MAUS Analysis User System
User Guide

Contents

Chapter 1

What Who and How?

MAUS (MICE Analysis User Software) is the MICE project's tracking, detector reconstruction and accelerator physics analysis framework. MAUS is designed to fulfil a number of functions for physicists interested in studying MICE data:

- Model the behaviour of particles traversing MICE
- Model the MICE detector's electronics response to particles
- Perform pattern recognition to reconstruct particle trajectories from electronics output
- Provide a framework for high level accelerator physics analysis
- Provide online diagnostics during running of MICE

In addition to MAUS's role within MICE, the code is also used for generic accelerator development, in particular for the Neutrino Factory.

1.1 Who Should Use MAUS

MAUS is intended to be used by physicists interested in studying the MICE data. MAUS is designed to function as a general tool for modelling particle accelerators and associated detector systems. The modular system, described in the API section, makes MAUS suitable for use by any accelerator or detector group wishing to perform simulation or reconstruction work.

1.2 Getting the Code and Installing MAUS

Installation is described in a separate document, available at `http://micewww.pp.rl.ac.uk/projects/maus/wiki/Install`

1.3 Running MAUS

MAUS contains several applications to perform various tasks. Two main applications are provided. `bin/simulate_mice.py` makes a Monte Carlo simulation of the experiment and `bin/analyze_data_offline.py` reconstructs an existing data file. Start a clean shell and move into the top level MAUS directory. Then type

```
> source env.sh
> ${MAUS_ROOT_DIR}/bin/simulate_mice.py
> ${MAUS_ROOT_DIR}/bin/analyze_data_offline.py
```

1.3.1 Run Control

The routines can be controlled by a number of settings that enable users to specify run configurations, as specified in this document. Most control variables can be controlled directly from the comamnd line, for example doing

```
> ${MAUS_ROOT_DIR}/bin/simulate_mice.py \
                    --simulation_geometry_filename Test.dat
```

to run the Monte Carlo against a given geometry. As another example, it is possible to run the data reconstruction against a given run

```
> cd ${MAUS_ROOT_DIR}
> ${MAUS_ROOT_DIR}/bin/analyze_data_offline.py \
                    --daq_data_file 02873 \
                    --daq_data_path src/input/InputCppDAQData
```

This will run against data in run `02873` looking for files in directory `src/input/InputCppDAQData`.

To get a (long) list of all command line variables use the `-h` switch.

```
> ${MAUS_ROOT_DIR}/bin/simulate_mice.py -h
```

More complex control variables can be controlled using a configuration file, which contains a list of configuration options.

```
> ${MAUS_ROOT_DIR}/bin/simulate_mice.py --configuration_file config.py
```

where a sample configuration file for the example above might look like

```
simulation_geometry_filename = "Test.dat"
```

Note that where on the command line a tag like `--variable value` was used, in the configuration file `variable = "value"` is used. In fact the configuration file is a python script. When loaded, MAUS looks for variables in it's variable list and loads them in as configuration options. Other variables are ignored. This gives users the full power of a scripting language while setting up run configurations. For example, one might choose to use a different filename,

```
import os
simulation_geometry_filename = os.path.join(
                          os.environ["MICEFILES"]
                          "Models/Configurations/Test.dat"
)
```

This configuration will then load the file at `$MICEFILES/Models/Configurations/Test.dat`

The default configuration file can be found at `src/common_py/ConfigurationDefaults.py` which contains a list of all possible configuration variables and is loaded by default by MAUS. Any variables not specified by the user are taken from the configuration defaults.

1.3.2 Other Applications

There are several other applications in the `bin` directory and associated subdirectories.

- `bin/examples` contains example scripts for accessing a number of useful features of the API
- `bin/utilities` contains utility functions that perform a number of useful utilities to do with data manipulation, etc
- `bin/user` contains analysis functions that our users have found useful, but are not necessarily thoroughly tested or documented
- `bin/publications` contains analysis code used for writing a particular (MICE) publication

1.4 Accessing Data

By default, MAUS writes data as a ROOT file. ROOT is a widely available high energy physics data analysis library, available from `''http://root.cern.ch''` and prepacked with the MAUS third party libraries. Two techniques are foreseen for accessing the data, either using `PyRoot` python interface or using a compiled C++ binary. Some mention of ROOT cint scripting tools is made below, but this is not supported by MAUS developers beyond the most basic usage.

1.4.1 Loading ROOT Files in Python Using PyROOT

The standard scripting tool in MAUS is python. The ROOT data structure can be loaded in python using the PyROOT package. An example of how to perform a simple analysis with PyROOT is available in `bin/examples/load_root_file.py`. This example runs the reconstruction code to produce an output data file `${MAUS_ROOT_DIR}/tmp/example_load_root_file.root` and then runs a toy analysis that plots digits at TOF1 for plane 0 and plane 1. This example produces two histograms, `tof1_digits_0_load_root_file.png` and `tof1_digits_1_load_root_file.png`.

1.4.2 Loading ROOT Files in C++ Compiled Analysis Code

The ROOT data structure can be loaded in C++ by compiling the Make file found in `bin/examples/load_root_file_cpp/Makefile`. This compiles the sample analysis in `bin/examples/load_root_file_cpp/load_root_file.cc`. For example,

```
$ source env.sh
$ cd ${MAUS_ROOT_DIR}/bin/examples
$ python load_root_file.py
$ cd ${MAUS_ROOT_DIR}/bin/examples/load_root_file_cpp/
$ make clean
$ make
$ ./load_root_file
```

This example performs a simple analysis against the data file generated by load_root_file.py, which is identical to the analysis performed by load_root_file.py. The executable produces two histograms, tof1_digits_0_load_root_file_cpp.png and tof1_digits_1_load_root_file_cpp.png; they should be identical to the histograms produced by load_root_file.py.

1.4.3 Loading ROOT Files on the ROOT Command Line

One can load ROOT files from the command line using the ROOT interactive display. It is first necessary to load the MAUS class dictionary. Then The TBrowser ROOT GUI can be used to browse to the desired location and interrogate the data structure interactively. For example,

```
$ source env.sh
$ root

  *******************************************
  *                                         *
  *        W E L C O M E  to  R O O T       *
  *                                         *
  *   Version   5.30/03   20 October 2011   *
  *                                         *
  *  You are welcome to visit our Web site  *
  *          http://root.cern.ch            *
  *                                         *
  *******************************************

ROOT 5.30/03 (tags/v5-30-03@41540, Oct 24 2011, 11:51:36 on linuxx8664gcc)

CINT/ROOT C/C++ Interpreter version 5.18.00, July 2, 2010
Type ? for help. Commands must be C++ statements.
Enclose multiple statements between { }.
root [0] .L $MAUS_ROOT_DIR/build/libMausCpp.so
root [1] TBrowser b
```

Note: ROOT infrastructure can only be used to plot data nested within up to two dynamic arrays. Data nested in three or more dynamic arrays is beyond the capabilities of ROOT interactive plotting tools; explicit loops over the data are required in a PyROOT script or C++ code. In general, working through the ROOT command line or ROOT macros is notoriously unreliable and is not supported by the MAUS development team; it is useful as a basic check of data integrity and no more.

More information on the data is available in the data structure chapter 2.

Chapter 2

Using and Modifying the Data Structure

MAUS operates on data in discrete blocks, primarily spills, with one spill representing the particle burst generated by one dip of the MICE target. Additionally, MAUS can write data into a JobHeader, RunHeader, RunFooter and JobFooter data type. Histograms for plotting in online mode are encoded into an Image data type. The top level branch in the data tree inherits from MAUSEvent<T>, defined in `src/common_cpp/DataStructure/MAUSEvent.hh` (C++) with type identified by GetEventType() string; in JSON the top level branch always has a `maus_event_type` member which is a string value corresponding to the output of MAUSEvent<T>::GetEventType(). A summary of configuration cards affecting Input, Output and data structure is shown below.

2.1 Metadata

Job metadata is stored in JobHeader and JobFooter data structures. (Data) Run metadata is stored in RunHeader and RunFooter data structures. The JobHeader is created at the start and end of an execution of the code and stores data on datacards, bzr state and so forth. The RunHeader is created at the start of each run and stores per run metadata such as the calibrations and cablings used. One RunHeader and RunFooter is written for each process in the entire *transform* and *merge* execution structure; so in multithreading mode this would yield one RunHeader and RunFooter for each Celery subprocess (which runs the Input/Transform) and an additional RunHeader and RunFooter for the merge/output process. In single threaded mode a single RunHeader and RunFooter is generated. The RunFooter and JobFooter are created at the end of the run and store run and job summary information. For more details on writing to these metadata types and multithreading modes, please see the section on API.

The Metadata is stored in ROOT in trees separate to the main Spill data tree. In JSON, these data are stored as separate lines often at the start and end of the run, and distinguished by the `maus_event_type` branch in the root. The structure of a MAUS output file is shown below.

Table 2.1: I/O control variables.

Name	Meaning
`input_root_file_name`	Set the file name used for reading input files by InputCppRoot module
`output_root_file_name`	Set the file name used for writing output files by OutputCppRoot module
`input_json_file_name`	Set the file name used for reading input files by InputPyJSON module
`input_json_file_type`	Set to `gzip` to read input from a gzipped file; set to `text` to read input from a plain text file
`output_json_file_name`	Set the file name used for writing output files by OutputPyJSON module
`output_json_file_type`	Set to `gzip` to write output as a gzipped file; set to `text` to write output as a plain text file
`header_and_footer_mode`	Set to `append` to write out job and run headers and footers; set to `dont_append` to suppress this output.

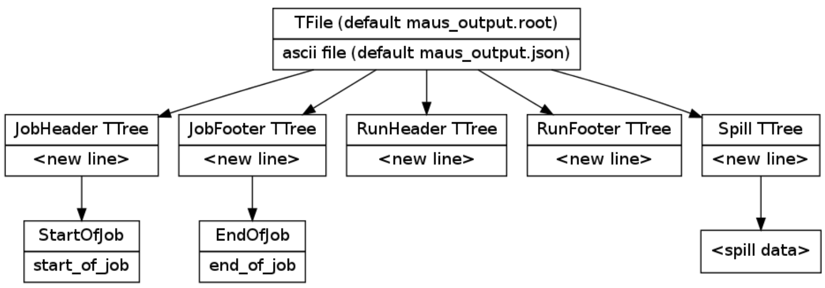

Figure 2.1: The MAUS file structure including metadata. The top label in each box describes the representation in C++/ROOT. The bottom label describes the representation in JSON.

2.2 The Spill Datastructure

The major part of the MAUS data structure therefore is a tree of which each entry corresponds to the data associated with one spill. The spill is separated into three main sections: the MCEventArray contains an array of data each member of which represents the Monte Carlo of a single primary particle crossing the system; the ReconEventArray contains an array of data each member of which corresponds to a particle event (i.e. set of DAQ triggers); and the DAQData corresponds to the raw data readout. Additionally there are branches for reconstructed scalars, which are handled spill by spill and EMR data, which also read out on the spill rather than event by event.

The MCEvent is subdivided into sensitive detector hits and some pure Monte Carlo outputs. The primary that led to data being created is held in the Primary branch. Here the random seed, primary position momentum and so forth

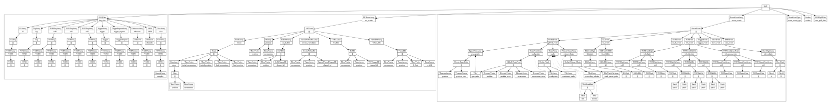

Figure 2.2: The MAUS output structure for a spill event. The top label in each box is the name of the C++ class and the bottom label is the json branch name. If a [] is shown, this indicates that child objects are array items.

is stored. Sensitive detector hits have Hit data (energy deposited, position, momentum, etc) and a detector specific ChannelId that represents the channel of the detector that was hit - e.g. for TOF this indexes the slab, plane and station. Virtual hits are also stored - these are not sensitive detector hits, rather output position, momenta etc of particles that cross a particular plane in space, time or proper time is recorded. Note virtual hits do not inherit from the Hit class and have a slightly different data structure.

The ReconEvent and DAQEvents are subdivided by detector. ReconEvents contain reconstructed particle data for each detector and the trigger. There is an additional branch that contains global reconstruction output, that is the track fitting between detectors.

The data can be written in two formats. The main data format is a ROOT binary format. This requires the ROOT package to read and write, which is a standard analysis/plotting package in High Energy Physics and is installed by the MAUS build script. The secondary data format is JSON. This is an ascii data-tree format that in principle can be read by any text editor. Specific JSON parsers are also available - for example, the python *json* module is available and comes prepackaged with MAUS.

2.3 Image Datastructure

There is a final data type that MAUS handles, the Image type. The Image data structure is written by ReducePyMatplotHistogram and ReducePyROOTHistogram data types. Image data is only available in JSON format. The data structure is shown in Fig. 2.3.

Each document contains a `maus_event_type` that should always be `Image`, and a list of images; the image data is encoded as a base 64 image and other data associated with the image is stored alongside. The `tag` names the image, while `image_type` describes the data format (png, jpeg, etc). OutputPyImage stores data with `image_type.tag` as the file name. `description` contains a description of the file and keywords describes a list of key phrases that can be used when searching.

2.4 Accessing ROOT files

For details on how to access the ROOT files, please see the introduction section of this document.

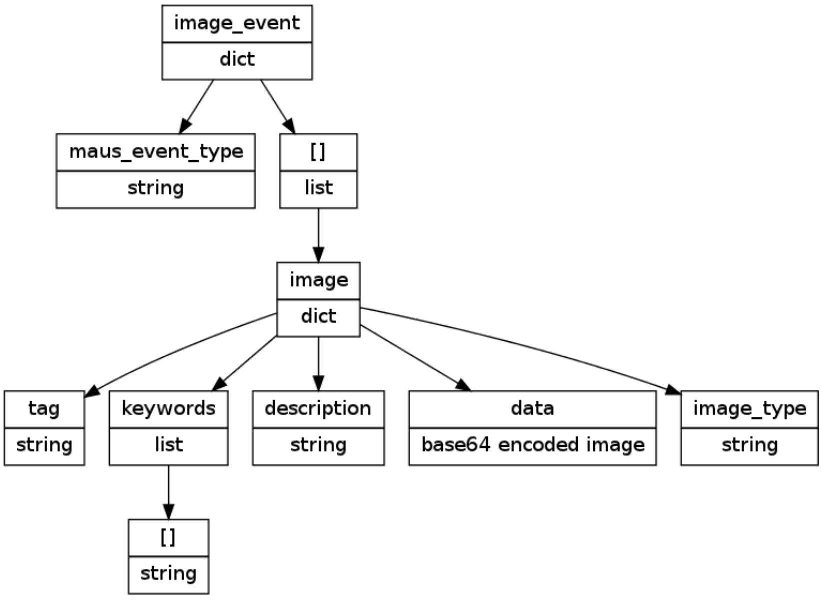

Figure 2.3: The MAUS output structure for an Image event. The top label in each box is the name of the JSON branch and the bottom label is the data type. If a [] is shown, this indicates that child objects are array items. Note there is no C++ implementation of Image events

2.5 Conversion to, and Working With, JSON

MAUS also provides output in the JSON data format. This is an ascii format with IO libraries available for C++, Python and other languages. Two utilities are provided to perform conversions, `bin/utilities/json_to_root.py` and `bin/utilities/root_to_json.py` for conversion from and to JSON format respectively. JSON Input and Output modules are provided, `InputPyJson` and `OutputPyJson`.

