User Guide - CLASS v5

Core Library for Advanced Scenarios Simulations

CNRS/IN2P3
Release date : 03/05/2018

Contents

Part I

Introduction

Figure 1: CLASS logo

The code CLASS (**C**ore **L**ibrary for **A**dvanced **S**cenario **S**imulation) is a dynamic fuel cycle simulation tool developed by CNRS/IN2P3 (Centre National de la Recherche Scientifique / Institut National de Physique NuclÃ©aire et de Physique des Particules) in collaboration with IRSN (Institut de Radioprotection et de SÃ»retÃ© NuclÃ©aire). The aim of the tool CLASS is to model an evolving electro-nuclear fleet. The main output is the evolution of isotopes in all facilities of a nuclear fleet. The reactor physics is located in the treatment of the fuel fabrication and its depletion in reactor. The code CLASS aims to be a useful tool for scenarios studies. CLASS main asset is its ability to implement any kind of reactor, either the system is innovative or standard. Indeed, the opportunity is given to each user to **build his own reactor model**.

Part II

First Steps

Chapter 1

Package Content

The CLASS package contains the followings :

- **data/** : folder containing nuclei properties

  FPyield_Fast_JEFF3.1.dat : file containing fission yield for fast reactors

  FPyield_Thermal_JEFF3.1.dat : file containing fission yield for thermal reactors

  Mass.dat : file containing molar masses

  SpontaneousFPyield.dat : file containing spontaneous fission yields

  chart.JEF3T : file containing decay constants and branching ratios

  HeatTox.dat : file containing conversion data for heat and radiotoxicity calculation

- **DATA_BASES/** : this folder contains decay data base and reactor data bases

  DECAY/ : decay data base

  FBR-Na/ : Models related to Fast reactor

  PWR/ : Models related to Pressurised Water Reactor

  REP_HFC/ : Models related to low moderated Pressurised Water Reactor

  ADS/ : Models related to Accelerator Driven System

- **documentation/**

  Manual/ : folder containing this user guide an its .tex sources

  Doxygen/ : folder containing the builded doxygen and its generation configuration

- **example/** : folder containing simple examples of CLASS input and an example of CLASS ouput reader

- **gui/** : folder containing sources of the graphical user interface for CLASS outputs

- **lib/** : folder containing the CLASS library (once compiled)

- **bin/** : folder containing the CLASSGui and the Google test binary (once compiled)

- **source/** : folder containing CLASS sources

  include/

  Model/ : folder that contain the sources related to the physics models (EquivalenceModel , XSModel and IrradiationModel)

  src/

  External/ : folder that contain the sources related to external classes used by the code CLASS

- **Utils/** : folder containing utility software related to reactor data base generation

  EQM/ : Example of software to generate equivalence model for building reactor fuel

  MURE2CLASS/ : Software to convert MURE (a fuel depletion code) output to EvolutionData format

  XSM/ : Software to generate cross section predictor

ROOT2DAT/ : Software to convert CLASS output into readable ASCII data file

ROOT2ROOT/ : Software to convert multiple CLASS Output into one single ROOT file (used for sensitivity analysis)

Chapter 2

Install procedure

2.1 Requirement

- User skills : Good knowledge of C++. Abilities in using Root[1]. Experience in depletion codes and neutron transport codes is required for building complex new reactor model.
- OS : CLASS is known to work under Linux (64 bits) and MacOSX (64 bits). It has never been tested on any Windows distribution.
- C++ compiler : we recommend to use a gnu compiler like gcc4.8 or above. For OSX, CLASS is working with native clang compiler.
- For DARWIN (OSX) users : Make sure you have installed XCode and its command line tools (if not dowload and install from AppStore). If using clang compiler, there's no possibility to use openmp.
- Root (CERN) : ROOT [Brun 97] is an analysis software developped by CERN. CLASS version 5 uses some ROOT version 6 features to run. CLASS uses Root to store output data. The graphical user interface CLASSGui is also based on Root. Some algorithms uses the TMVA module of Root.

2.2 Installation

2.2.1 Get the source from archive

Download the source of CLASS at the following adress : gitlab link. The archive is available at the location showed on figure2.1.

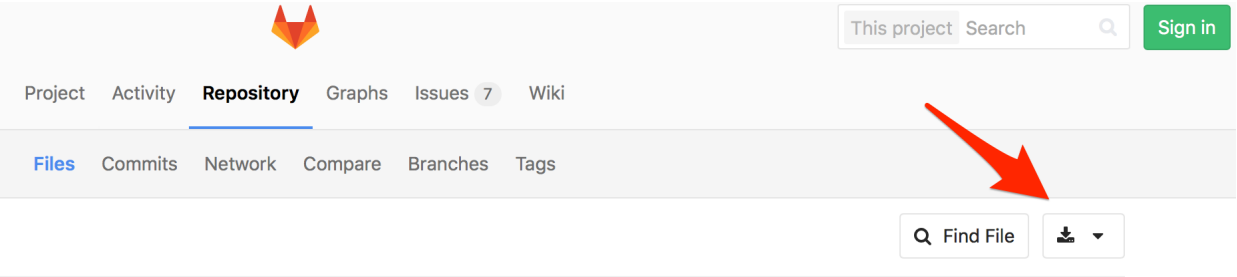

Figure 2.1: Archive for source of CLASS

2.2.2 from git public repository

It is also possible to clone the git repository with the following commands :

```
cd YourClassDirectory
git clone git@gitlab.in2p3.fr:sens/CLASS.git
```

[1] https://root.cern.ch/

2.2.3 CLASS Compilation

Run the following command for a complete install with associated google tests :

```
./install.sh --clean-build -gtest
```

Chapter 3

CLASS Execution

CLASS is a set of C++ libraries, there is no CLASS binary file. A CLASS executable has to be build by user using objects and methods defined in the CLASS package.
The compilation line for generating your executable from a .cxx file is the following : (You can find CLASS input examples in $CLASS_PATH/example/)

```
g++ -o CLASS_exec YourScenario.cxx -I $CLASS_include -L $CLASS_lib -lCLASSpkg `root-config
    --cflags` `root-config --libs` -fopenmp -lgomp -Wunused-result -lTMVA
```

Then type

```
./CLASS_exec
```

The following should show in your terminal :

Figure 3.1: CLASS running in a shell

Chapter 4

News, forum, troubleshooting, doxygen ...

CLASS has a forge[1] hosted by the IN2P3 where you can find :

- A forum[2] where you are invited to post your trouble about CLASS installation and usage. You may find the answer to your trouble on a already posted thread.
- A doxygen[3] where all the CLASS objects and methods are defined and explained. Note that the doxygen is also contained in $CLASS_PATH/documentation/doxygen/CLASSDoxygen.html
- News[4] : All the news related to CLASS

A Mailing List[5] also exist in order to be warned of all the change inside CLASS and to allow user to exchange directly on the code. One can join the mailing list through the following link[6].

[1] https://forge.in2p3.fr/projects/classforge
[2] https://forge.in2p3.fr/projects/classforge/boards
[3] https://forge.in2p3.fr/projects/classforge/embedded/doxygen/inherits.html
[4] https://forge.in2p3.fr/projects/classforge/news
[5] classuser-l@ccpntc02.in2p3.fr
[6] http://listserv.in2p3.fr/cgi-bin/wa?SUBED1=classuser-l&A=1

Part III

CLASS : General overview

Chapter 5

Generalities

5.1 Basic unit

Time in CLASS should be written in second. A special C++ type has been defined for this purpose : cSecond. This is a **long long int**. Power is always considered as thermal power in watt. Masses are in metric tons except for molar masses (in gram). Burnup is in units of GWd/tHM. With tHM stands for metric tons of heavy metal.

5.2 CLASS working process principle

image : shéma de principe de class

Chapter 6
Facilities descriptions

All the facilities in CLASS are regrouped inside a mother class called CLASSFacility (and inherit of all the properties of the CLASSFacility in a C++ way). Inside the CLASSFacility, 3 different types has been defined, the Reactor, the FabricationPlant (or more generally, all the fuel cycle front-end facilities) and the facilities of the back end of the fuel cycle.

6.1 CLASSFacility

The CLASSFacility should never be used directly in the main CLASS program (the one made to perform the simulation). The aim of this object is to regroup all the common properties of the nuclear facilities, such as common variables, methods, and builder.

6.2 Reactor

6.2.1 Generalities

The aim of this class is to deal with the evolution of the fuel inside a reactor.
The evolution of the fuel is **always** contain in the EvolutionData *fEvolutionDB*.
There are 2 way to provide the EvolutionData to the reactor. In the case of fixed fuel[1] the user need to provide it, using the appropriated constructor, the set function, or a CLASSFuelPlan. In the case of recycled fuel or unfixed fuel, the user need to provide a PhysicsModels, using the appropriated constructor, the set function, and/or a CLASSFuelPlan.

6.2.2 Use

There are 2 main ways to define a reactor, depending on the type of fuel loaded.

6.2.2.1 Fixed Fuel

Reactor using fixed fuel, which load always the same fresh fuel, and irradiates it to the same burnup (same spent fuel composition), can be declared as follow:

```
Reactor *MyReactor = new reactor(aCLASSLogger,          // CLASSLogger
                                 myFuel_EvolutionData,  // EvolutionData
                                 aBackEnd,              // BackEnd
                                 myRe_StartingTime,     // Starting Time
                                 myRe_LifeTime,         // Time of Life
                                 myRe_Power,            // Power
                                 myRe_HeavyMetalMass,   // HM mass
                                 myRe_BurnUp,           // BurnUp
                                 myRe_LoadFactor);      // LoadFactor
```

or

[1] Always the same input/output isotopic composition.

```
Reactor *MyReactor = new reactor(aCLASSLogger,          // CLASSLogger
                                 myFuel_EvolutionData,  // EvolutionData
                                 aBackEnd,              // BackEnd
                                 myRe_StartingTime,     // Starting Time
                                 myRe_LifeTime,         // Time of Life
                                 myRe_CycleTime,        // Time of Cycle
                                 myRe_HeavyMetalMass,   // HM mass
                                 myRe_BurnUp);          // BurnUp
```

The meaning of each arguments of the two constructor previously defined are summed up in the following table

Table 6.1: Arguments of Reactor constructors

Argument	type	meaning	unit
aCLASSLogger	CLASSLogger	Output messages	N.A.
myFuel_EvolutionData	EvolutionData	Fuel evolution description	N.A.
aBackEnd	CLASSBackEnd	Facility getting the spent fuel	N.A.
myRe_StartingTime	cSecond	Creation time	second
myRe_LifeTime	cSecond	Operation time	second
myRe_Power	double	Thermal power	Watt
myRe_HeavyMetalMass	double	Heavy metal mass	tons
myRe_BurnUp	double	Burn up at EOC	GWd/tHM
myRe_LoadFactor	double	Fraction of nominal power	.
myRe_CycleTime	cSecond	the cycle time	second

6.2.2.2 Reprocessed Fuel

In this case, the fuel is provided by an external facility, so called, the FabricationPlant. The way to build the reprocessed fresh fuel and to handle the fuel depletion calculation is done by the PhysicsModels. The main ways to defined a Reactor (with reprocessed fuel) are shown in the next two examples :

```
Reactor *MyReactor = new Reactor(aCLASSLogger,          // CLASSLogger
                                 myFuel_PhysicsModels,  // PhysicsModels
                                 aFabricationPlant,     // FabricationPlant
                                 aBackEnd,              // BackEnd
                                 myRe_StartingTime,     // Starting Time
                                 myRe_LifeTime,         // Time of Life
                                 myRe_Power,            // Power
                                 myRe_HeavyMetalMass,   // HM mass
                                 myRe_BurnUp,           // BurnUp
                                 myRe_LoadFactor);      // LoadFactor
```

or

```
Reactor *MyReactor = new Reactor(aCLASSLogger,          // CLASSLogger
                                 myFuel_PhysicsModels,  // PhysicsModels
                                 aFabricationPlant,     // FabricationPlant
                                 aBackEnd,              // BackEnd
                                 myRe_StartingTime,     // Starting Time
                                 myRe_LifeTime,         // Time of Life
                                 myRe_CycleTime,        // Time of Cycle
                                 myRe_HeavyMetalMass,   // HM mass
                                 myRe_BurnUp);          // BurnUp
```

