MANUAL

INL/EXT-14-xxxxx
Draft Release
Printed December 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.

Idaho National Laboratory

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

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

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 December 2014

RAVEN User Manual

Andrea Alfonsi,
Cristian Rabiti,
Joshua Cogliati,
Diego Mandelli,
Robert Kinoshita.

[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 12);
- $< NumThreads >$***, 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 quequing 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:

```python
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

This sub-section is organized in two different parts: 1) Continuous 1-D distributions; 2) Discrete 1-D distributions. These two chapters cover all the requirements for using the different distribution entities.

5.1.1 1-Dimensional Continuous Distributions.

In this paragraph 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.

5.1.1.1 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. Its support is $x \in (0, 1)$.
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>
----------------------------
```

5.1.1.2 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. Its support is $x \in [0, +\inf)$.
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>
----------------------------
```

5.1.1.3 Gamma Distribution

The **Gamma** distribution is a two-parameter family of continuous probability distributions. The common exponential distribution and chi-squared distribution are special cases

of the gamma distribution. Its support is $x \in (0, +\inf)$.
The specifications of this distribution must be defined within the xml block $< Gamma >$. 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, shape parameter;
- $< beta >$, float, required parameter, 1/scale or the inverse scale parameter;
- $< low >$, float, optional parameter, lower domain boundary.

```
----------------------------
Example:
----------------------------
<Distributions>
  ...
  <Gamma name='...'>
    <alpha>⋆⋆⋆</alpha>
    <beta>⋆⋆⋆</beta>
    <low>⋆⋆⋆</low>
  </Gamma>
  ...
</Distributions>
----------------------------
```

5.1.1.4 Logistic Distribution

The **Logistic** distribution is a continuous distribution similar to the normal distribution with a CDF that is an instance of a logistic function. It resembles the normal distribution in shape but has heavier tails (higher kurtosis). Its support is $x \in (-\inf, +\inf)$.
The specifications of this distribution must be defined within the xml block $< Logistic >$. 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:

- $< location >$, float, required parameter, it is the distribution mean;
- $< scale >$, float, required parameter, scale parameter that is proportional to the standard deviation.

```
-----------------------------
Example:
-----------------------------
<Distributions>
  ...
  <Logistic name='...'>
    <location>⋆⋆⋆</location>
    <scale>⋆⋆⋆</scale>
  </Logistic>
  ...
</Distributions>
-----------------------------
```

5.1.1.5 LogNormal Distribution

The **LogNormal** distribution is a continuous distribution with the logarithm of the random variable being normally distributed. Its support is $x \in (0, +\inf)$.
The specifications of this distribution must be defined within the xml block $< LogNormal >$.
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:

- $< mean >$, float, required parameter, it is the distribution mean or expected value (in log-scale);

- $< sigma >$, float, required parameter, standard deviation.

```
-----------------------------
Example:
-----------------------------
<Distributions>
  ...
  <LogNormal name='...'>
    <mean>⋆⋆⋆</mean>
    <sigma>⋆⋆⋆</sigma>
  </LogNormal>
  ...
</Distributions>
-----------------------------
```

5.1.1.6 Normal Distribution

The **Normal** distribution (or Gaussian) distribution is a continuous distribution. It is extremely useful because of the central limit theorem, which states that, under mild conditions, the mean of many random variables independently drawn from the same distribution is distributed approximately normally, irrespective of the form of the original distribution. Its support is $x \in (-\inf, +\inf)$.
The specifications of this distribution must be defined within the xml block $< Normal >$. 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:

- $< mean >$, float, required parameter, it is the distribution mean or expected value;
- $< sigma >$, float, required parameter, standard deviation.

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

```
<Distributions>
  ...
  <Normal name='...'>
    <mean>⋆⋆⋆</mean>
    <sigma>⋆⋆⋆</sigma>
  </Normal>
  ...
</Distributions>
----------------------------
```

5.1.1.7 Triangular Distribution

The **Triangular** distribution is a continuous distribution that has a triangular shape for the Pdf. It is often used where the distribution is only vaguely known, but, like the uniform distribution, upper and lower limits are "known", but a "best guess", the mode or center point, is also added. It has been recommended as a "proxy" for the beta distribution. Its support is $lower \leq x \leq upper$.
The specifications of this distribution must be defined within the xml block $< Triangular >$. 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:

- $< apex >$, float, required parameter, "best guess", also called, peak factor;
- $< min >$, float, required parameter, domain lower boundary;
- $< max >$, float, required parameter, domain upper boundary.

```
----------------------------
Example:
----------------------------
<Distributions>
  ...
  <Triangular name='...'>
```

```
    <apex>⋆⋆⋆</apex>
    <min>⋆⋆⋆</min>
    <max>⋆⋆⋆</max>
  </Triangular>
  ...
</Distributions>
----------------------------
```

5.1.1.8 Uniform Distribution

The **Uniform** distribution is a continuous distribution with a rectangular shaped Pdf. It is often used where the distribution is only vaguely known, but upper and lower limits are "known". Its support is $lower \leq x \leq upper$.
The specifications of this distribution must be defined within the xml block $< Uniform >$. 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:

- $< low >$, float, required parameter, domain lower boundary;
- $< high >$, float, required parameter, domain upper boundary.

```
----------------------------
Example:
----------------------------
<Distributions>
  ...
  <Uniform name='...'>
    <low>⋆⋆⋆</low>
    <high>⋆⋆⋆</high>
  </Uniform>
  ...
</Distributions>
----------------------------
```

5.1.1.9 Weibull Distribution

The **Weibull** distribution is a continuous distribution that is often used in the field of failure analysis; in particular it can mimic distributions where the failure rate varies over time. If the failure rate is:

- constant over time, then $k = 1$, suggests that items are failing from random events;
- decreases over time, then $k < 1$, suggesting "infant mortality";
- increases over time, then $k > 1$, suggesting "wear out" - more likely to fail as time goes by.

Its support is $x \in [0, +\inf)$.
The specifications of this distribution must be defined within the xml block $< Weibull >$. 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:

- $< k >$, float, required parameter, shape parameter;
- $< lambda >$, float, required parameter, scale parameter.

```
----------------------------
Example:
----------------------------
<Distributions>
  ...
  <Weibull name='...'>
    <lambda>***</lambda>
    <k>***</k>
  </Weibull>
  ...
</Distributions>
----------------------------
```

5.1.2 1-Dimensional Discrete Distributions.

RAVEN currently supports 3 discrete distributions. In the following paragraphs, the input requirements are reported.

5.1.2.1 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. Its support is $k \in 0, 1$.
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>
----------------------------
```

5.1.2.2 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. Its support is $k \in 0, 1, 2, ..., n$.
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>
----------------------------
```

5.1.2.3 Poisson Distribution

The **Poisson** distribution is a discrete probability distribution that expresses the probability of a given number of events occurring in a fixed interval of time and/or space if these events occur with a known average rate and independently of the time since the last event. Its support is $k \in 1, 2, 3, 4, ...$
The specifications of this distribution must be defined within the xml block $< Poisson >$. 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:

- $< mu >$, float, required parameter, mean rate of events/time.

```
-----------------------------
Example:
-----------------------------
<Distributions>
  ...
  <Poisson name='...'>
    <mu>⋆⋆⋆</mu>
  </Poisson>
  ...
</Distributions>
-----------------------------
```

5.2 N-Dimensional Probability Distributions

We have the MultiVariate Normal distributions and 3 different type of user-input ND distribution. These types depend on the type of interpolation scheme that the user request. The inputs requirements are explained in the following:
Diego I choose you!

6 Samplers

The sampler is probably the most important entity in the RAVEN framework. Indeed, it performs the driving of the specific sampling strategy and, hence, determines the effectiveness of the analysis, from both an accuracy and computational point of view. The samplers, that are available in RAVEN, can be categorized in three main classes:

- **Once-through**
- **Dynamic Event Tree (DET)**
- **Adaptive**

Before analyzing each sampler in details, it is important to mention that each type has a similar syntax to input the variables that need to be “sampled”. In the example below the variable “variableName” is going to be sampled by the Sampler “WhatEverSampler” using the distribution “aDistribution”.

```
<Simulation>
  ...
  <Samplers>
    ...
    <WhatEverSampler name=’whatever’>
      ...
     <variable name=’variableName’>
       ...
       <distribution>aDistribution</distribution>
       ...
     </variable>
      ...
    </WhatEverSampler>
    ...
  </Samplers>
  ...
</Simulation>
```

As reported in section 13, the variable naming syntax, for external driven codes, depends on the way the “code interface” has been implemented. For example, if the code has an input structure like the one reported below, the variable name would be “$I - Level|II - Level|variable$”; in this way, the relative code interface (and input parser) will know which variable needs to be perturbed and the “recipe” to access to it. As reported in 13, its syntax is chosen by the developer of the “code interface” and is implemented in the interface only (no modifications are needed in the RAVEN code).

```
--------------------------------------------
Example Input:
[I-Level]
  [./II-Level]
   variable = xxx
  [../]
[]
XML relative
----------
Example XML block:
<variable name=’’I−Level|II−Level|variable’’>
  <distribution>exampleDistribution</distribution>
</variable>
--------------------------------------------
```

6.1 Once-through Samplers.

The once-through sampler category collects all the strategies that perform the sampling of the input space without exploiting, through dynamic learning approaches, the information made available from the outcomes of calculation previously performed (adaptive sampling) and the common system evolution (patterns) that different sampled calculations can generate in the phase space (dynamic event tree). In the RAVEN framework, five different and well-known “once-through” samplers are available:

- **Monte Carlo (MC)**
- **Stratified**
- **Grid Based**

- **Response Surface Design of Experiment**
- **Factorial Design of Experiment**

From a practical point of view, these sampling strategies represent different ways to perturb the input space. In the following paragraphs, the input requirements and a small explaination of the different sampling methodologies are reported.

6.1.1 Monte Carlo.

Monte-Carlo sampling approach is one of the most well-known and used approaches to perform exploration of the input space. The main idea, behind it, is the random perturbation of the input space accordingly with uniform or parameter-based probability density functions.
The specifications of this Sampler must be defined within the xml block $< MonteCarlo >$. This xml-node needs to contain the attribute:

- **name**, *required string attribute*, user-defined name of this Sampler. 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);
- **limit**, *required integer attribute*, number of MonteCarlo samples needs to be generated;
- **initial_seed**, *optional integer attribute*, initial seeding of random number generator. *Default = random seed*;
- **reseedAtEachIteration**, *optional boolean/string attribute*, perform a re-seeding for each sample generated (True values = True, yes, y, t, si, dajie). *Default = False*;

In the **MonteCarlo** input block, the user needs to specify the variables need to be sampled. As already mentioned, these variables are inputted within consecutive xml blocks called $< variable >$:

- $< variable >$, xml node, required parameter. This xml-node needs to contain the attribute:
  - **name**, *required string attribute*, user-defined name of this variable.

  In the variable node, the following xml-nodes need to be specified:

- $< distribution >$***, string, required field.***. Name of the distribution that is associated to this variable. Its name needs to be contained in the **Distributions** block explained in sections 5.

```
--------------------------------------------------------------
Example:
--------------------------------------------------------------
<Samplers>
  ...
  <MonteCarlo name='MCname' limit='10' initial_seed='200286'
      reseedAtEachIteration='false'>
    <variable name='var1'>
      <distribution>***</distribution>
    </variable>
  </MonteCarlo>
  ...
</Samplers>
--------------------------------------------------------------
```

6.1.2 Grid.

Grid sampling approach is probably the simplest exploration approach that can be employed to explore the uncertain domain. The main idea, behind it, is the construction of a N-Dimensional grid, where each dimension is represented by the uncertain variables. This approach performs the sampling of each node of the grid. The sampling of the grid consists in evaluating the answer of the system under all the possible combinations among the different variables' values, with respect a predefined discretization metric. In RAVEN two discretization metrics are available: 1) Cumulative Distribution Function, and 2) Value. Thus, the grid meshing can be inputted either in probability or in absolute values.
The specifications of this Sampler must be defined within the xml block $< Grid >$. This xml-node needs to contain the attribute:

- **name**, *required string attribute*, user-defined name of this Sampler. 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);

In the **Grid** input block, the user needs to specify the variables need to be sampled. As already mentioned, these variables are inputted within consecutive xml blocks called $< variable >$:

- $< variable >$, xml node, required parameter. This xml-node needs to contain the attribute:

  – **name**, *required string attribute*, user-defined name of this variable.

  In the variable node, the following xml-nodes need to be specified:

  – $< distribution >$***, string, required field.***. Name of the distribution that is associated to this variable. Its name needs to be contained in the **Distributions** block explained in sections 5;

  – $< grid >$***, float or space separated floats, required field.***. The content of this xml node depends on the definition of the associated attributes:

    ∗ **type**, *required string attribute*, user-defined discretization metric type: 1) CDF, the grid is going to be specified based on Cumulative Distribution Function probability thresholds, and 2) $value$, the grid is going to be provided inputting absolute variable values;

    ∗ **construction**, *required string attribute*, how the grid needs to be constructed, independently by the its type (i.e. CDF or $value$).

    Based on the $construction$ type, the content of the $< grid >$ xml node and the requirements for other attributes change:

    ∗ *construction = "equal"*. The grid is going to be constructed equally-spaced (type = "value") or equally-probable (type == "CDF"). This construction type requires the definition of additional attributes:

      · **steps**, *required integer attribute*, number of equally-spaced/probable discretization steps.

      · **upperBound**, *required float attribute*, the upper limit of the grid. NB. This attribute must be specified if the **lowerBound** has not been defined;

      · **lowerBound**, *required float attribute*, the lower limit of the grid. NB. This attribute must be specified if the **upperBound** has not been defined;

      This construction type requires that the content of the xml node $< grid >$ represents the step size (either in probability or value). The attributes

lowerBound and **upperBound** are mutually exclusive (only one of them can be specified):
If the *upperBound* is present, the grid lower bound is going to be at the $upperBound - steps * stepSize$
If the *lowerBound* is present, the grid upper bound is going to be at the $lowerBound + steps * stepSize$ The lower and upper bounds are checked against the associated $< distribution >$ bounds. If one or both of them fell/s outside the distribution's bounds, the code is going to raise an error.

- *construction* = *"custom"*. The grid is going to directly be specified by the user. No additional attributes are needed.
  This construction type requires that the xml node $< grid >$ contains the actual mesh bins. For example, if the grid "type" is "CDF", in the body of $< grid >$, the user is going to specify CDF probability thresholds (nodalization in probability).

```
-----------------------------------------------------------
Example:
-----------------------------------------------------------
<Samplers>
  ...
  <Grid name='Gridname'>
    <variable name='var1'>
      <distribution>***</distribution>
      <grid type='value' construction='equal' steps='100'
          lowerBound='1.0'>0.2</grid>
    </variable>
    <variable name='var2'>
      <distribution>***</distribution>
      <grid type='CDF' construction='equal' steps='5'
          lowerBound='0.0'>0.2</grid>
    </variable>
    <variable name='var3'>
      <distribution>***</distribution>
      <grid type='value' construction='equal' steps='100'
          upperBound='21.0'>0.2</grid>
    </variable>
    <variable name='var4'>
      <distribution>***</distribution>
      <grid type='CDF' construction='equal' steps='5'
          upperBound='1.0'>0.2</grid>
    </variable>
    <variable name='var5'>
      <distribution>⋆⋆⋆</distribution>
      <grid type='value' construction='custom'>0.2 0.5 10.0</grid>
    </variable>
    <variable name='var6'>
      <distribution>⋆⋆⋆</distribution>
      <grid type='CDF' construction='custom'>0.2 0.5 1.0</grid>
    </variable>
  </Grid>
  ...
</Samplers>
----------------------------------------------------------
```

6.1.3 Stratified.

Stratified sampling approach is a method for the exploration of the input space that basically consists in dividing the uncertain domain into subgroups before sampling. In the "stratified" sampling, these subgroups must be:

- mutually exclusive: every element in the population must be assigned to only one stratum (subgroup);
- collectively exhaustive: no population element can be excluded.

Then simple random sampling or systematic sampling is applied within each stratum. It is worth to notice that the well-known Latin Hyper Cube sampling represents a specialization of the Stratified approach, when the domain strata are constructed in equally-probable CDF bins.
The specifications of this Sampler must be defined within the xml block $< Stratified >$. This xml-node needs to contain the attribute:

- **name**, *required string attribute*, user-defined name of this Sampler. 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);

In the **Stratified** input block, the user needs to specify the variables need to be sampled. As already mentioned, these variables are inputted within consecutive xml blocks called $< variable >$:

- $< variable >$, xml node, required parameter. This xml-node needs to contain the attribute:
  - **name**, *required string attribute*, user-defined name of this variable.

  In the variable node, the following xml-nodes need to be specified:

  - $< distribution >$***, string, required field.***. Name of the distribution that is associated to this variable. Its name needs to be contained in the **Distributions** block explained in sections 5;
  - $< grid >$***, float or space separated floats, required field.***. The content of this xml node depends on the definition of the associated attributes:
    - **type**, *required string attribute*, user-defined discretization metric type: 1) CDF, the grid is going to be specified based on Cumulative Distribution Function probability thresholds, and 2) $value$, the grid is going to be provided inputting absolute variable values;
    - **construction**, *required string attribute*, how the grid needs to be constructed, independently by the its type (i.e. CDF or $value$).

    Based on the $construction$ type, the content of the $< grid >$ xml node and the requirements for other attributes change:

    - *construction = "equal"*. The grid is going to be constructed equally-spaced (type = "value") or equally-probable (type == "CDF"). This construction type requires the definition of additional attributes:
      - **steps**, *required integer attribute*, number of equally-spaced/probable discretization steps.
      - **upperBound**, *required float attribute*, the upper limit of the grid. NB. This attribute must be specified if the **lowerBound** has not been defined;
      - **lowerBound**, *required float attribute*, the lower limit of the grid. NB. This attribute must be specified if the **upperBound** has not been defined;

      This construction type requires that the content of the xml node $< grid >$ represents the step size (either in probability or value). The attributes

  lowerBound and **upperBound** are mutually exclusive (only one of them can be specified):
  If the *upperBound* is present, the grid lower bound is going to be at the $upperBound - steps * stepSize$
  If the *lowerBound* is present, the grid upper bound is going to be at the $lowerBound + steps * stepSize$ The lower and upper bounds are checked against the associated $< distribution >$ bounds. If one or both of them fell/s outside the distribution's bounds, the code is going to raise an error.

  ∗ *construction* = *"custom"*. The grid is going to directly be specified by the user. No additional attributes are needed.
  This construction type requires that the xml node $< grid >$ contains the actual mesh bins. For example, if the grid "type" is "CDF", in the body of $< grid >$, the user is going to specify CDF probability thresholds (nodalization in probability).

As it is inferable from above, the input specifications for the **Stratified** sampler are similar the Grid sampler ones. It is important to mention again that for each zone (grid mesh) only a point, randomly selected, is picked and not all the nodal combinations (like in the Grid sampling).