An example json analysis is available in `bin/examples/load_json_file.py`/

2.6 Extending the Data Structure

The data structure can be extended in MAUS by adding extra classes to the existing data structure. The data classes are in `src/common_cpp/DataStructure`. In order to make these classes accessible to ROOT, the following steps must be taken:

- Add a new class in `src/common_cpp/DataStructure`.
- Ensure that default constructor, copy constructor, equality operator and destructor is present. The destructor must be virtual.
- Add `#include "src/common_cpp/Utils/VersionNumber.hh"` and a call to the `MAUS_VERSIONED_CLASS_DEF()` macro at the end of the class definition before the closing braces. `MAUS_VERSIONED_CLASS_DEF` calls the

ROOT ClassDef() macro which generates metaclasses based on information in the class. This is put into the (dynamically generated) `MausDataStructure.h,cc` files.

- Add the class to the list of classes in `src/common_cpp/DataStructure/LinkDef.hh`. This is required for the class to be linked properly to the main library, and a linker error will result if this step is not taken.

- Add any template definitions which you used, including STL classes (e.g. std::vector<MyClass> or whatever) to linkdef. Otherwise ROOT will generate a segmentation fault whenever the user tries to call functions of the templated class (but the code will link successfully in this case).

In order to make these classes accessible to JSON, it is necessary to add a new processor in `src/common_cpp/JsonCppProcessors`. There are a few default processors available.

- `src/common_cpp/JsonCppProcessors/ProcessorBase.hh` contains IProcessor pure interface class for all processors and ProcessorBase base class (which may contain some implementation)

- `src/common_cpp/JsonCppProcessors/PrimitivesProcessors.hh` contains processors for primitive types; BoolProcessor, IntProcessor, UIntProcessor, StringProcessor, DoubleProcessor

- `src/common_cpp/JsonCppProcessors/ArrayProcessors.hh` contains processors for array types. Two processors are available: PointerArrayProcessor which converts an STL vector of pointers to data; and ValueArrayProcessor which converts an STL vector of values to data.

- `src/common_cpp/JsonCppProcessors/ObjectProcessor.hh` contains a processor for object types. Most of the classes in the MAUS data structure are represented in JSON as objects (string value pairs) where each string names a branch and each value contains data, which may be another class.

- `src/common_cpp/JsonCppProcessors/ObjectMapProcessors.hh` contains a processor for converting from JSON objects to STL maps. This is useful for JSON objects that contain lots of branches all of the same type.

A script, `bin/user/json_branch_to_data_structure_and_cpp_processor.py` is available that analyses a JSON object or JSON tree of nested objects and converts to C++ classes. The script is provided "as-is" and it is expected that developers will check the output, adding comments and tests where appropriate.

2.6.1 Pointer Handling

MAUS can handle pointers for arrays and classes using ROOT native support (via the `TRef` and `TRefArray` classes) or the standard JSON reference syntax. JSON references are indexed by a path relative to the root value of a JSON document. JSON references are formatted like URIs, for example the JSON object `{"$ref":"#spill/recon_events/1"}` would index the second `recon_event` in the spill object (indexing from 0). MAUS can only handle paths relative to the top level of the JSON document for the same MAUS event. Absolute URIs,

URIs relative to another position in the JSON document or URIs to another MAUS event are not supported.

In MAUS, it is necessary to make a distinction between data that is stored as a value in C++ and JSON (value-as-data), data that is stored as a pointer in C++ and a value in JSON (pointer-as-data) and data that is stored as a pointer in C++ and JSON to some other data in the same tree (pointer-as-reference). In the latter case, the C++ parent object does not own the memory; rather it is owned by some other object in the same tree and *borrowed* by the C++ object holding the pointer-as-reference. The `TRef` and `TRefArray` classes provide this functionality by default; never owning the memory but only storing a relevant pointer. All objects referenced by a `TRef` or `TRefArray` must inherit from `TObject`. ROOT handles all memory management while writing to and reading from ROOT files, and the order of reading is unimportant, as long as both reference and value have been read before the reference is used.

Pointers-as-data are converted between JSON arrays and C++ objects using the `ObjectProcessor<ParentType>::RegisterPointerBranch<ChildType>` method. This takes a Processor for the ChildType as an argument. For C++ arrays / vectors, the Processor argument is instead a `PointerArrayProcessor<ArrayContents>`. Pointers-as-reference (TRef and TRefArray) are converted using the `ObjectProcessor<ParentType>::Regis` and `ObjectProcessor<ParentType>::RegisterTRefArray` methods respectively.

Other equivalent data formats, for example YAML, use a unique identifier to reference a pointer-as-reference and store the pointer-as-data in a reserved part of the data tree. There are some consequences of storing pointers-as-reference using the path to a pointer-as-data as implemented in MAUS.

- The user must specify which data is the primary data source (pointer-as-data) and which data is a cross reference (pointer-as-reference).
- Pointers-as-reference are position dependent. If the associated pointer-as-data is moved the pointer-as-reference can no longer be resolved. For example, inserting an element into an array can cause misalignment of pointers-as-reference.
- Pointer data will always be available at the location of the pointer-as-data in the JSON tree, even when using a parser that is not pointer aware.
- A unique identifier type algorithm can be implemented as a relatively simple extension of the data format outlined here; but it is relatively hard to extend a unique identifier algorithm to reference existing parts of the data tree.

Pointer Resolution

Conversion from C++ pointers to JSON pointers is handled in a type-safe way. Values-as-data are stored in the data tree converted at run time from JSON to C++ and vice versa. Pointers-as-data are handled in the same way as Values-as-data. Pointers-as-references are stored in the C++ data tree as a TRef (or TRefArray element) in the normal way, and in JSON as an address to the position in the tree to a pointer-as-data. It is an error to store a pointer-as-reference without storing an associated pointer-as-data as the pointer-as-reference cannot be converted, unless the pointer-as-reference is set to NULL (in which case it may be an error depending on caller settings). It is an error to store multiple C++ pointers-as-data to the same memory address as the conversion from C++ to JSON and back again would yield logically different data and the resolution of associated pointers-as-reference is dependent on the resolution order of the data tree, which is ill-defined.

In order to implement the data conversion, the pointers have to be resolved in a two-stage process. In the first stage, it is necessary to collect all of the pointers-as-data and pointers-as-reference by traversing the data tree. This is performed during the standard data conversion, but pointers-as-reference are left pointing to NULL. A mapping from the pointer-as-data in the original data format to the pointer-as-data in the converted data format is stored, together with a list of pointers-as-reference in the original data format and the necessary mutators in the converted data format. In the second stage MAUS iterates over the pointers-as-reference, finds the appropriate pointer-as-data and writes the location of the pointer-as-data to the pointer-as-reference in the converted data format. The code is templated to maintain full type-safety during this process.

Chapter 3

Introduction to the MAUS API

This chapter introduces the MAUS API framework and looks in depth at the structure of the classes and interfaces that it comprises of. Several example *minimal* implementations are given before a note on scalibility and extending the framework.

3.1 Motivation

The motivation behind the MAUS API framework was to provide MAUS developers with a flexible, well defined environment whilst minimising the job of actually implementing new functionality. The framework must be robust but also scabale enough to cope with both current and unforseen new functionality.

To achieve these goals the MAUS framework has been designed from the ground up with scalibility and ease of developer implementation in mind. It features seperate interface and abstraction layers. While the interfaces provide guarenteed minimal implementation to ensure code works, the abstraction layer provides a convienient centrallised place for common as well as tedious implementation that would otherwise become a distraction or bloat a developers code.

3.2 Everything starts with a 'Module'

A *Module* is the basic building block of the MAUS API framework it's design is layed out within the interface 'IModule' shown in 3.1. The interface in essence requires two public void functions *birth* and *death* which are responsible for the initialisation and finalisation of the module.

```
class IModule {
  public:
    virtual void birth(const std::string&) = 0;
    virtual void death()                   = 0;
};
```

Listing 3.1: The module interface 'IModule'

Accompanying the interface is an abstract base class *ModuleBase*3.2. This again provides flexibility as the abstraction is seperated from the definition of the interface such that a developer may (if they wish) choose *not* to have the abstracted behaviour but still have their module plug in to the rest of the MAUS framework. It should be noted however that the expected behaviour would be to inherit the abstractions from this base class.

In 3.2 the implementation of the interface can be seen with the definition of the public birth and death member functions. It is important to note the lack of the virtual specifier in this case. The intention here (as is good C++ practise) is that any derived classes do not overide (hide) these methods but rather implement the pure virtual *and private* *_birth* and *_death* functions instead. This enables the public functions to wrap and provide abstracted behaviour around the private ones.

It is worth noting at this point the addition of the class member *_classname* which is set in the constructor and represents the name of the module.

```
class ModuleBase : public virtual IModule {
  public:
    // Constructors & Destructors
    explicit ModuleBase(const std::string&);
    ModuleBase(const ModuleBase&);
    virtual ~ModuleBase();

  public:
    void birth(const std::string&);
    void death();

  protected:
    std::string _classname;

  private:
    virtual void _birth(const std::string&) = 0;
    virtual void _death()                   = 0;
};
```

Listing 3.2: The abstract module base class 'ModuleBase'

A minimal working implementation of a module would be as in 3.3. Note the implementation of the pure virtual private *_birth* and *_death* functions.

```
class MyModule : public ModuleBase {
  public:
    // Constructors & Destructors
    explicit MyModule(const std::string& s) : ModuleBase(s) {}
    MyModule(const MyModule& m) : ModuleBase(m) {}
    virtual ~ModuleBase() {}

  private:
    virtual void _birth(const std::string& s) {
      // Your initialisation code here
    }
```

```
    virtual void _death() {
      // Your finalisation code here
    }
};
```

Listing 3.3: A minimal working module

As is, this module 'MyModule' doesn't contain anything except the ability to be initialised and finalised. While generally a developer will extend one of the classes described in the next sections which derive from the ModuleBase it is worth noting that one can create a standalone module in this way.

3.3 Inputters

The first module type defined in the API is the inputter. This type of module is responsible for the generation of a data object be it by monte carlo methods or streaming a disk resident file. It's layout is defined in the *IInput* interface 3.4. As with the other module types defined in this chapter the IInput interface inherits from IModule picking up the pure virtual birth and death functions. In addition IInput defines a third pure virtual function *emitter*. This function is responsible for returning the data object.

The IInput interface is templated to allow for implementation specific data object return types.

```
template<typename T>
class IInput : public virtual IModule {
  public:
    virtual T* emitter() = 0;
};
```

Listing 3.4: The inputter interface 'IInput'

The associated abstract base class *InputBase* behaves in much the same way as for ModuleBase. Here the inheritance completes the diamond inheritance structure from both the IInput interface and the abstractions from ModuleBase. Note accordingly the use of the virtual inheritance. As with ModuleBase, it is expected that the developer creating an inputter module inherit from this class and implement the pure virtual private *_emitter* function.

```
template <typename T>
class InputBase : public virtual IInput<T>,
                  public ModuleBase {
  public:
    explicit InputBase(const std::string&);
    InputBase(const InputBase&);
    virtual ~InputBase();

  public:
    T* emitter();

  private:
    virtual T* _emitter() = 0;
};
```

Listing 3.5: The abstract inputter base class 'InputBase'

A minimal implementation of an inputter then would be as in 3.6. Note that here we are inheriting from the InputBase class template with a template parameter (data object type) of *Spill*. This in turn means that our minimal class implementation need not itself be a class template. As InputBase also inherits from the ModuleBase both the pure virtual private functions *_birth* and *_death* must be implemented.

```
class MyInput : public InputBase<Spill> {
  public:
    explicit MyInput(const std::string& s) :
        InputBase<Spill>(s) {}
    MyInput(const MyInput& m) : InputBase<Spill>(m) {}
    virtual ~MyInput() {}

  private:
    virtual void _birth(const std::string& s) {
      // Your initialisation code here
    }
    virtual void _death() {
      // Your finalisation code here
    }
    virtual Spill* _emitter() {
      // Your emitter code here
    }
  };
```

Listing 3.6: A minimal working inputter

3.4 Outputters

Outputters are responsible for doing something with the data once processed. Typically the final element in the chain an outputter can for example be responsible for writing the data to a persistant media or uploading it to a web server etc. The layout of an outputter is not dissimilar from that of the inputter as one might expect and is defined in the *IOutput* interface 3.7. As with the inputter the interface defines a class template with the template parameter being the data object type.

```
template<typename T>
class IOutput : public virtual IModule {
  public:
    virtual bool save(T*) = 0;
};
```

Listing 3.7: The outputter interface 'IOutput'

As ever there is the corresponding abstract base class *OutputBase* shown in 3.8. The _save member function is for the developer to implement and takes as an argument a pointer to the data object. The return value of this function is a simple bool type which represents the success/failure of the outputter to complete it's task.

```
template <typename T>
class OutputBase : public virtual IOutput<T>,
                   public ModuleBase {
  public:
    // Constructors & Destructors
    explicit OutputBase(const std::string&);
    OutputBase(const OutputBase&);
    virtual ~OutputBase();

  public:
    bool save(T*);

  private:
    virtual bool _save(T*) = 0;
};
```

Listing 3.8: The abstract outputter base class 'OutputBase'

3.5 Reducers

Reducers are data processors and usually come at the end of a chain of mappers (see section 3.6). They can accumulate data from several events in their internal state and do something with the information i.e. create a histogram. They are defined by the interface *IReduce* as in 3.9. Note as before this is also a class template with the template parameter being the data object type. The process method, having used the data then returns an object of the same type such that it can be passed to an outputter for storing/streaming etc.

```
template<typename T>
class IReduce : public virtual IModule {
  public:
    virtual T* process(T* t) = 0;
};
```

Listing 3.9: The reducer interface 'IReducer'

The corresponding adstract base class *ReduceBase* can be seen in 3.10.

```
template <typename T>
class ReduceBase : public virtual IReduce<T>,
                   public ModuleBase {
  public:
    // Constructors & Destructors
    explicit ReduceBase(const std::string&);
    ReduceBase(const ReduceBase&);
    virtual ~ReduceBase();

  public:
    T* process(T*);

  private:
```

```
    virtual T* _process(T*) = 0;
};
```

Listing 3.10: The abstract reducer base class 'ReduceBase'

3.6 Mappers

Similar to reducers, mappers are used to process data. They are defined by the *IMap* interface as in 3.11. Unlike reducers they have no internal state and hence the process method is defined const. The IMap interface defines a class template as with the other module types in this chapter. However unlike them it takes two template parameters, *INPUT* and *OUTPUT*, which represent the input and output data object types respectively. The reason for this was due to an upgrade to the original specification which required the mappers to be able to accept input types *other* than the expected type. This will become more clear when looking at the abstract base class. Surfice to say for now that when implementing a mapper the developer must give as template parameters those types which s/he expects to be input and output.

```
template <typename INPUT, typename OUTPUT>
class IMap : public virtual IModule {
  public:
    virtual OUTPUT* process(INPUT*) const = 0;
};
```

Listing 3.11: The map interface 'IMap'

The abstract base class *MapBase*, seen in 3.12 looks slightly different then from the other module types shown before precisly because of this upgraded functionality. Note the addition in this case of templated public member function which overloads the standard public process method. This overloaded method will be called in all cases where the input data object type is not the same as the expected type here denoted *INPUT*. Since there remains only the one pure virtual private _process method, this templated method attempts to perform an automatic conversion of the input data object to the type expected by the developer. This abstracted behaviour means that the developer can go ahead and write their mapper knowing that no matter what inputter is used in the chain their code will be able to run.