The meaning of each argument of the two constructors previously defined are summed up in the following table

Table 6.2: Arguments of Reactor constructors

Argument	type	meaning	unit
aCLASSLogger	CLASSLogger	Output messages	N.A.
myFuel_PhysicsModels	PhysicsModels	Fuel construction/evolution	N.A.
aFabricationPlant	FabricationPlant	Facility building the fuel	N.A.
aBackEnd	CLASSBackEnd	Facility getting the spent fuel	N.A.
myRe_StartingTime	cSecond	Creation time	second
myRe_LifeTime	cSecond	Operation time	second
myRe_Power	double	Thermal power	Watt
myRe_HeavyMetalMass	double	Heavy metal mass	tons
myRe_BurnUp	double	Burn up at EOC	GWd/tHM
myRe_LoadFactor	double	Fraction of nominal power	.
myRe_CycleTime	cSecond	the cycle time	second

6.2.3 CLASSFuelPlan

A reactor may changes of fuel type during its lifetime. To handle this, the user can destroy the reactor and build a new one with an other kind of fuel. In order to make the process more flexible, the CLASSFuelPlan has been added to the CLASS package. The following example explains how to make a reactor to change its fuel type and burn-up.

```
Reactor* MyReactor = new Reactor(gCLASS->GetLog(),          //Log
                                 EvolutionData0,              // DB
                                 Stock,                    // BackEnd
                                 StartingTime,       // Starting time
                                 LifeTime,             // Time of life
                                 Power_CP0,         // Power
                                 HMMass,     // HM mass
                                 BU0,              // BurnUp
                                 0.8);  //Load factor

MyReactor->GetFuelPlan()->AddFuel( ChangingFuelTime0, EvolutionData1, BU1);
MyReactor->GetFuelPlan()->AddFuel( ChangingFuelTime1, PhyMod, BU2);
```

At *ChangingFuelTime0* the reactor changes its fuel from *EvolutionData0* to *EvolutionData1* and its Burn-up from *BU0* to *BU1*. At *ChangingFuelTime1*, the reactor uses reprocessed fuel using the PhysicsModels *PhyMod*.

6.3 CLASSBackEnd

The CLASSBackEnd class is a mother class which aims to regroup all common properties of the facilities of the back end of the fuel cycle.
A CLASSBackEnd does not control its upstream. Its incoming material flux is pushed by its upstream facility (a Reactor, or an other CLASSBackEnd). It only controls its downstream flux.
This object is not supposed to be used explicitly in a CLASS input.

6.3.1 Storage

Storage is a CLASSBackEnd without associated downstream factory. All the incoming material are stored individually in different IsotopicVector (see figure 6.1). During the storage, the depletion by decay is taken into account. The storage has to be defined as follow :

```
Storage *Stock = new Storage(aCLASSLogger);
```

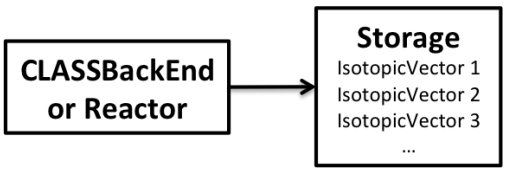

Figure 6.1: Storage

6.3.2 Pool

Pool is a CLASSBackEnd with an associated downstream factory. All incoming material will be pushed in the downstream factory after a certain cooling time. All the incoming material are stored individually in different IsotopicVector (the same way as the Storage) . During the cooling process, the depletion by decay is taken into account. The Pool has to be defined as follow :

```
Pool *MyPool = new Pool(aCLASSLogger, aCLASSBackEnd, 5*365.25*24.*3600);
```

In the previous example, a 5 years cooling time has been used. If no downstream facility is set, all the material will be sent, after the cooling time, to the WASTE of the Scenario. To do so :

```
Pool *MyPool = new Pool(aCLASSLogger, 5*365.25*24.*3600);
```

6.3.3 SeparationPlant

The role of the SeparationPlant is to separate an incoming IsotopicVector from a facility into an arbitrary number of outgoing CLASSBackEnd.
To define a SeparationPlant proceed as follow :

```
SeparationPlant* MySeparationPlant = new SeparationPlant(aCLASSLogger);
```

The separation process is instantaneous and it uses isotopic separation efficiencies. Efficiencies must be given as an IsotopicVector containing the separation efficiency for each nucleus. Note that it is possible to separate the incoming IsotopicVector in many, the users must provide as many isotopic separation efficiency as outgoing CLASSBackEnd.
In addition of an outgoing CLASSBackEnd and an associated isotopic separation efficiency, the user must provide a date for the separation to be effective. To do so :

```
IsotopicVector IV_MA; //Define Minor Actinides (MA) separation efficiencies
IV_MA.Add(93, 237, 0, 1.);
IV_MA.Add(95, 242, 1, 1.);
IV_MA.Add(96, 245, 0, 1.);
//...
MySeparationPlant->SetBackEndDestination(aCLASSBackEnd1 //destination of MA
                                         IV_MA, //Efficiencies
                                         2000*365.25*24.3600);//Time when the
                                             separation begin

IsotopicVector IV_Pu; //Defined Plutonium separation efficiencies
IV_Pu.Add(94, 238, 0, 0.8);
IV_Pu.Add(94, 239, 0, 0.8);
//...
MySeparationPlant->SetBackEndDestination(aCLASSBackEnd2,
                                         IV_Pu,
                                         2005*365.25*24.3600);

IsotopicVector IV_U;
IV_U += 0.5*ZAI(92, 235, 0);
IV_U += 0.5*ZAI(92, 238, 0);
//...
MySeparationPlant->SetBackEndDestination(aCLASSBackEnd3,
                                         IV_U,
                                         2015*365.25*24.3600);
```

In the present example defined above, the separation of Minor Actinides start in 2000, this separated material is sent to the CLASSBackEnd *aCLASSBackEnd1* (the rest goes to the WASTE). The separation of the plutonium start in 2005 (the separated Pu is sent to *aCLASSBackEnd2*) and the separation of uranium take place in 2010.
Note that between 2005 and 2010, both MA and Pu are separated and sent respectively to *aCLASSBackEnd1* and *aCLASSBackEnd2*, all the remaining isotopes are sent to the WASTE. After 2010, MA, Pu and U are separated and sent to their respective CLASSBackEnd facilities, the rest is still sent to WASTE.
Furthermore, the separation of Actinides Minor has an efficiency of 100%, Pu of 80% and U of 50%. Please refer to $CLASS_PATH/example/Separation.cxx for a simple CLASS input using the SeparationPlant.

6.4 Fabrication Plant

The FabricationPlant is the facility which takes care of the fuel fabrication. The "action" in FabricationPlant appends before the beginning of cycle of a reactor: One fabrication time (Fabrication duration) before the BOC, the building process of the fuel start.
First, the FabricationPlant sorts the different IsotopicVectors in the different inputs Storage according to the user priorities. Then, it asks the EquivalenceModel of the PhysicsModels associated to the reactor how to build a fuel with the correct properties using the available IsotopicVectors contained in the Storage. The EquivalenceModel provide a list of fraction to take in each IsotopicVectors in the Storage . According to this fraction list, the FabricationPlant takes the fraction in each IsotopicVector and build the reprocessed fuel. Once the reprocessed fuel is made, it asks the PhyscisModel to calculate its depletion and store the result in an EvolutionData. The reactor takes this EvolutionData from the FabricationPlant at its begining of cycle.
Between the fuel fabrication and the loading of the fuel in the reactor, the depletion of the fresh fuel by decay is taken into account.
Note that, the FabricationPlant provide to the EquivalenceModel a list of stock which has virtually decayed for the fabrication time.

To setup a FabricationPlant do as follow :

```
FabricationPlant *MyFabricationPlant = new FabricationPlant(gCLASS->GetLog(), 1*
    year);
MyFabricationPlant->SetFiFo();
```

In the previous example, the SetFifo() method set the first in first out priority for the stock usage. It means that the older IsotopicVector of the Storage is taken in priority by the FabricationPlant. If the younger IsotopicVector is wanted to be taken in priority : one should use SetFiFo(false).

The Storage used to extract the fissile part of the fuel is set using :

```
MyFabricationPlant->AddFissileStorage(Stock);
```

And if necessary it is possible to define a Storage where fertile isotopes will be extracted, using :

```
MyFabricationPlant->AddFertileStorage(Stock);
```

If no fertile Storage are defined, the fertile part is taken from outside of the Scenario. By default the unused part of the stock is sent to WASTE. But it is possible to set a storage where the unused part of the stock will be stored, using :

```
MyFabricationPlant->SetReUsableStorage(Stock);
```

Please refer to $CLASS_PATH/example/CloseCycle.cxx for a simple CLASS input using the FabricationPlant .

6.5 Pathway between Facilities

As explain previously, there are 3 different facility family, the FabricationPlant, the Reactor, and the CLASSBackEnd. All the facilities of type CLASSBackEnd can't get material from other facilities by itself. It is always an other facility which sends material in the CLASSBackEnd. On another hand, some CLASSBackEnd facilities can send material inside other facilities: the SeparationPlant and the Pool. The Storage can only store materials.
The reactor takes its fuel from a FabricationPlant and sends the irradiated fuel in a CLASSBackEnd. The FabricationPlant takes its materials from a storage and stored the reprocessed fuel until the beginning of cycle of the Reactor. In the following, 4 examples of pathways between facilities are listed. These examples are here to illustrate the possible pathways. These examples are not exhaustive. Furthermore, almost any composition between these examples could be made.

6.5.1 Reactor with fixed fuel and a Storage

Please refer to the CLASS input $CLASS_PATH/example/SimpleReactor.cxx

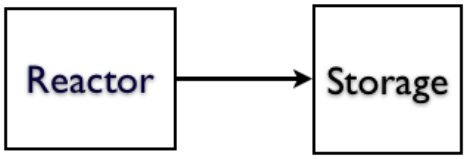

Figure 6.2: Shematic Pathway

```
CLASSLogger *Logger = new CLASSLogger("CLASS_OUTPUT.log",1,2);
EvolutionData* myFuel_EvolutionData = new EvolutionData(Logger, "/PATH/EvolData.
    dat");

Storage* MyStorage = new Storage(Logger);

Reactor *MyReactor = new Reactor(Logger, myFuel_EvolutionData, MyStorage, 0,
    40*365.25*24.3600, 900E6, 100, 45, 1);
```

6.5.2 Reactor with fixed fuel, a Pool and a Storage

Please refer to the CLASS input $CLASS_PATH/example/SimpleReactor2.cxx

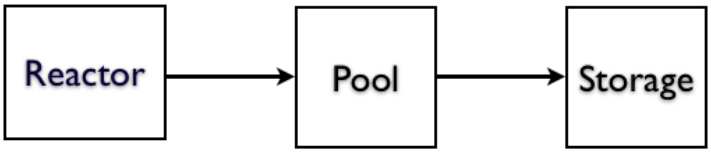

Figure 6.3: Shematic Pathway

```
CLASSLogger *Logger = new CLASSLogger("CLASS_OUTPUT.log",1,2);
EvolutionData* myFuel_EvolutionData = new EvolutionData(Logger, "/PATH/EvolData.
    dat");

Storage* MyStorage = new Storage(Logger);
Pool* MyPool = new Pool(Logger, MyStorage, 5*365.25*24*3600);

Reactor *MyReactor = new Reactor(Logger, myFuel_EvolutionData, MyPool, 0,
    40*365.25*24.3600, 900E6, 100, 45, 1);
```