```
------------------------------------------------------------
Example:
------------------------------------------------------------
<Samplers>
  ...
  <Stratified name='StratifiedName'>
    <variable name='var1'>
      <distribution>⋆⋆⋆</distribution>
      <grid type='CDF' construction='equal' steps='5'
          lowerBound='0.0'>0.2</grid>
    </variable>
    <variable name='var2'>
      <distribution>⋆⋆⋆</distribution>
      <grid type='value' construction='equal' steps='100'
          upperBound='21.0'>0.2</grid>
    </variable>
    <variable name='var3'>
      <distribution>⋆⋆⋆</distribution>
      <grid type='CDF' construction='custom'>0.2 0.5 1.0</grid>
    </variable>
  </Stratified>
  ...
</Samplers>
-----------------------------------------------------------------
```

6.1.4 Response Surface Design.

Response Surface Design approach is one of the most common Design of Experiment (DOE) methodology that are currently in use. It explores the relationships between several explanatory variables and one or more response variables. The main idea of RSM is to use a sequence of designed experiments to obtain an optimal response. RAVEN Currently employs two different algorithms that can be classified within this methodology family:

- **Box-Behnken** design: this methodology has the aim to achieve the following goals:
  - Each factor, or independent variable, is placed at one of three equally spaced values, usually coded as -1, 0, +1. (At least three levels are needed for the following goal);
  - The design should be sufficient to fit a quadratic model, that is, one containing squared terms and products of two factors;
  - The ratio of the number of experimental points to the number of coefficients in the quadratic model should be reasonable (in fact, their designs kept it in the range of 1.5 to 2.6);
  - The estimation variance should more or less depend only on the distance from the center (this is achieved exactly for the designs with 4 and 7 factors), and should not vary too much inside the smallest (hyper)cube containing the experimental points.

  Each design can be thought of as a combination of a two-level (full or fractional) factorial design with an incomplete block design. In each block, a certain number of factors are put through all combinations for the factorial design, while the other factors are kept at the central values.

- **Central Composite** design: the design consists of three distinct sets of experimental runs:

  - A factorial (perhaps fractional) design in the factors studied, each having two levels;
  - A set of center points, experimental runs whose values of each factor are the medians of the values used in the factorial portion. This point is often replicated in order to improve the precision of the experiment;
  - A set of axial points, experimental runs identical to the centre points except for one factor, which will take on values both below and above the median of the two factorial levels, and typically both outside their range. All factors are varied in this way.

  This methodology is useful for building a second order (quadratic) model for the response variable without needing to use a complete three-level factorial experiment.

All the parameters, needed for setting up the algorithms reported above, must be defined within the xml block $< ResponseSurfaceDesign >$. This xml-node needs to contain the attribute:

- **name**, *required string attribute*, user-defined name of this Sampler. 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);

In the **ResponseSurfaceDesign** input block, the user needs to specify the variables need to be sampled. As already mentioned, these variables are inputted within consecutive xml blocks called $< variable >$:

- $< variable >$, xml node, required parameter. This xml-node needs to contain the attribute:
  - **name**, *required string attribute*, user-defined name of this variable.

  In the variable node, the following xml-nodes need to be specified:

  - $< distribution >$***, string, required field.***. Name of the distribution that is associated to this variable. Its name needs to be contained in the **Distributions** block explained in sections 5;
  - $< boundaries >$***, xml node, required field.***. Within this block the boundaries for this variable are defined. This xml-node needs to contain the attribute:

    - **type**, *required string attribute*, how the boundaries are defined. This attribute can be:
      - either *CDF*: the boundaries are going to be provided as probability CDF thresholds
      - or *value*: the boundaries are going to be provided as variable values.

    Within the $< boundaries >$ xml block, the following sub-nodes need to be provided:

    - $< lower >$***, float, required field.***. The lower limit of this variable;
    - $< upper >$***, float, required field.***. The upper limit of this variable.

The main xml block $< ResponseSurfaceDesign >$ needs to contain an additional subnode called $< ResponseSurfaceDesignSettings >$. In this sub-node, the user needs to specify different settings, depending on the algorithm needs to be used:

- $< type >$***, string, required field.***. This xml node needs to contain the name of the algorithm needs to be used. Based on the chosen algorithm, other nodes need to be defined:
  - $< type > BoxBehnken < type/ >$. If Box-Behnken is specified, the following additional node might be inputted:
    - $< ncenters >$***, integer, optional field.***. The number of center points to include in the box. If this parameter is not specified, then a pre-determined number of points are automatically included. *Default = None*.

    NB. In order to employ the "Box-Behnken" design, at least 3 variables must be inputted.
  - $< type > CentralComposite < type/ >$. If Central Composite is specified, the following additional nodes might be inputted:
    - $< centers >$***, comma separated integers, optional field.***. The number of center points to be included. This block needs to contain 2 integers values separated by a comma. The first entry represents the number of centers to be added for the factorial block; the second one the one for the star block. *Default = 4,4*.
    - $< alpha >$***, string, optional field.***. In this node, the user might decides how α factor needs to be determined. Two options are available: 1) *alpha = orthogonal*, for orthogonal design, or 2) *alpha = rotatable*, for rotatable design. *Default = orthogonal*.

- $< face >$, ***string, optional field..*** In this node, the user defines how faces should be constructed. Three options are available: 1) *face* = *circumscribed*, for circumscribed facing; 2) *face* = *inscribed*, for inscribed facing; 3) *face* = *faced*, for faced facing. *Default* = *circumscribed*.

NB. In order to employ the “Central Composite” design, at least 2 variables must be inputted.

```
----------------------------------------------------------
Example:
----------------------------------------------------------
<Samplers>
  ...
    <ResponseSurfaceDesign name='BoxBehnkenRespDesign'>
        <ResponseSurfaceDesignSettings>
            <type>BoxBehnken</type>
            <ncenters>***</ncenters>
        </ResponseSurfaceDesignSettings>
        <variable name='var1' >
            <distribution >Gauss1</distribution>
            <boundaries type=”CDF”>
                <lower>0.0</lower>
                <upper>1.0</upper>
            </boundaries>
        </variable>
        <!-- N.B. at least 3 variables need to inputted
                in order to employ this algorithm
         -->
    </ResponseSurfaceDesign>
    <ResponseSurfaceDesign name='CentralCompositeRespDesign'>
        <ResponseSurfaceDesignSettings>
            <type>CentralComposite</type>
            <centers>***, ***</centers>
            <alpha>orthogonal</alpha>
            <face>circumscribed</face>
        </ResponseSurfaceDesignSettings>
        <variable name='var4' >
            <distribution >Gauss1</distribution>
            <boundaries type=”CDF”>
            <lower>***</lower>
            <upper>***</upper>
         </boundaries>
      </variable>
      <!-- N.B. at least 2 variables need to inputted
            in order to employ this algorithm
       -->
   </ResponseSurfaceDesign>
  ...
</Samplers>
---------------------------------------------------------
```

6.1.5 Factorial Design.

Factorial Design method is an important method to determine the effects of multiple variables on a response. Factorial design can reduce the number of experiments one has to perform by studying multiple factors simultaneously. Additionally, it can be used to find both main effects (from each independent factor) and interaction effects (when both factors must be used to explain the outcome). Factorial design tests all possible conditions. Because factorial design can lead to a large number of trials, which can become expensive and time-consuming, factorial design is best used for a small number of variables with few states (1 to 3). Factorial design works well when interactions between variables are strong and important and where every variable contributes significantly. RAVEN currently employs three different algorithms that can be classified within this methodology family:

- **General Full Factorial** design: this methodology explore the input space investigating all possible combinations of a set of factors (variables).
- **2-Level Fractional-Factorial** design: this methodology consists of a carefully chosen subset (fraction) of the experimental runs of a full factorial design. The subset is chosen so as to exploit the sparsity-of-effects principle to expose information about the most important features of the problem studied, while using a fraction of the effort of a full factorial design in terms of experimental runs and resources.
- **Plackett-Burman** design: this method is used to identify the most important factors early in the experimentation phase when complete knowledge about the system

is usually unavailable. It is an efficient screening method to identify the active factors (variables) using as few samples as possible. In Plackett-Burman designs, main effects have a complicated confounding relationship with two-factor interactions. Therefore, these designs should be used to study main effects when it can be assumed that two-way interactions are negligible.

All the parameters, needed for setting up the algorithms reported above, must be defined within the xml block $< FactorialDesign >$. This xml-node needs to contain the attribute:

- **name**, *required string attribute*, user-defined name of this Sampler. 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);

In the **FactorialDesign** input block, the user needs to specify the variables need to be sampled. As already mentioned, these variables are inputted within consecutive xml blocks called $< variable >$:

- $< variable >$, xml node, required parameter. This xml-node needs to contain the attribute:

  - **name**, *required string attribute*, user-defined name of this variable.

  In the variable node, the following xml-nodes need to be specified:

  - $< distribution >$***, string, required field.***. Name of the distribution that is associated to this variable. Its name needs to be contained in the **Distributions** block explained in sections 5;
  - $< grid >$***, float or space separated floats, required field.***. The content of this xml node depends on the definition of the associated attributes:
    - $*$ **type**, *required string attribute*, user-defined discretization metric type: 1) CDF, the grid is going to be specified based on Cumulative Distribution Function probability thresholds, and 2) $value$, the grid is going to be provided inputting absolute variable values;
    - $*$ **construction**, *required string attribute*, how the grid needs to be constructed, independently by the its type (i.e. CDF or $value$).

Based on the $construction$ type, the content of the $< grid >$ xml node and the requirements for other attributes change:

- $*$ *construction = "equal"*. The grid is going to be constructed equally-spaced (type = "value") or equally-probable (type == "CDF"). This construction type requires the definition of additional attributes:
    - · **steps**, *required integer attribute*, number of equally-spaced/probable discretization steps.
    - · **upperBound**, *required float attribute*, the upper limit of the grid. NB. This attribute must be specified if the **lowerBound** has not been defined;
    - · **lowerBound**, *required float attribute*, the lower limit of the grid. NB. This attribute must be specified if the **upperBound** has not been defined;

  This construction type requires that the content of the xml node $< grid >$ represents the step size (either in probability or value). The attributes **lowerBound** and **upperBound** are mutually exclusive (only one of them can be specified):
  If the *upperBound* is present, the grid lower bound is going to be at the $upperBound - steps * stepSize$
  If the *lowerBound* is present, the grid upper bound is going to be at the $lowerBound + steps * stepSize$ The lower and upper bounds are checked against the associated $< distribution >$ bounds. If one or both of them fell/s outside the distribution's bounds, the code is going to raise an error.
- $*$ *construction = "custom"*. The grid is going to directly be specified by the user. No additional attributes are needed.
  This construction type requires that the xml node $< grid >$ contains the actual mesh bins. For example, if the grid "type" is "CDF", in the body of $< grid >$, the user is going to specify CDF probability thresholds (nodalization in probability).

The main xml block $< FactorialDesign >$ needs to contain an additional sub-node called $< FactorialSettings >$. In this sub-node, the user needs to specify different settings, depending on the algorithm needs to be used:

- $< type >$***, string, required field.***. This xml node needs to contain the name of the algorithm needs to be used. Based on the chosen algorithm, other nodes need to be defined:

- $< type > full < type/ >$. Full factorial design. If “full” is specified, no additional nodes need to be inputted. NB. The Full factorial design does not have any limitations in the number of discretization bins can be inputted in the $< grid >$ xml node for each $< variable >$.
- $< type > 2levelFract < type/ >$. Two-levels Fractional-Factorial design. If “levelFract” is specified, the following additional nodes must be inputted:
  - $< gen >$***, space separated strings, required field.***. In this block the confounding mapping needs to inputted. By instance, in this block the user defines the decisions on a fraction of the full-factorial by allowing some of the factor main effects to be confounded with other factor interaction effects. This is done by defining an alias structure that defines, symbolically, these interactions. These alias structures are written like “C = AB” or “I = ABC”, or “AB = CD”, etc. These define how one column is related to the others.
  - $< genMap >$***, space separated strings, required field.***. In this block the user defines the mapping between the “gen” symbolic aliases and the variables that have been inputted in the $< FactorialDesign >$ main block.

  NB. The Two-levels Fractional-Factorial design is limited to 2 discretization bins that can be inputted in the $< grid >$ xml node for each $< variable >$.
- $< type > pb < type/ >$. Plackett-Burman design. If “pb” is specified, no additional nodes need to be inputted.
  NB. The Full factorial design does not have any limitations in the number of discretization bins can be inputted in the $< grid >$ xml node for each $< variable >$.

```
------------------------------------------------------------
Example:
------------------------------------------------------------
<Samplers>
  ...
    <FactorialDesign name='fullFactorial'>
        <FactorialSettings>
            <type>full</type>
        </FactorialSettings>
        <variable name='var1' >
            <distribution >***</distribution>
            <grid type='value' construction='custom' >0.02 0.03 0.5</grid>
    </variable>
    <variable name=’var2’ >
        <distribution >***</distribution>
        <grid type=’CDF’ construction=’custom’>0.5 0.7 1.0</grid>
    </variable>
</FactorialDesign>
<FactorialDesign name=’2levelFractFactorial’>
    <FactorialSettings>
        <type>2levelFract</type>
        <gen>a,b,ab</gen>
        <genMap>var1,var2,var3</genMap>
    </FactorialSettings>
    <variable name=’var1’ >
        <distribution >***</distribution>
        <grid type=’value’ construction=’custom’ >0.02 0.5</grid>
    </variable>
    <variable name=’var2’ >
        <distribution >***</distribution>
        <grid type=’CDF’ construction=’custom’>0.5 1.0</grid>
    </variable>
    <variable name=’var3’>
        <distribution >***</distribution>
        <grid type=’value’ upperBound=’4’ construction=’equal’
            steps=’1’>0.5</grid>
    </variable>
</FactorialDesign>
<FactorialDesign name=’pbFactorial’>
    <FactorialSettings>
       <type>pb</type>
    </FactorialSettings>
    <variable name=’var1’ >
        <distribution >***</distribution>
        <grid type=’value’ construction=’custom’ >0.02 0.5</grid>
    </variable>
    <variable name=’VarGauss2’ >
        <distribution >***</distribution>
        <grid type=’CDF’ construction=’custom’>0.5 1.0</grid>
    </variable>
    </FactorialDesign>
  ...
</Samplers>
-----------------------------------------------------------
```

6.2 Dynamic Event Tree (DET) Samplers.

The Dynamic Event Tree methodologies are designed to take the timing of events explicitly into account, which can become very important especially when uncertainties in complex phenomena are considered. Hence, the main idea of this methodology is to let a system code determine the pathway of an accident scenario within a probabilistic "environment". In this methodologies' family a continuous monitoring of the system evolution in the phase space is needed. In order to use the DET-based methods, the generic driven code needs to have, at least, an internal trigger system and, consequentially, a "restart" capability. In the RAVEN framework, three different DET samplers are available:

- **Dynamic Event Tree (DET)**
- **Hybrid Dynamic Event Tree (HDET)**
- **Adaptive Dynamic Event Tree (ADET)**

The ADET methodology represents an hybrid between the DET and Adaptive sampling approaches. For this reason, its input requirements are reported in the Adaptive Samplers' section (6.3).

6.2.1 Dynamic Event Tree.

Dynamic Event Tree sampling approach is a sampling strategy that is designed to take the timing of events, in transient/accident scenarios, explicitly into a account.
From an application point of view, a N-Dimensional grid is built on the CDF space. A single simulation is spooned and a set of triggers is added to the system code control logic. Every time a trigger gets activated (one of the CDF thresholds in the grid is overpassed), a new set of simulations (branches) is spooned. Each branch carries its conditional probability. In RAVEN code, the triggers are defined specifying a grid, using a predefined

discretization metric. In RAVEN two discretization metrics are available: 1) Cumulative Distribution Function, and 2) Value. Thus, the trigger thresholds can be inputted either in probability or in absolute values.
The specifications of this Sampler must be defined within the xml block $< DynamicEventTree >$. This xml-node needs to contain the attribute:

- **name**, *required string attribute*, user-defined name of this Sampler. 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);
- **print_end_xml**, *optional string/boolean attribute*, this attribute controls the dumping of a "summary" of the DET performed in an xml external output. *Default = False*.
- **maxSimulationTime**, *optional float attribute*, this attribute controls the maximum "mission" time of the simulation underneath. *Default = None*.

In the **DynamicEventTree** input block, the user needs to specify the variables need to be sampled. As already mentioned, these variables are inputted within consecutive xml blocks called $< variable >$:

- $< variable >$, xml node, required parameter. This xml-node needs to contain the attribute:
  - **name**, *required string attribute*, user-defined name of this variable.

  In the variable node, the following xml-nodes need to be specified:

  - $< distribution >$***, string, required field.***. Name of the distribution that is associated to this variable. Its name needs to be contained in the **Distributions** block explained in sections 5;
  - $< grid >$***, float or space separated floats, required field.***. The content of this xml node depends on the definition of the associated attributes:
    - ∗ **type**, *required string attribute*, user-defined discretization metric type: 1) CDF, the grid is going to be specified based on Cumulative Distribution Function probability thresholds, and 2) $value$, the grid is going to be provided inputting absolute variable values;
    - ∗ **construction**, *required string attribute*, how the grid needs to be constructed, independently by the its type (i.e. CDF or $value$).

Based on the $construction$ type, the content of the $< grid >$ xml node and the requirements for other attributes change:

- *construction = "equal"*. The grid is going to be constructed equally-spaced (type = "value") or equally-probable (type == "CDF"). This construction type requires the definition of additional attributes:
  - **steps**, *required integer attribute*, number of equally-spaced/probable discretization steps.
  - **upperBound**, *required float attribute*, the upper limit of the grid. NB. This attribute must be specified if the **lowerBound** has not been defined;
  - **lowerBound**, *required float attribute*, the lower limit of the grid. NB. This attribute must be specified if the **upperBound** has not been defined;

  This construction type requires that the content of the xml node $< grid >$ represents the step size (either in probability or value). The attributes **lowerBound** and **upperBound** are mutually exclusive (only one of them can be specified):
  If the *upperBound* is present, the grid lower bound is going to be at the $upperBound - steps * stepSize$
  If the *lowerBound* is present, the grid upper bound is going to be at the $lowerBound + steps * stepSize$ The lower and upper bounds are checked against the associated $< distribution >$ bounds. If one or both of them fell/s outside the distribution's bounds, the code is going to raise an error.
- *construction = "custom"*. The grid is going to directly be specified by the user. No additional attributes are needed.
  This construction type requires that the xml node $< grid >$ contains the actual mesh bins. For example, if the grid "type" is "CDF", in the body of $< grid >$, the user is going to specify CDF probability thresholds (nodalization in probability).

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

```
<Samplers>
  ...
  <DynamicEventTree name='DETname'>
    <variable name='var1'>
```