This automatic conversion is performed by a *converter* object which is retrieved from the *ConverterFactory* as described in **??**.

```
template <typename INPUT, typename OUTPUT>
class MapBase : public virtual IMap<INPUT, OUTPUT>,
                public ModuleBase {
  public:
    // Constructors & Destructors
    explicit MapBase(const std::string&);
    MapBase(const MapBase&);
    virtual ~MapBase();

  public:
    OUTPUT* process(INPUT*) const;
```

```
    template <typename OTHER> OUTPUT* process(OTHER*) const;

  private:
    virtual OUTPUT* _process(INPUT*) const = 0;
};
```

Listing 3.12: The abstract map base class ‘MapBase’

While at first glance this looks like it has added an extra layer of complexity for the developer, it’s actuall no extra work at all. This is due to the abstraction layer absorbing all the extra complexity and shielding the developer from it. By way of example, compare the minimal mapper example in 3.13 with that of the minimal inputter in 3.6. In this example it is expected that the mapper receive a data object of type Json::Value and will return the data in a type Spill. If now a particular inputter returns the data as type Spill we will still be able to use our mapper as a Spill to Json::Value converter will run on the data first to ensure the data is of the right type.

```
class MyMap : public MapBase<Json::Value, Spill> {
  public:
    // Constructors & Destructors
    explicit MyMap(const std::string& s) :
        MapBase<Json::Value, Spill>(s) {}
    MyMap(const MyMap& m) :
        MapBase<Json::Value, Spill>(m) {}
    virtual ~MyMap() {}

  private:
    virtual void _birth(const std::string& s) {
      // Your initialisation code here
    }
    virtual void _death() {
      // Your finalisation code here
    }
    virtual Spill* _process(Json::Value*) const {
      // Your processing code here
    }
};
```

Listing 3.13: A minimal working mapper

3.7 Scalability

It was an important motivation that the MAUS code be scalable for future unseen uses. To this end, the MAUS API framework is build upon the idea of a inheritance ladder as depicted in 3.7. The ladder is essentially an extension of the ‘dreaded diamond’ structure and allows for extension at any point. This figure shows the inheritance ladder for a *reducer* (see section3.5) but similar ladders exist for each of the other module types in the framework. The uppermost line of classes correspone to the interface layer while those on the second row represent the abstraction layer. The uncoloured elements represent possible extensions. The colourless box on the bottom, ‘MyReduce’, represents a developers implementation of the abstract ReduceBase. This has been touched on in this chapter already an represents a common inheritance from the abstract base. It is assumed that many such classes will be constructed. These classes are not considered extensions to the framework but rather elements which may be run within it.

The two leftmost colourless boxes do indeed represent an extension to the ladder an hence an extension to the framework. One may consider at some point in the future that there needs to be a more specialised sub class of the reducer. One can then implement a seperate interface and abstract base class for this and extend the ladder.

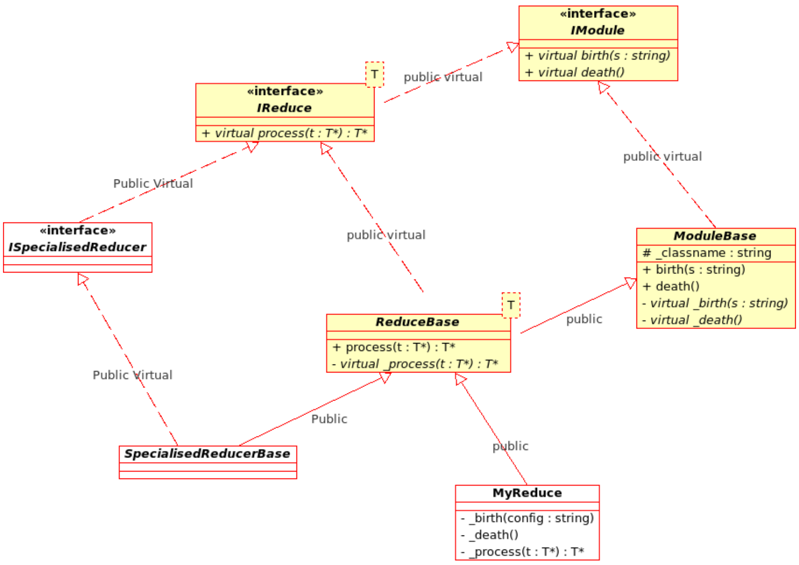

Figure 3.1: Inheritance ladder

3.8 Module Initialisation and Destruction

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 MICE data run or Monte Carlo run.

In MAUS, Inputters, Mappers, Reducers and Outputters are initialised at the start of every Job and destructed at the end of every Job. `birth(...)` for Inputters and Outputters is called at the start of every Job and `death()` is called at the end of every Job. The `birth(...)` for Mappers and Reducers is called at the start of every Run and `death()` is called at the end of every Run.

The logic is that for each code execution we typically want to access data from a single data source and output data to a single data file. But mappers and reducers are reinitialised for each run to enable loading of new calibrations, etc. It is required that all transient information about the reconstruction pertaining to a run - particularly ID of the calibration and cabling used - is recorded in the StartOfRun data structure. Any summary information on code execution during the run may be stored in the EndOfRun data structure. All transient information pertaining to a job - for example code version or bzr branch - should be recorded in the StartOfJob data structure. Any summary information on code execution during the job may be stored in the EndOfJob data structure.

3.9 Global Objects - Objects for Many Modules

There are some objects that sit outside the scope of the modular framework described above. Typically these are objects that do not belong to any one module, but need to be accessed by many. Examples are the logging functionality (Squeak), ErrorHandler, Configuration datacards, field maps, geometry description and Geant4 interfaces. These are accessed through the static singleton class `Globals` defined in `src/common_cpp/Utils/Globals.hh`. Initialisation is handled in `src/common_cpp/Globals/GlobalsManager.hh`. One Globals instance is initialised per subprocess when running in multiprocessing mode.

For python users, some Global objects can be accessed by reference to the `maus_cpp.globals module`.

3.9.1 Global Object Initialisation

Global objects are initialised before any modules in Go.py and deleted after all modules are deathed. Global object initialisation and destruction is handled at the Job level by `src/common_cpp/Globals/GlobalManager.hh` and called in python via `maus_cpp.globals` as above.

Run-by-run initialisation is handled by the RunActionManager, defined in `src/common_cpp/Utils/RunActionManager.hh`. The RunActionManager holds a list of objects inheriting from `RunActionBase` each of which defines functions to call at the start and end of each run.

Chapter 4

Running the Monte Carlo

The simulation module provides particle generation routines, GEANT4 bindings to track particles through the geometry and routines to convert modelled energy loss in detectors into digitised signals from the MICE DAQ. The Digitisation models are documented under each detector. Here we describe the beam generation and GEANT4 interface.

4.1 Beam Generation

Beam generation is handled by the MapPyBeamMaker module. Beam generation is separated into two classes. The MapPyBeamGenerator has routines to assign particles to a number of individual beam classes, each of which samples particle data from a predefined parent distribution. Beam generation is handled by the `beam` datacard.

The MapPyBeamMaker can either take particles from an external file, overwrite existing particles in the spill, add a specified number of particles from each beam definition, or sample particles from a binomial distribution. The random seed is controlled at the top level and different algorithms can be selected influencing how this is used to generate random seeds on each particle.

Each beam definition has routines for sampling from a multivariate gaussian distribution or generating ensembles of identical particles (called "pencil" beams here). Additionally it is possible to produce time distributions that are either rectangular or triangular in time to give a simplistic representation of the MICE time distribution.

The beam definition controls are split into four parts. The `reference` branch defines the centroid of the distribution; the `transverse` branch defines the transverse coordinates, x, y, px, py; the `longitudinal` branch defines the longitudinal coordinates - time and energy/momentum and the `coupling` branch defines correlations between longitudinal and transverse. Additionally a couple of parameters are available to control random seed generation and relative weighting between different beam definitions.

In transverse, beams are typically sampled from a multivariate gaussian.

The Twiss beam ellipse is defined by

$$\mathbf{B}_\perp = m \begin{pmatrix} \epsilon_x\beta_x/p & -\epsilon_x\alpha_x & 0 & 0 \\ -\epsilon_x\alpha_x & \epsilon_x\gamma_x p & 0 & 0 \\ 0 & 0 & \epsilon_y\beta_y/p & -\epsilon_y\alpha_y \\ 0 & 0 & -\epsilon_y\alpha_y & \epsilon_y\gamma_y p \end{pmatrix} \tag{4.1}$$

The Penn beam ellipse is defined by,

$$\mathbf{B}_\perp = m\epsilon_\perp \begin{pmatrix} \beta_\perp/p & -\alpha_\perp & 0 & -\mathcal{L}+\beta_\perp B_0/2p \\ -\alpha_\perp & \gamma_\perp p & \mathcal{L}-\beta_\perp B_0/2p & 0 \\ 0 & \mathcal{L}-\beta_\perp B_0/2p & \beta_\perp/p & -\alpha_\perp \\ -\mathcal{L}+\beta_\perp B_0/2p & 0 & -\alpha_\perp & \gamma_\perp p \end{pmatrix} \tag{4.2}$$

where parameters can be controlled in datacards as described below. Note that using the datacards it is possible to define a beam ellipse that is poorly conditioned (determinant nearly zero). In this case MAUS will print an error message like `Warning: invalid value encountered in double_scalars` for each primary.

4.2 GEANT4 Bindings

The GEANT4 bindings are encoded in the Simulation module. GEANT4 groups particles by run, event and track. A GEANT4 run maps to a MICE spill; a GEANT4 event maps to a single inbound particle from the beamline; and a GEANT4 track corresponds to a single particle in the experiment.

A number of classes are provided for basic initialisation of GEANT4.

- MAUSGeant4Manager: is responsible for handling interface to GEANT4. MAUSGeant4Manager handles initialisation of the GEANT4 bindings as well as accessors for individual GEANT4 objects (see below). Interfaces are provided to run one or many particles through the geometry, returning the relevant event data. The MAUSGeant4Manager sets and clears the event action before each run.

- MAUSPhysicsList: contains routines to set up the GEANT4 physical processes. Datacards settings are provided to disable stochastic processes or all processes and set a few parameters. In the end, the physics list set up gets called by the FieldPhaser.

- FieldPhaser: the field phaser is a MAUS-specific tool for automatically phasing fields, for example RF cavities, such that they ramp coincidentally with incoming particles. The FieldPhaser contains routines to fire test ("reference") particles through the accelerator lattice and phase fields appropriately. The FieldPhaser phasing routines are called after GEANT4 is first initialised.

- VirtualPlanes: the VirtualPlanes routines are designed to extract particle data from the GEANT4 tracking independently of the GEANT4 geometry. The VirtualPlanes routines watches for steps that step across some plane in physical space, or some time, or some proper time, and then interpolates from the step ends to the plane in question.

Table 4.1: Control parameters pertaining to all beam definitions.

Name	Meaning
`beam`	dict containing beam definition parameters.
The following cards should all be defined within the `beam` *dict.*	
`particle_generator`	Set to `binomial` to choose the number of particles by sampling from a binomial distribution. Set to `counter` to choose the number of particles in each beam definition explicitly. Set to `file` to generate particles by reading an input file. Set to `overwrite_existing` to generate particles by overwriting existing primaries.
`binomial_n`	When using a binomial `particle_generator`, this controls the number of trials to make. Otherwise ignored.
`binomial_p`	When using a binomial `particle_generator`, this controls the probability a trial yields a particle. Otherwise ignored.
`beam_file_format`	When using a file `particle_generator`, set the input file format - options are • `icool_for009` • `icool_for003`, • `g4beamline_bl_track_file` • `g4mice_special_hit` • `g4mice_virtual_hit` • `mars_1` • `maus_virtual_hit` • `maus_primary`
`beam_file`	When using a file `particle_generator`, set the input file name. Environment variables are automatically expanded by MAUS.
`file_particles_per_spill`	When using a file `particle_generator`, this controls the number of particles per spill that will be read from the file.
`random_seed`	Set the random seed, which is used to generate individual random seeds for each primary (see below).
`definitions`	A list of dicts, each item of which is a dict defining the distribution from which to sample individual particles.

Table 4.2: Individual beam distribution parameters.

Name	Meaning
The following cards should be inside a dict in the beam `definitions` *list.*	
`random_seed_algorithm`	Choose from the following options • `beam_seed`: use the `random_seed` for all particles • `random`: use a different randomly determined seed for each particle • `incrementing`: use the `random_seed` but increment by one each time a new particle is generated • `incrementing_random`: determine a seed at random before any particles are generated; increment this by one each time a new particle is generated
`weight`	When `particle_generator` is `binomial` or `overwrite_existing`, the probability that a particle will be sampled from this distribution is given by $weight/(sumofweights)$.
`n_particles_per_spill`	When `particle_generator` is `counter`, this sets the number of particles that will be generated in each spill.
`reference`	Dict containing the reference particle definition.
`transverse`	Dict defining the longitudinal phase space distribution.
`longitudinal`	Dict defining the longitudinal phase space distribution.
`coupling`	Dict defining any correlations between transverse and longitudinal.

Table 4.3: Beam distribution reference definition.

Name	Meaning
The following cards should be defined in each beam definition `reference` *dict.*	
`position`	dict with elements `x`, `y` and `z` that define the reference position (mm).
`momentum`	dict with elements `x`, `y` and `z` that define the reference momentum direction. Normalised to 1 at runtime.
`particle_id`	PDG particle ID of the reference particle.
`energy`	Reference energy.
`time`	Reference time (ns).
`random_seed`	Set to 0 - this parameter is ignored.

Table 4.4: Beam definition transverse parameters.

Name	Meaning
The following cards should be defined in each beam definition `transverse` *dict.*	
`transverse_mode`	Options are • `pencil`: x, py, y, py taken from `reference` • `penn`: cylindrical beam symmetric in x and y • `constant_solenoid`: cylindrical beam symmetric in x and y, with beam radius calculated from on-axis B-field to give constant beam radius along a solenoid. • `twiss`: beam with decoupled x and y beam ellipses.
`normalised_angular_momentum`	if `transverse_mode` is `penn` or `constant_solenoid`, set $\mathcal{L}$.
`emittance_4d`	if `transverse_mode` is `penn` or `constant_solenoid`, set $\epsilon_\perp$.
`beta_4d`	if `transverse_mode` is `penn`, set $\beta_\perp$.
`alpha_4d`	if `transverse_mode` is `penn`, set $\alpha_\perp$.
`bz`	if `transverse_mode` is `constant_solenoid`, set the B-field used to calculate $\beta_\perp$ and $\alpha_\perp$.
`beta_x`	if `transverse_mode` is `twiss`, set β_x.
`alpha_x`	if `transverse_mode` is `twiss`, set α_x.
`emittance_x`	if `transverse_mode` is `twiss`, set ϵ_x.
`beta_y`	if `transverse_mode` is `twiss`, set β_y.
`alpha_y`	if `transverse_mode` is `twiss`, set α_y.
`emittance_y`	if `transverse_mode` is `twiss`, set ϵ_y.

Table 4.5: Beam definition longitudinal parameters.

