MANUAL

INL/EXT-14-xxxxx
Draft Release
Printed August 2014

RAVEN User Manual

Prepared by
Idaho National Laboratory
Idaho Falls, Idaho 83415

The Idaho National Laboratory is a multiprogram laboratory operated by
Battelle Energy Alliance for the United States Department of Energy
under DOE Idaho Operations Office. Contract DE-AC07-05ID14517.

Issued by the Idaho National Laboratory, operated for the United States Department of Energy by Battelle Energy Alliance.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

INL/EXT-14-xxxxx
Draft Release
Printed August 2014

RAVEN User Manual

The RAVEN team:

Principal Investigator:

Cristian Rabiti, *the Boss*

Developers:

Andrea Alfonsi, *the funny guy*
Joshua Cogliati, *the computer ace*
Diego Mandelli, *born old*
Robert Kinoshita, *he is not Japanese*

[Title]

Contents

Figures

Tables

1 Introduction

RAVEN is a generic software framework to perform parametric and probabilistic analysis based on the response of complex system codes. The initial development was aimed to provide dynamic risk analysis capabilities to the Thermo-Hydraulic code RELAP-7, currently under development at the Idaho National Laboratory (INL). Although the initial goal has been fully accomplished, RAVEN is now a multi-purpose probabilistic and uncertainty quantification platform, capable to agnostically communicate with any system code. This agnosticism includes providing Application Programming Interfaces (APIs). These APIs are used to allow RAVEN to interact with any code as long as all the parameters that need to be perturbed are accessible by inputs files or via python interfaces. RAVEN is capable of investigating the system response, and investigating the input space using Monte Carlo, Grid, or Latin Hyper Cube sampling schemes, but its strength is focused toward system feature discovery, such as limit surfaces, separating regions of the input space leading to system failure, using dynamic supervised learning techniques. The development of RAVEN has started in 2012, when, within the Nuclear Energy Advanced Modeling and Simulation (NEAMS) program, the need to provide a modern risk evaluation framework became stronger. RAVEN principal assignment is to provide the necessary software and algorithms in order to employ the concept developed by the Risk Informed Safety Margin Characterization (RISMC) program. RISMC is one of the pathways defined within the Light Water Reactor Sustainability (LWRS) program. In the RISMC approach, the goal is not just the individuation of the frequency of an event potentially leading to a system failure, but the closeness (or not) to key safety-related events. Hence, the approach is interested in identifying and increasing the safety margins related to those events. A safety margin is a numerical value quantifying the probability that a safety metric (e.g. for an important process such as peak pressure in a pipe) is exceeded under certain conditions. The initial development of RAVEN has been focused on providing dynamic risk assessment capability to RELAP-7, currently under develop-ment at the INL and, likely, future replacement of the RELAP5-3D code. Most the capabilities that have been implemented having RELAP-7 as principal focus are easily deployable for other system codes. For this reason, several side activates are currently ongoing for coupling RAVEN with software such as RELAP5-3D, etc. The aim of this document is the explaination of the input requirements, focalizing on the input structure.

2 Installing and running RAVEN

How To Run!!!

3 Raven Input Structure

The RAVEN code does not have a fixed calculation flow, since all its basic objects can be combined in order to create a user-defined calculation flow. Thus, its input (xml) is organized in different xml blocks, each with a different functionality. The main input blocks are as follows:

- **Simulation**: The simulation block is the one that has inside the entire input, all the following blocks fit inside the simulation block;
- **RunInfo**: Block in which the calculation settings are specified (number of parallel simulations, etc.);
- **Distributions**: Distributions' container;
- **Samplers**: Exploration of the uncertain domain strategy specification;
- **Functions**: External functions container;
- **Models**: Models' specifications (e.g. Codes,ROM,etc.);
- **Steps**: Place where the single basic objects get combained;
- **Datas**: Internal Data object block;
- **Databases**: Databases block;
- **OutStream system**: Visualization and Printing system block.

Each of these blocks are explained in dedicated sections in the following chapters.

4 RunInfo

The $RunInfo$ block is the place where the user specifiy how the calculation needs to be performed. In this input block, several settings can be inputted, in order to define how to drive the calculation and set up, when needed, particular settings for the machine the code needs to run on (queue system if not PBS, etc.). In the following subsections, all the keywords are explained in detail.

4.1 RunInfo: input of calculation flow.

This sub-section contains the information regarding the xml nodes that define the settings of the calculation flow is going to be performed through RAVEN:

- $< WorkingDir >$, ***string, required field.*** in this block the user needs to specify the absolute or relative (with respect to the location where RAVEN is run from) path to a directory that is going to be used to store all the results of the calculations and where RAVEN looks for the files specified in the block $< Files >$. *Default = None*;
- $< Files >$, ***comma separated string, required field.*** these are the paths to the files required by the code, string from the $WorkingDir$;
- $< batchSize >$, ***integer, required field.***. This parameter specifies the number of parallel runs need to be run simultaneously (e.g., the number of driven code instances, e.g. RELAP5-3D, that RAVEN will spoon at the same time). *Default = 1*;
- $< Sequence >$, ***comma separated string, required field.*** ordered list of the step names that RAVEN will run (see Section 9);
- $< NumThreads >$, ***integer, optional field.*** this section can be used to specify the number of threads RAVEN should associate when running the driven software. For example, if RAVEN is driving a code named "FOO", and this code has multi-threading support, in here the user specify how many threads each instance of FOO should use (e.g. FOO –n-threads=$NumThreads$). *Default = 1 (or None when the driven code does not have multi-threading support)*;
- $< NumMPI >$, ***integer, optional field.*** this section can be used to specify the number of MPI cpus RAVEN should associate when running the driven software. For example, if RAVEN is driving a code named "FOO", and this code has MPI

support, in here the user specifies how many mpi cpus each instance of FOO should use (e.g. mpiexec FOO -np $NumMPI$). *Default = 1 (or None when the driven code does not have MPI support)*;

- $< totalNumCoresUsed >$***, integer, optional field.*** global number of cpus RAVEN is going to use for performing the calculation. When the driven code has MPI and/or Multi-threading support and the user decides to input $NumThreads > 1$ and $NumMPI > 1$, the totalNumCoresUsed = NumThreads*NumMPI*batchSize. *Default = 1*;

- $< precommand >$***, string, optional field.*** in here the user can specifies a command that needs to be inserted before the actual command that is used to run the external model (e.g., mpiexec -n 8 $precommand$ ./externalModel.exe (...)). *Default = None*;

- $< postcommand >$***, string, optional field.*** in here the user can specifies a command that needs to be appended after the actual command that is used to run the external model (e.g., mpiexec -n 8 ./externalModel.exe (...) $postcommand$). *Default = None*;

- $< MaxLogFileSize >$***, integer, optional field.*** every time RAVEN drives a code/-software, it creates a logfile of the code screen output. In this block, the user can input the maximum size of log file in bytes. *Defautl = Inf.* NB. This flag is not implemtend yet;

- $< deleteOutExtension >$***, comma separated string, optional field.*** if a run of an external model has not failed delete the outut files with the listed extension (e.g., $< deleteOutExtension > txt, pdf < /deleteOutExtension >$). *Default = None*.

- $< delSucLogFiles >$***, boolean, optional field.*** if a run of an external model has not failed (return code = 0), delete the associated log files. *Default = False*;

4.2 RunInfo: input of queue modes.

In this sub-section all the keyword (xml nodes) for setting the queue system are reported.

