CombLayer Guide

Stuart Ansell

August 15, 2017

Contents

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.

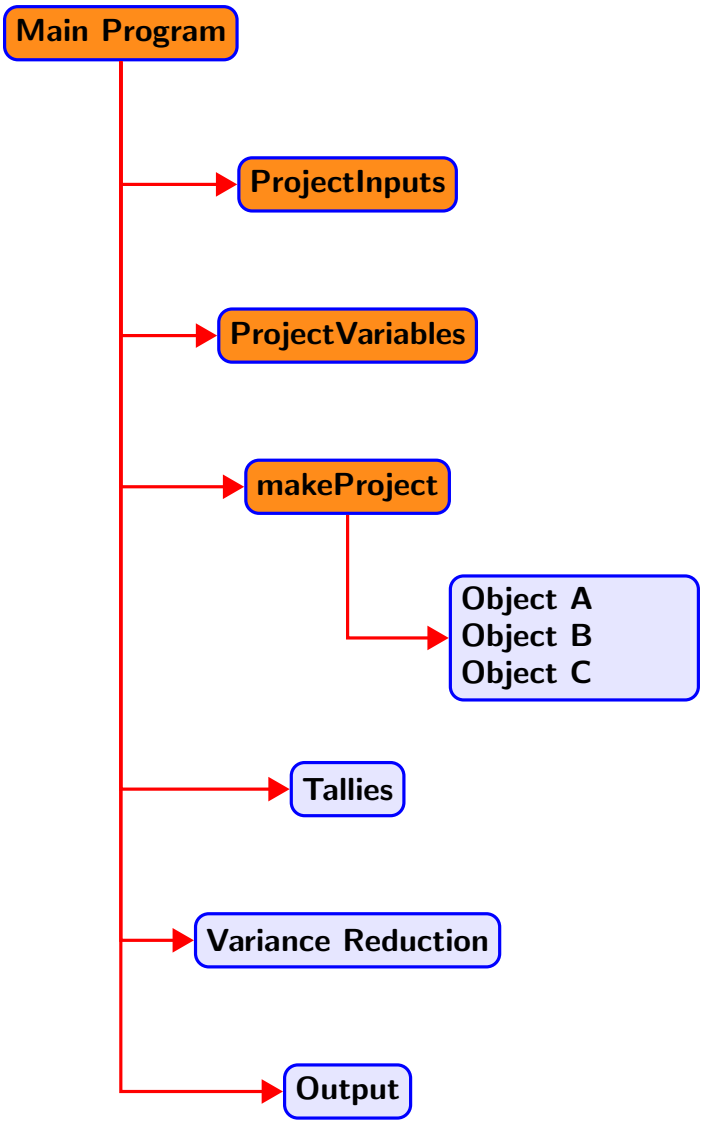


Figure 1: The main program calling sequence is shown. The parts in orange, are expected to be constructed by the user. Bespoke objects can be added for a project but it is not necessary.

2.1 Main

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

```
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 predominately written for the Linux platform using C++ compilers that support C++11 or greater. The code is available from
https://github.com/SAnsell/CombLayer,
either as a download of a zip file or by cloning/pulling the git repository.

3.1 Requirments

CombLayer needs to have the GNU Scietific Library [GSL] and the boost::regex system along with the STL libraries from your C++ compiler. The GSL can be avoided with the -NS flag in the getMk.pl and the CMake.pl script but some functionality will be lost, particularly in the choice of variance reduction methods.

Additionally, the primary build system uses cmake. There is another that just uses make but is significantly more time-consuming.

Functional documentation is supported using Doxygen and the construction of new cmake text files can be done via PERL scripts.

Currenly it is know that gcc version 4.6 and above can compile CombLayer as can clang (all tested versions). gcc 4.4 which is often the default on RedHat systems (2015) does not work.

3.2 Basic build method

If a clean directory is made and then the .zip file is uncompressed, the following commands should build a version of CombLayer.

