The MAUS Work Book

Adam Dobbs

Version 1.7, August 2015

Contents

1 Glossary of Terms

- **ADC** - Analogue-to-Digital Converter
- **API** - Application Programme Interface
- **BASH** - Bourne Again Shell, a linux command line programme
- **Bazaar** - A distributed version control system
- **CKOV** - Threshold cherenkov detectors
- **Datacard** - A list of variables passed into MAUS (actually a Python script)
- **DAQ** - Data Aquisition system
- **EMR** - Electron Muon Ranger, a calorimeter
- **KL** - KLOE-Lite detector
- **GEANT4** - Framework to simulate passage of particles through matter
- **GRID** - A network of computing resources used for large scale data processing
- **JSON** - Jave Script Object Notation, an ASCII format
- **MAUS** - MICE Analysis User Software
- **MICE** - Muon Ionization Cooling Experiment
- **MC** - Monte Carlo simulation
- **PID** - Particle Identification
- **ROOT** - The standard high energy physics analysis framework, see `https://root.cern.ch`
- **Spill** - Data class in MAUS, corresponds to the data produced in MICE for one dip of the pion production target
- **TDC** - Time-to-Digital Converter
- **TOF** - Time of Flight, or the associated detectors
- **Tracker** - Scintillating fibre trackers
- **UI** - User Interface
- **Unpacker** - A third party library bundled with MAUS, used internally to extract data from the MAUS DAQ binary format

2 MAUS Description

MAUS stands for MICE Analysis User Software. MICE itself refers to the Muon Ionization Cooling experiment, see the website for more details:

`http://www.mice.iit.edu`.

MAUS is the official software framework for the simulation and reconstruction of the MICE data. It also provides an environment which can be used for developing subsequent analysis tools, and provides the libraries neccessary for using other tools to access processed MICE data.

This workbook constitutes a quick start guide for MAUS. It covers the following standard tasks:

- Performing simulations;
- Unpacking and reconstructing raw MICE data;
- Accessing and analysing simulated and reconstructed data.

A much fuller description of MAUS can be found in the User Guide, available at:

`http://micewww.pp.rl.ac.uk/maus/MAUS_latest_version/maus_user_guide/index.html`.

3 Installing MAUS

This section provides a summary of how to install MAUS on a Linux operating system. MAUS officially supports Scientific Linux 6 and recent versions of CentOS only, however unofficially MAUS should compile on many modern linux distributions, including Ubuntu. A more detailed guide to installing MAUS, including a list of prerequisites, can be found on the MAUS wiki:

`http://micewww.pp.rl.ac.uk/projects/maus/wiki/Install`

3.1 Obtaining the Code

MAUS may be obtained in one of two ways. First, MAUS releases are available for download as gzipped tarballs from:

`http://micewww.pp.rl.ac.uk/maus/`

Alternatively MAUS may be branched using the Bazaar distributed version control system (see `http://bazaar.canonical.com`) from the Launchpad code hosting website. The MAUS page on Launchpad is `https://launchpad.net/maus`. A detailed guide to using Bazaar with MAUS can be found on the MAUS wiki at `http://micewww.pp.rl.ac.uk/projects/maus/wiki/Bzr_usage` (see section D). In order to branch the latest release use:

```
bzr branch lp:maus
```

In order to obtain the latest development code (not for use if doing analyses) use:

```
bzr branch lp:maus/merge
```

3.2 Installation

Once the MAUS source code has been obtained, move into the source root directory. Inside this directory is a BASH script named `install_build_test.sh`. Execute this script to perform the full MAUS installation. On multicore machines use the flag "`-j N`" where `N` is the number of processor threads to use. For example, to install with two threads:

```
./install_build_test.sh -j 2
```

If another fully built MAUS installation of the same release is already present on the system, then the third party libraries from that installation can be linked against instead of building them from scratch (saving a lot of disk space and time during installation). This is done by specifying the location of the other MAUS installation with a "`-t`" flag, for example:

```
./install_build_test.sh -t /path/to/other/maus
```

3.3 Preparing to run

Prior to running, the MAUS envionment variables must be loaded into the local shell environment each time a new shell is used. This is done by sourcing the script "env.sh" found in the MAUS root directory:

```
source env.sh
```

MAUS is now ready to use!

3.4 Rebuilding

If any code within MAUS is modified, it will need to be rebuilt for the changes to take effect. MAUS uses the *scons* build system (a Make replacement based on Python). To rebuild MAUS, simply type `scons` from the MAUS root directory (after sourcing the environment). A flag `-jN` may also be supplied to run on multiple threads. In order to clean the build before rebuilding, the command `scons -c` may be used.

3.5 Choosing the Unpacker

The unpacker is a third party library which ships with MAUS, and is used internally for extracting the raw binary data produced by the MICE DAQ. There are two versions of the unpacker present in MAUS, one corresponding to MICE Step I data, the other to MICE Step IV and later data. From MAUS release 0.9.5 the default unpacker is set to be Step IV.

To switch between unpackers, a simple utility script, `switch_unpacker.bash`, is provided in the MAUS root directory. To switch unpackers, source the MAUS environment and then source (not run) this script, supplying an argument of either of StepI or StepIV. This will edit `env.sh`, change the `MICE_ UNPACKER_VERSION` environment variable, rebuild the unpacker, and clean and rebuild MAUS.

