

MANUAL

INL/EXT-16-38178

Revision 1

Printed June 2016

RAVEN Theory Manual

Andrea Alfonsi, Cristian Rabiti, Diego Mandelli, Joshua Cogliati, Congjian Wang,
Paul W. Talbot, Daniel P. Maljovec, Curtis Smith

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.

Approved for unlimited release.



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-16-38178
Revision 1
Printed June 2016

RAVEN Theory Manual

Andrea Alfonsi
Cristian Rabiti
Diego Mandelli
Joshua Cogliati
Congjian Wang
Paul W. Talbot
Daniel P. Maljovec
Curtis Smith

Contents

1	Introduction	9
2	RAVEN overview	10
2.1	RAVEN initial development	10
2.2	Software infrastructure	11
2.2.1	Outlines	11
2.2.2	Probabilistic and Parametric Framework	11
2.2.3	Distribution	13
2.2.4	Sampler	14
2.2.4.1	Forward Samplers	14
2.2.4.2	Dynamic Event Tree Samplers	15
2.2.4.3	Adaptive Samplers	16
2.2.5	Models	18
2.2.5.1	Code	18
2.2.5.2	External Model	18
2.2.5.3	Post-Processor	19
2.2.5.4	Reduced Order Model	20
2.2.6	Simulation Environment	21
3	RAVEN Concepts	23
3.1	RAVEN Mathematical Background	23
3.1.1	System and Control Model	23
3.1.1.1	Splitting Approach for the Simulation of the Control System ...	24
3.1.1.2	Definition of the Monitored Variable Space	25
3.1.1.3	Definition of the Auxiliary Variable Space	25
3.1.2	Dynamic Systems Stochastic Modeling	26
3.1.2.1	General system of equations and variable classification	26
3.1.2.2	Probabilistic Nature of the Parameters Characterizing the Equation	27
3.1.2.3	Variables Subject to Random Motion	28
3.1.2.4	Discontinuously and Stochastically varying variables	29
3.1.3	Formulation of the equation set in a statistical framework	32
3.1.4	The Chapman-Kolmogorov Equation	33
3.1.4.1	Drift Process	34
3.1.4.2	Diffusion Process	34
3.1.4.3	Jumps in Continuous Space	34
3.1.4.4	Jumps in Discrete Space	35
3.2	RAVEN Entities and Analysis Flow	35
3.3	Raven Input Structure	39
4	Manual Formats	41
5	Manual Structure	42
6	Run a Single Instance of a Code and Load the Outputs	45
6.1	Single Run using the OutStream system for printing and create basic plots	45

6.2	Single Run using the OutStream System to Sub-plot and Selectively print.	50
7	Forward Sampling Strategies	54
7.1	Monte-Carlo	54
7.1.1	Monte-Carlo	55
7.1.2	Monte-Carlo sampling through RAVEN	55
7.2	Grid	65
7.2.1	Grid theory introduction	65
7.2.2	Grid sampling through RAVEN	66
7.3	Stratified	76
7.3.1	Stratified theory introduction	77
7.3.2	Stratified sampling through RAVEN	77
7.4	Sparse Grid Collocation	86
7.4.1	Sparse Grid Collocation Theory Introduction	88
7.4.1.1	Generalized Polynomial Chaos	88
7.4.1.2	Polynomial Index Set Construction	89
7.4.1.3	Anisotropy	91
7.4.1.4	Stochastic Collocation	92
7.4.1.5	Smolyak Sparse Grids	94
7.4.2	Sparse Grid Collocation sampling through RAVEN	95
8	Adaptive Sampling Strategies	104
8.1	Limit Surface Search Method	104
8.1.1	Limit Surface Theory	105
8.1.1.1	Limit Surface Search Algorithm	110
8.1.1.2	Acceleration through Multi-grid Approach	113
8.1.2	Limit Surface Search sampling through RAVEN	119
9	Sampling from Restart	127
10	Reduced Order Modeling	131
10.1	Reduced Order Modeling: Theory	132
10.1.1	Gaussian Process Models	134
10.1.2	Support Vector Machines	135
10.1.3	KNN Classifier and KNR Regressor	136
10.1.4	Multi-Dimensional Interpolation	136
10.1.4.1	Shepard's Method	137
10.1.4.2	Multi-Dimensional Spline	137
10.2	Reduced Order Modeling through RAVEN	138
11	Statistical Analysis	149
11.1	Statistical Analysis Theory	150
11.1.1	Expected Value	150
11.1.2	Standard Deviation and Variance	151
11.1.3	Skewness	152
11.1.4	Excess Kurtosis	153
11.1.5	Median	154

11.1.6	Percentile	154
11.1.7	Covariance and Correlation Matrices	155
11.1.8	Variance-Dependent Sensitivity Matrix	155
11.2	Statistical Analysis through RAVEN	156
12	RAVEN Theory by way of Examples: Data Mining	162
12.1	Data Mining Theory	162
12.1.1	Clustering	162
12.1.2	Hierarchical Methodologies	164
12.1.3	<i>K</i> -Means	164
12.1.4	Mean-Shift	165
12.1.5	DBSCAN	166
12.1.6	Dimensionality Reduction	167
12.1.7	Dimensionality Reduction: Linear Algorithms	168
12.1.8	Principal Component Analysis (PCA)	168
12.1.9	Multidimensional Scaling (MDS)	168
12.2	Data Mining through RAVEN	169
Appendices		176
A	Running RAVEN	176
References		177

1 Introduction

RAVEN [1] [2] [3] [4] is a software framework that allows the user to perform parametric and stochastic analysis based on the response of complex system codes. The initial development was designed to provide dynamic probabilistic risk analysis capabilities (DPRA) to the thermal-hydraulic code RELAP-7 [5], currently under development at Idaho National Laboratory (INL). Now, RAVEN is not only a framework to perform DPRA but it is a multi-purpose stochastic and uncertainty quantification platform, capable of communicating with any system code.

The provided Application Programming Interfaces (APIs) allow RAVEN to interact with any code as long as all the parameters that need to be perturbed are accessible by input files or via python interfaces. RAVEN is capable of investigating system response and explore input space using various sampling schemes such as Monte Carlo, grid, or Latin Hypercube. However, RAVEN strength lies in its system feature discovery capabilities such as: constructing limit surfaces, separating regions of the input space leading to system failure, and using dynamic supervised learning techniques.

The development of RAVEN started in 2012 when, within the Nuclear Energy Advanced Modeling and Simulation (NEAMS) program [6], the need of a modern risk evaluation framework arose. RAVEN's principal assignment is to provide the necessary software and algorithms in order to employ the concepts developed by the Risk Informed Safety Margin Characterization (RISMC) Pathway. RISMC is one of the pathways defined within the Light Water Reactor Sustainability (LWRS) program [7].

The goal of the RISMC approach is the identification not only of the frequency of an event which can potentially lead to system failure, but also the proximity (or lack thereof) to key safety-related events: the safety margin. 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. peak pressure in a pipe) is exceeded under certain conditions. Most of the capabilities, implemented having Reactor Excursion and Leak Analysis Program v.7 (RELAP-7) as a principal focus, are easily deployable to other system codes. For this reason, several side activates have been employed (e.g. RELAP5-3D [8], any Multiphysics Object Oriented Simulation Environment-based App, etc.) or are currently ongoing for coupling RAVEN with several different software. The aim of this document is to provide a set of detailed examples that can help the user to become familiar with the RAVEN code and to understand the theory behind key concepts.

2 RAVEN overview

RAVEN is a multi-purpose software framework designed to perform parametric and stochastic analysis based on the response of complex system codes. The initial development was designed to provide dynamic probabilistic risk analysis capabilities (DPRA) to the thermal-hydraulic code RELAP-7, currently under development at Idaho National Laboratory (INL).

Now, RAVEN is not only a framework to perform DPRA but it is a multi-purpose stochastic and uncertainty quantification platform, capable of communicating with any system code through the provided Application Programming Interfaces (APIs). These APIs allow RAVEN to interact with any code as long as all the parameters that need to be perturbed are accessible through input files or via `Python` interfaces. RAVEN is capable of investigating the system response as well as the input space using sampling schemes, such as Monte Carlo, Grid, Latin Hyper Cube. However, RAVEN strength is the set of system feature discovery capabilities. This section presents an overview of the software capabilities and their implementation schemes.

2.1 RAVEN initial development

The development of RAVEN started in 2012, when, within the NEAMS program, the need to provide a modern risk evaluation framework became stronger. RAVEN's principal assignment is to provide the necessary software infrastructure 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 specifically identifying the frequency of an event potentially leading to a system failure, but the closeness (or not) to key safety-related events. This approach may be used 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. as peak pressure in a pipe) is exceeded under certain conditions. The initial development of RAVEN has focused on providing dynamic risk assessment capability to RELAP-7, currently under development at the INL. Most of the capabilities implemented for RELAP-7 are easily deployable for other system codes. For this reason, several side activities are currently coupling RAVEN with several different software (e.g. RELAP5-3D, any MOOSE-based App, etc.). The goal of this document is to provide a set of commented examples that can help the user to become familiar with the RAVEN code usage.

2.2 Software infrastructure

2.2.1 Outlines

RAVEN has been developed in a highly modular way to enable easy integration of different programming languages (i.e., C++ Python) and, as already mentioned, coupling with any system code. RAVEN is composed of three main software systems that can operate either in coupled or stand-alone mode:

- Control logic system
- Graphical user interface
- Probabilistic and parametric framework.

The control logic system and the Graphical User Interface are currently available for RELAP-7 only. For this reason, attention is focused on the probabilistic and parametric framework.

2.2.2 Probabilistic and Parametric Framework

The probabilistic and parametric framework represents the core of the RAVEN analysis capabilities. The main idea behind the design of the system is the creation of a multi-purpose framework characterized by high flexibility with respect to the possible performable analysis. The framework is capable of constructing the analysis/calculation flow at run-time, interpreting the user-defined instructions and assembling the different analysis tasks following a user specified scheme. In order to achieve such flexibility, combined with reasonably fast development, a programming language naturally suitable for this kind of approach was needed: `Python`.

Hence, RAVEN is coded in `Python` and it is characterized by an object-oriented design. The core of the analysis performable through RAVEN is represented by a set of basic components (objects) the user can combine, in order to create the desired analysis flow. A list of these components and a summary of their most important functionalities are reported as follows:

- **Distribution:** In order to explore the input/output space, RAVEN requires the capability to perturb the input space (e.g., initial conditions and/or model coefficients of a system code). The input space is generally characterized by probability distribution functions (PDFs), which might need to be considered when a perturbation is applied. In this respect, a large library of PDFs is available.
- **Sampler:** A proper strategy to sample the input space is fundamental for the optimization of the computational time. In RAVEN, a “sampler” employs a unique perturbation strategy that

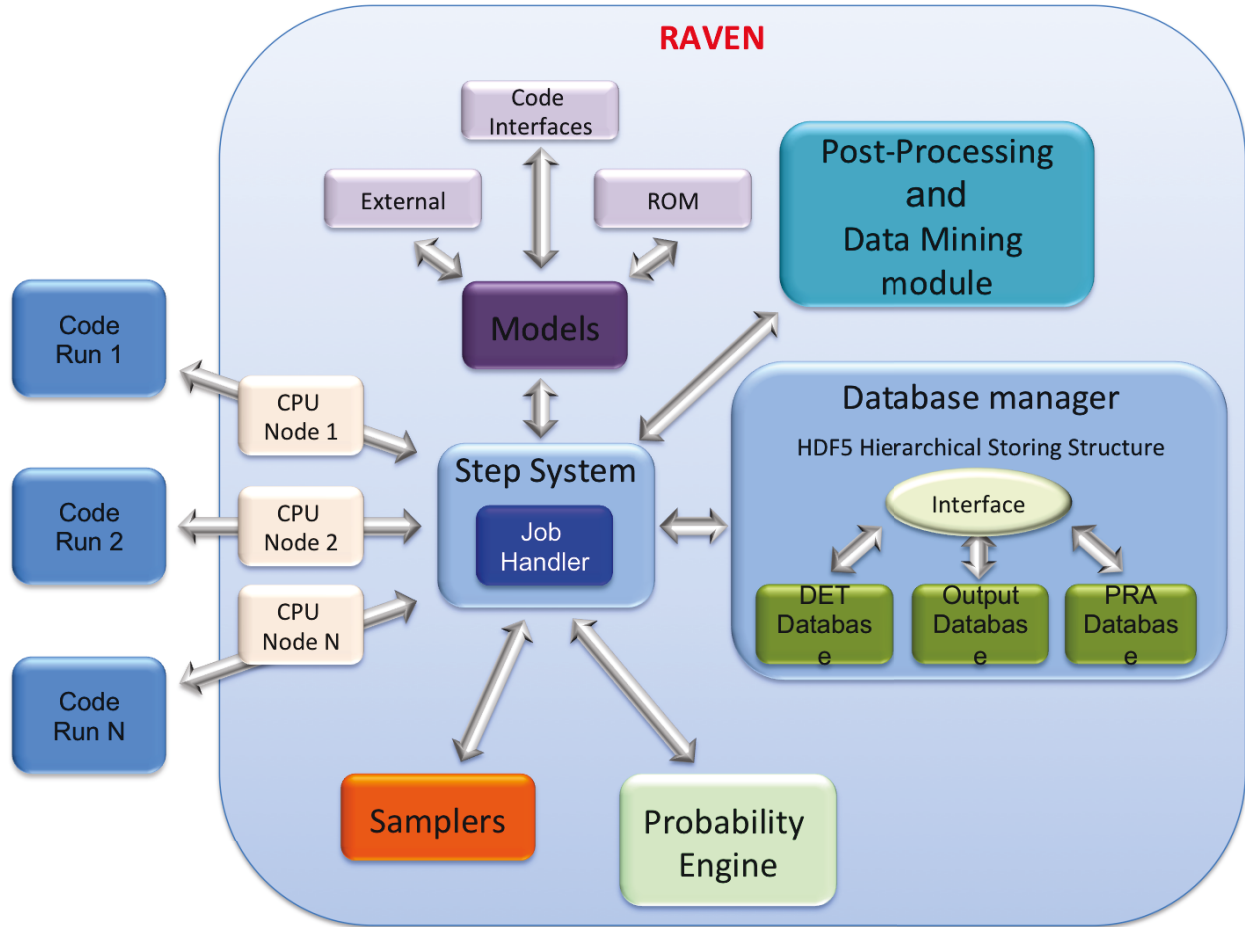


Figure 1. RAVEN framework layout.

is applied to the input space. The input space is defined through the connection of uncertain variables and their relative probability distributions.

- **Model:** A model is the representation of a physical system (e.g., a nuclear power plant); it is therefore capable of predicting the evolution of a system given a coordinate set in the input space. In addition it can represent an action on a data in order to extract key features (e.g. Data mining).
- **Reduced Order Model (ROM):** The evaluation of the system response, as a function of the coordinates in the input space, is very computationally expensive, especially when brute-force approaches (e.g. Monte Carlo methods) are chosen as sampling strategies. ROMs are used to reduce the analysis computational cost, reducing the number of needed points and prioritizing the area of the input space that needs to be explored. They can be considered as an artificial representation of the link between the input and output spaces for a particular

system.

The list above is not comprehensive of all the RAVEN framework components (visualization and storage infrastructure).

Figure 1 shows a schematic representation of the whole framework, highlighting the communication pipes among the different modules and engines. As can be seen, in the figure all the components reported above are schematically shown. In addition the data management, mining and processing modules are shown.

2.2.3 Distribution

The perturbation of the input space needs to be performed by the proper distribution functions. RAVEN provides, through an interface to the `Boost` library, the following variate (truncated and not) distributions: Bernoulli, Binomial, Exponential, Logistic, Lognormal, Normal, Poisson, Triangular, Uniform, Weibull, Gamma, and Beta.

The usage of uni-variate distributions for sampling initial conditions is based on the assumption that the uncertain parameters are not correlated with each other. Quite often uncertain parameters are subject to correlations and thus the uni-variate approach is not applicable. This happens when a generic outcome is dependent on different phenomena simultaneously (i.e. the outcome dependency description can not be collapsed to a function of a single variable). RAVEN currently supports both N-dimensional (N-D) PDFs. The user can provide the distribution values on either Cartesian or sparse grid, which determines the interpolation algorithm used in the evaluation of the imported CDF/PDF:

1. N-Dimensional Spline, for cartesian grids
2. Inverse weight, for sparse grids

Internally, RAVEN provides the needed N-D differentiation and integration algorithms to compute the PDF from the CDF and vice-versa.

The sampling methods use the distributions in order to perform probability-weighted perturbations. For example, in the Monte-Carlo approach, a random number $\in [0, 1]$ is generated (probability threshold) and the CDF, corresponding to that probability, is inverted in order to retrieve the parameter value usable in the simulation. The existence of the inverse for variate distributions is guaranteed by the monotonicity of the CDF. For N-D distributions this condition is not sufficient since the $CDF : X \rightarrow [0, 1], X \in R^N$ and therefore it could not be a bijective function. From an application point of view, this implies that the inverse of a N-D CDF is not unique. As an example, Figure 2 shows a multivariate normal distribution for a pipe failure as function of the pressure and temperature. The plane identifies an isoprobability surface (in this case, a line) that represents a

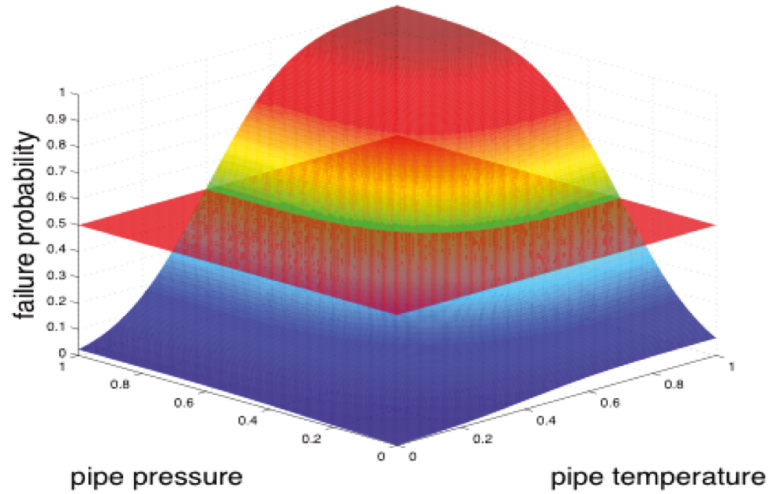


Figure 2. 2-D CDF, function of pressure and temperature.

probability threshold of 50 % in this example. Hence, the inverse of this CDF is an infinite number of points. Note that the standard inverse CDF approach cannot directly be employed. When multivariate distributions are used, RAVEN implements a surface search algorithm for identifying the iso-probability surface location. Once the location of the surface has been found, RAVEN chooses, randomly, one point on it.

2.2.4 Sampler

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:

- Forward
- Dynamic Event Tree (DET)
- Adaptive.

2.2.4.1 Forward Samplers

The *Forward* sampler category collects all of 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, several different and well-known forward samplers are available:

- Monte Carlo (MC)
- Stratified based, whose most known specialization is the Latin Hyper-Cube Sampling (LHS)
- Grid based
- Response Surface Design of Experiment
- Sparse Grid
- Factorials
- Etc.

Since all these sampling strategies are well known, as well as their properties, a detailed investigation of their application is not provided.

2.2.4.2 Dynamic Event Tree Samplers

In technological complex systems, as nuclear power plants, an accident scenario begins with an initiating event and then evolves over time through the interaction of dynamics and stochastic events. This mutual action leads to the production of infinitely many different scenarios, which define a continuous dynamic event tree with infinite branches. At each time point, the stochastic variability of the accident outcomes is determined by a multivariate probability distribution. The PRA analysis needs an approximation to this distribution for selected consequence variables. A way to achieve this goal is a classical Event Tree approach. In PRA analysis, Conventional Event Tree [2] sequences are run simultaneously starting from a single initiating event. The branches occur at user specified times and/or when an action is required by the operator and/or the system, creating a deterministic sequence of events based on the time of their occurrence. One of the disadvantages of this method is that the timing/sequencing of events and system dynamics is not explicitly accounted for in the analysis.

In order to overcome these limitations a “dynamic” approach is needed. The Dynamic Event Tree (DET) [9] [10] technique brings several advantages, among which is the fact that it simulates probabilistic system evolution in a way that is consistent with the severe accident model. This leads to a more realistic and mechanistically consistent analysis of the system taken into consideration.

The dynamic PRA [2], in general, and the Dynamic Event Tree methodologies in particular, 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.

From an application point of view, a N-D grid is built on the CDF space. A single simulation is spawned 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 violated), a new set of simulations (branches) is spawned. Each branch carries its own probability.

Figure 3 shows a practical example. In this particular case, it is assumed that the probability failure of a pipe depends on the fluid pressure magnitude. Three probability thresholds are defined on the cumulative distribution function. One simulation is spawned (0). As soon as the pressure of the fluid reaches a value corresponding to a 33% probability (CDF), a stop signal is sent and the framework starts two new simulations (branches). The branch in which the system evolved to the newer condition (pipe failed, red line) carries 33% of the probability, while the other the complementary amount. The same procedure is repeated at point 2.

Generally, not all the input space can be explored using a DET approach. For instance, usually the parameters affected by aleatory uncertainty are sampled using a dynamic event tree approach, while the ones characterized by epistemic uncertainty are sampled through “forward” sampling strategies.

As already mentioned, this strategy requires a tight interaction between the system code and the sampling driver (i.e., RAVEN framework). In addition, the system code must have a control logic capability (i.e. trigger system). For these reasons, the application of this sampling approach to a generic code needs a larger effort when compared to the other Samplers available in RAVEN. Currently, the DET is fully available for the thermal-hydraulic codes RELAP-7 and RELAP5-3D.

In the RAVEN framework, several different DET-based samplers are available:

- Dynamic Event Tree (aleatory sampling)
- Hybrid Dynamic Event Tree (aleatory and epistemic sampling)
- Adaptive Dynamic Event Tree (goal-oriented sampling for aleatory sampling)
- Adaptive Hybrid Dynamic Event Tree (goal-oriented sampling for aleatory and epistemic sampling).

2.2.4.3 Adaptive Samplers

A key feature available within RAVEN is the possibility to perform smart sampling (also known

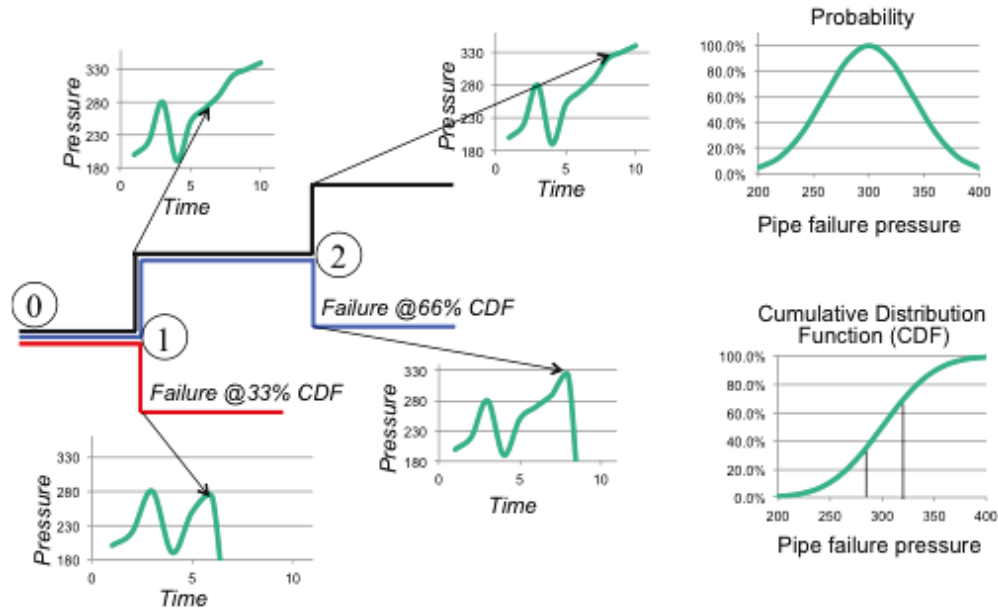


Figure 3. Dynamic Event Tree simulation pattern

as adaptive sampling) as an alternative to classical “forward” techniques. The motivation is that nuclear 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 unfeasible 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 ROM 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 RAVEN framework, several different adaptive samplers are available:

- Limit Surface Search
- Adaptive Dynamic Event Tree
- Adaptive Hybrid Dynamic Event Tree
- Adaptive Sparse Grid

- Adaptive Sobol Decomposition.

2.2.5 Models

The Model entity, in the RAVEN environment, represents a “connection pipeline” between the input and the output space. The RAVEN framework does not own any physical model (i.e., it does not possess the equations needed to simulate a generic physical system, such as Navier-Stocks equations, Maxwell equations, etc.), but implements APIs by which any generic model can be integrated and interrogated. The RAVEN framework provides APIs for four different model categories: Codes, Externals, Post-Processors (PPs) and ROMs. In the following paragraphs, a brief explanation of each of this Model categories is reported.

