CombLayer Guide

Stuart Ansell

December 4, 2024

Contents

Nomenclature

CADIS   Consistent Adjoint Driven Importance Sampling

1 Introduction

CombLayer is designed to facilitate the rapid production of complex MCNP(X) models that depend on a long list of ranged variables and a number of module flags. It is also intended to help with placement of tallies, maintaining consistant material files and some variance reduction.

1.1 Coding Convensions

CombLayer has some coding convensions beyond the standard Scott Myers Efficient C++ convensions [**?**]. These are typcially there for two reasons (i) that in a model-build system, a rapid build time is essential since it is nearly impossible to have a sub-test framework for any component as the whole MCNP(X) model is required to check if is it valid, (ii) the code is intended to be used without complete understanding. Therefore as much as possible, each component is independent without code repetition. Back-references are to be minimized both in the run-time calling path and in the code build dependencies.

1.1.1 Include files

Include files (.h) are forbidden to include other files. This does several things (a) it reduces the *dependency hell* where it is almost impossible to find the definition of a function and what it depends on. (b) optimization of the include tree can be carried out and dependency continuously observered.

Namespaces are a good method of removing global name pollution but many other C++ programs allows *using namespace X*, this is almost 100% forbidden except in the test for that particular namespace unit. This also applies to boost, stl, tr1 etc, to which helps distinguish external functions and domains.

2 Layout

The basic program structure is given by figure 1. The main program structure is normally copied from an existing project and the areas are constructed by the user. It is normal flow is to to call functions that: (i) define new input options to enter parameters from the command line, (ii) variables that your project is going to use, (iii) build the geometry via a call to a makeProject functoin (iv) set up tallies (v) generate variance reduction.There are other ways to construct the system but this allows a degree of autonomy from tallies/variance reduction and producing an appropiate output.
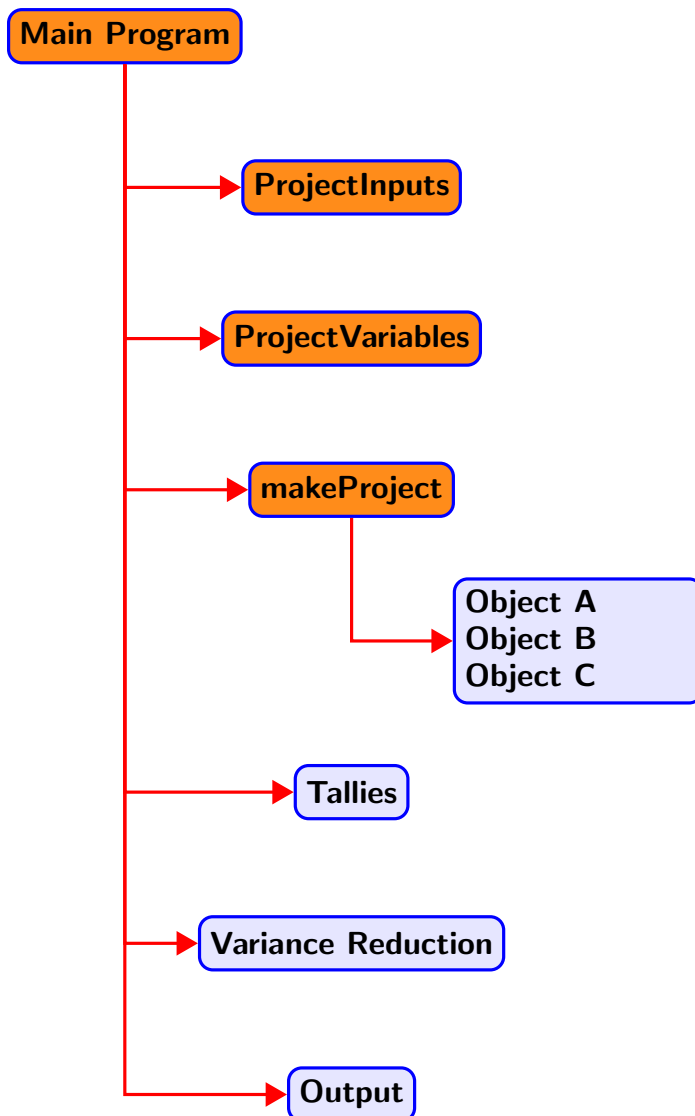

Figure 1: The main program calling sequence is shown. The parts in orange, are expected to be constructed by the user. Bespoke objects can be added for a project but it is not necessary.

2.1 Main

The main function for CombLayer follows a relatively linear template. Consider the example

```cpp
int
main(int argc, char* argv[])
{
  int exitFlag(0);                    // Value on exit
  ELog::RegMethod RControl("","main");
  mainSystem::activateLogging(RControl);
  std::string Oname;
  std::vector<std::string> Names;
  std::map<std::string,std::string> Values;

  // PROCESS INPUT:
  InputControl::mainVector(argc,argv,Names);
  mainSystem::inputParam IParam;
  createPipeInputs(IParam);

  Simulation* SimPtr=createSimulation(IParam,Names,Oname);
  if (!SimPtr) return -1;

  // The big variable setting
  setVariable::PipeVariables(SimPtr->getDataBase());
  InputModifications(SimPtr,IParam,Names);

  // Definitions section
  int MCIndex(0);
  const int multi=IParam.getValue<int>("multi");
  try
    {
      SimPtr->resetAll();

      pipeSystem::makePipe pipeObj;

      World::createOuterObjects(*SimPtr);
      pipeObj.build(SimPtr,IParam);
      SDef::sourceSelection(*SimPtr,IParam);

      SimPtr->removeComplements();
      SimPtr->removeDeadSurfaces(0);
      ModelSupport::setDefaultPhysics(*SimPtr,IParam);

      const int renumCellWork=tallySelection(*SimPtr,IParam);
      SimPtr->masterRotation();
      if (createVTK(IParam,SimPtr,Oname))
        {
          delete SimPtr;
          ModelSupport::objectRegister::Instance().reset();
          ModelSupport::surfIndex::Instance().reset();
          return 0;
        }

      if (IParam.flag("endf"))
        SimPtr->setENDF7();

      SimProcess::importanceSim(*SimPtr,IParam);
      SimProcess::inputPatternSim(*SimPtr,IParam);  // energy cut etc

      if (renumCellWork)
        tallyRenumberWork(*SimPtr,IParam);
      tallyModification(*SimPtr,IParam);
```

```
61        if (IParam.flag("cinder"))
62          SimPtr->setForCinder();
63
64        // Ensure we done loop
65        do
66          {
67            SimProcess::writeIndexSim(*SimPtr,Oname,MCIndex);
68            MCIndex++;
69          }
70        while(MCIndex<multi);
71
72        exitFlag=SimProcess::processExitChecks(*SimPtr,IParam);
73        ModelSupport::calcVolumes(SimPtr,IParam);
74        ModelSupport::objectRegister::Instance().write("ObjectRegister.txt");
75      }
76    catch (ColErr::ExitAbort& EA)
77      {
78        if (!EA.pathFlag())
79          ELog::EM<<"Exiting from "<<EA.what()<<ELog::endCrit;
80        exitFlag=-2;
81      }
82    catch (ColErr::ExBase& A)
83      {
84        ELog::EM<<"EXCEPTION FAILURE :: "
85                <<A.what()<<ELog::endCrit;
86        exitFlag= -1;
87      }
88    delete SimPtr;
89    ModelSupport::objectRegister::Instance().reset();
90    ModelSupport::surfIndex::Instance().reset();
91
92    return exitFlag;
93  }
94  \label{MainProg}
```

The Main program given in listing **??** highlights the areas that the user should be creating. The remainder of the main() function deals with trapping exceptions, loggin and building variance reduction and tallies into the model.

(i) **createPipeInputs** is a function to define which command line options [above the standard ones] this model should support. It doesn't do anything with them, just a list of options, number of arguements they can take and any default values that the options should take. All options defined here are access from the command line option with a - sign. E.g. *-r* as a renumber operation. In this form of the program, if the main program is run without any options, a list and very brief description of each option is shown (e.g. execute *./pipe*).

If no additional options are required, a call to *createInputs(IParam)* would be expected. Significant restructuring would need to take place to avoid that call.

(ii) setVariable::PipeVariables is the method that registers and sets a default value for all the variables that the model will use.

(iii) makePipe pipeObj and pipeObj.buid(SimPtr,IParam) are the main geometry building calls. Typically 100% of the geometry is built in this zone. It is not a place for tallies, variance reduction and other non-geometry items.

3 Installation

CombLayer is primarily designed for the Linux platform and requires a C++ compiler that supports C++11 or later. The code is available at https://github.com/SAnsell/CombLayer, either as a downloadable zip file or by cloning/pulling the Git repository:

```
git clone https://github.com/SAnsell/CombLayer.git
```

3.1 Requirments

CombLayer requires the GNU Scientific Library (GSL), boost::regex, as well as the {fmt} and STL libraries provided by your C++ compiler. The build process is managed with CMake, and the class reference documentation is generated using Doxygen. Both gcc and clang compilers are supported.

3.2 Basic build method

CombLayer can be built either in the source code directory or in a separate build directory. If built in the source directory, the command is

```
cmake ./
make
```

or for a separate build directory

```
cmake -B/path/to/buildDirectory -S/path/to/srcDirectory
make
```

This should make a number of executables, e.g. `ess`, `maxiv`, `reactor`, `fullBuild` etc, which can be used to make a model with commands listed in the `all.sh` script. For instance, the

```
singleItem --singleItem Octupole AA
```

command generates an output file named `AA1.x`, which is an MCNP model of an octupole magnet. To generate a FLUKA model, add the `-fluka` argument, and for PHITS models, use the `-phits` argument.

4 Link system

CombLayers geometry is composed of a set of objects that have slightly stronger rules than a typical MCNPX model. Obviously any MCNPX model can be represented as a CombLayer model and in the extreme case that is done by defining one object to contain the MCNPX model. However, the little benefit would be derived from such an approach.

