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 :

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

    delete fDataSetManager; // DSMTEST

    delete fLogger;
}
```

by :

```cpp
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 facility in CLASS project are regroup 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 be 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. Its includes 3 variables needed by CLASS environment:

```
private :
  int   fId;                //!< Identity of the Facility inside the Parc
  int   fFacilityType;      ///< Type of facility :
                            /// \li 4 reactor ,
                            /// \li 8 Pool ,
                            /// \li 16 FabricationPlant .
  Scenario*   fParc;        //!< Pointer to the main Parc
```

The *fId* variable correspond to the unique identity number allowing to differentiate all the facility of a certain type. The *fFacilityType* variable correspond to an identity number allowing to separate the different type of facilities. Those to variable are "**private**", which mean it is not possible to access to them directly, one **must** to use there Get-xxx() and the Set-xxx() function (even inside the daughter class such as reactor...). *fParc* is a pointer to the main park, which allow to access to the rest of the park. Be careful the *fParc* pointer main not be assigned if the facility is not included in a park...

The CLASSFacility also includes all the generic variable and method for time and simple fuel management:

```cpp
protected :
  bool    fIsStarted;     ///< True if Running, False Otherwise
  bool    fIsShutDown;    ///< True if the facility is stoped, False Otherwise
  bool    fIsShutDown;    ///< True if Reaching the End of a Facility Cycle


  cSecond    fInternalTime;    ///< Internal Clock
  cSecond    fInCycleTime;     ///< Time spend since the beginning of the last
      Cycle
  cSecond    fCycleTime;       ///< Cycle duration Time

  IsotopicVector  fInsideIV;   ///< All IV in the Facility (fuel for reactor,
      total for all others...)
  IsotopicVector  fCumulativeIVIn;   ///< All IV in the Facility (fuel for
      reactor, total for all others...)
  IsotopicVector  fCumulativeIVOut;  ///< All IV in the Facility (fuel for
      reactor, total for all others...)

    //********* Internal Parameter *********//
private :
  [...]
  cSecond    fCreationTime;    ///< CLASS Universal Time of Creation
  cSecond    fLifeTime;        ///< Time of life Of the Reactor (Operating's
      Duration)
```

fIsStarted, *fIsShutDown*, *fIsShutDown* allows to the Dump() method to determine the state of the facility and do what is necessary.

fInternalTime, *fInCycleTime*, *fCycleTime* variable allows the time the time management inside the facility. *fInternalTime* correspond to the last time until the evolution of the facility as been calculated. *fCycleTime* the time length of a cycle in this facility (note that it could be irradiation cycle in a Reactor, fabrication time in a FabricationPlant, or cooling time in a Pool). And *fInCycleTime* correspond to the time already past in the current cycle.

fCreationTime, *fLifeTime* are **private** and used to define the creation time of the facility, and its operation time length.

Besides all the time management of the facility, it also contain the basic tool for fuel management the three IsotopicVector : *fInsideIV*, *fCumulativeIVIn*, *fCumulativeIVOut*, which correspond respectively to the isotopic vector present in the facility, to the cumulative income inside the facility, and the cumulative outcome outside the facility.

By default af the Get-xxx() and Set-xxx() method associated to those variable are define, and some can be overloaded.

The CLASSFacility also comes with 2 virtual method (which means one must overloading it when defining a new facility) : *Evolution(cSecond t)* and *Dump().* They are both used to perform the evolution of the facility. Where the first one (*Evolution*) is used to specify the change inside the facility (mainly fuel evolution), the second one (*Dump*) is used to deal the exchange between facility, such as refilling a reactor or send a fuel to the stock at the end of cooling.

6.2 Reactor

6.2.1 Generalities

The aim of this class is to deal the evolution of the fuel inside a reactor. The fuel state of the reactor is describe in the IsotopicVector fInsideIV (which are inherit from the CLASSFacility class). Its evolution is **always** contain in the EvolutionData *fEvolutionDB*.
There are 2 way to provide the EvolutionData to the reactor. In the case of fixed fuel 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 Constructor

There are many ways to define a reactor.