2.2.5.1 Code

The *Code* model represents the communication pipe between the RAVEN framework and any system and physical code. This is performed through the implementation of interfaces directly operated by the framework.

The procedure of coupling a new code/application with RAVEN is a straightforward process. The coupling is performed through a `Python` interface that interprets the information coming from RAVEN and translates them to the input of the driven code. The coupling procedure does not require modifying RAVEN itself. Instead, the developer creates a new `Python` interface that is going to be embedded in RAVEN at run-time (no need to introduce hard-coded coupling statements). This interface needs to be placed in a folder (whatever name) located in (see figure 4):

```
path/to/raven/distribution/raven/framework/CodeInterfaces/
```

At the initialization stage, RAVEN imports all the Interfaces that are contained in this directory and performs some preliminary cross-checks.

If the coupled code is parallelized and/or multi-threaded, RAVEN will manage the system to optimize the computational resources in both workstations and High Performance Computing systems. Currently, RAVEN has APIs for RELAP5-3D, RELAP-7, any MOOSE-based application, SASS and Modelica.

2.2.5.2 External Model

The External model allows the user to create, in a `Python` file (imported, at run-time, in the

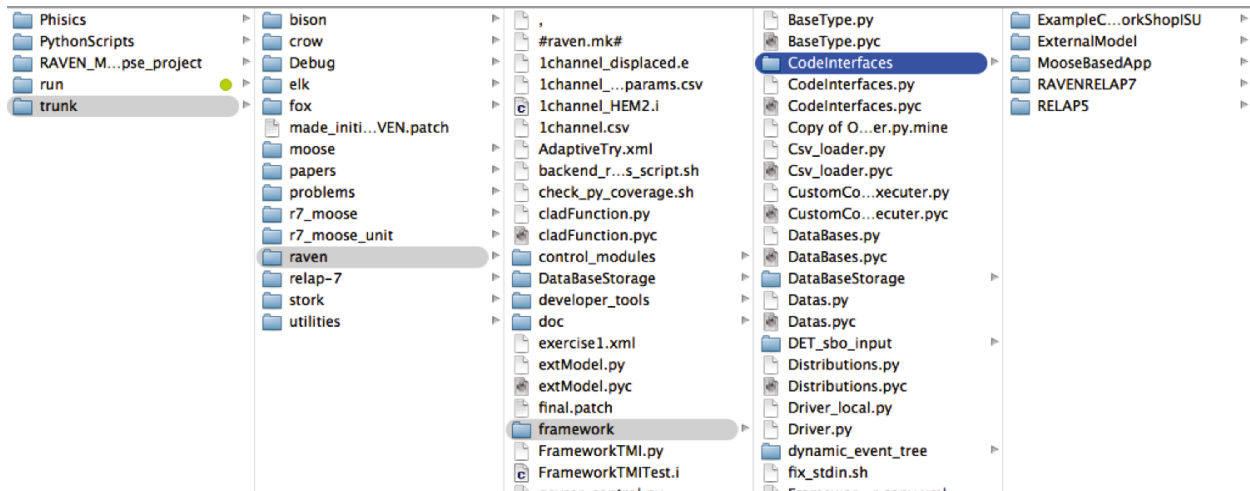


Figure 4. Code Interface Location.

RAVEN framework), its own model (e.g. set of equations representing a physical model, connection to another code, control logic, etc.). This model will be interpreted/used by the framework and, at run-time, will become part of RAVEN itself.

2.2.5.3 Post-Processor

The Post-Processor model represents the container of all of the post-processing capabilities in the RAVEN code. This model is aimed to process the data (for example, derived from the sampling of a physical code) to identify representative Figure of Merits (FOMs). This system has been designed and, presently, is under heavy development by the RAVEN team. Currently, the following post-processors are available:

- *Basic Statistics*, container of the algorithms to compute many of the most important statistical quantities. This post-processor is able to compute mean, sigma/variance, variation coefficient, skewness, kurtosis, median, percentiles and all of the principal matrix quantities such as covariance, sensitivity (either least-squared and variance weighted) and correlation matrices.
- *Comparison Statistics*, aimed to employ validation and verification metrics.
- *Limit Surface*, aimed to compute the limit surface in the input space (i.e. the hyper-surface that represents the boundary between the failure/success of the system).

- *Limit Surface Integral*, intended to compute the integral of the limit surface that corresponds to the probability of failure.
- *Safest Point*, 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 behavior randomly.
- *External Post-Processor*, user-defined post-processor.
- *Topological Decomposition*, aimed to compute an approximated hierarchical Morse-Smale complex which will add two columns to a data-set, performing a topological decomposition of such data-set.
- *Data Mining*, container of all the RAVEN data mining, clustering and dimensionality reduction techniques aimed to identify dominant and common patterns in high-dimensionality data.

2.2.5.4 Reduced Order Model

As briefly mentioned, a ROM is a mathematical representation of a system, used to predict a selected output space of a physical system. The “training” is a process that uses sampling of the physical model to improve the prediction capability (capability to predict the status of the system given a realization of the input space) of the ROM. More specifically, in RAVEN the ROM is trained to emulate a high fidelity numerical representation (system codes) of the physical system. Two general characteristics of these models can be generally assumed (even if exceptions are possible):

1. The higher the number of realizations in the training sets, the higher is the accuracy of the prediction performed by the ROM. This statement is true for most of the cases although some ROMs might be subject to the over-fitting issues. The over-fitting phenomenon is not discussed here, since its occurrence is highly dependent on the algorithm type, (and there is large number of ROM options available in RAVEN). Every time the user chooses a particular ROM algorithm to use, he should consult the relative literature;
2. The smaller the size of the input domain with respect to the variability of the system response, the more likely the surrogate model will be able to represent the system output space.

In most of the cases of interest, the information that is sought is related to defining the failure boundaries of a system with respect to perturbations in the input space. For this reason, in the development of RAVEN, it has been given priority to the introduction of a class of supervised learning algorithms, which are usually referred to as classifiers. A classifier is a ROM that is capable of representing the system behavior through a binary response (failure/success).

The first class of classifier introduced has been the Support Vector Machines (SVMs) [11] with several different kernels (polynomial of arbitrary integer order, radial basis function kernel, sigmoid) followed by a nearest-neighbor based classification using a K-D tree search algorithm. Currently, RAVEN supports around 40 different ROM methodologies. All these supervised learning algorithms have been imported via an API from the scikit-learn [12] library. In addition, the N-Dimensional spline and the inverse weight methods, that are currently available for the interpolation of N-Dimensional PDF/CDF, can also be used as ROMs.

2.2.6 Simulation Environment

RAVEN is perceived by the user as a pool of tools and data. Any action in which the tools are applied to the data is considered a calculation “step” in the RAVEN environment. Simplistically, a “step” can be seen as a **transfer function** between the input and output space through a Model (e.g., Code, External, ROM or Post-Processor). One of the most important step in the RAVEN framework is called “multi-run” that is aimed to handle calculations that involve multiple runs of a driven code (sampling strategies). Firstly, the RAVEN input file associates the variables to a set of PDFs and to a sampling strategy. The multi-run step is used to perform several runs in a block of a model (e.g., in a Monte Carlo sampling). At the beginning of each sub-sequential run, the sampler provides the new values of the variables to be perturbed. The code API places those values in the input file. At this point, the code API generates the run command and asks to be queued by the job handler. The job handler manages the parallel execution of as many runs as possible within a user-prescribed range and communicates with the step controller when a new set of output files are ready to be processed. The code API receives the new input files and collects the data in the RAVEN internal format. The sampler is queried to assess if the sequence of runs is ended, if not, the step controller asks for a new set of values from the sampler and the sequence is restarted as described in Figure 5. The job handler is currently capable to run different run instances of the code in parallel and can also handle codes that are multi-threaded or using any form of parallel implementation. RAVEN also has the capability to plot the simulation outcomes while the set of sampling is performed and store the data for later recovery.

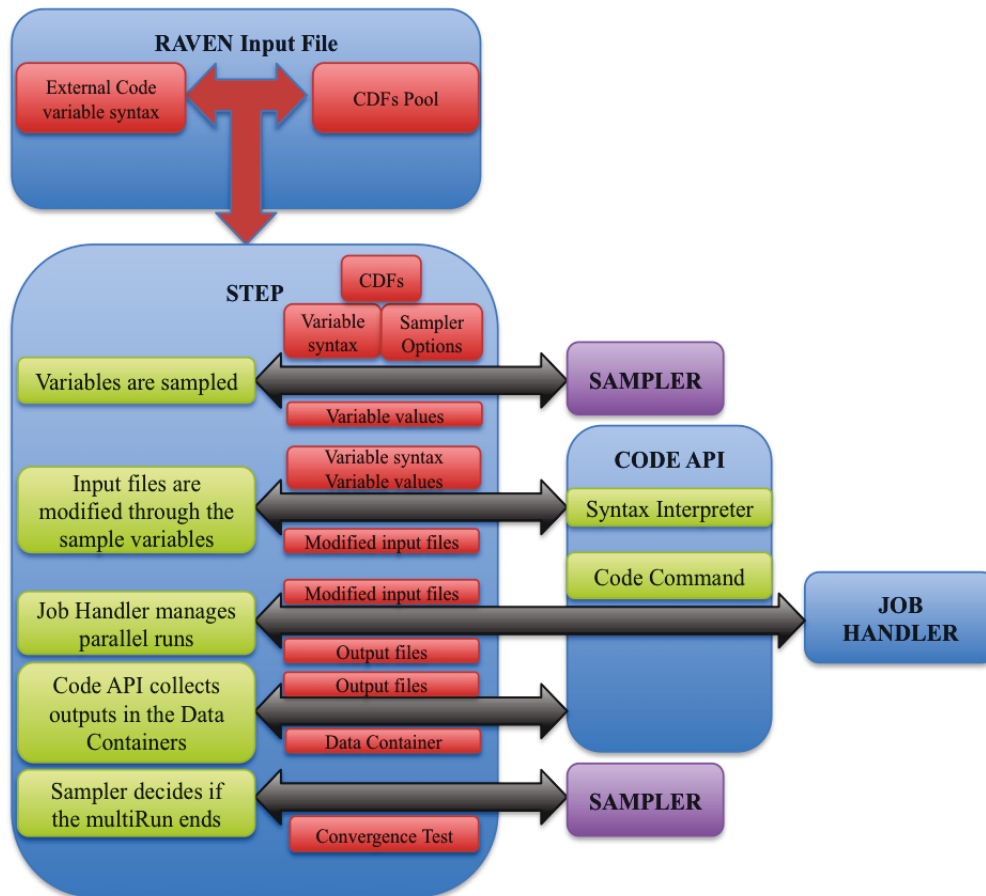


Figure 5. Calculation flow for a multi-run sampling.

3 RAVEN Concepts

After the brief overview of the RAVEN code, it is necessary to focus on the main concepts that are behind the design of the framework:

- *Mathematical Background*: Section 3.1 provides an description of the mathematical background of RAVEN, overall focalizing on the probabilistic dynamics
- *RAVEN entities*: Section 3.2 is aimed to provide an overview on how the different objects in RAVEN can interact with each other, generating the user-dependent analysis flow
- *RAVEN input main components*: Section 3.3 provides a brief introduction of the input structure, introducing some of the input “structure” that are going to be used in this manual. A detailed explanation of the input structure and keywords is reported in the user manual [13].

3.1 RAVEN Mathematical Background

3.1.1 System and Control Model

The first step is the derivation of the mathematical model representing, with a high level of abstraction, the plant and control system model. Let $\bar{\theta}(t)$ be a vector describing the system status in the phase space, characterized by the following governing equation:

$$\frac{\partial \bar{\theta}}{\partial t} = \bar{H}(\bar{\theta}(t), t) \quad (1)$$

In Equation above, the assumption of time differentiability of the trajectory equation $\bar{H}(\bar{\theta}(t), t)$ in the phase space has been taken. This assumption is not fully correct and generally required and it is used here, without missing of generality, for compactness of the notation.

It can now be performed an arbitrary decomposition of the phase space:

$$\bar{\theta} = \begin{pmatrix} \bar{x} \\ \bar{v} \end{pmatrix} \quad (2)$$

The decomposition is made in such a way that \bar{x} represent the unknowns solved by a system code (such as RELAP5-3D [8], RELAP7 [5], etc.) while \bar{v} are the variables directly controlled by the control system (e.g., automatic mitigation systems, operator actions, etc.).

The governing equation can be now cast in the following system of equations:

$$\begin{cases} \frac{\partial \bar{x}}{\partial t} = \bar{F}(\bar{x}, \bar{v}, t) \\ \frac{\partial \bar{v}}{\partial t} = \bar{V}(\bar{x}, \bar{v}, t) \end{cases} \quad (3)$$

Consequentially to this splitting, \bar{x} contains the state variables of the phase space that are continuous while \bar{v} contains discrete state variables that are usually handled by the control system (consequentially, named **control variables**). It can be noticed that the function $\bar{V}(\bar{x}, \bar{v}, t)$, representing the control system, does not depend on the knowledge of the complete status of the system but on a restricted subset that can be named **monitored variables** \bar{C} :

$$\begin{cases} \frac{\partial \bar{x}}{\partial t} = \bar{F}(\bar{x}, \bar{v}, t) \\ \bar{C} = \bar{G}(\bar{x}, t) \\ \frac{\partial \bar{v}}{\partial t} = \bar{V}(\bar{x}, \bar{v}, t) \end{cases} \quad (4)$$

where \bar{C} is a vector of smaller dimensionality than \bar{x} and, therefore, more convenient to handle. As it can be noticed, the standard nomenclature of *signals* (monitored variables) and *status* (control variables) is not adopted. Two principal reasons justify this decision:

- The definition of signals is tight to the definition of the control logic for each component and, therefore, relative rather than absolute in the overall system analysis. For example, it is possible the the *signals* for a component represent *status* of another one, determining an in-unique definition.
- The standard nomenclature becomes meaningless when this derivation is applied to Uncertainty Quantification (UQ).

3.1.1.1 Splitting Approach for the Simulation of the Control System

Equation 4 represents a fully coupled system of Partial Differential Equations (PDEs). To solve this system, an *operator splitting* approach is employed. This method is preferable in this context for several reasons, among which the following:

- In reality, the control system (automatic mitigation systems, operator actions, etc.) is always characterized by an intrinsic delay
- The reaction of the control system might make the system “move” among different discrete states; therefore, numerical errors will be always of first order unless the discontinuity is explicitly treated.

Employing the *operator splitting* approach, Equation 4 can be cast as follows:

$$\begin{cases} \frac{\partial \bar{x}}{\partial t} = \bar{F}(\bar{x}, \bar{v}_{t_{i-1}}, t) \\ \bar{C} = \bar{G}(\bar{x}, t) \\ \frac{\partial \bar{v}}{\partial t} = \bar{V}(\bar{x}, \bar{v}_{t_{i-1}}, t) \end{cases} \quad t_{i-1} \leq t \leq t_i = t_{i-1} + \Delta t_i \quad (5)$$

Hence, the system of equations is solved decomposing it into simpler sub-problems that are treated individually using specialized numerical algorithms.

3.1.1.2 Definition of the Monitored Variable Space

The contraction of the information from the \bar{x} space to the \bar{C} space is a crucial step. Since \bar{C} represents an arbitrary middle step, it is needed to define a set of rules that make this choice unique. \bar{C} is chosen such that:

- The solution of $\left. \frac{\partial \bar{v}}{\partial t} \right| = \bar{V}(\bar{x}, \bar{v}_{t_{i-1}}, t)$ can be carried along without any knowledge of the solution algorithm of $\left. \frac{\partial \bar{x}}{\partial t} \right| = \bar{F}(\bar{x}, \bar{v}_{t_{i-1}}, t)$. This requirement determines the minimum information contraction from \bar{x} to \bar{C} .
- All actions represented by $\bar{C} = \bar{G}(\bar{x}, t)$ require knowledge of the solution algorithm of $\left. \frac{\partial \bar{x}}{\partial t} \right| = \bar{F}(\bar{x}, \bar{v}_{t_{i-1}}, t)$. This requirement determines the maximum information contraction from \bar{x} to \bar{C} .

The intersection of the two sub-spaces defined above create a minimal unique set.

3.1.1.3 Definition of the Auxiliary Variable Space

In the previous sections, it has been determined that the needed information to model the dynamic system is contained in the solution vectors \bar{x} and \bar{v} . Even if \bar{x} and \bar{v} are sufficient to assess the system status at every point in time, it can result in an unpractical way to model the eventual control system. Let's suppose to model a component of a particular system that presents different behavior depending on other systems or operation behaviors. In order to define the status of this component in every point in time, the whole history of the system needs to be tracked. In order to remove these inefficiency, a set of auxiliary variables \bar{a} can be introduced. These variables are the ones that in the analysis of stochastic dynamics are artificially added into the phase space to a non-Markovian system to obtain back a Markovian behavior. In this way only the previous time-step information is needed to determine the status of the system.

Adding this additional system of variables, Equation 5 can be casted as follows:

$$\left\{ \begin{array}{l} \left. \frac{\partial \bar{x}}{\partial t} = \bar{F}(\bar{x}, \bar{v}_{t_{i-1}}, t) \right. \\ \quad \bar{C} = \bar{G}(\bar{x}, t) \\ \left. \frac{\partial \bar{a}}{\partial t} = \bar{A}(\bar{x}, \bar{C}, \bar{a}, \bar{v}_{t_{i-1}}, t) \right. \\ \left. \frac{\partial \bar{v}}{\partial t} = \bar{V}(\bar{C}, \bar{a}, \bar{v}_{t_{i-1}}, t) \right. \end{array} \right. \quad t_{i-1} \leq t \leq t_i = t_{i-1} + \Delta t_i \quad (6)$$

3.1.2 Dynamic Systems Stochastic Modeling

3.1.2.1 General system of equations and variable classification

In Section 3.1.1, the derivation of the governing equations for a controllable system have been reported. In this section, the mathematical framework of the modeling of dynamic stochastic systems, under uncertainties, is derived.

Dynamic stochastic systems are the ones whose dynamic is characterized by intrinsic randomness. Random behaviors, although present in nature, are often artificially introduced into physical models to account for the incapability of fully modeling part of the nature of the system behavior and/or of the phenomena bounding the physical problem.

The distinction between variables that are artificially considered aleatory and the ones intrinsically aleatory corresponds with the classical definition of epistemic and aleatory uncertainties. From a system simulation point of view it is more relevant how these variables, the sources of aleatory behavior, change in time. Possible examples of random elements are:

- random variability of parameters (e.g., uncertainty in physical parameters)
- presence of noise (background noise due to intrinsically stochastic behaviors or lack of detail in the simulation)
- Uncertainty in the initial and boundary conditions
- Random failure of components
- aging effects.

Before introducing the mathematical models for uncertainty, it can be beneficial to recall Equation 1, adding the initial conditions:

$$\begin{cases} \frac{\partial \bar{\theta}(t)}{\partial t} = \bar{H}(\bar{\theta}(t), t) \\ \bar{\theta}(t_0) = \bar{\theta}_0 \end{cases} \quad (7)$$

At this point, each source of uncertainty or stochastic behavior is considered and progressively added in Equation 7. For the scope of this derivation, it is convenient to split the phase space into *continuous* (e.g., temperature, pressure, enthalpy, etc.) and *discrete* (e.g., status of components, such as operational and failure states) variables as follows:

- $\bar{\theta}^c \in \Phi \subseteq \mathbb{R}^C$, the set of continuous variables
- $\bar{\theta}^d \in \Psi \subseteq \mathbb{N}^D$, the set of discrete variables
- $\bar{\theta}(t) = \bar{\theta}^c \oplus \bar{\theta}^d$.

Consequently, Equation 7 assumes the following form:

$$\begin{cases} \frac{\partial \bar{\theta}^c(t)}{\partial t} = f(\bar{\theta}^c, \bar{\theta}^d, t) \\ \frac{\partial \bar{\theta}^d(t)}{\partial t} = g(\bar{\theta}^c, \bar{\theta}^d, t) \\ \bar{\theta}^c(t_0) = \bar{\theta}_0^c \\ \bar{\theta}^d(t_0) = \bar{\theta}_0^d \end{cases} \quad (8)$$

Note that the time derivative operator has been also used for the time discontinuous variables, even if this is allowed only introducing complex extension of the time derivative operator. In this context, the $\frac{\partial}{\partial t}$ on the discontinuous space is employed for simplifying the notation only.

3.1.2.2 Probabilistic Nature of the Parameters Characterizing the Equation

As shown in Equation 9, The first stochastic behaviors to be introduced are the uncertainties associated with the:

- initial conditions (i.e. $\bar{\theta}^c$ and $\bar{\theta}^d$ at time t_0), and
- parameters characteristic of $f(\bar{\theta}^c, \bar{\theta}^d, t)$ and $g(\bar{\theta}^c, \bar{\theta}^d, t)$.

$$\begin{cases} \frac{\partial \bar{\theta}^c(t)}{\partial t} = f(\bar{\theta}^c, \bar{\theta}^d, \bar{\alpha}_{staz}, t) \\ \frac{\partial \bar{\theta}^d(t)}{\partial t} = g(\bar{\theta}^c, \bar{\theta}^d, \bar{\alpha}_{staz}, t) \\ \Pi(\bar{\theta}^c, t_0) \sim pdf(\bar{\theta}_0^c | \sigma_c^2) \\ \Pi(\bar{\theta}^d, t_0) \sim pdf(\bar{\theta}_0^d | \sigma_d^2) \\ \bar{\alpha}_{staz}(t) = \bar{\alpha}_{staz}(t_0) \sim pdf(\bar{\alpha}_{staz}^0 | \sigma_{staz}^2) \end{cases} \quad (9)$$

In Equation 9, $\Pi(\bar{\theta}^c, t_0)$ indicates the probability distribution of $\bar{\theta}^c$ at the initial time $t = t_0$ while $pdf(\mu | \sigma^2)$ represents a generic probability distribution function having mean value μ and sigma σ . The term $\bar{\alpha}_{staz}$ is the vector of parameters affected by uncertainty but not varying over time.

As already mentioned, Equation 9 considers uncertainties whose values do not change during the dynamic evolution of the system. This set of uncertainties accounts for most of the common source of aleatory behaviors. Examples of this kind of uncertainties are:

- *Uncertainty associated with the heat conduction coefficient:* This value is known (but uncertain) and has no physical reason to change during the simulation;
- *Uncertainty on failure temperature for a pipe:* This value is usually characterized by a probability distribution function but once the value has been set (like through random sampling) it will not change during the simulation.

From a modeling perspective, all the probabilistic behaviors connected to $\Pi(\bar{\theta}^c, t_0)$, $\Pi(\bar{\theta}^d, t_0)$ and $\bar{\alpha}_{staz}(t)$ can be modeled without changing the dimensionality of the phase space (hence, no alteration of the solution algorithm is required), simply performing sampling of the input space. In addition, the Markovian assumption is still preserved.

3.1.2.3 Variables Subject to Random Motion

The next aleatory component to be accounted for is the set of parameters that continuously change over time (i.e. $\bar{\alpha}_{brow}$). In other words, these parameters are referred as if they behave like a *Brownian motion*. While what commonly is indicated as *Brownian motion* has not impact at the character the space and time scales (characteristic of a physical system), there are parameters that have (or **appear** to have) such behavior. The *Brownian motion* characteristic of some variables can be completely synthetic, due to the lack of modeling details in the simulation model.

For instance, two examples of these randomly varying variables are:

- *Cumulative damage growth in material.* Experimental data and models representing this phenomenon show large uncertainties. There is also an intrinsic natural stochasticity driving the accumulation of the damage (natural Brownian motion);
- *Heat conductivity in the fuel gap during heating of fuel.* During some transients there are situations where the fuel is in contact with the clad while in others where there is the presence of a gap. While in nature this is a discontinuous transition, it is not usually possible to model in such a way, especially if vibrations of the fuel lead to high frequency oscillations. In this case, it would be helpful to introduce directly into the simulation a random noise characterizing the thermal conductivity when these transitions occur (synthetic Brownian motion).