NB: The `switch_unpacker.bash` script only works for full MAUS installations; if the MAUS installation links against third parties from another installation, the script will abort. In this case, change the unpacker version in the full MAUS installation first using the script, then in the linked MAUS installation edit env.sh so that `MICE_ UNPACKER_VERSION` is set to the correct value, and clean and rebuild MAUS manually.

4 An Introduction to the MAUS UI and API

4.1 The API

MAUS is written in both C++ and Python, with Python primarily in the code presented to the user, and C++ primarily in the backend to do the hard work. MAUS is split up into different modules which can be chained together to perform various tasks. Data is passed between each module via an instance of a class known simply as *Data*. Within the Data object the rest of the MAUS data structure is nested (see section 5).

There are four types of module present in MAUS:

- **Input** - modules which read in data, such as from the DAQ or a ROOT file
- **Map** - modules which perform most of the work of simulation and reconstruction
- **Reduce** - modules which accumulate data in order to produce plots
- **Output** - writes out processed data to e.g. a ROOT or JSON file

Modules are chained together using a Python script to create a functional MAUS programme. This is illustrated in figure 1.

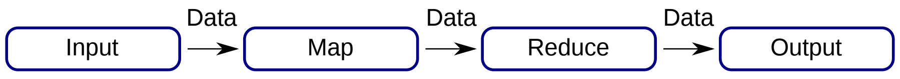


Figure 1: The MAUS Map - Reduce scheme.

4.2 Creating a MAUS programme

All the core MAUS modules and classes may be imported into Python ready for use by using the command:

```
import MAUS
```

Each desired input, map, reduce and output module must then be declared. For example, to declare an input module used to setup the spill structure ready for a simulation:

```
my_input = MAUS.InputPySpillGenerator()
```

There must only be one input and output module used. However, multiple maps and reducers may be used by grouping them together using the special modules `MapPyGroup` and `ReducePyGroup`. For example, to group together the maps needed for a GEANT4 beam simulation:

```
my_map.append(MAUS.MapPyBeamMaker())   # Beam construction
my_map.append(MAUS.MapCppSimulation()) # GEANT4 simulation
```

Also, if no maps or reducers are required, the "do nothing" modules, `MAUS.MapPyDoNothing()` and `MAUS.ReducePyDoNothing()` must be used.

After declaring all the required MAUS modules, a special class known as `datacards` for handling the passing of parameters must be declared. Datacards are collections of parameters used by MAUS, and are discussed in section 4.4. To use only parameters passed from the command line or via a datacard, with the rest set to their default values, the following declaration is used:

```
datacards = io.StringIO(u"")
```

Finally, all the modules and datacards are passed to a special class called "Go", which runs each module in turn and passes the data from one to another:

```
MAUS.Go(my_input, my_map, my_reduce, my_output, datacards)
```

4.3 A Minimal Working Example

A minimal working example which performs a GEANT4 beam simulation, reconstructs the Time-of-Flight (TOF) detector data and writes the output to a ROOT file is shown below:

```
import io   # Generic Python library for I/O
import MAUS # MAUS libraries

def run():
  my_input = MAUS.InputPySpillGenerator()
  my_map = MAUS.MapPyGroup()
  my_map.append(MAUS.MapPyBeamMaker())    # Beam construction
  my_map.append(MAUS.MapCppSimulation()) # GEANT4 simulation
  my_map.append(MAUS.MapPyMCReconSetup()) # Detector setup
  my_map.append(MAUS.MapCppTOFMCDigitizer()) # TOF MC Digitizer
  my_map.append(MAUS.MapCppTOFSlabHits())    # TOF Slab Hits
  my_map.append(MAUS.MapCppTOFSpacePoints()) # TOF Spacepoints
  my_reduce = MAUS.ReducePyDoNothing()
  my_output = MAUS.OutputCppRoot()
  datacards = io.StringIO(u"")
  MAUS.Go(my_input, my_map, MAUS.ReducePyDoNothing() \
          my_output, datacards)

if __name__ == '__main__':
    run()
```

4.4 Running a MAUS Programme and Using Datacards

Once the Python script has been created it can be executed using the Python interpreter (remembering to `source env.sh` first). For example to run a script called "my_programme.py" use:

```
python my_programme.py
```

It is also important to consider what parameter values the programme requires. MAUS uses a large set of parameters to configure how it runs, covering areas such simulation beam parameters, input and output file names, which geometry files to use, etc. The default values for all these parameters are set in the file `src/common_py/ConfigurationDefaults.py`. Parameters may be modified from their default values in one of three ways.

1. By passing arguments to the executable Python script. The required syntax is `--variable-name value`. For example to change the name of the output ROOT file from its default value of "maus_output.root" to "my_output.root" when running a programme defined by the Python script `my_programme.py`, the following command could be used:

   ```
   python my_programme.py --output_root_file_name my_output.root
   ```

2. With a datacards file. A datacards file is simply a list of parameters and values, one per line, of the form:

   ```
   variable=value
   ```

   A datacard may be passed to an executable by using the `--configuration_file` flag. For example, to run a script called "my_programme.py" together with a datacard called "datacard.py":

   ```
   python my_programme.py --configuration_file datacard.py
   ```

3. From within the executable Python script. By modifying the line

   ```
   datacards = io.StringIO(u"")
   ```

   parameters may be set by assigning them a value between the speech marks.