6.2.2.1 Normal constructor

```
Reactor();    ///< Normal Constructor.
```

Just define a simple reactor without fuel, starting time, mass of fuel, time of life or anything.

```
Reactor(CLASSLogger* log);
```

Just define a simple reactor without fuel, starting time, mass of fuel, time of life or anything, but set a CLASSLogger *log* to send the CLASS message.

6.2.2.2 Fixed fuel constructor

Constructor defining a reactor using fixed fuel.

```
Reactor(CLASSLogger* log, EvolutionData evolutivedb, CLASSBackEnd* CBE,
  cSecond creationtime, cSecond lifetime,
  double power, double HMMass, double BurnUp, double CapacityFactor = 1);
```

```
Reactor(CLASSLogger* log, EvolutionData evolutivedb, CLASSBackEnd* Pool,
  cSecond creationtime, cSecond lifetime,
  cSecond cycletime, double HMMass, double BurnUp);
```

The evolution of the fuel is given by the EvolutionData *evolutivedb* the mass of heavy metal in the reactor core by *HMMass*, the power by *Power* and its capacity factor by *CapacityFactor*.
To avoid mixing between constructor, only 2 constructors exist to set the triplet (power, burnup, cycle time). (if more 2 constructor would have the same number of input variables.)
After irradiation the fuel goes in the CLASSBackEnd CBE, which as seen section can be any CLASSBackEndFacility (SeparationPlant, Pool or storage).

6.2.2.3 Reprocessed fuel constructor

As well as in the fixed fuel constructor, the mass of heavy metal in the reactor core by *HMMass*, the power by *Power* and its capacity factor by *CapacityFactor*. Only 2 constructors exist to set the triplet (power, burnup, cycle time).
The PhysicsModels is *fuelDB*, and the FabricationPlant is *FabricationPlant*.

```
Reactor(CLASSLogger* log, PhysicsModels   fueltypeDB,
  FabricationPlant* fabricationplant, CLASSBackEnd* Pool,
  cSecond creationtime , cSecond lifetime, cSecond cycletime,
  double HMMass, double BurnUp);
```

```
Reactor(CLASSLogger* log, PhysicsModels   fueltypeDB,
  FabricationPlant* fabricationplant, CLASSBackEnd* Pool,
  cSecond creationtime , cSecond lifetime,
  double Power, double HMMass, double BurnUp, double CapacityFactor);
```

6.2.3 Main methods

Despite all the Get-xxx() and Set-xxx() method, Reactor possess 3 main method, Evolution(), Dump() and SetStorage() (yes this is a Set-xxx() method...).

6.2.3.1 Evolution

The evolution function is quite simple, it only follow the EvolutionData, containing all the information about the fuel evolution. When called the *Evolution(cSecond t)* method, the reactor will perform its evolution until the time *t*.

> **Note that, the reactor MUST have a time step at the exact second of its end of cycle, start or shutdown, if not the CLASS will break returning :**
> "Evolution is too long! There is a problem in Reactor evolution at ..."
> **as an error, giving also some time information about the break.**

Beside the verification if the reactor is started, and its start, the evolution can encounter 2 cases : if the evolution stop during and cycle to at the end of a cycle.

```
if( EvolutionTime + fInCycleTime == fCycleTime )     //End of Cycle
{
  fIsAtEndOfCycle = true;
  fInternalTime += EvolutionTime;         // Update Internal Time
  fInCycleTime += EvolutionTime;          // Update InCycleTime
  if(t >= GetCreationTime() + GetLifeTime()) // if the Next Cycle don't exist
    fIsShutDown = true;
}
else if(EvolutionTime + fInCycleTime < fCycleTime )     // During Cycle
{
  fInternalTime += EvolutionTime;         // Update Internal Time
  fInCycleTime += EvolutionTime;          // Update InCycleTime

  fInsideIV = fEvolutionDB.GetIsotopicVectorAt( (cSecond)(fInCycleTime/
      fEvolutionDB.GetPower()*fPower) ); // update the fuel composition
  if(t>=GetCreationTime() + GetLifeTime())  fIsShutDown = true;
}
else
{
  // Evolution goes after the end of cycle.... check it
  exit(1);
}
```

In the first case, the reactor will only update its *fInsideIV* to the corresponding state. In the second case, only flag the end of cycle, the rest of the job is don in the *Dump()* method.

