User Guide - CLASS v3.0

Core Library for Advanced Scenario Simulation

B. MOUGINOT[1] & B. LENIAU[2]

[1] Baptiste.Mouginot@subatech.in2p3.fr
[2] Baptiste.Leniau@subatech.in2p3.fr

CNRS:IN2P3:Subatech:Erdre

Abstract

Table of Contents

List of Figures

Part I

Introduction

code de scenar tatati c’est gnial ça sert à ça ça et ça ... donner le forge in2p3

Part II

First Steps

Chapter 1

Package Contents

Ya quoi dans ce que je viens de downloader

Chapter 2

Install procedure

2.1 Requirement

- User skills : Good knowledge of C++. Abilities in using Root (cern). Experience in depletion codes and neutron transport codes.
- OS : CLASS is known to work under Linux (64 bits) and MacOSX (64 bits). It has never been tested on any Windows distribution.
- Root (CERN) : CLASS uses Root to store output data. The graphical user interface CLASS-Gui is based on Root. Some algorithms uses the TMVA module of Root.
- C++ compiler : we recommend to use a gnu compiler like gcc4.8. If your platform is DARWIN (Mackintosh OSX) we strongly recommend not to use the clang compiler
  You should install macport. then types this following command in terminal :

```
sudo port install gcc48
sudo port select --set gcc mp-gcc48
```

IMPORTANT NOTE :

The actual root package (version 5.34/20) and earlier (and maybe latter) has a memory leak issue when using TMVA leading to a **freeze of your computer.** To avoid this dramatical error to happen do the following :
If the thread RootTalk [1] or RootSupport [2] indicates status solved then download and install the more recent ROOT version.
If the status is still unresolved proceed as follow :
Open with your favourite text editor the file $ROOTSYS/tmva/src/Reader.cxx ($ROOTSYS is the

[1]http://root.cern.ch/phpBB3/viewtopic.php?f=3&t=18360&p=78586&hilit=TMVA#p78586
[2]https://sft.its.cern.ch/jira/browse/ROOT-6551

path to your ROOT installation folder) and replace the following :

```
TMVA::Reader::~Reader( void )
{
   // destructor

   delete fDataSetManager; // DSMTEST

   delete fLogger;
}
```

by :

```
TMVA::Reader::~Reader( void )
{
   // destructor
   std::map<TString, IMethod* >::iterator itr;
   for( itr = fMethodMap.begin(); itr != fMethodMap.end(); itr++) {
      delete itr->second;
   }
   fMethodMap.clear();

   delete fDataSetManager; // DSMTEST

   delete fLogger;
}
```

then type in your terminal :

```
cd $ROOTSYS
sudo make -j
```

2.2 Installation

Decompress the CLASS.tar.gz in your wanted location [3]. Then type in terminal:

[3] $CLASS_PATH is the path of your CLASS installation folder

```
cd $CLASS_PATH/
mkdir lib
cd  source/src
make -j
make install
```

Then to install the Graphical User Interface :

```
cd $CLASS_PATH/gui
mkdir bin
make -j
```

Finally add the following environment variables (in your .tcsh or .csh):

```
setenv CLASS_PATH YourPathToCLASS
setenv CLASS_lib ${CLASS_PATH}/lib
setenv CLASS_include ${CLASS_PATH}/source/include
setenv PATH ${PATH}:${CLASS_PATH}/bin/gui
```

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 :

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

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.
- 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/annotated.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

All time in CLASS should be written in second. It corresponds to the cSecond, a CLASS c++ type, which are a **long long int** going, in 32 bits **and** 64 bits, up to $(2^{63}-1)$ s $\sim 2.9 \cdot 10^{11}$ years, enough for any electro-nuclear scenarios one can consider....

5.2 CLASS working process principle

image : shéma de principe de class

Chapter 6

Facilities descriptions

All the facilities in CLASS project are regrouped inside a large group 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 backend facilities.