The basic geometric system is to build a number of geometric classes and construct the model by incorporating those into the desired configuration. Each geometric class is designed to be built and an arbitary position and rotation, be of an undetermined number, and interact with its surroundings in a well defined manor.

In object orientated programming, functional rules and properties are normally added to objects by inherritance. CombLayer follows that pattern. As such most geometry item classes inherit from base classes within the `attachSystem` namespace.

4.1 AttachSystem Namespace

The CombLayer system is built around the interaction of FixedComp units, ContainedComp units and LinkUnits. The use of these and their interactions are the basic geometric building tools. These object reside within the attachSystem namespace.

Almost any geometric item can be designated as a FixedComp object. This is done by public inherriting from directly from the FixedComp, or by inherriting from one of the more specialised attachSystem objects e.g. Twin-Comp or LinearComp.

4.2 FixedComp

The basic FixedComp object holds the origin and the orthoganal basis set (X/Y/Z) for the geometry item being built. In addition it holds a number of LinkUnits which provide information about the outer (and/or inner) surfaces and positions on the geometric item.

As with all Object-Orientated (OO) constructions their is an implicit contract that the inherited object should adhere to. This is normally expressed as the *Liskov Substitution principle*: This principle states that functions that use pointers/references to the base object must be able to use the objects of derived classes without knowing it. In this case, that means that modification of the origin or the basis set should not invalidate it and that the object should do the expected thing. E.g. if the origin is shifted by 10 cm in the X direction the object should move by 10cm in the X direction. It also means that the basis set must remain orthogonal at all time.

Other than providing an origin and an basis set, the FixedComp has a number of link points. The link points are there to define joining surfaces, points and directions. Each link point defines all three parts.

For example a cube might have 6 linkUnits, and each linkUnit would have a point at the centre of a face, a direction that is normal to the face pointing outwards and a surface definition that is the surface pointing outwards. [Note that in the case that the link points define an inner volume, for example in a vacuum vessel, then the surfaces/normals should point towards the centre.]

The actual link surface does not need to be a simple surface. In the case, that an external surface needs multiple surfaces to define the external contact these can be entered into a link-rule. For example, if the cube above was replaces with a box with two cylindrical surfaces the link surface would be defined as the out going cylinder intersection with a plane choosing the side.

In the case of an equiry for the linkSurface (e.g. to do an line intersection) then it is the first surface that takes presidence. However, all actions can be carried out on the link-rule including line intersections etc.

4.3 ContainedComp

The ContainedComp defined both the external and interal enclosed volume of the geometric item. It is most often used to exclude the item from a larger enclosing geometric object: e.g. A moderator will be excluded from a reflector, or it can be used to exclude a part of the geometric item from another geometric object. E.g. two pipes which overlap can have one exclude itself from the other.

In CombLayer, the ContainedComp are considered the primary geometric item, i.e. it is the ContainedComp that is removed from the other items. However, it is used in a two stage process wereby cells are registered to be updated by the ContainedComp at a later date. This was to allow forward dependency planning but has more or less been superseded by the attachControl system.

5 Model Runtime control

C++ programs start from the main() function and in CombLayer the runtime control has been keep mostly in the main() function. Clearly that could be further refactored out but CombLayer lacks the sophisticated top level type abstraction that is required to do this in a generic way, so copy/pasted structure is used with variance to the particular model required. The sole advantage of the abscence of a top level abstraction is that the user is the freedom in writing new objects which allows other programs to be incorporated by making their main function a minor function and directly calling.

The structure of two example main()s will be compared from the units that exist with the stanard CombLayer distribution. That is *bilbau.cxx* and *reactor.cxx*. These build the delft reactor model and the Biblau low energy spallation source.

First part of the code is along list of #include's. They are the mainly dependency list of the objects *Simulation, weightManager, and tallySelector*. This can and should be copied at will. Do not make an file with them all in [see 1.1.1].

At the end of the include section there is typically, one or two model specific includes. These normally include *makeXXX.h* file and anything that they directly depend on. In the case of bilbau it is just *makeBib.h* whilst for reactor it is both *makeDelft.h* and *ReactorGrid.h*.

Tallies are the fundamental reason for running MCNPX. However, the manor in which MCNPX specifies tallies is not compatible with a vairiable defined model because in most cases the required tally is relative to an object whose position is unknown.

This problem has been addressed by allowing most tallies to use the FixedComp link system.

5.1 Tally System

The tally system is accessed either by a simple command line menu system, or via an XML file. The command line help system is very primative but can remind the user of the bais

5.2 Point Tally

Point tallies are fundamentally a 3D vector in space. In CombLayer, there are three levels of position available: (a) Real MCNP(X) output position, (b) CombLayer master origin position before master rotation, and (c) relative position to an object. Both (a) and (c) are well supported, however, to do option (b) there needs to be some real care with the layout of the calling sequence in the main() function. The fullBuild.cxx example is a suitable option to follow, but checking will be needed.

5.2.1 Free Point tally

The simplest way to put a point into CombLayer is to use a free point.

```
./prog −T point free 'Vec3D(300.0,10.0,5.0)' Output
```

This creates a point tally at (300,10,5) in the final output using neutron tallies with the default energy and time binning system.
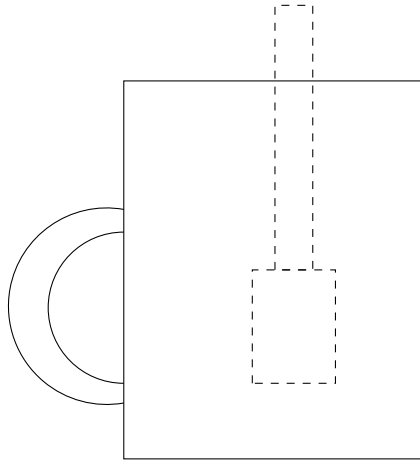
Figure 2: addToInsertForced: all cells of Spoon are excluded from all cells of Mug including non intersecting cells. In this example, the Spoon is excluded from the Mug handle even though they do not intersect.

5.3 Fluka Tally System

5.3.1 Surface Tally

The fluka surface tally is accessed via -T surface.

The

- Get real surface number by its relative number: SMap.realSurf(divIndex+103) (see createLinks methods)

5.4 How to put one object into another

Suppose, we are inserting Spoon into Mug. Mug is made up of N cells. Spoon is made of one contained component with outer surface. CombLayer provides several methods to put one object into another:

```
attachSystem :: addToInsertForced ( System ,    *Mug, *Spoon );
attachSystem :: addToInsertSurfCtrl ( System , *Mug, *Spoon );
attachSystem :: addToInsertControl ( System ,   *Mug, *Spoon );
attachSystem :: addToInsertLineCtrl ( System , *Mug, *Spoon );
```

5.4.1 addToInsertForced

The outer surface of the Spoon is excluded from the HeadRule of every single cell of Mug. Even if Mug contains cells which do not intersect with Spoon (e.g. its handle, see Fig. 2). *Forced* means *do it and do not think about it*, but at the same time it means that *I have got something wrong somewhere.* Normally this is that insufficient link points have been added to the object, or that the object is a set of split (single cell) volumes. However, there is the additional problem that the model may not be correctly constructed at this point, so that the other options seem not to work. This can be checked by adding a *SimProcess::writeIndexSim(System,"OutputFilename.txt",0);* in the code just before the call to insertForced. If there are undefined volumes then the model is not in a state that any of the *addToInsert* algorithms except *addToInsertForced* can be used.

(a) Cell 3 is excluded from cell 1; cells 2 and 4 not changed

(b) None of the cells are modified
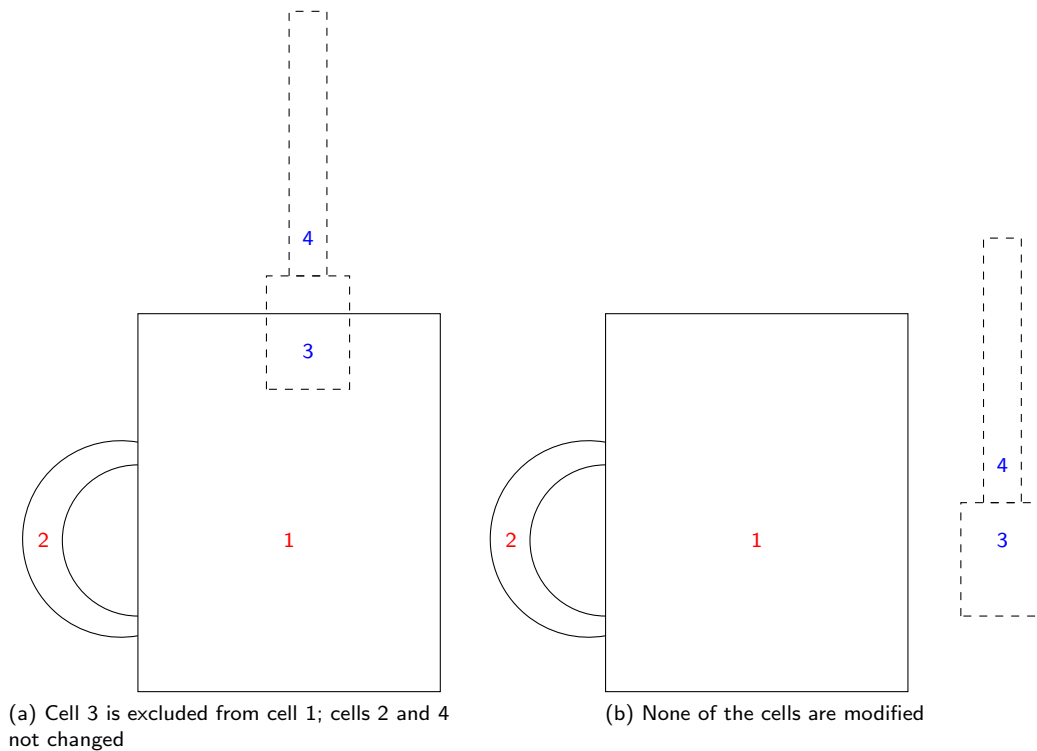
Figure 3: addToInsertSurfCtrl: only the needed cells are excluded

5.4.2 addToInsertSurfCtrl

The objective of this function is to use the surface intersections between Mug and spoon to determine which cells within Mug intersect the ContainedComp of spoon. The process is done on a cell - ContainedComp level.