Name	Meaning
The following cards should be defined in each beam definition `longitudinal` *dict.*	
`momentum_variable`	In all modes, set this variable to control which longitudinal variable will be used to control the input beam. Options are `energy`, `p`, `pz`.
`longitudinal_mode`	Options are • `pencil`: time, energy/p/pz taken from `reference` • `gaussian`: uncorrelated gaussians in time and energy/p/pz • `twiss`: multivariate gaussian in time and energy/p/pz • `uniform_time`: gaussian in energy/p/pz and uniform in time. • `sawtooth_time`: gaussian in energy/p/pz and sawtooth in time.
`beta_l`	In Twiss mode, set β_l
`alpha_l`	In Twiss mode, set α_l
`emittance_l`	In Twiss mode, set ϵ_l
`sigma_t`	In `gaussian` mode, set the RMS time.
`sigma_p`	
`sigma_energy`	In `gaussian`, `uniform_time`, `sawtooth_time` mode,
`sigma_pz`	set the RMS energy/p/pz.
`t_start`	In `uniform_time` and `sawtooth_time` mode, set the start time of the parent distribution
`t_end`	In `uniform_time` and `sawtooth_time` mode, set the end time of the parent distribution

Table 4.6: Beam definition coupling parameters.

Name	Meaning
The following cards should be defined in each beam definition `coupling` *dict.*	
`coupling_mode`	Set to `none` - not implemented yet.

- FillMaterials: (legacy) the FillMaterials routines are used to initialise a number of specific
- MICEDetectorConstruction: (legacy) the MICEDetectorConstruction routines provide an interface between the MAUS internal geometry representation encoded in MiceModules and GEANT4. MICEDetectorConstruction is responsible for calling the relevant routines for setting up the general engineering geometry, calling detector-specific geometry set-up routines and calling the field map set-up routines.
- MAUSVisManager the MAUSVisManager is responsible for handling interfaces with the GEANT4 visualisation.

The GEANT4 *Action* objects provide interfaces for MAUS-specific function calls at certain points in the tracking.

- MAUSRunAction: sets up the running for a particular spill. In MAUS, it just reinitialises the visualisation.
- MAUSEventAction: sets up the running for a particular inbound particle. At the beginning of each event, the virtual planes, tracking, detectors and stepping are all cleared. After the event the event data is pulled into the event data from each element.
- MAUSTrackingAction: is called when a new track is created or destroyed. If `keep_tracks` datacard is set to True, on particle creation, MAUSTrackingAction writes the initial and final track position and momentum to the output data tree. If `keep_steps` is set to True MAUSTrackingAction gets step data from MAUSSteppingAction and writes this also.
- MAUSSteppingAction: is called at each step of the particle. If `keep_steps` datacard is set to True, output step data is recorded. MAUSSteppingAction kills particles if they exceed the `maximum_number_of_steps` datacard. MAUSSteppingAction calls the VirtualPlanes routines on each step.
- MAUSStackingAction: is called when a new track is created, prioritising particle tracking. Handles killing particles based on the `kinetic_energy_threshold`, `default_keep_or_kill` and `keep_or_kill_particles` datacards.
- MAUSPrimaryGeneratorAction: is called at the start of every event and sets the particle data for each event. In MAUS, this particle generation is handled externally and so the MAUSPrimaryGeneratorAction role is to look for the primary object on the Monte Carlo event and convert this into a GEANT4 event object.

Table 4.7: Monte Carlo control parameters.

Name	Meaning
General Monte Carlo controls.	
`simulation_geometry_filename`	Filename for the simulation geometry - searches first in files tagged by environment variable `${MICEFILES}`, then in the local directory.
`simulation_reference_particle`	Reference particle used for phasing fields.
`keep_tracks`	Set to boolean true to store the initial and final position/momentum of each track generated by MAUS.
`keep_steps`	Set to boolean true to store every step generated by MAUS - warning this can lead to large output files.
`maximum_number_of_steps`	Set to an integer value. Tracks taking more steps are assumed to be looping.

Table 4.8: Physics list control parameters.

Physics list controls.	
physics_model	GEANT4 physics model used to set up the physics list.
physics_processes	Choose which physics processes normal particles observe during tracking. Options are • normal particles will obey normal physics processes, scattering and energy straggling will be active. • mean_energy_loss particles will lose a deterministic amount of energy during interaction with materials and will never decay. • none Particles will never lose energy or scatter during tracking and will never decay.
reference_physics_processes	Choose which physics processes the reference particle observes during tracking. Options are mean_energy_loss and none. The reference particle can never have stochastic processes enabled.
particle_decay	Set to boolean true to enable particle decay; set to boolean false to disable.
charged_pion_half_life	Set the half life for charged pions.
muon_half_life	Set the half life for muons.
production_threshold	Set the geant4 production threshold.
kinetic_energy_threshold	Threshold for kinetic energy of new particles at production. Particles with kinetic energy below this value will not be tracked.
default_keep_or_kill	If set to true, keep particles with type not listed in keep_or_kill_particles. If set to false, kill particles with type not listed in keep_or_kill_particles
keep_or_kill_particles	Maps string particle type name to boolean flag. If set to true, always keep particles of this type. If set to false, always kill particles of this type. If not set, apply default_keep_or_kill

Table 4.9: Visualisation control parameters.

Visualisation controls.	
geant4_visualisation	Set to boolean true to activate GEANT4 visualisation.
visualisation_viewer	Control which viewer to use to visualise GEANT4 tracks. Currently only vrmlviewer is compiled into GEANT4 by default. Users can recompile GEANT4 with additional viewers enabled at their own risk.
visualisation_theta	Set the theta angle of the camera.
visualisation_phi	Set the phi angle of the camera.
visualisation_zoom	Set the camera zoom.
accumulate_tracks	Set to 1 to accumulate all of the simulated tracks into one vrml file. 0 for multiple files.
default_vis_colour	Set the RGB values to alter the default colour of particles.
pi_plus_vis_colour	Set the RGB values to alter the colour of positive pions.
pi_minus_vis_colour	Set the RGB values to alter the colour of negative pions.
mu_plus_vis_colour	Set the RGB values to alter the colour of positive muons.
mu_minus_vis_colour	Set the RGB values to alter the colour of negative pions.
e_plus_vis_colour	Set the RGB values to alter the colour of positrons.
e_minus_vis_colour	Set the RGB values to alter the colour of electrons.
gamma_vis_colour	Set the RGB values to alter the colour of gammas.
neutron_vis_colour	Set the RGB values to alter the colour of neutrons.
photon_vis_colour	Set the RGB values to alter the colour of photons.

Chapter 5

Geometry

MAUS uses the online Configuration Database to store all of its geometries. These geometries have been transferred from CAD drawings which are modelling using the latest surveys and technical drawings available. The following section shall describe how to use the available executables to access and use these models.

5.1 Geometry Download

There are two executable files available to users both can be found in the directory /bin/utilities found within your copy of MAUS. The two files of interest are download_geometry.py and get_geometry_ids.py. These files do the following.

Upload Geometry

1. Set up the Configreader class and read the values provided by ConfigurationDefaults.py or by custom config files.
2. Instantiate an Uploader class object using the upload directory and geometry note taken from the configuration file.
3. The list of files which is created by the Uploader class is used to compress the geometry files into one zip file.
4. This zip file is then used as the argument for the upload_to_CDB method which takes the contents of the zip and then uploads this, as a single string to the CDB.

Optional If cleanup is specified in the configuration file then the file list and the original GDML files are the deleted leaving only the zip file.

Download Geometry

1. Set up the Configreader() class and read the values provided by ConfigurationDefaults.py or by custom config files.
2. Instantiate a Downloader class object and downloads either the current, time specified or run number zipped geometry to a temporary cache location.
3. The zip file is then unzipped in this location.

4. The Formatter class is then called and this class formats the GDMLs. The formatting alters the schema location of these files and points them to the correct local locations of the Materials GDML file. This formatting leaves the original GDMLs in the tmeporary cache and places the new formatted files in the download directory specified in the configuration file.
5. GDMLtoMAUS is then called with the location of the new formatted files as its argument. This class converts the GDMLs to the MICE Module text files using the XSLT stylesheets previously described.

Optional If specified in the configuration file the temporary cache location is removed along with the zip file and unzipped files.

Get Geometry IDs

1. Set up the Configreader() class and read the values provided by ConfigurationDefaults.py or by custom config files. This file takes start and stop time arguments to specify a period to search the CDB.
2. A CDB class object is then instantiated with the server specified in the configuration file.
3. The get ids method from the CDB class is called and the python dict which is downloaded is formatted and either printed to screen or to file as specified in the configuration file.

To use these files the user must use the arguments in the ConfigurationDefaults.py file. The arguments relating to these executables are as follows.

Table 5.1: Geometry control parameters.

Geometry controls.	
cdb_upload_url	Sets the upload url relating the the Configuration Database.
cdb_download_url	Sets the download url relating the the Configuration Database.
geometry_download_wsdl	Name of the web service used for downloads.
geometry_download_directory	Set the directory where you wish the geometry to be downloaded to.
geometry_download_by	This can be set to either *current, id or run_number*. Current will download the current valid geometry stored on the CDB. ID will download the geometry for the ID specified N.B ID numbers can be found using the get geometry ids executable. Run_number will download the geometry along with control room information for specified run.
geometry_download_run_number	Set the number of the run to be downloaded.
geometry_download_id	Set the number of the geometry ID to be downloaded.
geometry_download_cleanup	Set to True in order to cleanup the temporary files creaated during the download process. These are the zip file downloaded and the original GDML files from this zip file.
g4_step_max	Set the G4 step max number which will be set in the ParentGeometryFile. This relates to the size of steps carried out during the simulation.
geometry_upload_wsdl	Name of the web service used for uploads. For developers use only.
geometry_upload_directory	Set the the directory which stores the FastRad produced GDML files which will be stored on the CDB. For Developers use only.
geometry_upload_note	Write the description of the geometry which is going to be uploaded. This should describe what is in the beam line specifically what is new to the model. It should also include any other information the developer wishes the user to know. For developers use only.
geometry_upload_valid_from	Set the date-time format of the date when this geometry about to be uploaded is valid from. For developers use only.
geometry_upload_cleanup	Set to True in order to cleanup the temporary files creaated during the upload process. These are the file containing the list of GDMLs to be uploaded and also the original GDML files. For developers use only.
get_ids_start_time	Set the start time of the period which you would like to get the ids from the configuration database. Must be in date-time format.
get_ids_stop_time	Set the stop time of the period which you would like to get the ids from the configuration database. Must be in date-time format.

Chapter 6

How to Define a Geometry

Mice Modules are the objects that control the geometry and fields that are simulated in MAUS. They are used in conjunction with a datacard file, which provides global run control parameters. Mice Modules are created by reading in a series of text files when MAUS applications are run.

This geometry information is used primarily by the Simulation application for tracking of particles through magnetic fields. A few commands are specific to detector Reconstruction and accelerator beam Optics applications.

The Mice Modules are created in a tree structure. Each module is a parent of any number of child modules. Typically the parent module will describe a physical volume, and child modules will describe physical volumes that sit inside the parent module. Modules cannot be used to describe volumes that do not sit at least partially inside the volume if the parent module.

Each Mice Module file consists of a series of lines of text. Firstly the Module name is defined. This is followed by an opening curly bracket, then the description of the module and the placement of any child modules, and finally a closing curly bracket. Commands, curly brackets etc must be separated by an end of line character.

Comments are indicated using either two slashes or an exclamation mark. Characters placed after a comment on a line are ignored.

MAUS operates in a right handed coordinate system (x, y, z). In the absence of any rotation, lengths are considered to be extent along the z-axis, widths to be extent along the x-axis and heights to be extent along the y-axis. Rotations $(\theta_x, \theta_y, \theta_z)$ are defined as a rotation about the z-axis through θ_z, followed by a rotation about the y-axis through θ_y, followed by a rotation about the x-axis through θ_x.

Configuration File

The Configuration file places the top level objects in MICE. The location of the file is controlled by the datacard `simulation_geometry_file_name`. MAUS looks for the configuration file in the first instance in the directory

```
${MICEFILES}/Models/Configuration/<MiceModel>
```

where `${MICEFILES}` is a user-defined environment variable. If MAUS fails to find the file it searches the local directory.

The world volume is defined in the Configuration file and any children of the world volume are referenced by the Configuration file. The Configuration file looks like

```
Configuration <Configuration Name>
```

```
{
    Dimensions <x> <y> <z> <Units>
    <Properties>
    <Child Modules>
}
```

`<Configuration Name>` is the name of the configuration. Typically the Configuration file name is given by `<Configuration Name>.dat`. The world volume is always a rectangular box centred on $(0,0,0)$ with x, y, and z extent set by the Dimensions. Properties and Child Modules are described below and added as in any Module.

Substitutions

It is possible to make keyword substitutions that substitutes all instances of `<name>` with `<value>` in all Modules. The syntax is

```
Substitution <name> <value>
```

`<name>` must start with a single $ sign. Substitutions must be defined in the Configuration file. Note this is a direct text substitution in the MiceModules before the modules are parsed properly. So for example,

```
Substitution $Sub SomeText
PropertyString FieldType \$Sub}
PropertyDouble \$SubValue 10}
```

would be parsed as MAUS like

```
PropertyString FieldType SomeText}
PropertyDouble SomeTextValue 10}
```

Expressions

The use of equations in properties of type double and Hep3Vector is also allowed in place of a single value. So, for example,

```
PropertyDouble FieldStrength 0.5*2 T
```

would result in a FieldStrength property of 1 Tesla.

Expression Substitutions

Some additional variables can be defined in specific cases by MAUS itself for substitution into experssions, in which case they will start with @ symbol. For these variable substitutions, it is only possible to make the substitution into expressions. So for example,

```
PropertyDouble ScaleFactor 2*@RepeatNumber
```

Would substitute @RepeatNumber into the expression. @RepeatNumber is defined by MAUS when repeating modules are used (see RepeatModule2, below). Note the following code is not valid

```
PropertyString FileName File@RepeatNumber //NOT VALID
```

This is because Expression Substitutions can only be used in an expression (i.e. an equation).

Module Files

Children of the top level Mice Module are defined by Modules. MAUS will attempt to find a module in an external file. The location of the file is controlled by the parent Module. Initially MAUS looks in the directory

```
${MICEFILES}/Models/Modules/<Module>
```

If the Mice Module cannot be found, MAUS searches the local directory. If the module file still cannot be found, MAUS will issue a warning and proceed.

The Module description is similar in structure to the Configuration file:

```
Module <Module Name>
{
    Volume  <Volume Type>
    Dimensions  <Dimension1> <Dimension2> <Dimension3> <Units>
    <Properties>

    <Child Modules>
}
```

`<Module Name>` is the name of the module. Typically the Module file name is given by `<Module Name>.dat`.

The definition of Volume, Dimensions, Properties and Child Modules are described below.

Volume and Dimensions

The volume described by the MiceModule can be one of several types. Replace `<Volume Type>` with the appropriate volume below. Cylinder, Box and Tube define cylindrical and cuboidal volumes. Polycone defines an arbitrary volume of rotation and is described in detail below. Wedge describes a wedge with a triangular projection in the y-z plane and rectangular projections in x-z and x-y planes. Quadrupole defines an aperture with four cylindrical pole tips.