```
      <distribution>⋆⋆⋆</distribution>
      <grid type=’value’ construction=’equal’ steps=’100’
          lowerBound=’1.0’>0.2</grid>
    </variable>
    <variable name=’var2’>
      <distribution>⋆⋆⋆</distribution>
      <grid type=’CDF’ construction=’equal’ steps=’5’
          lowerBound=’0.0’>0.2</grid>
    </variable>
    <variable name=’var3’>
      <distribution>⋆⋆⋆</distribution>
      <grid type=’value’ construction=’equal’ steps=’100’
          upperBound=’21.0’>0.2</grid>
    </variable>
    <variable name=’var4’>
      <distribution>⋆⋆⋆</distribution>
      <grid type=’CDF’ construction=’equal’ steps=’5’
          upperBound=’1.0’>0.2</grid>
    </variable>
    <variable name=’var5’>
      <distribution>⋆⋆⋆</distribution>
      <grid type=’value’ construction=’custom’>0.2 0.5 10.0</grid>
    </variable>
    <variable name=’var6’>
      <distribution>⋆⋆⋆</distribution>
      <grid type=’CDF’ construction=’custom’>0.2 0.5 1.0</grid>
    </variable>
  </DynamicEventTree>
   ...
</Samplers>
-----------------------------------------------------------
```

6.2.2 Hybrid Dynamic Event Tree.

Hybrid Dynamic Event Tree sampling approach is a sampling strategy that represents an evolution of the Dynamic Event Tree method for the simultaneous exploration of the epistemic and aleatory uncertain space. In similar approaches, the uncertainties are generally treated employing a Monte-Carlo sampling approach (epistemic) and DET methodology (aleatory). The HDET methodology, developed within the RAVEN code, can reproduce the capabilities employed by this approach, but provides additional sampling strategies to the user. The epistemic or epistemic-like uncertainties can be sampled through the following strategies:

- Monte-Carlo;
- Grid sampling;
- Stratified (e.g., Latin Hyper Cube).

From a practical point of view, the user defines the parameters that need to be sampled by one or more different approaches. The HDET module samples those parameters creating a N-Dimensional Grid characterized by all the possible combinations of the input space coordinates coming from the different sampling strategies. Each coordinate in the input space represents a separated and parallel standard DET exploration of the uncertain domain. The HDET methodology allows the user to completely explore the uncertain domain employing one methodology. The addition of Grid sampling strategy among the approaches usable, allow the user to perform a discrete parametric study, under aleatory and epistemic uncertainties.
Regarding the input requirements, the HDET sampler is a "sub-type" of the $< DynamicEventTree >$ sampler. For this reason, its specifications must be defined within the xml block $< DynamicEventTree >$. This xml-node needs to contain the attribute:

- **name**, *required string attribute*, user-defined name of this Sampler. 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);
- **print_end_xml**, *optional string/boolean attribute*, this attribute controls the dumping of a "summary" of the DET performed in an xml external output. *Default = False*;
- **maxSimulationTime**, *optional float attribute*, this attribute controls the maximum "mission" time of the simulation underneath. *Default = None*.

In the **DynamicEventTree** input block, the user needs to specify the variables need to be sampled. As already mentioned, these variables are inputted within consecutive xml blocks called $< variable >$:

- $< variable >$, xml node, required parameter. This xml-node needs to contain the attribute:

  - **name**, *required string attribute*, user-defined name of this variable.

  In the variable node, the following xml-nodes need to be specified:

  - $< distribution >$***, string, required field.***. Name of the distribution that is associated to this variable. Its name needs to be contained in the **Distributions** block explained in sections 5;
  - $< grid >$***, float or space separated floats, required field.***. The content of this xml node depends on the definition of the associated attributes:
    - $*$ **type**, *required string attribute*, user-defined discretization metric type: 1) CDF, the grid is going to be specified based on Cumulative Distribution Function probability thresholds, and 2) $value$, the grid is going to be provided inputting absolute variable values;
    - $*$ **construction**, *required string attribute*, how the grid needs to be constructed, independently by the its type (i.e. CDF or $value$).

    Based on the $construction$ type, the content of the $< grid >$ xml node and the requirements for other attributes change:

    - $*$ *construction = "equal"*. The grid is going to be constructed equally-spaced (type = "value") or equally-probable (type == "CDF"). This construction type requires the definition of additional attributes:
      - · **steps**, *required integer attribute*, number of equally-spaced/probable discretization steps.
      - · **upperBound**, *required float attribute*, the upper limit of the grid. NB. This attribute must be specified if the **lowerBound** has not been defined;
      - · **lowerBound**, *required float attribute*, the lower limit of the grid. NB. This attribute must be specified if the **upperBound** has not been defined;

      This construction type requires that the content of the xml node $< grid >$ represents the step size (either in probability or value). The attributes **lowerBound** and **upperBound** are mutually exclusive (only one of them can be specified):
      If the *upperBound* is present, the grid lower bound is going to be at the $upperBound - steps * stepSize$

If the *lowerBound* is present, the grid upper bound is going to be at the $lowerBound + steps * stepSize$ The lower and upper bounds are checked against the associated $< distribution >$ bounds. If one or both of them fell/s outside the distribution's bounds, the code is going to raise an error.

$*$ *construction* = *"custom"*. The grid is going to directly be specified by the user. No additional attributes are needed.
This construction type requires that the xml node $< grid >$ contains the actual mesh bins. For example, if the grid "type" is "CDF", in the body of $< grid >$, the user is going to specify CDF probability thresholds (nodalization in probability).

In order to activate the **Hybrid Dynamic Event Tree** sampler, the main xml block $< DynamicEventTree >$ needs to contain, at least, an additional sub-node called $< HybridSamplerSettings >$. As already mentioned, the user can combine the Monte-Carlo, Stratified and Grid approaches in order to create a "pre-sampling" N-Dimensional grid, from whose nodes a standard DET method is employed. For this reason, the user can specify at maximum three $< HybridSamplerSettings >$ sub-nodes (i.e. one for each of the available once-through samplers). This sub-node needs to contain the attribute:

- **type**, *required string attribute*, type of pre-sampling strategy needs to be used. Available are: 1) MonteCarlo, 2) Grid, and 3) Stratified.

Independently on the type of "pre-sampler" that has been specified, the $< HybridSamplerSettings >$ must contain the variables need to be sampled. As already mentioned, these variables are inputted within consecutive xml blocks called $< variable >$:

- $< variable >$, xml node, required parameter. This xml-node needs to contain the attribute:

  - **name**, *required string attribute*, user-defined name of this variable.

  In the variable node, the following xml-nodes need to be specified:

  - $< distribution >$***, string, required field.***. Name of the distribution that is associated to this variable. Its name needs to be contained in the **Distributions** block explained in sections 5;

If the pre-sampling strategy is either **Grid** or **Stratified**, within the $< variable >$ blocks, the user needs to specify the sub-node $< grid >$. As for the standard DET, the content of this xml node depends on the definition of the associated attributes:

- **type**, *required string attribute*, user-defined discretization metric type: 1) CDF, the grid is going to be specified based on Cumulative Distribution Function probability thresholds, and 2) $value$, the grid is going to be provided inputting absolute variable values;
- **construction**, *required string attribute*, how the grid needs to be constructed, independently by the its type (i.e. CDF or $value$).

Based on the $construction$ type, the content of the $< grid >$ xml node and the requirements for other attributes change:

- *construction* = *"equal"*. The grid is going to be constructed equally-spaced (type = "value") or equally-probable (type == "CDF"). This construction type requires the definition of additional attributes:
  - **steps**, *required integer attribute*, number of equally-spaced/probable discretization steps.
  - **upperBound**, *required float attribute*, the upper limit of the grid. NB. This attribute must be specified if the **lowerBound** has not been defined;
  - **lowerBound**, *required float attribute*, the lower limit of the grid. NB. This attribute must be specified if the **upperBound** has not been defined;

  This construction type requires that the content of the xml node $< grid >$ represents the step size (either in probability or value). The attributes **lowerBound** and **upperBound** are mutually exclusive (only one of them can be specified):
  If the *upperBound* is present, the grid lower bound is going to be at the $upperBound - steps * stepSize$
  If the *lowerBound* is present, the grid upper bound is going to be at the $lowerBound + steps * stepSize$ The lower and upper bounds are checked against the associated $< distribution >$ bounds. If one or both of them fell/s outside the distribution's bounds, the code is going to raise an error.
- *construction* = *"custom"*. The grid is going to directly be specified by the user. No additional attributes are needed.

This construction type requires that the xml node $< grid >$ contains the actual mesh bins. For example, if the grid “type” is “CDF”, in the body of $< grid >$, the user is going to specify CDF probability thresholds (nodalization in probability).

```
-----------------------------------------------------
Example:
-----------------------------------------------------
<Samplers>
  ...
  <DynamicEventTree name=’HybridDETname’ print_end_xml=”True”>
      <HybridSamplerSettings type=’MonteCarlo’ limit=’2’>
          <variable name=’∗∗∗’ >
              <distribution>⋆⋆⋆</distribution>
          </variable>
          <variable name=’∗∗∗’ >
              <distribution>⋆⋆⋆</distribution>
              <grid type=’CDF’ construction=’equal’ steps=’1’
                  lowerBound=’0.1’>0.1</grid>
          </variable>
      </HybridSamplerSettings>
      <HybridSamplerSettings type=’Grid’>
          <!-- Point sampler way (directly sampling the
              variable) -->
          <variable name=’∗∗∗’ >
              <distribution>⋆⋆⋆</distribution>
              <grid type=’CDF’ construction=’equal’ steps=’1’
                  lowerBound=’0.1’>0.1</grid>
          </variable>
          <variable name=’∗∗∗’ >
              <distribution>⋆⋆⋆</distribution>
              <grid type=’CDF’ construction=’equal’ steps=’1’
                  lowerBound=’0.1’>0.1</grid>
          </variable>
      </HybridSamplerSettings>
      <HybridSamplerSettings type=’Stratified’>
          <!-- Point sampler way (directly sampling the
              variable ) -->
          <variable name=’∗∗∗’ >
```

```
              <distribution>***</distribution>
              <grid type=’CDF’ construction=’equal’ steps=’1’
                  lowerBound=’0.1’>0.1</grid>
          </variable>
          <variable name=’***’ >
              <distribution>***</distribution>
              <grid type=’CDF’ construction=’equal’ steps=’1’
                  lowerBound=’0.1’>0.1</grid>
          </variable>
      </HybridSamplerSettings>
      <!-- DYNAMIC EVENT TREE INPUT (it goes outside an
          inner block like HybridSamplerSettings) -->
        <Distribution name=’***’>
          <distribution >***</distribution>
          <grid type=’CDF’ construction=’custom’>0.1 0.8</grid>
        </Distribution>
  </DynamicEventTree>
  ...
</Samplers>
--------------------------------------------------------------------------
```

6.3 Adaptive Samplers.

The Adaptive Samplers’ family provides the possibility to perform smart sampling (also known as adaptive sampling) as an alternative to classical “once-through” techniques. The motivation is that system simulations are often computationally expensive, time-consuming, and high dimensional with respect to the number of input parameters. Thus, exploring the space of all possible simulation outcomes is infeasible using finite computing resources. During simulation-based probabilistic risk analysis, it is important to discover the relationship between a potentially large number of input parameters and the output of a simulation using as few simulation trials as possible. This is a typical context for performing adaptive sampling where a few observations are obtained from the simulation, a reduced order model (ROM) is built to represent the simulation space, and new samples are selected based on the model constructed. The reduced order model (see section 10.3) is then updated based on the simulation results of the sampled points. In this way, it is attempted to gain the most information possible with a small number of carefully selected sampled points, limiting the number of expensive trials needed to understand features of the system space. In the following, the specific use case of identifying the limit surface, i.e. the boundaries in the simu-lation space between system failure and system suc-cess (see Figure 4) is analyzed. For this scope two classes of algorithms are considered:

- Model-based algorithms;
- Data-based algorithms.

In the first class, the built reduced order model aims to approximate the real response function of the system as function of the input parameters. Once it is built, it is used to search for the points that are in the proximity of the limit surface using contour reconstruction based algorithms. Response function can be built using Support Vector Machines or Kriging based interpolators. On the other side, data-based algorithms do not build a response function based reduced order model but determine the location of the limit surface directly from the neighborhood graph constructed from the training data, without any dependencies on a particular prediction model. These algorithms begin the search of the limit surface by directly building a neighborhood structure as the surrogate model on the initial training data. It then creates a candidate set by first obtaining linearly interpolated points along spanning edges of the graph, and introducing a random perturbation along all dimensions to these points.
Currently, RAVEN provides support for two adaptive algorithm:

- Adaptive Sampler for Limit Surface Search;
- Adaptive Dynamic Event Tree.

In the following paragraphs, the input requirements and a small explanation of the different sampling methodologies are reported.

6.3.1 Adaptive Sampler.

Adaptive Sampler approach is an advanced methodology that employs a smart sampling around transition zones that determine a change in the status of the system (Limit Surface). To perform such sampling, RAVEN uses Reduced Order Models for predicting, in the input space, the location(s) of these transitions, in order to accelerate the exploration of

the input space in proximity of the Limit Surface.
The specifications of this Sampler must be defined within the xml block $< Adaptive >$. This xml-node needs to contain the attribute:

- **name**, *required string attribute*, user-defined name of this Sampler. 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);

In the **Adaptive** input block, the user needs to specify the variables need to be sampled. As already mentioned, these variables are inputted within consecutive xml blocks called $< variable >$:

- $< variable >$, xml node, required parameter. This xml-node needs to contain the attribute:
  - **name**, *required string attribute*, user-defined name of this variable.

  In the variable node, the following xml-node needs to be specified:
  - $< distribution >$***, string, required field.***. Name of the distribution that is associated to this variable. Its name needs to be contained in the **Distributions** block explained in sections 5.

In addition to the $< variable >$ nodes, the main xml node $< Adaptive >$ needs to contain two supplementary sub-nodes:

- $< Convergence >$***, float, required field.*** Convergence tolerance. The meaning of this tolerance depends on the definition of other attributes that might be defined in this xml node:
  - **limit**, *optional integer attribute*, the maximum number of adaptive samples. *Default = infinitive*;
  - **forceIteration**, *optional boolean attribute*, this attribute controls if at least a number of iterations equal to **limit** must be performed. *Default = False* ;
  - **weight**, *optional string attribute*, this attribute defines on what the convergence check need to be performed.

    - *weight = probability*, the convergence is checked in terms of probability (Cumulative Distribution Function);
    - *weight = none*, the convergence is checked on the hyper-volume in terms of absolute values.

    Default = probability;
  - **persistence**, *optional integer attribute*, this option is an additional convergence check. It represents the number of times the computed error needs to be below the inputted tolerance, in order to consider the algorithm in converged condition. *Default = 0*;
  - **subGridTol**, *optional string attribute*, in this attribute the user can define a tolerance for constructing a sub-grid on which the acceleration ROM is going to be tested. *Default = None*;

  Summarizing, this xml node contains the information that are needed in order to control this sampler convergence criterion.

- $< Assembler >$***, xml node, required field.*** This xml node contains a “list” of objects that are required (or optional) for the functionality of the Adaptive Sampler. The objects must be listed with a rigorous syntax that, except for the xml node tag, is common among all the objects. Each of these sub-nodes must contain 2 attributes that are used to map those within the simulation framework:
  - **class**, *required string attribute*, it is the main “class” the listed object is from. For example, it can be “Models”, “Functions”, etc;
  - **type**, *required string attribute*, it is the object identifier or sub-type. For example, it can be “ROM”, “External”, etc.

  The **Adaptive Sampler** approach requires or optionally accepts the following objects’ types:

  - $< ROM >$***, string, optional field.***. If inputted, the body of this xml node must contain the name of a ROM defined in the $< Models >$ block (see section 10.3);
  - $< Function >$***, string, required field.***.The body of this xml block needs to contain the name of an External Function defined within the $< Functions >$ main block (see section 9). This object represents the boolean function that defines the transition boundaries. This function must implement a method called *__residuumSign(self)*, that returns either -1 or 1, depending on the system conditions (see section 9;

- $< TargetEvaluation >$***, string, required field.***. The target evaluation object represents the container where the system evaluations are stored. From a practical point of view, this xml node must contain the name of a Data defined in the $< Datas >$ block (see section **??**). The adaptive sampling accepts “Datas” of type “TimePoint” and “TimePointSet” only.

```
-----------------------------------------------------------
Example:
-----------------------------------------------------------
XML INPUT:
-----------------------------------------------------------
<Samplers>
  ...
  <Adaptive name = 'AdaptiveName'>
    <Assembler>
        <ROM class = 'Models' type = 'ROM' >ROMname</ROM>
        <Function class = 'Functions' type = 'External'
            >FunctionName</Function>
        <TargetEvaluation class = 'Datas' type =
            'TimePointSet'>DataName</TargetEvaluation>
    </Assembler>
    <Convergence limit = '3000' subGridTol= '0.001' forceIteration = 'False'
        weight = 'none' persistence = '5'>
      1e-2
    </Convergence>
    <variable name = 'var1'>
        <distribution>***</distribution>
    </variable>
    <variable name = 'var2'>
        <distribution>***</distribution>
    </variable>
    <variable name = 'var3'>
        <distribution>***</distribution>
    </variable>
  </Adaptive>
  ...
</Samplers>
```

```
Example:

EXTERNAL FUNCTION:

def __residuumSign(self):
  if self.whatEverValue < self.OtherValue :
    return 1
  else:
    return -1
```

6.3.2 Adaptive Dynamic Event Tree.

Adaptive Dynamic Event Tree approach is an advanced methodology that employs a smart sampling around transition zones that determine a change in the status of the system (Limit Surface), using the support of a Dynamic Event Tree methodology. The main idea of the application of the previously explained adaptive sampling approach to the DET comes from the observation that the DET, when evaluated from a Limit Surface perspective, is intrinsically adaptive. For this reason, it appears natural to use the DET approach to perform a goal-function oriented pre-sampling of the input space.
To perform such sampling, RAVEN uses Reduced Order Models for predicting, in the input space, the location(s) of these transitions, in order to accelerate the exploration of the input space in proximity of the Limit Surface.
The specifications of this Sampler must be defined within the xml block $< AdaptiveDynamicEventTree >$.
This xml-node needs to contain the attributes:

- **name**, *required string attribute*, user-defined name of this Sampler. 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);
- **print_end_xml**, *optional string/boolean attribute*, this attribute controls the dumping of a “summary” of the DET performed in an xml external output. *Default = False*;
- **maxSimulationTime**, *optional float attribute*, this attribute controls the maximum “mission” time of the simulation underneath. *Default = None*.
- **mode**, *optional string attribute*, adaptive DET mode. This attribute controls when the adaptive search needs to begin. Two options are available:

  - *mode* = *post*, if this option is activated, the sampler firstly performs a standard Dynamic Event Tree. At end of it, it uses the outcomes to start the adaptive search in conjunction with the DET support;
  - *mode* = *online*, if this option is activated, the adaptive search starts at begin, during the initial standard Dynamic Event Tree. Whenever a transition is detected, the **Adaptive Dynamic Event Tree** starts it goal-oriented search using the DET as support;

  Default = post.

- **updateGrid**, *optional boolean attribute*, update DET grid flag. If it is true, each Adaptive Request is going to update the meshing of the initial DET grid. *Default = True*.

In the **AdaptiveDynamicEventTree** input block, the user needs to specify the variables need to be sampled. As already mentioned, these variables are inputted within consecutive xml blocks called $< variable >$:

- $< variable >$, xml node, required parameter. This xml-node needs to contain the attribute:

  - **name**, *required string attribute*, user-defined name of this variable.

  In the variable node, the following xml-nodes need to be specified:

  - $< distribution >$***, string, required field.***. Name of the distribution that is associated to this variable. Its name needs to be contained in the **Distributions** block explained in sections 5;
  - $< grid >$***, float or space separated floats, required field.***. The content of this xml node depends on the definition of the associated attributes:
    - **type**, *required string attribute*, user-defined discretization metric type: 1) CDF, the grid is going to be specified based on Cumulative Distribution Function probability thresholds, and 2) $value$, the grid is going to be provided inputting absolute variable values;
    - **construction**, *required string attribute*, how the grid needs to be constructed, independently by the its type (i.e. CDF or $value$).

    Based on the $construction$ type, the content of the $< grid >$ xml node and the requirements for other attributes change:

        - $*$ *construction* = *“equal”*. The grid is going to be constructed equally-spaced (type = “value”) or equally-probable (type == “CDF”). This construction type requires the definition of additional attributes:
            - · **steps**, *required integer attribute*, number of equally-spaced/probable discretization steps.
            - · **upperBound**, *required float attribute*, the upper limit of the grid. NB. This attribute must be specified if the **lowerBound** has not been defined;
            - · **lowerBound**, *required float attribute*, the lower limit of the grid. NB. This attribute must be specified if the **upperBound** has not been defined;

          This construction type requires that the content of the xml node $< grid >$ represents the step size (either in probability or value). The attributes **lowerBound** and **upperBound** are mutually exclusive (only one of them can be specified):
          If the *upperBound* is present, the grid lower bound is going to be at the $upperBound - steps * stepSize$
          If the *lowerBound* is present, the grid upper bound is going to be at the $lowerBound + steps * stepSize$ The lower and upper bounds are checked against the associated $< distribution >$ bounds. If one or both of them fell/s outside the distribution’s bounds, the code is going to raise an error.
        - $*$ *construction* = *“custom”*. The grid is going to directly be specified by the user. No additional attributes are needed.
          This construction type requires that the xml node $< grid >$ contains the actual mesh bins. For example, if the grid “type” is “CDF”, in the body of $< grid >$, the user is going to specify CDF probability thresholds (nodalization in probability).

In addition to the $< variable >$ nodes, the main xml node $< AdaptiveDynamicEventTree >$ needs to contain two supplementary sub-nodes:

- $< Convergence >$***, float, required field.*** Convergence tolerance. The meaning of this tolerance depends on the definition of other attributes that might be defined in this xml node:
    - **limit**, *optional integer attribute*, the maximum number of adaptive samples. *Default = infinitive*;

- **forceIteration**, *optional boolean attribute*, this attribute controls if at least a number of iterations equal to **limit** must be performed. *Default = False* ;
- **weight**, *optional string attribute*, this attribute defines on what the convergence check need to be performed.
  - ∗ *weight = probability*, the convergence is checked in terms of probability (Cumulative Distribution Function);
  - ∗ *weight = none*, the convergence is checked on the hyper-volume in terms of absolute values.

  Default = probability;
- **persistence**, *optional integer attribute*, this option is an additional convergence check. It represents the number of times the computed error needs to be below the inputted tolerance, in order to consider the algorithm in converged condition. *Default = 0*;
- **subGridTol**, *optional string attribute*, in this attribute the user can define a tolerance for constructing a sub-grid on which the acceleration ROM is going to be tested. *Default = None*;

Summarizing, this xml node contains the information that are needed in order to control this sampler convergence criterion.

- $< Assembler >$***, xml node, required field.*** This xml node contains a "list" of objects that are required (or optional) for the functionality of the Adaptive Sampler. The objects must be listed with a rigorous syntax that, except for the xml node tag, is common among all the objects. Each of these sub-nodes must contain 2 attributes that are used to map those within the simulation framework:
  - **class**, *required string attribute*, it is the main "class" the listed object is from. For example, it can be "Models", "Functions", etc;
  - **type**, *required string attribute*, it is the object identifier or sub-type. For example, it can be "ROM", "External", etc.

  The **Adaptive Sampler** approach requires or optionally accepts the following objects' types:
  - $< ROM >$***, string, optional field.***. If inputted, the body of this xml node must contain the name of a ROM defined in the $< Models >$ block (see section 10.3);

- $< Function >$***, string, required field.***.The body of this xml block needs to contain the name of an External Function defined within the $< Functions >$ main block (see section 9). This object represents the boolean function that defines the transition boundaries. This function must implement a method called *__residuumSign(self)*, that returns either -1 or 1, depending on the system conditions (see section 9;
- $< TargetEvaluation >$***, string, required field.***. The target evaluation object represents the container where the system evaluations are stored. From a practical point of view, this xml node must contain the name of a Data defined in the $< Datas >$ block (see section **??**). The adaptive sampling accepts “Datas” of type “TimePoint” and “TimePointSet” only.