The system of Equations 9 can be rewritten in the following form:

$$\left\{ \begin{array}{l} \frac{\partial \bar{\theta}^c(t)}{\partial t} = f(\bar{\theta}^c, \bar{\theta}^d, \bar{\alpha}_{staz}, \bar{\alpha}_{brow}, t) \\ \frac{\partial \bar{\theta}^d(t)}{\partial t} = g(\bar{\theta}^c, \bar{\theta}^d, \bar{\alpha}_{staz}, \bar{\alpha}_{brow}, t) \\ \frac{\partial \bar{\alpha}_{brow}}{\partial t} = b(\bar{\theta}^c, \bar{\theta}^d, \bar{\alpha}_{staz}, \bar{\alpha}_{brow}, t) \Gamma(t) \\ \Pi(\bar{\theta}^c, t_0) \sim pdf(\bar{\theta}_0^c, \sigma_c^2) \\ \Pi(\bar{\theta}^d, t_0) \sim pdf(\bar{\theta}_0^d, \sigma_d^2) \\ \bar{\alpha}_{staz}(t) = \bar{\alpha}_{staz}(t_0) \sim pdf(\bar{\alpha}_{staz}^0, \sigma_{staz}^2) \\ \bar{\alpha}_{brow}(t_0) \sim \bar{\alpha}_{brow}^0 \Gamma(t_0) \end{array} \right. \quad (10)$$

where $\Gamma(t)$ is 0-mean random noise and $\bar{\alpha}_{brow}$ is the set of parameters subject to *Brownian motion*. Clearly, the equation referring to the time change of the parameters subject to the *Brownian motion*

should be interpreted in the **Ito** sense [C. Gardiner, Stochastic Methods, Springer (2009)].

3.1.2.4 Discontinuously and Stochastically varying variables

The last and probably most difficult step is the introduction of parameters that are neither constant during the simulation nor continuously vary over time. As an example, consider a valve that, provided set of operating conditions, opens or closes. If this set of conditions is reached n times during the simulation, the probability of the valve correctly operating should be sampled n times. It is also foreseeable that the history of failure/success of the valve will impact future probability of failure/success. In this case the time evolution of such parameters (discontinuously stochastic changing parameters $\bar{\alpha}_{DS}$) is governed by the following equation:

$$\frac{\partial \bar{\alpha}_{DS}(t)}{\partial t} = \bar{\delta}(\bar{\alpha}_{DS}, \bar{\theta}^c, \bar{\theta}^d, \bar{\alpha}_{staz}, \bar{\alpha}_{brow}, t) \times \bar{V}(\bar{\alpha}_{DS}, \bar{\theta}^c, \bar{\theta}^d, \bar{\alpha}_{staz}, \bar{\alpha}_{brow}, t) \times \bar{p}\left(\int_{t_0}^t S(\bar{\theta}(t'), t') dt'\right) \quad (11)$$

where:

- The function $\bar{\delta}(\bar{\alpha}_{DS}, \bar{\theta}^c, \bar{\theta}^d, \bar{\alpha}_{staz}, \bar{\alpha}_{brow}, t)$ is the delta of Dirac of the instant on which the transition need to be evaluated (control logic signaling to the valve to open/close).
- The term $\bar{p}\left(\int_{t_0}^t S(\bar{\theta}(t'), t') dt'\right) = \bar{p}\left(\int_{t_0}^t \bar{\alpha}_{DS}, \bar{\theta}^c, \bar{\theta}^d, \bar{\alpha}_{staz}, \bar{\alpha}_{brow}, t dt'\right)$ represents the transition probability between different states (in case of the valve: open/close). Note that the time integral of the parameter history accounts for the memory of the component via the kernel $S(\bar{\theta}(t'), t')$.
- The term $\bar{V}(\bar{\alpha}_{DS}, \bar{\theta}^c, \bar{\theta}^d, \bar{\alpha}_{staz}, \bar{\alpha}_{brow}, t)$ is the rate of change of $\bar{\alpha}_{DS}$. For a discrete parameter, it is defined as the value of the instantaneous $\bar{\alpha}_{DS}$ change.

The introduction of the history dependency introduced in the term \bar{p} determines that the system cannot be considered Markovian if “countermeasures” are not taken. In order to make the system return to be Markovian, the phase space needs to be expanded (i.e., increase its dimensionality): the time at which the parameters changed status and their corresponding values $\{(\bar{\alpha}_{DS}, t_i)\} = \{\bar{\alpha}_{DS}, t_i\} = \bar{\alpha}_{DS}, \bar{t} \text{ (for } i = 1, \dots, n)$.

Equation 11 now assumes the form:

$$\frac{\partial \bar{\alpha}_{DS}(t)}{\partial t} = \bar{\delta}(\bar{\alpha}_{DS}, \bar{\theta}^c, \bar{\theta}^d, \bar{\alpha}_{staz}, \bar{\alpha}_{brow}, t) \times \bar{V}(\bar{\alpha}_{DS}, \bar{\theta}^c, \bar{\theta}^d, \bar{\alpha}_{staz}, \bar{\alpha}_{brow}, t) \times \bar{p}(\bar{\alpha}_{DS}, \bar{t}, \bar{\theta}^c, \bar{\theta}^d, \bar{\alpha}_{staz}, \bar{\alpha}_{brow}, t) \quad \text{for } t \geq t_n \quad (12)$$

This formulation introduces a phase space that is continuously growing over time $n \rightarrow \infty$. In this respect, it is useful to introduce and discuss possible assumptions:

1. The memory of the past is not affected by the time distance; in this case:

$$\bar{p} \left(\bar{\alpha}_{DS}, \bar{t}, \bar{\theta}^c, \bar{\theta}^d, \bar{\alpha}_{staz}, \bar{\alpha}_{brow}, t \right) = \bar{p} \left(\bar{\alpha}_{DS}, \bar{\theta}^c, \bar{\theta}^d, \bar{\alpha}_{staz}, \bar{\alpha}_{brow}, t \right) \quad (13)$$

The dimensionality of the phase space is still growing during the simulation since more and more sampling is performed, but the time integral is removed from the transition probability. A simple example of this situation is a component activated on demand in which failure is a function of all previous sampling, but not of when the component was sampled or in which sequence the outcome occurred.

2. The number of samples is determined before the simulation itself takes place (e.g., n times) In this case the different $\bar{\alpha}_{DS_i}$ could be treated explicitly as $\bar{\alpha}_{staz}$ while \bar{t} would still remain a variable to be added to the phase space (if simplification 1 is not valid) but of fixed dimension. In this case \bar{t} still needs to be computed and its expression is:

$$\bar{t}(t) = \int_{t_0}^t \bar{t} \bar{\delta} \left(\bar{\alpha}_{DS}, \bar{\theta}^c, \bar{\theta}^d, \bar{\alpha}_{staz}, \bar{\alpha}_{brow}, t \right) dt \quad (14)$$

The transition probability becomes:

$$\bar{p} \left(\int_{t_0}^t dt S(\bar{t}), \bar{\alpha}_{DS}, \bar{\theta}^c, \bar{\theta}^d, \bar{\alpha}_{staz}, \bar{\alpha}_{brow}, t \right) \quad (15)$$

For example, this is the case of a component that is sampled a fixed number of times for a given simulation while the contribution of the history to the transition probability might decay exponentially over time. This approximation might eliminate the memory from the system by adding n variables to the phase space t_i (for $i = 1, \dots, n$) thus restoring the Markovian characteristic.

3. Another possible approximation alternative to the previous one is that the memory of the system (here explicitly represented by $\int_{t_0}^t \bar{\alpha}_{DS} dt$) is limited only to a fixed number of steps back in the past. In this case n is always bounded. Therefore, adding $\{\bar{\alpha}_{DS_i}, t_i\}$, (for $i = 1, \dots, n$) would possibly preserve the system Markovian properties of the system. This approximation allows for eliminating the memory from the system by expanding the phase space $2n$ variables. From a software implementation point of view, this is the most complex situation since without any simplification we would have to deal with a system that is never reducible to a Markovian one and therefore forced to use the whole history of the system to forecast its evolution at each time step.

Assumption 1 limits this cost by restraining it to the set of values assumed by the variable but would still lead to very difficult to deal with situation. Assumption 2 would require an expansion of phase space to introduce the time at which the transitions happens but the value that the parameter will assume at each sampling could be treated as initial condition. Assumption 3 would instead require the expansion of the phase space for both the time and the values of the transitioning variables.

Based on the this simplifications, the system of Equations 10, accounting also for $\bar{\alpha}_{DS}$ can be cast into the form:

$$\left\{ \begin{array}{l} \frac{\partial \bar{\theta}^c(t)}{\partial t} = f\left(\bar{\theta}^c, \bar{\theta}^d, \bar{\alpha}_{staz}, \bar{\alpha}_{brow}, t\right) \\ \frac{\partial \bar{\theta}^d(t)}{\partial t} = g\left(\bar{\theta}^c, \bar{\theta}^d, \bar{\alpha}_{staz}, \bar{\alpha}_{brow}, t\right) \\ \frac{\partial \bar{\alpha}_{brow}}{\partial t} = b\left(\bar{\theta}^c, \bar{\theta}^d, \bar{\alpha}_{staz}, \bar{\alpha}_{brow}, t\right) \Gamma(t) \\ \frac{\partial \bar{\alpha}_{DS}(t)}{\partial t} = \bar{\delta}\left(\bar{\alpha}_{DS}, \bar{\theta}^c, \bar{\theta}^d, \bar{\alpha}_{staz}, \bar{\alpha}_{brow}, t\right) \times \bar{V}\left(\bar{\alpha}_{DS}, \bar{\theta}^c, \bar{\theta}^d, \bar{\alpha}_{staz}, \bar{\alpha}_{brow}, t\right) \times \\ \times \bar{p}\left(\int_{t_0}^t dt \bar{\alpha}_{DS}, \bar{\theta}^c, \bar{\theta}^d, \bar{\alpha}_{staz}, \bar{\alpha}_{brow}, t\right) \\ \Pi\left(\bar{\theta}^c, t_0\right) \sim pdf\left(\bar{\theta}_0^c, \sigma_c^2\right) \\ \Pi\left(\bar{\theta}^d, t_0\right) \sim pdf\left(\bar{\theta}_0^d, \sigma_d^2\right) \\ \bar{\alpha}_{staz}(t) = \bar{\alpha}_{staz}(t_0) \sim pdf\left(\bar{\alpha}_{staz}^0, \sigma_{staz}^2\right) \\ \bar{\alpha}_{brow}(t_0) \sim \bar{\alpha}_{brow}^0 \Gamma(t_0) \\ \bar{\alpha}_{DS}(t_0) = \bar{\alpha}_{DS}^0 \end{array} \right. \quad (16)$$

Introducing the Simplifications **1** and **3** (the most appropriated in this context), Equation 16 becomes:

$$\left\{ \begin{array}{l} \frac{\partial \bar{\theta}^c(t)}{\partial t} = f\left(\bar{\theta}^c, \bar{\theta}^d, \bar{\alpha}_{staz}, \bar{\alpha}_{brow}, t\right) \\ \frac{\partial \bar{\theta}^d(t)}{\partial t} = g\left(\bar{\theta}^c, \bar{\theta}^d, \bar{\alpha}_{staz}, \bar{\alpha}_{brow}, t\right) \\ \frac{\partial \bar{\alpha}_{brow}}{\partial t} = b\left(\bar{\theta}^c, \bar{\theta}^d, \bar{\alpha}_{staz}, \bar{\alpha}_{brow}, t\right) \Gamma(t) \\ \frac{\partial \bar{\alpha}_{DS}(t)}{\partial t} = \bar{\delta}\left(\bar{\alpha}_{DS}, \bar{\theta}^c, \bar{\theta}^d, \bar{\alpha}_{staz}, \bar{\alpha}_{brow}, t\right) \times \bar{V}\left(\bar{\alpha}_{DS}, \bar{\theta}^c, \bar{\theta}^d, \bar{\alpha}_{staz}, \bar{\alpha}_{brow}, t\right) \times \\ \times \bar{p}\left(\bar{\alpha}_{DS}, \bar{\theta}^c, \bar{\theta}^d, \bar{\alpha}_{staz}, \bar{\alpha}_{brow}, t\right) \\ \Pi\left(\bar{\theta}^c, t_0\right) \sim pdf\left(\bar{\theta}_0^c, \sigma_c^2\right) \\ \Pi\left(\bar{\theta}^d, t_0\right) \sim pdf\left(\bar{\theta}_0^d, \sigma_d^2\right) \\ \bar{\alpha}_{staz}(t) = \bar{\alpha}_{staz}(t_0) \sim pdf\left(\bar{\alpha}_{staz}^0, \sigma_{staz}^2\right) \\ \bar{\alpha}_{brow}(t_0) \sim \bar{\alpha}_{brow}^0 \Gamma(t_0) \\ \bar{\alpha}_{DS}(t_0) = \bar{\alpha}_{DS}^0 \end{array} \right. \quad (17)$$

This dissertation does not cover all the possible phenomena, but it provides a sufficient mathematical framework for extrapolating toward cases that are not explicitly treated.

Given the presence of all these sources of stochastic behaviors, every exploration of the uncertainties (through sampling strategies) only represents a possible trajectory of the system in the phase space. Hence, it is much more informative the assessment of the probability of a particular response, rather than the response itself.

The explanation of these concepts is demanded to next section.

3.1.3 Formulation of the equation set in a statistical framework

Based on the premises reported in the previous sections and assuming that at least one of the simplifications mentioned in Section 3.1.2.1 is applicable (i.e. the system can be casted as Markovian), it is needed to investigate the system evolution in terms of its probability density function in the global phase space $\bar{\theta}$ via the Chapman-Kolmogorov equation [14].

The integral form of the Chapman-Kolmogorov is the following:

$$\Pi(\bar{\theta}_3, t_3 | \bar{\theta}_1, t_1) = \int d\bar{\theta}_2 \Pi(\bar{\theta}_2, t_2 | \bar{\theta}_1, t_1) \Pi(\bar{\theta}_3, t_3 | \bar{\theta}_2, t_2) \quad \text{where } t_1 < t_2 < t_3 \quad (18)$$

while its differential form is:

$$\frac{\partial \Pi(\bar{\theta}, t | \bar{\theta}_0, t_0)}{\partial t} = \mathcal{L}_{CK}(\Pi(\bar{\theta}, t | \bar{\theta}_0, t_0)) \quad (19)$$

The transition from the integral to the differential form is possible under the following assumptions:

$$\lim_{\Delta t \rightarrow 0} \frac{1}{\Delta t} \int_{|\bar{\theta}_2 - \bar{\theta}_1| < \varepsilon} \Pi(\bar{\theta}_2, t + \Delta t | \bar{\theta}_1, t) d\bar{\theta}_2 = 0 \quad (20)$$

$$\lim_{\Delta t \rightarrow 0} \frac{1}{\Delta t} \Pi(\bar{\theta}_2, t + \Delta t | \bar{\theta}_1, t) = W(\bar{\theta}_2 | \bar{\theta}_1, t) \quad (21)$$

$$\lim_{\Delta t \rightarrow 0} \frac{1}{\Delta t} \int_{|\bar{\theta}_2 - \bar{\theta}_1| < \varepsilon} (\bar{\theta}_{2,i} - \bar{\theta}_{1,i}) \Pi(\bar{\theta}_2, t + \Delta t | \bar{\theta}_1, t) d\bar{\theta}_2 = A_i(\bar{\theta}_1, t) + \mathcal{O}(\varepsilon) \quad (22)$$

$$\lim_{\Delta t \rightarrow 0} \frac{1}{\Delta t} \int_{|\bar{\theta}_2 - \bar{\theta}_1| < \varepsilon} (\bar{\theta}_{2,i} - \bar{\theta}_{1,i}) (\bar{\theta}_{2,j} - \bar{\theta}_{1,j}) \Pi(\bar{\theta}_2, t + \Delta t | \bar{\theta}_1, t) d\bar{\theta}_2 = B_{i,j}(\bar{\theta}_1, t) + \mathcal{O}(\varepsilon) \quad (23)$$

The first condition guarantees the continuity of $\Pi(\bar{\theta}, t | \bar{\theta}_0, t_0)$, while the other three force the finite existence of three parameters that will be described in Section 8.1.3. Equation 25 can be furthermore decomposed into the continuous and discrete components:

$$\begin{cases} \Pi(\bar{\theta}_3^c, t_3 | \bar{\theta}_1^c, t_1) = \int \Pi(\bar{\theta}_2^c, t_2 | \bar{\theta}_1^c, t_1) \Pi(\bar{\theta}_3^c, t_3 | \bar{\theta}_2^c, t_2) d\bar{\theta}_2^c \\ \Pi(\bar{\theta}_3^d, t_3 | \bar{\theta}_1^d, t_1) = \int \Pi(\bar{\theta}_2^d, t_2 | \bar{\theta}_1^d, t_1) \Pi(\bar{\theta}_3^d, t_3 | \bar{\theta}_2^d, t_2) d\bar{\theta}_2^d \end{cases} \quad \text{where } t_1 < t_2 < t_3 \quad (24)$$

and its differential form is as follows:

$$\begin{cases} \frac{\partial \Pi(\bar{\theta}^c, t | \bar{\theta}_0^c, t_0)}{\partial t} = \mathcal{L}_{CK}^c(\Pi(\bar{\theta}^c, t | \bar{\theta}_0^c, t_0), \bar{\theta}^d, \bar{\alpha}_{brow}, \bar{\alpha}_{staz}, \bar{\alpha}_{DS}, t) \\ \frac{\partial \Pi(\bar{\theta}^d, t | \bar{\theta}_0^d, t_0)}{\partial t} = \mathcal{L}_{CK}^d(\Pi(\bar{\theta}^d, t | \bar{\theta}_0^d, t_0), \bar{\theta}^c, t) \end{cases} \quad (25)$$

where:

- $\Pi \left(\bar{\theta}^c, t | \bar{\theta}_0^c, t_0 \right)$ of the system to be in state $\bar{\theta}^c$ at time t given that the system was in $\bar{\theta}_0^c$ at time t_0 ;
- $\Pi \left(\bar{\theta}^d, t | \bar{\theta}_0^d, t_0 \right)$ of the system to be in state $\bar{\theta}^d$ at time t given that the system was in $\bar{\theta}_0^d$ at time t_0 ;
- $\mathcal{L}_{CK}^c(\cdot)$ and $\mathcal{L}_{CK}^d(\cdot)$ are specific Chapman- Kolmogorov operators that will be described in the following section.

3.1.4 The Chapman-Kolmogorov Equation

The system of equations 2, written in integral form, can be solved in a differential form through the Chapman-Kolmogorov (C-K) operator [14]:

$$\begin{aligned} \frac{\partial \Pi(\bar{\theta}^c, t | \bar{\theta}_0^c, t_0)}{\partial t} = & - \sum_i \frac{\partial}{\partial \bar{\theta}_i^c} \left(A_i \left(\bar{\theta}^c, \bar{\theta}^d, t \right) \Pi \left(\bar{\theta}^c, t | \bar{\theta}_0^c, t_0 \right) \right) + \\ & + \frac{1}{2} \sum_{i,j} \frac{\partial^2}{\partial \bar{\theta}_i^c \partial \bar{\theta}_j^c} \left(B_{i,j} \left(\bar{\theta}^c, \bar{\theta}^d, t \right) \Pi \left(\bar{\theta}^c, t | \bar{\theta}_0^c, t_0 \right) \right) + \\ & + \int \left(W \left(\bar{\theta}^c | \bar{\theta}'^c, \bar{\theta}^d, t \right) \Pi \left(\bar{\theta}'^c, t | \bar{\theta}_0^c, t_0 \right) - W \left(\bar{\theta}'^c | \bar{\theta}^c, \bar{\theta}^d, t \right) \Pi \left(\bar{\theta}^c, t | \bar{\theta}_0^c, t_0 \right) \right) d\bar{\theta}'^c \end{aligned} \quad (26)$$

$$\frac{\partial \Pi \left(\bar{\theta}^d, t | \bar{\theta}_0^d, t_0 \right)}{\partial t} = \sum_i W \left(\bar{\theta}^d | \bar{\theta}_i^d, \bar{\theta}^c, t \right) \Pi \left(\bar{\theta}_i^d, t | \bar{\theta}_0^d, t_0 \right) - W \left(\bar{\theta}_i^d | \bar{\theta}^d, \bar{\theta}^c, t \right) \Pi \left(\bar{\theta}^d, t | \bar{\theta}_0^d, t_0 \right) \quad (27)$$

where:

$$\begin{aligned} A_i \left(\bar{\theta}, t \right) = & \begin{cases} 0 & \text{if } \bar{\theta}_i \in \bar{\theta}^d \\ f \left(\bar{\theta}^c, \bar{\theta}^d, \alpha_{staz}, \alpha_{brow}, t \right) + \frac{1}{2} \frac{\partial b(\bar{\theta}^c, t)}{\partial \bar{\theta}^c} Qb \left(\bar{\theta}^c, t \right) & \text{if } \bar{\theta}_i \notin \bar{\theta}^d \end{cases} \\ B_{i,j} \left(\bar{\theta}, t \right) = & \begin{cases} 0 & \text{if } \bar{\theta}_i \text{ or } \bar{\theta}_j \in \bar{\theta}^d \\ b \left(\bar{\theta}^c, t \right) Qb^T \left(\bar{\theta}^c, t \right) & \text{if } \bar{\theta}_i \text{ or } \bar{\theta}_j \notin \bar{\theta}^d \end{cases} \end{aligned} \quad (28)$$

This system of equations is composed of four main terms that identify four different types of processes:

- Drift process
- Diffusion process
- Jumps in continuous space
- Jumps in discrete space (component state transitions).

These four processes are described in the following sub-sections.

3.1.4.1 Drift Process

The drift process is defined by the Liouville's equation:

$$\frac{\partial \Pi(\bar{\theta}^c, t | \bar{\theta}_0^c, t_0)}{\partial t} = \sum_i \frac{\partial}{\partial \bar{\theta}_i^c} \left(A_i(\bar{\theta}^c, \bar{\theta}^d, t) \Pi(\bar{\theta}^c, t | \bar{\theta}_0^c, t_0) \right) \quad (29)$$

It is important to note that this equation describes a completely deterministic motion, indicated by the equation:

$$\frac{\partial \bar{\theta}^c(t)}{\partial t} = A_i(\bar{\theta}^c, \bar{\theta}^d, t) \quad (30)$$

If $\bar{\theta}^c(\bar{\theta}_0^c, \bar{\theta}^d, t)$ is the solution of Equation 30, then then the solution of the Liouville's equation is:

$$\Pi(\bar{\theta}^c, t | \bar{\theta}_0^c, t_0) = \delta(\bar{\theta}^c - \bar{\theta}^c(\bar{\theta}_0^c, \bar{\theta}^d, t)) \quad (31)$$

provided the initial condition:

$$\Pi(\bar{\theta}^c, t | \bar{\theta}_0^c, t_0) = \delta(\bar{\theta}^c - \bar{\theta}_0^c) \quad (32)$$

3.1.4.2 Diffusion Process

This process is described by the Fokker-Plank equation:

$$\begin{aligned} \frac{\partial \Pi(\bar{\theta}^c, t | \bar{\theta}_0^c, t_0)}{\partial t} = & \sum_i \frac{\partial}{\partial \bar{\theta}_i^c} \left(A_i(\bar{\theta}^c, \bar{\theta}^d, t) \Pi(\bar{\theta}^c, t | \bar{\theta}_0^c, t_0) \right) + \\ & + \frac{1}{2} \sum_{i,j} \frac{\partial^2}{\partial \bar{\theta}_i^c \partial \bar{\theta}_j^c} \left(B_{i,j}(\bar{\theta}^c, \bar{\theta}^d, t) \Pi(\bar{\theta}^c, t | \bar{\theta}_0^c, t_0) \right) \end{aligned} \quad (33)$$

where $A_i(\bar{\theta}^c, \bar{\theta}^d, t)$ is the drift vector and $B_{i,j}(\bar{\theta}^c, \bar{\theta}^d, t)$ is the diffusion matrix.

Provided the initial condition in Equation 32, the Fokker-Plank equation describes a system moving with drift whose velocity is $A(\bar{\theta}^c, \bar{\theta}^d, t)$ on which is imposed a Gaussian fluctuation with covariance matrix $B(\bar{\theta}^c, \bar{\theta}^d, t)$.

3.1.4.3 Jumps in Continuous Space

This process is described by the Master equation:

$$\frac{\partial \Pi(\bar{\theta}^c, t | \bar{\theta}_0^c, t_0)}{\partial t} = \int \left(W(\bar{\theta}^c | \bar{\theta}'^c, \bar{\theta}^d, t) \Pi(\bar{\theta}'^c, t | \bar{\theta}_0^c, t_0) - W(\bar{\theta}'^c | \bar{\theta}^c, \bar{\theta}^d, t) \Pi(\bar{\theta}^c, t | \bar{\theta}_0^c, t_0) \right) d\bar{\theta}'^c \quad (34)$$

Provided the initial condition in Equation 32, it describes a process characterized by straight lines interspersed with discontinuous jumps whose distribution is given by $W(\bar{\theta}^c | \bar{\theta}'^c, \bar{\theta}^d, t)$

3.1.4.4 Jumps in Discrete Space

Transitions in the discrete space can occur in terms of jumps, then the formulation of

$$\frac{\partial \Pi(\bar{\theta}^d, t | \bar{\theta}_0^d, t_0)}{\partial t} = \mathcal{L}_{CK}^d(\Pi(\bar{\theta}^d, t | \bar{\theta}_0^d, t_0)) \quad (35)$$

is similar to the Master equation, recasted for a discrete phase space:

$$\frac{\partial \Pi(\bar{\theta}^d, t | \bar{\theta}_0^d, t_0)}{\partial t} = \sum_i \left(W(\bar{\theta}^d | \bar{\theta}_i^d, \bar{\theta}^c, t) \Pi(\bar{\theta}_i^d, t | \bar{\theta}_0^d, t_0) - W(\bar{\theta}_i^d | \bar{\theta}^d, \bar{\theta}^c, t) \Pi(\bar{\theta}^d, t | \bar{\theta}_0^d, t_0) \right) \quad (36)$$

3.2 RAVEN Entities and Analysis Flow

In the RAVEN code the number and types of possible analyses is potentially large (it is customization to many problem types).