In general, the physical volumes that scrape the beam are defined independently of the field maps. This is the more versatile way to do things, but there are some pitfalls which such an implementation introduces. For example, in hard-edged fields like pillboxes, tracking errors can be introduced when MAUS steps into the field region. This can be avoided by adding windows (probably made of vacuum material) to force GEANT4 to stop tracking, make a small step over the field boundary, and then restart tracking inside the field. However, such details are left for the user to implement.

Volume	**Dimension1**	**Dimension2**	**Dimension3**
None	No dimensions required. Note cannot define daughter Modules for this volume type.		
Cylinder	Radius	Length in z	Not used (leave blank)
Box	Width in x	Height in y	Length along z
Tube	Inner Radius	Outer Radius	Length in z
Trapezoid	Half Width in x	Half Height in y	Half Length in z
Wedge	See documentation below.		
Polycone	No dimensions required. Volume defined from external file.		
Quadrupole	No dimensions required. Dimensions defined from module properties.		
Multipole	No dimensions required. Dimensions defined from module properties.		
Boolean	No dimensions required. Dimensions defined from module properties.		
Sphere	See documentation below.		

Volume	Dimension1	Dimension2	Dimension3

Properties

Many of the features of MAUS that can be enabled in a module are described using properties. For example, properties enable the user to define detectors and fields. Properties can be either of several types: PropertyDouble, PropertyString, PropertyBool, PropertyHep3Vector or PropertyInt. A property is declared via

```
<Property Type> <Property Name> <Value> <Units if appropriate>
```

Different properties that can be enabled for Mice Modules are described elsewhere in this document. Properties of type PropertyDouble and PropertyHep3Vector can have units. Units are defined in the CLHEP library. Units are case sensitive; MAUS will return an error message if it fails to parse units. Combinations of units such as T/m or N*m can be defined using '*' and '/' as appropriate. Properties cannot be defined more than once within the same module.

Child Modules

Child Modules are defined with a position, rotation and scale factor. This places, and rotates, the child volume and any fields present relative to the parent volume. Scale factor scales fields defined in the child module and any of its children. Scale factors are recursively multiplicative; that is the field generated by a child module will be scaled by the product of the scale factor defined in the parent module and all of its parents.

The child module definition looks like:

```
Module <Module File Name>
{
    Position  <x position> <y position> <z position> <Units>
    Rotation  <x rotation> <y rotation> <z rotation> <Units>
    ScaleFactor <Value>
    <Define volume, dimensions and properties for this instance only>
}
```

MAUS searches for `<Module File Name>` first relative to `${MICEFILES}/Models/Modules/` and subsequently relative to the current working directory. The position and rotation default to $0, 0, 0$ and the `scale factor` defaults to 1.

- Volume, Dimension and Properties of the child module can be defined at the level of the parent; in this case these values will be used only for this instance of the module.
- If no file can be found, MAUS will press on regardless, attempting to build a geometry using the information defined in the parent volume.

Module Hierarchy and GEANT4 Physical Volumes

MAUS enables users to place arbitrary physical volumes in a GEANT4 geometry. The formatting of MAUS is such that users are encouraged to use the GEANT4 tree structure for placing physical volumes. This is a double-edged sword, in that it provides users with a convenient interface for building geometries, but it is not a terribly safe interface.

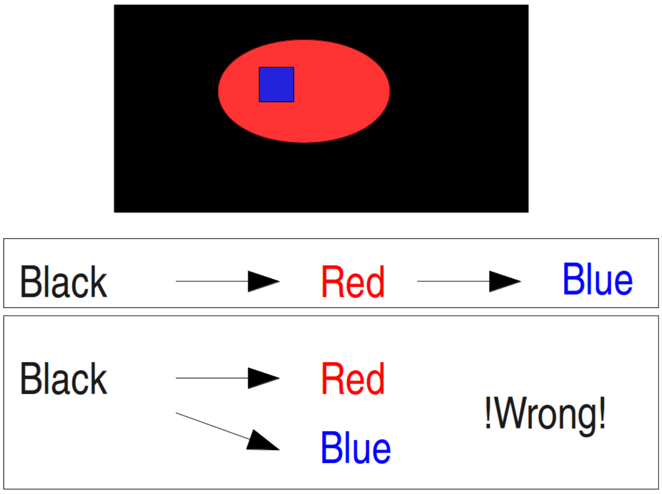

Figure 6.1: The diagram shows a schematic for a square placed inside a cylinder inside a rectangle. This nesting must be replicated in the MiceModules in order for the volumes to be correctly represented by MAUS.

Consider the cartoon of physical volumes shown above. Here there is a blue volume sitting inside a red volume sitting inside the black world volume. For the volumes to be represented properly, the module that represents the blue volume MUST be a child of the module that represents the red volume. The module that represents the red volume MUST, in turn, be a child of the module that represents the black volume, in this case the Configuration file.

What would happen if we placed the blue volume directly into the Black volume, i.e. the Configuration file? GEANT4 would silently ignore the blue volume, or the red volume, depending on the order in which they are added into the GEANT4 geometry. What would happen if we placed the blue volume overlapping the red and black volumes? The behaviour of GEANT4 is not clear in this case.

- Never allow a volume to overlap any part of another volume that is not it's direct parent.

It is possible to check for overlaps by setting the datacard *CheckVolumeOverlaps* to 1.

A Sample Configuration File

Below is listed a sample configuration file, which is likely to be included in the file *ExampleConfiguration.dat;* the actual name is specified by the datacard MiceModel.

```
Configuration ExampleConfiguration
{
    Dimensions 1500.0 1000.0 5000.0 cm
    PropertyString Material AIR
    Substitution $MyRedColour 0.75
    Module BeamLine/SolMag.dat
    {
        Position 140.0 0.0 -2175.0 cm
```

```
        Rotation 0.0 30.0 0.0 degree
        ScaleFactor 1.
    }
    Module BeamLine/BendMag.dat
    {
        Position 0.0 0.0 -1935.0 cm
        Rotation 0.0 15.0 0.0 degree
        ScaleFactor 1.
    }
    Module NoFile_Box1
    {
        Volume Box
        Dimension 1.0 1.0 1.0
        Position 0.0 0.0 200.0 cm
        Rotation 0.0 15.0 0.0 degree
        PropertyString Material  Galactic
        PropertyDouble RedColour $MyRedColour
    }
    Module NoFile_Box2}
    {
        Volume Box
        Dimension 0.5 0.5 0.5*3 m //z length = 0.5*3 = 1.5 m
        Rotation  0.0 15.0 0.0 degree //Rotation relative to parent
        PropertyString Material  Galactic
        PropertyDouble RedColour $MyRedColour
    }
}
```

A Sample Child Module File

Below is listed a sample module file, which is likely to be included in the file *SolMag.dat*; the actual location is specified by the module or configuration that calls FCoil. The module contains a number of properties that define the field.

```
Module SolMag
{
    Volume Tube
    Dimensions 263.0 347.0 210.0 mm
    PropertyString Material Al
    PropertyDouble BlueColour 0.75
    PropertyDouble GreenColour 0.75
    //field}
    PropertyString FieldType      Solenoid
    PropertyString FileName       focus.dat
    PropertyDouble CurrentDensity 1.
    PropertyDouble Length         210. mm
    PropertyDouble Thickness      84. mm
    PropertyDouble InnerRadius    263. mm
}
```

Chapter 7

Geometry and Tracking MiceModule Properties

In general, MAUS treats physical geometry distinct from fields. Fields can be placed overlapping physical objects, or entirely independently of them, as the user desires. Properties for various aspects of the physical and engineering model of the simulation are described below. This includes properties for sensitive detectors.

General Properties

There are a number of properties that are applicable to any MiceModule.

Property	**Type**	**Description**
Material	string	The material that the volume is made up from
Invisible	bool	Set to 1 to make the object invisible in visualisation, or 0 to make the object visible.
RedColour	double	Alter the colour of objects as they are visualised
GreenColour	double	
BlueColour	double	
G4StepMax	double	The maximum step length that Geant4 can make in the volume. Inherits values from the parent volumes.
G4TrackMax	double	The maximum track length and particle time of a track. Tracks outside this bound are killed. Inherits values from the parent volumes.
G4TimeMax	double	
G4KinMin	double	The minimum kinetic energy of a track. Tracks outside this bound are killed. Inherits values from the parent volumes.
SensitiveDetector	string	Set to the type of sensitive detector required. Possible sensitive detectors are *TOF*, *SciFi*, *CKOV*, *SpecialVirtual*, *Virtual*, *Envelope* or *EMCAL*.

Sensitive Detectors

A sensitive detector (one in which hits are recorded) can be defined by including the SensitiveDetector property. When a volume is set to be a sensitive detector MAUS will automatically record tracks entering, exiting and crossing the volume. Details such as the energy deposited by the track are sometimes also recorded in order to enable subsequent modelling of the detector response.

Some sensitive detectors use extra properties.

Scintillating Fibre Detector (SciFi)

Cerenkov Detector (CKOV)

Time Of Flight Counter (TOF)

Special Virtual Detectors

Special virtual detectors are used to monitor tracking through a particular physical volume. Normally particle tracks are written in the global coordinate system, although an alternate coordinate system can be defined. Additional properties can be used to parameterise special virtual detectors.

Property	Type	Description
ZSegmentation	int	Set the number of segments in the detector in Z, R or f. Defaults to 1.
PhiSegmentation	int	
RSegmentation	int	
SteppingThrough	bool	Set to true to record tracks stepping through, into, out of or across the volume. Defaults to true.
SteppingInto	bool	
SteppingOutOf	bool	
SteppingAcross	bool	
Station	int	Define an integer that is written to the output file to identify the station. Defaults to a unique integer identifier chosen by MAUS, which will be different each time the same Special Virtual is placed.
LocalRefRotation	Hep3 Vector	If set, record hits relative to a reference rotation in the coordinate system of the SpecialVirtual detector.
GlobalRefRotation	Hep3 Vector	If set, record hits relative to a reference rotation in the coordinate system of the Configuration.
LocalRefPosition	Hep3 Vector	If set, record hits relative to a reference position in the coordinate system of the SpecialVirtual detector.
GlobalRefPosition	Hep3 Vector	If set, record hits relative to a reference position in the coordinate system of the Configuration.

Virtual Detectors

Virtual detectors are used to extract all particle data at a particular plane, irrespective of geometry. Virtual detectors do not need to have a physical volume. The *plane* can be a plane in z, time, proper time, or a physical plane with some arbitrary rotation and translation.

Property	Type	Description
IndependentVariable	String	• If set to *t*, particle data will be written for particles at the time defined by the *PlaneTime* property. • If set to *tau*, particle data will be written for particles at the proper time defined by the PlaneTime property. • If set to *z*, particle data will be written for particles crossing the module's z-position. • If set to *u*, particle data will be written for particles crossing a plane extending in x and y.

Property	Type	Description
PlaneTime	Double	If *IndependentVariable* is *t* or *tau*, particle data will be written out at this time. Mandatory if *IndependentVariable* is *t* or *tau*.
RadialExtent	Double	If set, particles outside this radius in the plane of the detector will not be recorded by the Virtual detector.
GlobalCoordinates	Bool	If set to 0, particle data is written in the coordinate system of the module. Otherwise particle data is written in global coordinates.
MultiplePasses	String	Set how the VirtualPlane handles particles that pass through more than once. If set to Ignore, particles will be ignored on second and subsequent passes. If set to SameStation, particles will be registered with the same station number. If set to NewStation, particles will be registered with a NewStation number given by the *(total number of stations) + (this plane's station number)*, i.e. a new station number appropriate for a ring geometry.
AllowBackwards	Bool	Set to false to prevent backwards-going particles from being recorded. Default is true.

Envelope Detectors

Envelope detectors are a type of Virtual detector that take all of the properties listed under virtual detectors, above. In addition, in the optics application they can be used to interact with the beam envelope in a special way. The following properties can be defined for Envelope Detectors *in addition to* the properties specified above for virtual detectors.

The The EnvelopeOut properties are used to make output from the envelope for use in the Optics optimiser.

Property	Type	Description
EnvelopeOut1_Name	String	Defines the variable name that can be used as an expression substitution at the end of each iteration, typically substituted into the Score parameters in the optimiser (see optimiser, below).
EnvelopeOut1_Type	String	Defines the type of variable that will be calculated for the substitution. Options are • Mean • Covariance • Standard_Deviation • Correlation • Bunch_Parameter
EnvelopeOut1_Variable	String	Defines the variable that will be calculated for the substitution. Options are for Bunch_Parameter • ◦ *emit_6d* : 6d emittance ◦ *emit_4d:* 4d emittance (in x-y space) ◦ *emit_t:* 2d emittance (in time space) ◦ *emit_x:* 2d emittance (in x space) ◦ *emit_y:* 2d emittance (in y space) ◦ *beta_4d:* 4d transverse beta function ◦ *beta_t:* 2d longitudinal beta function ◦ *beta_x:* 2d beta function (in(x space) ◦ *beta_y:* 2d beta function (in y space) ◦ *alpha_4d:* 4d transverse alpha function ◦ *alpha_t:* 2d longitudinal alpha function ◦ *alpha_x:* 2d alpha function (in(x space) ◦ *alpha_y:* 2d alpha function (in y space) ◦ *gamma_4d:* 4d transverse gamma function ◦ *gamma_t:* 2d longitudinal gamma function ◦ *gamma_x:* 2d gamma function (in(x space) ◦ *gamma_y:* 2d gamma function (in y space) ◦ *disp_x:* x-dispersion ◦ *disp_y:* y-dispersion ◦ *ltwiddle:* normalised angular momentum ◦ *lkin:* standard angular momentum For Mean, Standard_Deviation, Covariance and Correlation, variables should be selected from the options • *x:* x-position • *y:y-position* • *t:* time • *px:* x-momentum • *py:* y-momentum • *E:* energy For Mean, a single variable should be selected and value corresponding to the reference trajectory will be returned. For Standard_Deviation, a single variable should be selected and the 1 sigma beam size will be returned. For Covariance and Correlation, two variables should be selected separated by a comma.

Unconventional Volumes

It is possible to define a number of volumes that use properties rather than the Dimensions keyword to define the volume size.

Volume Trapezoid

Volume Trapezoid gives a trapezoid which is not necessarily isosceles. Its dimensions are given by:

Property	Type	Description
TrapezoidWidthX1	Double	Gives width1 in x
TrapezoidWidthX2	Double	Gives width2 in x
TrapezoidWidthY1	Double	Gives height1 in y
TrapezoidWidthY2	Double	Gives height2 in y
TrapezoidLengthZ	Double	Gives length along z

Trapezoid Volume

A Trapezoid Volume is like a Wedge Volume (look visualization below) with the possibility to have different values for x width and 2 (non-zero) values for y.

Volume Wedge

A wedge is a triangular prism as shown in the diagram. Here the blue line extends along the positive z-axis and the red line extends along the x-axis.

Property	Type	Description
Dimensions	Hep3 Vector	1. Width of the prism in x 2. Open end height of the prism in y 3. Length of the prism in z

Volume Polycone

A polycone is a volume of rotation, defined by a number of points in r and z. The volume is found by a linear interpolation of the points.

Property	Type	Description
PolyconeType	string	Set to Fill to define a solid volume of rotation. Set to Cone to define a shell volume of rotation with an inner and outer surface.
FieldMapMode	string	The name of the file that contains the polycone data.

Volume Quadrupole

Quadrupoles are defined by an empty cylinder with four further cylinders that are approximations to pole tips.