- $< mode >$***, string, optional field.*** In this xml block, the user might specify which kind of protocol the parallel enviroment should use. By instance, RAVEN currently supports two pre-defined "modes":

– pbsdsh: this "mode" uses the pbsdsh protocol to distribute the program running; more information regarding this protocol can be found in ref.
  Mode "pbsdsh" automatically "understands" when it needs to generate the "qsub" command, inquiring the "machine eviroment":
  * If RAVEN is executed in the HEAD node of an HPC system, RAVEN generates the "qsub" command, instantiates and submits itself to the queue system;
  * If the user decides to execute RAVEN from an "interactive node" (a certain number of nodes that have been reserved in interactive PBS mode), RAVEN, using the "pbsdsh" system, is going to utilize the reserved resources (cpus and nodes) to distribute the jobs, but, obviously, it's not going to generate the "qsub" command.
– mpi: this "mode" uses mpiexec to distribute the program running; more information regarding this protocol can be found in ref.
  Mode "MPI" can either generate the "qsub" command or can execute in selected nodes.
  In order to make the "mpi" mode generate the "qsub" command, an additional keyword (xml sub-node) needs to be specified:
  * If RAVEN is executed in the HEAD node of an HPC system, the user needs to input a sub-node, $< runQSUB/ >$, right after the specification of the mpi mode (i.e. $< mode > mpi < runQSUB/ >< /mode >$). If the keyword is provided, RAVEN generates the "qsub" command, instantiates and submits itself to the queue system;
  * If the user decides to execute RAVEN from an "interactive node" (a certain number of nodes that have been reserved in interactive PBS mode), RAVEN, using the "mpi" system, is going to utilize the reserved resources (cpus and nodes) to distribute the jobs, but, obviously, it's not going to generate the "qsub" command.

  When the user decides to run in "mpi" mode without making RAVEN generate the "qsub" command, different options are available:
  * If the user decides to run in the local machine (either in local desktop/-workstation or remote machine), no additional keywords are needed (i.e. $< mode > mpi < /mode >$);
  * If the user decided to run in multiple nodes, the nodes' ids have to be specified:
    · the nodes' ids' can be specified in an external text file (nodes' ids separated by blank space). This file needs to be provided in the $mode$

xml node, introducing a sub-node named $nodefile$ (e.g. $< mode > mpi < nodefile > /tmp/nodes < /nodefile >< /mode >$);

- the nodes' ids' can be contained in an enviromental variable (nodes' ids separated by blank space). This variable needs to be provided in the $mode$ xml node, introducing a sub-node named $nodefileenv$ (e.g. $< mode > mpi < nodefileenv > NODEFILE < /nodefileenv >< /mode >$);
- if none of the above options are used, RAVEN is going to try finding the nodes' information in the enviroment variable $PBS\_NODEFILE$.

- $< NumNode >$**, *integer, optional field.*** this xml node is used to specify the number of nodes RAVEN should request when running in High Performance Computing (HPC) systems. *Default = None*;

- $< CustomMode >$**, *xml node, optional field.*** In this xml node, the "advanced" users can implement a newer "mode". Please refer to sub-section 4.3 for advanced users.

- $< quequingSoftware >$**, *string, optional field.*** RAVEN has support for PBS queuing system. If the platform provides a different quequing system, the user can specify its name here (e.g., PBS PROFESSIONAL, etc.). *Default = PBS PROFESSIONAL*;

- $< expectedTime >$***colum separated string, requested field (pbsdsh mode)*** . In this block the user specifies the time the whole calculation is expected to last. The syntax of this node is $hours : minutes : seconds$ (e.g. 40:10:30 =¿ 40 hours, 10 minutes, 30 seconds). After this period of time the HPC system will automatically stop the simulation (even if the simulation is not completed). It is preferable to rationally overstimate the needed time. *Default = None*;

4.3 RunInfo: Advanced Users.

This sub-section addresses some customizations of the running enviroment that are possible in RAVEN. // Firstly, all the keywords reported in the previous sections can be predefined by the user in an auxiliary xml input file. Every time RAVEN gets instantiated (i.e. the code is run), it looks for an optional file, named "default_runinfo.xml" contained in "\home\username\.raven\" directory (i.e. "\home\username\.raven\default_runinfo.xml'). This file (same syntax of the RunInfo block definable in the general input file) will be used

for defining default values for the data inputtable in the RunInfo block.In addition to the keywords defined in the previous sections, in the $< RunInfo >$, an additional keyword can be defined:

- $< DefaultInputFile >$**, *string, optional field.*** In this block the user can change the default xml input file RAVEN is going to look for if none has been provided as command-line argument. *Default* = *"test.xml"*.

As already mentioned, this file is read to define defaul data for the RunInfo block. This means that all the keywords, that lately are read in the actual input file, will be overridden by values in the actual RAVEN input file.
In section refsubsec:runinfoModes, it has been explained how RAVEN can handle the queue and parallel systems. If the currently available "modes" are not suitable for the user's system (workstation, HPC system, etc.), it is possible to define a custom "mode" modifying the $< RunInfo >$ block as follows:

```
<RunInfo>
  ...
  <CustomMode file="newMode.py" class="NewMode">
    aNewMode
  </CustomMode>
  <mode>aNewMode</mode>
  ...
</RunInfo>
```

The python file should override the functions in SimulationMode in Simulation.py. Generally modifySimulation will be overridden to change the precommand or postcommand parts which will be added before and after the executable command. In the following section an example is reported:

```
import Simulation
class NewMode(Simulation.SimulationMode):
  def doOverrideRun(self):
    # If doOverrideRun is true, then use runOverride
    # instead of running the simulation normally.
    # This method should call simulation.run somehow
    return True
```

```
  def runOverride(self):
    # this can completely override the Simulation's run method
    pass

  def modifySimulation(self):
    # modifySimulation is called after the runInfoDict
    # has been setup.
    # This allows the mode to change any parameters
    # that need changing. This typically modifies the
    # precommand and the postcommand that
    # are put infront of the command and after the command.
    pass

  def XMLread(self,xmlNode):
    # XMLread is called with the mode node,
    # and can be used to
    # get extra parameters needed for the simulation mode.
    pass
```

4.4 RunInfo: Examples.

In here we present some examples:

```
<RunInfo>
    <WorkingDir>externalModel</WorkingDir>
    <Files>lorentzAttractor.py</Files>
    <Sequence>MonteCarlo</Sequence>
    <batchSize>100</batchSize>
    <NumThreads>4</NumThreads>
    <mode>mpi</mode>
    <NumMPI>2</NumMPI>
</RunInfo>
```

Specifies the working directory (WorkingDir) where are located the files necessary (Files) to run a series of 100 (batchSize) Monte-Carlo calculations (Sequence). MPI (mode) mode is used along with 4 threads (NumThreads) and 2 mpi process per run (NumMPI).

5 Distributions

Author: Andrea Alfonsi

RAVEN provides support for several probability distributions. Currently, the user can choose among all the most important 1-Dimensional distributions and N-Dimensional ones, either custom or Multi-Variate.
The user needs to specify the probability distributions, that need to be used during the simulation, within the $< Distributions >$ xml block:

```
<Simulation>
  ...
  <Distributions>
    <!-- here all the distributions, that need to be used,
       are listed -->
  </Distributions>
  ...
</Simulation>
```

In the following two sub-sections, the input requirements for all of them are reported.

5.1 1-Dimensional Probability Distributions

In this sub-section, all the 1-D distributions', currently available in RAVEN, are reported. Firstly, all the probability distributions functions in the code can be truncated by the following keywords:

```
<lowerBound>***</lowerBound>
<upperBound>***</upperBound>
```

Obviously, each distribution already defines its validity domain (e.g. Normal distribution, [-inf,+inf]).
RAVEN currently provides support for 12 1-Dimensional distributions. In the following paragraphs, all the input requirements are reported and commented.