Generally, if only one or two parameters require setting, method 1 is easiest, otherwise use method 2.

NB: Only use one method at a time, or some of the settings may be ignored! Some parameters and their default values are listed in appendix B.

5 The Datastructure

MAUS uses a large and complex data structure to store all the MC simulation, raw detector and reconstruction data. A diagram illustrating the data structure can be found at:

`http://micewww.pp.rl.ac.uk/maus/MAUS_latest_version/maus_user_guide/node10.html`

Perhaps more helpfully, the data structure is extensively documented using the Doxygen documentation system, available at:

`http://micewww.pp.rl.ac.uk/maus/MAUS_latest_version/doc/index.html`

The top level object in the data structure is a class called simply *Data*. This contains run and job header and footer information (such as the version of MAUS used, the date of the run, etc) together with a single *Spill* object. A Spill represents all the experimental data associated with one dip of the MICE pion production target; all MC, raw detector and reconstruction data is contained within it.

The Spill is split into three main branches:

- *MCEvent* holds the MC truth data for every track together with all MC detector and virtual hits produced;
- *DAQData* holds the raw, unpacked data for each detector (such as ADC and TDC values)
- and *ReconEvent* holds the reconstructed data for each detector.

Examples showing how to access and navigate the data structure for analysis are given in section 9.

6 Geometry

MAUS uses a sophisticated GDML-based geometry for use in both simulation and reconstruction (by default the same geometry is used for both). The canonical geometry is stored in the MICE configuration database (CDB), while an older legacy geometry is stored locally in MAUS (currently still the default option).

The CDB is a bitemporal database, that is it stores geometry configurations based on upload date and date of validity. A particular CDB geometry must be downloaded to the local MAUS installation before it may be used by a programme. This can be done using a utility script, `download_geometry.py`, located in `bin/utilities`. The script queries the CDB and then downloads the required geometry. Geometries can be selected for download by a number of methods:

1. Retrieve current geometry:

   ```
   ./bin/utilities/download_geometry.py
      --geometry_download_by current
   ```

2. By geometry ID e.g.

   ```
   ./bin/utilities/download_geometry.py --geometry_download_id 50
   ```

3. By run number e.g.

   ```
   ./bin/utilities/download_geometry.py
      --geometry_download_by run_number
      --geometry_download_run_number 6008
   ```

By default, CDB geometries are stored in `files/geometry/download`. A simulation or reconstruction script can be told to use a particular geometry by using the flag `--simulation_geometry_file` followed by the absolute path and the top level geometry file name, `ParentGeometryFile.dat`. For simulation usage examples see section 7, and for reconstruction usage examples see section 8.

7 Running a MC Simulation

An example script for running a full MC simulation of MICE can be found in the bin directory:

```
./bin/simulate_mice.py
```

This will produce a ROOT output file which can be used for analysis.

It is generally desirable to customise the simulation in some way by passing datacard variables. For example to specify a particular geometry to use from the CDB, first download the desired geometry as described in section 6. A new reference simulation particle must also then be defined (due to differences between the local legacy geometry and the CDB geometries) via a datacard parameter. This can be defined in a datacard file as:

```
simulation_reference_particle = {"position":{"x":0.0, "y":0.0, "z"
    :1000.0}, "momentum":{"x":0.0, "y":0.0, "z":1.0}, "particle_id"
    :-13, "energy":226.0, "time":0.0, "random_seed":10, "spin":{"x"
    :0.0, "y":0.0, "z":1.0}}
```

This gives a reference particle starting at coordinates (0, 0, 1000) (in mm), $p_z = 1$ MeV/c, species μ^+ with energy 226 MeV. If the above line is stored in a file "datacard.py", the new simulation can be run with:

```
./bin/simulate_mice.py --simulation_geometry_file /path/to/maus/
    files/geometry/download/ParentGeometryFile.dat --
    configuration_file datacard.py
```

For details on the rest of the options available when performing simulations, see the full MAUS user guide.

8 Reconstructing Real MICE Data

8.1 Do I Need to be Reconstructing Data Myself?

Before reconstructing raw MICE data, it is important to stop and consider whether this is the best workflow for the task in hand. All MICE data should already be available unpacked and reprocessed from one of the standard locations listed in 9. This reprocessing will have been done using an official release version of MAUS with a standard set of parameters - hence **anyone thinking of analysing data for publication should use the official processed data instead of processing themselves**. If this does not dissuade you, then read on.

8.2 Obtaining the Raw Data

MICE data is recorded by MICE DAQ as custom format binary files. This data is available for download from a web interface hosted at:

`http://www.hep.ph.ic.ac.uk/micedata/`

For those with GRID credentials who are members of the MICE virtual organisation, the data may also be obtained from the GRID at LFC location `/grid/mice/MICE/`.