6.5.3 Reactor with fixed fuel, two SeprationPlant, a Pool and four Storage

Please refer to the CLASS input $CLASS_PATH/example/Separation.cxx

```
CLASSLogger *Logger = new CLASSLogger("CLASS_OUTPUT.log",1,2);
EvolutionData* myFuel_EvolutionData = new EvolutionData(Logger, "/PATH/EvolData.
    dat");

Storage* MyStorage1 = new Storage(Logger);
Storage* MyStorage2 = new Storage(Logger);
Storage* MyStorage3 = new Storage(Logger);
Storage* MyStorage4 = new Storage(Logger);

Pool* MyPool1 = new Pool(Logger, MyStorage1, 5*365.25*24*3600);

// SeparationPlant separate U5 from U8 which goes in Storage 3 and 4.
SeparationPlant* MySeparation1 = new SeparationPlant(Logger);
IsotopicVector IV_U8;
IV_U8.Add(92, 238, 0, 1);
MySeparationPlant1->SetBackEndDestination(MyStorage3, IV_U8, 0);

IsotopicVector IV_U5;
IV_U5 += 1*ZAI(92, 235, 0);
MySeparationPlant1->SetBackEndDestination(MyStorage4, IV_U5, 0);


// SeparationPlant separate Am Pu and U which goes respectively in myPool1,
    myStorage2 and mySeparationPlant1.
SeparationPlant* MySeparation2 = new SeparationPlant(Logger);
IsotopicVector IV_MA;
IV_MA.Add(95, 242, 1, 1.);
MySeparationPlant2->SetBackEndDestination(MyPool1, IV_MA, 0);

IsotopicVector IV_Pu;
IV_Pu.Add(94, 239, 0, 0.8);
MySeparationPlant2->SetBackEndDestination(MyStorage2, IV_Pu, 0);

IsotopicVector IV_U;
IV_U.Add(92, 238, 0, 0.5);
IV_U.Add(92, 235, 0, 0.5);
MySeparationPlant2->SetBackEndDestination(MySeparationPlant1, IV_U, 0);

Reactor *MyReactor = new Reactor(Logger, myFuel_EvolutionData, MySeparation2, 0,
    40*365.25*24.3600, 900E6, 100, 45, 1);
```

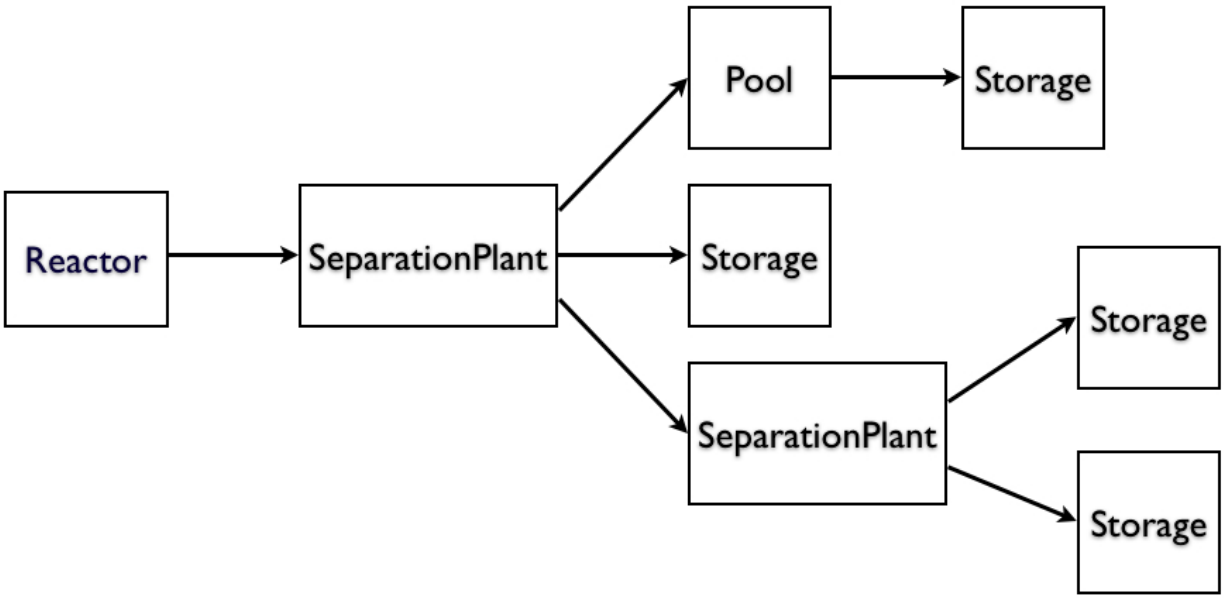

Figure 6.4: Shematic Pathway

6.5.4 Reactor, a FabricationPlant, a Pool and a Storage

Please refer to the CLASS input $CLASS_PATH/example/CloseCycle.cxx

```
CLASSLogger *Logger = new CLASSLogger("CLASS_OU TPUT.log",1,2);

IM_RK4 *IMRK4 = new IM_RK4(Logger);
EQM_LIN_PWR_MOX* EQMLINPWRMOX = new EQM_LIN_PWR_MOX(Logger, "/PATH/EQ_Lin.dat");
EQM_QUAD_PWR_MOX* EQMQUADPWRMOX = new EQM_QUAD_PWR_MOX(Logger, "/PATH/DBParam.dat"
    );
PhysicsModels* myFuel_PhysicsModel = new PhysicsModels(XSMOX, EQMQUADPWRMOX, IMRK4
    );

Storage* MyStorage = new Storage(Logger);
Pool* MyPool = new Pool(Logger, MyStorage, 5*365.25*24*3600);

FabricationPlant* myFabrication = new FabricationPlant(Logger, MyStorage,
    2*365.25*24*3600);

Reactor *MyReactor = new Reactor(Logger, myFuel_PhysicsModel, myFabrication,
    MyPool, 0, 40*365.25*24.3600, 900E6, 100, 45, 1);
```

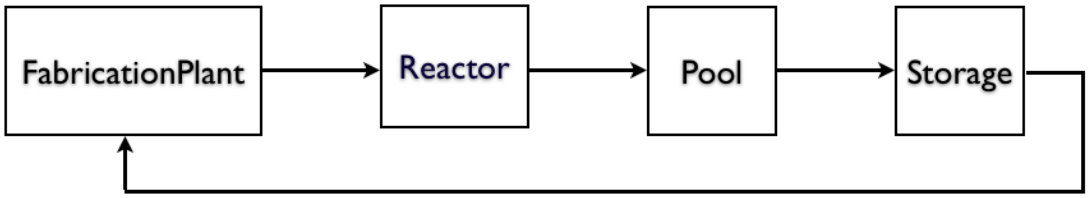

Figure 6.5: Shematic Pathway

Chapter 7

Other objects

7.1 ZAI

The ZAi object represents a nucleus, from its charge number, mass number and isomeric state. The object save the charge number Z, the mass number A and the isomeric state I of a nucleus : I=0 for ground state , I=1 for the first isomeric state ...
To declare a ZAI object proceed as follow :

```
ZAI U238 = ZAI(92, 238, 0);
```

This class includes the mains logical comparators (*e.g* ==, >, !=). Fill free to read the doxygen for more details on the methods associated to this class. (*e.g* A(), Z(), I(), N()...) .

7.2 IsotopicVector

7.2.1 Generality

The IsotopicVector object is a collection of ZAI. For each ZAI a quantity of nuclei is associated (IsotopicVector is a c++ map of ZAI and double, which corresponds to a sorted array of ZAI and its quantity).
Two main operations have been defined in the IsotopicVector class. The following illustrates the possible operations allowed for IsotopicVectors :

Definiton & Addition of nuclei

```
IsotopicVector IV_1;
IsotopicVector IV_2;

IV_1 += 23 * ZAI(92, 238, 0); // Add 23 nucleus of uranium 238 to ZAI_1
IV_1.Add(92, 235, 0, 52);     // Add 52 nucleus of uranium 235 to ZAI_1
```

Multiplication

```
IV_1 *= 100; // Multiply all the nuclei quantities by 100 -> resulting : 2300
    uranium 238 and 5200 uranium 235

IV_2 = IV_1 * 10; // IV_2 will be equal to 10 IV_1
```

Sum

```
IsotopicVector IV_sum = IV_1 + IV2; // IV_sum will be equal to 11 IV_1
```

Some additional operations have been also implemented, such as subtraction. It works as the sum, but if the result of the subtraction is negative for some nuclei, those nuclei are set to zero and the difference is added to the, so called, *fIsotopicQuantityNeeded*. If so, a warning will be written in the standard output : the terminal (see section 7.4).

7.2.2 Print method

You can use the Print() method to write the composition of an IsotopicVector. This method print all the quantities of all the ZAI present in the IsotopicVector (unit: quantity of nuclei).

7.2.3 GetTotalMass

Return the mass of the IsotopicVector **in tons** using :

```
double TotalMass = IV.GetTotalMass();
```

7.2.4 Multiplication between IsotopicVector

The result of this operation is an IsotopicVector, where each nucleus quantity is the product of the corresponding nucleus quantity of the two IsotopicVector.
In other words :
If a nucleus A is present in both IsotopicVector, with respective quantity α and β, the resulting IsotopicVector will contain $\alpha \times \beta$ nucleus A. If the nucleus A is not present in both IsotopicVector, the resulting IsotopicVector will not contain the nucleus A, as follow equation (7.1) :

$$IV_3 = IV_1 \times IV_2 = \sum_{i \in (IV_1 + IV_2)} (n_{1i} \times n_{2i})\, ZAI_i \tag{7.1}$$

By exemple, this method can be used to apply separation efficiency: one IsotopicVector containing real material and the other one containing separation efficiency of each nucleus.

7.3 EvolutionData

An EvolutionData aims to describe the evolution of an IsotopicVector through a physical process (decay or irradiation). The decay case is fully described in section 7.3.2.

In case of irradiation, it may also contains the evolution of the one group cross section. The evolution of the neutron flux and of the keff can be supplied but its not mandatory. The EvolutionData MUST contain the power and the heavy metal mass and it can contain the fuel type, reactor type and the cycle time.
These EvolutionData can be loaded into CLASS from a formatted ASCII file see section 7.3.1 as follow :

```
CLASSLogger *Logger = new CLASSLogger("CLASS_OUTPUT.log",1,2);

EvolutionData*  MyEvolutionData = new EvolutionData(Logger, "/PATH/Data.dat");
```

7.3.1 EvolutionData ASCII format

The formatted ASCII file describing the EvolutionData is formatted as follow:

Listing 7.1: Evolution Data format

```
time "0 t2 t3 ..."                  // in seconds
keff "k1 k2 k3 ..."                       // not mandatory entry
flux "phi1 phi2 phi3 ..."           //(neutron/(second.cm2))not mandatory entry
Inv "Z A I inv1 inv2 inv 3 ..." //in atoms
...
XSFis "Z A I xsfis1 xsfis2 xsfis3 ..."//in barns
...
XSCap "Z A I xscap1 xscap2 xscap3 ..."
...
XSn2n "Z A I xsn2n1 xsnsn2 xsn2n3 ..."
...
```

The meaning of each keyword is listed in table 7.1.

Table 7.1: .dat Key words meaning

Key words	Meaning
Inv	Inventory
XSFis	fission cross section
XSCap	(n,γ) cross section
XSn2n	$(n,2n)$ cross section
Value	**meaning**
Z	Charge number
A	Mass number
I	State (fundamental=0, 1^{st} excited =1, ...)

Each EvolutionName.dat files comes with a EvolutionName.info file, which describes the reactor, it is formatted like this :

```
Reactor "ReactorName"           //What ever string without space
Fueltype "FuelName"                     //What ever string without space
CycleTime "t"                                   //The final time simulated
    (years)
ConstantPower "P"                               //Simulated power (in W)
```

7.3.2 DecayDataBank

The radioactive decay is handled by a DecayDataBank. The DecayDataBank contains an EvolutionData for each nucleus of the nuclei chart. Each EvolutionData describes the evolution of the nucleus and all its daughters as a function of the time. The depletion of an isotopic vector corresponds to the sum of all its nucleus depletion contribution.

In other words, in CLASS, for each nucleus of the chart, a depletion calculation has been performed and compiled in a DecayDataBank.
The determination of an IsotopicVector depletion is performed as follow :
First, one determines the depletion of each nucleus of the IsotopicVector following the DecayDataBank, then sums all those contributions.