5.1.1 Bernoulli Distribution

The **Bernoulli** distribution is a discrete distribution of the outcome of a single trial with only two results, 0 (failure) or 1 (success), with a probability of success p. It is the simplest building block on which other discrete distributions of sequences of independent Bernoulli trials can be based. Basically, it is the binomial distribution (k = 1, p) with only one trial.
The specifications of this distribution must be defined within the xml block $< Bernoulli >$.
This xml-node needs to contain the attribute:

- **name**, *required integer attribute*, Name of this distribution. As for the other objects, this is the name that can be used to refer to this specific entity in other input blocks (xml).

This distribution can be initialized through the following keyword/s:

- $< p >$, float, required parameter, probability of success.

```
----------------------------
Example:
----------------------------
<Distributions>
  ...
  <Bernoulli name='...'>
    <p>***</p>
  </Bernoulli>
  ...
</Distributions>
----------------------------
```

5.1.2 Beta Distribution

The **Beta** distribution is a continuous distribution defined on the interval $[0, 1]$ parametrized by two positive shape parameters, denoted by α and β, that appear as exponents of the random variable and control the shape of the distribution. The distribution domain can be changed,specifying new boundaries, to fit the user needs.
The specifications of this distribution must be defined within the xml block $< Beta >$.
This xml-node needs to contain the attribute:

- **name**, *required string attribute*, user-defined name of this distribution. N.B. As for the other objects, this is the name that can be used to refer to this specific entity from other input blocks (xml).

This distribution can be initialized through the following keyword/s:

- $< alpha >$, float, required parameter, first shape parameter;
- $< beta >$, float, required parameter, second shape parameter;
- $< low >$, float, optional parameter, lower domain boundary;
- $< high >$, float, required parameter, upper domain boundary.

```
----------------------------
Example:
----------------------------
<Distributions>
  ...
  <Beta name='...'>
     <low>***</low>
     <high>***</high>
     <alpha>***</alpha>
     <beta>***</beta>
  </Beta>
  ...
</Distributions>
----------------------------
```

5.1.3 Binomial Distribution

The **Binomial** distribution is the discrete probability distribution of the number of successes in a sequence of *n* independent yes/no experiments, each of which yields success with probability *p*.
The specifications of this distribution must be defined within the xml block $< Binomial >$.
This xml-node needs to contain the attribute:

- **name**, *required string attribute*, user-defined name of this distribution. N.B. As for the other objects, this is the name that can be used to refer to this specific entity from other input blocks (xml).

This distribution can be initialized through the following keyword/s:

- $< p >$, float, required parameter, probability of success;
- $< n >$, integer, required parameter, number of experiment.

```
----------------------------
Example:
----------------------------
<Distributions>
  ...
  <Binomial name='...'>
    <n>***</n>
    <p>***</p>
  </Binomial>
  ...
</Distributions>
----------------------------
```

5.1.4 Exponential Distribution

The **Exponential** distribution is a continuous distribution that can be used to model the time between independent events that happen at a constant average time.
The specifications of this distribution must be defined within the xml block $< Exponential >$.
This xml-node needs to contain the attribute:

- **name**, *required string attribute*, user-defined name of this distribution. N.B. As for the other objects, this is the name that can be used to refer to this specific entity from other input blocks (xml).

This distribution can be initialized through the following keyword/s:

- $< lambda >$, float, required parameter, rate parameter.

```
----------------------------
Example:
----------------------------
```

```
<Distributions>
  . . .
  <Exponential name='...'>
    <lambda>***</lambda>
  </Exponential>
  . . .
</Distributions>
-------------------------
```

The **Gamma** distribution is a continuous distribution. It has three parameters, *low* for the lowest value, *alpha* is the shape parameter, and *beta* which is 1/scale or the inverse scale parameter.

```
<Gamma name='...'>
<low>***</low>
<alpha>***</alpha>
<beta>***</beta>
```

The **Logistic** distribution is a continuous distribution similar to the normal distribution with a CDF that is an instance of a logistic function. It has two parameters, *location* with is the most common value and the center, and *scale* which determines the shape.

```
<Logistic name='...'>
<location>***</location>
<scale>***</scale>
```

The **LogNormal** distribution is a continuous distribution with the logarithm of the random variable being normally distributed. It has two parameters, *mean* which is the expected value, and *sigma* which is the standard deviation.

```
<LogNormal name='...'>
<mean>***</mean>
<sigma>***</sigma>
```

The **Normal** distribution (or Gaussian) distribution is a continuous distribution which because of the central limit theorem, the mean of many distributions approximates a normal distribution. It has two parameters, *mean* which is the middle value, and *sigma* which is the standard deviation.

```
<Normal name='...'>
```

<mean>***</mean>
<sigma>***</sigma>

The **Poisson** distribution is a discrete distribution that expresses the probability of the number of events occurring in a fixed period of time. It has one parameter, *mu* the mean rate of events/time.

<Poisson name='...'>
<mu>***</mu>

The **Triangular** distribution is a continuous distribution that has a triangular shape for the PDF. The peak falls at the *apex* and the values run from *min* to *max*.

<Triangular name='...'>
<apex>***</apex>
<min>***</min>
<max>***</max>

The **Uniform** distribution is a continuous distribution with a rectangular shaped PDF. It goes from *low* to *high*.

<Uniform name='...'>
<low>***</low>
<hi>***</hi>

The **Weibull** distribution is a continuous distribution that can be used is failure analysis. It takes two parameters, *k* or the shape parameter, and *lambda* or the scale parameter.

<Weibull name='...'>
<lambda>***</lambda>
<k>***</k>

6 Samplers

Samplers own the sampling strategy (Type) and they generate the input values using the associate distribution. They do not have distributions inside.

There are different kind of samplers:

- **MonteCarlo**
- **Grid**
- **LHS**
- **Adaptive**
- **DynamicEventTree**
- **AdaptiveDynamicEventTree**

For MonteCarlo method:

- **MonteCarlo**
  - name: name of sampling method used
  - limit: number of samplings the MonteCarlo method will use
  - initial seed: (optional) initial number of the iterations
- variable
  - name: name of the variable that the code will do the sampling of
- distribution:
  - what kind of distribution the variable follows (it is chosen from the distribution block)

Example:

```
<MonteCarlo name='MC_Sampler' limit='1000'>
 <variable name='pressure'>
  <distribution>***</distribution>
 </variable>
</MonteCarlo>
```

In the example the code is sampling 1000 times (limit) the variable "pressure" with a distribution taken from the distribution block.

For the Grid method:

- **Grid**
  - name: name of the sampling method used
  - initial seed (optional)
- variable
  - name: name of the variable that the code will do the sampling of
- distribution: what kind of distribution the variable follows (it is chosen from the distribution block)
- grid
  - type
    - value
    - CDF
  - construction
    - equal
      - the size of the step, given in the input node, is the same for all steps
      - requires lowerbound or upperbound
      - requires steps
    - custom
      - no lowerbound or upperbound
      - no number of steps
      - in the input node it is necessary to input each step of the grid separated by a space

  - lowerbound or upperbound
  - steps
  - input node

Example:

```
<Grid name='Grid_Sampler'>
 <variable name='pressure'>
  <distribution>***</distribution>
  <grid type='value' construction='equal' steps='100'
     lowerBound='1.0'>0.2</grid>
 </variable>
</Grid>
```

In the example the code is sampling the variable "pressure" with a distribution "***" which was chosen by the distribution block. The grid ranges from 1.0 (lowerbound) to 21.0 (100 steps of equal size of 0.2).
For the LHS method:

- **LHS**
  - name: name of the sampling method used
  - initial seed (optional)
- variable
  - name: name of the variable that the code will do the sampling of
- distribution: what kind of distribution the variable follows (it is chosen from the distribution block)
- grid
  - type
    - value
    - CDF
  - construction
    - equal

        - the size of the step, given in the input node, is the same for all steps
        - requires lowerbound or upperbound
        - requires steps
      * custom
        - no lowerbound or upperbound
        - no number of steps
        - in the input node it is necessary to input each step of the grid separated by a space
    - lowerbound or upperbound
    - steps
    - input node

Example:

```
<LHS name='***' initial_seed='***'>
 <variable name='***'>
  <distribution>***</distribution>
  <grid type='***' construction='***' steps='***'
      lowerBound='***'>*****</grid>
 </variable>
</LHS>
```

The input structure is the same as the **Grid** input structure

For the Adaptive method:

- **Adaptive**
  - name
  - initial seed (optional)
- Convergence
  - limit: 'Integer'
  - persistence: 'Integer'
  - weight='probability' or 'None':

- subGridTol='None' or 'Float' :This is the tolerance used to construct the testing sub grid
- forcelteration='True' or 'False': this flag control if at least a self.limit number of iteration should be done

- variable
- distribution

```
<Adaptive name='***' initial_seed='***'>
 <Convergence limit='***' persistence='***' weight='***' subGridTol='***'
     forcelteration='***'>***</Convergence>
  <variable name='***'>
   <distribution>***</distribution>
  </variable>
</Adaptive>
```

For DynamicEventTree: ***

For AdaptiveDynamicEventTree: ***

7 Functions

This module contains interfaces to import external functions

- **External**
  - name=name of the function
  - file=file name where the function is
- variable
  - type=numpy.float64
  - variable inside the function that has been defined

```
<Functions>
 <External name='***' file='***'>
  <variable type='***'>***</variable>
 </External>
</Functions>
```

8 Models

The xml section ”models” contains the information regarding the code employed in the analysis (e.g., RAVEN/RELAP-7, RELAP-5 or an external model). A model is something that given an input will return an output reproducing some physical model it could be as complex as a stand alone code, a reduced order model trained somehow or something externally build and imported by the user. The available models are:

Dummy: it is a dummy model that just return the effect of the sampler. The values reported as input in the output are the output of the sampler and the output is the counter of the performed sampling

ROM: ROM stands for Reduced Order Model. All the models here, first learn than predict the outcome

ExternalModel: this model allows to interface with an external python module

Code: generic class that imports an external code into the framework

Projector: generic data manipulator

PostProcessor: an Action System. All the models here, take an input and perform an action

8.1 Dummy

Description

Summary

Example

8.2 ROM

Description

Summary

- name: name of the ROM model
- subType: 'SciKitLearn'. Imports the libraries from scikitlearn
- Features: input the variables set in the Samplers block separated by a comma
- SKLtype: input a model from http://scikit-learn.org/stable/modules/classes.html (LinearRegression in the example)
- Target: Input a set of variables inside the output space which the ROM has to be developed for.
- Parameters: From the SKLtype class are defined all the parameters required ($fit\_intercept$ and $normalize$ in the exemple)

Example

```
<Models>
 <ROM name='***' subType='***'>
  <Features>***,***</Features>
  <SKLtype>linear_model|LinearRegression</SKLtype>
  <Target>***</Target>
  <fit_intercept>***</fit_intercept>
  <normalize>***</normalize>
 </ROM>
```

8.3 External Model

Description

Summary

Example As an example we use the external model shown in lorentzAttractor.py which, given the 3-dimensional initial coordinates (x0, y0, z0), calculate the trajectory of a Lorentz attractor in the time interval $[0.0, 0.03]$ seconds. We want to perform sampling of the 3-dimensional initial conditions of the attractor using classical Monte-Carlo sampling. The user is required to specify:

- the initialize function: def initialize(self,runInfoDict,inputFiles)

- the function which create a new input: def createNewInput(self,myInput,samplerType,**Kwargs)
- the function which perform the actual calculation: def run(self,Input)

```
def initialize(self,runInfoDict,inputFiles):
  self.SampledVars = None
  self.sigma = 10.0
  self.rho   = 28.0
  self.beta  = 8.0/3.0
  return

def createNewInput(self,myInput,samplerType,**Kwargs):
  return Kwargs['SampledVars']

def run(self,Input):
   ...
```

```
<Models>
    <ExternalModel name='PythonModule' subType=''
        ModuleToLoad='externalModel/lorentzAttractor'>
        <variable type='float'>sigma</variable>
        <variable type='float'>rho</variable>
        <variable type='float'>beta</variable>
        <variable type='numpy.ndarray'>x</variable>
        <variable type='numpy.ndarray'>y</variable>
        <variable type='numpy.ndarray'>z</variable>
        <variable type='numpy.ndarray'>time</variable>
        <variable type='float'>x0</variable>
        <variable type='float'>y0</variable>
        <variable type='float'>z0</variable>
    </ExternalModel>
</Models>
```

8.4 Code

Description: This is the generic class that import an external code into the framework

Summary

Example

8.5 Projector

Description

Summary

Example

8.6 PostProcessor

Description

List variable, Input Data, Keyword sul tipo analisi statistica!!

Summary

Example

9 Steps

- SingleRun: This is the step that will perform just one evaluation
- MultiRun: This class implements one step of the simulation pattern where several runs are, needed without being adaptive.
  - * name = name of the step (sequence) defined in the RunInfo block, under the Sequence card
  - pauseAtEnd= if True the code will not go to next step until plots are closed manually by the user
  - Sampler:
    * class=Samplers
    * type: the type of sampler used in the Samplers block
    * name of the sampler
  - Model:
    * class= (Models)
    * type=(dummy, ROM, External Model, Code, Projector, PostProcessor)
    * name of the model
  - Input:
    * class=(Data e Files)
    * type:
      · if class = files —-> none
      · if class = Data —-> timepoint, timepointset, historie, histories
  - Output:
    * class: they are the output destinations
      · Datas
      · OutStreamManager
    * type:
      · if class=Datas —->(timepoint, timepointset, historie, histories)
      · if class=OutstreamManager —> type:print, plot
    * name of the output

<MultiRun name='***' pauseAtEnd='***'>
<Sampler class='Samplers' type='***'>***</Sampler>
<Input class='***' type='***'>***</Input>
<Model class='Models' type='***'>***</Model>
<Output class='Datas'*** type='***'</Output>
<Output class='OutStreamManager' type='***'>***</Output>
</MultiRun>

- Adaptive: this class implement one step of the simulation pattern where several runs are needed in an adaptive scheme
  - Sampler: class=samplers type=Adaptive¿???¡
  - Model: class=Models type=(dummy, ROM, External Model, Code, Projector, PostProcessor)
  - Function: it takes in a datas and generate the value of the goal functions, it gives the criteria for which i represent the limit surface (class=Functions)
  - Input:
  - TargetEvaluation: is the output datas that is used for the evaluation of the goal function, it represents the sampling points. It has to be declared among the outputs.
  - SolutionExport: if declared it is used to export the location of the goal functions = 0, it exports the limit surface
  - ROM: is boolean, it selects values of input (through sampling) to find the limit surface through the values that were given by the function.
  - Output:

<Adaptive name='***' pauseAtEnd='***'>
<Input class = 'Datas' type = 'TimePointSet'>***</Input>
<Sampler class = 'Samplers' type = 'Adaptive'>***</Sampler>
<TargetEvaluation class = 'Datas' type = '***'>***</TargetEvaluation>

- IODataBase: This step type is used only to extract or push information from/into a DataBase. If in the Databases block the Database is created (no directory or filename) the then the Databse will be an output of this block, otherwise, if it is uploaded then in this block it will be a Input

  - Input: class=(Datas or Databases) type=(if Databases, HDF5, if Datas:timepoint, timepointset, historie, histories)
  - Output: class=(Datas or Databases) type=(if Databases, HDF5, if Datas:timepoint, timepointset, historie, histories)

- RomTrainer: This step type is used only to train a ROM.