```
-----------------------------------------------------------
Example:
-----------------------------------------------------------
XML INPUT:
<Samplers>
  ...
  <AdaptiveDynamicEventTree name = 'AdaptiveName'>
    <Assembler>
        <ROM class = 'Models' type = 'ROM' >ROMname</ROM>
        <Function class = 'Functions' type = 'External'
            >FunctionName</Function>
        <TargetEvaluation class = 'Datas' type =
            'TimePointSet'>DataName</TargetEvaluation>
    </Assembler>
    <Convergence limit = '3000' subGridTol= '0.001' forceIteration = 'False'
        weight = 'none' persistence = '5'>
      1e-2
    </Convergence>
    <variable name = 'var1'>
        <distribution>***</distribution>
        <grid type='CDF' construction='custom'>0.1 0.8</grid>
    </variable>
    <variable name = 'var2'>
        <distribution>***</distribution>
        <grid type='CDF' construction='custom'>0.1 0.8</grid>
    </variable>
    <variable name = 'var3'>
        <distribution>***</distribution>
        <grid type='CDF' construction='custom'>0.1 0.8</grid>
    </variable>
  </AdaptiveDynamicEventTree>
  ...
</Samplers>


EXTERNAL FUNCTION:
def __residuumSign(self):
  if self.whatEverValue < self.OtherValue :
    return  1
  else:
    return -1
```

7 Datas

As it could infer from the previous chapters, in the RAVEN code different entities interact to each other in order to create, ideally, an infinite number of different calculation flows. These interactions are performed through a system that is "understandable" by each "entity". This system, neglecting the grammar imprecision, is called "Datas" system. The "Datas" system is a container of Data objects of different type that can be constructed during the evolution of the particular calculation flow, can be used as input or output of particular Model (see Roles' meaning in section 10), etc. Currently, RAVEN support 4 different data types, each with a particular conceptual meaning:

- **TimePoint**: The *TimePoint* entity, as the name suggests, "describes" the state of the system in a certain point in time. In other words, it can be considered a mapping between a set of parameters in the input space and the resulting outcomes in the output space at a particular time;
- **TimePointSet**: The *TimePointSet* object is, obviously, a collection of single *TimePoint(s)*. It can be considered a mapping between multiple sets of parameters in the input space and the resulting sets of outcomes in the output space at a particular point in time;
- **History**: The *History* entity "describes" the temporal evolution of the state of the system within a certain input domain;
- **Histories**: The *Histories* object is, obviously, a collection of single *History(ies)*. t can be considered a mapping between multiple sets of parameters in the input space and the resulting sets of temporal evolutions in the output space.

As inferable from the brief description reported above, each "Data" object represents a mapping between a set of parameters and the resulting outcomes. The Data objects are defined within the main XML block called $< Datas >$:

```
------------------------------
<Simulation>
  ...
  <Datas>
```

```
    <WhatEverData name='***'>
      . . .
    </WhatEverData>
  </Datas>
    . . .
</Simulation>
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
```

The specifications of each “Data” type must be defined within:

- **TimePoint** $=> < TimePoint >$
- **TimePointSet** $=> < TimePointSet >$
- **History** $=> < History >$
- **Histories** $=> < Histories >$

Independently on the type of Data, the respective XML node needs (or not) to contain the attribute:

- **name**, *required string attribute*, user-defined name of this Data. 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);
- **inputTs**, *optional integer attribute*, This attribute can be used to specify at which “time step” the input space needs to be retrieved. NB. If the user wants to take those conditions from the end of the simulation, it can directly input “-1”. *Default = 0*;
- **operator**, *optional string attribute*, The operator attribute is aimed to perform simple operations on the data to be stored. The 3 options currently available are *operator = max*, *operator = min*, *operator = average*, with obvious meaning. *Default = None*;
- **hierarchical**, *optional boolean attribute*, If True this data is going to be constructed, if possible, in an hierarchical fashion. *Default = False*;

In each XML node (e.g. $< TimePoint >$ or $< Histories >$), the user needs to specify the following sub nodes:

- $< Inputs >$***, comma separated string, required field.***. List of input parameters this data is connected to. NB. In case the “Data” type is either *TimePoint* or *History*, this XML node must contain the attribute **history**, where the name of the associated history needs to be placed;

- $< Outputs >$***, comma separated string, required field.***. List of output parameters this data is connected to;

```
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
<Datas>
    <TimePoint name=’∗∗∗’ inputTs=’−1’ operator=’max’ hierarchical=’False’>
        <Input history=’a_history_name’>⋆⋆⋆,⋆⋆⋆,⋆⋆⋆</Input>
        <Output>⋆⋆⋆,⋆⋆⋆</Output>
    </TimePoint>
    <TimePointSet name=’∗∗∗’>
        <Input>⋆⋆⋆,⋆⋆⋆,⋆⋆⋆</Input>
        <Output>⋆⋆⋆,⋆⋆⋆</Output>
     </TimePointSet>
     <History name=’∗∗∗’>
        <Input history=’a_history_name’>⋆⋆⋆,⋆⋆⋆,⋆⋆⋆</Input>
        <Output>⋆⋆⋆,⋆⋆⋆</Output>
     </History>
     <Histories name=’∗∗∗’>
        <Input>⋆⋆⋆,⋆⋆⋆,⋆⋆⋆</Input>
        <Output>⋆⋆⋆,⋆⋆⋆</Output>
     </Histories>
</Datas>
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
```

8 Databases

The RAVEN framework provides the capability to store and retrieve data to/from an external database. Currently RAVEN has support for only a database type called **HDF5**. This database, depending on the data format is receiving, will organize itself in a “parallel” or “hierarchical” fashion. The user can create as many database objects as needed. The DataBase objects are defined within the main XML block called $< DataBases >$:

```
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
<Simulation>
   . . .
  <DataBases>
        . . .
        <HDF5 name=”***”/>
        <HDF5 name=”***”/>
        . . .
  </DataBases>
   . . .
</Simulation>
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
```

The specifications of each DataBase of type HDF5 needs to be defined within the XML block $< HDF5 >$, that needs (or not) to contain the attributes:

- **name**, *required string attribute*, user-defined name of this Data. 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);
- **directory**, *optional string attribute*, this attribute can be used to specify a particular directory path where to create the database, if no *filename* is specified, or from where open an already existing one, if *filename* is provided. *Default = raven/framework/DataBaseStorage*;
- **filename**, *optional string attribute*, this attribute can be used to specified the filename of an HDF5 that already exists in the *directory*. This is the only way to let RAVEN know that an HDF5 should be opened and not overwritten. NB. When this attribute is not specified, the newer database filename is going to be named *name*.h5. *Default = None*;
- **compression**, *optional string attribute*, compression algorithm to be used. Available are:

- *compression* = *gzip*, best where portability is required. Good compression, moderate speed;
- *compression* = *lzf*, Low to moderate compression, very fast.

Default = *no compression*;

```
<DataBases>
    <HDF5 name=”***” directory=’’path_to_a_dir’’ compression=’’lzf”’/>
    <HDF5 name=”***” filename=’’existing_hdf5.h5’’/>
</DataBases>
```

9 Functions

The RAVEN code provides support for the usage of user-defined external functions. These functions are python modules, with a format that is automatically interpretable by the RAVEN framework. For example, the user can define its own method to perform a particular post-processing activity and the code is going to embed and use the function as it is an active part of the code itself. In this section, all the information needed to define the XML input syntax and the format of the accepted functions are here reported.
The specifications of an external Function must be defined within the XML block $< External >$. This XML node needs to contain the attributes:

- **name**, *required string attribute*, user-defined name of this Function. 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);
- **file**, *required string attribute*, file name with its absolute or relative path. NB. If a relative path is specified, it must be noticed it is relative with respect to where the user runs the code.

In order to make the RAVEN code aware of the variables the user is going to manipulate/use in its own python function, the variables need to be specified in the $< External >$ input block. The user needs to input, within this block, only the variables that RAVEN needs to be aware of (i.e. the variables are going to directly be used by the code) and not the local variables that the user does not want to, for example, store in a RAVEN internal object. These variables are inputted within consecutive XML blocks called $< variable >$:

- $< variable >$, string, required parameter. In the body of this XML node, the user needs to specify the name of the variable. This variable needs to match a variable used/defined in the external python function.

When the external function variables are defined, at run time, RAVEN initialize those and take track of their values during the simulation. Each variable defined in the $< External >$ block is available in the function as a python "self". In the following, an example of an user-defined external function is reported (python module and its relative XML input specifications).

```
Python Function Example:
```

```python
import numpy as np

def residuumSign(self):
  if self.var1 < self.var2 :
    return  1
  else:
    return -1
```

```
----------------------------------------
XML Example:
----------------------------------------
<Simulation>
  ...
  <Functions>
    ...
    <External name=’whatever’ file=’path_to_python_file’>
     ...
     <variable>var1</variable>
     <variable>var2</variable>
     ...
    </External>
    ...
  </Functions>
   ...
</Simulation>
----------------------------------------
```

10 Models

In the RAVEN code a crucial entity is represented by a Model. A model is an object that employs a mathematical representation of a phenomenology, either physical or of other nature (e.g. statistical operators, etc.). From a practical point of view, it can be seen, as a "black box" that, given an input, returns an output.
In the RAVEN code, a strict classification of the different models is present. As obviously, each "class" of models is represented by the definition reported above, but it can be further classified based on the peculiar functionalities:

- **Code**. This "class" is the representation of an external system code that employs an high fidelity physical model;
- **Dummy**. The "Dummy" object is a model that acts as "transfer" tool. The only action it performs is transferring the the information in the input space (inputs) into the output space (outputs). For example, it can be used to check the effect of a Sampling strategy, since its outputs are the sampled parameters' values (input space) and a counter that keeps track of the number of times an evaluation has been requested;
- **ROM**. A ROM is a mathematical model of fast solution trained to predict a response of interest of a physical system. The "training" process is performed by "sampling" the response of a physical model with respect variation of its parameters subject to probabilistic behavior. The results (outcomes of the physical model) of those sampling are fed into the algorithm representing the ROM that tunes itself to replicate those results;
- **ExternalModel**. As the name suggests, an external model is an entity that is embedded in the RAVEN code at run time. This object allows the user to create a python module that is going to be treated as a predefined internal model object;
- **PostProcessor**. The post-processor "class" of objects is the container of all the actions that can be performed to manipulate and process the data in order to extract key information, such as statistical quantities, etc.

Before analyzing each model in details, it is important to mention that each type needs to be contained in the main XML node $< Models >$, as reported below:

Example:

```
<Simulation>
 . . .
 <Models>
   . . .
   <WhatEverModel name='whatever'>
     . . .
   </WhatEverModel>
   . . .
 </Models>
 . . .
</Simulation>
```

In the following sub-sections each **Model** type is fully analyzed and described.

10.1 Code

As already mentioned, the model **Code** is the representation of an external system software that employs an high fidelity physical model. The link between RAVEN and the driven code is performed at run time, through coded interfaces that are the responsible of transferring the information from the code to RAVEN and vice versa. In section 13 all the available interfaces are reported and, for advanced users, section **??** explains how to couple a newer code.
The specifications of this Model must be defined within the xml block $< Code >$. This XML node needs to contain the attributes:

- **name**, *required string attribute*, user-defined name of this Model. 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);
- **subType**, *required string attribute*, in this attribute the user selects the code that needs to be associated to this Model. NB. See section 13 to check which codes are currently supported.

In the **Code** input block, the following XML sub-nodes are available:

- $< executable >$ ***, string, required field.***. In this node, the user needs to specify the path of the executable to be used. NB. In this node, either the absolute or relative path can be inputted;
- $< alias >$ ***, string, optional field.***. In the $< alias >$ block the user can specify aliases for some variables of interest coming from the code this model refers to. These aliases can be used in the whole input to refer to the code variables. In the body of this node the user specifies the name of the variable that RAVEN will look for in the output files of the code. The actual alias, usable throughout the input, are instead defined in the attribute **variable**. NB. The user can specify as many aliases as needed. *Default = None*.

Example:

```
-------------------------------------------------------------------
<Simulation>
  ...
  <Models>
    ...
    <Code name=’***’ subType=’RAVEN_Driven_code’>
      <executable>path_to_executable</executable>
      <alias variable=’internal_variable_name1’>
        External_Code_Variable_Name_1
      </alias>
      <alias variable=’internal_variable_name2’>
        External_Code_Variable_Name_2
      </alias>
    </Code>
    ...
  </Models>
  ...
</Simulation>
-------------------------------------------------------------------
```

10.2 Dummy.

The model **Dummy** is an object that acts as “transfer” tool. The only action it performs is transferring the the information in the input space (inputs) into the output space (outputs).

For example, it can be used to check the effect of a Sampling strategy, since its outputs are the sampled parameters' values (input space) and a counter that keeps track of the number of times an evaluation has been requested.
The specifications of this Model must be defined within the xml block $< Dummy >$. This XML node needs to contain the attributes:

- **name**, *required string attribute*, user-defined name of this Model. 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);
- **subType**, *required string attribute*, this attribute must be kept empty.

If this model, in a *Step*, is associated to a *Data* with the role of **Output**, it expects that one of the output parameters of such *Data* is identified by the keyword "OutputPlaceHolder" (see section 12).

Example:

```
<Simulation>
  ...
  <Models>
    ...
    <Dummy name='***' subType=''/>
    ...
  </Models>
  ...
</Simulation>
```

10.3 ROM

A Reduced Order Model (ROM) is a mathematical model of fast solution trained to predict a response of interest of a physical system. The "training" process is performed by "sampling" the response of a physical model with respect variation of its parameters subject, for example, to probabilistic behavior. The results (outcomes of the physical model) of those sampling are fed into the algorithm representing the ROM that tunes itself to replicate those results. RAVEN supports several different types of ROMs, both internally developed and imported through an external library called "SciKitLearn" [1]. Currently in RAVEN the Reduced Order Models are classified in 4 main "classes" that, once chosen, provide access to several different algorithms:

- **NDspline;**
- **NDinvDistWeigth;**
- **microSphere;**
- **SciKitLearn.**

The specifications of this Model must be defined within the XML block $< ROM >$. This XML node needs to contain the attributes:

- **name**, *required string attribute*, user-defined name of this Model. 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);
- **subType**, *required string attribute*, in this attribute the user defines which of the main "classes" needs to be used, choosing among the previously reported types. Obviously, this choice conditions the subsequent the required and/or optional $< ROM >$ sub nodes.

In the **ROM** input block, the following XML sub-nodes are required, independently on the "main" class inputted in the attribute *subType*:

- $< Features >$ ***, comma separated string, required field.***. In this node, the user needs to specify the names of the features of this ROM. NB. These parameters are going to be requested for the training of this object (see section 12;
- $< Target >$ ***, comma separated string, required field.***. This XML node contains a comma separated list of the targets of this ROM. By Instance, these parameters are the Figure of Merits this ROM is supposed to predict. NB. These parameters are going to be requested for the training of this object (see section 12.

As already mentioned, all the types and meaning of the remaining sub-nodes depend on the main "class" type specified in the attribute *subType*. In the following sections the specifications of each type are reported.

10.3.1 NDspline.

The main “class” NDspline contains a single ROM type, based on a N-Dimensional spline interpolation/extrapolation. The spline interpolation is a form of interpolation where the interpolant is a special type of piecewise polynomial called a spline. The interpolation error can be made small even when using low degree polynomials for the spline. Spline interpolation avoids the problem of Runge’s phenomenon, in which oscillation can occur between points when interpolating using high degree polynomials.
In order to use this Reduced Order Model, the $< ROM >$ attribute *subType* needs to be “NDspline” (i.e. *subType* = *“NDspline”*). No further XML sub-nodes are required.
NB. This ROM type must be trained from a Regular Cartesian Grid. By instance, it can only be trained from the outcomes of a Grid Sampling strategy.

Example:

```
<Simulation>
  . . .
  <Models>
    . . .
    <ROM name=’∗∗∗’ subType=’NDspline’>
       <Features>⋆⋆⋆,⋆⋆⋆,⋆⋆⋆</Features>
       <Target>⋆⋆⋆,⋆⋆⋆</Target>
     </ROM>
    . . .
  </Models>
  . . .
</Simulation>
```

10.3.2 NDinvDistWeigth.

The main “class” NDinvDistWeigth contains a single ROM type, based on a N-Dimensional Inverse Distance Weighting formulation. Inverse Distance Weighting (IDW) is a type of deterministic method for multivariate interpolation with a known scattered set of points. The assigned values to unknown points are calculated with a weighted average of the values available at the known points.

In order to use this Reduced Order Model, the $< ROM >$ attribute *subType* needs to be “NDinvDistWeigth” (i.e. *subType* = *“NDinvDistWeigth”*). The specification of the ROM *“NDinvDistWeigth”* needs to be completed inputting, within the main XML node $< ROM >$, of the following sub-node:

- $< p >$ ***, integer, required field.***. This node contains an $integer > 0$ that represents the “power parameter’. For the choice of value for $< p >$,it is necessary to consider the degree of smoothing desired in the interpolation/extrapolation, the density and distribution of samples being interpolated, and the maximum distance over which an individual sample is allowed to influence the surrounding ones (lower p means greater importance for points faraway).

Example:

```
-----------------------------------------------------------------
<Simulation>
  ...
  <Models>
    ...
    <ROM name='***' subType='NDinvDistWeigth'>
       <Features>***,***,***</Features>
       <Target>***</Target>
       <p>3</p>
     </ROM>
    ...
  </Models>
  ...
</Simulation>
-----------------------------------------------------------------
```

10.3.3 MicroSphere.

Not yet functional. Its validity for prediction purposes needs to be still assessed.

10.3.4 SciKitLearn.

The main "class" SciKitLearn represents the container of several Reduced Order Models that are available in RAVEN through the external library SciKitLearn [1].
In order to use this Reduced Order Model, the $< ROM >$ attribute *subType* needs to be "SciKitLearn" (i.e. *subType = "SciKitLearn"*). The specifications of the ROM *"SciKitLearn"* depends on value assumed by the following sub-node within the main XML node $< ROM >$:

- $< SKLtype >$ ***, vertical bar (|) separated string , required field.***. This nodes contains a string that represents the ROM type that needs to be used. As mentioned, its format is, for example, $< SKLtype >$*mainSKLclass* | *algorithm* $< /SKLtype >$: the first word (before symbol |) represents the main class of algorithms; the second word (after symbol |) represents the specific algorithm.

Based on the $< SKLtype >$ several different algorithms are available. In the following paragraphs a brief explanation and the input requirements are reported for each of them.

10.3.4.1 Linear Models.

The LinearModels' type of algorithms implement generalized linear models. It includes Ridge regression, Bayesian Regression, Lasso and Elastic Net estimators computed with Least Angle Regression and coordinate descent. It also implements Stochastic Gradient Descent related algorithms. In the following, all the linear models available in RAVEN are reported.

10.3.4.1.1 Linear Model: Automatic Relevance Determination regression

The *Automatic Relevance Determination* regressor is a hierarchical Bayesian approach where there are hyperparameters which explicitly represent the relevance of different input features. These relevance hyperparameters determine the range of variation for the parameters relating to a particular input, usually by modelling the width of a zero-mean Gaussian prior on those parameters. If the width of that Gaussian is zero, then those parameters are constrained to be zero, and the corresponding input cannot have any effect on the predictions, therefore making it irrelevant. ARD optimizes these hyperparameters to discover which inputs are relevant. In order to use the *Automatic Relevance Determination* regressor, the user needs to set the sub-node $< SKLtype > ARDRegression < /SKLtype >$. In addition to this XML node, several other need (or not) be inputted:

- $< n_iter >$***, integer, optional field.***. Maximum number of iterations. *Default = 300*;
- $< tol >$***, float, optional field.***. Stop the algorithm if w has converged. *Default = 1.e-3*;
- $< alpha_1 >$***, float, optional field.***. Hyper-parameter : shape parameter for the Gamma distribution prior over the alpha parameter. *Default = 1.e-6*;
- $< alpha_2 >$***, float, optional field.***. Hyper-parameter : inverse scale parameter (rate parameter) for the Gamma distribution prior over the alpha parameter. *Default = 1.e-6*;
- $< lambda_1 >$***, float, optional field.***. Hyper-parameter : shape parameter for the Gamma distribution prior over the lambda parameter. *Default = 1.e-6*;
- $< lambda_2 >$***, float, optional field.***. Hyper-parameter : inverse scale parameter (rate parameter) for the Gamma distribution prior over the lambda parameter. *Default = 1.e-6*;
- $< compute_score >$***, boolean, optional field.***. If True, compute the objective function at each step of the model. *Default = False*;
- $< threshold_lambda >$***, float, optional field.***. Threshold for removing (pruning) weights with high precision from the computation. *Default = 1.e+4*;
- $< fit_intercept >$***, float, optional field.***. whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (e.g. data is expected to be already centered). *Default = True*;
- $< normalize >$***, boolean, optional field.***. If True, the regressors X will be normalized before regression. *Default = False*;
- $< verbose >$***, boolean, optional field.***. Verbose mode when fitting the model. *Default = False*;

10.3.4.1.2 Linear Model: Bayesian ridge regression

The *Bayesian ridge regression* estimates a probabilistic model of the regression problem as described above. The prior for the parameter w is given by a spherical Gaussian:

$$p(w|\lambda) = \mathcal{N}(w|0, \lambda^{-1}\mathbf{I_p}) \tag{1}$$

The priors over α and λ are chosen to be gamma distributions, the conjugate prior for the precision of the Gaussian. The resulting model is called Bayesian Ridge Regression, and is similar to the classical Ridge. The parameters w, α and λ are estimated jointly during the fit of the model. The remaining hyperparameters are the parameters of the gamma priors over α and λ. These are usually chosen to be non-informative. The parameters are estimated by maximizing the marginal log likelihood. In order to use the *Bayesian ridge regression* regressor, the user needs to set the sub-node $< SKLtype > BayesianRidge < /SKLtype >$. In addition to this XML node, several others need (or not) to be inputted:

- $< n_iter >$***, integer, optional field.***. Maximum number of iterations. *Default = 300*;
- $< tol >$***, float, optional field.***. Stop the algorithm if w has converged. *Default = 1.e-3*;
- $< alpha_1 >$***, float, optional field.***. Hyper-parameter : shape parameter for the Gamma distribution prior over the alpha parameter. *Default = 1.e-6*;
- $< alpha_2 >$***, float, optional field.***. Hyper-parameter : inverse scale parameter (rate parameter) for the Gamma distribution prior over the alpha parameter. *Default = 1.e-6*;
- $< lambda_1 >$***, float, optional field.***. Hyper-parameter : shape parameter for the Gamma distribution prior over the lambda parameter. *Default = 1.e-6*;
- $< lambda_2 >$***, float, optional field.***. Hyper-parameter : inverse scale parameter (rate parameter) for the Gamma distribution prior over the lambda parameter. *Default = 1.e-6*;
- $< compute_score >$***, boolean, optional field.***. If True, compute the objective function at each step of the model. *Default = False*;
- $< threshold_lambda >$***, float, optional field.***. Threshold for removing (pruning) weights with high precision from the computation. *Default = 1.e+4*;

- $< fit_intercept >$***, float, optional field.***. whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (e.g. data is expected to be already centered). *Default = True*;

- $< normalize >$***, boolean, optional field.***. If True, the regressors X will be normalized before regression. *Default = False*;

- $< verbose >$***, boolean, optional field.***. Verbose mode when fitting the model. *Default = False*;

10.3.4.1.3 Linear Model: Elastic Net

The *Elastic Net* is a linear regression with combined L1 and L2 priors as regularizer. It minimizes the objective function:

$$1/(2*n_samples)*||y-Xw||_2^2+alpha*l1_ratio*||w||_1+0.5*alpha*(1-l1_ratio)*||w||_2^2 \tag{2}$$

In order to use the *Elastic Net* regressor, the user needs to set the sub-node $< SKLtype > ElasticNet < /SKLtype >$. In addition to this XML node, several others need (or not) to be inputted:

- $< alpha >$***, float, optional field.***. Constant that multiplies the penalty terms. alpha = 0 is equivalent to an ordinary least square, solved by the LinearRegression object. *Default = 1.0*;

- $< l1_ratio >$***, float, optional field.***. The ElasticNet mixing parameter, with $0 <= l1_ratio <= 1$. For $l1_ratio = 0$ the penalty is an L2 penalty. For $l1_ratio = 1$ it is an L1 penalty. For $0 < l1_ratio < 1$, the penalty is a combination of L1 and L2. *Default = 0.5*;

- $< fit_intercept >$***, boolean, optional field.***. Whether the intercept should be estimated or not. If False, the data is assumed to be already centered. *Default = True*;

- $< normalize >$***, boolean, optional field.***. If True, the regressors X will be normalized before regression. *Default = False*;

- $< max_iter >$***, integer, optional field.***. The maximum number of iterations. *Default = 300*;

- $< tol >$***, float, optional field.***. The tolerance for the optimization: if the updates are smaller than tol, the optimization code checks the dual gap for optimality and continues until it is smaller than tol.. *Default = 1.0e-4*;

- $< warm_start >$ ***, boolean, optional field.***. When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution.. *Default = False*;

- $< positive >$ ***, float, optional field.***. When set to True, forces the coefficients to be positive. *Default = False*.

10.3.4.1.4 Linear Model: Elastic Net CV

The *Elastic Net CV* is a linear regression similar to Elastic Net model but with an iterative fitting along a regularization path. The best model is selected by cross-validation.
In order to use the *Elastic Net CV* regressor, the user needs to set the sub-node $< SKLtype >$ $ElasticNetCV$ $< /SKLtype >$. In addition to this XML node, several others need (or not) to be inputted:

- $< l1_ratio >$ ***, float, optional field.***. Float flag between 0 and 1 passed to ElasticNet (scaling between l1 and l2 penalties). For $l1_ratio = 0$ the penalty is an L2 penalty. For $l1_ratio = 1$ it is an L1 penalty. For $0 < l1_ratio < 1$, the penalty is a combination of L1 and L2 This parameter can be a list, in which case the different values are tested by cross-validation and the one giving the best prediction score is used. Note that a good choice of list of values for $l1_ratio$ is often to put more values close to 1 (i.e. Lasso) and less close to 0 (i.e. Ridge), as in [.1, .5, .7, .9, .95, .99, 1]. *Default = 0.5*;

- $< eps >$ ***, float, optional field.***. Length of the path. eps=1e-3 means that $alpha_min/alpha_max = 1e-3$. *Default = 0.001*;

- $< n_alphas >$ ***, integer, optional field.***. Number of alphas along the regularization path, used for each $l1_ratio$. *Default = 100*;

- $< precompute >$ ***, boolean or string, optional field.***. Whether to use a precomputed Gram matrix to speed up calculations. If set to ’auto’ let us decide. The Gram matrix can also be passed as argument. Available are [True — False — ‘auto’ — array-like]. *Default = 1.0*;

- $< max_iter >$ ***, integer, optional field.***. The maximum number of iterations. *Default = 300*;

- $< tol >$ ***, float, optional field.***. The tolerance for the optimization: if the updates are smaller than tol, the optimization code checks the dual gap for optimality and continues until it is smaller than tol.. *Default = 1.0e-4*;

- $< positive >$ ***, float, optional field.***. When set to True, forces the coefficients to be positive. *Default = False*.

10.3.4.1.5 Linear Model: Least Angle Regression model

The *Least Angle Regression model* (LARS) is a regression algorithm for high-dimensional data. LARS algorithm provides a means of producing an estimate of which variables to include, as well as their coefficients, when a response variable is determined by a linear combination of a subset of potential covariate.
In order to use the *Least Angle Regression model* regressor, the user needs to set the sub-node $< SKLtype > Lars < /SKLtype >$. In addition to this XML node, several others need (or not) to be inputted:

- $< n_nonzero_coef >$ ***, integer, optional field.***. Target number of non-zero coefficients. *Default = 500*;

- $< fit_intercept >$ ***, boolean, optional field.***. Whether the intercept should be estimated or not. If False, the data is assumed to be already centered. *Default = True*;

- $< verbose >$ ***, boolean, optional field.***. Verbose mode when fitting the model. *Default = False*;

- $< precompute >$ ***, boolean or string, optional field.***. Whether to use a precomputed Gram matrix to speed up calculations. If set to 'auto' let us decide. The Gram matrix can also be passed as argument. Available are [True — False — 'auto' — array-like]. *Default = 1.0*;

- $< normalize >$ ***, boolean, optional field.***. If True, the regressors X will be normalized before regression. *Default = False*;

- $< eps >$ ***, float, optional field.***. The machine-precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems. Unlike the tol parameter in some iterative optimization-based algorithms, this parameter does not control the tolerance of the optimization. *Default =2.22e-16*;

- $< fit_path >$ ***, boolean, optional field.***. f True the full path is stored in the coef_path_ attribute. If you compute the solution for a large problem or many targets, setting fit_path to False will lead to a speedup, especially with a small alpha. *Default = True*.

10.3.4.1.6 Linear Model: Cross-validated Least Angle Regression model

The *Cross-validated Least Angle Regression model* is a regression algorithm for high-dimensional data. It is similar to LARS method, but the best model is selected by cross-validation.
In order to use the *Cross-validated Least Angle Regression model* regressor, the user needs to set the sub-node $< SKLtype > LarsCV < /SKLtype >$. In addition to this XML node, several others need (or not) to be inputted:

- $< fit_intercept >$ ***, boolean, optional field.***. Whether the intercept should be estimated or not. If False, the data is assumed to be already centered. *Default = True*;
- $< verbose >$ ***, boolean, optional field.***. Verbose mode when fitting the model. *Default = False*;
- $< normalize >$ ***, boolean, optional field.***. If True, the regressors X will be normalized before regression. *Default = False*;
- $< precompute >$ ***, boolean or string, optional field.***. Whether to use a precomputed Gram matrix to speed up calculations. If set to 'auto' let us decide. The Gram matrix can also be passed as argument. Available are [True — False — 'auto' — array-like]. *Default = 1.0*;
- $< max\_ iter >$ ***, integer, optional field.***. The maximum number of iterations. *Default = 300*;
- $< max\_ n\_ alphas >$ ***, integer, optional field.***. The maximum number of points on the path used to compute the residuals in the cross-validation. *Default = 1000*;
- $< eps >$ ***, float, optional field.***. The machine-precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems. Unlike the tol parameter in some iterative optimization-based algorithms, this parameter does not control the tolerance of the optimization. *Default =2.22e-16*;

10.3.4.1.7 Linear Model trained with L1 prior as regularizer (aka the Lasso)

pass

10.3.4.1.8 Lasso linear model with iterative fitting along a regularization path

pass

10.3.4.1.9 Lasso model fit with Least Angle Regression pass

10.3.4.1.10 Cross-validated Lasso, using the LARS algorithm pass

10.3.4.1.11 Lasso model fit with Lars using BIC or AIC for model selection pass

10.3.4.1.12 Ordinary least squares Linear Regression pass

10.3.4.1.13 Logistic Regression pass

10.3.4.1.14 Multi-task Lasso model trained with L1/L2 mixed-norm as regularizer pass

10.3.4.1.15 Multi-task ElasticNet model trained with L1/L2 mixed-norm as regularizer pass

10.3.4.1.16 Orthogonal Mathching Pursuit model (OMP) pass

10.3.4.1.17 Cross-validated Orthogonal Mathching Pursuit model (OMP) pass

10.3.4.1.18 Passive Aggressive Classifier pass

10.3.4.1.19 Passive Aggressive Regressor pass

10.3.4.1.20 Perceptron pass

10.3.4.1.21 Randomized Lasso pass

10.3.4.1.22 Randomized Logistic Regression pass

10.3.4.1.23 Linear least squares with l2 regularization pass

10.3.4.1.24 Classifier using Ridge regression pass

10.3.4.1.25 Ridge classifier with built-in cross-validation pass

10.3.4.1.26 Ridge regression with built-in cross-validation pass

10.3.4.1.27 Linear classifiers (SVM, logistic regression, a.o.) with SGD training pass

10.3.4.1.28 Linear model fitted by minimizing a regularized empirical loss with SGD pass

10.3.4.1.29 Compute Least Angle Regression or Lasso path using LARS algorithm pass

10.3.4.1.30 Compute Lasso path with coordinate descent pass

10.3.4.1.31 Stabiliy path based on randomized Lasso estimates pass

10.3.4.1.32 Gram Orthogonal Matching Pursuit (OMP) pass

10.3.4.2 Support Vector Machines.

The LinearModels' type of algorithms implement generalized linear models. It includes Ridge regression, Bayesian Regression, Lasso and Elastic Net estimators computed with

Least Angle Regression and coordinate descent. It also implements Stochastic Gradient Descent related algorithms. In the following, all the linear models available in RAVEN are reported.

10.3.4.2.1 Linear Support Vector Classifier

pass

10.3.4.2.2 C-Support Vector Classification

pass

10.3.4.2.3 Nu-Support Vector Classification

pass

10.3.4.2.4 Support Vector Regression

pass

10.3.4.3 Multi Class.

Multiclass classification means a classification task with more than two classes; e.g., classify a set of images of fruits which may be oranges, apples, or pears. Multiclass classification makes the assumption that each sample is assigned to one and only one label: a fruit can be either an apple or a pear but not both at the same time. In the following, all the multi-class models available in RAVEN are reported.

10.3.4.3.1 One-vs-the-rest (OvR) multiclass/multilabel strategy

The *One-vs-the-rest (OvR) multiclass/multilabel strategy*, also known as one-vs-all, consists in fitting one classifier per class. For each classifier, the class is fitted against all the other classes. In addition to its computational efficiency (only n_classes classifiers are needed), one advantage of this approach is its interpretability. Since each class is represented by one and one classifier only, it is possible to gain knowledge about the class by inspecting its corresponding classifier. This is the most commonly used strategy and is a fair default choice.
In order to use the *One-vs-the-rest (OvR) multiclass/multilabel* classifier, the user needs to set the sub-node $< SKLtype > multiClass \mid OneVsRestClassifier < /SKLtype >$. In addition to this XML node, several others need (or not) to be inputted:

- $< estimator >$ **, boolean, required field.**. An estimator object implementing fit and one of decision_function or predict_proba. This XML node needs to contain the attributes:
  - **estimatorType**, *required string attribute*, this attribute an other reduced order mode type that needs to be used for the construction of the multi-class algorithms; each sub-sequential node depends on the chosen ROM;

10.3.4.3.2 One-vs-one multiclass strategy

The *One-vs-one multiclass strategy* consists in fitting one classifier per class pair. At prediction time, the class which received the most votes is selected. Since it requires to fit n_classes * (n_classes - 1) / 2 classifiers, this method is usually slower than one-vs-the-rest, due to its O(n_classes$\hat{2}$) complexity. However, this method may be advantageous for algorithms such as kernel algorithms which do not scale well with n_samples. This is because each individual learning problem only involves a small subset of the data whereas, with one-vs-the-rest, the complete dataset is used n_classes times.
In order to use the *One-vs-one multiclass* classifier, the user needs to set the sub-node $< SKLtype > multiClass | OneVsOneClassifier < /SKLtype >$. In addition to this XML node, several others need (or not) to be inputted:

- $< estimator >$ **, boolean, required field.**. An estimator object implementing fit and one of decision_function or predict_proba. This XML node needs to contain the attributes:
  - **estimatorType**, *required string attribute*, this attribute an other reduced order mode type that needs to be used for the construction of the multi-class algorithms; each sub-sequential node depends on the chosen ROM;

10.3.4.3.3 Error-Correcting Output-Code multiclass strategy

The *Error-Correcting Output-Code multiclass strategy* consists in representing each class with a binary code (an array of 0s and 1s). At fitting time, one binary classifier per bit in the code book is fitted. At prediction time, the classifiers are used to project new points in the class space and the class closest to the points is chosen. The main advantage of these strategies is that the number of classifiers used can be controlled by the user, either for compressing the model ($0 < code_size < 1$) or for making the model more robust to errors ($code_size > 1$).
In order to use the *Error-Correcting Output-Code multiclass* classifier, the user needs to set the sub-node $< SKLtype > multiClass | OutputCodeClassifier < /SKLtype >$. In addition to this XML node, several others need (or not) to be inputted:

- $< estimator >$ ***, xml node, required field.***. An estimator object implementing fit and one of decision_function or predict_proba. This XML node needs to contain the attributes:
  - **estimatorType**, *required string attribute*, this attribute an other reduced order mode type that needs to be used for the construction of the multi-class algorithms; each sub-sequential node depends on the chosen ROM;
- $< code_size >$ ***, float, required field.***. ercentage of the number of classes to be used to create the code book. A number between 0 and 1 will require fewer classifiers than one-vs-the-rest. A number greater than 1 will require more classifiers than one-vs-the-rest.

10.3.4.4 Naive Bayes.

Naive Bayes methods are a set of supervised learning algorithms based on applying Bayes' theorem with the "naive" assumption of independence between every pair of features. Given a class variable y and a dependent feature vector x_1 through x_n, Bayes' theorem states the following relationship:

$$P(y \mid x_1, \dots, x_n) = \frac{P(y) P(x_1, \dots x_n \mid y)}{P(x_1, \dots, x_n)} \tag{3}$$

Using the naive independence assumption that

$$P(x_i | y, x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n) = P(x_i | y), \tag{4}$$

for all i, this relationship is simplified to

$$P(y \mid x_1, \dots, x_n) = \frac{P(y) \prod_{i=1}^{n} P(x_i \mid y)}{P(x_1, \dots, x_n)} \tag{5}$$

Since $P(x_1, \dots, x_n)$ is constant given the input, we can use the following classification rule:

$$P(y \mid x_1, \dots, x_n) \propto P(y) \prod_{i=1}^{n} P(x_i \mid y) \Downarrow \tag{6}$$

$$\hat{y} = \arg\max_y P(y) \prod_{i=1}^{n} P(x_i \mid y), \tag{7}$$

and we can use Maximum A Posteriori (MAP) estimation to estimate $P(y)$ and $P(x_i \mid y)$; the former is then the relative frequency of class y in the training set. The different naive

Bayes classifiers differ mainly by the assumptions they make regarding the distribution of $P(x_i \mid y)$. In spite of their apparently over-simplified assumptions, naive Bayes classifiers have worked quite well in many real-world situations, famously document classification and spam filtering. They require a small amount of training data to estimate the necessary parameters. (For theoretical reasons why naive Bayes works well, and on which types of data it does, see the references below.) Naive Bayes learners and classifiers can be extremely fast compared to more sophisticated methods. The decoupling of the class conditional feature distributions means that each distribution can be independently estimated as a one dimensional distribution. This in turn helps to alleviate problems stemming from the curse of dimensionality. On the flip side, although naive Bayes is known as a decent classifier, it is known to be a bad estimator, so the probability outputs from predict_proba are not to be taken too seriously. In the following, all the Naive Bayes available in RAVEN are reported.

10.3.4.4.1 Gaussian Naive Bayes

The *Gaussian Naive Bayes strategy* implements the Gaussian Naive Bayes algorithm for classification. The likelihood of the features is assumed to be Gaussian:

$$P(x_i \mid y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right) \tag{8}$$

The parameters σ_y and μ_y are estimated using maximum likelihood.
In order to use the *Gaussian Naive Bayes strategy*, the user needs to set the sub-node $< SKLtype > naiveBayes \mid GaussianNB < /SKLtype >$. No additional XML nodes are needed to be inputted.

10.3.4.4.2 Multinomial Naive Bayes

The *Multinomial Naive Bayes* implements the naive Bayes algorithm for multinomially distributed data, and is one of the two classic naive Bayes variants used in text classification (where the data are typically represented as word vector counts, although tf-idf vectors are also known to work well in practice). The distribution is parametrized by vectors $\theta_y = (\theta_{y1}, \ldots, \theta_{yn})$ for each class y, where n is the number of features (in text classification, the size of the vocabulary) and θ_{yi} is the probability $P(x_i \mid y)$ of feature i appearing in a sample belonging to class y. The parameters θ_y is estimated by a smoothed version of maximum likelihood, i.e. relative frequency counting:

$$\hat{\theta}_{yi} = \frac{N_{yi} + \alpha}{N_y + \alpha n} \tag{9}$$

where $N_{yi} = \sum_{x \in T} x_i$ is the number of times feature i appears in a sample of class y in the training set T, and $N_y = \sum_{i=1}^{|T|} N_{yi}$ is the total count of all features for class y. The smoothing priors $\alpha \geq 0$ accounts for features not present in the learning samples and prevents zero probabilities in further computations. Setting $\alpha = 1$ is called Laplace smoothing, while $\alpha < 1$ is called Lidstone smoothing.
In order to use the *Multinomial Naive Bayes* strategy, the user needs to set the sub-node $< SKLtype > naiveBayes \mid MultinomialNB < /SKLtype >$. In addition to this XML node, several others need (or not) to be inputted:

- $< alpha >$***, float, optional field.***. Additive (Laplace/Lidstone) smoothing parameter (0 for no smoothing). *Default = 1.0*;
- $< fit\_prior >$***, boolean, required field.***. Whether to learn class prior probabilities or not. If false, a uniform prior will be used. *Default = False*;
- $< class\_prior >$***, array-like float (n_classes), optional field.***. Prior probabilities of the classes. If specified the priors are not adjusted according to the data. *Default = None*.

10.3.4.4.3 Bernoulli Naive Bayes

The *Bernoulli Naive Bayes* implements the naive Bayes training and classification algorithms for data that is distributed according to multivariate Bernoulli distributions; i.e., there may be multiple features but each one is assumed to be a binary-valued (Bernoulli, boolean) variable. Therefore, this class requires samples to be represented as binary-valued feature vectors; if handed any other kind of data, a *Bernoulli Naive Bayes* instance may binarize its input (depending on the binarize parameter). The decision rule for Bernoulli naive Bayes is based on

$$P(x_i \mid y) = P(i \mid y) x_i + (1 - P(i \mid y))(1 - x_i) \tag{10}$$

which differs from multinomial NB's rule in that it explicitly penalizes the non-occurrence of a feature i that is an indicator for class y, where the multinomial variant would simply ignore a non-occurring feature. In the case of text classification, word occurrence vectors (rather than word count vectors) may be used to train and use this classifier. *Bernoulli Naive Bayes* might perform better on some datasets, especially those with shorter documents. It is advisable to evaluate both models, if time permits.
In order to use the *Bernoulli Naive Bayes* strategy, the user needs to set the sub-node $< SKLtype > naiveBayes \mid BernoulliNB < /SKLtype >$. In addition to this XML node, several others need (or not) to be inputted:

- $< alpha >$ ***, float, optional field.***. Additive (Laplace/Lidstone) smoothing parameter (0 for no smoothing). *Default = 1.0*;
- $< binarize >$***, float, optional field.***. Threshold for binarizing (mapping to booleans) of sample features. If None, input is presumed to already consist of binary vectors.. *Default = None*;
- $< fit_prior >$ ***, boolean, optional field.***. Whether to learn class prior probabilities or not. If false, a uniform prior will be used. *Default = False*;
- $< class_prior >$ ***, array-like float (n_classes), optional field.***. Prior probabilities of the classes. If specified the priors are not adjusted according to the data. *Default = None*.

10.3.4.5 Neighbors.

The *Neighbors* class provides functionality for unsupervised and supervised neighbors-based learning methods. Unsupervised nearest neighbors is the foundation of many other learning methods, notably manifold learning and spectral clustering. Supervised neighbors-based learning comes in two flavors: classification for data with discrete labels, and regression for data with continuous labels. The principle behind nearest neighbor methods is to find a predefined number of training samples closest in distance to the new point, and predict the label from these. The number of samples can be a user-defined constant (k-nearest neighbor learning), or vary based on the local density of points (radius-based neighbor learning). The distance can, in general, be any metric measure: standard Euclidean distance is the most common choice. Neighbors-based methods are known as non-generalizing machine learning methods, since they simply “remember” all of its training data (possibly transformed into a fast indexing structure such as a Ball Tree or KD Tree.). In the following, all the Neighbors’ models available in RAVEN are reported.