DecayDataBank can be defined as follow :

```
CLASSLogger *Logger = new CLASSLogger("CLASS_OUTPUT.log",1,2);

DecayDataBank* DecayDB = new DecayDataBank(Logger, "/PATH/Decay.idx");
```

In the previous example a DecayDataBank has been defined using the file Decay.idx file. This file lists all the path to EvolutionDatas (each one corresponding to the depletion of one nucleus). The format of the .idx file is the following :

```
Z1 A1 I1 PATH/ZAI1.dat
...
Zn An In PATH/ZAIn.dat
```

A DecayDataBank can be find in $CLASS_PATH/DATA_BASES/DECAY/ALL/

7.4 Log management : CLASSLogger

In CLASS, all messages are handled by the CLASSLogger object. There are 4 verbose levels, see table 7.2.

Table 7.2: Verbose levels

level #	meaning	informations
0	ERROR	This is the default. *It makes the code to stop*
1	WARNING	LVL 0 + something may go wrong but the code continue running
2	INFO	LVL 1 + simple informations about ongoing process
3	DEBUG	LVL 2 + each method begin and end

There are two outputs for these messages : the standard output (terminal) and a logfile. For each output a verbose level can be assigned as follow :

```
CLASSLogger *Logger = new CLASSLogger("CLASS_OUTPUT.log",1,2);
```

In the preceding example, verbose level 1 (WARNING) has been set for the terminal output and level 2 (INFO) for the second output which is the logfile named CLASS_OUTPUT.log.

Chapter 8
Scenario

The Scenario object aims to regroup all facilities.

8.1 Fill the scenario

In order to evolve during a dynamic fuel cycle calculation, each facility need to be added in the scenario. To do so, five "adding methods" have been implemented :

```
CLASSLogger *Logger = new CLASSLogger("CLASS_OUTPUT.log",1,2);
Scenario *gCLASS=new Scenario(Logger, 1977*year);
//1977*year = starting time of the scenario
gCLASS->AddPool(myPool);
gCLASS->AddReactor(myReactor);
gCLASS->AddStorage(myStorage);
gCLASS->AddFabricationPlant(myFabricationplant);
gCLASS->AddSeparationPlant(mySeparationplant);
//or
gCLASS->Add(myPool);
gCLASS->Add(myReactor);
gCLASS->Add(myStorage);
gCLASS->Add(myFabricationplant);
gCLASS->Add(mySeparationplant);
```

Furthermore, one need to add a DecayDataBase to the Scenario, using :

```
DecayDataBank* DecayDB = new DecayDataBank(Logger, "/PATH/Decay.idx");

gCLASS->SetDecayDataBase(DecayDB);
```

8.2 OutPut

8.2.1 General Output

In addition to all facilities added to the Scenario, the output contain also other general information, see table 8.1.

Table 8.1: General Information in CLASS Output

Output Name	Unit	description
AbsoluteTime	Number [Second]	Time at the step
ParcPower	Number [Watt]	Effective thermal power of the Scenario *only working reactor are taked into account*
WASTE	IsotopicVector	Waste produced by the scenario
STOCK	IsotopicVector	All the material in all the Storage
OUTINCOME	IsotopicVector	All material taken from outside the Scenario
COOLING	IsotopicVector	All the material present in all the Pool
FUELFABRICATION	IsotopicVector	All the material present in all the FabricationPlant
REACTOR	IsotopicVector	All the material present in all the Reactor
INCYLE	IsotopicVector	All the material in the cycle *Reactor + Pool + Fabrication + Storage*
TOTAL	IsotopicVector	All the material in the Scenario *Reactor + Pool + Fabrication + Storage + Waste*

8.2.2 Output names

The CLASS output is saved in a ROOT format, each element of the Scenario is added to a ROOT TTree, filled at each time step. By default the output file name is "CLASS_Default.root" and the ROOT TTree name is "Data". It is possible to change those names using :

```
gCLASS->SetOutputFileName("MyFileName.root");
gCLASS->SetOutputTreeName("MyTTreeName");
```

8.2.3 Output Frequency

By default, a snapshot of the scenario is done every years. To change this frequency use :

```
gCLASS->SetTimeStep(365.25*24*3600/12); // monthly output
```

8.2.4 Reading a CLASS ouput

There is an easy way and an hard way to read CLASS outputs. The easy one is via the Graphical User Interface CLASSGui (see section V). This is an user friendly approach but as it is graphical it comes with its limitations. The hard way allows to access anything in your scenario evolution. This last approach requieres to write a specific .cxx file in which you can access any information you want by using the objects and methods of CLASS libraries. A detailed example of this approach can be found in $CLASS_PATH/example/ReadOuput.cxx. You are encouraged to read carefully the comments in this file to learn how to access the data you want.

Part IV

Physics Models

Chapter 9

Description and implementation

A PhysicsModels is related to one or several reactors, it is a container of three models :

- Equivalence Model : Tells to the Fabrication Plant how to build the fuel.
- XS Model : "Calculates" the mean cross sections of this fuel and sends it to the Bateman Solver.
- Irradiation Model : It is the Bateman Solver. User can choose between different numerical methods.

A PhysicsModels is called in the CLASS input like the following example :

Implementation in a .cxx :

Listing 9.1: PhysicsModels

```
...
#include "XS/XSM_MLP.hxx"
#include "Irradiation/IM_RK4.hxx"
#include "Equivalence/EQM_PWR_MLP_MOX.hxx"
int main()
{
        ....

        EQM_PWR_MLP_MOX* Equivalence    = new EQM_PWR_MLP_MOX( "
            PathToTMVAWeightFile/TMVAWeightFile.xml" );
        XSM_MLP* XS = new XSM_MLP( gCLASS->GetLog(),"PathToTMVAWeighstFolder" ,
            OneMLPPerTimeStep );
        IM_RK4* Solver = new IM_RK4( gCLASS->GetLog() );
        PhysicsModels* PHYMOD = new PhysicsModels( XS , Equivalence , Solver );

        ...
        Reactor *PWR_MOX = new Reactor(log, PHYMOD, fabricationplant, Pool,
            creationtime, lifetime, cycletime, HMMass, BurnUp);
        ...
}
```

In this latter example a PhysicsModels called "PHYMOD" is defined, it contains the bateman solver "Solver" which is the Runge Kutta (4^{th} order) method. The mean cross sections predictor, "XS", used is based on a Multi Layer Perceptron. The Equivalence Model "Equivalence" is the one used for PWR MOX fuels. The arguments of the 3 objects constructor are explained in their corresponding sections.

All the existing models are defined in the following sections, furthermore, the way to build a new model is presented.

Chapter 10

Equivalence Model

The aim of an equivalence model is to predict the content of fissile element needed in a fuel to reach a given burnup or to satisfied criticality conditions.

10.1 Available Equivalence Models

The CLASS package contains, at the moment, 9 different equivalence models where three are related to the building of fuels for a PWR-MOX , one to the building of PWR-UOX fuels, one for the FBR-Na MOX, two dedicated to fast breeder and one suitable for all non-breeder reactors, an other model allows to handle (Pu,Am,U)O_2 fuel loaded in PWR :

10.1.1 PWR-MOX models :

The following models returns the molar fraction $\%_{Pu}$ of plutonium needed to reach a given burnup according to the plutonium isotopic composition available in stocks.

10.1.1.1 Linear BU model : EQM_PWR_LIN_MOX

It was initially applied for MOX fuel, but because of the lack of precision, this model could be deprecated (at least in the PWR MOX case). It remain in the CLASS packages only because it was present historically.
Nevertheless it could be use as an example for similar model for other fuel. This model suppose it is possible to describe the maximal burnup accessible for a set fuel using its initial composition using a simple linear modelisation (equation 10.1):

$$BU_{max} = \alpha_0 + \sum_i^N \alpha_i \cdot n_i, \tag{10.1}$$

where BU_{max} represent the maximal accessible burnup for the fuel, n_i the isotopic fraction of the isotope i, N the number of isotope present in the fuel, and the α_i the parameter of the model. The main difficulty concerning this model, is the determination of the α_i: to be correct the α_i should be fitted on a set of evolution data which are not constrain to reach an unique burnup, but a large burnup region. One can see the problem guessing it is possible to build a set a fuel evolution reaching exactly a unique burnup (45 GWd/t by example), the χ^2 minimization of the α_i will end up with $\alpha_0 = 45$ and all the other at zero. That why, when using a linear burnup description model, one should test the validity of the model, on many random compositions by example...

10.1.1.2 Quadratic Model : EQM_PWR_QUAD_MOX

The $\%_{Pu}$ is calculated according a quadratic model. See equation 10.2.

$$\%_{Pu} = \alpha_0 + \sum_{i \in Pu}^{N} \left(\alpha_i \cdot n_i \ + \sum_{j \leq i} \alpha_{ij} \cdot n_i \cdot n_j \right), \tag{10.2}$$

where n_i is the molar proportion (in $\%mol.$) of isotope i [1] in the fresh plutonium vector. α_{ij}, α_i and α_0 are the weights resulting from a minimization procedure and are related to one targeted burnup

[1] from ^{238}Pu to ^{242}Pu

and one fuel management. Furthermore, ^{241}Am from ^{241}Pu decay is not one of the considered component of the model (n_i), instead the model considers a fixed time since plutonium separation.

The file containing the weights is formatted as follow :

```
PARAM "238Pu 238Pu*238Pu 238Pu*239Pu 238Pu*240Pu 238Pu*241Pu 238Pu*242Pu 239Pu 239
    Pu*239Pu 239Pu*240Pu 239Pu*241Pu 239Pu*242Pu 240Pu 240Pu*240Pu 240Pu*241Pu 240
    Pu*242Pu 241Pu 241Pu*241Pu 241Pu*242Pu 242Pu 242Pu*242Pu 1"
```

Where 238Pu stands for $\alpha_{^{238}Pu}$ and it is the first order weight related to the molar proportion of ^{238}Pu and 1 is α_0. The weights are in units of $\%mol. \cdot \%mol.^{-1}$ for α_i in units of $\%mol. \cdot \%mol.^{-2}$ for α_{ij} and in units of $\%mol.$ for α_0. The Keyword "PARAM" has to be present in the file before the α values. For more informations about this model and the generation of the coefficients please refer to reference [Moug 15].

Implementation in a .cxx

Listing 10.1: Equivalence Model EQM_QUAD_MOX

```
...
#include "Equivalence/EQM_PWR_QUAD_MOX.hxx"
...
int main()
{
...
EQM_PWR_QUAD_MOX* Equivalence = new     EQM_PWR_QUAD_MOX( LogObject, AlphasFile );
// or
// EQM_PWR_QUAD_MOX* Equivalence = new  EQM_PWR_QUAD_MOX( AlphasFile );
...
}
```

With LogObject a CLASSLogger object (see section 7.4) and AlphasFile a string which is the complete path to the file containing the weights (the α parameters)

10.1.1.3 Neural network model : EQM_PWR_MLP_MOX

This equivalence model is based on a Multi Layer Perceptron (MLP) and predict the amount of plutonium needed to reach **any burnup**. The MLP inputs are the isotopic compositions of the plutonium (**including** ^{241}Am), the enrichment of depleted uranium, and the targeted burnup. The output is the plutonium content needed to reach the burnup. This method uses the neural networks of the root module TMVA [Hoec 07]. To executes this model, TMVA is run in CLASS and need a .xml file. This file contains the neural network architecture and the weights resulting from the training procedure.

Implementation in a .cxx :

Listing 10.2: Equivalence Model EQM_MLP_PWR_MOX

```
...
#include "Equivalence/EQM_PWR_MLP_MOX.hxx"
...
int main()
{
...
EQM_PWR_MLP_MOX* Equivalence = new      EQM_PWR_MLP_MOX( LogObject, TMVAWeightPath
    );
// or
// EQM_PWR_MLP_MOX.* Equivalence = new  EQM_PWR_MLP_MOX( TMVAWeightPath );
...
```

With LogObject a CLASSLogger object (see section 7.4) and TMVAWeightPath a string containing the path to the .xml file.