Each basic action (sampling, printing, etc.) is encapsulated in a dedicated object (named “**Entity**”). Each object is inactive till it is connected with other objects in order to perform a more complex process. For example, the *Sampler* entity, aimed to employ a perturbation action, becomes active only in case it gets associated with a *Model*, that is the internal representation of a physical model (e.g., a system code).

RAVEN provides support for several **Entities**, which branch in several different categories/algorithms:

- **RunInfo:**

The RunInfo **Entity** is an information container which describes how the overall computation should be performed. This **Entity** accepts several input settings that 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 Portable Batch System-PBS, etc.).

- **Files:**

The Files **Entity** defines any files that might be needed within the RAVEN run. This could include inputs to the Model, pickled ROM files, or Comma Separated Value (CSV) files for post-processors, to name a few.

- **DataObjects:**

The DataObjects system is a container of data objects of various types that can be constructed during the execution of desired calculation flow. These data objects can be used as input or output for a particular *Model Entity*. Currently RAVEN supports four different data types, each with a particular conceptual meaning:

- *Point* describes the state of the system at a certain point (e.g., 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 point in the phase space (e.g., in time).
- *PointSet* is a collection of individual Point objects. 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 (e.g., in time).
- *History* describes the temporal evolution of the state of the system within a certain input domain;
- *HistorySet* is a collection of individual History objects. It can be considered a mapping between multiple sets of parameters in the input space and the resulting sets of temporal evolution in the output space.

The DataObjects represent the preferred way to transfer the information coming from a Model (e.g., the driven code) to all the other RAVEN systems (e.g., Out-Stream system, Reduced Order Modeling component, etc.).

- **Databases:**

RAVEN provides the capability to store and retrieve data to/from an external database. Currently RAVEN supports only a database type called *HDF5*. This database, depending on the data format it is receiving, will organize itself in a “parallel” or “hierarchical” fashion. The user can create as many database **Entities** as needed.

- **Samplers:**

The Samplers **Entity** is the container of all the algorithms designed to perform the perturbation of the input space. The Samplers can be categorized into three main classes:

- *Forward*. Sampling strategies that do not leverage the information coming from already evaluated realizations in the input space. For example, Monte-Carlo, Stratified (LHS), Grid, Response Surface, Factorial Design, Sparse Grid, etc.
- *Adaptive*. Sampling strategies that take advantages of the information coming from already evaluated realizations of the input space, adapting the sampling strategies to key figures of merits. For example, Limit Surface search, Adaptive HDMR, etc.
- *Dynamic Event Tree*. Sampling strategies that perform the exploration of the input space based on the dynamic evolution of the system, employing branching techniques. For example, Dynamic Event Tree, Hybrid Dynamic Event Tree, etc.

- **OutStreams:**

The OutStreams node is the **Entity** used for data exporting and dumping. The OutStreams support 2 actions:

- *Print*. This Out-Stream is able to print out (in a Comma Separated Value format) all the information contained in:
 - * DataObjects
 - * Reduced Order Models.
- *Plot*. This Out-Stream is able to plot 2-Dimensional, 3-Dimensional, 4-Dimensional (using color mapping) and 5-Dimensional (using marker size). Several types of plot are available, such as scatter, line, surfaces, histograms, pseudo-colors, contours, etc.

- **Distributions:**

The Distributions **Entity** is the container of all the stochastic representations of random variables. Currently, RAVEN supports:

- *1-Dimensional* continuous and discrete distributions, such as Normal, Weibull, Binomial, etc.
- *N-Dimensional* distributions, such as Multivariate Normal, user-inputted N-Dimensional distributions.

- **Models:**

The Models **Entity** represents the projection from the input to the output space. Currently, RAVEN defines the following sub-categories:

- *Code*, the sub- **Entity** that represent the driven code, through external code interfaces (see [13])
- *ExternalModel*, the sub- **Entity** that represents a physical or mathematical model that is directly implemented by the user in a Python module
- *ROM*, the sub- **Entity** that represent the Reduced Order Model, interfaced with several algorithms
- *PostProcessor*, the sub- **Entity** that is used to perform action on data, such as computation of statistical moments, correlation matrices, etc.

The Model **Entity** can be seen as a transfer function between the input and output space.

- **Functions:**

The Functions **Entity** is the container of all the user-defined functions, such as Goal Functions in adaptive sampling strategies, etc.

All these action-objects are combined together to create a peculiar analysis flow, which is specified by the user in an additional **Entity** named **Steps**. This **Entity** represents the core of the analysis,

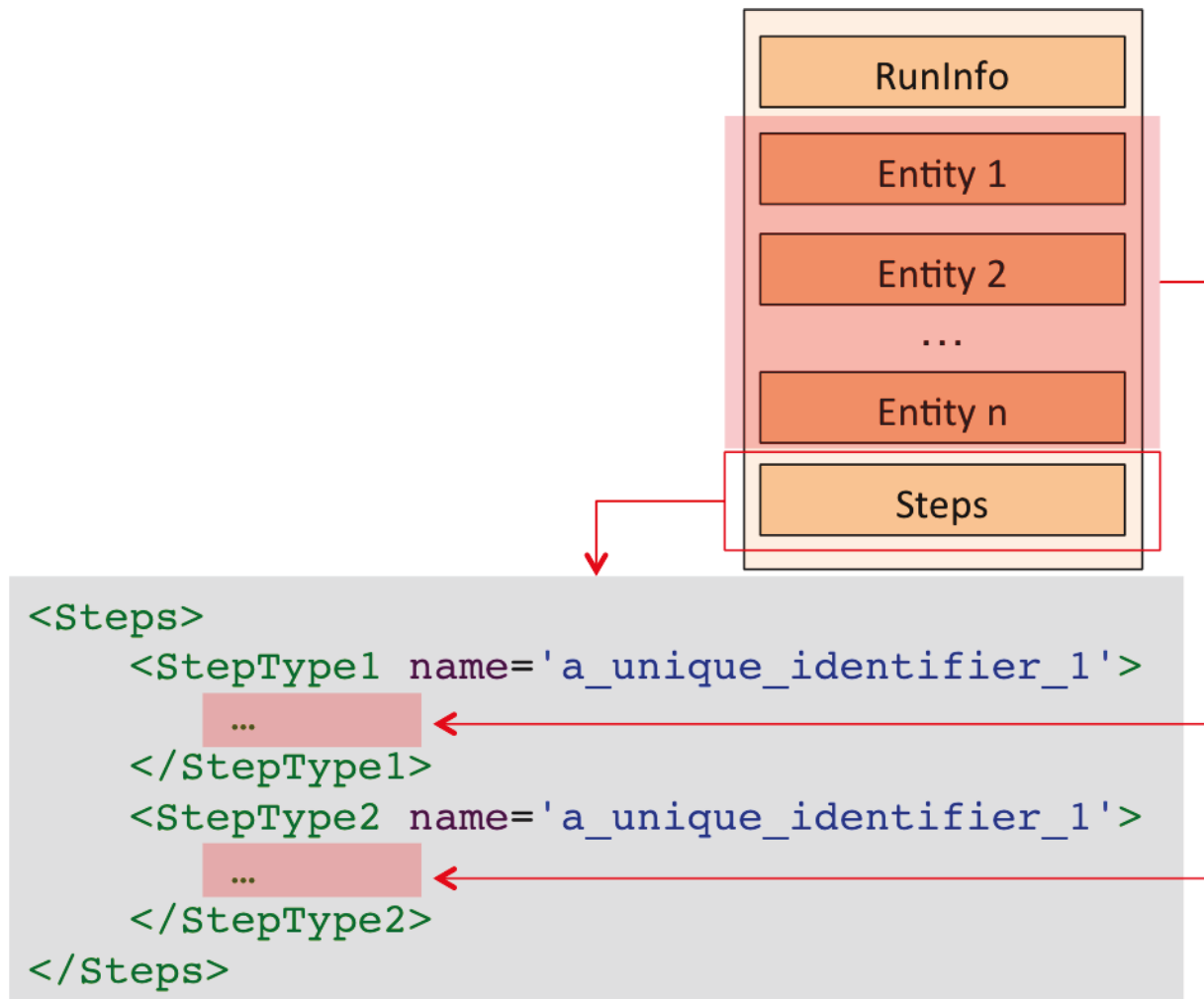


Figure 6. Example of the Steps **Entity** and its connection in the input file.

since it is the location where the multiple objects get finally linked in order to perform a combined action on a certain *Model* (see Fig. 6). In order to perform this linking, each **Entity** defined in the Step needs to “play” a Role:

- *Input*
- *Output*
- *Model*
- *Sampler*

- *Function*
- *ROM*
- *SolutionExport*, the **Entity** that is used to export the solution of a *Sampler*.

Currently, RAVEN supports 4 different types of *Steps*:

- *SingleRun*, perform a single run of a model
- *MultiRun*, perform multiple runs of a model
- *RomTrainer*, perform the training of a Reduced Order Model (ROM)
- *PostProcess*, post-process data or manipulate RAVEN entities
- *IOStep*, step aimed to perform multiple actions:
 - construct/update a Database from a DataObjects and vice-versa
 - construct/update a Database or a DataObjects object from CSV files
 - stream the content of a Database or a DataObjects out through an OutStream
 - store/retrieve a ROM to/from an external File using Pickle module of Python

3.3 Raven Input Structure

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

- **<Simulation>**: The root node containing the entire input, all of the following blocks fit inside the *Simulation* block.
- **<RunInfo>**: Specifies the calculation settings (number of parallel simulations, etc.).
- **<Files>**: Specifies the files to be used in the calculation.
- **<Distributions>**: Defines distributions needed for describing parameters, etc.
- **<Samplers>**: Sets up the strategies used for exploring an uncertain domain.
- **<DataObjects>**: Specifies internal data objects used by RAVEN.

- **<Databases>**: Lists the HDF5 databases used as input/output to a RAVEN run.
- **<OutStreams>**: Visualization and Printing system block.
- **<Models>**: Specifies codes, ROMs, post-processing analysis, etc.
- **<Functions>**: Details interfaces to external user-defined functions and modules. the user will be building and/or running.
- **<Steps>**: Combines other blocks to detail a step in the RAVEN workflow including I/O and computations to be performed.

Each of these blocks are explained in dedicated sections of the user manual [13].

4 Manual Formats

In order to highlight some parts of the manual having a particular meaning (input structure, examples, terminal commands, etc.), specific formats have been used. This section provides the formats with a specific meaning:

- ***Python Coding:***

```
class AClass():
    def aMethodImplementation(self):
        pass
```

- ***XML input example:***

```
<MainXMLBlock>
...
<aXMLnode name='anObjectName' anAttribute='aValue'>
    <aSubNode>body</aSubNode>
</aXMLnode>
<!-- This is commented block -->
...
</MainXMLBlock>
```

- ***Bash Commands:***

```
cd trunk/raven/
./raven_libs_script.sh
cd ../../
```

5 Manual Structure

This manual is intended to provide an overview of the RAVEN capabilities through the explanation of multiple commented examples. To speed up the learning process, the examples are organized in an ascending complexity order, from simple data manipulation and visualization to full Probabilistic Risk Assessment and Uncertainty Quantification analysis. In addition, each example is followed by a brief explanation of the theoretical background of the methods that have been used. It is important to notice that this document is intended to be consulted in conjunction with the user manual [13].

To generalize the examples to any driven software, a simple `Python` code (conventionally called **AnalyticBateman**) has been developed (located at “*doc/user_guide/ravenInputs/physicalCode*”). It solves a system of ordinary differential equations (ODEs), of the form:

$$\begin{cases} \frac{d\mathbf{X}}{dt} = \mathbf{S} - \mathbf{L} \\ \mathbf{X}(t = 0) = \mathbf{X}_0 \end{cases} \quad (37)$$

where:

- \mathbf{X}_0 , initial conditions
- \mathbf{S} , source terms
- \mathbf{L} , loss terms

For example, this code is able to solve a system of equations as follows:

$$\begin{cases} \frac{dx_1}{dt} = \phi(t) \times \sigma_{x_1} - \lambda_{x_1} \\ \frac{dx_2}{dt} = \phi(t) \times \sigma_{x_2} - \lambda_{x_2} + x_1(t) \times \lambda_{x_1} \\ x_1(t = 0) = x_1^0 \\ x_2(t = 0) = 0.0 \end{cases} \quad (38)$$

The input of the **AnalyticBateman** code is in XML format. For example, the following is the reference input for a system of 4 Ordinary Differential Equations (ODEs) that is going to be used for all the examples reported in this manual:

```
<AnalyticBateman>
  <totalTime>300</totalTime>
  <powerHistory>1 1 1</powerHistory>
  <flux>1.e14 1.e14 1.e14</flux>
```

```

<stepDays>0 100 200 400</stepDays>
<timeSteps>3 5 5</timeSteps>
<nuclides>
  <A>
    <equationType>N1</equationType>
    <initialMass>1.0</initialMass>
    <decayConstant>0.000000005</decayConstant>
    <sigma>8</sigma>
    <ANumber>230</ANumber>
  </A>
  <B>
    <equationType>N2</equationType>
    <initialMass>1.0</initialMass>
    <decayConstant>0.000000007</decayConstant>
    <sigma>5</sigma>
    <ANumber>200</ANumber>
  </B>
  <C>
    <equationType>N3</equationType>
    <initialMass>1.0</initialMass>
    <decayConstant>0.000000008</decayConstant>
    <sigma>3</sigma>
    <ANumber>150</ANumber>
  </C>
  <D>
    <equationType>N4</equationType>
    <initialMass>1.0</initialMass>
    <decayConstant>0.000000009</decayConstant>
    <sigma>1</sigma>
    <ANumber>100</ANumber>
  </D>
</nuclides>
</AnalyticBateman>

```

The code outputs the time evolution of the 4 variables (A, B, C, D) in a CSV file, producing the following output:

RAVEN is able to directly retrieve CSV files as output; for this reason, the *GenericInterface* (see [13]-Chapter “Existing Interfaces”) is going to be used to drive the code.

Table 1. Reference case sample results.

time	A	C	B	D
0	1.0	1.0	1.0	1.0
2880000.0	0.983434738239	0.977851848235	1.01011506729	1.01013172275
5760000.0	0.967143884376	0.956202457404	1.01936231677	1.02036100400
8640000.0	0.951122892771	0.935040450532	1.02777406275	1.03067925987
10368000.0	0.941637968936	0.922572556179	1.03243314106	1.03690947068
12096000.0	0.932247632016	0.910273757371	1.03680933440	1.04316700086
13824000.0	0.922950938758	0.898141730426	1.04090912054	1.04945015916
15552000.0	0.913746955315	0.886174183908	1.04473885709	1.05575729317
17280000.0	0.904634757153	0.874368858183	1.04830478357	1.06208678854
20736000.0	0.886682064542	0.851235986899	1.05466958557	1.07480659230
24192000.0	0.869085647400	0.828725658721	1.06005115510	1.08759739100
27648000.0	0.851838435355	0.806820896763	1.06449535534	1.10044757060
31104000.0	0.834933498348	0.785505191756	1.06804634347	1.11334606143
34560000.0	0.818364043850	0.764762489077	1.07074662835	1.12628231792

6 Run a Single Instance of a Code and Load the Outputs

The simplest exercise that can be performed is to run the driven code (**AnalyticBateman** in our example), loading the results of a single run into RAVEN, printing and plotting some variables. As detailed in the RAVEN user manual ([13]-Chapters “DataObjects” and “Databases”) and in Chapter /refsub:EntitiesAndFlow RAVEN uses two classes of objects to store the data coming from a driven code (outputs):

- **DataObjects:** The DataObjects represent the preferred way to transfer the information coming from a Model (the driven code, in this case) to all the other RAVEN systems (e.g. OutStream system, Reduced Order Modeling component, etc.).
- **Databases.**

As easily inferable from the user manual ([13]-Chapter “OutStream”), the DataObjects can be exported into a CSV file and plotted (2-D and 3-D plots) linking them into the OutStream system. The following subsections report examples on how to use these systems running a single instance of the driven code.

6.1 Single Run using the OutStream system for printing and create basic plots

In this Section, the user can learn how to use RAVEN to run a single instance of a driven code, plotting and printing the results.

The goal of this Section is to learn how to:

1. Set up a simple RAVEN input for running a driven code;
2. Load the output of the code into the RAVEN DataObjects system;
3. Print out what contained in the DataObjects;
4. Generate basic plots of the code results.

In order to accomplish these tasks, the following RAVEN **Entities** (XML blocks in the input files) are needed:

1. **RunInfo:**

```

<RunInfo>
  <Sequence>Single, write-History</Sequence>
  <WorkingDir>SectionVI.I</WorkingDir>
  <batchSize>1</batchSize>
</RunInfo>

```

As reported in Section 3.2, the *RunInfo* **Entity** is intended to set up the analysis that the user wants to perform. In this specific case, two steps (**<Sequence>**) are going to be sequentially run using a single processor (**<batchSize>**).

2. *Files:*

```

<Files>
  <Input name="referenceInput.xml"
    type="input">referenceInput.xml</Input>
</Files>

```

Since the driven code uses a single input file, in this Section the original input is placed. As described in the user manual [] the attribute **name** represents the alias that is going to be used in all the other input blocks in order to refer to this file.

3. *Models:*

```

<Models>
  <Code name="testModel" subType="GenericCode">
    <executable>
      ../physicalCode/analyticalbateman/AnalyticalDplMain.py
    </executable>
    <clargs arg="python" type="prepend"/>
    <clargs arg="" extension=".xml" type="input"/>
    <clargs arg="" extension=".csv" type="output"/>
    <prepend>python</prepend>
  </Code>
</Models>

```

Since the driven code already dumps its outputs in CSV format, there is no need to create an ad-hoc code interface and the GenericCode interface can be directly used. In addition, since the **AnalyticBateman** code is written in Python, it is necessary to specify that the code needs to be run pre-pending the expression “Python”.

4. *DataObjects:*

```

<DataObjects>
  <PointSet name="pointValues">
    <Input>InputPlaceholder</Input>

```

```

    <Output>A,B,C,D</Output>
  </PointSet>
  <HistorySet name="history">
    <Input>InputPlaceholder</Input>
    <Output>A,B,C,D,time</Output>
  </HistorySet>
</DataObjects>

```

In this block, two *DataObjects* are defined: 1) PointSet named “pointValues”, 2) History-Set named “history”. Note that a special keyword is inputted in the **<Input>** node. This keyword is used when a *DataObjects Entity* needs to be constructed without any linking with respect to the input space. Indeed, in this case, the model input space is not perturbed though a sampling strategies; the code is executed through the original input file (“referenceInput.xml”). In the **<Output>** node all the requested variables are inputted.

5. OutStreams:

```

<OutStreams>
  <Print name="pointValues">
    <type>csv</type>
    <source>pointValues</source>
  </Print>
  <Print name="history">
    <type>csv</type>
    <source>history</source>
  </Print>
  <Plot dim="2" name="historyPlot" overwrite="false"
    verbosity="debug">
    <plotSettings>
      <plot>
        <type>line</type>
        <x>history|Output|time</x>
        <y>history|Output|A</y>
        <color>blue</color>
      </plot>
      <plot>
        <type>line</type>
        <x>history|Output|time</x>
        <y>history|Output|B</y>
        <color>red</color>
      </plot>
      <plot>
        <type>line</type>
        <x>history|Output|time</x>

```

```

        <y>history|Output|C</y>
        <color>yellow</color>
    </plot>
    <plot>
        <type>line</type>
        <x>history|Output|time</x>
        <y>history|Output|D</y>
        <color>black</color>
    </plot>
    <xlabel>time (s)</xlabel>
    <ylabel>evolution (kg)</ylabel>
</plotSettings>
<actions>
    <how>png,screen</how>
    <title>
        <text> </text>
    </title>
</actions>
</Plot>
</OutStreams>

```

In this block, both the Out-Stream types are constructed:

- *Print*:
 - named “pointValues” connected with the *DataObjects* **Entity** “pointValues” (**<source>**)
 - named “history” connected with the *DataObjects* **Entity** “history” (**<source>**)

When this objects get used, all the information contained in the linked *DataObjects* are going to be dumped in CSV files (**<type>**).

- *Plot*: a single **<Plot>** **Entity** is defined, containing the line plots of the 4 output variables (*A, B, C, D*) in the same figure. This object is going to generate a PNG file and an interactive Plot on the screen.

6. Steps:

```

<Steps>
    <SingleRun name="Single">
        <Input class="Files"
            type="input">referenceInput.xml</Input>
        <Model class="Models"
            type="Code">testModel</Model>
        <Output class="DataObjects"
            type="PointSet">pointValues</Output>
    </SingleRun>
</Steps>

```



```

<Output class="DataObjects"
  type="HistorySet">history</Output>
<Output class="OutStreams"
  type="Print">pointValues</Output>
</SingleRun>
<IOStep name="writeHistory" pauseAtEnd="True">
  <Input class="DataObjects"
    type="HistorySet">history</Input>
  <Output class="OutStreams"
    type="Print">history</Output>
  <Output class="OutStreams"
    type="Plot">historyPlot</Output>
</IOStep>
</Steps>

```

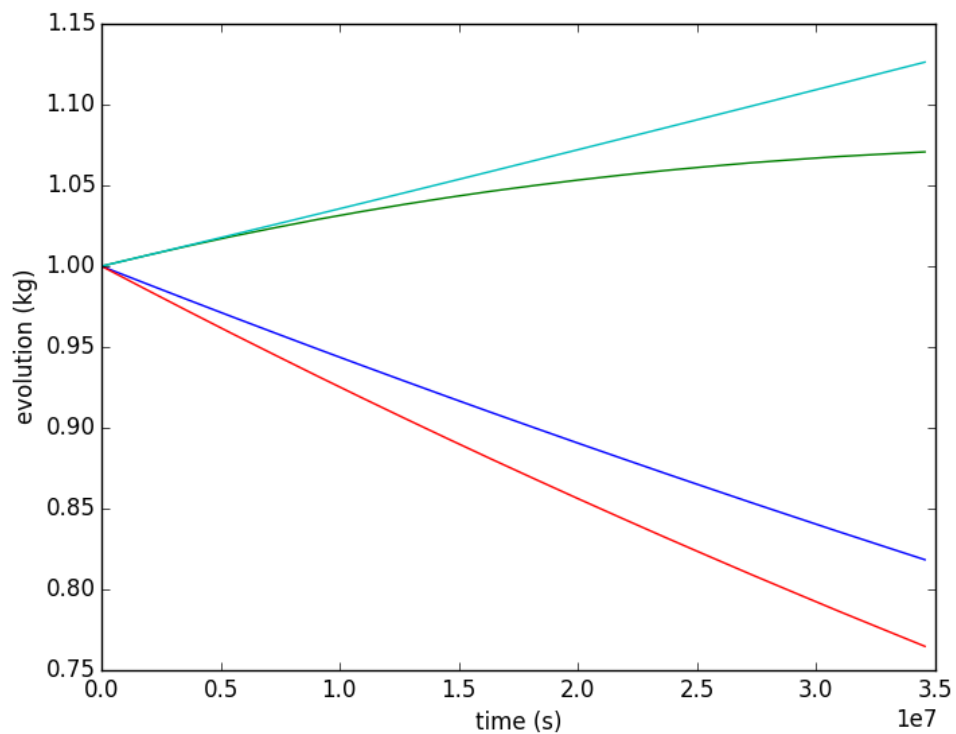


Figure 7. Plot of the history for variables A, B, C, D .

Finally, all the previously defined **Entities** can be combined in the **<Steps>** block. Thus, two **<Steps>** have been inputted:

- **<SingleRun>** named “Single”, used to run the single instance of the driven code and collect the outputs in the two *DataObjects*. In addition, it can be seen that an additional object has been placed among the **<Output>**(s). Indeed, an *OutStreams* can be an **<Output>** in any Step type (as long as the linked *DataObjects* plays a whatever role in the Step)
- **<IOStep>** named “writeHistory”, used to 1) dump the “history” *DataObjects* **Entity** in a CSV file and 2) plot the data in the PNG file and on the screen.

Tables 2 and 3 show the results dumped by the OutStreams *Print*. As previously mentioned, Figure 7 reports the four plots (four variables) drawn in the same picture.

Table 2. “history” HistorySet CSV output file.