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 these 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.

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

6.2.2.1 Fixed Fuel

Reactor using fixed fuel, which load always the same fresh fuel, and unload it with always the same burnup (same spent fuel...), to declare a reactor proceed 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

```
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) is 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 arguments of the two constructor 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

6.3 CLASSBackEnd

The CLASSBackEnd class is a master class which aims to regroup all common properties of the fuel back-end facilities. All other back-end facilities in CLASS inherit of the CLASSBackEnd. In CLASS, 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 CLASSBack end without associated downstream factory. All the incoming material are stored individually. During the storage, the depletion by decay is taken into account. The storage has to be defined as follow :

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

6.3.2 Pool

Pool is a CLASSBack end with an associated downstream factory. All incoming material will be pushed in the downstream factory after a set time, so called CoolingTime. All the incoming material are stored individually. During the cooling process, the depletion by decay is taken into account. The storage 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 pushed after cooling 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 follow the isotopic separation efficiency. It 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 a 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;
IV_MA.Add(93, 237, 0, 1.);
IV_MA.Add(95, 242, 1, 1.);
IV_MA.Add(96, 245, 0, 1.);
// ...
MySeparationPlant->SetBackEndDestination(aCLASSBackEnd1,
          IV_MA,
          2000*365.25*24.3600);

IsotopicVector IV_Pu;
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, the separation of Minor Actinides start in 2000 sending it to the CLASS-BackEnd *aCLASSBackEnd1* (the rest going to the WASTE). The separation of the plutonium start in 2005 (send in the *aCLASSBackEnd2*) and the separation of uranium in 2010.
Note that between 2005 and 2010, both MA and Pu are separated and sent respectively to *aCLASS-BackEnd1* 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 send to WASTE.
Furthermore, the separation of Actinides Minor has an efficiency of 100%, Pu of 80% and U of 50%.

6.4 Fabrication Plant

The FabricationPlant is the facility which takes care about the fuel Fabrication. The "action" in FabricationPlant appends before the beginning of Cycle of a reactor: One fabrication time (Fabrication duration) before the BOC, it start the building process of the fuel.
First it sort the different stock in the different input Storage according the users priority. Then take the EquivalenceModel in PhysicsModels of the reactor, ask it how to build a fuel with the correct

properties using some stock available. The EquivalenceModel provide a list a fraction to take in each stock. According to this fraction list, the FabricationPlant take the fraction is each stock and build the reprocessed fuel. Once the reprocess fuel is made, it ask to the PhyscisModel to calculate its evolution and store it in EvolutionData form until the reactor load the fuel.
Between the fuel fabrication and the fuel loading in the reactor, the deplay through decay of the fuel is of course taking into account.
Note that, the FabricationPlant provide to the EquivalenceModel a list of stock which have virtually decay during the fabrication time in order to build a proper fuel.

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.

One must also provide a list of Storage used to extract the Fissile part of the fuel by using :

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

And if necessary it is possible to storage to extract fertile isotopes using :

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

If no Fertile Storage are defined, the fertile part is taken from outside of the Scenario. By default the unuse part of the stock is send to WASTE.But it is possible to set a storage where the unuse part of the stock using :

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

6.5 PathWay between Faiclity

As explain previously, there are 3 different facility family, the FabricationPlant, the reactor, and the CLASSBackEnd. The CLASSBackEnd facilities can't pull material inside, there is always a other facility which push material inside the CLASSBackEnd, but some can also push material inside other facilities: the SeparationPlant and the Pool. The Storage can only store materials.
The reactor take is fuel in a FabricationPlant and push the irradiated fuel in a CLASSBackEnd.

The FabricationPlant take its materials inside storage and stock the reprocessed fuel its makes unties the BoC. We propose in the following 4 example of pathway between difference facilities. The point here is only to illustrated possible pathway, but the illustration may not be exhaustive. Furthermore, almost any composition between these examples could be made.

6.5.1 Reactor with fixed fuel and a Storage

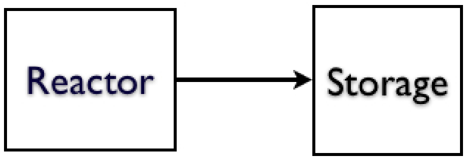

Figure 6.1: Shematic Pathway