In order to make his own .xml file one need to have a training data containing the fresh fuel composition and the achievable burnup of many examples. The fuel composition is characterized by the mean of :

- The plutonium composition (*i.e :* %mol. of ^{238}Pu, ^{239}Pu, ^{240}Pu, ^{241}Pu, ^{242}Pu, and ^{241}Am)
- The plutonium content (*i.e :* $\frac{Pu}{Pu+U}$)
- The ^{235}U content in the depleted uranium.

The file $CLASS_PATH/DATA_BASES/PWR/MOX/EQModel/EQM_MLP_PWR_MOX_3batch.xml has been generated from the file $CLASS_PATH/Utils/EQM/PWR_MOX_MLP/Train_MLP.cxx
To train a new MLP from your own training sample proceed as follow :

```
cd $CLASS_PATH/Utils/EQM/PWR_MOX_MLP
g++ -o Train_MLP `root-config --cflags` Train_MLP.cxx `root-config --glibs` -lTMVA -
    I$ROOTSYS/tmva/test/
Train_MLP YourTrainingData.root
```

Where YourTrainingData.root is a root file containing a TTree filled with fuel compositions and corresponding burnups. The .xml file will be generated in a folder named weight. The results of the testing procedure of the MLP are in a file named TMVA_MOX_Equivalence.root but will be presented to you graphically as soon as the training and the testing procedure are finished.

To make your YourTrainingData.root file you have to fill a TTree with your data. To do so, create a .cxx file and copy past this :

```
        TFile*   fOutFile = new TFile("YourTrainingData.root","RECREATE"); //
            create the .root file
        TTree*   fOutT = new TTree("Data", "Data");//create the TTree

//---------------------INITIALISATIONN--------------------
//WARNING : keep the same variable names :
        double U5_enrichment  = 0;
        double Pu8= 0;
        double Pu9 = 0;
        double Pu10= 0;
        double Pu11= 0;
        double Pu12 = 0;
        double Am1= 0;
        double BU= 0; //BU means burnup
        double teneur              = 0; //French for content (here Pu content)
//---------------------BRANCHING--------------------------
        fOutT->Branch(  "U5_enrichment", &U5_enrichment, "U5_enrichment/D"      );
        fOutT->Branch(  "Pu8", &Pu8, "Pu8/D");
        fOutT->Branch(  "Pu9", &Pu9, "Pu9/D");
        fOutT->Branch(  "Pu10", &Pu10, "Pu10/D");
        fOutT->Branch(  "Pu11", &Pu11, "Pu11/D" );
        fOutT->Branch(  "Pu12", &Pu12, "Pu12/D");
        fOutT->Branch(  "Am1", &Am1     , "Am1/D");
        fOutT->Branch(  "BU", &BU       ,"BU/D");
        fOutT->Branch(  "teneur"        ,&teneur,"teneur/D");
//---------------------FILLING--------------------------
        int Nex=NumberOfDifferentExample;
        for(int ex=0;ex<Nex;ex++)
        { //--------Fresh Fuel Composition------------
                U5_enrichment   =  fU5_enrichment[ex];
                Pu8=fPu8[ex];
                Pu9= fPu9[ex];
                Pu10=fPu10[ex];
                Pu11=fPu11[ex];
                Pu12=fPu12[ex];
                Am1=fAm1[ex];
                teneur=fteneur[ex];
                //-------Corresponding maximal burnup-------
                BU =  BurnUps[ex];
                //---Fill the tree with this fuel composition and this burnup
                    ------
                fOutT->Fill();
        }
        fOutFile->Write();
        delete fOutT;
        fOutFile-> Close();
        delete fOutFile;
}
```

Then, build the arrays fU5_enrichment, fPu8 ... with your data, compile and execute. For more informations about this model please refer to [Leni 15]

Available weight file (.xml) :

- **$CLASS_PATH/DATA_BASES/PWR/MOX/EQModel/EQM_MLP_PWR_MOX_3ba**
  : Generated with 5000 MURE evolutions with different fuel composition, using a full mirrored assembly calculation with JEFF3.1.1 cross section and fission yield data bases. Valid for mono-recycling of plutonium and a fuel management of 3 batches. More details about the generation of this .xml file can be found in reference [Leni 15].

10.1.2 PWR-Am model

This model is based on the same philosophy of the *EQM_PWR_MLP_MOX* model. The only difference is in the number of inputs of the MLP (additional isotopes : Americium 241 , isomeric 242 , 243). The MLP weights given with the package are for a third batch reloading pattern. The

weight are suitable to work with plutonium and americium coming from the reprocessing of PWR-UOX fuels.

10.1.3 PWR-UOX model :

10.1.3.1 Linear Model: EQM_LIN_UOX

Predict the quantity of ^{235}U needed to reach the wanted burn-up :

$$N_{^{235}U} = A * BurnUp^2 + B * BurnUp + C \tag{10.3}$$

See in **$CLASS_PATH/DATA_BASES/PWR/UOX** for available model.

10.1.4 FBR-Na-MOX model :

This model is used to compute the plutonium content needed for a fast reactor loaded with MOX fuel.

10.1.4.1 Baker & Ross Model: EQM_FBR_BakerRoss_MOX

It calculates the plutonium content (E) needed for the FBR Na loaded with a given Pu vector according to :

$$E = \frac{E_{ref} - \sum_{fertile} N_i W_i}{\sum_{fissile} N_i W_i - \sum_{fertile} N_i W_i} \tag{10.4}$$

with :

$$W_i = \frac{\alpha_i - \alpha_{^{238}U}}{\alpha_{^{239}Pu} - \alpha_{^{238}U}} \tag{10.5}$$

and

$$\alpha_i = \bar{\nu_i} \cdot \sigma_i^{fis} - \sigma_i^{cap} \tag{10.6}$$

With E_{ref} the plutonium content needed for a FBR Na to satisfy criticality condition at begining of cycle ($k_{eff}(t=0) = 1.00$) with a reference fresh fuel composition. The reference plutonium composition is 100% ^{239}Pu and uranium is 100% ^{238}U. $\bar{\nu_i}$ is the average number of total neutron emitted per fission, σ_i^{fis} is the mean fission cross section of nucleus i and σ_i^{cap} is the mean capture cross section of nucleus i. The default values of the weight W_i given in the constructor have been calculated from a MURE/MCNP run of an ESFR lire reactor loaded with a fresh fuel composition given in table and allowing to access $k_{eff}(t=0) = 1.00$. To implement this model in your CLASS input proceed as follow :

Implementation in a .cxx

Listing 10.3: Equivalence Model EQM_BakerRoss_FBR_MOX

```
...
#include "Equivalence/EQM_FBR_BakerRoss_MOX.hxx"
...
int main()
{
...
EQM_FBR_BakerRoss_MOX* Equivalence = new        EQM_FBR_BakerRoss_MOX( ); //the
    default weight and Eref are used
// or
 EQM_FBR_BakerRoss_MOX* Equivalence = new EQM_FBR_BakerRoss_MOX( Weight_U_235,
     Weight_Pu_238,  Weight_Pu_240, Weight_Pu_241, Weight_Pu_242, Weight_Am_241,
     Eref);
;
...
}
```

10.1.5 General non breeder model

This model called EQM_MLP_kinf can be applied for any non breeder reactors and for fuel constituted with a fertile and a fissile part. It determines the fissile content needed to reach an user defined maximal burnup (BU_{target}) according a user defined number of batches N (for the loading pattern) and a threshold on the multiplication factor ($k_{threshold}$). The fissile content is varied until the maximal burnup (BU_{max}) is equal to BU_{target}. The maximal burnup BU_{max} for a given fresh fuel composition, a given number of batch and a given $k_{threshold}$ is such as :

$$< k_\infty >^{batch} (BU_{max}) = \frac{1}{N} \sum_{i=1}^{i=N-1} k_\infty (i * BU_{max}/N) = k_{threshold}$$

The k_∞ is predicted with a multi layer percetron. To implement this model in your CLASS input proceed as follow :

Implementation in a .cxx

Listing 10.4: Equivalence Model EQM_BakerRoss_FBR_MOX

```
...
#include "Equivalence/EQM_MLP_Kinf.hxx"
...
int main()
{
...

 EQM_MLP_Kinf* Equivalence = new EQM_MLP_Kinf( TMVAWeightPath, NumOfBatch,
     InformationFile , CriticalityThreshold
...
}
```

Where TMVAWeightPath is the path to the weight file of the MLP (.xml file) , NumOfBatch is the number of batches for the loading pattern and CriticalityThreshold is the $k_{threshold}$. InformationFile contains information regarding the MLP inputs and are listed above (the quotes have to be removed):

Listing 10.5: Information file format

```
Specific Power (W/gHM) :
k_specpower "specificpower" // the power density in Watt per gram of heavy metal

Maximal burnup (GWd/tHM) : // for the algorithm initialization : a relatively high
    Burnup value
k_maxburnup "BUmax"                // e.g 100 for a PWR

Maximal fissile content (molar proportion) : // for the algorithm initialization :
    a relatively high fissile content
k_maxfiscontent "maxFisContent"  // e.g 0.25 for a PWR MOX

Z A I Name (input MLP) : // name for the MLP inputs
k_zainame "Z1 A1 I1 Name1"
...
k_zainame "Z2 A2 I2 Name2"
...
Fissile Liste (Z A I) : // the fissile list to be taken in the stocks for fuel
    manufacturing
k_fissil "Z1 A1 I1"
..
k_fissil "Z2 A2 I2"

Fertile Liste (Z A I Default Proportion) :// the fertile list to be taken in the
    stocks for fuel manufacturing
k_fertil "Z1 A1 I1 prop"
..
k_fertil "Z2 A2 I2 prop2"
```

A weight file (.xml) and .nfo file can be found in :
$CLASS_PATH/DATA_BASES/PWR/MOX/EQModel/MLP_Kinf/MLP

In order to make his own .xml file one need to have a training data containing the fresh fuel composition and the k evolution over time.
First, you have to convert your depletion calculations in .dat format (see section 7.3.1 for the fomat definition). If you use MURE depletion code, you can find the convertor software in :
$CLASS_PATH/Utils/MURE2CLASS/.
Once the set of .dat files is generated you can convert this files in one .root file to be red by the training algorithm of the MLP. The software is located in :
$CLASS_PATH/Utils/EQM/MLP_Kinf/GenerateRootFile.cxx. Go to this folder and edit this .cxx file and look for @@@change to make the apropriate changes. Then compile and execute.
Then, you are good to go to the training/testing process : Edit TrainMLP.cxx and make the appropriate changes (looking for @@@changes).Then compile and execute. You should find your .xml file in :
$CLASS_PATH/Utils/EQM/MLP_Kinf/weight/ . You can also check the testing results
Create a folder in $CLASS_PATH/DATA_BASES with the name you want. Move the .xml and .nfo in this location , make sure these two files have the same name (except their extension of course), and voila.

10.1.6 General breeder models

10.1.6.1 $k_{eff}(t=0)$ prediction using MLP

This model aims to predict the fissile content satisfying $k_{eff}(t=t_{user})=k_{user}$. A MLP is used to predict the k_{eff} for a given irradiation time. Then the fissile content is adjusted until $k_{eff}=k_{user}$. A MLP weight file is given in $DATA_BASES/FBR_Na/MOX/EQModel/MLP_K_EFF_BOC and is tuned to predict the k_{eff} of an ESFR like reactor loaded with MOX fuel at BOC ($t_{user}=0$). To change the t_{user} you have to train your neural network to predict k_{eff} at this irradiation time. In order to make his own .xml file one need to have a training data containing the fresh fuel composition and the k at the t_{user}.

First, you have to convert your depletion calculations in .dat format (see section 7.3.1 for the fomat definition). If you use MURE depletion code, you can find the convertor software in : $CLASS_PATH/Utils/MURE2CLASS/.
Once the set of .dat files is generated you can convert this files in one .root file to be red by the training algorithm of the MLP. The software is located in : $CLASS_PATH/Utils/EQM/FBR_MLP_Keff/GenerateRootFile.cxx. Go to this folder and edit this .cxx file and look for @@@change to make the apropriate changes. Then compile and execute. Then, you are good to go to the training/testing process : Edit TrainMLP.cxx and make the appropriate changes (looking for @@@changes).Then compile and execute. You should find your .xml file in : $CLASS_PATH/Utils/EQM/FBR_MLP_Keff/weight/ . You can also check the testing results Create a folder in $CLASS_PATH/DATA_BASES with the name you want. Move the .xml and .nfo in this location , make sure these two files have the same name (except their extension of course), and voila.