A	C	B	D	time
1	1	1	1	0
0.983434738	0.977851848	1.010115067	1.010131723	2880000
0.967143884	0.956202457	1.019362317	1.020361004	5760000
0.951122893	0.935040451	1.027774063	1.03067926	8640000
0.941637969	0.922572556	1.032433141	1.036909471	10368000
0.932247632	0.910273757	1.036809334	1.043167001	12096000
0.922950939	0.89814173	1.040909121	1.049450159	13824000
0.913746955	0.886174184	1.044738857	1.055757293	15552000
0.904634757	0.874368858	1.048304784	1.062086789	17280000
0.886682065	0.851235987	1.054669586	1.074806592	20736000
0.869085647	0.828725659	1.060051155	1.087597391	24192000
0.851838435	0.806820897	1.064495355	1.100447571	27648000
0.834933498	0.785505192	1.068046343	1.113346061	31104000
0.818364044	0.764762489	1.070746628	1.126282318	34560000

Table 3. “pointValues” PointSet CSV output file.

InputPlaceholder	A	C	B	D
0.0	0.81836404385	0.764762489077	1.07074662835	1.12628231792

6.2 Single Run using the OutStream System to Sub-plot and Selectively print.

This Section shows how to use RAVEN to create sub-plots (multiple plots in the same figure) and how to select only some variable from the *DataObjects* in the *Print* OutStream.

The goals of this Section are about learning how to:

1. Print out what contained in the DataObjects, selecting only few variables
2. Generate sub-plots (multiple plots in the same figure) of the code results

To accomplish these tasks, the *OutStreams* **Entity** in the input defined in the previous Section (6.1) needs to be modified as follows:

1. *Print*:

```
<Print name="pointValues">
  <type>csv</type>
  <source>pointValues</source>
  <what>Output</what>
</Print>
<Print name="history">
  <type>csv</type>
  <source>history</source>
  <what>Output | A, Output | D</what>
</Print>
```

With respect to the *Print* nodes defined in the previous Section (6.1), it can be noticed that an additional node has been added: **<what>**. The *Print* **Entity** “pointValues” is going to extract and dump only the variables that are part of the Output space (*A*, *B*, *C*, *D* and not *InputPlaceHolder*). The *Print* **Entity** “history” is instead going to print the Output space variables *A* and *D*.

2. *Plot*:

```
<Plot dim="2" name="historyPlot" overwrite="false"
  verbosity="debug">
  <plotSettings>
    <gridSpace>2 2</gridSpace>
    <plot>
      <type>line</type>
      <x>history|Output|time</x>
      <y>history|Output|A</y>
      <color>blue</color>
      <gridLocation>
        <x>0</x>
        <y>0</y>
      </gridLocation>
```

```

</plot>
<plot>
  <type>line</type>
  <x>history|Output|time</x>
  <y>history|Output|B</y>
  <color>red</color>
  <gridLocation>
    <x>1</x>
    <y>0</y>
  </gridLocation>
</plot>
<plot>
  <type>line</type>
  <x>history|Output|time</x>
  <y>history|Output|C</y>
  <color>yellow</color>
  <gridLocation>
    <x>0</x>
    <y>1</y>
  </gridLocation>
</plot>
<plot>
  <type>line</type>
  <x>history|Output|time</x>
  <y>history|Output|D</y>
  <color>black</color>
  <gridLocation>
    <x>1</x>
    <y>1</y>
  </gridLocation>
</plot>
<xlabel>time (s)</xlabel>
<ylabel>evolution (kg)</ylabel>
</plotSettings>
<actions>
  <how>png,screen</how>
  <title>
    <text> </text>
  </title>
</actions>
</Plot>

```

Note that the *Plot* **Entity** does not differ much with respect to the one in Section 6.1: 1)

the additional sub-node `<gridSpace>` has been added. This node is needed to define how the figure needs to be partitioned (discretization of the grid). In this case a 2 by 2 grid is requested. 2) in each `<plot>` the node `<gridLocation>` is placed in order to specify in which position the relative plot needs to be placed. For example, in the following grid location, the relative plot is going to be placed at the bottom-right corner.

```
<gridLocation>
  <x>1</x>
  <y>1</y>
</gridLocation>
```

The CSV tables generated by the *Print Entities* are not reported, since the only differences with respect to Tables 2 and 3 are related to the number of columns (variables) dumped out. Figure 8 reports the four plots (four variables) drawn in the same picture.

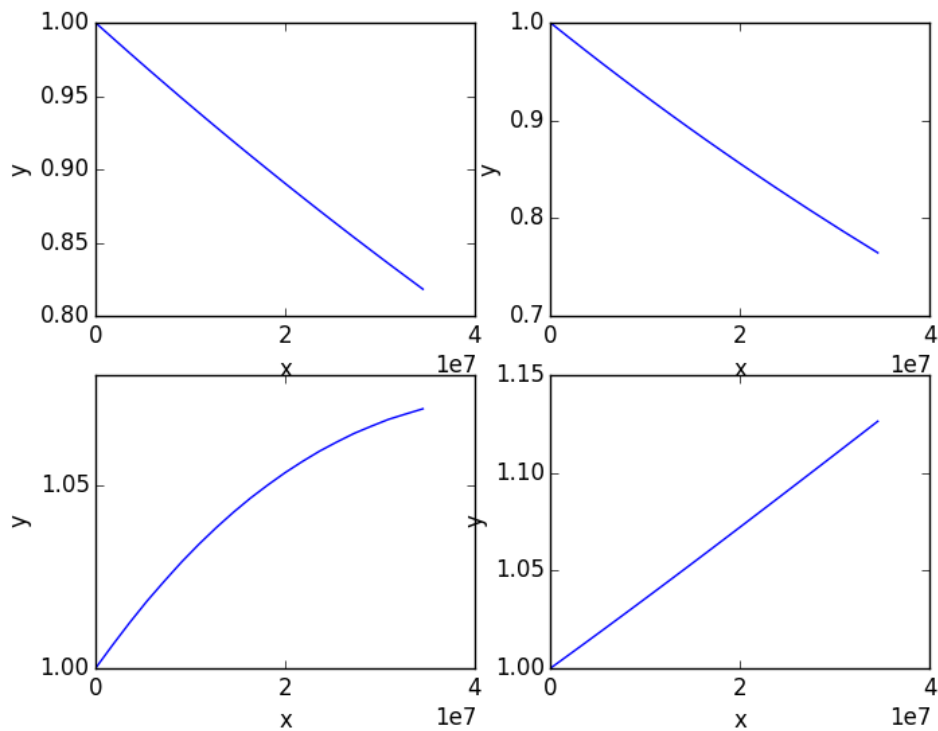


Figure 8. Subplot of the history for variables A, B, C, D .

7 Forward Sampling Strategies

In order to perform UQ and dynamic probabilistic risk assessment (DPRA), a sampling strategy needs to be employed. The sampling strategy aims to perturb the input space (domain of the uncertainties) to explore the response of a complex system in relation to selected FOMs.

The most widely used strategies to perform UQ and PRA are generally collected into the category that, in RAVEN, is called **Forward** samplers. The **Forward** sampler category collects all the strategies that perform the sampling of the input space without exploiting, through learning approaches, the information made available from the outcomes of evaluation previously performed (adaptive sampling) and the common system evolution (patterns) that different sampled calculations can generate in the phase space (Dynamic Event Tree).

As mentioned in Section 2.2.4, RAVEN has different **Forward** samplers:

- *Monte-Carlo*
- *Grid-based*
- *Stratified* and its specialization named *Latin Hyper Cube*.

In addition, RAVEN possesses advanced **Forward** sampling strategies that:

- Build a grid in the input space selecting evaluation points based on characteristic quadratures as part of stochastic collocation for generalized polynomial chaos method (*Sparse Grid Collocation* sampler);
- Use high-density model reduction (HDMR) a.k.a. Sobol decomposition to approximate a function as the sum of increasing complexity interactions (*Sobol* sampler).

In the following subsections, the theory behind these sampling methodologies is going to be explained by way of applied RAVEN examples.

7.1 Monte-Carlo

The Monte-Carlo method is probably one of the most used methodologies in several disciplines. In this section, a brief theoretical background is reported. In addition, it is shown how to employ this methodology with RAVEN.

7.1.1 Monte-Carlo

The Monte-Carlo method approximates an expectation by the sample mean of a function of simulated random variables. It is based on the laws of large numbers in order to approximate expectations. In other words, it approximates the average response of multiple FOMs relying on multiple random sampling of the input space.

Consider a random variable (eventually multidimensional) X having probability mass function or probability density function $pdf_X(x)$, which is greater than zero on a set of values χ . Then the expected value of a function f of X is as follows:

$$\begin{aligned}\mathbb{E}(f(X)) &= \sum_{x \in \chi} f(x) pdf_X(x) && \text{if } X \text{ discrete} \\ \mathbb{E}(f(X)) &= \int_{x \in \chi} f(x) pdf_X(x) && \text{if } X \text{ continuous}\end{aligned}\tag{39}$$

Now consider n samples of X , (x_1, \dots, x_n) , and compute the mean of $f(x)$ over the samples. This computation represents the Monte-Carlo estimate:

$$\mathbb{E}(f(X)) \approx \tilde{f}_n(x) = \frac{1}{n} \sum_{i=1}^n f(x_i)\tag{40}$$

If $\mathbb{E}(f(X))$ exists, then the law of large numbers determines that for any arbitrarily small ε :

$$\lim_{n \rightarrow \infty} P(|\tilde{f}_n(X) - \mathbb{E}(f(X))| \geq \varepsilon) = 0\tag{41}$$

The above equation suggests that as n gets larger, then the probability that $\tilde{f}_n(X)$ deviates from the $\mathbb{E}(f(X))$ becomes smaller. In other words, the more samples are spooned, the more closer the Monte-Carlo estimate of X gets to the real value.

In addition, $\tilde{f}_n(X)$ represent an unbiased estimate for $\mathbb{E}(f(X))$:

$$\mathbb{E}(\tilde{f}_n(X)) = \mathbb{E}\left(\frac{1}{n} \sum_{i=1}^n f(x_i)\right) = \frac{1}{n} \sum_{i=1}^n \mathbb{E}(f(x_i)) = \mathbb{E}(f(X))\tag{42}$$

7.1.2 Monte-Carlo sampling through RAVEN

The goals of this section are about learning how to:

1. Set up a simple Monte-Carlo sampling for perturbing the input space of a driven code
2. Load the outputs of the code into the RAVEN DataObjects system (HistorySet and PointSet)
3. Print on file what contained in the DataObjects

4. Generate plots of the code results.

In order to accomplish these tasks, the following RAVEN **Entities** (XML blocks in the input files) are needed:

1. *RunInfo*:

```
<RunInfo>
  <JobName>ChapterVII-I/MonteCarlo</JobName>
  <Sequence>sample,writeHistories</Sequence>
  <WorkingDir>ChapterVII-I/MonteCarlo</WorkingDir>
  <batchSize>12</batchSize>
</RunInfo>
```

As reported in Section 3.2, the *RunInfo* **Entity** is intended to set up the analysis that the user wants to perform. In this specific case, two steps (**<Sequence>**) are sequentially run using 12 processors (**<batchSize>**). This means that 12 instances of the driven code are run simultaneously. Every time a simulation ends, a new one is launched.

2. *Files*:

```
<Files>
  <Input name="referenceInput.xml"
    type="input">referenceInput.xml</Input>
</Files>
```

Since the driven code uses a single input file, in this section the original input is placed. As detailed in the user manual the attribute **name** represents the alias that is going to be used in all the other input blocks in order to refer to this file.

3. *Models*:

```
<Models>
  <Code name="testModel" subType="GenericCode">
    <executable>
      ../physicalCode/analyticalbateman/AnalyticalDplMain.py
    </executable>
    <clargs arg="python" type="prepend"/>
    <clargs arg="" extension=".xml" type="input"/>
    <clargs arg="" extension=".csv" type="output"/>
    <prepend>python</prepend>
  </Code>
</Models>
```


The Model here is represented by the **AnalyticalBateman**, which already exports its output file in a CSV format (standard format that RAVEN can read). For this reason, the *Generic-Code* interface is used.

4. *Distributions*:

```
<Distributions>
  <Uniform name="sigma">
    <lowerBound>1</lowerBound>
    <upperBound>10</upperBound>
  </Uniform>
  <Uniform name="decayConstant">
    <lowerBound>0.000000005</lowerBound>
    <upperBound>0.000000010</upperBound>
  </Uniform>
</Distributions>
```

In the Distributions XML section, the stochastic model for the uncertainties treated by the Monte-Carlo sampling is reported. In this case two distributions are defined:

- $\sigma \sim \mathbb{U}(1, 10)$, used to model the uncertainties associated with the Model $\sigma(s)$
- $\text{decayConstant} \sim \mathbb{U}(0.5e-8, 1e-8)$, used to model the uncertainties associated with the Model *decay constants*.

5. *Samplers*:

```
<Samplers>
  <MonteCarlo name="monteCarlo">
    <samplerInit>
      <limit>100</limit>
    </samplerInit>
    <variable name="sigma-A">
      <distribution>sigma</distribution>
    </variable>
    <variable name="decay-A">
      <distribution>decayConstant</distribution>
    </variable>
    <variable name="sigma-B">
      <distribution>sigma</distribution>
    </variable>
    <variable name="decay-B">
      <distribution>decayConstant</distribution>
    </variable>
    <variable name="sigma-C">
      <distribution>sigma</distribution>
    </variable>
  </MonteCarlo>
</Samplers>
```

```

</variable>
<variable name="decay-C">
  <distribution>decayConstant</distribution>
</variable>
<variable name="sigma-D">
  <distribution>sigma</distribution>
</variable>
<variable name="decay-D">
  <distribution>decayConstant</distribution>
</variable>
</MonteCarlo>
</Samplers>

```

To employ the Monte-Carlo sampling strategy (100 samples), a **<MonteCarlo>** node needs to be inputted. The Monte-Carlo method is going to be employed on *eight* model variables. Note that all the *decay-%* and *sigma-%* variables are associated with the same distributions *decayConstant* and *sigma*, respectively.

6. *DataObjects*:

```

<DataObjects>
  <PointSet name="samples">
    <Input>
      sigma-A, sigma-B, sigma-C, sigma-D,
      decay-A, decay-B, decay-C, decay-D
    </Input>
    <Output>A, B, C, D, time</Output>
  </PointSet>
  <HistorySet name="histories">
    <Input>
      sigma-A, sigma-B, sigma-C, sigma-D,
      decay-A, decay-B, decay-C, decay-D
    </Input>
    <Output>A, B, C, D, time</Output>
  </HistorySet>
</DataObjects>

```

In this block, two *DataObjects* are defined: 1) PointSet named “samples”, 2) HistorySet named “histories”. Note that in the **<Input>** node all the uncertainties perturbed through the Monte-Carlo strategy are listed. By this, any realization in the input space is linked to the outputs listed in the **<Output>** node.

7. *OutStreams*:

```

<OutStreams>
  <Print name="samples">
    <type>csv</type>
    <source>samples</source>
  </Print>
  <Print name="histories">
    <type>csv</type>
    <source>histories</source>
  </Print>
  <Plot dim="2" name="historiesPlot" overwrite="false"
    verbosity="debug">
    <plotSettings>
      <gridSpace>2 2</gridSpace>
      <plot>
        <type>line</type>
        <x>histories|Output|time</x>
        <y>histories|Output|A</y>
        <color>blue</color>
        <gridLocation>
          <x>0</x>
          <y>0</y>
        </gridLocation>
        <xlabel>time (s)</xlabel>
        <ylabel>evolution A(kg)</ylabel>
      </plot>
      <plot>
        <type>line</type>
        <x>histories|Output|time</x>
        <y>histories|Output|B</y>
        <color>red</color>
        <gridLocation>
          <x>1</x>
          <y>0</y>
        </gridLocation>
        <xlabel>time (s)</xlabel>
        <ylabel>evolution B(kg)</ylabel>
      </plot>
      <plot>
        <type>line</type>
        <x>histories|Output|time</x>
        <y>histories|Output|C</y>
        <color>yellow</color>

```

```

        <gridLocation>
            <x>0</x>
            <y>1</y>
        </gridLocation>
        <xlabel>time (s)</xlabel>
        <ylabel>evolution C(kg)</ylabel>
    </plot>
    <plot>
        <type>line</type>
        <x>histories|Output|time</x>
        <y>histories|Output|D</y>
        <color>black</color>
        <gridLocation>
            <x>1</x>
            <y>1</y>
        </gridLocation>
        <xlabel>time (s)</xlabel>
        <ylabel>evolution D(kg)</ylabel>
    </plot>
</plotSettings>
<actions>
    <how>png,screen</how>
    <title>
        <text> </text>
    </title>
</actions>
</Plot>
<Plot dim="3" name="samplesPlot3D" overwrite="false"
    verbosity="debug">
    <plotSettings>
        <gridSpace>2 2</gridSpace>
        <plot>
            <type>scatter</type>
            <x>samples|Input|sigma-A</x>
            <y>samples|Input|decay-A</y>
            <z>samples|Output|A</z>
            <color>blue</color>
            <gridLocation>
                <x>0</x>
                <y>0</y>
            </gridLocation>
            <xlabel>sigma</xlabel>

```

```

        <ylabel>decay</ylabel>
        <zlabel>final A</zlabel>
    </plot>
    <plot>
        <type>scatter</type>
        <x>samples|Input|sigma-B</x>
        <y>samples|Input|decay-B</y>
        <z>samples|Output|B</z>
        <color>red</color>
        <gridLocation>
            <x>1</x>
            <y>0</y>
        </gridLocation>
        <xlabel>sigma</xlabel>
        <ylabel>decay</ylabel>
        <zlabel>final B</zlabel>
    </plot>
    <plot>
        <type>scatter</type>
        <x>samples|Input|sigma-C</x>
        <y>samples|Input|decay-C</y>
        <z>samples|Output|C</z>
        <color>yellow</color>
        <gridLocation>
            <x>0</x>
            <y>1</y>
        </gridLocation>
        <xlabel>sigma</xlabel>
        <ylabel>decay</ylabel>
        <zlabel>final C</zlabel>
    </plot>
    <plot>
        <type>scatter</type>
        <x>samples|Input|sigma-D</x>
        <y>samples|Input|decay-D</y>
        <z>samples|Output|D</z>
        <color>black</color>
        <gridLocation>
            <x>1</x>
            <y>1</y>
        </gridLocation>
        <xlabel>sigma</xlabel>

```

```

        <ylabel>decay</ylabel>
        <zlabel>final D</zlabel>
    </plot>
    <xlabel>sigma</xlabel>
    <ylabel>decay</ylabel>
    <zlabel>final response</zlabel>
</plotSettings>
<actions>
    <how>png,screen</how>
    <title>
        <text> </text>
    </title>
</actions>
</Plot>
</OutStreams>

```

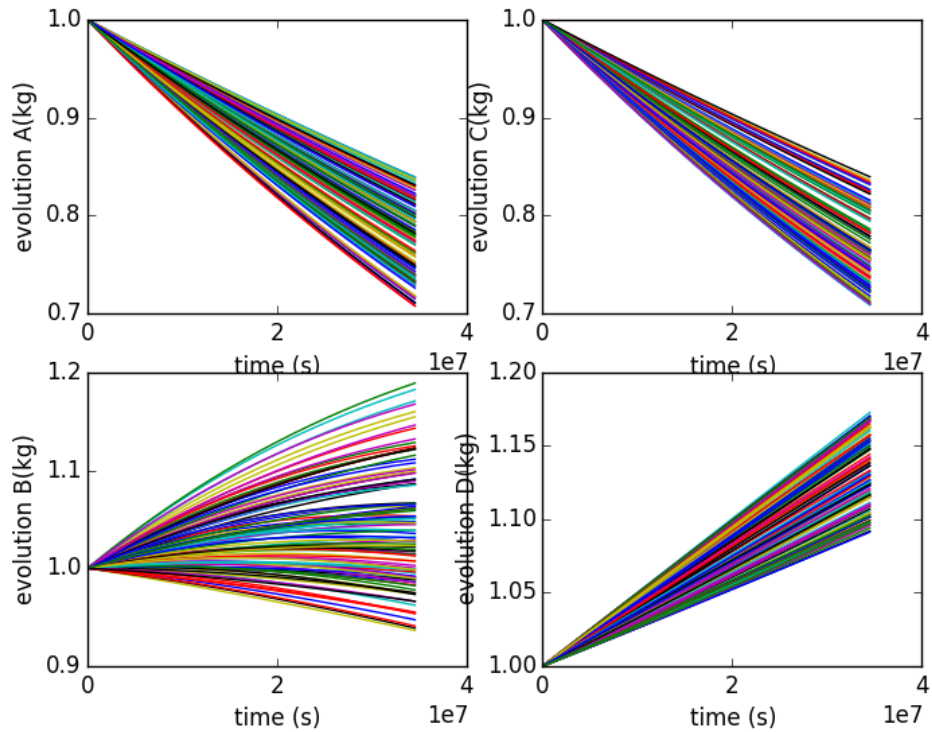


Figure 9. Plot of the histories generated by the Monte Carlo sampling for variables A, B, C, D .

In this block, both the Out-Stream types are constructed:

- *Print*:
 - named “samples” connected with the *DataObjects* **Entity** “samples” (**<source>**)
 - named “histories” connected with the *DataObjects* **Entity** “histories” (**<source>**)

When these objects get used, all the information contained in the linked *DataObjects* are going to be exported in CSV files (**<type>**).

- *Plot*:
 - named “historiesPlot” connected with the *DataObjects* **Entity** “samples”. This plot will draw the final state of the variables A, B, C, D with respect to the input variables $\sigma(s)$ and $\text{decay}(s)$.
 - named “samplesPlot3D” connected with the *DataObjects* **Entity** “histories”. This plot will draw the evolution of the variables A, B, C, D ;

Note that both plots are of type *SubPlot*. Four plots are going to be placed in each of the figures.

8. Steps:

```
<Steps>
  <MultiRun name="sample">
    <Input      class="Files"
      type="input">referenceInput.xml</Input>
    <Model      class="Models"
      type="Code">testModel</Model>
    <Sampler    class="Samplers"
      type="MonteCarlo">monteCarlo</Sampler>
    <Output     class="DataObjects"
      type="PointSet">samples</Output>
    <Output     class="DataObjects"
      type="HistorySet">histories</Output>
  </MultiRun>
  <IOStep name="writeHistories" pauseAtEnd="True">
    <Input class="DataObjects"
      type="HistorySet">histories</Input>
    <Input class="DataObjects"
      type="PointSet">samples</Input>
    <Output      class="OutStreams"
      type="Plot">samplesPlot3D</Output>
    <Output      class="OutStreams"
      type="Plot">historyPlot</Output>
    <Output      class="OutStreams"
      type="Print">samples</Output>
```

```

<Output class="OutStreams"
  type="Print">histories</Output>
</IOStep>
</Steps>

```

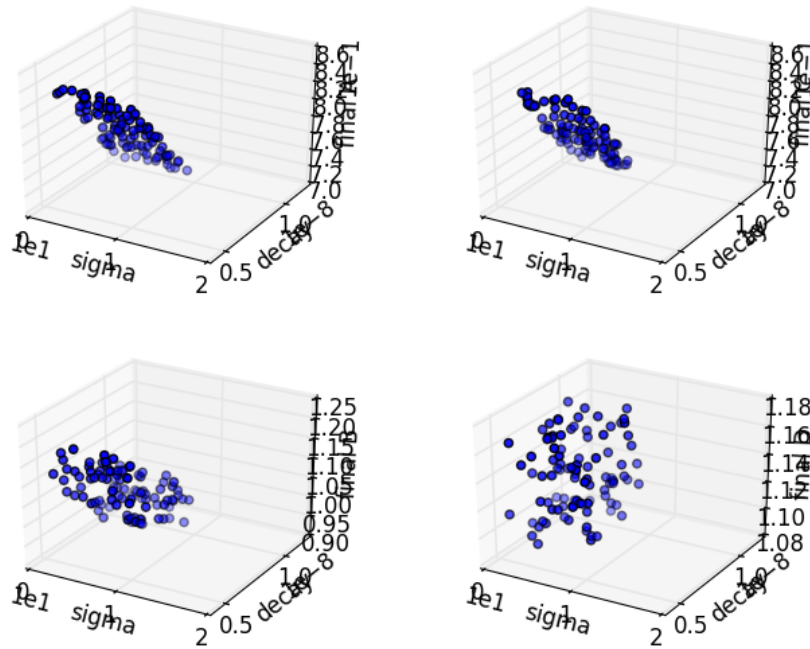


Figure 10. Plot of the samples generated by the MC sampling for variables A, B, C, D .

Finally, all the previously defined **Entities** can be combined in the **<Steps>** block. As inferable, two **<Steps>** have been inputted:

- **<MultiRun>** named “sample”, used to run the multiple instances of the driven code and collect the outputs in the two *DataObjects*. As it can be seen, the **<Sampler>** is inputted to communicate to the *Step* that the driven code needs to be perturbed through the Monte-Carlo sampling;
- **<IOStep>** named “writeHistories”, used to 1) dump the “histories” and “samples” *DataObjects* **Entity** in a CSV file and 2) plot the data in the PNG file and on the screen.