The data is distributed as tar archives. Once extracted the binary data files can be identified by their filenames, which follow the format `<run_number>.nnn` where `nnn` is a number corresponding to the data block of the run respresented by the file, beginning at 000.

8.3 Unpacking and Reconstructing

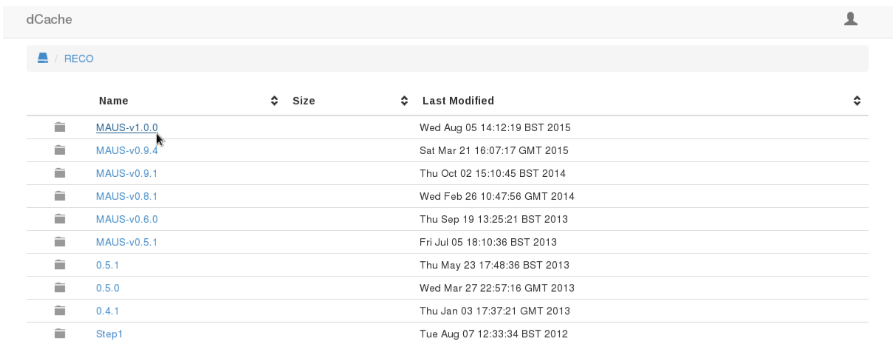
An example script which unpacks the binary data and reconstructs data from all the MICE dectors can be found in the bin directory. Parameters indicating the location and name of the binary data file must be passed in, together with the appropriate geometry for the run being analysed (the desired geometry must be downloaded first, as described in section 6). For example:

```
python ./bin/analyze_data_online.py \
         --daq_data_path /path/to/data/ \
         --daq_data_file file.name \
         --simulation_geometry_file /path/to/maus/files/geometry/
            download/ParentGeometryFile.dat
```

(Note we specify the simulation geometry file, as the reconstruction geometry will default to the same value.) This will produce a ROOT file containing all the unpacked raw data and the reconstrucuted data, ready for analysis.

9 Analysing Output Data

9.1 Obtaining Data for Analysis



dCache

RECO

	Name	Size	Last Modified
	MAUS-v1.0.0		Wed Aug 05 14:12:19 BST 2015
	MAUS-v0.9.4		Sat Mar 21 16:07:17 GMT 2015
	MAUS-v0.9.1		Thu Oct 02 15:10:45 BST 2014
	MAUS-v0.8.1		Wed Feb 26 10:47:56 GMT 2014
	MAUS-v0.6.0		Thu Sep 19 13:25:21 BST 2013
	MAUS-v0.5.1		Fri Jul 05 18:10:36 BST 2013
	0.5.1		Thu May 23 17:48:36 BST 2013
	0.5.0		Wed Mar 27 22:57:16 GMT 2013
	0.4.1		Thu Jan 03 17:37:21 GMT 2013
	Step1		Tue Aug 07 12:33:34 BST 2012

Figure 2: The MICE data webstore, reconstruction section.

Data from simulation may be produced by MAUS as described in section 7. For real data, as mentioned in section 8, the preferred route is for analysts to use MICE data which has been unpacked and reconstructed centrally on the GRID. This ensures everyone is using the same standard reconstruction parameters from an official release of MAUS, and is especially important if the analysed data is intended for publication. If it is still desirable to reconstruct data locally with MAUS prior to analysis, see section 8. Otherwise, centrally processed MICE data is made available from the MICE webstore:

`http://www.hep.ph.ic.ac.uk/micedata/RECO/`

The front page of the reconstruction section of the webstore is shown in figure 2. The data in the webstore is organised by MAUS version used to perform the reconstruction, by parameter set used in the reconstruction (assigned a unique number known as the batch iteration number or BIN)

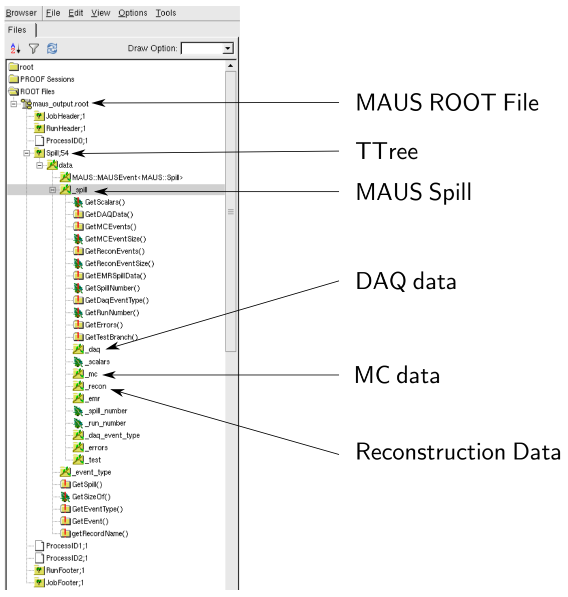


Figure 3: The top few levels of a MAUS ROOT file, as shown by a ROOT TBrowser.

and by run number. The data for each run is then stored in a tarball file archive which may be downloaded. Inside the tar archive there are a number of log files and parameter files. The data itself is stored in a ROOT file following the naming convention `xxxxx_recon.root`, where `xxxxx` is the run number.