```
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

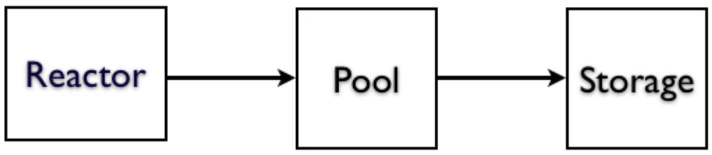

Figure 6.2: Shematic Pathway

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

Storage* MyStorage = new Storage(Logger);
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

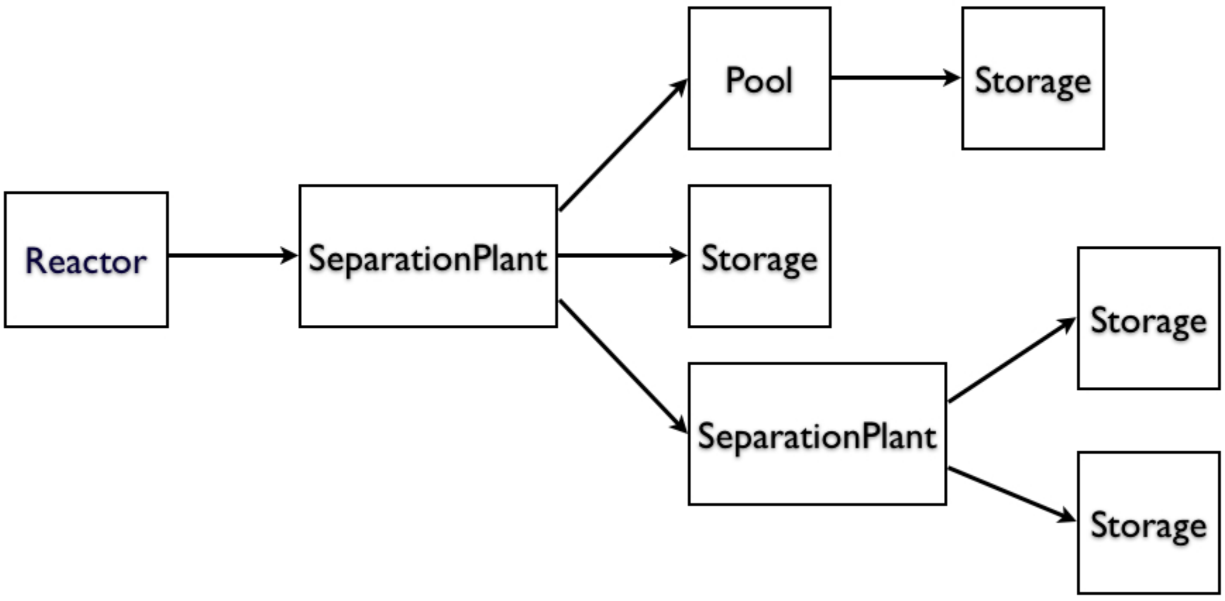

Figure 6.3: Shematic Pathway

```
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.
SeparationPlan* 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 mySeparationPlan1.
SeparationPlan* 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);
```