  <**RomTrainer name**='***'>
   <**Input class**='Datas' **type**='TimePointSet'>***</**Input**>

- PostProcess:

- OutStreamStep:

10 Datas

- TimePoint: A couple of points inside the input and output spaces.
- TimePointSet: A set of couples of points inside the input and output spaces.
- History: A set of couple of points inside the input space and a time dependent array inside the output space. Because time is not a continuous variable inside the RAVEN environment, each array is associated with an array of time points. The input space points and the output space array are correlated by model. (see section *models*)
- Histories: A set of couple of points inside the input space and a set of array inside the output space. The input space points and the output space array are correlated by model. (see section *models*)

Input: sampling variables inside relap or any TH code

Output: variables inside the model output (OutPlaceHolder is a keyword for a something that doesn't have an output)

```
<Datas>
 <TimePointSet name='***'>
  <Input>***,***,***</Input>
  <Output>***,***</Output>
 </TimePointSet>
</Datas>
```

11 Databases

- name: database name
- directory:
  - if it is not specified a directory, such directory will be created. It will be named DataBaseStorage in the working directory with inside the .h5 file containing the database.
  - if input directory name is given the code will look for DataBaseStorage folder and pick the file from filename from inside such directory
- filename: input file to be read from, if not there such file will be created

Example:

```
<DataBases>
    <HDF5 name="***" directory='***' filename='***'/>
</DataBases>
```

12 OutStream system

Author: Andrea Alfonsi

The PRA and UQ framework provides the capabilities to visualize and dump out the data that are generated, imported (from a system code) and post-processed during the analysis. These capabilities are contained in the "OutStream" system. Actually, two different OutStream types are available:

- Print, module that lets the user dump the data contained in the internal objects
- Plot, module, based on MatPlotLib [**?**], aimed to provide advanced plotting capabilities

Both the types listed above only accept as "input" a *Data* object type. This choice has been taken since the "*Datas*" system (see section **??**) has the main advantages, among the others, of ensuring a standardized approach for exchanging the data/meta-data among the different framework entities. Every module can project its outcomes into a *Data* object. This provides, to the user, the capability to visualize/dump all the modules' results. As already mentioned [put reference to the xml input section], the RAVEN framework input is based on the **E**xtensible **M**arkup **L**anguage (**XML**) format. Thus, in order to activate the "*OutStream*" system, the input needs to contain a block identified by the "$< OutStreamManager >$" tag (as shown below).

```
------------------------------------------------------
<OutStreamManager>
   <!-- "OutStream" objects that need to be created-->
</OutStreamManager>
------------------------------------------------------
```

In the "OutStreamManager" block an unlimited number of "Plot" and "Print" sub-blocks can be inputted. The input specifications and the main capabilities for both types are reported in the following sections.

12.1 Printing system

The Printing system has been created in order to let the user dump the data, contained in the internal data objects (see [reference to Data(s) section]), out at anytime during the calculation. Currently, the only available output is a **C**omma **S**eparated **V**alue (**CSV**) file.

In the near future, an XML formatted file option will be available. This will facilitate the exchanging of results and provide the possibility to dump the solution of an analysis and "restart" another one constructing a *Data* from scratch. The XML code, that is reported below, shows different ways to request a *Print* OutStream. The user needs to provide a name for each sub-block (XML attribute). These names are then used in the *Steps'* blocks in order to activate the Printing options at anytime. As shown in the examples below, every *Print* block must contain, at least, the two required tags:

- $< type >$, the output file type (csv or xml). *Note, only* ***csv*** *is currently available*
- $< source >$, the *Data* name (one of the *Data* defined in the "*Datas*" block)

If only these two tags are provided (as in the "first-example" below), the output file will be filled with the whole content of the "d-name" *Data*.

```
------------------------------------------------------------
<OutStreamManager>
  <Print name='first_example'>
    <type>csv</type>
    <source>d-name</source>
  </Print>
  <Print name='second-example'>
    <type>csv</type>
    <source>d-name</source>
    <variables>Output</variables>
  </Print>
  <Print name='third-example'>
    <type>csv</type>
    <source>d-name</source>
    <variables>Input</variables>
  </Print>
  <Print name='forth-example'>
    <type>csv</type>
    <source>d-name</source>
    <variables>Input|var-name-in,Output|var-name-out</variables>
  </Print>
</OutStreamManager>
------------------------------------------------------------
```

If just few parts of the $< source >$ are important for a particular analysis, the additional XML tag $< variables >$ can be provided. In this block, the variables that need to be

dumped must be inputted, in a comma separated format. The available options, for the $< variables >$ sub-block, are listed below:

- **Output**, the output space will be dumped out (see "second-example")
- **Input**, the input space will be dumped out (see "third-example")
- **Input—var-name-in/Output—var-name-out**, only the particular variables "var-name-in" and "var-name-out" will be reported in the output file (see "forth-example")

Note that all the XML tags are case-sensitive but not their content.

12.2 Plotting system

The Plotting system provides all the capabilities to visualize the analysis outcomes, in real-time or at the post-processing stage. The system is based on the Python library MatPlotLib [**?**]. MatPlotLib is a 2D/3D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. This external tool has been wrapped in the RAVEN framework, and is usable by the user. Since it was unfeasible to support, in the source code, all the interfaces for all the available plot types, the RAVEN Plotting system directly provide a formatted input structure for 11 different plot types (2D/3D). The user may request a plot not present among the supported ones, since the RAVEN Plotting system has the capability to construct on the fly the interface for a Plot, based on XML instructions. This capability will be discussed in the sub-section 12.2.2.

12.2.1 Plot input structure

In order to create a plot, the user needs to add, within the $< OutStreamManager >$ block, a $< Plot >$ sub-block. As for the *Print* OutStream, the user needs to specify a name as attribute of the plot. This name will then be used to request the plot in the *Steps'* block. In addition, the Plot block may need the following attributes:

- **dim**, *required integer attribute*, define the dimensionality of the plot: 2 (2D) or 3 (3D)
- **interactive**, *optional bool attribute (default=False)'*, specify if the Plot needs to be interactively created (real-time screen visualization)

- **overwrite**, *optional bool attribute (default=False)'*, if the plot needs to be dumped into picture file/s, does the code need to overwrite them every time a new plot (with the same name) is requested?

As shown, in the XML input example below, the body of the Plot XML input contains two main sub-nodes:

- $< actions >$, where general control options for the figure layout are defined (see [])
- $< plot\_settings >$, where the actual plot options are provided

These two main sub-block are discussed in the following paragraphs.