A catalogue of MICE runs, including run numbers, run conditions, beamline optics, number of triggers taken, etc. is available via the MICE CDB viewer:

`http://cdb.mice.rl.ac.uk/cdbviewer/`

9.2 Understanding the ROOT file

By default, simulated and reconstructed data produced by MAUS is stored in a ROOT file, and data processed on the GRID is also stored in this format. The top level object in a MAUS ROOT file is a ROOT TTree called "Spill". This TTree contains a branch holding a MAUS data structure Data object, which in turn contains a MAUS data structure Spill object (not to be confused with the TTree called Spill). All the raw detector, MC and reconstruction data is found within this Spill object. The top few levels of a MAUS ROOT file, displayed in a TBrowser, are shown in figure 3.

There are three standard ways of accessing the data stored in the ROOT file: via ROOT directly (either manually through the ROOT interpretter or with an interpretted ROOT script); via Python using PyROOT (again either through the Python interpretter directly or with a Python script); or with a C++ programme using the ROOT libraries. All three methods require the MAUS data structure library to be loaded to interpret the data (described in the sections below).

Both the ROOT and Python methods allow the data inside the TTree to be accessed either directly from the TTree (using "`tree->Draw(...)`"), or loaded back into data structure classes in memory.

C++ can only use the latter. Direct plotting from a the TTree allows for quick and easy browsing of data, whereas loading data back into data structure classes allows greater versatility in analysis.

It is important to note that *ROOT allows direct plotting from a TTree only down to a few levels of nested data* - hence data nested too deeply into the MAUS data structure cannot be plotted directly from the Spill TTree, either by the ROOT or Python interpretter. In order to plot such data, the data from each spill must be loaded back from the TTree into data structure objects in memory.

Interactive plotting directly from TTrees is discussed for ROOT in section 9.3.3 and for Python in section 9.4.2. Loading data back into data structure classes is discussed for ROOT in section 9.3.4, for Python in section 9.4.3 and for C++ in section 9.5.

9.3 Analysis with the ROOT interpreter

9.3.1 Loading Data

Source the MAUS environment and load an interactive ROOT session. From the ROOT command line load the MAUS data structure library:

```
.L $MAUS_ROOT_DIR/build/libMausCpp.so
```

Load the ROOT output file produced by MAUS:

```
TFile f1("maus_output.root")
```

9.3.2 ROOT Scripts

Commands may be automated with a ROOT script, which is simply a collection of ROOT commands contained between an opening { and closing } in a text file. An important difference to note however is that loading the MAUS data structure library follows a slightly different format when in a script compared to using the ROOT interpreter directly. The library must now be loaded by calling gSystem->Load:

```
maus_root_dir = TString(gSystem->Getenv("MAUS_ROOT_DIR"));
gSystem->Load(maus_root_dir + "/build/libMausCpp.so");
```

9.3.3 Plotting directly from a TTree

Once the data structure and MAUS root file have been loaded, data can be explored using a GUI via the ROOT TBrowser:

```
TBrowser b
```

A example of using the TBrowser to navigate the data structure and plot a histogram is shown in figure 4.

The ROOT command line may also be used to plot data interactively. For example, the histogram shown in figure 4 could be drawn directly using the command:

```
Spill.Draw("data._spill._recon->_scifi_event->_scifispacepoints.
    _npe")
```

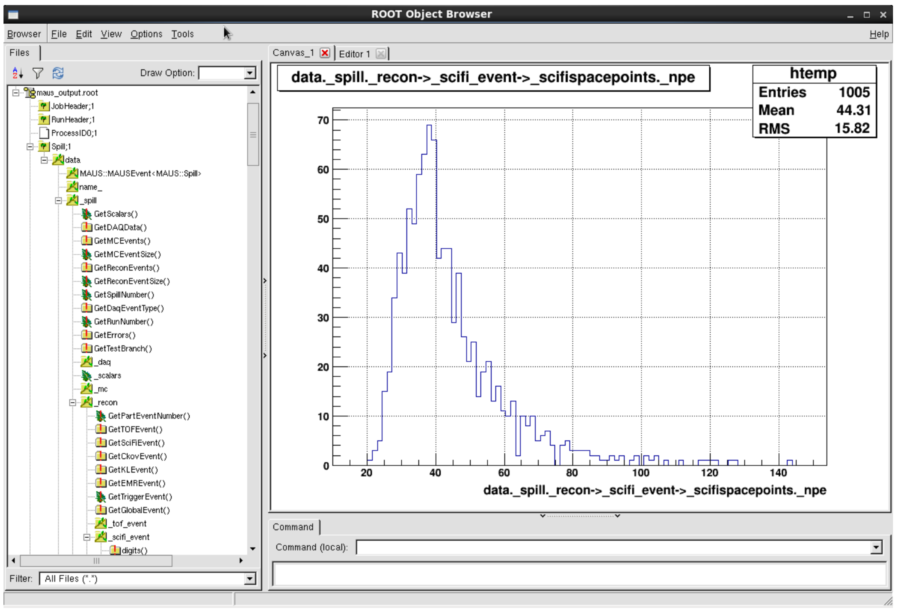

ROOT Object Browser
Browser File Edit View Options Tools Help
Canvas_1
Editor 1
data._spill._recon->_scifi_event->_scifispacepoints._npe
htemp
Entries 1005
Mean 44.31
RMS 15.82
data._spill._recon->_scifi_event->_scifispacepoints._npe

Figure 4: A ROOT TBrowser plotting the number of photoelectrons in every SciFi SpacePoint for a short MC run.

