating fiber tracker digits
    my_map.append(MAUS.MapCppTrackerRecon())  # reconstruct tracks in each tracker

    my_map.append(MAUS.MapCppEMRPlaneHits())  # reconstruct electron-muon-ranger plane hits
    my_map.append(MAUS.MapCppEMRSpacePoints())  # reconstruct spacepoints
    my_map.append(MAUS.MapCppEMRRecon())  # reconstruct tracks

    # Add reducer functionality here
    my_reduce = MAUS.ReduceCppTOFPlot()  # TOF reducer to plot reconstructed hits and spacepoints

    # The Go() drives all the components you pass in then put all the output
    # Options are OutputCppRoot, OutputPyJSON, or OutputPyRootImage
    my_output = MAUS.OutputCppRoot()
    MAUS.Go(my_input, my_map, my_reduce, MAUS.OutputCppRoot())

    if __name__ == '__main__':
        run()

2.3. Testing

MAUS has a set of tests at the unit level and integration level. Unit tests are implemented
against a single function and check that each function operates as intended by the developer
and can achieve a high level of coverage and good test complexity. Integration tests operate
against a complete workflow and check that the overall design of the code meets the specifications
laid out, and interfaces with external codes or systems operate correctly. The MAUS team aims
to provide unit test coverage that executes 70–80 % of the total code base. This level of test
coverage typically results in a code that performs the major workflows without any problem.
MAUS operates a continuous integration stack on test servers, using Jenkins [10] to mimic the
offline deployment environment. Developers are asked to test their code on the test server before
proposing their branch be merged with the mainline trunk.

2.4. Release and deployment

Typically MAUS has worked on a 2-4 week release cycle but this has slowed down since MAUS is now in a more mature phase undergoing only incremental improvements. Developers are given notice before the development trunk is tagged as a release candidate at which stage it is built and tested on the central test server. Upon passing the tests, it is tagged with the final release version and released to the experiment. Each release version is deployed in the control room for use in the online reconstruction. A build of the release is also pushed to the GRID via CVMFS [11] for use in batch reconstruction and simulation. The current release version is 2.8.0.

3. Summary

3.1. Summary

The MICE experiment has developed an analysis and reconstruction software framework to serve both the particle physics and accelerator physics needs of the experiment. The architecture is inspired by the MapReduce paradigm and the software is written in Python and C++. The framework has been designed to aid ease in development and provides support for representing data in ROOT and JSON formats. Several industry-standard practices such as code coverage tests, and continuous integration testing have been adopted to ensure the quality and robustness of the software. While improvements continue, MAUS is in routine action reconstructing and simulating data for use in several analyses.

Acknowledgments

We thank all the developers, past and present, for their contributions to the development and improvement of MAUS.

References

[1] Geer S 2009 “Muon Colliders and Neutrino Factories,” Ann. Rev. Nucl. Part. Sci. 59 345–367.
[2] Bogomilov M et al. 2012 “The MICE Muon Beam on ISIS and the beam-line instrumentation of the Muon Ionisation Cooling Experiment” JINST 7 P05009, arXiv:1203.4089
[3] Adams D et al. “Characterisation of the muon beams for the Muon Ionisation Cooling Experiment” Eur. Phys. J. C73 102582, arXiv:1306.1509
[4] http://scons.org
[5] http://bazaar.canonical.com
[6] http://launchpad.net
[7] Rene Brun and Fons Rademakers 1997 Nucl. Inst. Meth. in Phys. Res. A 389 81-86, also http://root.cern.ch/
[8] Agostinelli S et al. 2003 “GEANT4 - A Simulation Toolkit” Nucl. Instrum. Meth. A 506 250-303
[9] http://json.org
[10] http://jenkins-ci.org
[11] Buncic P et al. 2010 J. Phys. Conf. Ser. 219 042003