Property	Type	Description
PhysicalLength	double	The length of the quadrupole container.
QuadRadius	double	The distance from the quad centre to the outside of the quad.
PoleTipRadius	double	The distance from the quad centre to the pole tip.
CoilRadius	double	

Property	Type	Description
CoilHalfWidth	double	
BeamlineMaterial	string	The material from which the beamline volume is made.
QuadMaterial	string	The material from which the quadrupole volume is made.

Volume Multipole

Multipoles are defined by an empty box with an arbitrary number of cylinders that are approximations to pole tips. Poles are placed around the centre of the box with n-fold symmetry. Multipoles can be curved, in which case poles cannot be defined – only a curved rectangular aperture will be present.

Property	Type	Description
ApertureCurvature	double	Radius of curvature of the multipole aperture. For now curved apertures cannot have poles. Set to 0 for a straight aperture.
ApertureLength	double	Length of the multipole aperture.
NumberOfPoles	int	Number of poles.
PoleCentreRadius	double	The distance from the centre of the aperture to the centre of the cylindrical pole.
PoleTipRadius	double	The distance from the centre of the aperture to the tip of the cylindrical pole.
ApertureInnerHeight	double	The inner full height of the aperture.
ApertureInnerWidth	double	The inner full width of the aperture.
AppertureOuterHeight	double	The outer full height of the aperture.
ApertureOuterWidth	double	The outer full width of the aperture.

Volume Boolean

Boolean volumes enable several volumes to be combined to make very sophisticated shapes from a number of elements. Elements can be combined either by union, intersection or subtraction operations. A union creates a volume that is the sum of two elements; an intersection creates a volume that covers the region where two volumes intersect each other; and a subtraction creates a volume that contains all of one volume except the region that another volume sits in.

Boolean volumes combine volumes modelled by other MiceModules (submodules), controlled using the properties listed below. Only the volume shape is used; position, rotation and field models etc are ignored. Materials, colours and other relevant properties are all taken only from the Boolean Volume's properties.

Note that unlike in other parts of MAUS, submodules for use in Booleans (BaseModule, BooleanModule1, BooleanModule2 ...) must be defined in a separate file, either defined in $MICEFILES/Models/Modules or in the working directory.

Also note that visualisation of boolean volumes is rather unreliable. Unfortunately this is a feature of GEANT4. An alternative technique is to use special virtual detectors to examine hits in boolean volumes.

Property	Type	Description
BaseModule	string	Name of the physical volume that the BooleanVolume is based on. This volume will be placed at (0,0,0) with no rotation, and all subsequent volumes will be added, subtracted or intersected with this one.
BooleanModule1	string	The first module to add. MAUS will search for the MiceModule with path $MICEFILES/Models/Modules/<BooleanModule1>.
BooleanModule1Type	string	The type of boolean operation to apply, either "Union", "Intersection" or "Subtraction".

Property	Type	Description
BooleanModule1Pos	Hep3 Vector	The position of the new volume with respect to the Base volume.
BooleanModule1Rot	Hep3 Vector	The rotation of the new volume with respect to the Base volume.
BooleanModuleN	string	Add extra modules as required. Replace "N" with the module number. N must be a continuous series incrementing by 1 for each new module. Note that the order in which modules are added is important – (A-B) ∪ C is different to A-(B ∪ C).
BooleanModuleNType	string	
BooleanModuleNPos	Hep3 Vector	
BooleanModuleNRot	Hep3 Vector	

Volume Sphere

A sphere is a spherical shell, with options for opening angles to make segments.

Property	Type	Description
Dimensions	Hep3 Vector	The x value defines the inner radius. The y value defines the outer radius of the shell. The z value is not used.
Phi	Hep3 Vector	The x value defines the start opening angle in phi. The y value defines the end opening angle. The z value is not used. Phi values must be in the range 0 to 360 degrees. If undefined, defaults to the range 0-360 degrees.
Theta	Hep3 Vector	The x value defines the start opening angle in theta. The y value defines the end opening angle. The z value is not used. Theta values must be in the range 0 to 180 degrees. If undefined, defaults to the range 0-360 degrees.

Repeating Modules

It is possible to set up a repeating structure for e.g. a repeating magnet lattice. The RepeatModule property enables the user to specify that a particular module will be repeated a number of times, with all properties passed onto the child module, but with a new position, orientation and scale factor. Each successive repetition will be given a translation and a rotation relative to the coordinate system of the previous repetition, enabling the construction of circular and straight accelerator lattices. Additionally, successive repetitions can have fields scaled relative to previous repetitions, enabling for example alternating lattices.

Property	Type	Description
RepeatModule	bool	Set to 1 to enable repeats in this module.
NumberOfRepeats	int	Number of times the module will be repeated in addition to the initial placement.
RepeatTranslation	Hep3 Vector	Translation applied to successive repeats, applied in the coordinate system of the previous repetition.
RepeatRotation	Hep3 Vector	Rotation applied to successive repeats, applied in the coordinate system of the previous repetition.
RepeatScaleFactor	double	ScaleFactor applied to successive repeats, applied relative to previous repetition's scale factor.

The RepeatModule2 property also enables the user to specify that a particular module will be repeated a number of times. In this case, MAUS will set a substitution variable @RepeatNumber that holds an index between 0 and NumberOfRepeats. This can be used in an expression in to provide a versatile interface between user and accelerator lattice.

Property	Type	Description
RepeatModule2	bool	Set to 1 to enable repeats in this module.
NumberOfRepeats	int	Number of times the module will be repeated in addition to the initial placement.

Beam Definition and Beam Envelopes

The Optics application can be used to track a trajectory and associated beam envelope through the accelerator structure. Optics works by finding the Jacobian around some arbitrary trajectory using a numerical differentiation. This is used to define a linear mapping about this trajectory, which can then be used to transport the beam envelope.

A beam envelope is defined by a reference trajectory and a beam ellipse. The reference trajectory takes its position and direction from the position and rotation of the module. If no rotation is defined the reference trajectory is taken along the z-axis. The magnitude of the momentum and the initial time of the reference trajectory is defined by properties. RF cavities are phased using the reference trajectory defined here.

The beam ellipse is represented by a matrix, which can either be set using

- Twiss-style parameters in (x, px), (y, py) and (t, E) spaces.
- Twiss-style parameters in (t, E) space and Penn-style parameters in a cylindrically symmetric (x, px, y, py) space.
- A 6x6 beam ellipse matrix where the ellipse equation is given by $\mathbf{X.T()MX} = 1$.

The Penn ellipse matrix is given by

$$M = \begin{pmatrix} \epsilon_L mc\frac{\beta_L}{p} & -\epsilon_L mc\alpha_L & 0 & 0 & 0 & 0 \\ & \epsilon_L mc\gamma_L p & \frac{D_x}{E}V(E) & \frac{D'_x}{E}V(E) & \frac{D_y}{E}V(E) & \frac{D'_y}{E}V(E) \\ & & \epsilon_T mc\frac{\beta_T}{p} & -\epsilon_T mc\alpha_T & 0 & -\epsilon_T mc(\frac{q}{2}\beta_T\frac{B_z}{P} - L) \\ & & & \epsilon_T mc\gamma_T p & \epsilon_T mc(\frac{q}{2}\beta_T\frac{B_z}{P} - L) & 0 \\ & & & & \epsilon_T mc\frac{\beta_T}{p} & -\epsilon_T mc\alpha_T \\ & & & & & \epsilon_L mc\gamma_T p \end{pmatrix}$$

Here L is a normalised canonical angular momentum, q is the reference particle charge, B_z is the nominal on-axis magnetic field, p is the reference momentum, m is the reference mass, ϵ_T is the transverse emittance, β_T and α_T are the transverse Twiss-like functions, ϵ_L is the longitudinal emittance and β_L and α_L are the longitudinal Twiss-like functions. Additionally D_x, D_y, D'_x and D'_y are the dispersions and their derivatives with respect to z and $V(E)$ is the variance of energy (given by the $(2, 2)$ term in the matrix above).

The Twiss ellipse matrix is given by

$$M = \begin{pmatrix} \epsilon_L mc\frac{\beta_L}{p} & -\epsilon_L mc\alpha_L & 0 & 0 & 0 & 0 \\ & \epsilon_L mc\gamma_L p & \frac{D_x}{E}V(E) & \frac{D'_x}{E}V(E) & \frac{D_y}{E}V(E) & \frac{D'_y}{E}V(E) \\ & & \epsilon_x mc\frac{\beta_x}{p} & -\epsilon_x mc\alpha_x & 0 & 0 \\ & & & \epsilon_x mc\gamma_x p & 0 & 0 \\ & & & & \epsilon_y mc\frac{\beta_y}{p} & -\epsilon_y mc\alpha_y \\ & & & & & \epsilon_y mc\gamma_y p \end{pmatrix}$$

Here p is the reference momentum, m is the reference mass, e_i, b_i and a_i are the emittances and Twiss functions in the (t,E), (x,p_x) and (y,p_y) planes respectively, D_x, D_y, D'_x, D'_y are the dispersions and their derivatives with respect to z and V(E) is the variance of energy (given by the (2,2) term in the matrix above).

Property	**Type**	**Description**
EnvelopeType	string	Set to *TrackingDerivative* to evolve a beam envelope in the Optics application.
BeamType	string	Set to *Random* to generate a beam using the parameters below for the Simulation application. Set to *Pencil* to generate a pencil beam (with no random distribution). Set to *ICOOL*, *Turtle*, *MAUS_PrimaryGenHit* or *G4BeamLine* to use a beam file.
Pid	int	The particle ID of particles in the envelope or beam.
Time	double	Set the time of the envelope reference trajectory
Longitudinal Variable	string	Set the longitudinal variable used to define the reference trajectory momentum. Options are *Energy*, *KineticEnergy*, *Momentum* and *ZMomentum*.
Energy	double	Define the value of the longitudinal variable used to calculate the mean momentum and energy. The usual relationship $E^2+p^2c^2=m^2c^4$ applies. Kinetic energy E_k is related to energy E by $E_k+m=E$.
KineticEnergy	double	
Momentum	double	
ZMomentum	double	
EllipseDefinition	string	Define the beam ellipse that will be used in calculating the evolution of the Envelope, or used to generate a beam for BeamType *Random*. Options are *Twiss*, *Penn* and *Matrix*.
The following properties are only used if EllipseDefinition is set to Twiss		
Emittance_X	double	Emittance in each 2d subspace, (x,px), (y,py) and (t,E).
Emittance_Y	double	
Emittance_L	double	
Beta_X	double	Twiss b function in each 2d subspace, (x,px), (y,py) and (t,E).
Beta_Y	double	
Beta_L	double	
Alpha_X	double	Twiss a function in each 2d subspace, (x,px), (y,py) and (t,E).
Alpha_Y	double	
Alpha_L	double	
The following properties are only used if EllipseDefinition is set to Matrix		
Covariance(t,t)	double	Set the 6x6 matrix that will be used in the to define the beam ellipse. Covariances should be covariances of elements of the matrix (x,Px,y,Py,t,E). This must be a positive definite matrix, i.e. determinant > 0. Note that this means that at least the 6 terms on the diagonal must be defined. Other terms default to 0.
Covariance(t,E)	double	
Covariance(t,x)	double	
...	double	
Covariance(Py,Py)	double	
The following properties are only used if EllipseDefinition is set to Penn		
Emittance_T	double	Transverse emittance for the 4d (x,px,y,py) subspace.
Emittance_L	double	Longitudinal emittance for the 2d (t,E) subspace.
Beta_T	double	Transverse beta for the 4d (x,px,y,py) subspace.
Beta_L	double	Longitudinal beta for the 2d (t,E) subspace.
Alpha_T	double	Transverse alpha for the 4d (x,px,y,py) subspace.
Alpha_L	double	Longitudinal alpha for the 2d (t,E) subspace.
Normalised AngularMomentu	double	Normalised angular momentum for the transverse phase space.

Property	Type	Description
Bz	double	Nominal magnetic field on the reference particle.
The following properties are used if EllipseDefinition is set to Penn or Twiss		
Dispersion_X	double	Dispersion in x (x-energy correlation).
Dispersion_Y	double	Dispersion in y (y-energy correlation).
DispersionPrime_X	double	D' in x (Px-energy correlation).
DispersionPrime_Y	double	D' in y (Py-energy correlation).
The following properties are only relevant for generating a beam envelope		
RootOutput	string	Output file name for writing output beam envelope in ROOT binary format.
LongTextOutput	string	Output file name for writing output beam envelope in string format.
ShortTextOutput	string	Output file name for writing output beam envelope in string format. This abbreviated output omits some of the fields that are present in LongTextOutput files.
BeamOutput	string	If a BeamType is defined, this property controls the file name to which beam data is written.
Delta_t	double	Offset in time used for calculating numerical derivatives. Default is 0.1 ns.
Delta_E	double	Offset in energy used for calculating numerical derivatives. Default is 1 MeV.
Delta_x	double	Offset in x position used for calculating numerical derivatives. Default is 1 mm.
Delta_Px	double	Offset in x momentum used for calculating numerical derivatives. Default is 1 MeV/c.
Delta_y	double	Offset in y position used for calculating numerical derivatives. Default is 1 mm.
Delta_Py	double	Offset in y momentum used for calculating numerical derivatives. Default is 1 MeV/c.
Max_Delta_t	double	Maximum offsets when polyfit algorithm is used. In some cases the offset can keep increasing without limit unless these maximum offsets are defined. Default is no limit.
Max_Delta_E	double	
Max_Delta_x	double	
Max_Delta_Px	double	
Max_Delta_y	double	
Max_Delta_Py	double	
The following properties are only relevant for generating a particle beam		
UseAsReference	Bool	If set to true and the datacard *FirstParticleIsReference* is set to 0, the first event in the Module will be used as the reference particle that sets cavity phases. This particle will then have the mean trajectory (i.e. no gaussian distribution).
BeamFile	string	If the BeamType is *ICOOL*, *Turtle, MAUS_PrimaryGenHit* or *G4BeamLine*, this property defines the name of the file containing tracks for MAUS.
NumberOfEvents	int	Set the maximum number of events to take from this module. If other modules are defined, MAUS will iterate over the modules until it the datacard *numEvts* is reached or all modules have been run to *NumberOfEvents*. Default is for MAUS to keep tracking from the first module it finds until *numEvts* is reached.

Optimiser

It is possible to define an optimiser for use in the Optics application. The optimiser enables the user to vary parameters in the MiceModule file and try to find some optimum setting. For each value of the parameters, MAUS Optics will calculate a score; the optimiser attempts to find a minimum value for this score.

Property	Type	Description
Optimiser	string	Controls the function used for optimising. For now Minuit is the only available option.