Figures 9 and 10 report the evolution of the variables A, B, C, D and their final values, respectively.

7.2 Grid

The Grid sampling method (also known as Full Factorial Design of Experiment) represents one of the simplest methodologies that can be employed in order to explore the interaction of multiple random variables with respect selected FOMs. In this section, a brief theoretical background is reported. In addition, it is shown how to employ this methodology with RAVEN.

7.2.1 Grid theory introduction

The goal of the Grid-based sampling strategy is to explore the interaction of multiple random variables (i.e., uncertainties) with respect to selected FOMs. Indeed, this method is mainly used to perform parametric analysis of the system response rather than a probabilistic one. This method discretizes the domain of the uncertainties in a user-defined number of intervals (see Figure 11) and record the response of the model (e.g. a system code) at each coordinate (i.e., combination of the uncertainties) of the grid.

This method starts from the assumption that each coordinate on the grid is representative, with respect to the FOMs of interest, of the surrounding grid cell. In other words, it is assumed that the response of a system does not significantly change within the hyper-volume surrounding each grid coordinate (red square in Figure 11).

Similarly to what has been already reported for the Monte-Carlo sampling, consider a random variable X having PDF $pdf_X(x)$ and, consequentially, CDF $cdf_X(x)$ in the domain χ . Then the expected value of a function f of X is as follows:

$$\begin{aligned}\mathbb{E}(f(X)) &= \sum_{x \in \chi} f(x) pdf_X(x) & \text{if } X \text{ discrete} \\ \mathbb{E}(f(X)) &= \int_{x \in \chi} f(x) pdf_X(x) & \text{if } X \text{ continuous}\end{aligned}\tag{43}$$

In the Grid approach, the domain of X is discretized in a finite number of intervals. Recall that each node of this discretization is representative of the surrounding hyper-volume. This means that a weight needs to be associated with each coordinate of the resulting grid:

$$w_i = cdf_X(x_{i+1/2}) - cdf_X(x_{i-1/2})\tag{44}$$

Now consider a n -discretization of the domain of X , (x_1, \dots, x_n) and compute the mean of $f(x)$ over the discretization. Based on the previous equation, the computation of the expected value of $f(x)$ is as follows:

$$\mathbb{E}(f(X)) \approx \tilde{f}_n(x) = \frac{1}{\sum_{i=1}^n w_i} \sum_{i=1}^n f(x_i) \times w_i\tag{45}$$

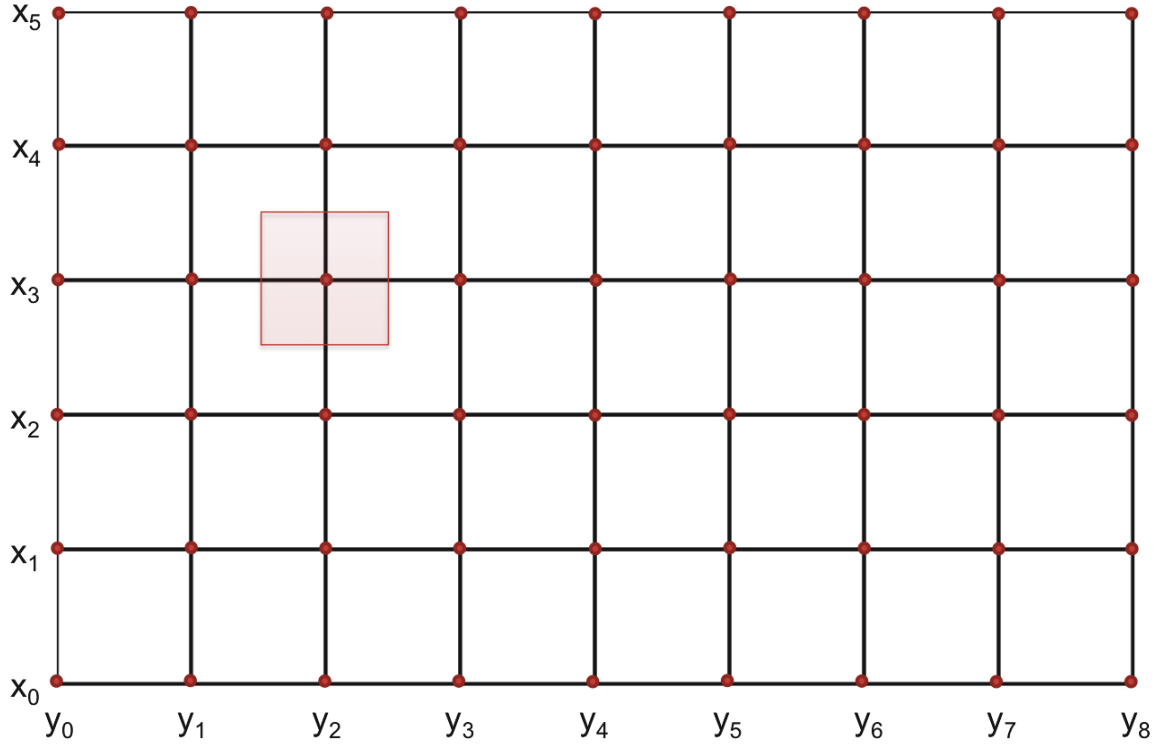


Figure 11. Example of 2-Dimensional grid discretization.

If the number of uncertainties under consideration is greater than one (m), the above equation becomes:

$$\mathbb{E}(f(\bar{X})) \approx \tilde{f}_n(\bar{x}) = \frac{1}{\sum_{i=1}^n \prod_{j=1}^m w_{i,j}} \sum_{i=1}^n f(\bar{x}_i) \times \prod_{j=1}^m w_{i,j} \quad (46)$$

7.2.2 Grid sampling through RAVEN

The goal of this section is to show how to:

1. Set up a simple Grid sampling for performing a parametric analysis of a driven code
2. Load the outputs of the code into the RAVEN DataObjects system
3. Print out what contained in the DataObjects
4. Generate basic plots of the code result.

In order to accomplish these tasks, the following RAVEN **Entities** (XML blocks in the input files) are required:

1. *RunInfo*:

```
<RunInfo>
  <JobName>ChapterVII-I/Grid</JobName>
  <Sequence>sample,writeHistories</Sequence>
  <WorkingDir>ChapterVII-I/Grid</WorkingDir>
  <batchSize>12</batchSize>
</RunInfo>
```

As shown in Section 3.2, the *RunInfo* **Entity** is intended to set up the desired analysis. In this specific case, two steps (**<Sequence>**) are sequentially run using 12 processors (**<batchSize>**). This means that 12 instances of the driven code are run simultaneously. Every time a simulation ends, a new one is launched.

2. *Files*:

```
<Files>
  <Input name="referenceInput.xml"
    type="input">referenceInput.xml</Input>
</Files>
```

Since the driven code uses a single input file, in this section the original input is placed. As described in the user manual [13] the attribute **name** represents the alias that is used in all the other input blocks in order to refer to this file.

3. *Models*:

```
<Models>
  <Code name="testModel" subType="GenericCode">
    <executable>
      ../physicalCode/analyticalbateman/AnalyticalDplMain.py
    </executable>
    <clargs arg="python" type="prepend"/>
    <clargs arg="" extension=".xml" type="input"/>
    <clargs arg="" extension=".csv" type="output"/>
    <prepend>python</prepend>
  </Code>
</Models>
```

The Model here is represented by the **AnalyticalBateman**, which already dumps its output file in a CSV format (standard format that RAVEN can read). For this reason, the *Generic-Code* interface is used.

4. Distributions:

```
<Distributions>
  <Uniform name="sigma">
    <lowerBound>1</lowerBound>
    <upperBound>10</upperBound>
  </Uniform>
  <Uniform name="decayConstant">
    <lowerBound>0.000000005</lowerBound>
    <upperBound>0.000000010</upperBound>
  </Uniform>
</Distributions>
```

In the Distributions XML section, the stochastic model for the uncertainties treated by the Grid sampling are reported. In this case two distributions are defined:

- $\sigma \sim \mathbb{U}(1, 10)$, used to model the uncertainties associated with the Model $\sigma(s)$
- $\text{decayConstant} \sim \mathbb{U}(0.5e-8, 1e-8)$, used to model the uncertainties associated with the Model decay constants .

5. Samplers:

```
<Samplers>
  <Grid name="grid">
    <variable name="sigma-A">
      <distribution>sigma</distribution>
      <grid construction="equal" steps="1" type="value">2
        4.0</grid>
    </variable>
    <variable name="decay-A">
      <distribution>decayConstant</distribution>
      <grid construction="custom" type="value">0.000000005
        0.000000008</grid>
    </variable>
    <variable name="sigma-B">
      <distribution>sigma</distribution>
      <grid construction="equal" steps="1"
        type="CDF">0.1 0.8</grid>
    </variable>
    <variable name="decay-B">
      <distribution>decayConstant</distribution>
      <grid construction="equal" steps="1"
        type="CDF">0.1 0.8</grid>
    </variable>
  </Grid>
```

```

<variable name="sigma-C">
  <distribution>sigma</distribution>
  <grid construction="equal" steps="1"
    type="value">4 5</grid>
</variable>
<variable name="decay-C">
  <distribution>decayConstant</distribution>
  <grid construction="equal" steps="1"
    type="CDF">0.1 0.5</grid>
</variable>
<variable name="sigma-D">
  <distribution>sigma</distribution>
  <grid construction="equal" steps="1"
    type="CDF">0.4 0.8</grid>
</variable>
<variable name="decay-D">
  <distribution>decayConstant</distribution>
  <grid construction="equal" steps="1"
    type="CDF">0.1 0.8</grid>
</variable>
</Grid>
</Samplers>

```

To employ the Grid sampling strategy, a **<Grid>** node needs to be specified. As shown above, in each variable section, the **<grid>** is defined. The number of samples finally requested are equal to $n_{samples} = \prod_{i=1}^n n_{steps_i+1} = 256$. Note that, for each variable, can be defined either in probability (CDF) or in absolute value.

6. DataObjects:

```

<DataObjects>
  <PointSet name="samples">
    <Input>
      sigma-A, sigma-B, sigma-C, sigma-D,
      decay-A, decay-B, decay-C, decay-D
    </Input>
    <Output>A, B, C, D, time</Output>
  </PointSet>
  <HistorySet name="histories">
    <Input>
      sigma-A, sigma-B, sigma-C, sigma-D,
      decay-A, decay-B, decay-C, decay-D
    </Input>
    <Output>A, B, C, D, time</Output>
  </HistorySet>
</DataObjects>

```

```
</HistorySet>
</DataObjects>
```

In this block, two *DataObjects* are defined: 1) PointSet named “samples”, and 2) HistorySet named “histories”. In the **<Input>** node all the variables perturbed through the Grid strategy are listed. In this way, any realization in the input space is linked to the outputs listed in the **<Output>** node.

7. OutStreams:

```
<OutStreams>
  <Print name="samples">
    <type>csv</type>
    <source>samples</source>
  </Print>
  <Print name="histories">
    <type>csv</type>
    <source>histories</source>
  </Print>
  <Plot dim="2" name="historiesPlot" overwrite="false"
    verbosity="debug">
    <plotSettings>
      <gridSpace>2 2</gridSpace>
      <plot>
        <type>line</type>
        <x>histories|Output|time</x>
        <y>histories|Output|A</y>
        <color>blue</color>
        <gridLocation>
          <x>0</x>
          <y>0</y>
        </gridLocation>
        <xlabel>time (s)</xlabel>
        <ylabel>evolution A(kg)</ylabel>
      </plot>
      <plot>
        <type>line</type>
        <x>histories|Output|time</x>
        <y>histories|Output|B</y>
        <color>red</color>
        <gridLocation>
          <x>1</x>
          <y>0</y>
        </gridLocation>
      </plot>
    </plotSettings>
  </Plot>
```

```

        <xlabel>time (s)</xlabel>
        <ylabel>evolution B(kg)</ylabel>
    </plot>
    <plot>
        <type>line</type>
        <x>histories|Output|time</x>
        <y>histories|Output|C</y>
        <color>yellow</color>
        <gridLocation>
            <x>0</x>
            <y>1</y>
        </gridLocation>
        <xlabel>time (s)</xlabel>
        <ylabel>evolution C(kg)</ylabel>
    </plot>
    <plot>
        <type>line</type>
        <x>histories|Output|time</x>
        <y>histories|Output|D</y>
        <color>black</color>
        <gridLocation>
            <x>1</x>
            <y>1</y>
        </gridLocation>
        <xlabel>time (s)</xlabel>
        <ylabel>evolution D(kg)</ylabel>
    </plot>
</plotSettings>
<actions>
    <how>png,screen</how>
    <title>
        <text> </text>
    </title>
</actions>
</Plot>
<Plot dim="3" name="samplesPlot3D" overwrite="false"
    verbosity="debug">
    <plotSettings>
        <gridSpace>2 2</gridSpace>
    <plot>
        <type>scatter</type>
        <x>samples|Input|sigma-A</x>

```

```

<y>samples|Input|decay-A</y>
<z>samples|Output|A</z>
<color>blue</color>
<gridLocation>
  <x>0</x>
  <y>0</y>
</gridLocation>
<xlabel>sigma</xlabel>
<ylabel>decay</ylabel>
<zlabel>final A</zlabel>
</plot>
<plot>
  <type>scatter</type>
  <x>samples|Input|sigma-B</x>
  <y>samples|Input|decay-B</y>
  <z>samples|Output|B</z>
  <color>red</color>
  <gridLocation>
    <x>1</x>
    <y>0</y>
  </gridLocation>
  <xlabel>sigma</xlabel>
  <ylabel>decay</ylabel>
  <zlabel>final B</zlabel>
</plot>
<plot>
  <type>scatter</type>
  <x>samples|Input|sigma-C</x>
  <y>samples|Input|decay-C</y>
  <z>samples|Output|C</z>
  <color>yellow</color>
  <gridLocation>
    <x>0</x>
    <y>1</y>
  </gridLocation>
  <xlabel>sigma</xlabel>
  <ylabel>decay</ylabel>
  <zlabel>final C</zlabel>
</plot>
<plot>
  <type>scatter</type>
  <x>samples|Input|sigma-D</x>

```



```

        <y>samples|Input|decay-D</y>
        <z>samples|Output|D</z>
        <color>black</color>
        <gridLocation>
            <x>1</x>
            <y>1</y>
        </gridLocation>
        <xlabel>sigma</xlabel>
        <ylabel>decay</ylabel>
        <zlabel>final D</zlabel>
    </plot>
    <xlabel>sigma</xlabel>
    <ylabel>decay</ylabel>
    <zlabel>final response</zlabel>
</plotSettings>
<actions>
    <how>png,screen</how>
    <title>
        <text> </text>
    </title>
</actions>
</Plot>
</OutStreams>

```

In this block, both the Out-Stream types are constructed:

- *Print*:
 - named “samples” connected with the *DataObjects* **Entity** “samples” (<source>)
 - named “histories” connected with the *DataObjects* **Entity** “histories” (<source>).

When these objects get used, all the information contained in the linked *DataObjects* are going to be exported in CSV files (<type>).

- *Plot*:
 - named “historiesPlot” connected with the *DataObjects* **Entity** “samples”. This plot will draw the final state of the variables A, B, C, D with respect to the input variables $\sigma(s)$ and $\text{decay}(s)$.
 - named “samplesPlot3D” connected with the *DataObjects* **Entity** “histories”. This plot will draw the evolution of the variables A, B, C, D .

As it can be noticed, both plots are of type *SubPlot*. Four plots are placed in each of the figures.

8. Steps:

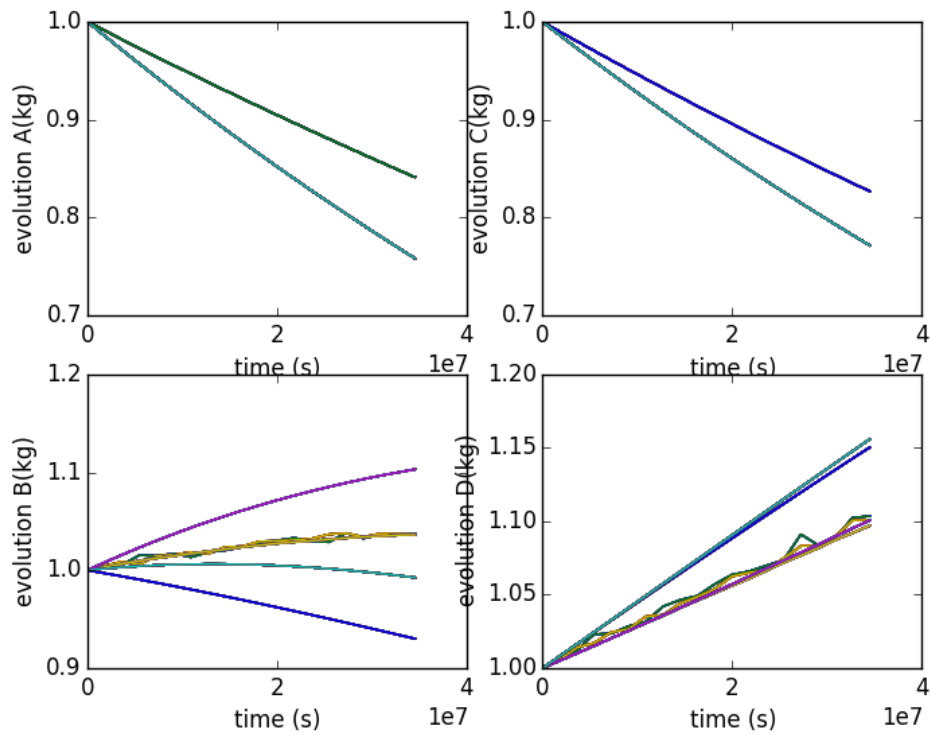


Figure 12. Plot of the histories generated by the Grid sampling for variables A , B , C , D .

```

<Steps>
  <MultiRun name="sample">
    <Input      class="Files"
      type="input">referenceInput.xml</Input>
    <Model      class="Models"
      type="Code">testModel</Model>
    <Sampler    class="Samplers"
      type="Grid">grid</Sampler>
    <Output     class="DataObjects"
      type="PointSet">samples</Output>
    <Output     class="DataObjects"
      type="HistorySet">histories</Output>
  </MultiRun>
  <IOStep name="writeHistories" pauseAtEnd="True">
    <Input class="DataObjects"

```

```

    type="HistorySet">histories</Input>
<Input class="DataObjects"
  type="PointSet">samples</Input>
<Output class="OutStreams"
  type="Plot">samplesPlot3D</Output>
<Output class="OutStreams"
  type="Plot">historyPlot</Output>
<Output class="OutStreams"
  type="Print">samples</Output>
<Output class="OutStreams"
  type="Print">histories</Output>
</IOStep>
</Steps>

```

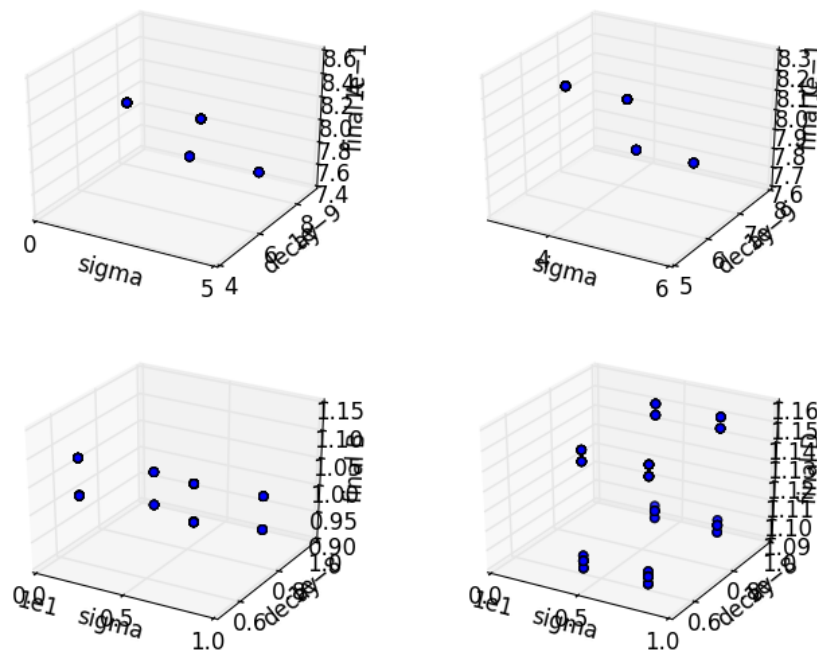


Figure 13. Plot of the samples generated by the Grid sampling for variables A, B, C, D .

Finally, all the previously defined **Entities** can be combined in the **<Steps>** block. As inferable, two **<Steps>** have been inputted:

- **<MultiRun>** named “sample”, is used to run the multiple instances of the code and collect the outputs in the two *DataObjects*. As it can be seen, the **<Sampler>** is inputted to communicate to the *Step* that the driven code needs to be perturbed through the Grid sampling
- **<IOStep>** named “writeHistories”, used to 1) dump the “histories” and “samples” *DataObjects* **Entity** in a CSV file and 2) plot the data in the PNG file and on the screen.

Figures 12 and 13 report the evolution of the variables A, B, C, D and their final values, respectively.

7.3 Stratified

The Stratified sampling is a class of methods that relies on the assumption that the input space (i.e., uncertainties) can be separated in regions (strata) based on similarity of the response of the system for input set within the same strata. Following this assumption, the most rewarding (in terms of computational cost vs. knowledge gain) sampling strategy would be to place one sample for each region. In this way, the same information is not collected more than once and all the prototypical behavior are sampled at least once. In Figure 14, the Stratified sampling approach is exemplified.

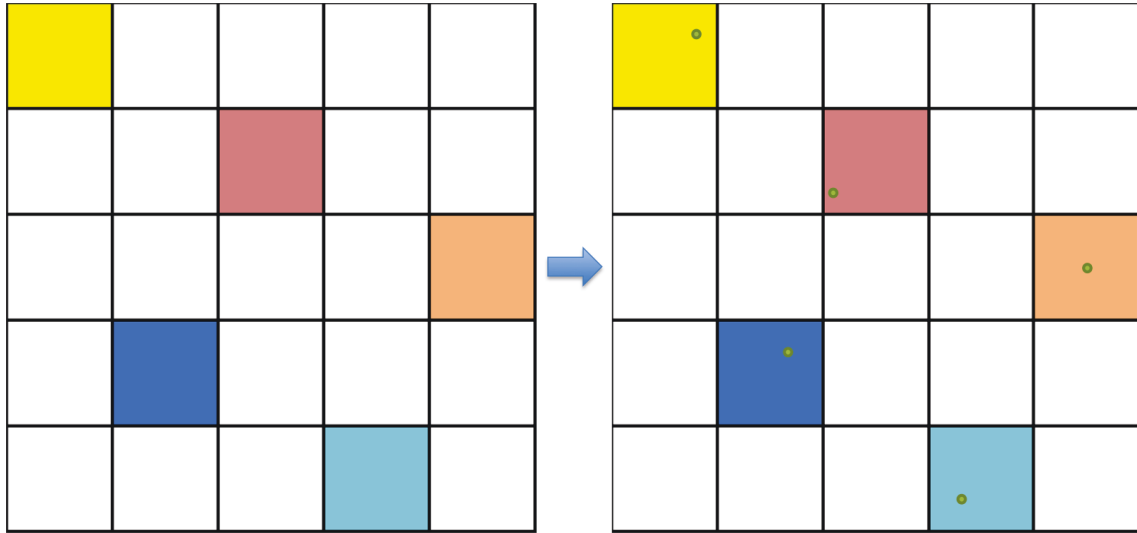


Figure 14. Example of Stratified sampling approach.

In this section, a brief theoretical background is reported. In addition, it is shown how to employ this methodology with RAVEN.

7.3.1 Stratified theory introduction

The Stratified sampling approach is a method for the exploration of the input space that consists of 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. The Latin Hyper-Cube sampling represents a specialized version of the stratified approach, when the domain strata are constructed in equally-probable CDF bins.

Similarly to what has been already reported for the Grid sampling, consider a set of m random variables X_j , $j = 1, \dots, m$ having PDFs $pdf_{X_j}(x_j)$ and, consequentially, CDF $cdf_{X_j}(x_j)$ in the domain χ_j . Then the expected value of a function f of X_j , $j = 1, \dots, m$ is as follows:

$$\begin{aligned}\mathbb{E}(f(\bar{X})) &= \sum f(x) \prod_{j=1}^m pdf_{X_j}(x_j) \quad \text{if } \bar{X} \text{ discrete} \\ \mathbb{E}(f(\bar{X})) &= \int f(x) \prod_{j=1}^m pdf_{X_j}(x_j) \quad \text{if } \bar{X} \text{ continuous}\end{aligned}\tag{47}$$

In the Stratified approach, the domain of \bar{X} is discretized in a set of strata. Consequentially, similarly to the Grid sampling, a weight needs to be associated with each realization in the input space:

$$w_i = \frac{\prod_{j=1}^m [cdf_{X_j}(x_{i,j+1}) - cdf_{X_j}(x_{i,j})]}{\sum_{points} \prod_{j=1}^m [cdf_{X_j}(x_{i,j+1}) - cdf_{X_j}(x_{i,j})]}\tag{48}$$

Each realization carries a weight representative of each stratum.