For this model to work a nfo file is also required the format is given above :

Listing 10.6: Information file format

```
Specific Power (W/gHM) :
k_specpower "specificpower" // the power density in Watt per gram of heavy metal

Maximal fissile content (molar proportion) : // for the algorithm initialization :
     a relatively high fissile content
k_maxfiscontent "maxFisContent"  // e.g 0.4 for a FBR MOX

Z A I Name (input MLP) : // name for the MLP inputs
k_zainame "Z1 A1 I1 Name1"
...
k_zainame "Z2 A2 I2 Name2"
...
Fissile Liste (Z A I) : // the fissile list to be taken in the stocks for fuel
    manufacturing
k_fissil "Z1 A1 I1"
..
k_fissil "Z2 A2 I2"

Fertile Liste (Z A I Default Proportion) :// the fertile list to be taken in the
    stocks for fuel manufacturing
k_fertil "Z1 A1 I1 prop"
..
k_fertil "Z2 A2 I2 prop2"
```

To implement this model in your CLASS input proceed as follow :

Implementation in a .cxx

Listing 10.7: Equivalence Model EQM_BakerRoss_FBR_MOX

```
...
#include "Equivalence/EQM_FBR_MLP_Keff.hxx"
...
int main()
{
...

EQM_FBR_MLP_Keff* Equivalence = EQM_FBR_MLP_Keff( TMVAWeightPath,  keff_user )
//TMVAWeightPath is the path to the .xml file
..
}
```

10.1.6.2 Upper and lower limits on $< k_\infty >^{batch}$

In order to take into account the impact of the loading pattern this model used the $< k_\infty >^{batch}$ function defined in section 10.1.5. The fissile content is such as this value is contained in a user defined range. We suggest to use $1/P_{noleak}$ as the lower bound and $1/P_{noleak} + |\rho_{controlRods}|$ as upper bound. With P_{noleak} is the no leak probability (~ 0.88 for ESFR like reactor) and $\rho_{controlRods}$ the anti-reactivity of the control rods (estimated for a ESFR like to be 2000 pcm for all control rods put at half of the core high). Not than in rare occasion no solution are available for a given fissile and fertile composition. The $k_\infty(t)$ is determined using a MLP. In order to make his own .xml follow the same procedure as the one explained in section 10.1.5.

To implement this model in your CLASS input proceed as follow :

Implementation in a .cxx

Listing 10.8: Equivalence Model EQM_BakerRoss_FBR_MOX

```
...
#include "Equivalence/EQM_FBR_MLP_Kinf_BOUND.hxx"
...
int main()
{
...

EQM_FBR_MLP_Kinf_BOUND* Equivalence = EQM_FBR_MLP_Kinf_BOUND( TMVAWeightPath,
    NumOfBatch , LowerK, UpperK)
//TMVAWeightPath is the path to the .xml file
// NumOfBatch is the number of batches for the fuel loading pattern
// LowerK is the lower bound on <k_{\infty}>^{batch}$
// UpperK is the upper bound on <k_{\infty}>^{batch}$

..
}
```

For this model to work a nfo file is also required the format is given above :

Listing 10.9: Information file format

```
Specific Power (W/gHM) :
k_specpower "theSpecificPower" // i.e the power density in watt per gram of heavy
    metal

Time (s) :// the time used to train the MLP
K_TIMESTEP "0 t1 t2 ..."


Z A I Name (input MLP) : // name for the MLP inputs
k_zainame "Z1 A1 I1 Name1"
...
k_zainame "Z2 A2 I2 Name2"
...
Fissile Liste (Z A I) : // the fissile list to be taken in the stocks for fuel
    manufacturing
k_fissil "Z1 A1 I1"
..
k_fissil "Z2 A2 I2"

Fertile Liste (Z A I Default Proportion) :// the fertile list to be taken in the
    stocks for fuel manufacturing
k_fertil "Z1 A1 I1 prop"
..
k_fertil "Z2 A2 I2 prop2"
```

A weight file (.xml) and .nfo file can be found in
$CLASS_PATH/DATA_BASES/FBR_Na/MOX/EQModel/MLP_K_INF_BOUND

10.2 How to build an Equivalence Model

The strength of CLASS is to allow the user to build his own Physics models, this section explains how to build a new equivalence model and to incorporate it into CLASS.

First you have to create the file EQM_NAME.cxx and EQM_NAME.hxx, where NAME is a name you choose. Then open with a text editor the .hxx and copy past the following replacing NAME by the name you want.

Listing 10.10: EQM_NAME.hxx

```
#ifndef _EQM_NAME_HXX
#define _EQM_NAME_HXX
#include "EquivalenceModel.hxx"
using namespace std;
/*! Define a EQM_NAME
  Explain briefly what is it.
  @author YourName
  @version 3.0
*/

class EQM_NAME : public EquivalenceModel
{
        public :
        /*Constructor*/
        EQM_NAME(/*parameters*/ ); //!< Explain what is the parameters (if any)

        /**This function IS the equivalence model **/
        double GetFissileMolarFraction(IsotopicVector Fissil,IsotopicVector Fertil
            ,double BurnUp); //!<Return the molar fraction of fissile element

        private :
        /*Your private variables*/
};
#endif
```

Open the .cxx file and copy past the following in it (replacing NAME by the same name you used in the .hxx).

Listing 10.11: EQM_NAME.cxx

```
#include "EquivalenceModel.hxx"
#include "EQM_NAME.hxx"
#include "CLASSLogger.hxx"
/*Whatever include you need*/
//-------------------------------------------------------------------------
//               EQM_NAME
//
//       Brief description
//-------------------------------------------------------------------------
//Constructor(s)
EQM_NAME::EQM_NAME(/*parameters*/)
{
//.... Do whatever you want with your parameters
/*
        Fill the two isotopic vectors fFissileList and fFertileList
        see explanation in the manual
*/
        //Fertile
        ZAI U8(92,238,0);
        ZAI U5(92,235,0);
        double U5_enrich= 0.0025;
        fFertileList = U5*U5_enrich + U8*(1-U5_enrich);

        //Fissile
        ZAI Pu8(94,238,0);
        ZAI Pu9(94,239,0);
        //...
        fFissileList = Pu8*1+Pu9*1+ /*...*/;
}
//-------------------------------------------------------------------------
double EQM_NAME::GetFissileMolarFraction(IsotopicVector Fissil,IsotopicVector
    Fertil,double BurnUp)
{
//Code your Equivalence Model : This function has to return the molar fraction  of
     fissile in the fuel needed to reach the BurnUp(GWd/tHM) according to the
    composition of the Fissil and Fertil vectors
}
```

In the constructor (EQM_NAME::EQM_NAME) you have to fill two isotopic vectors named **fFissileList** and **fFertileList**. Don't declare these isotopic vector in the .hxx, there are already declared in the file src/EquivalenceModel.hxx. fFissileList is used by the FabricationPlant to do the chemical separation of the fissile element from the other present in stock. For instance, for the plutonium, add the ZAI ^{238}Pu, ^{239}Pu, ^{240}Pu, ^{241}Pu and ^{242}Pu. fFertile List is used by the FabricationPlant the same way fFissileList is used but you have to define a default IsotopicVector to be used if you didn't provide a fertile stock to your FabricationPlant. In the example given above the fertile is depleted uranium and the proportion of each isotope is given (^{234}U is unheeded). Now you have to build the function **GetFissileMolarFraction(IsotopicVector Fissil, IsotopicVector Fertil, double BurnUp)**. Its parameters are provided by the FabricationPlant and are :

- IsotopicVector Fissile : it is the proportion of each nucleus you give in the fFissileList plus the proportion of the nuclei that appears during the fabrication time (time given in the FabricationPlant constructor, is default is 2 years)
- IsotopicVector Fertile : it is the proportion of each nucleus you give in the fFertileList plus the proportion of the nuclei that appears during the fabrication time. If you didn't provide any fertile stock to your FabricationPlant then it's the default vector given in the EQM_NAME constructor.
- double BurnUp : The maximal average burnup for your fuel to reach (in GWd/tHM).

Fill free to have a look at the models present in $CLASS_PATH/source/Model/Equivalence to get inspiration.

Now that your equivalence model is ready two choices are offered to you. You can compile the two files of your model with your CLASS input or you can add this model to the CLASS package. The second option will modify the CLASS software and we will be no longer able to troubleshoot your scenario. So use the second option only if you are a completely independent user !

10.2.1 Compile your equivalence model with your CLASS executable :

```
g++ -g -O -I $CLASS_include -L $CLASS_lib -lCLASSpkg `root-config --cflags`
        `root-config --libs` -fopenmp -lgomp -Wunused-result -c My_MODEL.cxx

\rm CLASS* ; g++ -o CLASS_exec MyScenario.cxx My_MODEL.o -I $CLASS_include -L $CLASS_lib -
    lCLASSpkg `root-config --cflags` `root-config --libs` -fopenmp -lgomp -Wunused-result
```

10.2.2 Your equivalence model in the CLASS library :

Move your EQM_NAME.hxx and EQM_NAME.cxx in $CLASS_PATH/source/Model/Equivalence/. Then open with your favourite text editor the file $CLASS_PATH/source/src/Makefile, find "OBJMODEL" and add $(EQM)/EQM_NAME.o within the others $(EQM) objects. Then re-compile CLASS, fix the compilation errors ;) and voilà your equivalence model is now available in the CLASS library.

Chapter 11
XS Model

The aim of a mean cross section model (XSModel) is to predict the mean cross sections of a fuel built by an EquivalenceModel (EQM) (see section 10). The mean cross sections are required to compute fuel depletion in a reactor.

11.1 Available XS Models

There is, for the moment, 2 XSModel in CLASS :

11.1.1 Pre-calculated XS : XSM_CLOSEST

This method looks, in a data base, for a fresh fuel with a composition **close** to the brandy new fuel built by the EquivalenceModel. Here, close means that the fresh fuel in the data base minimizes the distance d (see equation 11.1).

$$d = \sqrt{\sum_i w_i \cdot (n_i^{DB} - n_i^{new})^2}, \tag{11.1}$$

where n_i^{DB} is the number of nuclei i in one element of the data base and n_i^{new} the number of nuclei i in the new fuel built by the EQM. w_i is a weight associated to each isotopes, its value is 1 by default. When the closest evolution in the database is found, the corresponding mean cross sections are extracted and used for the calculation of the depletion of the new fuel.

Implementation in a .cxx :

Listing 11.1: Cross section Model XSM_CLOSEST

```
...
#include "XS/XSM_CLOSEST.hxx"
...
int main()
{
        XSM_CLOSEST* XSMOX = new XSM_CLOSEST( gCLASS->GetLog(), PathToIdxFile );
        //or
        //XSM_CLOSEST* XSMOX = new XSM_CLOSEST( PathToIdxFile );
}
```

With LogObject a CLASSLogger object (see section 7.4) and PathToIdxFile a string containing the path to the .idx file. The .idx file lists all the EvolutionData (see section 7.3) of the data base. This file is formatted as follow :

```
TYPE "NameOfTheFuel(withoutspace)"
"PATH_TO_DATA_BASE/EvolutionName.dat"
"PATH_TO_DATA_BASE/OtherEvolutionName.dat"
....
```

Each EvolutionName.dat file contains a formatted fuel depletion calculation. the format of a EvolutionData ASCII file is detailed in section 7.3.1. The number of .dat files has an influence on the model accuracy. Furthermore, the initial composition of the different fuel depletion calculations has to be representative of the fresh fuel compositions encounter in a scenario.

For MURE user only : The program $CLASS_PATH/Utils/MURE2CLASS converts a list of MURE evolutions to a list of .dat and .info files and creates the .idx file, type in terminal the following command for more details.

```
\$CLASS\_PATH/Utils/MURE2CLASS -h
```

Users of others fuel depletion code (*e.g* VESTA, ORIGEN, MONTEBURNS, SERPENT ....) have to create their own program to generate these files.