Property	**Type**	**Description**
Algorithm	string	For *Minuit* optimiser, controls the *Minuit* algorithm used. In general Simplex is a good option to use here. An alternative is Migrad. See Minuit documentation (for example at http://root.cern.ch/root/html/TMinuit.html) for further information. Minuit attempts to minimise the score function defined by the Score properties.
NumberOfTries	int	Maximum number of iterations MAUS will make in order to find the optimum value.
StartError	double	Guess at the initial error in the score.
EndError	double	Required final error in the score for the optimisation to converge successfully.
RebuildSimulation	bool	Set to False to tell MAUS not to rebuild the simulation on each iteration. This should be used to speed up the optimiser when a parameter is used that does not change the field maps. Default is true.
Parameter1_Start	double	Seed value for the parameter, that is used in the first iteration.
Parameter1_Name	string	Name of the parameter. This name is used as an expression substitution variable elsewhere in the code and should start with @. See Expression Substitutions above for details on usage of expression substitutions.
Parameter1_Delta	double	Estimated initial error on the parameter. Default is 1.
Parameter1_Fixed	bool	Set to true to fix the parameter (so that it is excluded from the optimisation). Default is false.
Parameter1_Min	double	If required, set to the minimum value that the parameter can hold.
Parameter1_Max	double	If required, set to the maximum value that the parameter can hold.
Parameter2_Start	...	Define an arbitrary number of parameters. Parameters must be numbered consecutively, and each parameter must have at least the start value and name defined. The optimiser will attempt to optimise against a score that is calculated by summing the Score1, Score2,... parameters on each iteration.
...	...	
Parameter2_Max	...	
Score1	double	
Score2	...	
...	...	

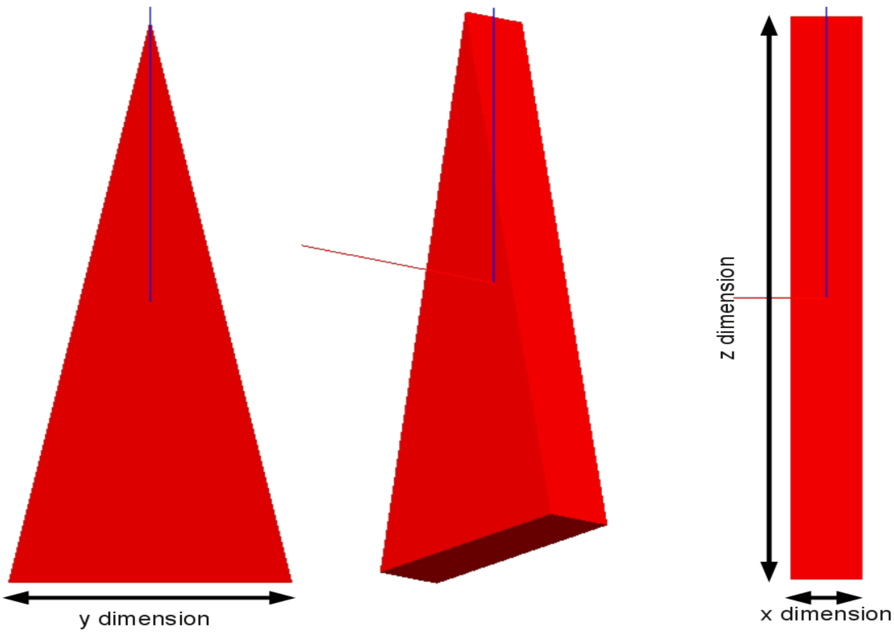

Figure 7.1: Schematic of the geometry of a Wedge volume.

Chapter 8

Field Properties

Invoke a field using PropertyString FieldType <fieldtype>. The field will be placed, normally centred on the MiceModule Position and with the appropriate Rotation. Further options for each field type are specified below, and relevant datacards are also given. If a fieldtype is specified some other properties must also be specified, while others may be optional, usually taking their value from defaults specified in the datacards. Only one fieldtype can be specified per module. However, fields from multiple modules are superimposed, each transformed according to the MiceModule specification. This enables many different field configurations to be simulated using MAUS.

To use BeamTools fields, datacard FieldMode Full must be set. This is the default.

Property	**Type**	**Description**
FieldType	string	Set the field type of the MiceModule.

FieldType CylindricalField

Sets a constant magnetic field in a cylindrical region symmetric about the z-axis of the module.

Property	**Type**	**Description**
ConstantField	Hep3 Vector	The magnetic field that will be placed in the region.
Length	double	The physical extent of the region.
FieldRadius	double	

FieldType RectangularField

Sets a constant magnetic field in a rectangular region.

Property	**Type**	**Description**
ConstantField	Hep3 Vector	The magnetic field that will be placed in the region.
Length	double	The physical extent of the region.
Width	double	
Height	double	

Property	Type	Description

FieldType Solenoid

MAUS simulates solenoids using a series of current sheets. The field for each solenoid is written to a field map on a rectangular grid and can then be reused. The field from each current sheet is calculated using the formula for current sheets detailed in MUCOOL Note 281, *Modeling solenoids using coil, sheet and block conductors*.

Property	Type	Description
FileName	string	Read or write solenoid data to the filename. If different modules have the same filename, MAUS assumes they are the same.
FieldMapMode	string	If set to Read, MAUS will attempt to read existing data from the FileName. If set to Write, MAUS will generate new data and write it to the FileName. If set to Analytic, MAUS will calculate fields directly without interpolating. If set to WriteDynamic acts as in Write except the grid extent and grid spacing at each point is chosen dynamically to some tolerance defined in the FieldTolerance property. Takes default from datacard SolDataFiles (Write).
Length	double	Coil physical parameters. Only used in Write/Analytic mode where they are mandatory.
Thickness	double	
InnerRadius	double	
CurrentDensity	double	
ZExtentFactor	double	Field map extends to length + ZExtentFactor*innerRadius in Write mode. Takes default from datacard SolzMapExtendFactor (10.). Map size is chosen dynamically in WriteDynamic mode.
RExtentFactor	double	Field map extends to radius RExtentFactor*innerRadius in Write mode. Takes default from datacard SolrMapExtendFactor (2.018...). Avoid allowing grid nodes to fall on sheets.
NumberOfZCoords	int	Number of coordinates in z in field map grid in Write mode. Takes default from datacard NumberNodesZGrid (500).
NumberOfRCoords	int	Number of coordintes in r in field map grid in Write mode. Takes default from datacard NumberNodesRGrid (100).
NumberOfSheets	int	Number of sheets used to calculate the field. Takes default from datacard DefaultNumberOfSheets (10).
FieldTolerance	double	Mandatory when FieldMapMode is WriteDynamic. If field map mode is write dynamic, this datacard controls the tolerance on errors in the field with which the field grid and the grid extent will be chosen.
Interpolation Algorithm	string	Choose the interpolation algorithm. Options are BiLinear for a linear interpolation in r and z, or LinearCubic for a linear interpolation in r and a cubic spline in z. Default is LinearCubic.
IsAmalgamated	bool	Set to 1 to add the coil to CoilAmalgamtion parent field (see below).

FieldType FieldAmalgamation

During tracking, MAUS stores a list of fields and for each one MAUS checks to see if tracking is performed through a particular field map's bounding box. This can be slow if a large number of fields are present. One way to speed this up is to store contributions from many coils in a single CoilAmalgamation. A CoilAmalgamation searches through child modules for solenoids with PropertyBool IsAmalgamated set to true. If it finds such a coil, it will add the field generated by the solenoid to its own field map and disable the coil.

Property	Type	Description
Length	double	The Length of the field map generated by the CoilAmalgamation.
RMax	double	The maximum radius of the field map generated by the CoilAmalgamation.
Interpolation Algorithm	string	Choose the interpolation algorithm. Options are BiLinear for a linear interpolation in r and z, or LinearCubic for a linear interpolation in r and a cubic spline in z. Default is LinearCubic.
ZStep	double	Step size of the field map generated by the CoilAmalgamation.
RStep	double	

FieldType DerivativesSolenoid

This is an alternative field model for solenoids that uses a power law expansion of the on-axis magnetic field and its derivatives, and an exponential fall-off for the fringe field. The fringe field is defined in the same way as other end fields, but note that HardEdged end field type is not available for solenoids and will result in an error.

Property	Type	Description
PeakField	double	Nominal peak field of the solenoid.
ZMax	double	Maximum z-half length of the solenoid bounding box in the local coordinate system of the magnet.
RMax	double	Maximum radius of the solenoid bounding box in the local coordinate system of the magnet.
MaxEndPole	int	Maximum derivative used in calculating the end field of the solenoid.

Phasing Models

MAUS has a number of models for phasing RF cavities.

When CavityMode is Unphased, MAUS attempts to phase the cavity itself. When using CavityMode Unphased MAUS needs to know when particles enter, cross the middle, and leave cavities. To phase a cavity, MAUS builds a virtual detector in the centre of the cavity that is used for phasing and then fires a reference particle through the system. Stochastic processes are always disabled during this process, while mean energy loss can be disabled using the datacard ReferenceEnergyLossModel. If a reference particle crosses a plane through the centre of a cavity, it sets the phase of the cavity to the time at which the particle crosses.

The field of the cavity during phasing is controlled by the property Field-DuringPhasing. There are four modes:

- *None*: Cavity fields are disabled during phasing
- *Electrostatic*: An electrostatic field with no positional dependence given by PeakEField*sin(ReferenceParticlePhase) is enabled during phasing.
- *TimeVarying*: A standard time varying field is applied during phasing, initially with arbitrary phase relative to the reference particle. MAUS uses a Newton-Raphson method to find the appropriate reference phase iteratively, with tolerance set by the datacard PhaseTolerance.
- *EnergyGainOptimised*: A standard time varying field is applied during phasing, initially with arbitrary phase and peak field relative to the reference particle. MAUS uses a 2D Newton-Raphson method to find the appropriate reference phase and peak field iteratively, so that the reference particle crosses the cavity centre with phase given by property ReferenceParticlePhase and gains energy over the whole cavity given by

property EnergyGain with tolerances set by the datacards PhaseTolerance and RFDeltaEnergyTolerance.

Tracking Stability Around RF Cavities

Usually RF cavities have little or no fringe field, and this can lead to some instability in the tracking algorithms. When MAUS makes a step into an RF cavity volume, the tracking algorithms tend to smooth out a field in a non-physical way. This can be prevented by either (i) making the step size rather small in the RF cavity or (ii) forcing MAUS to stop tracking by adding a physical volume at the entrance of the RF cavity (a window, typically made of vacuum). Doing this should improve stability of tracking.

FieldType PillBox

Sets a PillBox field in a particular region. MAUS represents pillboxes using a sinusoidally varying TM010 pill box field, with non-zero field vector elements given by

$$B_\phi = J_1(k_r r)\cos(\omega t)$$
$$E_z = J_0(k_r r)\cos(\omega t)$$

Here J_n are Bessel functions and k_r is a constant. See, for example, SY Lee VI.1. All other fields are 0.

Property	Type	Description
Length	double	Length of the region in which the field is present.
CavityMode	string	Phasing mode of the cavity - options are Phased, Unphased and Electrostatic.
FieldDuringPhasing	string	Controls the field during cavity phasing – options are None, Electrostatic, TimeVarying and EnergyGainOptimised.
EnergyGain	double	WhenFieldDuringPhasing is set to EnergyGainOptimised, controls the peak electric field.
Frequency	double	The cavity frequency.
PeakEField	double	The peak field of the cavity. Not used when the FieldDuringPhasing is EnergyGainOptimised.
TimeDelay	double	In Phased mode the time delay (absolute time) of the cavity.
PhasingVolume	string	Set to SpecialVirtual to make the central volume a special virtual.
ReferenceParticle Energy	double	In Electrostatic mode, MAUS calculates the peak field and the field the reference particle sees using a combination of the reference particle energy, charge and phase. Take defaults from datacards NominalKineticEnergy and MuonCharge
ReferenceParticle Charge	double	
ReferenceParticle Phase	double	MAUS tries to phase the field so that the reference particle crosses the cavity at ReferenceParticlePhase (units are angular). 0^o corresponds to no energy gain, 90^o corresponds to operation on-crest. Default from datacard rfAcclerationPhase.

FieldType RFFieldMap

Sets a cavity with an RF field map in a particular region. RFFieldMap uses the same phasing algorithm as described above.

Property	Type	Description
Length	double	Length of the region in which the field is present.
CavityMode	string	Phasing mode of the cavity - options are Phased and Unphased. RFFieldMaps cannot operated in Electrostatic mode.

Property	Type	Description
FieldDuringPhasing	string	Controls the field during cavity phasing – options are None, Electrostatic, TimeVarying and EnergyGainOptimised.
EnergyGain	double	WhenFieldDuringPhasing is set to EnergyGainOptimised, controls the peak electric field.
Frequency	double	The cavity frequency.
PeakEField	double	The peak field of the cavity. Not used when the FieldDuringPhasing is EnergyGainOptimised.
TimeDelay	double	In Phased mode the time delay (absolute time) of the cavity.
PhasingVolume	string	Set to SpecialVirtual to make the central volume a special virtual.
ReferenceParticle Energy	double	In Electrostatic mode, MAUS calculates the peak. field and the field the reference particle sees using a combination of the reference particle energy, charge and phase. Take defaults from datacards NominalKineticEnergy and MuonCharge
ReferenceParticle Charge	double	
ReferenceParticle Phase	double	MAUS tries to phase the field so that the reference particle crosses the cavity at ReferenceParticlePhase (units are angular). 0^{o} corresponds to no energy gain, 90^{o} corresponds to operation on-crest. Default from datacard rfAcclerationPhase.
FileName	string	The file name of the field map file.
FileType	string	The file type of the field map. Only supported option is SuperFishSF7.

FieldType Multipole

Creates a multipole of arbitrary order. Fields are generated using either a hard edged model, with no fringe fields at all; or an Enge model similar to ZGoubi and COSY. In the former case fields are calculated using a simple polynomial expansion. In the latter case fields are calculated using the polynomial expansion with an additional exponential drop off. Fields can be superimposed onto a bent coordinate system to generate a sector multipole with arbitrary fixed radius of curvature.

Unlike most other field models in MAUS, the zero position corresponds to the center of the entrance of the multipole; and the multipole extends in the +z direction.

The method to define end fields is described in the section EndFieldTypes below

Property	Type	Description
Pole	int	The reference pole of the magnet. 1=dipole, 2=quadrupole, 3=sextupole etc.
FieldStrength	double	Scale the field strength in the good field region. For dipoles, this sets the dipole field; for quadrupoles this sets the field gradient. Note that for some end field settings there can be no good field region (e.g. if the end length is >˜ centre length).
Height	double	Height of the field region.
Width	double	Width or delta radius of the field region.
Length	double	Length of the field along the bent trajectory.
EndFieldType	string	Set to HardEdged to disable fringe fields. Set to Enge or Tanh to use those models, as described elsewhere. Default is HardEdged.
CurvatureModel	string	Choose the model for curvature. Straight forces no curvature. Constant gives a constant radius of curvature; StraightEnds gives a constant radius of curvature along the length of the multipole with straight end fields beyond this length. MomentumBased gives radius of curvature determined by a momentum and a total bending angle.

Property	Type	Description
ReferenceCurvature	double	Radius of curvature of the magnet in Constant or StraightEnds mode. Set to 0 for a straight magnet. Default is 0.
ReferenceMomentum	double	Reference momentum used to calculate the radius of curvature of a dipole in MomentumBased mode. Default is 0.
BendingAngle	double	The angle used to calculate the radius of curvature of a dipole in MomentumBased mode. Note that this is mandatory in MomentumBased mode.

FieldType CombinedFunction

This creates superimposed dipole, quadrupole and sextupole fields with a common radius of curvature. The field is intended to mimic the first few terms in a multipole expansion of a field like

$$B(y=0) = B_0 \left(\frac{r}{r_0}\right)^k$$

The field index is a user defined parameter, while the dipole field and radius of curvature can either be defined directly by the user or defined in terms of a reference momentum and total bending angle. Fields are calculated as in the multipole field type defined above.