The process is as follows:

1. Deconvolves both the Spoon's containedComponent boundary into surfaces.

2. Deconvolves the Mug into surfaces.

3. Triple Loops over surfaces of both Mug and Spoon

   (a) Calculate intersection of each surface:surface:surface triplet
   (b) If a point is within CC and the Mug Cells exclude the CC from the MCell

Thus the spoon is inserted only into those cells of Mug which it intersects (Fig. 3).

It is not always better to call `addToInsertSurfCtrl` instead of `addToInsertForced` in cases that if is certain that an intersection can must take place (particularly if the CC / Inserting cells have large numbers of surfaces).

`addToInsertSurfCtrl` is a very expensive function to call, because you have to check all the surface triplets. So, it runs significantly slower than addToInsertForced, but the geometry will be faster.

The two remaining methods provide similar functionality but with less computational overhead, however, there are cell constructs which will cause them to fail.

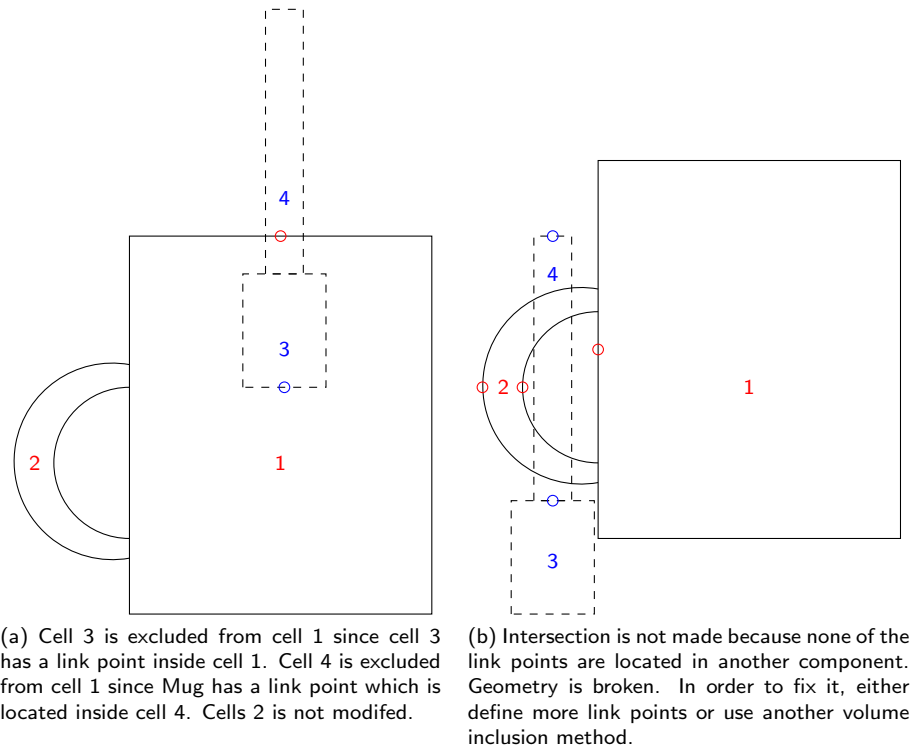(a) Cell 3 is excluded from cell 1 since cell 3 has a link point inside cell 1. Cell 4 is excluded from cell 1 since Mug has a link point which is located inside cell 4. Cells 2 is not modifed.

(b) Intersection is not made because none of the link points are located in another component. Geometry is broken. In order to fix it, either define more link points or use another volume inclusion method.

Figure 4: addToInsertControl: intersection relies on the link points

5.4.3 addToInsertControl

It's a very simple method. The link points from Spoon are uses as a test for each of the cells within Mug. The method checks if any of these link point fit inside each of the cells of Mug. If it does, then it cuts Spoon from the Mug cell (Fig. 4). It is possible to add a vector of link points to check as a parameter to limit the search.

5.4.4 addToInsertLineCtrl

If we have a (big) contained component (Mug) and some (small) object which clips it (Spoon) and we are not certain that the link points are contained within each Mug cell, then **addToInsertControl** can not be reliably used. However, addToInsertLineCtrl adds an additional check to addToInsertControl, because it constructs all the connecting lines between the link points, and if the line intersects the other cell, an intersection is made (Fig. 5).

6 Components

6.1 ObjectRegister

The objectRegister is a singleton object [it should be per simulation], which keeps each and then deletes when at its lifetime end, each object registered with it. It only accepts two types of object, a dummy name object and a FixedComp object.

If a dummy object is required, the name (and possibly number) of the object is provided and the objectRegister singleton provides a unique range of cell and surface numbers, typically 10,000 units of each, but can be user selected. This is its only responsibility and to ensure that the name is unique.

(a) Intersection is not made because the line connecting two link points intersects cell 4 outside of a cell belonging to the Mug

(b) Cell 4 is excluded from cell 2 since at least one line connecting two link points intersects the Mug handle at a point which is valid within the spoon (shown by red filled circles)
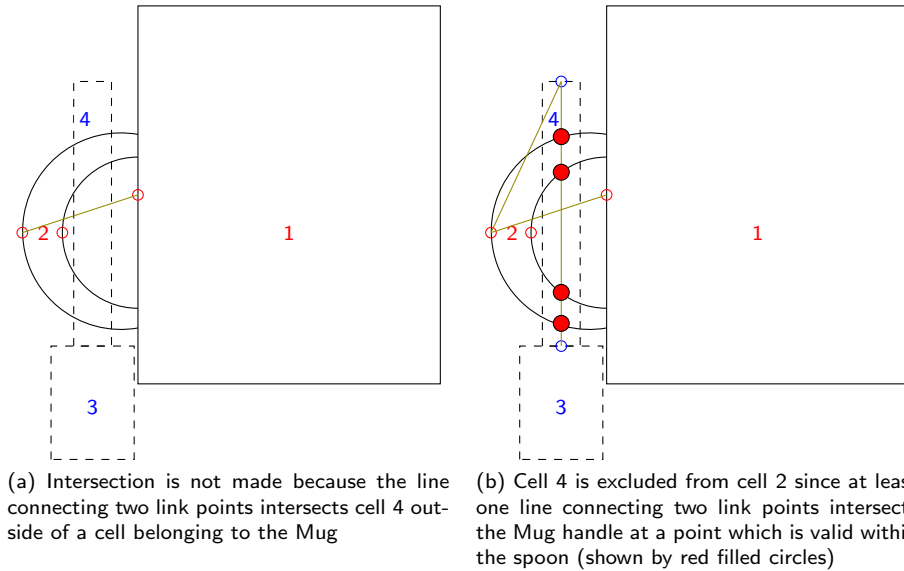
Figure 5: addToInsertLineCtrl: intersection relies on the lines connecting link points (not all link points and connections are shown).

Significantly more complex is the FixedComp registration, in this case a *std::shared_ptr* of FixedComp must be provided by the calling method. Obviously, for a shared_ptr the object memory must be allocated, i.e. an initial `new object(...)` is normally called directly or previously. A typical structrue might be:

```
std::shared_ptr<BeamPipe> A  = new BeamPipe("LongPipe");

ModelSupport::objectRegister& OR=
    ModelSupport::objectRegister::Instance();

OR.addObject(A);
```

From this example, the BeamPipe class is inherrited from FixedComp, this is manditory. A tempory reference *OR* is created by calling the static Instance() method. All singletons in CombLayer provide an Instance() method for this purpose. Then the object pointer is referenced to the objectRegiste with *addObject*.

However, hidden from view is a call to objectRegister in FixedComp's constructor, which is certain to be called as all registered object must derive from this class. That occures during the operator new call and results in the allocation of the cell/surface numerical range. If it is necessary to trap that error, the try/catch block must be around the new opertor. The main exception that is possible if an existing object already exists with the same name.

6.2   EXT Command

MCNP(X) provides the EXT card for biasing the direction of the particles after collisions. The card can be configured with a stretching parameter value between -1.0 and 1.0 and an optional vector or direction associated with it. If a vector is not given the stretching parameter is applied in the direction of the neutron travel.

MCNP(X) only accepts the direction to be X,Y,Z which is highly limiting in the CombLayer environment, so it is only partially supported.

The other two options vector and non-vector are supported.

6.3   -wExt entry

The first method of entry is via the command line option -wExt. This command takes a sequence of additional values which are split into a *zone* and *type* region. The *zone* region is based on the cells that are to be biased. This can be give with the commands:

6.3.1   Zone

- **all** : Apply to all non-void cells

- **Object [name]** : Apply to all objects within the object name

- **Cell [Range]** : Apply to all objects within the range

Name can be a compound name of type objectName:CellMapName. This would just select those cells within the cellMap unit of the particular object.

6.3.2   Zone

7   User guide

This section describes how to use CombLayer from a user's (i.e. non-developer) point of view. In this guide, it is assumed that the user has no C++ or coding experience.

The guide is focused on the ESS model, which can be generated by running.

```
./ess −r modelOut
```

This command produces the MCNP input file `modelOut1.x` as well as two other files: `ObjectRegister.txt` and `Renumber.txt`.

The single flag `−r` is optional for MCNP6 as it causes the objects and surfaces in MCNP to be renumbered sequentially and to fit within the 100,000 object/surface limits of MCNPX.

7.1   Variables

In the beginning of the input file there is a commented list of variables which define the geometry:

```
c ——————————————————————————————————————
c ———————————— VARIABLE CARDS ——————————
c ——————————————————————————————————————
c ABunkerFloorDepth 120
c ABunkerFloorThick 100
c ABunkerLeftAngle 0
c ABunkerLeftPhase −65
c ABunkerNLayers 1
...
```

The variable name consists of the component name and its corresponding parameter. For instance, the first variable `ABunkerFloorDepth` in the list above sets the floor depth of the component called `ABunker`.