9.3.4 Reloading the Datastructure

To load data back into MAUS data structure classes from the TTree a MAUS Data object pointer must be assigned to the memory address of the MAUS Data object held by the TTree. All the entries in the TTree can then be looped over which will update the pointer to the next spill's Data object each time. A short ROOT script example which reproduces the histogram in figure 4 is shown below, and is also available to download from `http://micewww.pp.rl.ac.uk/documents/140` as "root_analysis_example.C".

```
{
  // Load the MAUS data structure
  maus_root_dir = TString(gSystem->Getenv("MAUS_ROOT_DIR"));
  gSystem->Load(maus_root_dir + "/build/libMausCpp.so");

  // Set up the ROOT file and data pointer
  TFile f1("maus_output.root"); // Load the MAUS output file
  TTree *T = f1.Get("Spill");    // Pull out the TTree
  MAUS::Data *data_ptr; // Variable to store Data from each spill
  // Set address of data_ptr to Data object in TTree
  T->SetBranchAddress("data", &data_ptr);

  // Create the histogram
  TH1D* h1 = new TH1D("npe", "SciFi Spacepoint NPE", 100, 0, 150);
  h1->GetXaxis()->SetTitle("Number of photoelectrons");
  h1->SetLineColor(kBlue);
  // Loop over all spills
  for (size_t i = 0; i < T->GetEntries(); ++i) {
    T->GetEntry(i);  // Update the spill pointed to by data_ptr
    MAUS::Spill* spill = data_ptr->GetSpill();  // Get the spill
    if (spill == NULL ||
        !(spill->GetDaqEventType() == "physics_event")) continue;
    std::cout << "Spill: " << spill->GetSpillNumber() << "\n";
    std::vector<MAUS::ReconEvent*>* revts = spill->GetReconEvents
       ();
    // Loop over recon events in spill
    for ( size_t j = 0; j < revts->size(); ++j ) {
      if ( !revts->at(j) ) continue; // Check event pointer
      MAUS::SciFiEvent* sfevt = revts->at(j)->GetSciFiEvent();
      std::vector<SciFiSpacePoint*> spnts = sfevt->spacepoints();
      // Loop over spacepoints
      for ( size_t k = 0; k < spnts.size(); ++k ) {
        h1->Fill(spnts[k]->get_npe()); // Fill histo with npe
      }
    }
  }
  // Draw the histogram
  TCanvas * c1 = new TCanvas("c1", "SF Spacepoint NPE", 900, 600);
  h1->Draw();
}
```

9.4 Analysis with Python

9.4.1 Loading Data

Source the MAUS environment and load an interactive Python session. From the Python interpreter load ROOT and the MAUS data structure library:

```
>>> import libMausCpp
>>> from ROOT import *
```

MAUS data structure classes are now accessible. For example, to create a Spill object:

```
>>> spill = MAUS.Spill()
```

NB: Remember to add the trailing "()" or the class contructor will not be called.

A MAUS output file may now be loaded:

```
>>> f1 = TFile("maus_output.root")
```

9.4.2 Plotting directly from a TTree

Once ROOT and the MAUS data structure are loaded, all the tools available in ROOT are available from Python. For example, to load a TBrowser use:

```
>>> b = TBrowser()
```

To reproduce the histogram in simply use

```
>>> Spill.Draw("data._spill._recon->_scifi_event->
    _scifispacepoints._npe")
```

9.4.3 Reloading the Datastructure

The same principles used in the ROOT case apply to reloading data in to the MAUS data structure classes from the TTree. An example script which reproduces the histogram in figure 4 is shown below, and is also available to download from `http://micewww.pp.rl.ac.uk/documents/140` as "python_analysis_example.py".

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

import libMausCpp
from ROOT import *

# Set up the ROOT file and data pointer
f1 = TFile("maus_output.root")
t1 = f1.Get("Spill")
data_ptr = MAUS.Data()
t1.SetBranchAddress("data", data_ptr)

# Create the histogram
h1 = TH1D("h1", "Spacepoint NPE", 100, 0, 150);
h1.GetXaxis().SetTitle("Number of photoelectrons")
h1.SetLineColor(kBlue)

# Loop over all spills
for i in range(1, t1.GetEntries()):
    t1.GetEntry(i) # Update the spill pointed to by data_ptr
    spill = data_ptr.GetSpill()
    if spill.GetDaqEventType() == "physics_event":
        # Loop over recon events in spill
        for j in range(spill.GetReconEvents().size()):
            sfevt = spill.GetReconEvents()[j].GetSciFiEvent()
            # Loop over spacepoints
            for k in range(sfevt.spacepoints().size()):
                spoint = sfevt.spacepoints()[k]
                h1.Fill(spoint.get_npe())