6.2.3.2 Dump

This method deal all the exchange with the other facility of the scenario. It is divided in two part. The first one trash the used fuel, detecting which case it is (normal end of a cycle, end of cycle ad reactor shutdown, or shutdown during a cycle).

```
if(fIsAtEndOfCycle  && !fIsShutDown )
{
  if(fIsStarted  )              // A Cycle has already been done
  {
    fOutBackEndFacility->AddIV(fInsideIV);
    AddCumulativeIVOut(fInsideIV);
  }
  else fIsStarted = true;         // Just start the first cycle

}
else if (fIsAtEndOfCycle  && fIsShutDown )       // shutdown at end of Cycle
{

  fOutBackEndFacility->AddIV(fIVOutCycle);
  AddCumulativeIVOut(fIVOutCycle);
  fInsideIV.Clear();
  fInCycleTime = 0;
  fIsStarted = false;          // shut down the Reactor
}
else if (!fIsAtEndOfCycle && fIsShutDown )        // shutdown during Cycle
{
  fOutBackEndFacility->AddIV(fInsideIV);
  AddCumulativeIVOut(fInsideIV);
  fInsideIV.Clear();
  fInCycleTime = 0;
  fIsStarted = false;          // shut down the Reactor
}
```

The second part correspond to refilling of the reactor... For the fixed fuel :

```
fIsAtEndOfCycle = false;
GetParc()->AddGod(fIVInCycle);
fInsideIV  = fIVBeginCycle;
AddCumulativeIVIn(fIVBeginCycle);
fInCycleTime = 0;
```

where the fuel material is taken outside of the scenario.

Or for the reprocessed fuel :

```
fIsAtEndOfCycle = false;
SetNewFuel(fFabricationPlant->GetReactorEvolutionDB(GetId()));
fFabricationPlant->TakeReactorFuel(GetId());
fInsideIV  = fIVBeginCycle;
AddCumulativeIVIn(fIVBeginCycle);
fInCycleTime = 0;
```

where the fuel and the corresponding EvolutionData is builded by a FabricationPlant using the PhysicsModel.

6.2.3.3   SetStorage

The SetStorage() method must be use with caution !! Because its use will modify the normal operation of the reactor. It can be used only in the case of a fixed fuel reactor and not stock management (all stock ar mixed inside the storage), when one want to take the material inside the stock instead of outside of the scenario. In this case, an isotopic separation of the provided stock is done, the needed isotope are taken from the stock, the missing part is provided by the OutSide of the scenario. The refilling part of the Dump() method becomes :

```
fIsAtEndOfCycle = false;
IsotopicVector BuildIVtmp ;
IsotopicVector GodPart;
//Get The Storage Compostion
BuildIVtmp.Add(fStorage->GetInsideIV().GetIsotopicQuantity());
//Get the rest after IVIn creation
BuildIVtmp -= fIVInCycle;
//Get the God part form this rest
GodPart.Add(BuildIVtmp.GetIsotopicQuantityNeeded()) ;
//Take what you can from Storage...
fStorage->TakeFromStock( fIVInCycle - GodPart);
//And Get the rest from God
GetParc()->AddGod(GodPart);
fInsideIV  = fIVBeginCycle;
AddCumulativeIVIn(fIVBeginCycle);
fInCycleTime = 0;
```

Again this method is really tricky to use and need to be used with great caution!!!

6.2.4 CLASSFuelPlan

If you don't want to change the fuel type (from a fixed fuel to an other, a fixed fuel to a reprocessed one, changing the final burnup of a fuel...) inside a reactor during its operation stay away from this part.

The aim of this class is to allow user to change the fuel parameter during the operation of the reactor, by example setting a reactor with a UOX fuel enriched at 3% for 3 cycles, then changing to 5% UOX or even to MOX. All fuel in a CLASSFUelPLan are associated to a burnup. The reactor updating his cycle time to this burnup(considering the power remain the same in a reactor during all its life.