Only variables that have needed to be examined are included in this output. Several components are optional and if not built then their corresponding variables are not seen in the output. All variables start with an initial value

and stateless variables are prohibited. Most variables are defined appropiately within the CombLayer program and obviously can be changed by editing the code and recompiling. However for a simple variable change that is excessive work so there are a number of methods to change variables from the command line.

7.1.1 How to change variables

Any of these variables can be changed either via a command line arguments or an XML file.

As an example, consider changing the Beryllium reflector height. First of all, we need to find out which variable we need to change and therefore find out the name of the Be reflector component in CombLayer.

To do this, open the `MCNP` geometry and click on any Be reflector cell. Currently, it's cell number 5 (exact number depends on the CombLayer version you are using).

Now we need to find out which component this cell belongs to. Find this cell number in the `Renumber.txt` file:

```
grep " 5 " Renumber.txt
Surf Change:1000006 5
Cell Changed :1000005 5 Object:BeRef (topBe)
```

It shows that the corresponding Be reflector object is called `BeRef`.

Now we need to find out which `BeRef` variable is responsible for its height:

```
grep BeRef a1.x
c BeRefHeight 74.2
c BeRefLowRefMat Be5H2O
c BeRefLowWallMat Stainless304
c BeRefRadius 34.3
c BeRefTargSepMat Void
c BeRefTopRefMat Be5H2O
c BeRefTopWallMat Stainless304
c BeRefWallThick 3
c BeRefWallThickLow 0
```

We can guess from this list that the variable we need is called `BeRefHeight`.

Changing variables via command line In order to change a variable via command line arguments, run:

```
./ess −r −v BeRefHeight 50 modelOut
```

Several variables can be changed, e.g.:

```
./ess −r −v BeRefHeight 50 −v BeRefRadius 35 modelOut
```

Note that it is possible that you miss spell one of the variables, if this is the case then you will be presented

```
./ess −r −v BeRefHHH 1 modelOut
Failure to find variable name BeRefHHH              MainProcess[F]::setRunTimeVariable
Exiting from BeRefHHH not found                     ::main
Exit Stack:                                         ::main
::main                                              ::main
  MainProcess::InputModifications                   ::main
    MainProcess::setVariables                       ::main
      MainProcess[F]::setRunTimeVariable            ::main
```

Like most CombLayer error messages it is expected that you read them from top to bottom. So the first thing it tells you is that variable BeRefHHH does not exist in the model. Second that this is a fatal error and finally the calling stack that generated that error. If you are not debugging the code etc, then only concern yourself with the error/warning messages above the line *Exit Stack:*.

It is also possible to use strings and Vec3D objects as variables:

```
./ess −r −v BeRefTopRefMat Nickel \
       −v ABunkerQuakePtA0 'Vec3D(1200,191.2,0.0)' modelOut
```

will set the Top reflector material to *Nickel* and the start of the dilitation joint at 1200,191.2,0 (relative to target centre. Note that on bash you will need to hard quote (single quote) the Vec3D value or it will be split into commands.

Changing variables with XML file   Create an XML file with the following content:

```xml
<?xml version="1.0" encoding="ISO−8859−1" ?>
<metadata_entry>
  <Variables>
  <variable name="BeRefHeight" type="double">50</variable>
  <variable name="BeRefTopRefMat" type="string">Nickel</variable>
  </Variables>
</metadata_entry>
```

and generate the modified geometry:

```
./ess −r −x model.xml modelOut
```

All the variables can be exported in the XML file by running

```
./ess −r −X modelOut.xml modelOut
```

Note that the -X command will overwrite existing files.

The commands can be done combined simulantiously with both XML and command line and output.

```
./ess −r −X modelOut.xml −x model.xml −v BeRefHeight 80.0 modelOut
```

This will first read the model.xml file and change the variables. Second it will change `BeRefHeight` to 80, and finally write out the XML file of all variables to the file `modelOut.xml`.

Dealing with dependent variables   If some other variables depend on the variable you are going to change, it can break the geometry. Consider, for instance, changing the BeRef radius from the baseline value of 34.3 to $40\,\mathrm{cm}$:

```
./ess −r modelOut
```

this generates the $34.3\,\mathrm{cm}$ model[1] as shown in Fig. 6a.

Now let's set it to $40\,\mathrm{cm}$:

```
./ess −r −v BeRefRadius 40 modelOut
```

This geometry is shown in Fig. 6b and it is broken since now `BeRef` intersects with the Bulk component.

---

[1] $34.3\,\mathrm{cm}$ is the baseline BeRef radius in the commit `e47bf6d`.

(a) Baseline radius of $34.3\,\mathrm{cm}$

(b) `BeRefRadius` increased to $40\,\mathrm{cm}$ without taking into account dependent variables, which produced geometric errors

(c) `BeRefRadius` increased to $40\,\mathrm{cm}$ taking into account dependent variables
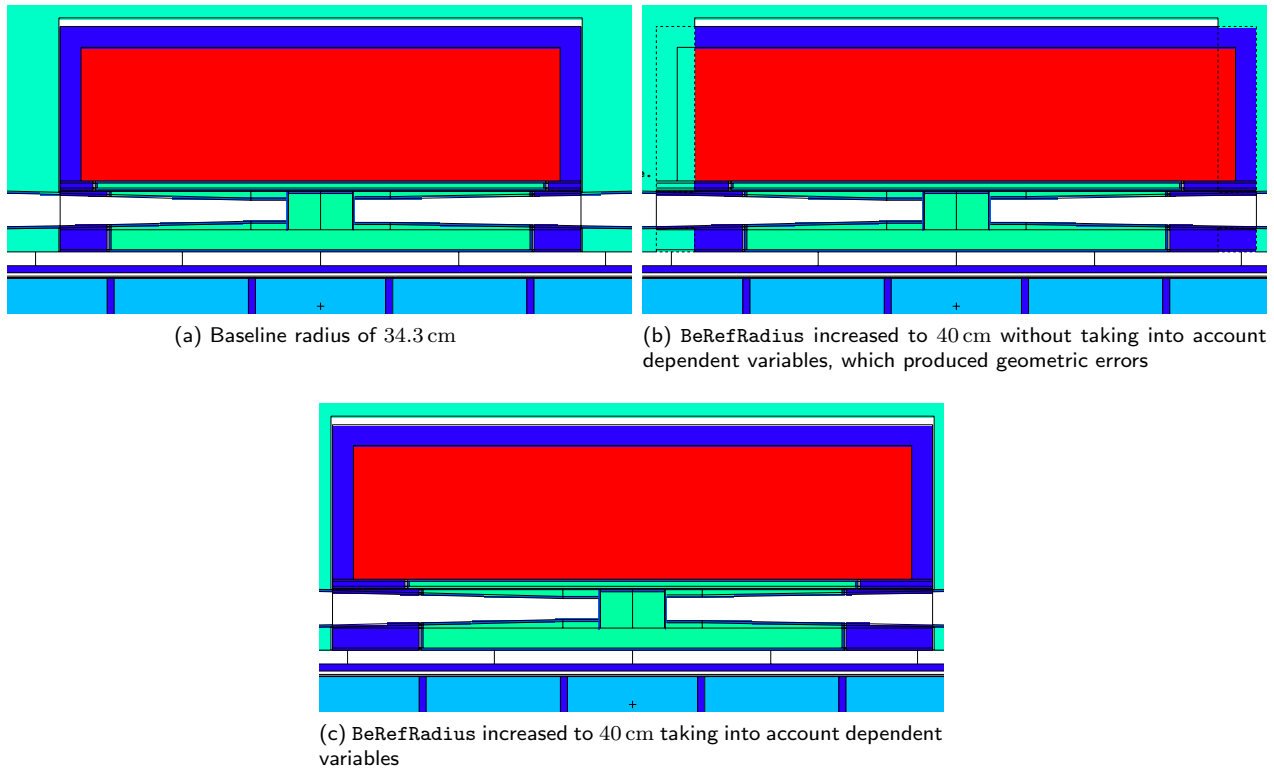
Figure 6: Geometries with different BeRef radii

In order to find out which other variables depend on the given one (`BeRefRadius` in our case), find it in the variables setup in the C++ variable definition:

```
grep BeRefRadius Model/essBuild/*.cxx
Model/essBuild/essVariables.cxx:    Control.addVariable("BeRefRadius",34.3);
Model/essBuild/essVariables.cxx:    Control.addParse<double>("BulkRadius1",
                                       "BeRefRadius+BeRefWallThick+0.2");
```

It means that we have to adjust the `BulkRadius1` variable accordingly. This variable depends upon both `BeRefRadius` and `BeRefWallThick`. Find out the `BeRefWallThick` value:

```
grep BeRefWallThick modelOut1.x
c BeRefWallThick 3
```

Therefore the `BulkRadius1` value must be $40 + 3 + 0.2 = 43.2\,\mathrm{cm}$:

```
./ess −r −v BeRefRadius 40 −v BulkRadius1 43.2 modelOut
```

which produces the correct geometry shown in Fig. 6c.

Important note   Sometimes in the C++ variable definitions the dependence is not set explicitly, i.e. in our case `BulkRadius1` would be defined just by the value:

```
Model/essBuild/essVariables.cxx:    Control.addParse<double>("BulkRadius1",43.2);
```

In this case we have to inspect the geometry manually in order to find out which other variables we need to change to produce the correct input deck.

7.2 Variance reduction

7.2.1 New way: FW-CADIS

We consider our result (response of a detector) R to be a convolution of particle angular flux $\Psi(\vec{P})$ determined by all the parameters $\vec{P}$ (position $\vec{r}$, energy E, angular coordinate $\hat{\Omega}$, i.e. all phase space) and a determenistic cross section[2] Haghighat2003, page 33 $\sigma_D(\vec{P})$ that relates to that:

$$R = \int \Psi(\vec{P})\sigma_D(\vec{P})d\vec{P} \tag{1}$$

That allows you to have a selectivity because a neutron must cross a certain surface or have a certain energy or direction and this is encoded into $\sigma_D(\vec{P})$.