10.3.4.5.1 Nearest Neighbors

The *Nearest Neighbors* implements unsupervised nearest neighbors learning. It acts as a uniform interface to three different nearest neighbors algorithms: BallTree, KDTree, and a brute-force algorithm.

In order to use the *Nearest Neighbors* strategy, the user needs to set the sub-node $< SKLtype > neighbors | NearestNeighbors < /SKLtype >$. In addition to this XML node, several others need (or not) to be inputted:

- $< n_neighbors >$***, integer, optional field.***. Number of neighbors to use by default for k_neighbors queries.. *Default = 5*;
- $< radius >$***, float, optional field.***. Range of parameter space to use by default for :meth‘radius_neighbors‘ queries. *Default = 1.0*;
- $< algorithm >$***, string, optional field.***. Algorithm used to compute the nearest neighbors:
  - *ball_ tree* will use BallTree;
  - *kd_ tree* will use KDtree;
  - *brute* will use a brute-force search;
  - *auto* will attempt to decide the most appropriate algorithm based on the values passed to fit method.

  NB. fitting on sparse input will override the setting of this parameter, using brute force. *Default = auto*;
- $< leaf\_ size >$***, integer, optional field.***. Leaf size passed to BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem. *Default = 30*;
- $< p >$***, integer, optional field.***. Parameter for the Minkowski metric. When $p = 1$, this is equivalent to using manhattan distance (L1), and euclidean distance (L2) for $p = 2$. For arbitrary p, minkowski distance (L_p) is used. *Default = 2*.

10.3.4.5.2 K Neighbors Classifier

The *K Neighbors Classifier* is a type of instance-based learning or non-generalizing learning: it does not attempt to construct a general internal model, but simply stores instances of the training data. Classification is computed from a simple majority vote of the nearest neighbors of each point: a query point is assigned the data class which has the most representatives within the nearest neighbors of the point. It implements learning based on the k nearest neighbors of each query point, where k is an integer value specified by the user.
In order to use the *K Neighbors Classifier*, the user needs to set the sub-node $< SKLtype >$ $neighbors \mid KNeighborsClassifier < /SKLtype >$. In addition to this XML node, several others need (or not) to be inputted:

- $< n\_neighbors >$***, integer, optional field.***. Number of neighbors to use by default for k_neighbors queries.. *Default = 5*;
- $< weights >$***, string, optional field.***. Weight function used in prediction. Possible values:
  - *uniform* : uniform weights. All points in each neighborhood are weighted equally;
  - *distance* : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.

  Default = uniform;
- $< radius >$***, float, optional field.***. Range of parameter space to use by default for :meth'radius_neighbors' queries. *Default = 1.0*;
- $< algorithm >$***, string, optional field.***. Algorithm used to compute the nearest neighbors:
  - *ball_tree* will use BallTree;
  - *kd_tree* will use KDtree;
  - *brute* will use a brute-force search;
  - *auto* will attempt to decide the most appropriate algorithm based on the values passed to fit method.

  NB. fitting on sparse input will override the setting of this parameter, using brute force. *Default = auto*;
- $< metric >$***, string, optional field.***. the distance metric to use for the tree. The default metric is minkowski, and with p=2 is equivalent to the standard Euclidean metric. *Default = minkowski*;
- $< leaf\_size >$***, integer, optional field.***. Leaf size passed to BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem. *Default = 30*;
- $< p >$***, integer, optional field.***. Parameter for the Minkowski metric. When $p = 1$, this is equivalent to using manhattan distance (L1), and euclidean distance (L2) for $p = 2$. For arbitrary p, minkowski distance (L_p) is used. *Default = 2*.

10.3.4.5.3 Radius Neighbors Classifier

The *Radius Neighbors Classifier* is a type of instance-based learning or non-generalizing learning: it does not attempt to construct a general internal model, but simply stores instances of the training data. Classification is computed from a simple majority vote of the nearest neighbors of each point: a query point is assigned the data class which has the most representatives within the nearest neighbors of the point. It implements learning based on the number of neighbors within a fixed radius r of each training point, where r is a floating-point value specified by the user.

In order to use the *Radius Neighbors Classifier*, the user needs to set the sub-node $< SKLtype > neighbors | RadiusNeighbors < /SKLtype >$. In addition to this XML node, several others need (or not) to be inputted:

- $< n\_neighbors >$ ***, integer, optional field.***. Number of neighbors to use by default for k_neighbors queries.. *Default = 5*;
- $< weights >$ ***, string, optional field.***. Weight function used in prediction. Possible values:
  - *uniform* : uniform weights. All points in each neighborhood are weighted equally;
  - *distance* : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.

  Default = uniform;
- $< radius >$ ***, float, optional field.***. Range of parameter space to use by default for :meth‘radius_neighbors‘ queries. *Default = 1.0*;
- $< algorithm >$ ***, string, optional field.***. Algorithm used to compute the nearest neighbors:
  - *ball_tree* will use BallTree;
  - *kd_tree* will use KDtree;
  - *brute* will use a brute-force search;
  - *auto* will attempt to decide the most appropriate algorithm based on the values passed to fit method.

NB. fitting on sparse input will override the setting of this parameter, using brute force. *Default = auto*;

- $< metric >$ ***, string, optional field.***. the distance metric to use for the tree. The default metric is minkowski, and with p=2 is equivalent to the standard Euclidean metric. *Default = minkowski*;

- $< leaf\_ size >$ ***, integer, optional field.***. Leaf size passed to BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem. *Default = 30*;

- $< p >$ ***, integer, optional field.***. Parameter for the Minkowski metric. When $p = 1$, this is equivalent to using manhattan distance (L1), and euclidean distance (L2) for $p = 2$. For arbitrary p, minkowski distance (L_p) is used. *Default = 2*;

- $< outlier\_ label >$ ***, integer, optional field.***. Label, which is given for outlier samples (samples with no neighbors on given radius). If set to None, ValueError is raised, when outlier is detected. *Default = None*.

10.3.4.5.4 K Neighbors Regressor

The *K Neighbors Regressor* can be used in cases where the data labels are continuous rather than discrete variables. The label assigned to a query point is computed based the mean of the labels of its nearest neighbors. It implements learning based on the k nearest neighbors of each query point, where k is an integer value specified by the user.
In order to use the *K Neighbors Regressor*, the user needs to set the sub-node $< SKLtype >$ $neighbors \mid KNeighborsRegressor < /SKLtype >$. In addition to this XML node, several others need (or not) to be inputted:

- $< n_neighbors >$ ***, integer, optional field.***. Number of neighbors to use by default for k_neighbors queries.. *Default = 5*;

- $< weights >$ ***, string, optional field.***. Weight function used in prediction. Possible values:

  - *uniform* : uniform weights. All points in each neighborhood are weighted equally;

  - *distance* : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.

  Default = uniform;

- $< radius >$ ***, float, optional field.***. Range of parameter space to use by default for :meth'radius_neighbors' queries. *Default = 1.0*;

- $< algorithm >$ ***, string, optional field.***. Algorithm used to compute the nearest neighbors:

  - *ball_tree* will use BallTree;
  - *kd_tree* will use KDtree;
  - *brute* will use a brute-force search;
  - *auto* will attempt to decide the most appropriate algorithm based on the values passed to fit method.

  NB. fitting on sparse input will override the setting of this parameter, using brute force. *Default = auto*;

- $< metric >$ ***, string, optional field.***. the distance metric to use for the tree. The default metric is minkowski, and with p=2 is equivalent to the standard Euclidean metric. *Default = minkowski*;

- $< leaf\_size >$ ***, integer, optional field.***. Leaf size passed to BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem. *Default = 30*;

- $< p >$ ***, integer, optional field.***. Parameter for the Minkowski metric. When $p = 1$, this is equivalent to using manhattan distance (L1), and euclidean distance (L2) for $p = 2$. For arbitrary p, minkowski distance (L_p) is used. *Default = 2*.

10.3.4.5.5 Radius Neighbors Regressor

The *Radius Neighbors Regressor* can be used in cases where the data labels are continuous rather than discrete variables. The label assigned to a query point is computed based the mean of the labels of its nearest neighbors. It implements learning based on the neighbors within a fixed radius r of the query point, where r is a floating-point value specified by the

user.
In order to use the *Radius Neighbors Regressor*, the user needs to set the sub-node $< SKLtype > neighbors \mid RadiusNeighborsRegressor < /SKLtype >$. In addition to this XML node, several others need (or not) to be inputted:

- $< n\_neighbors >$ ***, integer, optional field.***. Number of neighbors to use by default for k_neighbors queries.. *Default = 5*;
- $< weights >$ ***, string, optional field.***. Weight function used in prediction. Possible values:
  - *uniform* : uniform weights. All points in each neighborhood are weighted equally;
  - *distance* : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.

  Default = uniform;
- $< radius >$ ***, float, optional field.***. Range of parameter space to use by default for :meth'radius_neighbors' queries. *Default = 1.0*;
- $< algorithm >$ ***, string, optional field.***. Algorithm used to compute the nearest neighbors:
  - *ball_tree* will use BallTree;
  - *kd_tree* will use KDtree;
  - *brute* will use a brute-force search;
  - *auto* will attempt to decide the most appropriate algorithm based on the values passed to fit method.

  NB. fitting on sparse input will override the setting of this parameter, using brute force. *Default = auto*;
- $< metric >$ ***, string, optional field.***. the distance metric to use for the tree. The default metric is minkowski, and with p=2 is equivalent to the standard Euclidean metric. *Default = minkowski*;

- $< leaf\_size >$ ***, integer, optional field.***. Leaf size passed to BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem. *Default = 30*;

- $< p >$ ***, integer, optional field.***. Parameter for the Minkowski metric. When $p = 1$, this is equivalent to using manhattan distance (L1), and euclidean distance (L2) for $p = 2$. For arbitrary p, minkowski distance (L_p) is used. *Default = 2*;

- $< outlier\_label >$ ***, integer, optional field.***. Label, which is given for outlier samples (samples with no neighbors on given radius). If set to None, ValueError is raised, when outlier is detected. *Default = None*.

10.3.4.5.6 Nearest Centroid Classifier

The *Nearest Centroid classifier* is a simple algorithm that represents each class by the centroid of its members. It also has no parameters to choose, making it a good baseline classifier. It does, however, suffer on non-convex classes, as well as when classes have drastically different variances, as equal variance in all dimensions is assumed.
In order to use the *Nearest Centroid Classifier*, the user needs to set the sub-node $< SKLtype > neighbors \mid NearestCentroid < /SKLtype >$. In addition to this XML node, another might be inputted:

- $< n\_neighbors >$ ***, float, optional field.***. Threshold for shrinking centroids to remove features. *Default = None*.

10.3.4.6 Quadratic Discriminant Analysis.

The *Quadratic Discriminant Analysis* is a classifier with a quadratic decision boundary, generated by fitting class conditional densities to the data and using Bayes' rule.The model fits a Gaussian density to each class.
In order to use the *Quadratic Discriminant Analysis Classifier*, the user needs to set the sub-node $< SKLtype > qda \mid QDA < /SKLtype >$. In addition to this XML node, several others need (or not) to be inputted:

- $< priors >$ ***, array-like (n_classes), optional field.***. Priors on classes. *Default = None*;

- $< reg\_ param >$ ***, float, optional field.***. Regularizes the covariance estimate as (1-reg_param)*Sigma + reg_param*Identity(n_features). *Default = 0.0*.

10.3.4.7 Tree.

Decision Trees (DTs) are a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features.

- Some advantages of decision trees are:
- Simple to understand and to interpret. Trees can be visualized.
- Requires little data preparation. Other techniques often require data normalization, dummy variables need to be created and blank values to be removed. Note however that this module does not support missing values.
- The cost of using the tree (i.e., predicting data) is logarithmic in the number of data points used to train the tree.
- Able to handle both numerical and categorical data. Other techniques are usually specialized in analyzing datasets that have only one type of variable.
- Able to handle multi-output problems.
- Uses a white box model. If a given situation is observable in a model, the explanation for the condition is easily explained by boolean logic. By contrast, in a black box model (e.g., in an artificial neural network), results may be more difficult to interpret.
- Possible to validate a model using statistical tests. That makes it possible to account for the reliability of the model.
- Performs well even if its assumptions are somewhat violated by the true model from which the data were generated.

The disadvantages of decision trees include:

- Decision-tree learners can create over-complex trees that do not generalise the data well. This is called overfitting. Mechanisms such as pruning (not currently supported), setting the minimum number of samples required at a leaf node or setting the maximum depth of the tree are necessary to avoid this problem.
- Decision trees can be unstable because small variations in the data might result in a completely different tree being generated. This problem is mitigated by using decision trees within an ensemble.
- The problem of learning an optimal decision tree is known to be NP-complete under several aspects of optimality and even for simple concepts. Consequently, practical decision-tree learning algorithms are based on heuristic algorithms such as the greedy algorithm where locally optimal decisions are made at each node. Such algorithms cannot guarantee to return the globally optimal decision tree. This can be mitigated by training multiple trees in an ensemble learner, where the features and samples are randomly sampled with replacement.
- There are concepts that are hard to learn because decision trees do not express them easily, such as XOR, parity or multiplexer problems.
- Decision tree learners create biased trees if some classes dominate. It is therefore recommended to balance the dataset prior to fitting with the decision tree.

In the following, all the linear models available in RAVEN are reported.

10.3.4.7.1 Decision Tree Classifier

The *Decision Tree Classifier* is a classifier that is based on the decision tree logic.
In order to use the *Decision Tree Classifier*, the user needs to set the sub-node $< SKLtype > tree \mid DecisionTreeClassifier < /SKLtype >$. In addition to this XML node, several others need (or not) to be inputted:

- $< criterion >$ ***, string, optional field.***. The function to measure the quality of a split. Supported criteria are “gini” for the Gini impurity and “entropy” for the information gain. *Default = gini*;
- $< splitter >$ ***, string, optional field.***. The strategy used to choose the split at each node. Supported strategies are “best” to choose the best split and “random” to choose the best random split. *Default = best*;

- $< max\_features >$***, int, float or string, optional field.***. The number of features to consider when looking for the best split:
  - If int, then consider max_features features at each split.
  - If float, then max_features is a percentage and int(max_features * n_features) features are considered at each split.
  - If "auto", then max_features=sqrt(n_features);
  - If "sqrt", then max_features=sqrt(n_features);
  - If "log2", then max_features=log2(n_features);
  - If None, then max_features=n_features.

  NB. The search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features.
  Default = None;
- $< max\_depth >$***, integer, optional field.***. The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples. Ignored if max_samples_leaf is not None. *Default = None*;
- $< min\_samples\_split >$***, integer, optional field.***. The minimum number of samples required to split an internal node. *Default = 2*;
- $< min\_samples\_leaf >$***, integer, optional field.***. The minimum number of samples required to be at a leaf node. *Default = 1*;
- $< max\_leaf\_nodes >$***, integer, optional field.***. Grow a tree with max_leaf_nodes in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes. If not None then max_depth will be ignored. *Default = None*.

10.3.4.7.2 Decision Tree Regressor

The *Decision Tree Regressor* is a Regressor that is based on the decision tree logic.
In order to use the *Decision Tree Regressor*, the user needs to set the sub-node $< SKLtype >$ $tree \mid DecisionTreeRegressor < /SKLtype >$. In addition to this XML node, several others need (or not) to be inputted:

- $< criterion >$ ***, string, optional field.***. The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain. *Default = gini*;

- $< splitter >$ ***, string, optional field.***. The strategy used to choose the split at each node. Supported strategies are "best" to choose the best split and "random" to choose the best random split. *Default = best*;

- $< max\_features >$ ***, int, float or string, optional field.***. The number of features to consider when looking for the best split:

  - If int, then consider max_features features at each split.
  - If float, then max_features is a percentage and int(max_features * n_features) features are considered at each split.
  - If "auto", then max_features=sqrt(n_features);
  - If "sqrt", then max_features=sqrt(n_features);
  - If "log2", then max_features=log2(n_features);
  - If None, then max_features=n_features.

  NB. The search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features.
  Default = None;

- $< max\_depth >$ ***, integer, optional field.***. The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples. Ignored if max_samples_leaf is not None. *Default = None*;

- $< min\_samples\_split >$ ***, integer, optional field.***. The minimum number of samples required to split an internal node. *Default = 2*;

- $< min\_samples\_leaf >$ ***, integer, optional field.***. The minimum number of samples required to be at a leaf node. *Default = 1*;

- $< max\_leaf\_nodes >$ ***, integer, optional field.***. Grow a tree with max_leaf_nodes in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes. If not None then max_depth will be ignored. *Default = None*.

10.3.4.7.3 Extra Tree Classifier

The *Extra Tree Classifier* is an extremely randomized tree classifier. Extra-trees differ from classic decision trees in the way they are built. When looking for the best split to separate the samples of a node into two groups, random splits are drawn for each of the max_features randomly selected features and the best split among those is chosen. When max_features is set 1, this amounts to building a totally random decision tree.
In order to use the *Extra Tree Classifier*, the user needs to set the sub-node $< SKLtype >$ $tree \mid ExtraTreeClassifier < /SKLtype >$. In addition to this XML node, several others need (or not) to be inputted:

- $< criterion >$ ***, string, optional field.***. The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain. *Default = gini*;

- $< splitter >$ ***, string, optional field.***. The strategy used to choose the split at each node. Supported strategies are "best" to choose the best split and "random" to choose the best random split. *Default = best*;

- $< max\_features >$ ***, int, float or string, optional field.***. The number of features to consider when looking for the best split:

  - If int, then consider max_features features at each split.
  - If float, then max_features is a percentage and int(max_features * n_features) features are considered at each split.
  - If "auto", then max_features=sqrt(n_features);
  - If "sqrt", then max_features=sqrt(n_features);
  - If "log2", then max_features=log2(n_features);
  - If None, then max_features=n_features.

  NB. The search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features.
  Default = None;

- $< max\_depth >$ ***, integer, optional field.***. The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples. Ignored if max_samples_leaf is not None. *Default = None*;

- $< \ min\_ \ samples\_ \ split \ >$ ***, integer, optional field.***. The minimum number of samples required to split an internal node. *Default = 2*;

- $< min\_ \, samples\_ \, leaf >$ ***, integer, optional field.***. The minimum number of samples required to be at a leaf node. *Default = 1*;

- $< max\_ \, leaf\_ \, nodes >$ ***, integer, optional field.***. Grow a tree with max_leaf_nodes in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes. If not None then max_depth will be ignored. *Default = None*.

10.3.4.7.4 Extra Tree Regressor

The *Extra Tree Regressor* is an extremely randomized tree regressor. Extra-trees differ from classic decision trees in the way they are built. When looking for the best split to separate the samples of a node into two groups, random splits are drawn for each of the max_features randomly selected features and the best split among those is chosen. When max_features is set 1, this amounts to building a totally random decision tree.
In order to use the *Extra Tree Regressor*, the user needs to set the sub-node $< SKLtype >$ $tree \mid ExtraTreeRegressor \ < \ /SKLtype \ >$. In addition to this XML node, several others need (or not) to be inputted:

- $< \ criterion \ >$ ***, string, optional field.***. The function to measure the quality of a split. Supported criteria are “gini” for the Gini impurity and “entropy” for the information gain. *Default = gini*;

- $< \ splitter \ >$ ***, string, optional field.***. The strategy used to choose the split at each node. Supported strategies are “best” to choose the best split and “random” to choose the best random split. *Default = best*;

- $< max\_ \, features >$ ***, int, float or string, optional field.***. The number of features to consider when looking for the best split:

  - If int, then consider max_features features at each split.
  - If float, then max_features is a percentage and int(max_features * n_features) features are considered at each split.
  - If “auto”, then max_features=sqrt(n_features);
  - If “sqrt”, then max_features=sqrt(n_features);

  - If “log2”, then max_features=log2(n_features);
  - If None, then max_features=n_features.

  NB. The search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features.
  Default = None;

- $< max\_depth >$***, integer, optional field.***. The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples. Ignored if max_samples_leaf is not None. *Default = None*;

- $< min\_ samples\_ split >$ ***, integer, optional field.***. The minimum number of samples required to split an internal node. *Default = 2*;

- $< min\_ samples\_ leaf >$***, integer, optional field.***. The minimum number of samples required to be at a leaf node. *Default = 1*;

- $< max\_ leaf\_ nodes >$***, integer, optional field.***. Grow a tree with max_leaf_nodes in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes. If not None then max_depth will be ignored. *Default = None*.

10.3.4.8 Gaussian Process.

Gaussian Processes for Machine Learning (GPML) is a generic supervised learning method primarily designed to solve regression problems. The advantages of Gaussian Processes for Machine Learning are:

- The prediction interpolates the observations (at least for regular correlation models);

- The prediction is probabilistic (Gaussian) so that one can compute empirical confidence intervals and exceedance probabilities that might be used to refit (online fitting, adaptive fitting) the prediction in some region of interest;

- Versatile: different linear regression models and correlation models can be specified. Common models are provided, but it is also possible to specify custom models provided they are stationary.

The disadvantages of Gaussian Processes for Machine Learning include:

- It is not sparse. It uses the whole samples/features information to perform the prediction;
- It loses efficiency in high dimensional spaces – namely when the number of features exceeds a few dozens. It might indeed give poor performance and it loses computational efficiency;
- Classification is only a post-processing, meaning that one first need to solve a regression problem by providing the complete scalar float precision output y of the experiment one attempt to model.

In order to use the *Gaussian Process Regressor*, the user needs to set the sub-node $< SKLtype > GaussianProcess \mid GaussianProcess < /SKLtype >$. In addition to this XML node, several others need (or not) to be inputted:

- $< regr >$***, string, optional field.***. A regression function returning an array of outputs of the linear regression functional basis. The number of observations n_samples should be greater than the size p of this basis. Available built-in regression models are: ’constant’, ’linear’, ’quadratic’. *Default = constant*;
- $< corr >$***, string, optional field.***. A stationary autocorrelation function returning the autocorrelation between two points x and x’. Default assumes a squared-exponential autocorrelation model. Built-in correlation models are: ’absolute_exponential’, ’squared_exponential’,’generalized_exponential’, ’cubic’, ’linear’. *Default = squared_exponential*;
- $< beta0 >$***, float, array-like, optional field.***. The regression weight vector to perform Ordinary Kriging (OK). *Default = Universal Kriging*;
- $< storage\_ mode >$***, string, optional field.***. A string specifying whether the Cholesky decomposition of the correlation matrix should be stored in the class (storage_mode = ‘full’) or not (storage_mode = ‘light’). *Default = full*;
- $< verbose >$***, boolean, optional field.***. boolean specifying the verbose level. *Default = False*;
- $< theta0 >$***, float, array-like, optional field.***. An array with shape (n_features,) or (1,). The parameters in the autocorrelation model. If thetaL and thetaU are also specified, theta0 is considered as the starting point for the maximum likelihood estimation of the best set of parameters. *Default = [1e-1]*;

- $< thetaL >$, ***float, array-like, optional field.***. An array with shape matching theta0's. Lower bound on the autocorrelation parameters for maximum likelihood estimation. *Default = None*;

- $< thetaU >$, ***float, array-like, optional field.***. An array with shape matching theta0's. Upper bound on the autocorrelation parameters for maximum likelihood estimation. *Default = None*;

- $< normalize >$, ***boolean, optional field.***. Input X and observations y are centered and reduced wrt means and standard deviations estimated from the n_samples observations provided. *Default = True*;

- $< nugget >$, ***float, optional field.***. Introduce a nugget effect to allow smooth predictions from noisy data.The nugget is added to the diagonal of the assumed training covariance; in this way it acts as a Tikhonov regularization in the problem. In the special case of the squared exponential correlation function, the nugget mathematically represents the variance of the input values. *Default = 10. * MACHINE_EPSILON*;

- $< optimizer >$, ***string, optional field.***. A string specifying the optimization algorithm to be used. Available optimizers are: 'fmin_cobyla', 'Welch'. *Default = fmin_cobyla*;

- $< random\_ start >$, ***integer, optional field.***. The number of times the Maximum Likelihood Estimation should be performed from a random starting point. The first MLE always uses the specified starting point (theta0), the next starting points are picked at random according to an exponential distribution (log-uniform on [thetaL, thetaU]). *Default = 1*;

- $< random\_ state >$, ***integer, optional field.***. Seed of the internal random number generator. *Default = random.*

Example:

```
<Simulation>
  ...
  <Models>
    ...
   <ROM name='***' subType='SciKitLearn'>
```

```
      <Features>***,***</Features>
      <SKLtype>linear_model|LinearRegression</SKLtype>
      <Target>***</Target>
      <fit_intercept>***</fit_intercept>
      <normalize>***</normalize>
    </ROM>
     ...
  </Models>
   ...
</Simulation>
------------------------------------------------------------
```

10.4 External Model

As the name suggests, an external model is an entity that is embedded in the RAVEN code at run time. This object allows the user to create a python module that is going to be treated as a predefined internal model object. In other words, the **External Model** is going to be treated by RAVEN as a normal external Code (e.g. it is going to be called in order to compute a whatever quantity, based on a whatever input).
The specifications of an External Model must be defined within the XML block $< ExternalModel >$. This XML node needs to contain the attributes:

- **name**, *required string attribute*, user-defined name of this External Model. 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);
- **subType**, *required string attribute*, This string must be kept empty;
- **ModuleToLoad**, *required string attribute*, file name with its absolute or relative path. NB. If a relative path is specified, it must be noticed it is relative with respect to where the user runs the code.

In order to make the RAVEN code aware of the variables the user is going to manipulate/use in its own python Module, the variables need to be specified in the $< ExternalModel >$ input block. The user needs to input, within this block, only the variables that RAVEN needs to be aware of (i.e. the variables are going to directly be used by the code) and not the local variables that the user does not want to, for example, store in a RAVEN internal object. These variables are inputted within consecutive XML blocks called $< variable >$:

- $< variable >$, string, required parameter. In the body of this XML node, the user needs to specify the name of the variable. This variable needs to match a variable used/defined in the external python model.

When the external function variables are defined, at run time, RAVEN initialize those and take track of their values during the simulation. Each variable defined in the $< ExternalModel >$ block is available in the module (each method implemented) as a python "self".
In the External Python module, the user can implement all the methods that are needed for the functionality of the model, but only the following methods are going, if present, called by the framework:

- ***def _readMoreXML**, OPTIONAL METHOD*, This method can be implemented by the user if the XML input that belongs to this External Model needs to be extended to contain other information. The information read needs to be stored in "self" in order to be available to all the other methods (e.g. if the user needs to add a couple of newer XML nodes with information needed by the algorithm implemented in the "run" method);

- ***def initialize**, OPTIONAL METHOD*, Initialization method. In this function the user can implement all the actions need to be performed at the initialization stage;

- ***def createNewInput**, OPTIONAL METHOD*, Method to create a new input with the information coming from the RAVEN framework. In this function the user can retrieve the information coming from the RAVEN framework, during the employment of a calculation flow, and use them to construct a new input that is going to be transferred to the "run" method;

- ***def run**, REQUIRED METHOD*, This is the actual location where the user needs to implement the model action (e.g. resolution of a set of equations, etc.). This function is going to receive the Input(or Inputs) generated either by the External Model "createNewInput" method or the internal RAVEN one;

In the following sub-sections, all the methods are going to be analyzed in detail.

10.4.1 Method: def _ readMoreXML.

As already mentioned, the **readMoreXML** method can be implemented by the user if the XML input that belongs to this External Model needs to be extended to contain other information. The information read needs to be stored in “self” in order to be available to all the other methods (e.g. if the user needs to add a couple of newer XML nodes with information needed by the algorithm implemented in the “run” method). If this method is implemented in the **External Model**, RAVEN is going to call it when the node $< ExternalModel >$ is found parsing XML input file. The method receives from RAVEN an attribute of type “xml.etree.ElementTree”, containing all the sub-nodes and attribute of the XML block $< ExternalModel >$.
In the following an example is reported:

```
-------------------------------------
<Simulation>
  ...
  <Models>
    ...
    <ExternalModel name=’AnExtModule’
                                subType=’’
                                ModuleToLoad=’path_to_external_module’>
       <variable>sigma</variable>
       <variable>rho</variable>
       <variable>outcome</variable>
       <!--
          here we define other nodes RAVEN does not read
             automatically
          => We need to implement, in the external module
             ``AnExtModule''
          the readMoreXML method
        -->
        <newNodeWeNeedToRead>
           whatNeedsToBeRead
        </newNodeWeNeedToRead>
    </ExternalModel>
    ...
  </Models>
  ...
```

```
</Simulation>
```

```
def _readMoreXML(self,xmlNode):
  # the xmlNode is passed in by RAVEN framework
  # <newNodeWeNeedToRead> is unknown (in the RAVEN framework)
  # we have to read it on our own
  # get the node
  ourNode = xmlNode.find(‘‘newNodeWeNeedToRead’’)
  # get the information in the node
  self.ourNewVariable = ourNode.text
  # end function
```

10.4.2 Method: def initialize.

The **initialize** method can be implemented in the **External Model** in order to initialize some variables needed by it. For example, it can be used to compute a quantity needed by the "run" method before performing the actual calculation). If this method is implemented in the **External Model**, RAVEN is going to call it at the initialization stage of each "Step" (see section 12. RAVEN will communicate, thorough a set of method attributes, all the information that are generally needed to perform a initialization:

- runInfo, a dictionary containing information regarding how the calculation is set up (e.g. number of processors, etc.). It contains the following attributes:
  - *DefaultInputFile*, Default input file to use
  - *SimulationFiles*, the xml input file
  - *ScriptDir*, the location of the pbs script interfaces
  - *FrameworkDir*, the directory where the framework is located
  - *WorkingDir*, the directory where the framework should be running
  - *TempWorkingDir*, the temporary directory where a simulation step is run
  - *NumMPI*, the number of mpi process by run
  - *NumThreads*, Number of Threads by run

  - *numProcByRun*, Total number of core used by one run (number of threads by number of mpi)
  - *batchSize*, number of contemporaneous runs
  - *ParallelCommand*, the command that should be used to submit jobs in parallel (mpi)
  - *numNode*, number of nodes
  - *procByNode*, number of processors by node
  - *totalNumCoresUsed*, total number of cores used by driver
  - *quequingSoftware*, quequing software name
  - *stepName*, the name of the step currently running
  - *precommand*, Add to the front of the command that is run
  - *postcommand*, Added after the command that is run
  - *delSucLogFiles*, If a simulation (code run) has not failed, delete the relative log file (if True)
  - *deleteOutExtension*, If a simulation (code run) has not failed, delete the relative output files with the listed extension (comma separated list, for example: 'e,r,txt')
  - *mode*, Running mode. Curently the only modes supported are pbsdsh and mpi
  - *expectedTime*, How long the complete input is expected to run
  - *logfileBuffer*, logfile buffer size in bytes
- inputs, a list of all the inputs that have been specified in the "Step" is using this model.

In the following an example is reported:

```
def initialize(self,runInfo,inputs):
 # Let's suppose we just need to initialize some variables
  self.sigma = 10.0
  self.rho   = 28.0
  # end function
```

10.4.3 Method: def createNewInput.

The **createNewInput** method can be implemented by the user to create a new input with the information coming from the RAVEN framework. In this function the user can retrieve the information coming from the RAVEN framework, during the employment of a calculation flow, and use them to construct a new input that is going to be transferred to the "run" method. The new input created needs to be returned to RAVEN (i.e. "return NewInput"). RAVEN communicates, thorough a set of method attributes, all the information that are generally needed to create a new input: myInput,samplerType,**Kwargs

- inputs, python list, a list of all the inputs that have been defined in the "Step" is using this model;
- samplerType, string, the type of Sampler, if a Sampling strategy is employed; None otherwise;
- Kwargs, dictionary, a dictionary containing several informations (that can change based on the "Step" type). If a Sampling strategy is employed, this dictionary contains another one identified by the keyword "SampledVars", in which the variables perturbed by the sampler are reported;

NB. If the "Step" that is using this Model has as input(s) an object of main class type "Datas" (see section **??**), the internal "createNewInput" method is going to convert it in a dictionary of values.
In the following an example is reported:

```
def createNewInput(self,inputs,samplerType,**Kwargs):
  # in here the actual createNewInput of the
  # model is implemented
  if samplerType == ``MonteCarlo'':
    avariable = inputs[``something'']*inputs[``something2'']
  else:
    avariable = inputs[``something'']/inputs[``something2'']
  return avariable*Kwargs[``SampledVars''][``aSampledVar'']
```

10.4.4 Method: def run.

As stated previously, the only method the MUST be present in an External Module is the **run** function. In this function , the user needs to implement the algorithm that RAVEN has to execute. The **run** method is generally called after having inquired the “createNewInput” method (either the internal or the user-implemented one). The only attribute this method is going to receive by is a Python list of inputs (the inputs coming from the “createNewInput” method). If the user wants RAVEN to collect the results of this method, the outcomes of interest need to be stored in “self”. NB. RAVEN is trying to collect the values of the variables listed in the XML block $< ExternalModel >$ only.
In the following an example is reported:

```
def run(self,Input):
  # in here the actual run of the
  # model is implemented
  input = Input[0]
  self.outcome = self.sigma*self.rho*input[‘‘whatEver’’]
```

10.5 PostProcessor

A Post-Processor (PP) can be considered as an Action performed on a set of data or other type of objects. Most of the post-processors contained in RAVEN, employ a mathematical operation on the data given as “input”/ RAVEN supports several different types of PPs. Currently in RAVEN the following Post-Processors are available:

- **BasicStatistics;**
- **ComparisonStatistics;**
- **SafestPoint;**
- **LimitSurface;**
- **PrintCSV;**
- **LoadCsvIntoInternalObject.**

The specifications of this Model must be defined within the XML block $< PostProcessor >$. This XML node needs to contain the attributes:

- **name**, *required string attribute*, user-defined name of this Model. 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);
- **subType**, *required string attribute*, in this attribute the user defines which of the post-processors needs to be used, choosing among the previously reported types. Obviously, this choice conditions the subsequent the required and/or optional $< PostProcessor >$ sub nodes.

As already mentioned, all the types and meaning of the remaining sub-nodes depend on the post-processor type specified in the attribute *subType*. In the following sections the specifications of each type are reported.

10.5.0.1 BasicStatistics.

The **BasicStatistics** post-processor is the container of the algorithms to compute all the most important statistical quantities.
In order to use the *BasicStatistics* post-processor, the user needs to set the sub-node $< subType > BasicStatistics < /subType >$. In addition to this XML node, several others need (or not) to be inputted:

- $< what >$***, comma separated string, required field.***. List of quantities need to be computed. Currently the quantities available are:
  - **covariance matrix,** Covariance matrix;
  - **NormalizedSensitivity,** Matrix of normalized sensitivity coefficients;
  - **sensitivity,** Matrix of sensitivity coefficients;
  - **pearson,** Matrix of sensitivity coefficients;
  - **expectedValue**, Expected value;
  - **sigma,** Standard deviation;
  - **variationCoefficient,** Coefficient of variation (sigma/expected value);
  - **variance,** Variance;
  - **skewness,** Skewness;

  - **kurtois,** Kurtois;
  - **median,** Data median;
  - **percentile,** 95 percentile.

  If all the quantities need to be computed, the user can input in the body of $< what >$ the string “all”.

- $< parameters >$ ***, comma separated string, required field.***. List of the parameters on which the previous operations need to be applied on (e.g., massFlow, Temperature);

- $< methodsToRun >$ ***, comma separated string, optional field.***. The method names of an external Function that need to be run before computing any of the predefined quantities If this XML node is inputted, the $< Assembler >$ node must be present. *Default = None*;

- $< Assembler >$***, xml node, required field.*** This xml node contains a “list” of objects that are optional for the functionality of the **BasicStatistics** post-processor. The objects must be listed with a rigorous syntax that, except for the xml node tag, is common among all the objects. Each of these sub-nodes must contain 2 attributes that are used to map those within the simulation framework:

  - **class**, *required string attribute*, it is the main “class” the listed object is from;
  - **type**, *required string attribute*, it is the object identifier or sub-type.

  The **BasicStatistics** post-processor approach optionally accepts the following object type:

  - $< Function >$***, string, required field.***.The body of this xml block needs to contain the name of an External Function defined within the $< Functions >$ main block (see section 9). This object needs to containt the methods listed in the node $< methodsToRun >$.;

Example: ——————————————————-