6.5.4 Reactor, a FabricationPlant, a Pool and a Storage

```
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,
    2*365.25*24*3600);

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

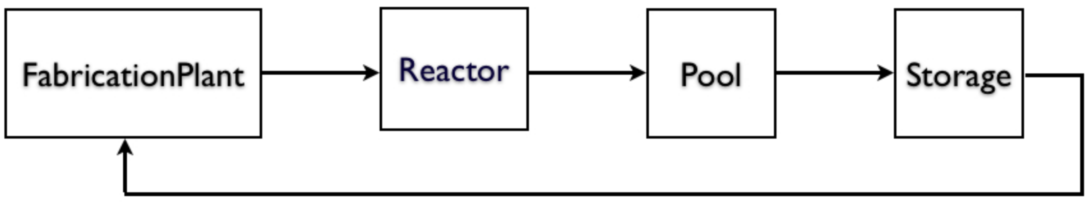

Figure 6.4: Shematic Pathway

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 number 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 pincipales operation have been defined on IsotopicVector. The following illustrates the possible operation 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 += 52 * ZAI(92, 235, 0); // 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 on the terminal.
@@ Link WARNING
To insure the quality and the reliability of the simulation, the fIsotopicQuantityNeeded MUST remain empty.

7.2.2 Print method

You can use the Print() method to write the composition of an IsotopicVector. When printing the IsotopicVector composition present nuclei, as well as the *needed* one, are written with their corresponding quantity (unit: nucleus number).

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.

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 the keff and are not mandatory. Note that neutron flux and keff are not used in CLASS. The EvolutionData MUST contain the power and can contain the heavy metal mass, 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 (@@BaM)
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 $PATH_TO_CLASS/Data/@@@.

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.

Part IV

Physics Models

Chapter 8

Description and implementation

A Physics Models 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 method.

A Physics model is called in the CLASS input like the following example :

Implementation in a .cxx :

Listing 8.1: PhysicsModels

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

  EQM_MLP_MOX* Equivalence  = new EQM_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 Physics model called "PHYMOD" is defined, it contains the bateman solver "Solver" which is the Runge Kutta (4th 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 its corresponding sections.

All the existing models are define in the following sections, furthermore, the way to build its own Model is presented.

Chapter 9

Equivalence Model

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

9.1 Available Equivalence Models

The CLASS package contains, for the moment, 4 different equivalence models where three are related to the building of fuels for a PWR-MOX and one to the building of PWR-UOX fuels :

9.1.1 PWR-MOX models :

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

9.1.1.1 Linear BU model : EQM_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 burn-up accessible for a set fuel using its initial composition using a simple linear modelisation (equation 9.1):

$$BU_{max} = \alpha_0 + \sum_{i}^{N} \alpha_i \cdot n_i, \tag{9.1}$$

where BU_{max} represent the maximal accessible burn-up 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 burn-up, but a large burn-up region. One can see the problem guessing it is possible to build a set a fuel evolution reaching exactly a unique burn-up (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 burn-up description model, one should test the validity of the model, on many random compositions by example...

9.1.1.2 Quadratic Model : EQM_QUAD_MOX

The $\%_{Pu}$ is calculated according a quadratic model. See equation 9.2.

$$\%_{Pu} = \alpha_0 + \sum_{i \in Pu}^{N} \left(\alpha_i \cdot n_i + \sum_{j \leq i} \alpha_{ij} \cdot n_i \cdot n_j \right), \tag{9.2}$$

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 burn-up 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. For instance the α given in file $CLASS_PATH/DataBase/Equivalence/PWR_MOX_45GW_3Batch_2y.dat are representative of a PWR-MOX with a maximal burn-up of $45 GWd/tHM$, a fuel management of 3 batches, and a time between separation and irradiation of 2 years.

The file containing the weights is formatted as follow :

```
PARAM "238Pu 238Pu*238Pu 238Pu*239Pu 238Pu*240Pu 238Pu*241Pu 238Pu*242Pu 239Pu
    239Pu*239Pu 239Pu*240Pu 239Pu*241Pu 239Pu*242Pu 240Pu 240Pu*240Pu 240Pu
   *241Pu 240Pu*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 [@@PAPIER BAM].

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

Implementation in a .cxx

Listing 9.1: Equivalence Model EQM_QUAD_MOX

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

Available weight file (.dat) :

- **@@@ BAM**
- **@@@ BAM**
- ...