Cross section $\sigma_D(\vec{P})$ is a *determenistic* cross section, which constructed to be representative of the quantity that the simulation is going to measure. For example, if the flux going through a cell for the energy between $1\,\mathrm{eV}$ and $2\,\mathrm{eV}$ is the quantity of interest, then $\sigma_D = 1$ when the particle is located in the given cell and energy is between these two values, otherwise $\sigma_D = 0$.

We will now imply the integral over $d\vec{P}$, and imply that there is a unique Hamiltonian, that can express the flux at all points in the phase space are equal to the source value. It is basically second order differential equation. The Boltzmann transport equation expresses the particle balance in a phase space as[3]:

$$H\Psi = q \tag{2}$$

where $q(\vec{P})$ is the source response function.

So we operate on the whole flux, and this allows the field to be evolved. Hamiltonian always operates on flux in order to evolve it to a new type. We will assume that we start at some source at time zero and we will evolve it, and (2) has to hold for all time since we have conservation.

Conjugated form:

$$H^*\Psi^* = q^* \tag{3}$$

This is the general form. As an example, for a simple neutron that under, elastic scattering and absorption only we will see that equation (2) becomes the classical scattering equation [2, Eq. 1]:

$$\left[\hat{\Omega} \cdot \nabla + \Sigma_t(\vec{r}, E)\right]\Psi(r, \hat{\Omega}, E) = \int_0^\infty dE' \int_{4\pi} d\hat{\Omega}' \, \Sigma_s(\vec{r}, E' \to E, \hat{\Omega}' \cdot \hat{\Omega}) \, \Psi(\vec{r}, \hat{\Omega}', E') + q(\vec{r}, \hat{\Omega}, E) \tag{4}$$

$$\left[-\hat{\Omega} \cdot \nabla + \Sigma_t(\vec{r}, E)\right]\Psi^*(\vec{r}, \hat{\Omega}, E) = \int_0^\infty dE' \int_{4\pi} d\hat{\Omega}' \, \Sigma_s(\vec{r}, E \to E', \hat{\Omega} \cdot \hat{\Omega}') \, \Psi^*(\vec{r}, \hat{\Omega}', E') + q^*(\vec{r}, \hat{\Omega}, E) \tag{5}$$

These two cases are the forward going (i.e. from the source to the tally region) and the backward going from the tally to the source. Note the the flux must be the same is we assume reversibility.

We can [with a lot of algebra/Mathematics] show that they obey the "adjoint property",

$$< \Psi^*, H\Psi > = < \Psi, H^*\Psi^* > \tag{6}$$

and therefore

$$< \Psi, q^* > = < \Psi^*, q > \tag{7}$$

---

[2] Detector cross section, response function.
[3] More detailed derivations can be found in [1].

where the Dirac signs, $<>$, refer to integration over all variables $\vec{P}$ (the inner product).

Now if we set $q^* \equiv \sigma_D$, then we have

$$< q^*, \Psi >=< \sigma_D, \Psi >= R =< q, \Psi^* > \tag{8}$$

this is the same as

$$R = \int \Psi(\vec{P})q^*(\vec{P})d\vec{P}$$

and

$$R = \int \Psi^*(\vec{P})q(\vec{P})d\vec{P}$$

Assumptions   We have made the following two assumptions:

1. Time reversal for neutrons completely valid;

2. If we start with the neutron at our tally point and run it backwards, we get exactly the same thing as running it forwards.

This means that we are operating with one-group/multi-group equations, i.e. not allowing the neutrons to change energy or allowing the neutrons to change energy only into discrete probabilities.

This situation is valid if we consider only elastic scattering and absorption processes, and we believe that this approximation should be enough for variance reduction purposes.

Now we state that absorption is all absorption capture + all loss, where a neutron losses energy and falls out of the group (i.e. it's much bigger absorption cross section than the stated one, see $\sigma_A^\dagger$ below).

The contribution of the individual piece of flux to R (what we really want to know) can be expressed by

$$w(\vec{P}) = \frac{R}{\Psi^*(\vec{P})} \tag{9}$$

where $\Psi^*(\vec{P})$ is adjoint flux (backward going flux with respect to $\Psi(\vec{P})$). ans w is the inverse of the probability importance of a region in $\vec{P}$.

At the same time we know that we can express our weight as

$$w = w_0 \cdot q(\vec{P}) \tag{10}$$

$q(\vec{P})$ we can calculate using the Hamiltonian in (2).

This leaves us with the new expression for R:

$$R = \int d\Omega \int d\vec{r} \int dE \, \Psi(\vec{r}, E, \Omega) \, f(\vec{r}, E, \Omega) \tag{11}$$

where $f(\vec{r}, E, \Omega)$ is an approximation of the transfer function from the phase point, $P(\vec{r}, E, \Omega)$ to the detector. The CADIS paper uses a proper Monte-Carlo determined $f(\vec{r}, E, \Omega)$ but CombLayer does not do this is approximates the term (for speed).

The CADIS authors find their version of the fuction based on the true weight of the number of particles in the given cell (times the Monte Carlo density). CombLayer approximates the particles in a given cell by direct transport in and a number of Markov-Chain steps (but not to convergence), which often is zero steps.

7.3 CombLayer determination of R

CombLayer needs to find approximations for both Ψ and f. First consider Ψ. CombLayer is not going get the exact value for Ψ because it means we need to run full simulation. However, we can find Ψ in a one-group approximation. To do this, we need σ_T and $\sigma_A^\dagger$, where

σ_T The total interaction cross section [including all absorption terms].

$\sigma_A^\dagger$ Probability that a neutron is lost from the group *plus* the probability that a neutron is being absorbed.

CADIS divides energy range into 33 energy bins. CombLayer could support that many but with the CombLayer approximations that would be excessive. It is suggested that for a typical spallation neutron monte-carlo, between 5 and 7 is a good starting number. For electron - photon problems between 2 to 3 energy bins seems to be a good starting point.

When we run our variance reduction, we are going to calculate two things:

- We are going to tell the program what σ_D is, and that's easy because it's defined by the tally.

- We have to roughly define where the source of our particles is going to start from. In principle that's where the proton start, and that would be a good thing except that in reality that's not the most efficient thing at all. You can multiply the variance reductions together, therefore it's better to define several weight window meshes (i.e. first — from Target to Moderator, second — from Moderator to Monolith Insert, third — Insert to Bunker Wall etc).

So, you can have as many source points as you like, and as many adjoint pionts as you like, but you need a minimum of one of each.

Not all sources are points: some are planes, some are cylinders. At the moment in CL there are 3 types of shapes: a point, a plane, a cone. More shapes and head rules will be added in the future. Equally, same shapes can be set as the adjoint sources (where we are focusing into). Points work reasonably well, but should be extended to cones/cylinders.

The user defines the source point and the adjoint point, and CombLayer computes R based on (11) and weight based on (9)... and you are done!

Normalisation   Previous one sided variance reduction, not base on both a source term and an adjoint form, lead to an unbiased variance reduction that is not normalised. If the source and the tally variance reduction terms are both used then the above theory does not give rise to any need to do normalisation. However, due to the need to issues like doing a quick and simple simulation were a source is approximated or multiple tallies normalisation may be required. For example if a simulation is started from the ESS target but the flux down a beam line is required two sources may be used e.g one at the target and one at the exit of the monolith. There is currently no weighting system for the contribution of both variances [which should be done using the probability of exiting the monolith] and thus the variance reduction mesh can be significantly mis-scaled.

Therefore, there is the option to re-normalised the whole variance reduction mesh by adding the flag `--wwgNorm meshName AValue BValue` such that the new normalisation range between low/high is limited between A and B.

There is a rounding error of the order $10\,\%$ to $15\,\%$. It comes mainly because sometimes we divide very small numbers (of the range 10^{-30}). So, you can do normalisation here, but SA can't be bothered most of the time.

That's the principle of the variance reduction. In addition, there is a whole pile of tools which can help you out when things are not going in the appropriate way. For example, you can change the density of the entire model

(for the variance reduction only, not for the real run). If you decrease it (say, to 80%), it would oversample the area in the end[4].

You can also put a limit to indicate that you do not want to write variance reduction down to 10^{-317}, because it overpopulates too much the given cell. Normally, this limit is 10^{-46} to 10^{-20}.

7.4 How to use it

There are two types of variance reduction in MCNP and in CombLayer. One is a cell-based, the other one is a mesh-based variance reduction. Stuart Ansell had modified his MCNP version to allow both at the same time, there are further modifications to PHITS and FLUKA to support this as well. The following are the general commands that are not specific to Monte-Carlo code choise.

7.4.1 Mesh-based variance reduction

```
maxiv −w −wWWG \
 −−weightSource S0 'Vec3D(1000.0, 0.0, 14.0)' \
 −−weightSource S1 TDC29BeamStopA:back 'Vec3D(1500.0, 0.0, 14.0)' \
 −−wwgPlane P1 TDC29BeamStopA:back 'Vec3D(0,1,0)' 'Vec3D(0,240,0)' \
 −−wwgEnergy E1 0.01 2.0 1300.0 \
 −−wwgMesh M1 'Vec3D(−400,9900,−30)' 'Vec3D(−320,9970,30)' 30 50 1 \
 −−comment "mesh particle grid energyCut density r2length r2power" \
 −−wwgCreate SS0 all M1 E1[0] S0 0.9 1.0 1.2 \
 −−wwgCreate TS0 all M1 E1[0] S1 0.9 1.0 1.2 \
 −−wwgCADIS SSX[0] SS0[0] TS0[0] S0 S1 0.009 1.0 1.5 \
 −−wwgCreate SS0 all M1 E1[1] S0 0.009 1.0 1.5 \
 −−wwgCreate TS0 all M1 E1[1] S1 0.009 1.0 1.5 \
 −−wwgCADIS SSX[1] SS0[1] TS0[1] S0 S1 0.009 1.0 1.5 \
 −−wwgNorm SSX[0] 8 \
 −−wwgNorm SSX[1] 5 −1 0.0 −20 \
 −−wwgActive SSX Log \
 AA
```

Summary of commands

-w needs to be the first flag in the list of biasing-related flags

-wWWG we are interested in the mesh-based weight window generator

--weightSource defines a (source or adjoint) *single point* in space. It's possible to define as many as necessary, and not all of them need to be used. In this example, there are two point sources defined: one at an absolute point (S0) and one at a relative to a FixedComp linkPoint (S1). The names can be any valid string, for simplicity in the example, they are called $S1$ and $S1$.