12.2.1.1 "Actions" input block

The input in the $< actions >$ sub-node is common to all the Plot types, since, in it, the user specifies all the controls that need to be applied to the figure style. This block must be unique in the definition of the $< Plot >$ main block. In the following list, all the predefined "actions" are reported:

- $< how >$, comma separated list of output types:
  - *screen*, show the figure on the screen in interactive mode
  - *pdf*, save the figure as a Portable Document Format file (PDF)
  - *png*, save the figure as a Portable Network Graphics file (PNG)
  - *eps*, save the figure as a Encapsulated Postscript file (EPS)
  - *pgf*, save the figure as a LaTeX PGF Figure file (PGF)
  - *ps*, save the figure as a Postscript file (PS)
  - *gif*, save the figure as a Graphics Interchange Format (GIF)
  - *svg*, save the figure as a Scalable Vector Graphics file (SVG)
  - *jpeg*, save the figure as a jpeg file (JPEG)
  - *raw*, save the figure as a Raw RGBA bitmap file (RAW)
  - *bmp*, save the figure as a Windows bitmap file (BMP)
  - *tiff*, save the figure as a Tagged Image Format file (TIFF)
  - *svgz*, save the figure as a Scalable Vector Graphics file (SVGZ)

- $< title >$, as the name suggests , within this block the user can specify the title of the figure. In the body, few other keywords (required and not) are present:
  - $< text >$, string type, title of the figure
  - $< kwargs >$, within this block the user can specify optional parameters with the following format:

```
-------------------------
<kwargs>
  <param1>value1</param1>
  <param2>value2</param2>
</kwargs>
-------------------------
```

    The kwargs block is able to convert whatever string into a python type (for example $< param1 > '1stKeyword' : 45 < /param1 >$ will be converted into a dictionary, $< param2 > [56, 67] < /param2 >$ into a list, etc.). For reference regarding the available kwargs, see "matplotlib.pyplot.title" method in [**?**].

- $< label\_format >$, within this block the default scale formating can be modified. In the body, few keywords can be specified (all optional):
  - $< style >$, string, the style of the number notation, 'sci' or 'scientific' for scientific, 'plain' for plain notation. Default = scientific
  - $< scilimits >$, tuple, (m, n), pair of integers; if style is 'sci', scientific notation will be used for numbers outside the range 10'm':sup: to 10'n':sup:. Use (0,0) to include all numbers. NB. The value for this keyword, needs to be inputted between brackets [for example, (5,6)]. Default = (0,0)
  - $< useOffset >$, bool or double, if True, the offset will be calculated as needed; if False, no offset will be used; if a numeric offset is specified, it will be used. Default = False
  - $< axis >$, string, the axis where to apply the defined format, 'x','y' or 'both'. Default = 'both'. NB. If this action will be used in a 3-D plot, the user can input 'z' as well and 'both' will apply this format to all three axis.
- $< figure\_properties >$, within this block the user specifies how to customize the figure style/quality. Thus, through this "action" the user has got full control on the quality of the figure, its dimensions, etc. This control is performed by the following keywords:

  - $< figsize >$, tuple (optional), (width, hight), in inches
  - $< dpi >$, integer, dots per inch
  - $< facecolor >$, string, set the figure background color (please refer to "matplotlib.figure.Figure" in [**?**] for a list of all the colors available)
  - $< edgecolor >$, string, the figure edge background color (please refer to "matplotlib.figure.Figure" in [**?**] for a list of all the colors available)
  - $< linewidth >$, self explainable keyword
  - $< frameon >$, bool, if False, suppress drawing the figure frame

- $< range >$, the range "action" allows to specify the ranges of all the axis. All the keywords in the body of this block are optional:
  - $< ymin >$, double (optional), lower boundary for y axis
  - $< ymax >$, double (optional), upper boundary for y axis
  - $< xmin >$, double (optional), lower boundary for x axis
  - $< xmax >$, double (optional), upper boundary for x axis
  - $< zmin >$, double (optional), lower boundary for z axis. NB. Obviously, this keyword is effective in 3-D plots only
  - $< zmax >$, double (optional), upper boundary for z axis. NB. Obviously, this keyword is effective in 3-D plots only

- $< camera >$, the camera item is available in 3-D plots only. Through this "action", it is possible to orientate the plot as wished. The controls are:
  - $< elevation >$, double (optional), stores the elevation angle in the z plane
  - $< azimuth >$, double (optional), stores the azimuth angle in the x,y plane

- $< scale >$, the scale block allows the specification of the axis scales:
  - $< xscale >$, string (optional), scale of the x axis. Three options are available: "linear","log","symlog". Default = linear
  - $< yscale >$, string (optional), scale of the y axis. Three options are available: "linear","log","symlog". Default = linear
  - $< zscale >$, string (optional), scale of the z axis. Three options are available: "linear","log","symlog". Default = linear. NB. Obviously, this keyword is effective in 3-D plots only

- $< add\_text >$, same as title
- $< autoscale >$, the autoscale block is a convenience method for simple axis view autoscaling. It turns autoscaling on or off, and then, if autoscaling for either axis is on, it performs the autoscaling on the specified axis or axes. The following keywords are available:
  - $< enable >$, bool (optional), True turns autoscaling on, False turns it off. None leaves the autoscaling state unchanged. Default = True
  - $< axis >$, string (optional), string, the axis where to apply the defined format, 'x','y' or 'both'. Default = 'both'. NB. If this action will be used in a 3-D plot, the user can input 'z' as well and 'both' will apply this format to all three axis.
  - $< tight >$, bool (optional), if True, set view limits to data limits; if False, let the locator and margins expand the view limits; if None, use tight scaling if the only artist is an image, otherwise treat tight as False.
- $< horizontal\_line >$, this "action" provides the ability to draw a horizontal line in the current figure. This capability might be useful, for example, if the user wants to highlight a trigger, function of a variable. The following keywords are settable:
  - $< y >$, double (optional), y coordinate. Default = 0
  - $< xmin >$, double (optional), starting coordinate on the x axis. Default = 0
  - $< xmax >$, double (optional), ending coordinate on the x axis. Default = 1
  - $< kwargs >$, within this block the user can specify optional parameters with the following format:

```
--------------------------
<kwargs>
  <param1>value1</param1>
  <param2>value2</param2>
</kwargs>
--------------------------
```

    The kwargs block is able to convert whatever string into a python type (for example $< param1 > '1stKeyword' : 45 < /param1 >$ will be converted into a dictionary, $< param2 > [56, 67] < /param2 >$ into a list, etc.). For reference regarding the available kwargs, see "matplotlib.pyplot.axhline" method in [**?**].

NB. This capability is not available for 3-D plots.

- $< vertical\_line >$, similarly to the "horizontal_line" action, this block provides the ability to draw a vertical line in the current figure. This capability might be useful, for example, if the user wants to highlight a trigger, function of a variable. The following keywords are settable:
  - $< x >$, double (optional), x coordinate. Default = 0
  - $< ymin >$, double (optional), starting coordinate on the y axis. Default = 0
  - $< ymax >$, double (optional), ending coordinate on the y axis. Default = 1
  - $< kwargs >$, within this block the user can specify optional parameters with the following format:

    ```
    -------------------------
    <kwargs>
      <param1>value1</param1>
      <param2>value2</param2>
    </kwargs>
    -------------------------
    ```

    The kwargs block is able to convert whatever string into a python type (for example $< param1 > '1stKeyword' : 45 < /param1 >$ will be converted into a dictionary, $< param2 > [56, 67] < /param2 >$ into a list, etc.). For reference regarding the available kwargs, see "matplotlib.pyplot.axvline" method in [**?**].

  NB. This capability is not available for 3-D plots.
- $< horizontal\_rectangle >$, this "action" provides the possibility to draw, in the current figure, a horizontally orientated rectangle . This capability might be useful, for example, if the user wants to highlight a zone in the plot. The following keywords are settable:
  - $< ymin >$, double (required), starting coordinate on the y axis
  - $< ymax >$, double (required), ending coordinate on the y axis
  - $< xmin >$, double (optional), starting coordinate on the x axis. Default = 0
  - $< xmax >$, double (optional), ending coordinate on the x axis. Default = 1
  - $< kwargs >$, within this block the user can specify optional parameters with the following format:

```
-------------------------
 <kwargs>
   <param1>value1</param1>
   <param2>value2</param2>
 </kwargs>
-------------------------
```

The kwargs block is able to convert whatever string into a python type (for example $< param1 > '1stKeyword' : 45 < /param1 >$ will be converted into a dictionary, $< param2 > [56, 67] < /param2 >$ into a list, etc.). For reference regarding the available kwargs, see "matplotlib.pyplot.axhspan" method in [**?**].