9.1.1.3 Neural network model : EQM_MLP_MOX

This equivalence model is based on a Multi Layer Perceptron (MLP) and predict the amount of plutonium needed to reach **any burn-up**. The MLP inputs are the isotopic compositions of the plutonium (**including** ^{241}Am), the enrichment of depleted uranium, and the targeted burn-up. The output is the plutonium content needed to reach the burn-up. This method uses the neural networks of the root module TMVA (@@@ Ref TMVA). 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 9.2: Equivalence Model EQM_MLP_PWR_MOX

```
...
#include "Equivalence/EQM_MLP_PWR_MOX.hxx"
...
int main()
{
...
EQM_MLP_PWR_MOX* Equivalence = new  EQM_MLP_PWR_MOX( LogObject, TMVAWeightPath
    );
// or
// EQM_MLP_PWR_MOX.* Equivalence = new  EQM_MLP_PWR_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 burn-up 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/DataBases/Equivalence/EQM_MLP_PWR_MOX_3batch.xml has been generated from the file $CLASS_PATH/Utils/Equivalence/PWR_MOX_MLP/Train_MLP.cxx To train a new MLP from your own training sample proceed as follow :

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

Where YourTrainingData.root is a root file containing a TTree filled with fuel compositions and corresponding burn-ups. 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
/*********************INITIALISATIONNN******************/
//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 Burn-Up
  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 Burn-up******/
    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 [@@Papier BaL].

Available weight file (.xml) :

- **$CLASS_PATH/DataBases/Equivalence/EQM_MLP_PWR_MOX_3batch.xml** : 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[@@@BaL paper].

9.1.2 PWR-UOX model :

9.1.2.1 Linear Model: EQM_LIN_UOX

@@@BAM

9.2 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 9.3: 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 9.4: 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 Fissil : 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 Fertil : 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 burn-up 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 !

9.2.1 Compile your equivalence model with your CLASS executable :

@@BAM

9.2.2 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 10

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 9). The mean cross sections are required to compute fuel depletion in a reactor.

10.1 Available XS Models

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

10.1.1 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 10.1).

$$d = \sqrt{\sum_i w_i \cdot (n_i^{DB} - n_i^{new})^2}, \tag{10.1}$$

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 10.1: 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 more details on this method please refer to [ref @@@ BAM physor].

Available .idx file :

- **@@@ BAM**
- **@@@ BAM**
- ...

For MURE user only : The program $CLASS_PATH/Utils/XS/CLOSEST/WriteDataBase 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/XS/CLOSEST/WriteDataBase -h
@@BAM
```

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

10.1.2 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 10.2: 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 setted 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. Format of InfoFileName is :

Listing 10.3: Information file format

```
ReactorType :"ReactorName"  //without space
FuelType :"FuelName" //without space
Heavy Metal (t) :"m"
Thermal Power (W) :"P"     //power corresponding to the heavy metal mass
Time (s) :"0 t2 t3 t4 ..." //Time when the cross section are updated
Z A I Name (input MLP) : //see explanations below
"z a i InputName"
"z2 a2 i2 InputName2"
"..."
```

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) :
94 238 0 Pu8//(if Pu8 is the variable name used for 238Pu proportion in fresh
   fuel in your training sample)
...
```

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 [@@@REFF] 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 10.3). 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.

10.2 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 10.4: 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 10.5: 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 !

10.2.1 Compile your cross section model with your CLASS executable :

@@BAM

10.2.2 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.

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

Chapter 11

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).

11.1 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 11.1: 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).

11.1.1 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 11.2: IRM_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 11.3: IRM_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.

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

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 !

11.1.2 Compile your cross section model with your CLASS executable :

@@BAM

11.1.3 Your cross section 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.

Part V

CLASSGui : The results viewer

Part VI

Input examples

Part VII

In development