--wwgEnergy defines the boundary of the energy bins. If there are N energy boundaries, there are N-1 energy bins. There is no implicit bin boundary at 0.0, it that is require it must be explicit included in the list.

Format: EnergyName(string) $[e_0\ e_1\ e_2]$(double)

The name can be any valid string (gain

$e_0\ e_1\ e_2$ .... these are the energy bin boundaries. The minimum is 2 values (for one energy bin), and they should be in increasing order.

---
[4]See the `--wwgCalc` flag in Sec. 7.4.1.

–wwgMesh defines a spacial mesh.

$Vec3D(LowPoint)$ Low point of the box

$Vec3D(HighPoint)$ High point of the box

NX NY NZ : Number of bins in each orthogonal axis

Note that each Vec3D point can be a generalized FixedComp linkPoint, or a PointMap value. Examples :

1. –wwgMesh Vec3D(-100,-100,-100) Vec3D(300,300,300) 100 50 200
2. –wwgMesh Vec3D(-100,0,0) BeamStop:EndPoint 100 50 20

Example 1 : is a box between two point 400x400x400 in size and with X,Y,Z cells 100, 50 and 200 respectively. This is a lot of cells!

Example 2 : This is a box which start at x=-100 and goes to the PointMap positon given in FixedComp BeamStop

–wwgCreate Fill a energy/grid mesh from a source point Syntax:

SS0 New name for the mesh (it can be any string).

all particles to use for mesh, using full name e.g. neutron they can be listed together e.g. photon,electron with a comma.

M1 Name of the grid to use for this mesh (defined with –wwgMesh)

E1[1] Name of the energy vector (–wwgEnergy) and the index of the energy boundary

S0 Name of the source point/plane (–wwgSource)

0.009 density factor. Normally the trick is to decrease density in order to populate remote cells.

1.0 fractional change of length

1.5 the power of r^2. It is an effective buildup term.

–wwgCADIS The CADIS approximation basically calculates the weighted fraction between source to cell and cell to tally. This is carries out on a energy point by energy point basis. Currently both meshes in the CADIS calculation have to be identical, the energy groups do not need to be.

SSX[0] Create a new group called SSX and carry out the CADIS calculation for the first energy bin. The mesh/energy bins will be taken from the source mesh.

SS0[0] Use mesh SS0 and use the first energy bin as the source

TS0[0] Use mesh SS1 as the adjoint (tally).

S0 The source point to use for tracking from source to cell.

S1 The source point that is use for tracking from cell to point. It is the effective tally point.

0.009 density factor. Normally the trick is to decrease density in order to populate remote cells.

1.0 fractional change of length

1.5 the power of r^2. It is an effective buildup term.

–wwgNORM The command applied renormalization to part (all) of a mesh, so the weights are between 1.0 and a very low number. The command works in natural log space.

SSX[1] The mesh and energy bin to be normalized

8 The range for the weight values (exp(8))

-1 The highest value in the weights. Thus the range runs from exp(-1) to exp(-9) in the output.

0 All values above exp(0) in the un-normalized mesh are treaded as exp(0).

-20 All values below exp(-20) in the un-normalized mesh are treaded as exp(-20).

–wwgVTK Write out the mesh to a VTK file.

testA.vtk Filename for the output

SSX[1] The mesh and energy bin to be written.

Log The file is either W (typically a small value) or the -log(W). So the numbers are positive and run from 0 to a value.

In MCNP, the input deck, the weight-window parameters are defined as

```
wwp:n 8 1.4 5 −1 −2 0
```

where `switchn=-2`. In the non-modified `MCNP` this value indicates that "the lower weight-window bounds should be read from an external `wwinp` file containing either cell- or mesh-based lower weight window bounds". SA modifies `MCNP`, and this value means *both* cell and weight. It's possible to use `switchn=-2` for non-modified `MCNP` versions, and the rule is that when you've got outside of the mesh boundary, you score importance only. In your input file you have a list of the cells and cell weights, and with regular `MCNP` you will get a warning saying that the cell weights being ignored. `MCNP` will happily run, and if a particle drops outside of the boundary and has a scatter, you will be warned that importances are being sampled instead of `wwinp` and you will get the default importance which in this case is 1.

> Show the exact message

Modification of MCNP   `MCNP` uses a single array to mean either the mesh or the cell grid, that's why it can't do both at the same time. SA added a multiple array in order to be able to have several simplified meshes, which can be put in arbitrary way. These meshes can be ordered. We test the first one first. If you want to have a mesh in a mesh, you simply put the final mesh at the top, so you can have exactly the same behaviour as you currently have, but you can also have much better behaviour in the sense that two meshes do not need to be orthogonal, they can follow bending in a curving beam line, which is useful[5]. Or you can have a fixed mesh around the moderator, which does not have to respect whichever direction of beam line we are taking away. And SA can have that mesh isolated and separate.

Modifications to FLUKA   Fluka provides a user set of functions, and for importance provides usibms.f, whichi is called if BIASING is set with SDUM=USER. This select which particles and which cells will call the user defined user-im

7.4.2   Old way: cell-based biasing

Cell-based variance reduction is the old method implemented in CombLayer.

```bash
#! /bin/bash

./ess −r −defaultConfig Single ODIN −angle objAxis odinAxis 0 \
    −w \
    −−weightEnergyType energy 0.1 0.95 1.0 0.85 10.0 0.5 100.0 0.4 5000.0 0.3 \
    −−weightSource 'Vec3D(200.0,4.0,13.7)' \
    −−weightPlane  'Vec3D(2800,0,0)' 'Vec3D(1,0,0)' \
    −−weightObject G2BLineTop20       SS0 0.0 1.0 0.9 1.0 2.0 \
    −−weightObject CBunkerWallMainWall1 TP0 0.0 1.0 0.9 1.0 2.0 \
    −−weightObject CBunkerWallMainWall2 TP0 0.0 1.0 0.9 1.0 2.0 \
    −−voidUnMask \
    TA
```

-w should precede all weight-related arguments.

−−weightEnergyType defines energy grid. With the `wwg` card you can either use this expression or `−−wwgE`. If use use the word "energy" then the format is energy (below $0.1\,\mathrm{MeV}$) followed by default weight for that bin (0.95) etc. If you drop the word "energy" then you just set the energy grid. In the given example for energies below the default weight is 0.95 etc. Alternatively you can use pre-defined keywords: `basic`, `high`, `mid` and `flat`[6].

---

[5]Especially for LOKI, which has two $\sim$5° bends.
[6]Energy-weight binnings for these keywords are defined in `System/weights/WeightControl.cxx`

–weightSource This argument defines points in space. We can define as many as we want, but only those used with `weightObject` will be used.

–weightPlane This argument defines planes. We can define as many as we want, but only those used with `weightObject` will be used.

–weightObject `BBunkerWallMainWall1 TP1 1.0 0.15 1e-20` : object we are interested in calculating; TP1: tally plane 1; `1.0 1.0 0.15 1e-20`: energy above $1\,\mathrm{MeV}$ (below it we are not interested — use default values); `1.0`: scale factor; `0.15`: density factor; `1e-20`: minimum weight

–voidUnMask By default (without this argument) the importance of the surrounding void cell is zero. This argument sets it to one. It can be useful if we believe that particles travelling through the void cell can contribute into the tally.

–weightObject BBunkerWallMainWall1 TP1 1.0 0.15 1e-20

object we calculate ↗ ↗

tally plane 1

↖ Min energy
(below it we are not interested —
use default values)

Here we need to define an object we are interested in calculating, and the idea is exactly the same as we had previously. When you run it, it takes longer time to generate the weight window since the implementation is not perfect: it calculates every individual cell and calculate track through it. In the new method described in 7.4.1 SA takes a cell and track a line through it and removes from the need to be calculated every cell it goes through .

In the generated input deck the cells 687–701 belong to the wall and they are being evaluated:

```
wwe:n  0.1  1  10  100  5000
wwp:n  2  1.4  5  0  0
#     wwn1:n    wwn2:n    wwn3:n    wwn4:n    wwn5:n
c   c          c          c          c          c
...
685    9.50e−01 8.50e−01 5.00e−01 4.00e−01 3.00e−01
686    9.50e−01 8.50e−01 5.00e−01 4.00e−01 3.00e−01
687    9.50e−01 2.73e−01 1.61e−01 1.29e−01 9.64e−02
688    9.50e−01 2.72e−01 1.60e−01 1.28e−01 9.60e−02
689    9.50e−01 2.95e−01 1.74e−01 1.39e−01 1.04e−01
690    9.50e−01 3.24e−01 1.90e−01 1.52e−01 1.14e−01
691    9.50e−01 3.60e−01 2.12e−01 1.70e−01 1.27e−01
692    9.50e−01 3.63e−01 2.14e−01 1.71e−01 1.28e−01
693    9.50e−01 3.98e−01 2.34e−01 1.87e−01 1.41e−01
694    9.50e−01 4.21e−01 2.48e−01 1.98e−01 1.49e−01
695    9.50e−01 4.72e−01 2.78e−01 2.22e−01 1.67e−01
696    9.50e−01 4.93e−01 2.90e−01 2.32e−01 1.74e−01
697    9.50e−01 5.47e−01 3.22e−01 2.58e−01 1.93e−01
698    9.50e−01 5.85e−01 3.44e−01 2.76e−01 2.07e−01
699    9.50e−01 6.59e−01 3.88e−01 3.10e−01 2.33e−01
700    9.50e−01 7.03e−01 4.13e−01 3.31e−01 2.48e−01
701    9.50e−01 8.07e−01 4.75e−01 3.80e−01 2.85e−01
702    9.50e−01 8.50e−01 5.00e−01 4.00e−01 3.00e−01
```