```
./CMake.pl
cmake ./
make
```

This should make a number of executables, e.g. ess, simple, fullBuild etc. These can be used to make a simple model with commands like

```
./simple -r AA
```

This will produce an output file AA1.x which is a MCNP model.

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. TwinComp 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*.

5.1 makeModel

The makeModel object is the place that creates, initializes and manages inquires for the instances of all the geometric components. Primary objects need to be created and registered with the objectRegister **??**. The makeModel component is

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.

5.2 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

5.3 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.

5.3.1 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.

- 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). *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.

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. Loop over each cell in Mug : MCell
   (a) Calculate intersection of each surface:surface:surface triplet from the set of CC surfaces and MCell surface
   (b) If a point is within CC and the MCell exclude the CC from the MCell and goto next cell

Thus the spoon is inserted only into those cells of Mug which it intersects.

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 a bit 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.

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. It is possible to add a vector of link points to check as a parameter to limit the search.

5.4.4 addToInsertLineCtrl

Imagine we have a (big) contained component (Mug) and some (small) object which clips it (Spoon). The link points are **not** in the Mug (therefore **addToInsertControl** can not be used), but the lines which connect them are in the Mug. The method checks the lines connecting the link points and sorts out the intersections.

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.

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 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`.

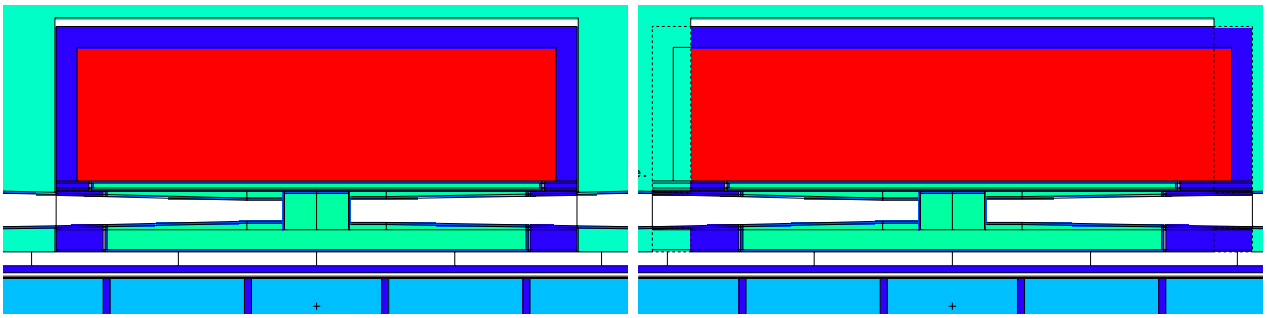
Dealing with dependent variables If some other variables depend on the variable you are going to change, it can break the geometry. Consider, for instance, changing the BeRef radius from the baseline value of 34.3 to 40 cm:

```
./ess -r modelOut
```

this generates the 34.3 cm model[1] as shown in Fig. 2a.

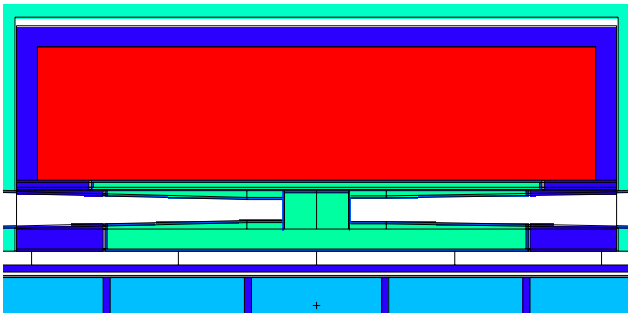
Now let's set it to 40 cm:

[1] 34.3 cm is the baseline BeRef radius in the commit `e47bf6d`.



(a) Baseline radius of 34.3 cm

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



(c) `BeRefRadius` increased to 40 cm taking into account dependent variables

Figure 2: Geometries with different BeRef radii

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

This geometry is shown in Fig. 2b and it is broken since now `BeRef` intersects with the Bulk component.

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 $40 + 3 + 0.2 = 43.2$ cm:

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

which produces the correct geometry shown in Fig. 2c.

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 FW-CADIS

We consider our result R to be a convolution of some flux $\Psi(\vec{P})$ determined by all the parameters $\vec{P}$ (position $\vec{r}$, energy E, angular coordinate $\Phi\vec{\Omega}$) and a determenistic cross section $\sigma_D(\vec{P})$ that relates to that:

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

That allowes 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 eV and 2 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 transport equations can be written as:

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

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:

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

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:

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

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

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 tha the flux must be the same is we assume reversibility.

We can [with a lot of algebra/Mathematica] show that they obey the identity

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

and therefore

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

where $<>$ is the inner product, which is the integral over $\vec{P}$.

Now if we set $q^* \equiv \Sigma_d$, then we have check the '{' symbol and 'q' in the line below

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

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 adjoined 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 (Where did we lose Φ?):

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

where $f(\vec{r}, E, \Omega)$ is an approximation of what? from the CADIS paper. They can find an approximation for this based on the true weight of the number of particles in the given cell (times the Monte Carlo density).

Explain

CADIS is the name of the program they wrote.

First, we need to find Ψ, but we are not going to find it because it means we need to run full simulation. Therefore we will only 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 does not support so many (SA says he tried 7).

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, as long as you have an adjoint point to go with it. 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 normalized. 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 normalization. However, due to the need to issues like doing a quick and simple simulation were a source is approximated or multiple tallies normalization may be required. For example if a simulation is started from the ESS target but the flux down a beamline 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-normalized the whole variance reduction mesh by adding the flag `--wwgNorm AValue BValue` such that the new normalization 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[2].

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}.

[2] See the `--wwgCalc` flag in Sec. 7.3.1.

7.3 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. SA had modified his MCNP version to allow both at the same time.

7.3.1 Mesh-based variance reduction

Git hash: 7348754