11.1.2 XS predictor : XSM_MLP

This method calculates the mean cross sections by the mean of a set of neural networks (MLP from TMVA module) . There is two configurations available :

- One MLP per nuclear reaction and per time step (this one is deprecated and not describe in this manual) .
- One MLP per nuclear reaction. the irradiation time is one of the MLP inputs.

Implementation in a .cxx :

Listing 11.2: Cross section Model XSM_MLP

```
...
#include "XS/XSM_MLP.hxx"
...
int main()
{       ...
        XSM_MLP* XSMOX = new XSM_MLP( ClassLog, PathToWeightFolder, InfoFileName,
            OneMLPPerTime );
//or
//XSM_MLP* XSMOX = new XSM_MLP(PathToWeightFolder, InfoFileName, OneMLPPerTime);
...
}
```

PathToWeightFolder (string) is the path to the folder containing the weight files (.xml files). **OneMLPPerTime** is a boolean set to true if there is one MLP per reaction and per time step. **InfoFileName** (string) is the name of the file located in PathToWeightFolder which is informing on the reactor and on the inputs of the XS_MLP model. Its default name is Data_Base_Info.nfo . The InfoFileName contains keywords beginnings with k_ (note that it is not case sensitive) and corresponding value(s). Any comments can be added. The quotes must be removed.

Listing 11.3: Information file format

```
ReactorType :
K_REACTOR "ReactorName" //without space
FuelType :
K_FUEL "FuelName" //without space
Heavy Metal (t) :
K_MASS "m"
Thermal Power (W) :
K_POWER "P"     //power corresponding to the heavy metal mass
Time (s) :
K_TIMESTEP "0 t2 t3 t4 ..." //Time when the cross section are updated
Z A I Name (input MLP) : //see explanations below
K_ZAINAME "z a i InputName"
K_ZAINAME "z2 a2 i2 InputName2"
"..."
Fuel range (Z A I min max) :
K_ZAIL "z a i min max" //minimal and maximal proportion of the zai in the fresh
    fuel (heavy nuclei only, ie without oxygen)"
K_ZAIL "z2 a2 i2 min2 max2"
```

The input of MLPs are the atomic proportion of each nuclei present in the fresh fuel (plus time if OneMLPPerTime=false). The InfoFile has to indicates the variable names (nuclei name) you used for the **training of your MLPs**. For instance if the fresh fuel contains ^{238}Pu you will write in the InfoFile :

```
...
Z A I Name (input MLP) :
K_ZAINAME 94 238 0 Pu8//(if Pu8 is the variable name used for 238Pu proportion in
    fresh fuel in your training sample)
...
```

The tag "Fuel range (Z A I min max) :" corresponds to the validity domain of the XSM_MLP model. This indication is not mandatory but its useful to know if the fuel we calculate the cross section is in the domain of validity of the model.

Available XSM_MLP :

- $CLASS_PATH/DATA_BASES/PWR/MOX/XSModel/30Wg_FullMOX : The weight files and .nfo file contained in this folder are representative of a PWR MOX. With the MOX coming from PWR UO2 spent fuels. The specific power is 30W/g oxide. To perform this data base, MURE depletion calculations have been performed using a full MOX assembly with mirror boundaries.
- $CLASS_PATH/DATA_BASES/FBR_Na/MOX/XSModel/ESFR_48Wg : The weight files and .nfo file contained in this folder are representative of a FBR-Na MOX. The specific power is 48W/g oxide. To perform this data base, MURE depletion calculations have been performed using a 1/12 of ESFR like core with mirror boundaries.
- $CLASS_PATH/DATA_BASES/PWR/UOX/XSModel/30Wg_FullUOX : The weight files and .nfo file contained in this folder are representative of a PWR UOX. The specific power is 30W/g oxide. To perform this data base, MURE depletion calculations have been performed using a full UOX assembly with mirror boundaries.
- $CLASS_PATH/DATA_BASES/PWR/MOX_Am/XSModel/30Wg_FullMOX_Am : The weight files and .nfo file contained in this folder are representative of a PWR loaded with (Pu,U,Am)O_2. Plutonium and Americium compositions are representative of compositions in UOX spent fuels. The specific power is 30W/g oxide. To perform this data base, MURE depletion calculations have been performed using an assembly with mirror boundaries.

Training MLPs for cross sections prediction :

Preparation of the training sample :

Like for the equivalence model, first of all you have to create a training sample. This is one of the most important thing since the way of filling the hyperspace of the MLP inputs will influence the accuracy of your model. We suggest to used the Latin Hyper Cube method [McKa 00] to generate many fresh fuel compositions, then, calculates with your favourite neutron transport code (MCNP, MORET, SERPENT ...) the mean cross sections of each fresh fuel for different irradiation time. Please refer to [REFFFBAL MLPXS] for more informations about the space filling and the validation of this cross sections predictor. Once all your calculations are complete you have to convert them into the .dat format (see code frame 7.1). Then type :

```
cd $CLASS_PATH/Utils/XS/MLP/BuildInput
```

Open the file Gene.cxx, looks for @@Change and make the appropriate changes. Then type :

```
g++ -o Gene Gene.cxx `root-config --cflags` `root-config --libs`
Gene PATH_To_dat_Folder/
```

Where PATH_To_dat_Folder/ is the path to the folder containing the .dat files. This program should have built two files :

- TrainingInput.root : This root file contains the fresh fuel inventories and the cross sections values of all the read .dat files. You can plot the data with the root command line tool if you wish. This file is the **Training and testing sample** that will be used for the TMVA training and testing procedure.
- TrainingInput.cxx : This file contains, in a vector, the names of all the MLP outputs. The number of lines in this file is the number of MLP that will be train.

Training and testing procedure :

Once the two TrainingInput (.cxx and .root) are generated type :

```
cd $CLASS_PATH/Utils/XS/MLP/Train
```

Look for @@Change in the file Train_XS.cxx , and make the appropriate changes. Then type :

```
g++ -o Train_XS  `root-config --cflags` Train_XS.cxx `root-config --glibs` -lTMVA
```

According the number of "events" in your .root file and the number of cross sections, the training time can be very very very long. You might want to decrease the number of events (this will probably deteriorate the model accuracy) : look for nTrain_Regression in Train_XS.cxx and change its value to your wanted number of events. And/Or you may want to use more than one processor or perhaps a supercomputer : This is completely doable since the program Train_XS trains only one MLP (one cross section). Indeed the execution line is the following :

```
Train_XS i
```

where i is the index of the cross section in the vector created in TrainingInput.cxx. So feel free to create a script to run the training on a wanted number of processors. For instance let's say you have 40 cross sections and 4 processors, creates 4 files (make them executable) and in the first one type :

```
Train_XS 0
Train_XS 1
...
TrainXS 9
```

continue in the second file, and so on. Then execute all of them. The architecture and weights of each MLP (.xml files) are stored in the folder weights. Rename this folder by the name of the reactor and fuel, then create in this folder the information file (see code frame 11.3). And voilà your new XSM_MLP is ready to be used.

After each training (using by default the half of the events) a testing procedure (using the other half) is performed. This latter consists on executing the trained MLP with input data from a known sample and compare the MLP result to the true value. These data and other informations about the training are stored in file **Training_output_i.root**, with i the index of the cross section. In order to see either the MLPs predictions are accurate or not, the root macro $CLASS_PATH/Utils/XS/MLP/Train/deviations.C plot the distribution of relative differences between model executions and the true values and a Gaussian fit of it. Then, the mean and the standard deviation of the Gaussian fit are stored in file **XS_accuracy.dat** (format : XSName mean std.dev.). Type the following to get, in file XS_accuracy.dat, the mean and the standard deviation of all the MLPs (with N the number of cross sections (number of MLPs)) :

```
cd $CLASS\_PATH/Utils/XS/MLP/Train/
root
.L deviations.C
for(int i=0;i<N;i++) {stringstream ss;ss<<"Training_output_"<<i<<".root";deviations(ss.str()
    .c_str(),0,kTRUE,kFALSE,kFALSE); }
```

The closest to 0 the mean is and the smaller standard deviation, the better.

11.2 How to build an XS Model

The strength of CLASS is to allow the user to build his own Physics models, this section explains how to build a new cross section model and to incorporate it into CLASS. First you have to create the file XSM_NAME.cxx and XSM_NAME.hxx, where NAME is a name you choose. Then open with a text editor the .hxx and copy past the following replacing NAME by the name you want.

Listing 11.4: XSM_NAME.hxx

```
#ifndef _XSM_NAME_HXX
#define _XSM_NAME_HXX
#include "XSModel.hxx"
// add include if needed
using namespace std;
//------------------------------------------------------------------------//
/*!
 Define a XSM_NAME
describe your model
 @authors YourName
 @version 1.0
 */
//------------------------------------------------------------------------
class XSM_NAME : public XSModel
{
        public :

        XSM_NAME(/*parameters (if any)*/);

        ~XSM_NAME();

        EvolutionData GetCrossSections(IsotopicVector IV,double t=0);

        private :
        //your private variables and methods
};
#endif
```

Open the .cxx file and copy past the following in it (replacing NAME by the same name you used in the .hxx).

Listing 11.5: XSM_NAME.cxx

```
#include "XSModel.hxx"
#include "XSM_NAME.hxx"
#include "CLASSLogger.hxx"
#include "StringLine.hxx"

#include <TGraph.h>
//_____________________________________________________________________________
//
//               XSM_NAME
//_____________________________________________________________________________
XSM_NAME::XSM_NAME(/*parameters (if any)*/)
{
// do what you want : for instance save path of eventual files
}
//_____________________________________________________________________________
XSM_NAME::~XSM_NAME()
{
 //delete pointer if any; clear map if any ; empty vector if any
}
//_____________________________________________________________________________
EvolutionData XSM_NAME::GetCrossSections(IsotopicVector IV ,double t)
{
        EvolutionData EvolutionDataFromXSM_NAME = EvolutionData();
        /*************DATA BASE INFO****************/
        EvolutionDataFromXSM_NAME.SetReactorType(fDataBaseRType);//Give the
            reactor name
        EvolutionDataFromXSM_NAME.SetFuelType(fDataBaseFType);//Give the fuel name
        EvolutionDataFromXSM_NAME.SetPower(fDataBasePower);//Set the power W
        EvolutionDataFromXSM_NAME.SetHeavyMetalMass(fDataBaseHMMass);//
            corresponding to this mass (t)

        map<ZAI,TGraph*> ExtrapolatedXS[3];
//... Fill the 3 maps ExtrapolatedXS  according to your model and the
// fresh fuel composition given by argument IsotopicVector IV
// argument double t may be not used.

        /*****THE CROSS SECTIONS***/
        EvolutionDataFromXSM_NAME.SetFissionXS(ExtrapolatedXS[0]);
        EvolutionDataFromXSM_NAME.SetCaptureXS(ExtrapolatedXS[1]);
        EvolutionDataFromXSM_NAME.Setn2nXS(ExtrapolatedXS[2]);

return EvolutionDataFromXSM_NAME;
}
```

Then, edit these two files to make the function XSM_NAME::GetCrossSections to return the cross sections in a EvolutionData object. (*In this case, the EvolutionData only contains the 1 group cross section without the inventory evolution, the power and the corresponding mass.*)
To do so you have to fill three maps (ExtrapolatedXS in .cxx), one for fission, one for (n,γ), and one for $(n,2n)$. Each map associates a nucleus (a ZAI) to a TGraph. A TGraph is a root object, here, it contains the cross section (barns) evolution over time (seconds). If your are not comfortable with TGraph refer to the root website [1]

Now that your cross section model is ready, two choices are offered to you. You can compile the two files of your model with your CLASS input or you can add this model to the CLASS package. The second option will modify the CLASS software and we will be no longer able to troubleshoot your scenario. So use the second option only if you are a completely independent user !