There is one problem here: since we've reduced the density, these numbers are not aggressive enough. Therefore for actual run we should set the density factor to 1, so the weights will be more reasonable:

```
wwe:n  0.1  1  10  100  5000
wwp:n  2  1.4  5  0  0
#     wwn1:n    wwn2:n    wwn3:n    wwn4:n    wwn5:n
c   c          c          c          c          c
...
685    9.50e−01 8.50e−01 5.00e−01 4.00e−01 3.00e−01
```

```
686    9.50e−01 8.50e−01 5.00e−01 4.00e−01 3.00e−01
687    9.50e−01 4.39e−04 2.58e−04 2.07e−04 1.55e−04
688    9.50e−01 4.26e−04 2.50e−04 2.00e−04 1.50e−04
689    9.50e−01 7.39e−04 4.35e−04 3.48e−04 2.61e−04
690    9.50e−01 1.36e−03 8.02e−04 6.42e−04 4.81e−04
691    9.50e−01 2.78e−03 1.64e−03 1.31e−03 9.82e−04
692    9.50e−01 2.93e−03 1.72e−03 1.38e−03 1.03e−03
693    9.50e−01 5.43e−03 3.19e−03 2.55e−03 1.92e−03
694    9.50e−01 7.89e−03 4.64e−03 3.71e−03 2.78e−03
695    9.50e−01 1.68e−02 9.89e−03 7.91e−03 5.93e−03
696    9.50e−01 2.24e−02 1.32e−02 1.06e−02 7.92e−03
697    9.50e−01 4.51e−02 2.66e−02 2.12e−02 1.59e−02
698    9.50e−01 7.08e−02 4.16e−02 3.33e−02 2.50e−02
699    9.50e−01 1.56e−01 9.16e−02 7.32e−02 5.49e−02
700    9.50e−01 2.39e−01 1.41e−01 1.12e−01 8.43e−02
701    9.50e−01 6.02e−01 3.54e−01 2.83e−01 2.12e−01
702    9.50e−01 8.50e−01 5.00e−01 4.00e−01 3.00e−01
```

There is very little in this program from stop you from being stupid if you want to be. There is no warnings. You need to understand what you are doing.

The reason the density factor is there because sometimes you look at the simulation and see that neutrons are not going far enough. So, you tweak the density (normally from 0.9 to 0.95), and you are still not sure whether it is completely legitimate.

The weights $10^{-5} - 10^{-3}$ are the numbers we expect to see for a single point process.

This weight window generator has been in CombLayer since the middle of 2016.

In the new mesh-based weight window generator you've lost nothing, but you gain some new things.

7.4.3   Technical details

Angular term approximation   For a given cell the integral you have to do is

$$\phi^\dagger(\vec{r}, E) = \frac{\int_\Omega \phi(\vec{r}, E, \hat{\Omega}) \phi^\dagger(\vec{r}, E, \hat{\Omega})}{\int_\Omega \phi(\vec{r}, E, \hat{\Omega})} \tag{12}$$

and you need some angular term, but you do not calculate the angular term. You approximate it in a really brutal approximation way:

$$\phi(\vec{r}, E, \hat{\Omega}_{ijk}) = \frac{\phi_{000}(\phi_{000} + (\phi_{ijk} - \phi_{-i-j-k})/2)}{26\phi_{000}} \tag{13}$$

This is the cell I wish to compute something for. So I need the adjoint flux on this cell. And I am going to compute the integral (12).

The angular parts proportionate as (13). Each cell as 26 neighbours (Fig. 8). The approximation is done by whole set of single subtractions from the cell in all directions.

The ijk is standard Einstein summation. You do this summation over positive and negative directions, and across the diagonals all the time. Then you can put $\phi(\vec{r}, E, \hat{\Omega}_{ijk})$ calculated in (13) into (12).

Then you do exactly the same thing for $\phi^\dagger(\vec{r}, E, \hat{\Omega}_{ijk})$ as well, and that gives the value of $\phi^\dagger(\vec{r}, E)$ in (12).
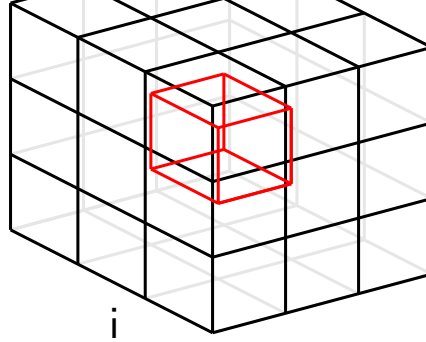
Figure 7: Cell with its neighbours

Markov chain update   You also must have done the Markov chain update. You populate the flux with the initial source, and then you do the Markov chain update for cells to contribute to each others local cells.

$$\sigma_{\text{total}} = \sigma_{\text{elastic}} + \sigma^*_{\text{abs}}$$

For every cell we compute the probability of going to its neighbour. We say that no cell can go to anything other than its neighbour. The idea is not to make an $\mathcal{O}(n^2)$ computation, but the $\mathcal{O}(n \log n)$ one.

So, first of all I compute every neighbour, so I have a Markov chain matrix.[7]

And notice contribution probability from going to the given cell from one of its neighbour cells (all the touching neighbours are considered).

Then you use the Markov chain process, where you integrate this [???] matrix together with $n \to \infty$ and you get stability.

You take the value in all the cells according to that prescription.

The Markov chain process allows you to have a structure within the mesh, which you do not need to be carefully align with to do staff.

Comparison with ADVANTAG   The adjoint process the CADIS people do indeed accounts for that, but they run full simulation to do it using discrete ordinance (?). CombLayer does not do these calculations because they take long time.

If you really need CADIS adjoint and ADVANTAGE, go and do that. The CombLayer implementation is somewhere half way stage between doing it manually and the ADVANTAG way.

If the CombLayer way does not work, maybe you have to do ADVANTAG, but it can be tricky to set it up.

ADVANTAG really computes the angular phase term, while Stuart just approximates it.

The beauty of CombLayer way is that it does not take much time to set it up. It's simple: you either list the object you are interested in and put the variance reduction for it, or you set the grid you are interested in and you put basic flags down.

It's work in progress. More will be added to it.

Stuart is using it for long beam lines.

[2] is the only paper where they say what exactly needs to be done. You compute this adjoint flux; compute R, which eventually will be your weight term; compute q, which you are going to multiply through by to get this term which is going invert by computing this cross integral. You use equations 3 and 4, because you have both, and you just simply solve. And you are done.

This is approx what SA does with the approximation he made.

It will probably not work, but for SA it worked for the wall and for the roof.

7.4.4   Cell-based biasing

Cell-based weight window can be generated with the following arguments:

---

[7]The matrix is too big, so we store only its sparse components.

```bash
#! /bin/bash

./ess −r −defaultConfig Single ODIN −angle objAxis odinAxis 0 \
    −w \
    −−weightEnergyType energy 0.1 0.95 1.0 0.85 10.0 0.5 100.0 0.4 5000.0 0.3 \
    −−weightSource 'Vec3D(200.0,4.0,13.7)' \
    −−weightPlane  'Vec3D(2800,0,0)' 'Vec3D(1,0,0)' \
    −−weightObject G2BLineTop20        SS0 0.0 1.0 0.9 1.0 2.0 \
    −−weightObject CBunkerWallMainWall1 TP0 0.0 1.0 0.9 1.0 2.0 \
    −−weightObject CBunkerWallMainWall2 TP0 0.0 1.0 0.9 1.0 2.0 \
    −−voidUnMask \
    TA
```

The purpose of the variance reduction is to change the weight in the cell by the following equation

$$w_{mod} = \mathsf{scaleFactor} \times \frac{\exp(-\sigma \times \rho \times \mathsf{densityFactor} \times r \times \mathsf{rScale})}{(\mathsf{rScale} \times r)^{\mathsf{r2Power}}} \tag{14}$$

Note the repeate of rScale in the equation, thus effectively separating densityFactor from rScale. For each –weightObject the weight in the cells in modifed by w_{mod} in the equation above.

-defaultConfig Single ODIN These optional arguments build the ODIN beam line without building any other beamlines (see Fig. 8)..

-angle objAxis odinAxis 0 This rotates the model about the master z axis followed by around the master y axis such that the FixedComp link-point axis specified after objAxis is collinear with the x axis in the final output. In this example the FixedComp is the *odinAxis* and the link point is zero (which signifies to used the FixedComp origin and Y axis rather than a linkPoint). This rotation simplifies the weight window source plane setup for the current example and used here for illustration purpose.

-w should precede all weight-related arguments and sets up the model for variance analysis.

–weightEnergyType defines energy grid[8]. The following variants of syntax are possible:

1. If use the word *energy* then the format is energy followed by initial weight for that bin (in our example: for energies below $0.1\,\mathrm{MeV}$ the initial weight is 0.95; for energies between 0.1 and $1\,\mathrm{MeV}$ the initial weight is 0.85 etc).

2. If you drop the word *energy* then you just set the energy grid and all the initial weights are taken as 1.0

3. Alternatively you can use pre-defined keywords: `basic`, `high`, `mid` and `flat`[9].

–weightSource This argument defines a point in space. We can define as many as we want (by adding several `weightSource` arguments), but only those used with `weightObject` will be used. When referenced they will be called S0, S1, S2 ... etc based on the order they appear in the command line. This particular point is shown by a circle in the left side of Fig. 8a.

–weightPlane This argument defines a plane. Similarly to `weightSource`, we can define as many planes as we want, but only those used with `weightObject` will be used. When referenced they will be called P0, P1, P2 ... etc based on the order they appear in the command line. This particular plane is shown by a dashed vertical line in the right side of Fig. 8a.

–weightObject Define objects we would like to make cell-based variance reduction to.