Now consider an n -strata of the domain of \bar{X} , and compute the expected value of $f(x)$ over the discretization. Based on the previous equation, the computation of the expected value of $f(x)$ is as follows:

$$\mathbb{E}(f(\bar{X})) \approx \tilde{f}_n(\bar{x}) = \frac{1}{\sum_{i=1}^n w_i} \sum_{i=1}^n f(\bar{x}_i) \times w_i\tag{49}$$

7.3.2 Stratified sampling through RAVEN

The goal of this section is to show how to:

1. Set up a simple Stratified sampling in order to perform a parametric analysis on a driven code

2. Load the outputs of the code into the RAVEN DataObjects system
3. Print out what contained in the DataObjects
4. Generate basic plots of the code result.

To accomplish these tasks, the following RAVEN **Entities** (XML blocks in the input files) are defined:

1. *RunInfo*:

```
<RunInfo>
  <JobName>ChapterVII-I/Stratified</JobName>
  <Sequence>sample,writeHistories</Sequence>
  <WorkingDir>ChapterVII-I/Stratified</WorkingDir>
  <batchSize>12</batchSize>
</RunInfo>
```

As reported in Section 3.2, the *RunInfo Entity* is intended to set up the analysis that the user wants to perform. In this specific case, two steps (<Sequence>) are run using 12 processors (<batchSize>). This means that 12 instances of the driven code are run simultaneously. Every time a simulation ends, a new one is launched.

2. *Files*:

```
<Files>
  <Input name="referenceInput.xml"
    type="input">referenceInput.xml</Input>
</Files>
```

Since the considered code uses a single input file, in this section the original input is placed. The attribute **name** represents the alias that is going to be used in all the other input blocks in order to refer to this file.

3. *Models*:

```
<Models>
  <Code name="testModel" subType="GenericCode">
    <executable>
      ../physicalCode/analyticalbateman/AnalyticalDplMain.py
    </executable>
    <clargs arg="python" type="prepend"/>
    <clargs arg="" extension=".xml" type="input"/>
    <clargs arg="" extension=".csv" type="output"/>
    <prepend>python</prepend>
  </Code>
</Models>
```

```
</Code>
<Models>
```

The Model here is represented by the **AnalyticalBateman**, which already dumps its output file in a CSV format (standard format that RAVEN can read). For this reason, the *Generic-Code* interface is used.

4. *Distributions*:

```
<Distributions>
  <Uniform name="sigma">
    <lowerBound>1</lowerBound>
    <upperBound>10</upperBound>
  </Uniform>
  <Uniform name="decayConstant">
    <lowerBound>0.000000005</lowerBound>
    <upperBound>0.000000010</upperBound>
  </Uniform>
</Distributions>
```

In the Distributions XML section, the stochastic model for the uncertainties treated by the Stratified sampling are reported. In this case two distributions are defined:

- $\sigma \sim \mathbb{U}(1, 10)$, used to model the uncertainties associated with the Model $\sigma(s)$
- $\text{decayConstant} \sim \mathbb{U}(0.5e-8, 1e-8)$, used to model the uncertainties associated with the Model *decay constants*.

5. *Samplers*:

```
<Stratified name="stratified">
  <variable name="sigma-A">
    <distribution>sigma</distribution>
    <grid construction="equal" steps="100"
      type="value">2 4.0</grid>
  </variable>
  <variable name="decay-A">
    <distribution>decayConstant</distribution>
    <grid construction="equal" steps="100"
      type="value">0.000000005 0.000000008</grid>
  </variable>
  <variable name="sigma-B">
    <distribution>sigma</distribution>
    <grid construction="equal" steps="100"
      type="CDF">0.1 0.8</grid>
  </variable>
</Stratified>
```

```

</variable>
<variable name="decay-B">
  <distribution>decayConstant</distribution>
  <grid construction="equal" steps="100"
    type="CDF">0.1 0.8</grid>
</variable>
<variable name="sigma-C">
  <distribution>sigma</distribution>
  <grid construction="equal" steps="100"
    type="value">1.0 5</grid>
</variable>
<variable name="decay-C">
  <distribution>decayConstant</distribution>
  <grid construction="equal" steps="100"
    type="CDF">0.1 0.5</grid>
</variable>
<variable name="sigma-D">
  <distribution>sigma</distribution>
  <grid construction="equal" steps="100"
    type="CDF">0.4 0.8</grid>
</variable>
<variable name="decay-D">
  <distribution>decayConstant</distribution>
  <grid construction="equal" steps="100"
    type="CDF">0.1 0.8</grid>
</variable>
</Stratified>

```

To employ the Stratified sampling strategy, a **<Stratified>** node needs to be specified. In each variable section, the **<grid>** is defined. It is important to mention that the number of **steps** needs to be the same for each of the variables, since, as reported in previous section, the Stratified sampling strategy it discretizes the domain in strata. The number of samples finally requested is equal to $n_{samples} = n_{steps} = 100$. If the grid for each variables is defined in CDF and of **type** = "equal", the Stratified sampling corresponds to the well-known Latin Hyper Cube sampling.

6. *DataObjects:*

```

<DataObjects>
  <PointSet name="samples">
    <Input>
      sigma-A, sigma-B, sigma-C, sigma-D,
      decay-A, decay-B, decay-C, decay-D
    </Input>
  </PointSet>
</DataObjects>

```



```

    <Output>A,B,C,D,time</Output>
  </PointSet>
  <HistorySet name="histories">
    <Input>
      sigma-A,sigma-B,sigma-C,sigma-D,
      decay-A,decay-B,decay-C,decay-D
    </Input>
    <Output>A,B,C,D,time</Output>
  </HistorySet>
</DataObjects>

```

In this block, two *DataObjects* are defined: 1) PointSet named “samples”, 2) HistorySet named “histories”. In the **<Input>** node all the variables perturbed through the Stratified strategy are listed. In this way, any realization in the input space is linked to the outputs listed in the **<Output>** node.

7. OutStreams:

```

<OutStreams>
  <Print name="samples">
    <type>csv</type>
    <source>samples</source>
  </Print>
  <Print name="histories">
    <type>csv</type>
    <source>histories</source>
  </Print>
  <Plot dim="2" name="historiesPlot" overwrite="false"
    verbosity="debug">
    <plotSettings>
      <gridSpace>2 2</gridSpace>
      <plot>
        <type>line</type>
        <x>histories|Output|time</x>
        <y>histories|Output|A</y>
        <color>blue</color>
        <gridLocation>
          <x>0</x>
          <y>0</y>
        </gridLocation>
        <xlabel>time (s)</xlabel>
        <ylabel>evolution A(kg)</ylabel>
      </plot>
    </plot>
  </Plot>
</OutStreams>

```

```

    <type>line</type>
    <x>histories|Output|time</x>
    <y>histories|Output|B</y>
    <color>red</color>
    <gridLocation>
        <x>1</x>
        <y>0</y>
    </gridLocation>
    <xlabel>time (s)</xlabel>
    <ylabel>evolution B(kg)</ylabel>
</plot>
<plot>
    <type>line</type>
    <x>histories|Output|time</x>
    <y>histories|Output|C</y>
    <color>yellow</color>
    <gridLocation>
        <x>0</x>
        <y>1</y>
    </gridLocation>
    <xlabel>time (s)</xlabel>
    <ylabel>evolution C(kg)</ylabel>
</plot>
<plot>
    <type>line</type>
    <x>histories|Output|time</x>
    <y>histories|Output|D</y>
    <color>black</color>
    <gridLocation>
        <x>1</x>
        <y>1</y>
    </gridLocation>
    <xlabel>time (s)</xlabel>
    <ylabel>evolution D(kg)</ylabel>
</plot>
</plotSettings>
<actions>
    <how>png,screen</how>
    <title>
        <text> </text>
    </title>
</actions>

```

```

</Plot>
<Plot dim="3" name="samplesPlot3D" overwrite="false"
  verbosity="debug">
  <plotSettings>
    <gridSpace>2 2</gridSpace>
    <plot>
      <type>scatter</type>
      <x>samples|Input|sigma-A</x>
      <y>samples|Input|decay-A</y>
      <z>samples|Output|A</z>
      <color>blue</color>
      <gridLocation>
        <x>0</x>
        <y>0</y>
      </gridLocation>
      <xlabel>sigma</xlabel>
      <ylabel>decay</ylabel>
      <zlabel>final A</zlabel>
    </plot>
    <plot>
      <type>scatter</type>
      <x>samples|Input|sigma-B</x>
      <y>samples|Input|decay-B</y>
      <z>samples|Output|B</z>
      <color>red</color>
      <gridLocation>
        <x>1</x>
        <y>0</y>
      </gridLocation>
      <xlabel>sigma</xlabel>
      <ylabel>decay</ylabel>
      <zlabel>final B</zlabel>
    </plot>
    <plot>
      <type>scatter</type>
      <type>scatter</type>
      <x>samples|Input|sigma-C</x>
      <y>samples|Input|decay-C</y>
      <z>samples|Output|C</z>
      <color>yellow</color>
      <gridLocation>
        <x>0</x>

```

```

        <y>1</y>
    </gridLocation>
    <xlabel>sigma</xlabel>
    <ylabel>decay</ylabel>
    <zlabel>final C</zlabel>
</plot>
<plot>
    <type>scatter</type>
    <x>samples|Input|sigma-D</x>
    <y>samples|Input|decay-D</y>
    <z>samples|Output|D</z>
    <color>black</color>
    <gridLocation>
        <x>1</x>
        <y>1</y>
    </gridLocation>
    <xlabel>sigma</xlabel>
    <ylabel>decay</ylabel>
    <zlabel>final D</zlabel>
</plot>
<xlabel>sigma</xlabel>
<ylabel>decay</ylabel>
<zlabel>final response</zlabel>
</plotSettings>
<actions>
    <how>png,screen</how>
    <title>
        <text> </text>
    </title>
</actions>
</Plot>
</OutStreams>

```

In this block, both the Out-Stream types are constructed:

- *Print*:
 - named “samples” connected with the *DataObjects* **Entity** “samples” (**<source>**)
 - named “histories” connected with the *DataObjects* **Entity** “histories” (**<source>**).

When these objects get used, all the information contained in the linked *DataObjects* are going to be exported in CSV files (**<type>**).

- *Plot*:

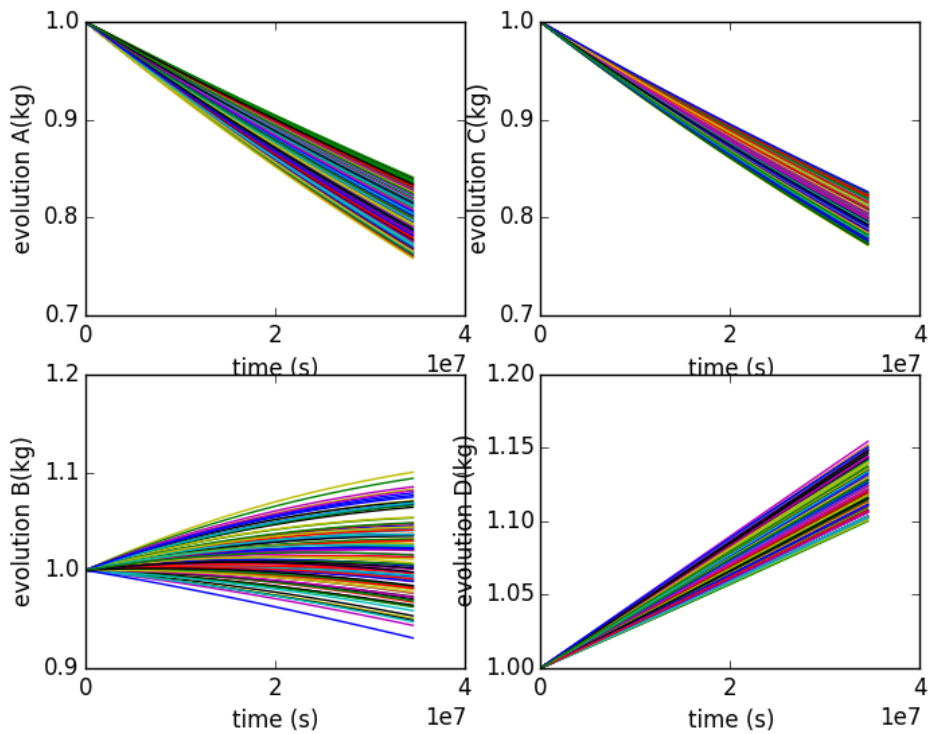


Figure 15. Plot of the histories generated by the Stratified sampling for variables A, B, C, D .

- named “historiesPlot” connected with the *DataObjects* **Entity** “samples”. This plot will draw the final state of the variables A, B, C, D with respect to the input variables $\sigma(s)$ and $\delta(s)$
- named “samplesPlot3D” connected with the *DataObjects* **Entity** “histories”. This plot will draw the evolution of the variables A, B, C, D .

As it can be noticed, both plots are of type *SubPlot*. Four plots are going to be placed in each of the figures.

8. Steps:

```
<Steps>
  <MultiRun name="sample">
    <Input          class="Files"
      type="input">referenceInput.xml</Input>
    <Model          class="Models"
      type="Code">testModel</Model>
```

```

<Sampler class="Samplers"
  type="Stratified">stratified</Sampler>
<Output class="DataObjects"
  type="PointSet">samples</Output>
<Output class="DataObjects"
  type="HistorySet">histories</Output>
</MultiRun>
<IOStep name="writeHistories" pauseAtEnd="True">
  <Input class="DataObjects"
    type="HistorySet">histories</Input>
  <Input class="DataObjects"
    type="PointSet">samples</Input>
  <Output class="OutStreams"
    type="Plot">samplesPlot3D</Output>
  <Output class="OutStreams"
    type="Plot">historyPlot</Output>
  <Output class="OutStreams"
    type="Print">samples</Output>
  <Output class="OutStreams"
    type="Print">histories</Output>
</IOStep>
</Steps>

```

Finally, all the previously defined **Entities** can be combined in the **<Steps>** block. As inferable, two **<Steps>** have been inputted:

- **<MultiRun>** named “sample”, used to run the multiple instances of the driven code and collect the outputs in the two *DataObjects*. As it can be seen, the **<Sampler>** is inputted to communicate to the *Step* that the driven code needs to be perturbed through the Stratified sampling.
- **<IOStep>** named “writeHistories”, used to 1) dump the “histories” and “samples” *DataObjects* **Entity** in a CSV file and 2) plot the data in the PNG file and on the screen.

As previously mentioned, Figures 15 and 16 report the evolution of the variables A, B, C, D and their final values, respectively.

7.4 Sparse Grid Collocation

The Sparse Grid Collocation sampler represents an advanced methodology to perform Uncertainty Quantification. They aim to explore the input space leveraging the information contained in the

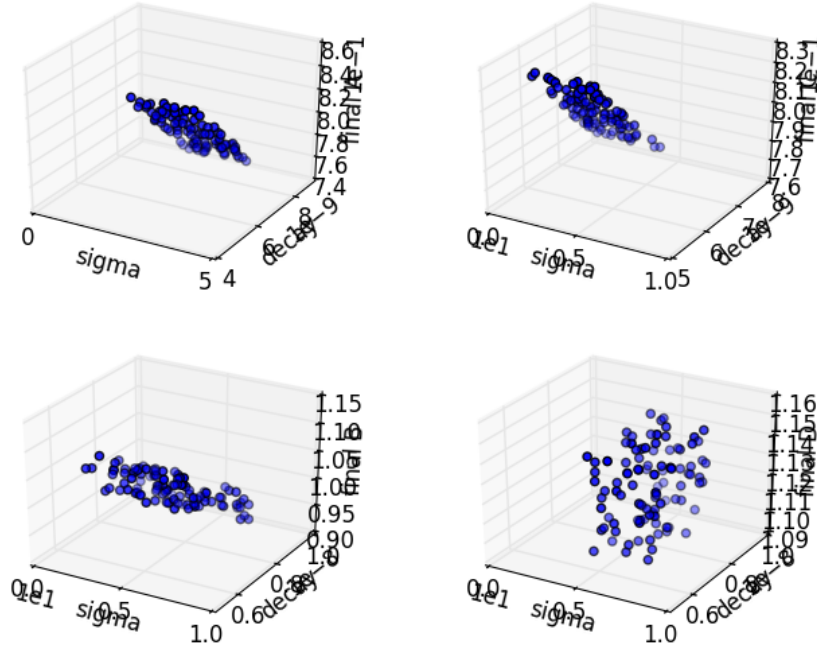


Figure 16. Plot of the samples generated by the Stratified sampling for variables A, B, C, D .

associated probability density functions. It builds on generic Grid sampling by selecting evaluation points based on characteristic quadratures as part of stochastic collocation for generalized polynomial chaos uncertainty quantification. In collocation an N-D grid is constructed, with each uncertain variable providing an axis. Along each axis, the points of evaluation correspond to quadrature points necessary to integrate polynomials. In the simplest (and most naive) case, a N-D tensor product of all possible combinations of points from each dimension's quadrature is constructed as sampling points. The number of necessary samples can be reduced by employing Smolyak-like sparse grid algorithms, which use reduced combinations of polynomial orders to reduce the necessary sampling space.

In this Section, a brief theoretical background is reported. In addition, it is shown how to employ this methodology with RAVEN.

7.4.1 Sparse Grid Collocation Theory Introduction

7.4.1.1 Generalized Polynomial Chaos

In general, polynomial chaos expansion (PCE) methods seek to interpolate the simulation code as a combination of polynomials of varying degree in each dimension of the input space. Originally Wiener proposed expanding in Hermite polynomials for Gaussian-normal distributed variables [15]. Askey and Wilson generalized Hermite polynomials to include Jacobi polynomials, including Legendre and Laguerre polynomials [16]. Xiu and Karniadakis combines these concepts to perform PCE for a range of Gaussian-based distributions with corresponding polynomials, including Legendre polynomials for uniform distributions, Laguerre polynomials for Gamma distributions, and Jacobi polynomials for Beta distributions [17].

In each of these cases, a probability-weighted integral over the distribution can be cast in a way that the corresponding polynomials are orthogonal over the same weight and interval. These chaos Wiener-Askey polynomials were used by Xiu and Karniadakis to develop the generalized polynomial chaos expansion method (gPC), including a transformation for applying the same method to arbitrary distributions (as long as they have a known inverse CDF) [17]. Two significant methodologies have grown from gPC application. The first makes use of Lagrange polynomials to expand the original function or simulation code, as they can be made orthogonal over the same domain as the distributions [18]; the other uses the Wiener-Askey polynomials [17].

Consider a simulation code that produces a quantity of interest u as a function $u(Y)$ whose arguments are the uncertain, distributed input parameters $Y = (Y_1, \dots, Y_n, \dots, Y_N)$. A particular realization ω of Y_n is expressed by $Y_n(\omega)$, and a single realization of the entire input space results in a solution to the function as $u(Y(\omega))$. Obtaining a realization of $u(Y)$ may take considerable computation time and effort. $u(Y)$ gets expanded in orthonormal multidimensional polynomials $\Phi_k(Y)$, where k is a multi-index tracking the polynomial order in each axis of the polynomial Hilbert space, and $\Phi_k(Y)$ is constructed as

$$\Phi_k(Y) = \prod_{n=1}^N \phi_{k_n}(Y_n), \quad (50)$$

where $\phi_{k_n}(Y_n)$ is a single-dimension Wiener-Askey orthonormal polynomial of order k_n and $k = (k_1, \dots, k_n, \dots, k_N)$, $k_n \in \mathbb{N}^0$. For example, given $u(y_1, y_2, y_3)$, $k = (2, 1, 4)$ is the multi-index of the product of a second-order polynomial in y_1 , a first-order polynomial in y_2 , and a fourth-order polynomial in y_4 . The gPC for $u(Y)$ using this notation is

$$u(Y) \approx \sum_{k \in \Lambda(L)} u_k \Phi_k(Y), \quad (51)$$

where u_k is a scalar weighting polynomial coefficient. The polynomials used in the expansion are determined by the set of multi-indices $\Lambda(L)$, where L is a truncation order. In the limit that

Λ contains all possible combinations of polynomials of any order, Eq. 50 is exact. Practically, however, Λ is truncated to some finite set of combinations, discussed in section 7.4.1.2.

Using the orthonormal properties of the Wiener-Askey polynomials,

$$\int_{\Omega} \Phi_k(Y) \Phi_{\hat{k}}(Y) \rho(Y) dY = \delta_{k\hat{k}}, \quad (52)$$

where $\rho(Y)$ is the combined PDF of Y , Ω is the multidimensional domain of Y , and δ_{nm} is the Dirac delta, an expression of the polynomial expansion coefficients can be isolated. We multiply both sides of Eq. 50 by $\Phi_{\hat{k}}(Y)$, integrate both sides over the probability-weighted input domain, and sum over all \hat{k} to obtain the coefficients, sometimes referred to as polynomial expansion moments,

$$u_k = \frac{\langle u(Y) \Phi_k(Y) \rangle}{\langle \Phi_k(Y)^2 \rangle}, \quad (53)$$

$$= \langle u(Y) \Phi_k(Y) \rangle, \quad (54)$$

where we use the angled bracket notation to denote the probability-weighted inner product,

$$\langle f(Y) \rangle \equiv \int_{\Omega} f(Y) \rho(Y) dY. \quad (55)$$

When $u(Y)$ has an analytic form, these coefficients can be solved by integration; however, in general other methods must be applied to numerically perform the integral. While tools such as Monte Carlo integration can be used to evaluate the integral, the properties of Gaussian quadratures, because of the probability weights and domain, can be harnessed. This stochastic collocation method is discussed in section 7.4.1.4.

7.4.1.2 Polynomial Index Set Construction

The main concern in expanding a function in interpolating multidimensional polynomials is choosing appropriate polynomials to make up the expansion. There are many generic ways by which a polynomial set can be constructed. Here three static approaches are presented:

- Tensor Product
- Total Degree
- Hyperbolic Cross.

In the nominal tensor product case, $\Lambda(L)$ contains all possible combinations of polynomial indices up to truncation order L in each dimension, as:

$$\Lambda_{\text{TP}}(L) = \left\{ \bar{p} = (p_1, \dots, p_N) : \max_{1 \leq n \leq N} p_n \leq L \right\}. \quad (56)$$

The cardinality of this index set is $|\Lambda_{\text{TP}}(L)| = (L + 1)^N$. For example, for a two-dimensional input space ($N=2$) and truncation limit $L = 3$, the index set $\Lambda_{\text{TP}}(3)$ is given in Table 4, where the notation $(1, 2)$ signifies the product of a polynomial that is first order in Y_1 and second order in Y_2 .

(3,0)	(3,1)	(3,2)	(3,3)
(2,0)	(2,1)	(2,2)	(2,3)
(1,0)	(1,1)	(1,2)	(1,3)
(0,0)	(0,1)	(0,2)	(0,3)

Table 4. Tensor Product Index Set, $N = 2, L = 3$.

It is evident there is some inefficiencies in this index set. First, it suffers dramatically from the *curse of dimensionality*; that is, the number of polynomials required grows exponentially with increasing dimensions. Second, the total order of polynomials is not considered. Assuming the contribution of each higher-order polynomial is smaller than lower-order polynomials, the $(3,3)$ term is contributing sixth-order corrections that are likely smaller than the error introduced by ignoring fourth-order corrections $(4,0)$ and $(0,4)$. This leads to the development of the *total degree* (TD) and *hyperbolic cross* (HC) polynomial index set construction strategies [19].

In TD, only multidimensional polynomials whose *total* order at most L are permitted,

$$\Lambda_{\text{TD}}(L) = \left\{ \bar{p} = (p_1, \dots, p_N) : \sum_{n=1}^N p_n \leq L \right\}. \quad (57)$$

The cardinality of this index set is $|\Lambda_{\text{TD}}(L)| = \binom{L+N}{N}$, which grows with increasing dimensions much more slowly than TP. For the same $N = 2, L = 3$ case above, the TD index set is given in Table 5.

(3,0)			
(2,0)	(2,1)		
(1,0)	(1,1)	(1,2)	
(0,0)	(0,1)	(0,2)	(0,3)

Table 5. Total Degree Index Set, $N = 2, L = 3$

In HC, the *product* of polynomial orders is used to restrict allowed polynomials in the index set. This tends to polarize the expansion, emphasizing higher-order polynomials in each dimension but lower-order polynomials in combinations of dimensions, as:

$$\Lambda_{\text{HC}}(L) = \left\{ \bar{p} = (p_1, \dots, p_N) : \prod_{n=1}^N p_n + 1 \leq L + 1 \right\}. \quad (58)$$

The cardinality of this index set is bounded by $|\Lambda_{\text{HC}}(L)| \leq (L+1)(1+\log(L+1))^{N-1}$. It grows even more slowly than TD with increasing dimension, as shown in Table 6 for $N=2, L=3$.