The usage of the CLASSFuelPlan is quiet simple... On just need to add to an existing CLASSFuelPlan, a time (to perform the fuel switch), a fuel and an associated burnup. One should note that the fuel in the reactor only change at the end of a cycle (finish the current cycle even after a date to change fuel type.)

```
void AddFuel(cSecond time,  EvolutionData* fuel, double BurnUp)
void AddFuel(cSecond time,  PhysicsModels* fuel, double BurnUp)
```

For the one who want to go further, when declaring a reactor (using any type of fuel) (see 6.2.2), a CLASSFuelPlan is created inside the reactor. So it is possible to create a reactor using the normal reactor creator (which include a fuel...) and add after some fuel to the plan using *CLASSFuelPlan* GetFuelPlan()* method of the reactor, then the AddFuel method of CLASSFuelPlan. ie :

```
CLASSLogger *MyLogger = new CLASSLogger("CLASS_OUTPUT.log",1,2);
EvolutionData* STD900 = new EvolutionData(MyLogger, "STD900.dat",false);
EvolutionData* GARANCE = new EvolutionData(MyLogger, "GARANCE.dat",false);
Reactor* MyReactor = new Reactor(MyLogger, STD900, Cooling_UOX, 0.,
    40*3600*24*365.25, 900e6, 106, 33, 0.8);
Myreactor->GetFuelPlan()->AddFuel(2001*3600*24*365.25, GRANACE, 42);
```

If one want to switch between fixed fuel (e.g. PWR UOX) to recycled fuel (e.q. PWR MOX) one need to define a FabricationPlant to the reactor :

```
IM_RK4 *IMRK4 = new IM_RK4(MyLogger);
EQM_LIN_PWR_MOX* EQMLINPWRMOX = new EQM_LIN_PWR_MOX(MyLogger, "EQ_Lin.dat");
EQM_QUAD_PWR_MOX* EQMQUADPWRMOX = new EQM_QUAD_PWR_MOX(MyLogger, "DB.dat");
PhysicsModels* PHYMOD = new PhysicsModels(XSMOX, EQMQUADPWRMOX, IMRK4);
Myreactor->GetFuelPlan()->AddFuel( MOXTime, PHYMOD, 35);
Myreactor->SetFabricationPlant(FP_MOX);
```

6.3 CLASSBackEnd

6.3.1 Storage

6.3.2 Pool

6.4 Fabrication Plant

Chapter 7

Other objects

7.1 ZAI

7.2 IsotopicVector

7.3 Log management : CLASSLogger

7.4 EvolutionData

Chapter 8

Decay data bases

Part IV

Physics Models

Chapter 9

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 9.1: 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 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 burn-up or to satisfied criticality conditions.

10.1 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 :

10.1.1 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.

10.1.1.1 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 10.1):

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

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...

10.1.1.2 Quadratic Model : EQM_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 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 $45GWd/tHM$, 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 α_{238Pu} 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 10.1: 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 7.3) and AlphasFile a string which is the complete path to the file containing the weights (the α parameters)

Available weight file (.dat) :

- @@@ **BAM**

- @@@ **BAM**

- ...

10.1.1.3   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 10.2: 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 7.3) 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 :

```cpp
  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].

10.1.2   PWR-UOX model :

10.1.2.1   Linear Model: EQM_LIN_UOX

@@@BAM

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 10.3: EQM_NAME.hxx

```cpp
#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 10.4: EQM_NAME.cxx

```cpp
#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 !

10.2.1   Compile your equivalence model with your CLASS executable :

@@BAM

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).

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

(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

```cxx
...
#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 7.3) and PathToIdxFile a string containing the path to the .idx file. The .idx file lists all the EvolutionData (see section 7.4) 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 fuel depletion calculation formatted as follow :

Listing 11.2: 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 11.1. 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].

Table 11.1: .dat Key words meaning

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

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.

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 11.3: 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 11.4: 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 11.2). 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 11.4). 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 11.5: XSM_NAME.hxx

```cpp
#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 11.6: XSM_NAME.cxx

```cpp
#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. 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 !

11.2.1 Compile your cross section model with your CLASS executable :

@@BAM

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.

---

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

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 8).

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 7.3).

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

```cpp
#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

```cpp
#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 7.4) 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 !

12.1.2 Compile your cross section model with your CLASS executable :

@@BAM

12.1.3 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