[1] http://root.cern.ch/root/html/TGraph.html

11.2.1 Compile your cross section model with your CLASS executable :

```
g++ -g -O -I $CLASS_include -L $CLASS_lib -lCLASSpkg `root-config --cflags`
        `root-config --libs` -fopenmp -lgomp -Wunused-result -c My_MODEL.cxx

\rm CLASS* ; g++ -o CLASS_exec MyScenario.cxx My_MODEL.o -I $CLASS_include -L $CLASS_lib -
    lCLASSpkg `root-config --cflags` `root-config --libs` -fopenmp -lgomp -Wunused-result
```

11.2.2 Your cross section model in the CLASS library :

Move your XSM_NAME.hxx and XSM_NAME.cxx in $CLASS_PATH/source/Model/XS/. Then open with your favourite text editor the file $CLASS_PATH/source/src/Makefile, find "OBJMODEL" and add $(XSM)/XSM_NAME.o within the others $(XSM) objects. Then re-compile CLASS, fix the compilation errors ;) and voilà your cross section model is now available in the CLASS library.

Chapter 12
Irradiation Model

The irradiation model is the Bateman equations solver. It is used for the calculation of fuel depletion in reactor. The decay depletion (without neutron flux) is not managed by an irradiation model but with a decay data bases (see section 7.3.2).

12.1 Available Irradiation Model

At the moment, there is two Irradiation Model available. The two solvers differs according to the numerical integration method used. The Irradiation Model IM_RK4 uses the fourth order Runge-Kutta method. And IM_Matrix uses the development in a power series of the exponential of the Bateman matrix.

Implementation in a .cxx :

Listing 12.1: Irradiation Model

```
#include "CLASSHeaders.hxx"
#include "Irradiation/IM_RK4.hxx"
//#include "Irradiation/IM_Matrix.hxx"
..
using namespace std;
int main()
{
//...
        IM_RK4* Solver = new IM_RK4(LogObject); // or new IM_RK4(); // uses a
            default logfile
//      IM_Matrix* Solver = new IM_Matrix(LogObject); // or new IM_Matrix(); //
    uses default logfile
        PhysicsModels* PHYMOD = new PhysicsModels(XSMOX, EQMLINPWRMOX, Solver);
//...
}
```

LogObject is a CLASSLogger object (see section 7.4).

12.1.1 How to build an Irradiation Model

The strength of CLASS is to allow the user to build his own Physics models, this section explains how to build a new Bateman solver (Irradiation Model) and to incorporate it into CLASS. First you have to create the file IRM_NAME.cxx and IRM_NAME.hxx, where NAME is a name you choose. Then open with a text editor the .hxx and copy past the following replacing NAME by the name you want.

Listing 12.2: lRM_NAME.hxx

```
#ifndef _IRM_NAME_HXX
#define _IRM_NAME_HXX

#include "IrradiationModel.hxx"
using namespace std;
class CLASSLogger;
class EvolutionData;
//-------------------------------------------------------------------------//
/*!
 Define a IM_NAME
Description
 @author YourName
 @version 3.0
 */
//-------------------------------------------------------------------------
class IM_NAME : public IrradiationModel
{
        public :
        IM_NAME(); //constructor

        /*!
        virtual method called to perform the irradiation calculation using a set
            of cross sections.
         \param IsotopicVector IV isotopic vector to irradiate
         \param EvolutionData XSSet set of corss section to use to perform the
             evolution calculation
         */
        EvolutionData GenerateEvolutionData(IsotopicVector IV, EvolutionData XSSet
            , double Power, double cycletime);
        //}
        private :
        //declare your private variables here
};
#endif
```

Open the .cxx file and copy past the following in it (replacing NAME by the same name you used in the .hxx).

Listing 12.3: lRM_NAME.cxx

```
#include "IRM_NAME.hxx"
#include "CLASSLogger.hxx"
#include <TGraph.h>
//Add whatever includes
using namespace std;
//_____________________________________________________________________________
IRM_NAME::IRM_NAME():IrradiationModel(new CLASSLogger("IRM_NAME.log"))
{
        // do what you want
}
//_____________________________________________________________________________
EvolutionData IRM_NAME::GenerateEvolutionData(IsotopicVector FreshFuelIV,
    EvolutionData XSSet, double Power, double cycletime)
{
        EvolutionData GeneratedDB = EvolutionData(GetLog());
        GeneratedDB.SetPower(Power );
        GeneratedDB.SetReactorType(ReactorType );

//Your Solver algorithm has to  fill GeneratedDB with the calculated inventories
//using :
GeneratedDB.NucleiInsert(pair<ZAI, TGraph*> (ZAI(Z,A,I), new TGraph(SizeOfpTime,
    pTime, pZAIQuantity)));

        return GeneratedDB;
}
```

The function **GenerateEvolutionData** returns a *EvolutionData* (see section 7.3) containing the inventories evolution over time. This has to be done according to the fresh fuel composition (**FreshFuelIV**), to the mean cross sections (**XSSet**), to the (**Power** : thermal power (W)) and to the irradiation time (**cycletime** (seconds)). To fill this *EvolutionData* you have to call the method **NucleiInsert** which associates a nucleus (a ZAI) to a root object TGraph [1]. This TGraph is the evolution (**pZAIQuantity** in atoms) of this associated nucleus (**ZAI(Z,A,I)**) over time (**pTime** in seconds). This TGraph has **SizeOfpTime** points.

After making the appropriate changes in this two files to make the function **GenerateEvolutionData** to return the fuel evolution (fill free to look at $CLASS_PATH/source/Model/Irradiation/*xx to get inspiration), two choices are offered to you. You can compile the two files of your model with your CLASS input or you can add this model to the CLASS package. The second option will modify the CLASS software and we will be no longer able to troubleshoot your scenario. So use the second option only if you are a completely independent user !

12.1.2 Compile your Irradiation model with your CLASS executable :

```
g++ -g -O -I $CLASS_include -L $CLASS_lib -lCLASSpkg `root-config --cflags`
        `root-config --libs` -fopenmp -lgomp -Wunused-result -c My_MODEL.cxx

\rm CLASS* ; g++ -o CLASS_exec MyScenario.cxx My_MODEL.o -I $CLASS_include -L $CLASS_lib -
    lCLASSpkg `root-config --cflags` `root-config --libs` -fopenmp -lgomp -Wunused-result
```

12.1.3 Your Irradiation model in the CLASS library :

Move your IRM_NAME.hxx and IRM_NAME.cxx in $CLASS_PATH/source/Model/Irradiation/. Then open with your favourite text editor the file $CLASS_PATH/source/src/Makefile, find "OBJMODEL" and add $(IM)/IRM_NAME.o within the others $(IM) objects. Then re-compile CLASS, fix the compilation errors ;) and voilà your irradiation model is now available in the CLASS library.

[1] http://root.cern.ch/root/html/TGraph.html

Part V

CLASSGui : The results viewer

To use the CLASSGui :

```
CLASSGui MyCLASSOutput.root
```

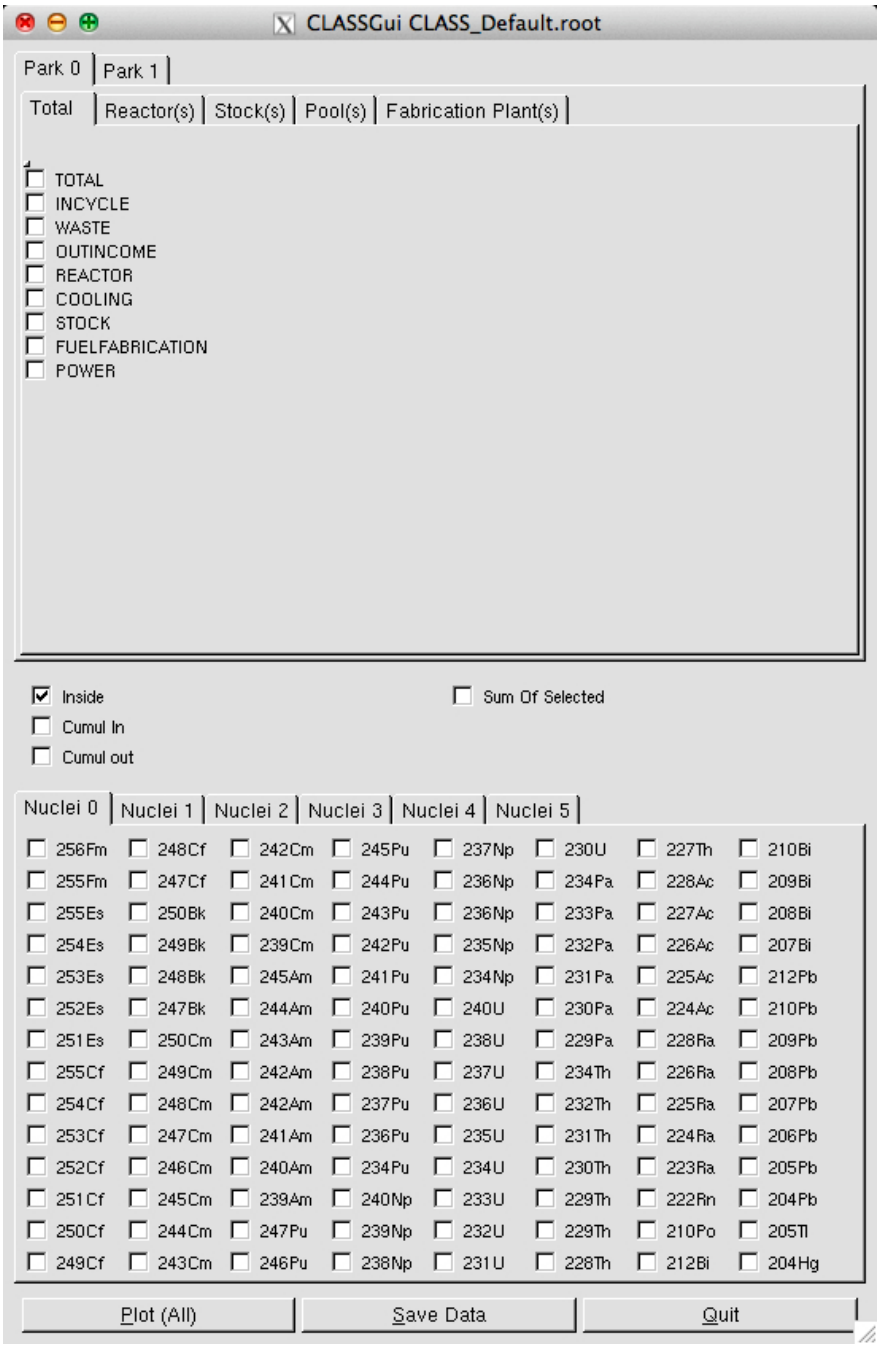

Figure 12.1: The graphical user interface for CLASS outputs

Bibliography

[Brun 97] R. Brun and F. Rademakers. "ROOT: An object oriented data analysis framework". *Nucl. Instrum. Meth.*, Vol. A389, pp. 81–86, 1997.

[Hoec 07] A. Hoecker, P. Speckmayer, J. Stelzer, J. Therhaag, E. von Toerne, and H. Voss. "TMVA: Toolkit for Multivariate Data Analysis". *PoS*, Vol. ACAT, p. 040, 2007.

[Leni 15] B. Leniau, B. Mouginot, N. Thiolliere, X. Doligez, A. Bidaud, F. Courtin, M. Ernoult, and S. David. "A neural network approach for burn-up calculation and its application to the dynamic fuel cycle code CLASS". *Annals of Nuclear Energy*, Vol. 81, pp. 125–133, 2015.

[McKa 00] M. McKay, R. Beckman, and W. Conover. "A comparison of three methods for selecting values of input variables in the analysis of output from a computer code". *Technometrics*, Vol. 42, No. 1, pp. 55–61, 2000.

[Moug 15] B. Mouginot, B. Leniau, N. Thiolliere, A. Bidaud, F. Courtin, X. Doligez, and M. Ernoult. "{MOX} fuel enrichment prediction in {PWR} using polynomial models". *Annals of Nuclear Energy*, Vol. 85, No. , pp. 812 – 819, 2015.