NB. This capability is not available for 3-D plots.

- $< vertical\_rectangle >$, this "action" provides the possibility to draw, in the current figure, a vertically orientated rectangle . This capability might be useful, for example, if the user wants to highlight a zone in the plot. The following keywords are settable:
  - $< xmin >$, double (required), starting coordinate on the x axis
  - $< xmax >$, double (required), ending coordinate on the x axis
  - $< ymin >$, double (optional), starting coordinate on the y axis. Default = 0
  - $< ymax >$, double (optional), ending coordinate on the y axis. Default = 1
  - $< kwargs >$, within this block the user can specify optional parameters with the following format:

```
-------------------------
 <kwargs>
   <param1>value1</param1>
   <param2>value2</param2>
 </kwargs>
-------------------------
```

The kwargs block is able to convert whatever string into a python type (for example $< param1 > '1stKeyword' : 45 < /param1 >$ will be converted into a dictionary, $< param2 > [56, 67] < /param2 >$ into a list, etc.). For reference regarding the available kwargs, see "matplotlib.pyplot.axvspan" method in [**?**].

NB. This capability is not available for 3-D plots.

- $< axes\_box >$, this keyword controls the axes' box. No body and its value can be 'on' or 'off'.
- $< axis\_properties >$, this block is used to set axis properties. There are not fixed keywords. If only a single property needs to be set, it can be specified as body of this block, otherwise a dictionary-like string needs to be provided. For reference regarding the available keys, refer to "matplotlib.pyplot.axis" method in [**?**].
- $< grid >$, this block is used to define a grid that needs to be added in the plot. The following keywords can be inputted:
  - $< b >$, double (required), starting coordinate on the x axis
  - $< which >$, double (required), ending coordinate on the x axis
  - $< axis >$, double (optional), starting coordinate on the y axis. Default = 0
  - $< kwargs >$, within this block the user can specify optional parameters with the following format:

```
-------------------------
<kwargs>
  <param1>value1</param1>
  <param2>value2</param2>
</kwargs>
-------------------------
```

    The kwargs block is able to convert whatever string into a python type (for example $< param1 > '1stKeyword' : 45 < /param1 >$ will be converted into a dictionary, $< param2 > [56, 67] < /param2 >$ into a list, etc.). For reference regarding the available kwargs, see "matplotlib.pyplot.grid" method in [**?**].

12.2.1.2 "plot_settings" input block

The sub-block identified by the keyword $< plot\_settings >$ is used to define the plot characteristics.Within this sub-section at least a $< plot >$ block must be present. the $< plot >$ sub-section may not be unique within the $< plot\_settings >$ definition; the number of $< plot >$ sub-blocks is equal to the number of plots that need to be placed in the same figure. For example, in the following XML cut, a "line" and a "scatter" type are combined in the same figure.

```
-----------------------------------------------------------
<OutStreamManager>
  <Plot name='example2PlotsCombined' dim='2'>
    <actions>
      <!-- Actions -->
    </actions>
    <plot\_settings>
      <plot>
       <type>line</type>
       <x>d-type|Output|x1</x>
       <y>d-type|Output|y1</y>
      </plot>
      <plot>
       <type>scatter</type>
       <x>d-type|Output|x2</x>
       <y>d-type|Output|y2</y>
      </plot>
      <xlabel>label X</xlabel>
      <ylabel>label Y</ylabel>
    </plot\_settings>
  </Plot>
</OutStreamManager>
-----------------------------------------------------------
```

As already mentioned, within the $< plot\_settings >$ block, at least a $< plot >$ sub-block needs to be inputted. Independently from the plot type, some keywords are mandatory:

- $< type >$, string, the plot type (for example, line, scatter, wireframe, etc.)
- $< x >$, string, the parameter needs to be considered as x coordinate
- $< y >$, string, the parameter needs to be considered as y coordinate
- $< z >$, string (required in 3-D plots only), the parameter needs to be considered as z coordinate

In addition other plot-dependent keywords, reported in section 12.2.1.3, can be provided. Under the $< plot\_settings >$ block other keywords, common to all the plots the user decided to combine in the figure, can be inputted, such as:

- $< xlabel >$, string, x axis label
- $< ylabel >$, string, y axis laber
- $< zlabel >$, string (available in 3-D plots only), z axis label

12.2.1.3 Predefined Plotting System: 2D/3D

sgagagga ——————————————————————————

```
<OutStreamManager>
  <Plot name='2DHistoryPlot' dim='2' interactive='False' overwrite='False'>
    <actions>
      <how>pdf,png,eps</how>
      <title>
        <text> </text>
      </title>
    </actions>
    <plot_settings>
      <plot>
        <type>line</type>
        <x>stories|Output|time</x>
        <y>stories|Output|pipe1_Hw</y>
        <kwargs>
          <color>green</color>
          <label>pipe1-Hw</label>
        </kwargs>
      </plot>
      <plot>
        <type>line</type>
        <x>stories|Output|time</x>
        <y>stories|Output|pipe1_aw</y>
        <kwargs>
          <color>blue</color>
          <label>pipe1-aw</label>
        </kwargs>
      </plot>
      <xlabel>time [s]</xlabel>
```

```
      <ylabel>evolution</ylabel>
    </plot_settings>
  </Plot>
</OutStreamManager>
```

12.2.2 Interpreted Plotting instruction

13 Existing Interfaces

13.1 RELAP5 Interface

13.1.1 Files

In the $< Files >$ section, as specified before, there should be specified all the files needed for the code to run. In the case of RELAP5 most of the times the files needed are the following:

- RELAP5 Input file
- Table file or files that RELAP needs to run
- relapdata.py
- relap5run.py

Example:

<**Files**>`X10.i,tpfh2o,relapdata.py,RELAP5run.py`</**Files**>

It is a good practice to put inside the working directory all of these files and also:

- the RAVEN input file
- the executable file for RELAP5
- the license for the executable for RELAP5

13.1.2 Sequence

In the $< Sequence >$ section there should be specified the names of the steps declared in the $< Steps >$ block. So if we called the first multirun: "Grid_Sampler" and the second multirun: "MC_Sampler" in the sequence section we should see this:

<**Sequence**>`Grid_Sampler,MC_Sampler`</**Sequence**>

13.1.3 batchSize and mode

For the $< batchSize >$ and $< mode >$ sections please take a look at the RunInfo block in the previous chapters.

13.1.4 RunInfo

After all of these blocks are filled out here is a standard example of what a RunInfo block can look like:

```
<RunInfo>
    <WorkingDir>~/working_dir</WorkingDir>
    <Files>inputfilerelap.i,tpfh2o,relapdata.py,RELAP5run.py</Files>
    <Sequence>Grid_Sampler,MC_Sampler</Sequence>
    <batchSize>1</batchSize>
        <mode>pbsdsh</mode>
    <expectedTime>1:00:00</expectedTime>
    <ParallelProcNumb>1</ParallelProcNumb>
</RunInfo>
```

13.1.5 Models

For the $< Models >$ block here is a standard example of what can be used to use RELAP5 as the external model:

```
<Models>
    <Code name='MyRELAP' subType='Relap5'><executable>python
       RELAP5run.py</executable></Code>
</Models>
```

13.1.6 Distributions

In the $< Distribution >$ block are defined the distributions that are going to be used for the sampling of the variables defined in the $< Samplers >$ block. For all the possibile

distributions and all their possible inputs please see the chapter about Distributions. Here we give a general example of three different distributions, with the names:

```
<Distributions debug='True'>
    <Triangular name='BPfailtime'>
        <apex>5.0</apex>
        <min>4.0</min>
                <max>6.0</max>
    </Triangular>
    <LogNormal name='BPrepairtime'>
        <mean>0.75</mean>
        <sigma>0.25</sigma>
    </LogNormal>
        <Uniform name='ScalFactPower'>
        <low>1.0</low>
        <hi>1.2</hi>
    </Uniform>
 </Distributions>
```

It is good practice to name the distribution something similar to what kind of variable is going to be sampled, since there might be many variables with the same kind of distributions but different input parameters.