```
<Simulation>
  ...
  <Models>
    ...
    <PostProcessor name=”∗∗∗” subType=’BasicStatistics’ debug=’true’>
```

```
       <!-- => you can here specify what type of figure of
          merit
          you need to compute
         <what>expectedValue,sigma,variance,kurtois,pearson,covariance</wha
        -->
       <what>expectedValue</what>
       <parameters>x01,x02</parameters>
       <methodsToRun>failureProbability</methodsToRun>
    </PostProcessor>
     ...
  <Models>
  ...
<Simulation>
----------------------------------------------
```

10.5.0.2 ComparisonStatistics.

To be finalized.

10.5.0.3 SafestPoint.

The **SafestPoint** post-processor provides the coordinates of the farthest point from the limit surface that is given as an input. The safest point coordinates are expected values of the coordinates of the farthest points from the limit surface in the space of the ”controllable” variables based on the probability distributions of the ”non-controllable” variables. The term ”controllable” identifies those variables that are under control during the system operation, while the ”non-controllable” variables are stochastic parameters affecting the system behaviour randomly.
The ”SafestPoint” post-processor requires the set of points belonging to the limit surface, which must be given as an input. The ”Assembler” subsection requires the probability distributions of both ”controllable” and ”non-controllable” variables.
The sampling method used by the ”SafestPoint” is a ”value” or ”CDF” grid. At present only the ”equal” grid type is available.

- $< Assembler >$: the probability distributions of the ”controllable” and ”non-controllable” variables are required.

- $< controllable >$: the list of the controllable variables is given here. Each variable is associated with its name and the two items below:
  - $< distribution >$: the name of the probability distribution associated with the controllable variable is specified here.
  - $< grid >$: type, number of steps and tolerance of the sampling grid are defined here.
- $< non - controllable >$: the list of the non-controllable variables is given here. Each variable is associated with its name and the two items below:
  - $< distribution >$: the name of the probability distribution associated with the non-controllable variable is specified here.
  - $< grid >$: type, number of steps and tolerance of the sampling grid are defined here.

Example:

```
<Simulation>
    . . .
    <Models>
        . . .
  <PostProcessor name=’SP’ subType=’SafestPoint’>
      <Assembler>
        <Distribution class = ’Distributions’ type =
            ’Normal’>x1_dst</Distribution>
        <Distribution class = ’Distributions’ type =
            ’Normal’>x2_dst</Distribution>
        <Distribution class = ’Distributions’ type =
            ’Normal’>gammay_dst</Distribution>
      </Assembler>
      <controllable>
        <variable name = ’x1’>
          <distribution>x1_dst</distribution>
          <grid type = ’value’ steps = ’20’>1</grid>
        </variable>
        <variable name = ’x2’>
          <distribution>x2_dst</distribution>
          <grid type = 'value' steps = '20'>1</grid>
        </variable>
        </controllable>
        <non−controllable>
          <variable name = 'gammay'>
            <distribution>gammay_dst</distribution>
            <grid type = 'value' steps = '20'>2</grid>
          </variable>
        </non−controllable>
    </PostProcessor>
      . . .
   </Models>
   . . .
</Simulation>
----------------------------------------------
```

10.5.0.4 LimitSurface.

The **LimitSurface** post-processor is aimed to identify the transition zones that determine a change in the status of the system (Limit Surface).
In order to use the *LimitSurface*, the user needs to set the sub-node $< subType > LimitSurface < /subType >$. In addition to this XML node, several others need (or not) to be inputted:

- $< parameters >$ ***, comma separated string, required field.***. List of the parameters that define the uncertain domain and from which the LS needs to be computed;
- $< tollerance >$ ***, float, optional field.***. Absolute value converge tollerance. This value defines the coarsens of the evaluation grid. *Default= 1.e-4*;
- $< Assembler >$***, xml node, required field.*** This xml node contains a "list" of objects that are required (or optional) for the functionality of the Adaptive Sampler. The objects must be listed with a rigorous syntax that, except for the xml node tag, is common among all the objects. Each of these sub-nodes must contain 2 attributes that are used to map those within the simulation framework:
  - **class**, *required string attribute*, it is the main "class" the listed object is from. For example, it can be "Models", "Functions", etc;

- **type**, *required string attribute*, it is the object identifier or sub-type. For example, it can be “ROM”, “External”, etc.

The **LimitSurface** post-processor requires or optionally accepts the following objects’ types:

- $< ROM >$***, string, optional field.***. If inputted, the body of this xml node must contain the name of a ROM defined in the $< Models >$ block (see section 10.3);
- $< Function >$***, string, required field.***.The body of this xml block needs to contain the name of an External Function defined within the $< Functions >$ main block (see section 9). This object represents the boolean function that defines the transition boundaries. This function must implement a method called *__residuumSign(self)*, that returns either -1 or 1, depending on the system conditions (see section 9.

Example:

```
----------------------------------------------
<Simulation>
  ...
  <Models>
    ...
   <PostProcessor name=”computeLimitSurface” subType=’LimitSurface’
      debug=’True’>
      <what>all</what>
      <parameters>x0,y0</parameters>
      <parameters>x0,y0</parameters>
      <Assembler>
         <ROM class=’Models’ type=’ROM’ >Acc</ROM>
         <!--You can add here a ROM defined in Models
            block.If not Present,
                a nearest algorithm is going to be used
          -->
         <Function class=’Functions’ type=’External’
            >goalFunctionForLimitSurface</Function>
      </Assembler>
   </PostProcessor>
    ...
   </Models>
   ...
</Simulation>
-----------------------------------------------------
```

10.5.0.5 PrintCSV.

TO BE MOVED TO STEP “IOSTEP”

10.5.0.6 LoadCsvIntoInternalObject.

TO BE MOVED TO STEP “IOSTEP”

11 OutStream system

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 [2], 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 7) 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.

11.1 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.

11.2 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 [2]. 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 **??**.

11.2.1 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.

11.2.1.1 "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 [2].

- $< label\_format >$, 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 [2] for a list of all the colors available)
  - $< edgecolor >$, string, the figure edge background color (please refer to “matplotlib.figure.Figure” in [2] 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 [2].

  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 [2].

  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 [2].

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 [2].

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 [2].
- $< grid >$, 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 [2].

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

In addition other plot-dependent keywords, reported in section 11.2.1.3, 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, optional parameter, x axis label;
- $< ylabel >$, string, optional parameter, y axis laber;
- $< zlabel >$, string, optional parameter (3D plots only), z axis label;
- $< colorMap >$, string, the parameter needs to be used to define a color map.

As already mentioned, the Plot system accepts as parameter (i.e., x, y, z, colorMap) the **Datas** object only. Considering the structure of "Datas", the parameters are inputted as follow:
$DataObjectName|Input(or)Output|variableName$.
If the "variableName" contains the symbol |, it must be surrounded by brackets:
$DataObjectName|Input(or)Output|(whatever|variableName)$.

11.2.1.3 Predefined Plotting System: 2D/3D

As already mentioned above, the Plotting system provides specialized input structure for several different kind of plots:

- *2 Dimensional plots*:
  - *scatter*. 2 dimensional scatter plot. It used to create a scatter plot of x vs y, where x and y are sequences of numbers of the same length;
  - *line*. 2 dimensional line plot. It used to create a line plot of x vs y, where x and y are sequences of numbers of the same length;
  - *histogram*. 2 dimensional histogram plot. Compute and draw the histogram of x. It must be noticed that this plot accepts only the XML node $< x >$ even if it is considered as 2D plot type;
  - *stem*. 2 dimensional stem plot. A stem plot plots vertical lines at each x location from the baseline to y, and places a marker there;
  - *step*. 2 dimensional step plot;

- *pseudocolor*. 2 dimensional scatter plot. It creates a pseudocolor plot of a two dimensional array. The two dimensional array is built creating a mesh from $< x > and < y >$ data, in conjunction with the data inputted in $< colorMap >$;
- *contour*. 2 dimensional contour plot. It plots the contour lines. Contour plot is built creating a plot from $< x > and < y >$ data, in conjunction with the data inputted in $< colorMap >$;
- *filledContour*. 2 dimensional contour plot. It plots the filled contour leveles. Filled contour plot is built creating a plot from $< x > and < y >$ data, in conjunction with the data inputted in $< colorMap >$;

- *3 Dimensional plots*:
  - *scatter*. 3 dimensional scatter plot. It is used to create a scatter plot of (x,y) vs z, where x, y, z are sequences of numbers of the same length;
  - *line*. 3 dimensional line plot. It used to create a line plot of (x,y) vs z, where x, y, z are sequences of numbers of the same length;
  - *stem*. 3 dimensional stem plot. It creates a 3 Dimensional stem plot of (x,y) vs z;
  - *surface*. 3 dimensional surface plot. Create a surface plot of (x,y) vs z. By default it will be colored in shades of a solid color, but it also supports color mapping;
  - *wireframe*. 3 dimensional wire-frame plot. No color mapping is supported;
  - *tri-surface*. 3 dimensional tri-surface plot. It is a surface plot with automatic triangulation;
  - *contour3D*. 3 dimensional contour plot. It plots the contour lines. Contour plot is built creating a plot from $< x >, < y > and < z >$ data, in conjunction with the data inputted in $< colorMap >$
  - *filledContour3D*. 3 dimensional filled contour plot. It plots the filled contour leveles. Filled contour plot is built creating a plot from $< x >, < y > and < z >$ data, in conjunction with the data inputted in $< colorMap >$;
  - *histogram*. 3 dimensional histogram plot.Compute and draw the histogram of x and y. It must be noticed that this plot accepts only the XML nodes $< x > and < y >$ even if it is considered as 3D plot type

As already mentioned, the settings for each plot type are inputted within the XML block called $< plot >$. The sub-nodes that can be inputted depends on the plot type: each plot

type has its own set of parameters that can be specified.
In the following sub-sections all the options for the plot types listed above are reported.

11.2.2 2D & 3D Scatter plot.

In order to create a "scatter" plot, either 2D or 3D, the user needs to write in the $< type >$ body the keyword "scatter". In order to customize the plot, the user can define the following XML sub-nodes:

- $< s >$***, integer, optional field.***. Size in points$\hat{2}$. *Default* = *20*;
- $< c >$***, string, optional field.***. color or sequence of color. $< c >$ can be a single color format string, or a sequence of color specifications of length N, or a sequence of N numbers to be mapped to colors using the cmap and norm specified via kwargs. Note that $< c >$ should not be a single numeric RGB or RGBA sequence because that is indistinguishable from an array of values to be colormapped. $< c >$ can be a 2-D array in which the rows are RGB or RGBA;
- $< marker >$***, string, optional field.***. Marker type. *Default* = *o*;
- $< alpha >$***, string, optional field.***. The alpha blending value, between 0 (transparent) and 1 (opaque). *Default* = *None* ;
- $< linewidths >$***, string, optional field.***. Widths of lines. Note that this is a tuple, and if you set the linewidths argument you must set it as a sequence of floats. *Default* = *None*;
- $< kwargs >$, 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.scatter" method in [2].

11.2.3 2D & 3D Line plot.

In order to create a "line" plot, either 2D or 3D, the user needs to write in the $< type >$ body the keyword "line". In order to customize the plot, the user can define the following XML sub-nodes:

- $< interpolationType >$***, string, optional field.***. Type of interpolation algorithm to use for the data. Available are "nearest", "linear", "cubic", "multiquadric", "inverse", "gaussian", "Rbflinear", "Rbfcubic", "quintic", "thin_plate". *Default = linear*;

- $< interpPointsX >$***, integer, optional field.***. Number of points need to be used for interpolation of x axis;

- $< interpPointsY >$***, integer, optional field.***. Number of points need to be used for interpolation of y axis (only 3D line plot);

11.2.4 2D & 3D Histogram plot.

In order to create a "histogram" plot, either 2D or 3D, the user needs to write in the $< type >$ body the keyword "histogram". In order to customize the plot, the user can define the following XML sub-nodes:

- $< bins >$***, integer or array_like, optional field.***. Number of bins if an integer is inputted or a sequence of edges if a python list is defined. *Default = 10*;

- $< normed >$***, boolean, optional field.***. if True normalize the histogram to 1. *Default = False*;

- $< weights >$***, sequence, optional field.***. An array of weights, of the same shape as x. Each value in x only contributes its associated weight towards the bin count (instead of 1). If normed is True, the weights are normalized, so that the integral of the density over the range remains 1. *Default = None*;

- $< cumulative >$***, boolean, optional field.***. If True, then a histogram is computed where each bin gives the counts in that bin plus all bins for smaller values. The last bin gives the total number of datapoints. If normed is also True then the histogram is normalized such that the last bin equals 1. If cumulative evaluates to less than 0 (e.g., -1), the direction of accumulation is reversed. In this case, if normed is also True, then the histogram is normalized such that the first bin equals 1. *Default = False* ;

- $< histtype >$***, string, optional field.***. The type of histogram to draw:

  - **bar** is a traditional bar-type histogram. If multiple data are given the bars are aranged side by side;
  - **barstacked** is a bar-type histogram where multiple data are stacked on top of each other;
  - **step** generates a lineplot that is by default unfilled;
  - **stepfilled** generates a lineplot that is by default filled;

  Default = bar;

- $< align >$***, string, optional field.***. Controls how the histogram is plotted.

  - **left** bars are centered on the left bin edge;
  - **mid** bars are centered between the bin edges;
  - **right** bars are centered on the right bin edges;

  Default = mid;

- $< orientation >$***, string, optional field.***. Orientation of the histogram:

  - **horizontal**;
  - **vertical**;

  Default = vertical;

- $< rwidth >$***, float, optional field.***. The relative width of the bars as a fraction of the bin width. *Default = None*;

- $< log >$***, boolean, optional field.***. Set a log scale. *Default = False*;

- $< color >$***, string, optional field.***. Color of the histogram. *Default = blue*;

- $< stacked >$***, boolean, optional field.***. If True, multiple data are stacked on top of each other If False multiple data are aranged side by side if histtype is ‘bar’ or on top of each other if histtype is ‘step’. *Default* = *False*;
- $< kwargs >$, 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.hist” method in [2].

11.2.5 2D & 3D Stem plot.

In order to create a “stem” plot, either 2D or 3D, the user needs to write in the $< type >$ body the keyword “stem”. In order to customize the plot, the user can define the following XML sub-nodes:

- $< linefmt >$***, string, optional field.***. Line style. *Default* = *b-*;
- $< markerfmt >$***, string, optional field.***. Marker format. *Default* = *bo*;
- $< basefmt >$***, string, optional field.***. Base format. *Default* = *r-*;
- $< kwargs >$, 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.stem” method in [2].

11.2.6 2D Step plot

In order to create a 2D “step” plot, the user needs to write in the $< type >$ body the keyword “step”. In order to customize the plot, the user can define the following XML sub-nodes:

- $< where >$***, string, optional field.***. Positioning:
  - **pre**, the interval from x[i] to x[i+1] has level y[i+1];
  - **post**, that interval has level y[i];
  - **mid**, the jumps in y occur half-way between the x-values;

  Default = mid;
- $< kwargs >$, 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.step” method in [2].

11.2.7 2D Pseudocolor plot

In order to create a 2D “pseudocolor” plot, the user needs to write in the $< type >$ body the keyword “pseudocolor”. In order to customize the plot, the user can define the following

XML sub-nodes:

- $< interpolationType >$***, string, optional field.***. Type of interpolation algorithm to use for the data. Available are "nearest", "linear", "cubic", "multiquadric", "inverse", "gaussian", "Rbflinear", "Rbfcubic", "quintic", "thin_plate". *Default = linear*;
- $< interpPointsX >$***, integer, optional field.***. Number of points need to be used for interpolation of x axis;
- $< kwargs >$, 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.pcolor" method in [2].

11.2.8 2D Contour or filledContour plots.

In order to create a 2D "Contour" or "filledContour" plot, the user needs to write in the $< type >$ body the keyword "contour" or "filledContour", respectively. In order to customize the plot, the user can define the following XML sub-nodes:

- $< number\_bins >$***, integer, optional field.***. Number of bins. *Default = 5*;
- $< interpolationType >$***, string, optional field.***. Type of interpolation algorithm to use for the data. Available are "nearest", "linear", "cubic", "multiquadric", "inverse", "gaussian", "Rbflinear", "Rbfcubic", "quintic", "thin_plate". *Default = linear*;

- $< interpPointsX >$***, integer, optional field.***. Number of points need to be used for interpolation of x axis;
- $< kwargs >$, 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.contour” method in [2].

11.2.9 3D Surface Plot.

In order to create a 3D “Surface” plot, the user needs to write in the $< type >$ body the keyword “surface”. In order to customize the plot, the user can define the following XML sub-nodes:

- $< rstride >$***, integer, optional field.***. Array row stride (step size). *Default = 1*;
- $< cstride >$***, integer, optional field.***. Array column stride (step size). *Default = 1*;
- $< cmap >$***, string, optional field.***. Color map. *Default = jet*;
- $< antialiased >$***, boolean, optional field.***. Antialiased rendering. *Default = False*;
- $< linewidth >$***, integer, optional field.***. Widths of lines. Note that this is a tuple, and if you set the linewidths argument you must set it as a sequence of floats. *Default = 0*;
- $< interpolationType >$***, string, optional field.***. Type of interpolation algorithm to use for the data. Available are “nearest”, “linear”, “cubic”, “multiquadric”, “inverse”, “gaussian”, “Rbflinear”, “Rbfcubic”, “quintic”, “thin_plate”. *Default = linear*;

- $< interpPointsX >$***, integer, optional field.***. Number of points need to be used for interpolation of x axis;

- $< interpPointsY >$***, integer, optional field.***. Number of points need to be used for interpolation of y axis;

- $< kwargs >$, 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.contour" method in [2].

11.2.10 3D Wireframe Plot.

In order to create a 3D "Wireframe" plot, the user needs to write in the $< type >$ body the keyword "wireframe". In order to customize the plot, the user can define the following XML sub-nodes:

- $< rstride >$***, integer, optional field.***. Array row stride (step size). *Default = 1*;

- $< cstride >$***, integer, optional field.***. Array column stride (step size). *Default = 1*;

- $< cmap >$***, string, optional field.***. Color map. *Default = jet*;

- $< interpolationType >$***, string, optional field.***. Type of interpolation algorithm to use for the data. Available are "nearest", "linear", "cubic", "multiquadric", "inverse", "gaussian", "Rbflinear", "Rbfcubic", "quintic", "thin_plate". *Default = linear*;

- $< interpPointsX >$***, integer, optional field.***. Number of points need to be used for interpolation of x axis;

- $< interpPointsY >$***, integer, optional field.***. Number of points need to be used for interpolation of y axis;

- $< kwargs >$, 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.contour" method in [2].

11.2.11 3D Tri-surface Plot.

In order to create a 3D "Tri-surface" plot, the user needs to write in the $< type >$ body the keyword "tri-surface". In order to customize the plot, the user can define the following XML sub-nodes:

- $< color >$***, string, optional field.***. Color of the surface patches. *Default* = *b*;

- $< shade >$***, boolean, optional field.***. Apply shading. *Default* = *False*;

- $< cmap >$***, string, optional field.***. Color map. *Default* = *jet*;

- $< kwargs >$, 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.contour” method in [2].

11.2.12 3D Contour or filledContour plots.

In order to create a 3D “Contour” or “filledContour” plot, the user needs to write in the $< type >$ body the keyword “contour3D” or “filledContour3D”, respectively. In order to customize the plot, the user can define the following XML sub-nodes:

- $< number\_bins >$***, integer, optional field.***. Number of bins. *Default = 5*;
- $< interpolationType >$***, string, optional field.***. Type of interpolation algorithm to use for the data. Available are “nearest”, “linear”, “cubic”, “multiquadric”, “inverse”, “gaussian”, “Rbflinear”, “Rbfcubic”, “quintic”, “thin_plate”. *Default = linear*;
- $< interpPointsX >$***, integer, optional field.***. Number of points need to be used for interpolation of x axis;
- $< interpPointsY >$***, integer, optional field.***. Number of points need to be used for interpolation of y axis;
- $< kwargs >$, 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.contour” method in [2].

11.2.13 Example XML input.

```
<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 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:

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

References

[1]

[2] J. Hunter and M. D. et al., “Matplotlib documentation,” 2003.

v1.28

Idaho National Laboratory