```
ess -r -w -wWWG \
--weightSource 'Vec3D(1000.0, 0.0, 14.0)' \
--weightSource 'Vec3D(1500.0, 0.0, 14.0)' \
--weightPlane 'Vec3D(1500.0, 0.0, 14.0)' 'Vec3D(-1, 0, 0)' \
--wwgCalc SS0 1.0 \
--wwgCalc TP0 -2 1 2 -4\
--wwgCADIS 'Vec3D(1000.0, 0.0, 14.0)' \
--wwgXMesh 1000 40 1600 \
--wwgYMesh -50 10 50 \
--wwgZMesh -100 20 100 \
--wwgE 1.0 \
--wwgVTK testWWG.vtk \
  a
```

-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 should be used. Here we defined two point sources.

–wwg[XYZ]Mesh defines a mesh (if we need it). Format: min nbins max.

–wwgE defines energy grid with MCNP notation (below 1 MeV). There is no variance reduction above last energy bin.

–wwgVTK optionally you can output the mesh file into a VTK file for plotting.

–wwgCalc define source and adjoined points. We must define at least one source and *at least* one adjoined point. Syntax:

SS0 First 'S': true source point. 'S0': first point in the list.

1 means energy range above 1 MeV (-1 would mean below 1 MeV).

TP0 First 'T': adjoined point (tally). 'P0': from the 1st plane in the list.

-2 define energy grid. Negative number: up till this energy (cut-off energy).

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

2 the power of r^2. SA does not find this parameter very useful.

1 absolute minimal weight you can have. If the number is negative, it means the exponent (10^{-n}). If it's not negative, it must be between 0 and 1 (remember that weights can not go above 1).

–wwgCADIS Finally you better do the CADIS summation — the integral over the whole thing. The final weight value has to still come from the intrinsic integral point, and that should be the source point in order to do normalisation correctly (SA believes it's semi-redundant, but he is still using it anyway. In principle one should integrate all the source points, weight them and not need this coordinate, but currently SA take one point approximated. It will be changed in the future.)

The source terms must cover the full energy range. It's completely acceptable that an individual source term does not cover full energy range, but somewhere you must have at least one other source term that covers the rest.

When you run it, the output is very verbose:

```
...
CADIS norm[7700]: -46.0517 0.693147 == 1577.5 5 -95          WWGWeight::CADISnorm
CADIS norm[7800]: -46.0517 0.693147 == 1592.5 -45 -95        WWGWeight::CADISnorm
CADIS norm[7900]: -46.0517 0.693147 == 1592.5 5 -95          WWGWeight::CADISnorm
sumR== -7.37277 0.000628129                                  WWGWeight::CADISnorm
Min == 0.000314064 0.000314064 1                                 WWG::updateWM
Warning : No WWG normalization step                       WWGControl::wwgNormalize
```

Every 100 points during integration it writes you the ratios of contribution of flux to a given point and the source to the given point. So, it finds that you get about 10^9 (0.693147) contributions from the source at the start of the wall, but once you get through the wall you get 10^{46} (-46.0517) contribution. You end up with the minimum weight (which is in fact the maximal weight ???), which is the normalisation factor.
sumR are normalisation terms and its value (-7.37277) says we are out of normalisation a bit (therefore we can maybe do something with it).

1e9 does not correspond to the output