13.1.7 Samplers

In the $< Samplers >$ block we want to define the variables that are going to be sampled. **Example**: We want to do the sampling of 3 variables:

- Battery Fail Time
- Battery Repair Time
- Scaling Factor Power Rate

We are going to sample these 3 variables using two different sampling methods: grid and MonteCarlo.

In RELAP5 the sampler reads the variable in this way: Given the name, the first number is the card number, the second number is the word number. In this example we are sampling:

- For card 0000588 (trip) the word 6 (battery failure time)
- For card 0000575 (trip) the word 6 (battery repari time)
- For card 20210000 (reactor power) the word 4 (reactor scaling factor)

We proceed to do so for both the Grid sampling and the MonteCarlo sampling.

```
<Samplers debug='True'>
    <Grid name='Grid_Sampler' initial_seed ='210491' >
        <variable name='0000588:6'>
            <distribution>BPfailtime</distribution>
            <grid type='value' construction='equal' lowerBound='0.0'
                steps='10'>2880</grid>
        </variable>
        <variable name='0000575:6'>
            <distribution>BPrepairtime</distribution>
            <grid type='value' construction='equal' lowerBound='0.0'
                steps='10'>2880</grid>
        </variable>
                <variable name='20210000:4'>
            <distribution>ScalFactPower</distribution>
            <grid type='value' construction='equal' lowerBound='1.0'
                steps='10'>0.02</grid>
        </Grid>
        <MonteCarlo name='MC_Sampler' limit='1000'>
                <variable name='0000588:6'>
            <distribution>BPfailtime</distribution>
                </variable>
                <variable name='0000575:6'>
            <distribution>BPrepairtime</distribution>
                </variable>
                <variable name='20210000:4'>
            <distribution>ScalFactPower</distribution>
                </variable>
```

```
        </MonteCarlo>
</Samplers>
```

It can be seen that each variable is connected with a proper distribution defined in the $< Distributions >$ block. Here is how the Input would be read for the first variable:

We are sampling a a variable situated in word 6 of the card 0000588 using a Grid sampling method. The distribution that this variable is following is a Triangular distribution (see section above). We are sampling this variable beginning from 0.0 in 10 *equal* steps of 2880.

13.1.8 Steps

For a RELAP interface the steps that are probably going to be used are the $< MultiRun >$. First we need to name the step: this name is one the ones used in the $< Sequence >$ block. In this case: Grid_Sampler and MC_Sampler.

```
<MultiRun name='Grid_Sampler' debug='True' re-seeding='210491'>
```

With this step with need to import all the files needed for the simulation:

- RELAP input file
- RELAP5run.py
- relapdata.py
- elements tables – tpfh2o

```
<Input class='Files' type=''>inputrelap.i</Input>
<Input class='Files' type=''>RELAP5run.py</Input>
<Input class='Files' type=''>relapdata.py</Input>
<Input class='Files' type=''>tpfh2o</Input>
```

We then need to define which model it is used:

```
<Model class='Models' type='Code'>MyRELAP</Model>
```

We then need to specify which Sampler is used, and this can be done as follows:

```
<Sampler class='Samplers' type='Grid'>Grid_Sampler</Sampler>
```

And lastly we need to specify what kind of Output the used wants. For example the user might want to make a DataBase (in RAVEn the DataBase crated is a HDF5 file. Here is a classical example:

```
<Output class='DataBases' type='HDF5'>MC_out</Output>
```

Following is the example of two MultiRun steps which use different sampling methods (grid and montecarlo), and creating two different DataBases for each one:

```
<Steps debug='True'>
<MultiRun name='Grid_Sampler' debug='True' re-seeding='210491'>
    <Input class='Files' type='' >X10.i</Input>
    <Input class='Files' type='' >RELAP5run.py</Input>
    <Input class='Files' type='' >relapdata.py</Input>
    <Input class='Files' type='' >tpfh2o</Input>
    <Model class='Models' type='Code' >MyRELAP</Model>
    <Sampler class='Samplers' type='Grid' >Grid_Sampler</Sampler>
    <Output class='DataBases' type='HDF5' >Grid_out</Output>
</MultiRun>
    <MultiRun name='MC_Sampler' debug='True' re-seeding='210491'>
    <Input class='Files' type='' >X10.i</Input>
    <Input class='Files' type='' >RELAP5run.py</Input>
    <Input class='Files' type='' >relapdata.py</Input>
    <Input class='Files' type='' >tpfh2o</Input>
    <Model class='Models' type='Code' >MyRELAP</Model>
    <Sampler class='Samplers' type='MonteCarlo' >MC_Sampler</Sampler>
    <Output class='DataBases' type='HDF5' >MC_out</Output>
</MultiRun>
</Steps>
```

13.1.9 DataBases

As shown in the $< Steps >$ block, the code is creating two DataBases called Grid_out and MC_out. So the user needs to input the following

```
<DataBases>
```

```
        <HDF5 name="Grid_out"/>
        <HDF5 name="MC_out"/>
</DataBases>
```

As listed before, this will create two DataBases. The files are .h5 extension files.

13.2 RELAP7 Interface

13.2.1 Files

In the $< Files >$ section, as specified before, there should be specified all the files needed for the code to run. In the case of RELAP7 most of the times the files needed are the following:

- RELAP7 Input file
- Control Logic file

Example:

```
<Files>nat_circ,control_logic.py</Files>
```

13.2.2 Models

For the $< Models >$ block RELAP7 uses the RAVEN executable. Here is a standard example of what can be used to use RELAP7 as the model:

```
<Models>
    <Code name='MyRAVEN'
        subType='RAVEN'><executable>~path/to/RAVEN-opt</executable></Code>
</Models>
```

13.2.3 Distributions

The $< Distributions >$ block, when using RELAP7 has to be specified also through the control logic. Given the names of the distributions, and their parameters, a python file should be used for the control logic. For example, it is required the sampling of a normal distribution for the primary pressure in RELAP7.

```
<Distributions>
 <Normal name="Prim_Pres">
 <mean>1000000</mean>
 <sigma>100<sigma/>
 </Normal>
</Distributions>
```

The python file associated to it should look like this:

```
def initial_function(monitored, controlled, auxiliary)
    print("monitored",monitored,"controlled",
    controlled,"auxiliary",auxiliary)

    controlled.pressure_in_pressurizer =
     distributions.Prim_Pres.getDistributionRandom()
    return
```

13.2.4 Samplers

In the $< Samplers >$ block there are going to be defined both which kind of sampling method used, the variables not sampled inside the control logic and of course which distributions will follow the chosen sampling method. For the example it was chosen a Monte-Carlo sampling with a 500 runs. The global initial pressure wasn't specified in the control logic so it is sampled in this block. It is also specified that such initial pressure has to follow the same distribution as the primary pressure.

```
<Samplers>
 <MonteCarlo name="MC_samp" limit="500">
  <variable name="GlobalParams|global_init_P">
   <distribution>Prim_Pres</distribution>
  </variable>
```

</MonteCarlo>
</Samplers>

13.2.5 Steps


INL
Idaho National Laboratory