Property	Type	Description
Pole	int	The reference pole of the magnet. 1=dipole, 2=quadrupole, 3=sextupole etc.
BendingField	double	The nominal dipole field B_0. Note that this is mandatory in all cases except where CurvatureModel is MomentumBased, when the BendingAngle and ReferenceMomentum is used to calculate the dipole field instead.
FieldIndex	double	The field index k.
Height	double	Height of the field region.
Width	double	Width or delta radius of the field region.
Length	double	Length of the field along the bent trajectory.
EndFieldType	string	Set to HardEdged to disable fringe fields. Set to Enge or Tanh to use those models, as described elsewhere. Default is HardEdged.
CurvatureModel	string	Choose the model for curvature. Straight forces no curvature. Constant gives a constant radius of curvature; StraightEnds gives a constant radius of curvature along the length of the multipole with straight end fields beyond this length. MomentumBased gives radius of curvature determined by a momentum and a total bending angle.
ReferenceCurvature	double	Radius of curvature of the magnet in Constant or StraightEnds mode. Set to 0 for a straight magnet. Default is 0.
ReferenceMomentum	double	Reference momentum used to calculate the radius of curvature of a dipole in MomentumBased mode. Default is 0.
BendingAngle	double	The angle used to calculate the radius of curvature of a dipole in MomentumBased mode. Note that this is mandatory in MomentumBased mode.

EndFieldTypes

In the absence of current sources, the magnetic field can be calculated from a scalar potential using the standard relation

$$\vec{B} = \nabla V_n$$

The scalar magnetic potential of the n^{th}-order multipole field is given by

$$V_n = \sum_{q=0}^{q_m} \sum_{m=0}^{n} n!^2 \frac{G^{(2q)}(s)(r^2+y^2)^q \sin(\frac{m\pi}{2}) r^{n-m} y^m}{4^q q!(n+q)!m!(n-m)!}$$

where *G(s)* is either the double Enge function,

$$G(s) = E[(x - x_0)/\lambda] + E[(-x - x_0)/\lambda] - 1$$
$$E(s) = \frac{B_0}{R_0^n} \frac{1}{1 + \exp(C_1 + C_2 s + C_3 s^2 + ...)}$$

or G(s) is the double tanh function,

$$G(s) = \tanh[(x + x_0)/\lambda]/2 + \tanh[(x - x_0)/\lambda]/2$$

and *(r, y, s)* is the position vector in the rotating coordinate system. Note that here s is the distance from the nominal end of the field map.

Property	**Type**	**Description**
EndFieldType	string	Set to HardEdged to disable fringe fields. Set to Enge or Tanh to use those models, as described elsewhere. Default is HardEdged.
The following properties are used for EndFieldType Tanh		
EndLength	double	Set the l parameter that defines the rapidity of the field fall off.
CentreLength	double	Set the x_0 parameter that defines the length of the flat field region.
MaxEndPole	int	Set the maximum pole that will be calculated – should be larger than the multipole pole.
The following properties are used for EndFieldType Enge		
EndLength	double	Set the l parameter that defines the rapidity of the field fall off.
CentreLength	double	Set the x_0 parameter that defines the length of the flat field region.
MaxEndPole	int	Set the maximum pole that will be calculated – should be larger than the multipole pole.
Enge1	double	Set the parameters C_i as defined in the Enge function above.
Enge2	double	
...	double	
EngeN	double	

FieldType MagneticFieldMap

Reads or writes a magnetic field map in a particular region. Two sorts of field maps are supported; either a 2d field map, in which cylindrical symmetry is assumed, or a 3d field map.

For 2d field maps, MAUS reads or writes a file that contains information about the radial and longitudinal field components. This is intended for solenoidal field maps where only radial and longitudinal field components are present. Note that in write mode, MAUS assumes cylindrical symmetry of the fields. In this case, MAUS writes the x and z components of the magnetic field at points on a grid in x and z. Fields with an electric component are excluded from this summation.

For 3d field maps, MAUS reads a file that contains the position and field in cartesian coordinates and performs a linear interpolation. This requires quite large field map files; the file size can be slightly reduced by using certain symmetries, as described below. It is currently not possible to write 3d field maps.

Property	**Type**	**Description**
FieldMapMode	string	Set to Read to read a field map; and Write to write a field map.
FileName	string	The file name that is used for reading or writing.
FileType	string	The file format. Supported options in Read mode are MAUStext, MAUSbinary, g4beamline, icool, g4bl3dGrid. Only MAUStext is supported in Write mode. Default is MAUStext.

Property	Type	Description
Symmetry	string	Symmetry option for g4bl3dGrid file type. Options are None, Dipole or Quadrupole. None uses the field map as is, while Dipole and Quadrupole reflect the octant between the positive x, y and z axes to give an appropriate field for a dipole or quadrupole.
ZStep	double	Step size in z and r. Mandatory in Write mode but not used in Read mode (where step size comes from the map file).
RStep	double	
ZMin	double	Upper and lower bounds in z and r. Mandatory in Write mode but not used in Read mode (where boundaries come from the map file).
ZMax	double	
RMin	double	
RMax	double	

Some file formats are described below. I am working towards making the file format more generic and hence possibly easier to use, but backwards compatibility will hopefully be maintained.

MAUStext Field Map Format

The native field map format used by MAUS in text mode is described below.

```
# GridType = Uniform N = number_rows
# Z1 = z_start Z2 = z_end dZ = z_step
# R1 = r_start R2 = r_end dR = r_step
Bz_Values  Br_Values
...    ...
<Repeat as necessary>
```

In this mode, field maps are represented by field values on a regular 2d grid that is assumed to have solenoidal symmetry, i.e. cylindrical symmetry with no tangential component.

Name	Type	Description
`number_rows`	double	Number of rows in the field map file.
`z_start`	double	Position of the grid start along the z axis.
`z_end`	double	Position of the grid end along the z axis.
`z_step`	double	Step size in z.
`r_start`	double	Position of the grid start along the r axis.
`r_end`	double	Position of the grid end along the r axis.
`r_step`	double	Step size in r.
`Bz_Values`	double	*Bz* field value.
`Br_Values`	double	*Br* field value.

g4bl3dGrid Field Map Format

The file format for 3d field maps is a slightly massaged version of a file format used by another code, g4beamline. In this mode, field maps are represented by field values on a regular cartesian 3d grid.

```
number_x_points number_y_points number_z_points global_scale
1 X [x_scale]
2 Y [y_scale]
3 Z [z_scale]
4 BX [bx_scale]
5 BY [by_scale]
6 BZ [bz_scale]
0
X_Values   Y_Values  Z_Values  Bx_values  By_values  Bz_values
```

```
...    ...    ...    ...    ...    ...
<Repeat as necessary>
```

where text in bold indicates a value described in the following table

Name	Type	Description
number_x_points	double	Number of points along x axis.
number_y_points	double	Number of points along y axis.
number_z_points	double	Number of points along z axis.
global_scale	double	Global scale factor applied to all x, y, z and Bx, By, Bz values.
x_scale	double	Scale factor applied to all x values.
y_scale	double	Scale factor applied to all y values.
z_scale	double	Scale factor applied to all z values.
bx_scale	double	Scale factor applied to all Bx values.
by_scale	double	Scale factor applied to all By values.
bz_scale	double	Scale factor applied to all Bz values.
X_Values	double	List (column) of each x value.
Y_Values	double	List (column) of each y value.
Z_Values	double	List (column) of each z value.
Bx_Values	double	List (column) of each Bx value.
By_Values	double	List (column) of each By value.
Bz_Values	double	List (column) of each Bz value.

Chapter 9

TOF Detector

This chapter describes the time-of-flight (TOF) simulation and reconstruction software. The simulation is designed to produce digits similar to "real data" and the reconstruction is agnostic about whether the digits are from simulation or data acquisition.

Simulation

- **Geometry**
  For the most upstream TOF – TOF0 – to be simulated, it is essential that the z where the beam starts be upstream of the detector.

  In the standard Step VI geometry as described in `Stage6.dat`, this is at `-14200 mm` and for the Step IV geometry described in `Stage4.dat` it is at `2773 mm`

  The internal geometry of the TOF detector and the positioning of the slabs are defined in the MiceModules represenation . The numbering convention is the same as that for the DAQ and is described in MICE-Notes 251 and 286. It is worth keeping in mind the plane numbering convention since the current naming scheme is suboptimal:

  - `station` refers to the TOF station – TOF0, TOF1, TOF2
  - `plane` refers to the horizontal/vertical planes within a station
  - `plane 0` means horizontal slabs – slabs are oriented horizontally. They measure y
  - `plane 1` means vertical slabs – slabs are oriented vertically. They measure x

  The z locations of TOF0 and TOF1 are specified in the `Beamline.dat` file and the z of TOF2 is specified in the main geometry description file, for e.g. `Stage6.dat`

- **Hits**
  GEANT hits are generated for all tracks which pass through a TOF slab. "True" TOF hits are described by the `MAUS::Hit` class and contain the GEANT4 information prior to digitization. The members of the class are listed below.

Table 9.1: True TOF hit class members.
The GEANT TOF hits are encoded with the following information.

Name	Meaning
`channel_id`	Class `TOFChannelId*` contains `station,plane,slab`
`energy_deposited`	`double` – energy deposited by track in the slab
`position`	`ThreeVector` – x, y, z of hit at the slab
`momentum`	`ThreeVector` – p_x, p_y, p_z of particle at slab
`time`	`double` – hit time
`charge`	`double` – PDG charge of particle that produced this hit
`track_id`	`G4Track` – ID of the GEANT track that produced this hit
`particle_id`	`ThreeVector` – PDG code of the particle that produced this hit

Digitization

Each GEANT hit in the TOF is associated with a slab based on the geometry described in the TOF MiceModules. If a hit's position does not correspond to a physical slab (for instance if the hit is outside the fiducial volume) the hit is not digitized. The energy deposited in the slab and the hit time are then digitized as described below.

- **Charge digitization** The energy deposited by a hit in a slab is first converted to units of photoelectrons. The photoelectron yield from a hit is attenuated by the distance from the hit to the PMT, then smeared by the photoelectron resolution. The yields from all hits in a given slab are then added and the summed photoelectron yield is converted to ADC (In principle, this should be done not on an event-by-event basis but rather on a trigger-basis. In the absence of a real trigger, all hits in a slab are now merged)
- **Time digitization** The hit time is propogated to the PMTs at either end of the slab. The speed of light in the scintillator, based on earlier calibration, is controlled by the `TOFscintLightSpeed` data card. The time is then smeared by the PMT time resolution and converted to TDC.

After converting the energy deposit to ADC and the time to TDC, the TDC values are "uncalibrated" so that at the reconstruction stage they can be corrected just as is done with real data.

The data cards that control the digitization are listed in Table 9.2.

NOTE: Do not modify the default values.

Reconstruction

The reconstruction software treats both data and Monte Carlo the same way. In the case of real data, the input to the reconstruction chain is TOF Digits (`MapCppTOFDigit`) and in the case of Monte Carlo it is the digitized information from `MapCppTOFMCDigitizer`.

- **Digits** (`MapCppTOFDigit,MapCppTOFMCDigitizer`) Digits are formed from the V1724 ADCs and V1290 TDCs.

Table 9.2: Data cards for TOF digitization.

Name	Meaning	Default
TOFconversionFactor	conversion	0.005 MeV
TOFpmtTimeResolution	resolution for smearing the PMT time	0.1 ns
TOFattenuationLength	light attenuation in slabs	1400 mm
TOFadcConversionFactor	conversion from charge to ADC	0.125
TOFtdcConversionFactor	conversion from time to TDC	0.025
TOFpmtQuantumEfficiency	PMT collection efficiency	0.25
TOFscintLightSpeed	propogation speed in slab	170 mm/ns

- **Slab Hits** (MapCppTOFSlabHits) The SlabHits routine takes individual PMT digits and associates them to reconstruct the hit in the slab. All PMT digits are considered. If there are multiple hits associated with a PMT, the hit which is earliest in time is taken to be the real hit. Then, if both PMTs on a slab have hits, the SlabHit is formed. The TDC values are converted to time (ToftdcConversionFactor) and the hit time and charge associated with the slab hit are taken to be the average of the two PMT times and charges respectively. In addition, the charge product of the PMT charges is also formed.

- **Space Points** (MapCppTOFSpacePoints) A space point pixel in the TOF is a combination of x and y slab hits. All combinations of x and y slab hits in a given station are considered. If the station is a trigger station, an attempt is made to find the "trigger pixel" – i.e. the x, y combination that triggered this event. This is done by applying calibration corrections to the slab hits, and then asking if the average time in this pixel is consistent with the trigger within some tolerance. In other words, if t_x and t_y are the times corresponding to the x and y slab hits, is $\frac{t_{x,calib}+t_{y,calib}}{2} < t_{triggercut}$? If no x, y combination produces a trigger pixel, the space point reconstruction stops and no space points are formed. This is because to apply the calibration corrections to the slab hit times, it is essential know the trigger pixel.

  Once a trigger pixel is found, all x, y slab hit combinations are again treated as space point candidates. The calibration corrections are applied to these hit times. If $\mid t_x - t_y \mid$ is consistent with the resolution of the detector, the combination is said to be a space point. The space point thus formed contains the following information

This is used by the reconstuction of the TOF detectors| #TOF_cabling_file = "/files/cabling/TOFChannelMap.txt"
#TOF_TW_calibration_file = "/files/calibration/tofcalibTW_dec2011.txt"
#TOF_T0_calibration_file = "/files/calibration/tofcalibT0_trTOF1_dec2011.txt"
#TOF_T0_calibration_file = "/files/calibration/tofcalibT0_trTOF0.txt"
#TOF_Trigger_calibration_file = "/files/calibration/tofcalibTrigger_trTOF1_dec2011.txt"
#TOF_Trigger_calibration_file = "/files/calibration/tofcalibTrigger_trTOF0.txt"
the date for which we want the cabling and calibration # date can be 'current' or a date in YYYY-MM-DD
#TOF_calib_date_from = 'current' TOF_calib_date_from = '2010-08-10 00:00:00'

Table 9.3: TOFSpacePoint class members.

Name	Meaning
`pixel_key`	`string` encoded with the TOF station,plane,slab
`slabY`	`int` encoded with the TOF station,plane,slab
`slabX`	`int` encoded with the TOF station,plane,slab
`time`	`double` – calibrated space point time
`charge`	`int` – average of the charges of the constitutent slabs
`charge_product`	`int` – average of charge products of the constitutent slabs
`dt`	`double` – time difference between the x and y slabs = resolution

Table 9.4: Data cards for TOF reconstruction.

Name	Meaning	Default
`TOF_trigger_station`	conversion	`0.005 MeV`
`TOF_findTriggerPixelCut`	resolution for smearing the PMT time	`0.1 ns`
`TOF_makeSpacePiontCut`	PMT collection efficiency	`0.25`
`Enable_t0_correction`	light attenuation in slabs	`1400 mm`
`Enable_triggerDelay_correction`	conversion from charge to ADC	`0.125`
`Enable_timeWalk_correction`	conversion from time to TDC	`0.025`

Table 9.5: Data cards for accessing calibrations from CDB.

Name	Meaning	Default
TOF_calib_date_from	conversion	'2010-08-10 00:00:00'
TOF_cabling_date_from	conversion	current

TOF_cabling_date_from = 'current' Enable_timeWalk_correction = True Enable_triggerDelay_correction = True Enable_t0_correction = True

Database

- Constants in the CDB
- Datacards
- Routines to access