# Draw the histogram
c1 = TCanvas("c1", "SF Spacepoint NPE", 900, 600)
h1.Draw()
raw_input("Press Enter to finish...")
```

9.4.4 ROOT alternatives

Once the MAUS data structure has been repopulate from the TTree as described above, alternative tools may be used to produce the desired histograms, plots, etc. MAUS comes bundled with the popular Python *numpy* and *matplotlib* packages, which are ready to use in the Python interpreter.

9.5 Analysis with C++

The fastest method for large scale data analysis is to compile a C++ programme which calls on the ROOT and MAUS data structure headers and libraries. The programme structure is very close to that of the ROOT script described in section 9.3.4. An important difference to note is that instead of loading the MAUS output ROOT file directly, a custom C++ streamer class, *irstream* may be used, which serialises the TFile data ready for use.

An example source file which recreates the histogram in figure 3 is produced below, and is also available to download from `http://micewww.pp.rl.ac.uk/documents/140` as "cpp_analysis_example.cc". It produces an executable which takes the MAUS output ROOT file an argument and will display the histogram to screen. NB: In order to display the histogram while the application is running, the ROOT class "TApplication" must be used. If the purpose of the application was just to save the output to, say, a graphics file, this would not be needed.

```
#include <iostream>
#include <string>
#include <vector>

#include "TCanvas.h"
#include "TH1D.h"
#include "TApplication.h"

#include "src/common_cpp/JsonCppStreamer/IRStream.hh"
#include "src/common_cpp/DataStructure/Spill.hh"
#include "src/common_cpp/DataStructure/Data.hh"
#include "src/common_cpp/DataStructure/ReconEvent.hh"
#include "src/common_cpp/DataStructure/SciFiEvent.hh"
#include "src/common_cpp/DataStructure/SciFiSpacePoint.hh"

/** Access Tracker data using ROOT */

int main(int argc, char *argv[]) {
  // First argument to code should be the input ROOT file name
  std::string filename = std::string(argv[1]);

  // Set up ROOT app, input file, and MAUS data class
  TApplication theApp("App", &argc, argv);
  std::cout << "Opening file " << filename << std::endl;
  irstream infile(filename.c_str(), "Spill");
  MAUS::Data data;
  // Create the histogram
  TH1D* h1 = new TH1D("npe", "SciFi Spacepoint NPE", 100, 0, 150);
  h1->GetXaxis()->SetTitle("Number of photoelectrons");
  h1->SetLineColor(kBlue);

  // Loop over all spills
  while ( infile >> readEvent != NULL ) {
    infile >> branchName("data") >> data;
    MAUS::Spill* spill = data.GetSpill();
    if (spill == NULL || spill->GetDaqEventType() != "
       physics_event")
      continue;
    std::cout << "Spill: " << spill->GetSpillNumber() << "\n";
    std::vector<MAUS::ReconEvent*>* revts = spill->GetReconEvents
       ();

    // Loop over recon events in spill
    for ( size_t i = 0; i < revts->size(); ++i ) {
      if ( !revts->at(i) ) continue;
      MAUS::SciFiEvent* sfevt = revts->at(i)->GetSciFiEvent();

      // Loop over spacepoints
      std::vector<MAUS::SciFiSpacePoint*> spnts = sfevt->
         spacepoints();
      std::vector<MAUS::SciFiSpacePoint*>::iterator spnt;
      for ( spnt = spnts.begin(); spnt != spnts.end(); ++spnt ) {
        h1->Fill((*spnt)->get_npe());
      }
    } // ~Loop over Recon events
  } // ~Loop over all spills

  // Draw the histogram
  TCanvas * c1 = new TCanvas("c1", "SF Spacepoint NPE", 900, 600);
  c1->cd();
  h1->Draw();
  theApp.Run();
}
```

Due to the considerable number of dependencies, it is easier to use a build system to compile this, rather than running the compiler directly. Below is an example Makefile which can be used to compile and link the above the example (assuming a source file of “cpp_analysis_example.cc”.). It is also available to download from `http://micewww.pp.rl.ac.uk/documents/140` as “Makefile”.

```
CC = g++
CFLAGS = -DDEBUG -ggdb -Wall

all : cpp_analysis_example

cpp_analysis_example : cpp_analysis_example.cc
  $(CC) $(CFLAGS) cpp_analysis_example.cc -o cpp_analysis_example
     \
                        -I${MAUS_ROOT_DIR}/ \
                        -I${MAUS_ROOT_DIR}/src/common_cpp \
                        -I${MAUS_ROOT_DIR}/src/legacy \
                        -I${MAUS_THIRD_PARTY}/third_party/install/
                           include \
                        -I${ROOTSYS}/include \
                        -L${MAUS_ROOT_DIR}/build/ \
                        `root-config --ldflags` \
                        `${ROOTSYS}/bin/root-config --glibs` \
                        -lMausCpp \
                         -Wl,--no-as-needed