---

[8]With the `wwg` card you can either use this expression or `--wwgE`

[9]Energy-weight binnings for these keywords are defined in `System/weights/WeightControl.cxx`

G2BLineTop20 Object name. It's also possible to specify range of objects within an object or cell name via `CellMap`.

SS0 This is a two part reference. The first letter (S) implies that the item is considered to emmit neutrons. The second characters imply that this occurs from the position of the first source point defined above.

energyCut=0.0 Energy cut. Defines min energy for variance reduction (i.e. no modification will occur to energy bins below this value). However, if the number is negative, then if defines a -ve maximum energy past which no variance modification will be made.

scaleFactor=1.0 Linear scale factor in equation **??** It's transport to go from *a* cell to another cell. Scale factor is taken out in the mesh based variance reduction because it is identical in effect to the command [-wwgNorm scaleFactor 1.0]

densityFactor=0.9 Density scale factor. Normally we set it a bit less than one to make it easier to do transport through thick layers

rScale=1.0 Length is adjusted to a different scale effect

r2Pow=2.0 Scale for the length exponent in the distance effect. It should be increased in high absorbing regions and decreased in regions that are effectively in a forward going direction for the higher energy scattering (effectively mimicking forward angle scattering)

–voidUnMask The unpopulated world is made of sphere of importance 0 both inside and out. All objects that are places in the world must go in the inner spherical volume and they (unless specified) will have importance 1.0. In the case that the tally extends into this inner region sphere (e.g. for a dose after a shielding wall) must have either (a) a void volume added to the model or (b) the inner part of the world shere set to importance zero. This flag does option (b) [This flag may also be used to allow cross talk between two beamlines for example]

Notes

- In order to set up biasing, you need at least one source, if you one or more tally (adjoint) points then you must have at least one source.

- Source can be defined either as a point (via **–weightSource**) or a plane (via **–weightPlane**). In the example above we defined point source. However the defined sources and planes do nothing until you use them in **–weightObject**.

- General rule to get the things into work: try different setups and compare results. In particular, higher energy transport is forward going and it is easy to over weight the splitting when particle would predominately scatter in a forward direction. This can be overcome by either reducing the r^2 power or decreasing the density. Similarly, resonance streaming dominates thick shielding transport but the code does not account for that, so to approximate the lower attenuation coefficient the density can be reduced using the density factor. Finally, for H_2 systems (e.g. polyethelene shielding) the high energy lose term for the hydrogen is not correctly weighted (CombLayer uses single group approximation) and additional r^2 or Rlength scaling is worth cosidering.
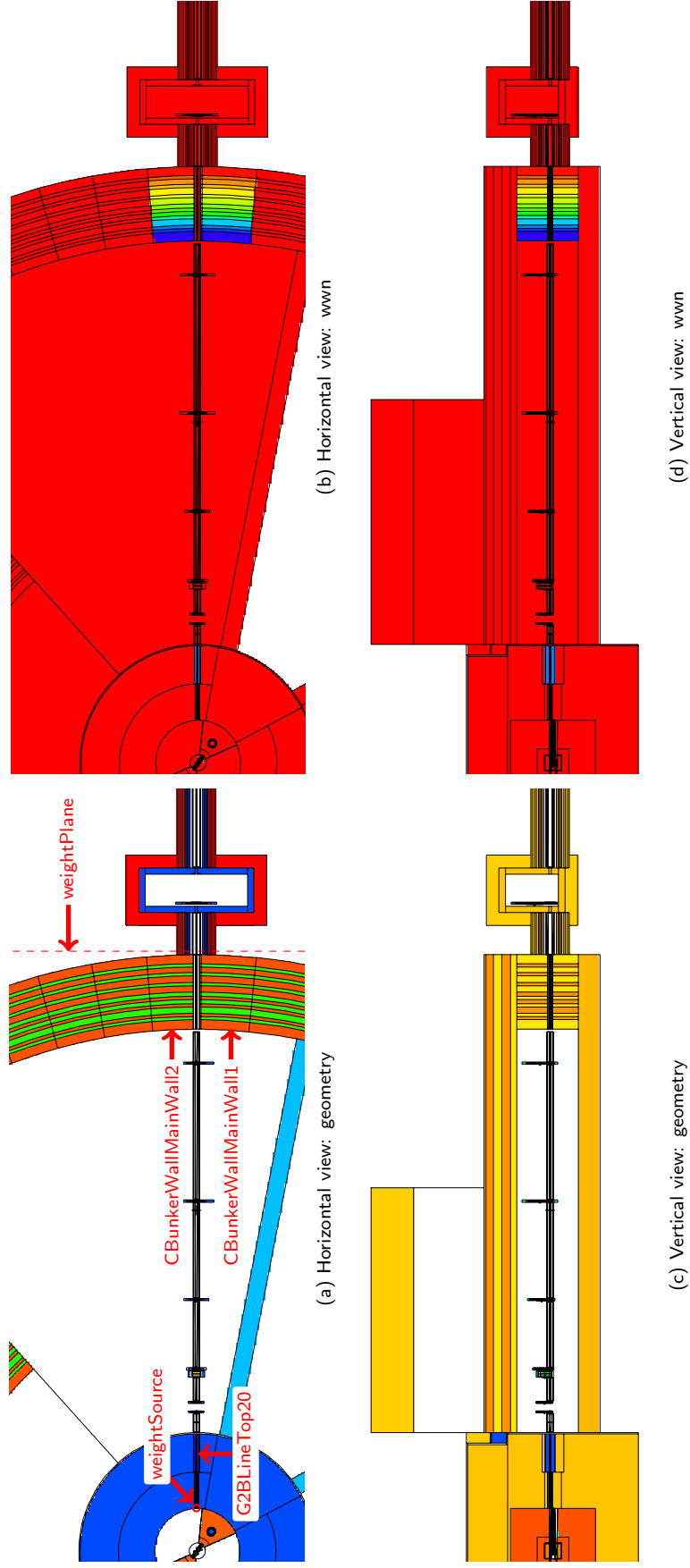
(a) Horizontal view: geometry

(b) Horizontal view: wwn

(c) Vertical view: geometry

(d) Vertical view: wwn

Figure 8: Cell-based weight window

8 Examples

9 Example 1

9.1 Purpose: Creating the initial world

This example is the first stage on the process of building a full model. The only purpose of this example is to create an unpopulated world (a large void sphere) and generate an executable that can output a valid MCNP(X), FLUKA or PHITS output.

9.2 Creating required files

Assuming that we are going to build a new project that does not currently exist and for this case, we will call it *example* [10].

9.2.1 Create a new main() function

Move into the Main directory:

Choose an existing project and copy it – in this case copy the *xrayHut.cxx* file into *example.cxx*

9.3 Description of the main file

The next step is to edit the main file and chenage it from working with the xrayHut model and using your new example model. First let use overview the different parts of the file Main/example.cxx.

Consider main file *Main/example.cxx*:

- **includes [lines 22-58]** : This are all the standard include. They allows the construction of the makeExample class. CombLayer uses an independent include file system in that no .h file contains any other include statement.

- **ELog construction [lines 60-67]** : This is standard static constructors for the controlled streams (they are like std::cout but have more control and colour output). Don't touch!

- **activateLogging [lines 75]** : This activates the logging system, that will give output with the class and method that output came from. It also gives a full calling structure of exceptions. You update this area if you want more (less) output.

- **setMaterialDataBase [lines 77]** : CombLayer provides a list of named (not only numbers) materials that the user can use. This is where they are populated. Addional materials can be added in this function or separate functions if needed.

- **createSimulation [lines 78]** : This does NOT create the model but it produces the empty simulation of the correct type e.g. MCNP or FLUKA. This is controlled by the inputs on the command line (e.g –FLUKA).

- **createInputs [lines 86]** : This function tells the CombLayer executable what input flags (e.g. –tally) will be recognised by the code. The call to createInputs creates the general input flag (e.g. –FLUKA) that are common to all CombLayer executables, but if you need/want any special calls for your code, then create a new funtion here (e.g. createExampleInputs) and typically that function will call createInputs adding the

---

[10] The completed version of this test can be obtained by checking out the example1 branch of CombLayer

special extra parts you need. This code only registers that these flags will be accepted on the command line. Parsing and acting on this input will be handled later.

- **exampleVariables [lines 93]** : This is a user written function. It populates all the variables for the model that you are building. Every variables should be defined by this function.

- **makeExample BOj [lines 95-96]** : This creates the class which is going to construct the whole geometric model. Doesn't matter if this is simple (as this case) or super complex, just one call.

- **buildFullSimulation [lines 100-104]** : This does the construction of the output file (based normally on inputs etc). It constructs tallies/estimators and applies variance reduction etc

- **Catch [lines 108-124]** : This is the catch block if errors of various types are generated. In principle, you sould never see a General Exception except from your code but... while *ExitAbort* is for conditions that the code has not yet been written to deal with (e.g. Torus intersections) and *Exception failure* (the commonest error) is for things that the code can't process in respect to inputs (e.g. setting the radius of a cylinder to a negative number).

- **delete and return [lines 126-128] : These are the clean up lines to free memory and exit with appropiate error number (0 for success).**

9.4   Changes for the main file

Edit example.cxx and make standard changes because we don't want name overlap. In the *#include*'s change

- *#include "xrayHutVariables.h"* → *#include "exampleVariables.h"*

- *#include "makeHutch.h"* → *#include "makeExample.h"*

- *setVariable::xrayHutVariables(SimPtr-¿getDataBase());* → *setVariable::exampleVariables(SimPtr-¿getDataBase());*

- *exampleSystem::makeXrayHut BObj;* → *exampleSystem::makeExample BObj;*

References

[1] Alireza Haghighat and John C. Wagner. Monte Carlo variance reduction with determenistic importance functions. *Progress in Nuclear Energy*, 42(1):25–53, 2003.

[2] Madicken Munk, R.N. Slaybaugh, Tara M. Pandya, Seth R. Johnson, and T. M. Evans. FW/CADIS-Ω: an angle-informed hybrid method for deep-penetration radiation transport. arXiv:1612.00793, Dec 2016.