clean:
  rm cpp_analysis_example
```

A Module Descriptions

A.1 Input

Module	Description
InputCppDAQOfflineData	Unpack real data offline
InputCppDAQOnlineData	Used for online reconstruction
InputCppRoot	Read in data from a ROOT file
InputPyJSON	Read in data from a JSON file
InputPySpillGenerator	Make a spill structure for simulation

A.2 Output

Module	Description
OutputCppRoot	Output data to a ROOT file
OutputPyImage	Output data as an image file
OutputPyJSON	Output data as a JSON file

A.3 Map

Module	**Precursor**	**Description**
MapCppEMRMCDigitization	MapPyMCReconSetup	Digitise EMR MC data
MapCppEMRPlaneHits	MapPyMCReconSetup	Create EMR bar hits
MapCppEMRRecon	MapCppEMRPlaneHits	Reconstruct the EMR data
MapCppGlobalPID	MapCppGlobalTrackMatching	Perform PID using Global data
MapCppGlobalReconImport	Detector maps	Import detector info to Global tracking
MapCppGlobalTrackMatching	MapCppGlobalReconImport	Reconstruct Global tracks
MapCppKLCellHits	MapCppKLDigits OR	Construct hits in the KL
	MapCppKLMCDigitizer	
MapCppKLDigits	MapPyReconSetup	Produce KL digits from real data
MapCppKLMCDigitizer	MapPyMCReconSetup	Produce KL digits from MC data
MapCppSimulation	MapPyBeamMaker OR	GEANT4 sim of beam through MICE
	MapPyBeamlineSimulation	
MapCppTOFDigits	MapPyReconSetup	Produce TOF digits from real data
MapCppTOFMCDigitizer	MapPyMCReconSetup	Produce TOF digits from MC data
MapCppTOFSlabHits	MapCppTOFDigits OR	Reconstruct TOF slab hits
	MapCppTOFMCDigitizer	
MapCppTOFSpacePoints	MapCppTOFSlabHits	Reconstruct TOF spacepoints
MapCppTrackerDigits	MapPyReconSetup	Produce Tracker digits from real data
MapCppTrackerMCDigitization	MapPyMCReconSetup	Produce Tracker digits from MC data
MapCppTrackerMCNoise	MapCppSimulation	Add noise to the Tracker simulation
MapCppTrackerRecon	MapCppTrackerDigits OR	Reconstruct the Tracker data
	MapCppTrackerMCDigitization	
MapPyBeamlineSimulation	InputPySpillGenerator	Generate beam with G4BeamLine
MapPyBeamMaker	InputPySpillGenerator	Generate beam with GEANT4
MapPyCkov	MapPyMCReconSetup	Reconstruct the CKOV data
MapPyDoNothing	None	No maps
MapPyGroup	None	Used to add multiple maps
MapPyMCReconSetup	MapCppSimulation	Setup for MC reconstrucuted data
MapPyReconSetup	InputCppDAQOfflineData OR	Setup for real reconstrucuted data
	InputCppDAQOnlineData	
MapPyScalersDump	InputCppDAQOfflineData OR	Print scaler data
	InputCppDAQOnlineData	

A.4 Reduce

Module	Description
ReduceCppPatternRecognition	Display tracker pattern recognition tracks
ReducePyCkovPlot	Display CKOV data
ReducePyDoNothing	No reducer
ReducePyKLPlot	Display data
ReducePyTOFPlot	Display TOF data

B Common Parameters and their Defaults

All the MAUS parameters and their defaults values can be found in:

`src/common_py/ConfigurationDefaults.py`

A table of some common parameters is produced below.

Parameter	Default	Description
daq_data_file	'05466.001'	The real data file
daq_data_path	'src/input/InputCppDAQData'	Location of real data
input_root_file_name	'maus_input.root'	Input ROOT file name
geometry_download_directory	"files/geometry/download"	CDB geometry download directory
output_root_file_name	'maus_output.root'	Output ROOT file name
keep_only_muon_tracks	False	Simulate muon tracks
particle_decay	True	Should particles decay
physics_model	"QGSP_BERT"	Physics package
physics_processes	"standard"	Physics processes
reconstruction_geometry_filename	""	The simulation geometry
		(blank $\Rightarrow$ use simulation geometry)
reference_physics_processes	"mean_energy_loss"	Physics processes of ref. part.
simuation_geometry_file	"Test.dat"	The simulation geometry
spill_generator_number_of_spills	100	# of spills to simulate
verbose_level	1	Amount of screen output

C Navigating the Source Code

Description	Location
Executables	bin, bin/user
Documentation	doc
Input modules	src/input
Map modules	src/map
Reduce modules	src/reduce
Output modules	src/output
C++ backend code	src/common_cpp
Python backend code	src/common_py
Datastructure	src/common_cpp/Datastructure
Legacy Geometry	src/legacy/FILES/Models
Tests	tests
Third party libraries	third_parties

D Useful Information

- The MICE website: `http://mice.iit.edu`
- The MAUS wiki: `http://micewww.pp.rl.ac.uk/projects/maus/wiki`
- Download MAUS: `http://micewww.pp.rl.ac.uk/maus/`
- The MICE datastore: `http://www.hep.ph.ic.ac.uk/micedata/`
- The MICE CDB viewer: `http://cdb.mice.rl.ac.uk/cdbviewer/`
- Example analysis code: `http://micewww.pp.rl.ac.uk/documents/140`