(3,0)
(2,0)
(1,0) (1,1)
(0,0) (0,1) (0,2) (0,3)

Table 6. Hyperbolic Cross Index Set, $N=2, L=3$

It has been shown that the effectiveness of TD and HC as index set choices depends strongly on the regularity of the response [19]. TD tends to be most effective for infinitely-continuous response surfaces, while HC is more effective for surfaces with limited smoothness or discontinuities.

7.4.1.3 Anisotropy

While using TD or HC to construct the polynomial index set combats the curse of dimensionality present in TP, it is not eliminated and continues to be an issue for problems of large dimensionality. Another method that can be applied to mitigate this issue is index set anisotropy, or the unequal treatment of various dimensions. In this strategy, weighting factors $\alpha = (\alpha_1, \dots, \alpha_n, \dots, \alpha_N)$ are applied in each dimension to allow additional polynomials in some dimensions and less in others. This change adjusts the TD and HC construction rules as follows, where $|\alpha|_1$ is the one-norm of α .

$$\tilde{\Lambda}_{TD}(L) = \left\{ \bar{p} = (p_1, \dots, p_N) : \sum_{n=1}^N \alpha_n p_n \leq |\alpha|_1 L \right\}, \quad (59)$$

$$\tilde{\Lambda}_{\text{HC}}(L) = \left\{ \bar{p} = (p_1, \dots, p_N) : \prod_{n=1}^N (p_n + 1)^{\alpha_n} \leq (L+1)^{|\alpha|_1} \right\} \quad (60)$$

As it is desirable to obtain the isotropic case from a reduction of the anisotropic cases, define the one-norm for the weights is defined as:

$$|\alpha|_1 = \frac{\sum_{n=1}^N \alpha_n}{N}. \quad (61)$$

Considering the same case above ($N=2, L=3$), it can be applied weights $\alpha_1=5, \alpha_2=3$, and the resulting index sets are Tables 7 (TD) and 8 (HC).

There are many methods by which anisotropy weights can be assigned. Often, if a problem is well-known to an analyst, it may be enough to use heuristics to assign importance arbitrarily.

(2,0)				
(1,0)	(1,1)	(1,2)		
(0,0)	(0,1)	(0,2)	(0,3)	(0,4)

Table 7. Anisotropic Total Degree Index Set, $N = 2, L = 3$.

(1,0)			
(0,0)	(0,1)	(0,2)	(0,3)

Table 8. Anisotropic Hyperbolic Cross Index Set, $N = 2, L = 3$.

Otherwise, a smaller uncertainty quantification solve can be used to roughly determine sensitivity coefficients (such as Pearson coefficients), and the inverse of those can then be applied as anisotropy weights. Sobol coefficients obtained from first- or second-order HDMR, an additional sampling strategy present in RAVEN, could also serve as a basis for these weights. A good choice of anisotropy weight can greatly speed up convergence; however, a poor choice can slow convergence considerably, as computational resources are used to resolve low-importance dimensions.

7.4.1.4 Stochastic Collocation

Stochastic collocation is the process of using collocated points to approximate integrals of stochastic space numerically. In particular consider using Gaussian quadratures (Legendre, Hermite, Laguerre, and Jacobi) corresponding to the polynomial expansion polynomials for numerical integration. Quadrature integration takes the form:

$$\int_a^b f(x)\rho(x) = \sum_{\ell=1}^{\infty} w_{\ell}f(x_{\ell}), \quad (62)$$

$$\approx \sum_{\ell=1}^{\hat{L}} w_{\ell}f(x_{\ell}), \quad (63)$$

where w_{ℓ}, x_{ℓ} are corresponding points and weights belonging to the quadrature set, truncated at order \hat{L} . At this point, this \hat{L} should not be confused with the polynomial expansion truncation order L . This expression can be simplified using the operator notation

$$q^{(\hat{L})}[f(x)] \equiv \sum_{\ell=1}^{\hat{L}} w_{\ell}f(x_{\ell}). \quad (64)$$

A nominal multidimensional quadrature is the tensor product of individual quadrature weights and points, and can be written

$$Q^{(\mathbf{L})} = q_1^{(\hat{L}_1)} \otimes q_2^{(\hat{L}_2)} \otimes \dots, \quad (65)$$

$$= \bigotimes_{n=1}^N q_n^{(\hat{L}_n)}. \quad (66)$$

It is worth noting that each quadrature may have distinct points and weights; they need to not be constructed using the same quadrature rule. In general, one-dimensional Gaussian quadrature excels in exactly integrating polynomials of order $2p - 1$ using p points and weights; equivalently, it requires $(p + 1)/2$ points to integrate an order p polynomial.

For convenience, the coefficient integral to be evaluated is here reported again, Eq. 53.

$$u_k = \langle u(Y) \Phi_k(Y) \rangle. \quad (67)$$

This integral can be approximated with the appropriate Gaussian quadrature as

$$u_k \approx Q^{(\hat{\mathbf{L}})}[u(Y) \Phi_k(Y)], \quad (68)$$

where bold vector notation is used to note the order of each individual quadrature, $\hat{\mathbf{L}} = [\hat{L}_1, \dots, \hat{L}_n, \dots, \hat{L}_N]$. For clarity, the bold notation is removed and it is assumed a one-dimensional problem, which extrapolates as expected into the multidimensional case.

$$u_k \approx q^{(\hat{L})}[u(Y) \Phi_k(Y)], \quad (69)$$

$$= \sum_{\ell=1}^{\hat{L}} w_\ell u(Y_\ell) \Phi_k(Y_\ell). \quad (70)$$

To determine the quadrature order \hat{L} needed to accurately integrate this expression, consider the gPC formulation for $u(Y)$ in Eq. 50 and replace it in the sum,

$$u_k \approx \sum_{\ell=1}^{\hat{L}} w_\ell \Phi_k(Y_\ell) \sum_{k \in \Lambda(L)} u_{\hat{k}} \Phi_{\hat{k}}(Y_\ell). \quad (71)$$

Using orthogonal properties of the polynomials, this reduces as $\hat{L} \rightarrow \infty$ to

$$u_k \approx \sum_{\ell=1}^{\hat{L}} w_\ell u_k \Phi_k(Y_\ell)^2. \quad (72)$$

Thus, the integral, to the same error introduced by truncating the gPC expansion, the quadrature is approximating an integral of order $2k$. As a result, the quadrature order should be ordered:

$$p = \frac{2k + 1}{2} = k + \frac{1}{2} < k + 1, \quad (73)$$

so it can conservatively used $p = k + 1$. In the case of the largest polynomials with order $k = L$, the quadrature size \hat{L} is the same as $L + 1$. It is worth noting that if $u(Y)$ is effectively of much higher-order polynomial than L , this equality for quadrature order does not hold true; however, it also means that gPC of order L will be a poor approximation.

While a tensor product of highest-necessary quadrature orders could serve as a suitable multidimensional quadrature set, we can make use of Smolyak-like sparse quadratures to reduce the number of function evaluations necessary for the TD and HC polynomial index set construction strategies.

7.4.1.5 Smolyak Sparse Grids

Smolyak sparse grids [20] are an attempt to discover the smallest necessary quadrature set to integrate a multidimensional integral with varying orders of predetermined quadrature sets. In RAVEN case, the polynomial index sets determine the quadrature orders each one needs in each dimension to be integrated accurately. For example, the polynomial index set point (2,1,3) requires three points in Y_1 , two in Y_2 , and four in Y_3 , or

$$Q^{(2,1,3)} = q_1^{(3)} \otimes q_2^{(2)} \otimes q_3^{(4)}. \quad (74)$$

The full tensor grid of all collocation points would be the tensor product of all quadrature for all points, or

$$Q^{(\Lambda(L))} = \bigotimes_{k \in \Lambda} Q^{(k)}. \quad (75)$$

Smolyak sparse grids consolidate this tensor form by adding together the points from tensor products of subset quadrature sets. Returning momentarily to a one-dimensional problem, introduce the notation

$$\Delta_k^{(\hat{L})}[f(x)] \equiv (q_k^{(\hat{L})} - q_{k-1}^{(\hat{L})})[f(x)], \quad (76)$$

$$q_0^{(\hat{L})}[f(x)] = 0. \quad (77)$$

A Smolyak sparse grid is then defined and applied to the desired integral in Eq. 53,

$$S_{\Lambda,N}^{(\hat{L})}[u(Y)\Phi_k(Y)] = \sum_{k \in \Lambda(L)} \left(\Delta_{k_1}^{(\hat{L}_1)} \otimes \cdots \otimes \Delta_{k_N}^{(\hat{L}_N)} \right) [u(Y)\Phi_k(Y)]. \quad (78)$$

Equivalently, and in a more algorithm-friendly approach,

$$S_{\Lambda,N}^{(\hat{L})}[u(Y)\Phi_k(Y)] = \sum_{k \in \Lambda(L)} c(k) \bigotimes_{n=1}^N q_n^{(\hat{L}_n)}[u(Y)\Phi_k(Y)] \quad (79)$$

where

$$c(k) = \sum_{\substack{j=\{0,1\}^N, \\ k+j \in \Lambda}} (-1)^{|j|_1}, \quad (80)$$

using the traditional 1-norm for $|j|_1$. The values for u_k can then be calculated as

$$u_k = \langle u(Y) \Phi_k(Y) \rangle, \quad (81)$$

$$\approx S_{\Lambda, N}^{(\hat{L})}[u(Y) \Phi_k(Y)]. \quad (82)$$

With this numerical method to determine coefficients, a complete method for performing SCgPC analysis in an algorithmic manner is obtained.

7.4.2 Sparse Grid Collocation sampling through RAVEN

The goals of this section are about learning how to:

1. Set up a Sparse Grid Collocation sampling for the construction of a suitable surrogate model of a driven code
2. Construct a GaussPolynomialRom surrogate model (training stage)
3. Use the constructed GaussPolynomialRom surrogate model instead of the driven code.

To accomplish these tasks, the following RAVEN **Entities** (XML blocks in the input files) need to be defined:

1. *RunInfo*:

```
<RunInfo>
  <JobName>ChapterVII-I/SparseGrid</JobName>
  <Sequence>
    sample, Ntrain, sampleROM, writeHistories
  </Sequence>
  <WorkingDir>ChapterVII-I/SparseGrid</WorkingDir>
  <batchSize>12</batchSize>
</RunInfo>
```

As reported in section 3.2, the *RunInfo* **Entity** is intended to set up the analysis that the user wants to perform. In this specific case, four steps (**<Sequence>**) are going to be sequentially run using twelve processors (**<batchSize>**).

2. *Files*:

```
<Files>
  <Input name="referenceInput.xml"
    type="input">referenceInput.xml</Input>
</Files>
```

Since the driven code uses a single input file, in this section the original input is placed. As detailed in the user manual the attribute `name` represents the alias that is going to be used in all the other input blocks in order to refer to this file.

3. *Models*:

```
<Models>
  <Code name="testModel" subType="GenericCode">
    <executable>
      ../physicalCode/analyticalbateman/AnalyticalDplMain.py
    </executable>
    <clargs arg="python" type="prepend"/>
    <clargs arg="" extension=".xml" type="input"/>
    <clargs arg="_" extension=".csv" type="output"/>
    <prepend>python</prepend>
  </Code>
  <ROM name="NROM" subType="GaussPolynomialRom">
    <Target>A,B,C,D</Target>
    <Features>
      sigma-A,sigma-B,sigma-C,
      sigma-D,decay-A,decay-B,
      decay-C,decay-D
    </Features>
    <IndexSet>TotalDegree</IndexSet>
    <PolynomialOrder>3</PolynomialOrder>
    <Interpolation poly="Legendre"
      quad="Legendre">sigma-A</Interpolation>
    <Interpolation poly="Legendre"
      quad="Legendre">sigma-B</Interpolation>
    <Interpolation poly="Legendre"
      quad="Legendre">sigma-C</Interpolation>
    <Interpolation poly="Legendre"
      quad="Legendre">sigma-D</Interpolation>
    <Interpolation poly="Legendre"
      quad="Legendre">decay-A</Interpolation>
    <Interpolation poly="Legendre"
      quad="Legendre">decay-B</Interpolation>
    <Interpolation poly="Legendre"
      quad="Legendre">decay-C</Interpolation>
    <Interpolation poly="Legendre"
      quad="Legendre">decay-D</Interpolation>
  </ROM>
</Models>
```


As mentioned above, the goal of this example is the generation of a `GaussPolynomialRom` for sub-sequential usage instead of the original code. Indeed, in addition to the previously explained Code model, the ROM of type *GaussPolynomialRom* is here specified. The ROM will be generated through a Sparse Grid Collocation sampling strategy. All the 4 targets A, B, C, D are going to be modeled through this ROM as function of the uncertain parameters *sigmas* and *decays*.

4. *Distributions*:

```
<Distributions>
  <Uniform name="sigmaA">
    <lowerBound>6.9</lowerBound>
    <upperBound>8.1</upperBound>
  </Uniform>
  <Uniform name="sigmaB">
    <lowerBound>3.9</lowerBound>
    <upperBound>5.1</upperBound>
  </Uniform>
  <Uniform name="sigmaC">
    <lowerBound>1.9</lowerBound>
    <upperBound>3.1</upperBound>
  </Uniform>
  <Uniform name="sigmaD">
    <lowerBound>0.9</lowerBound>
    <upperBound>1.1</upperBound>
  </Uniform>
  <Uniform name="decayConstantA">
    <lowerBound>0.0000000038</lowerBound>
    <upperBound>0.0000000052</upperBound>
  </Uniform>
  <Uniform name="decayConstantB">
    <lowerBound>0.0000000058</lowerBound>
    <upperBound>0.0000000072</upperBound>
  </Uniform>
  <Uniform name="decayConstantC">
    <lowerBound>0.0000000068</lowerBound>
    <upperBound>0.0000000082</upperBound>
  </Uniform>
  <Uniform name="decayConstantD">
    <lowerBound>0.0000000078</lowerBound>
    <upperBound>0.0000000092</upperBound>
  </Uniform>
</Distributions>
```

In the Distributions XML section, the stochastic model for the uncertainties treated by the Sparse Grid Collocation sampling are reported. In this case eight distributions are defined:

- $\sigma_A \sim \mathbb{U}(6.9, 8.1)$, used to model the uncertainty associated with the Model *sigma-A*
- $\sigma_B \sim \mathbb{U}(3.9, 5.1)$, used to model the uncertainty associated with the Model *sigma-B*
- $\sigma_C \sim \mathbb{U}(1.9, 3.1)$, used to model the uncertainty associated with the Model *sigma-C*
- $\sigma_D \sim \mathbb{U}(0.9, 1.1)$, used to model the uncertainty associated with the Model *sigma-D*
- $\text{decayConstant}_A \sim \mathbb{U}(3.8e-9, 5.2e-9)$, used to model the uncertainty associated with the Model *decay-A*
- $\text{decayConstant}_B \sim \mathbb{U}(5.8e-9, 7.2e-9)$, used to model the uncertainty associated with the Model *decay-B*
- $\text{decayConstant}_C \sim \mathbb{U}(6.8e-9, 8.2e-9)$, used to model the uncertainty associated with the Model *decay-C*
- $\text{decayConstant}_D \sim \mathbb{U}(7.8e-9, 9.2e-9)$, used to model the uncertainty associated with the Model *decay-D*.

5. Samplers:

```
<Samplers>
  <SparseGridCollocation name="NSG" parallel="1">
    <variable name="sigma-A">
      <distribution>sigmaA</distribution>
    </variable>
    <variable name="sigma-B">
      <distribution>sigmaB</distribution>
    </variable>
    <variable name="sigma-C">
      <distribution>sigmaC</distribution>
    </variable>
    <variable name="sigma-D">
      <distribution>sigmaD</distribution>
    </variable>
    <variable name="decay-A">
      <distribution>decayConstantA</distribution>
    </variable>
    <variable name="decay-B">
      <distribution>decayConstantB</distribution>
    </variable>
```

```

<variable name="decay-C">
  <distribution>decayConstantC</distribution>
</variable>
<variable name="decay-D">
  <distribution>decayConstantD</distribution>
</variable>
<ROM class="Models" type="ROM">NRom</ROM>
</SparseGridCollocation>
</Samplers>

```

In order to employ the Sparse Grid Collocation sampling strategy, a **<SparseGridCollocation>** node needs to be inputted. As it can be seen from above, each variable is associated to a different distribution, defined in the **<Distributions>** block. In addition, the *GaussPolynomialRom* **<ROM>** is inputted. The setting of this ROM (e.g. polynomial order, Index set method, etc.) determines how the Stochastic Collocation Method is employed.

6. *DataObjects*:

```

<DataObjects>
  <PointSet name="inputPlaceholder">
    <Input>
      sigma-A, sigma-B,
      sigma-C, sigma-D,
      decay-A, decay-B, decay-C,
      decay-D
    </Input>
    <Output>OutputPlaceholder</Output>
  </PointSet>
  <PointSet name="samples">
    <Input>
      sigma-A, sigma-B,
      sigma-C, sigma-D,
      decay-A, decay-B, decay-C,
      decay-D
    </Input>
    <Output>A, B, C, D, time</Output>
  </PointSet>
  <PointSet name="samplesROM">
    <Input>
      sigma-A, sigma-B,
      sigma-C, sigma-D,
      decay-A, decay-B, decay-C,
      decay-D
    </Input>
  </PointSet>
</DataObjects>

```

```

        <Output>A, B, C, D</Output>
    </PointSet>
    <HistorySet name="histories">
        <Input>
            sigma-A, sigma-B,
            sigma-C, sigma-D,
            decay-A, decay-B, decay-C,
            decay-D
        </Input>
        <Output>A, B, C, D, time</Output>
    </HistorySet>
</DataObjects>

```

In this block, four *DataObjects* are defined: 1) PointSet named “samples” used to collect the final outcomes of the code, 2) HistorySet named “histories” in which the full time responses of the variables A, B, C, D are going to be stored, 3) PointSet named “inputPlaceholder” used as input of the sampling applied on the constructed ROM and 4) PointSet named “samplesROM” used to collect the final outcomes of the ROM perturbations.

7. Steps:

```

<Steps>
    <MultiRun name="sample">
        <Input class="Files"
            type="input">referenceInput.xml</Input>
        <Model class="Models" type="Code">testModel</Model>
        <Sampler class="Samplers"
            type="SparseGridCollocation">NSG</Sampler>
        <Output class="DataObjects"
            type="PointSet">samples</Output>
        <Output class="DataObjects"
            type="HistorySet">histories</Output>
    </MultiRun>
    <RomTrainer name="Ntrain">
        <Input class="DataObjects"
            type="PointSet">samples</Input>
        <Output class="Models" type="ROM">NROM</Output>
    </RomTrainer>
    <MultiRun name="sampleROM" pauseAtEnd="false">
        <Input class="DataObjects"
            type="PointSet">inputPlaceholder</Input>
        <Model class="Models" type="ROM">NROM</Model>
        <Sampler class="Samplers"
            type="SparseGridCollocation">NSG</Sampler>
    </MultiRun>
</Steps>

```

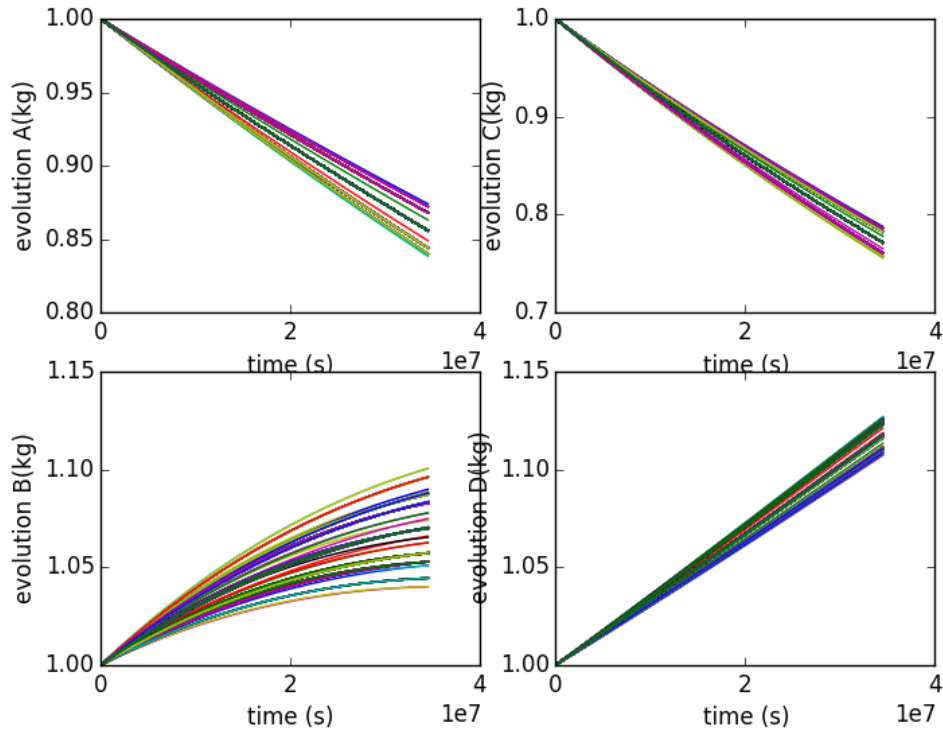


Figure 17. Plot of the samples generated by the Stratified sampling for variables A, B, C, D .

```

<Output class="DataObjects"
  type="PointSet">samplesROM</Output>
</MultiRun>
<IOStep name="writeHistories" pauseAtEnd="True">
  <Input class="DataObjects"
    type="HistorySet">histories</Input>
  <Input class="DataObjects"
    type="PointSet">samples</Input>
  <Input class="DataObjects"
    type="PointSet">samplesROM</Input>
  <Output
    class="OutStreams"
    type="Plot">samplesPlot3D</Output>
  <Output
    class="OutStreams"
    type="Plot">historyPlot</Output>
  <Output
    class="OutStreams"
    type="Plot">samplesROMPlot3D</Output>

```

```

<Output          class="OutStreams"
  type="Print">samples</Output>
<Output          class="OutStreams"
  type="Print">samplesROM</Output>
<Output          class="OutStreams"
  type="Print">histories</Output>
</IOStep>
</Steps>

```

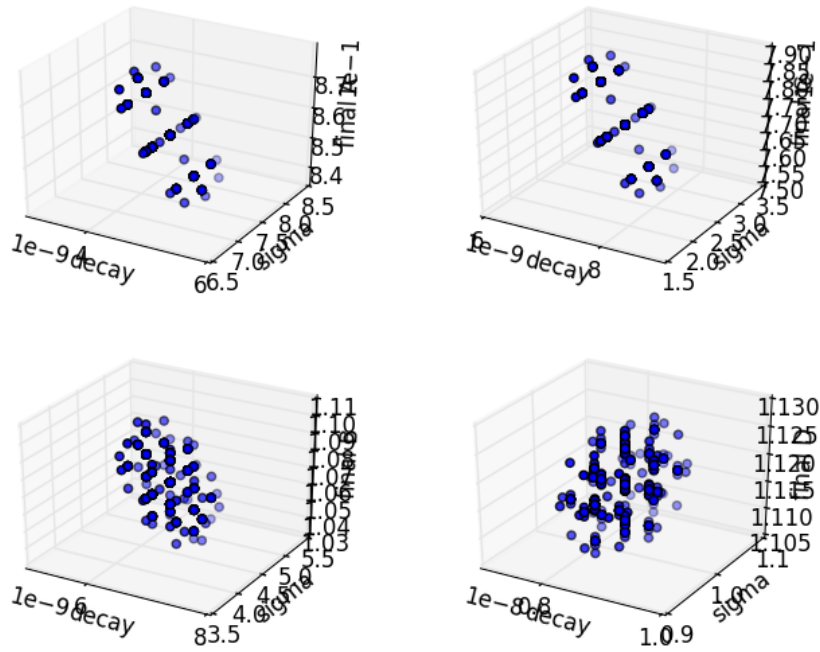


Figure 18. Plot of the samples generated by the Stratified sampling for variables A, B, C, D .

Finally, the previously defined **Entities** can be combined in the **<Steps>** block. As inferable, four **<Steps>** have been inputted:

- **<MultiRun>** named “sample”, used to run the multiple instances of the driven code and collect the outputs in the two *DataObjects*. As it can be seen, the **<Sampler>** is inputted to communicate to the *Step* that the driven code needs to be perturbed through the Sparse Grid Collocation sampling;

- **<RomTrainer>** named “Ntrain”, used to train (i.e., construct) the GaussPolynomial ROM. This step is essential if the user want to use the ROM in later steps;
- **<MultiRun>** named “sampleROM”, used to run the multiple instances of the previously constructed ROM and collect the outputs in the *samplesROM DataObjects*. As it can be seen, the same Sparse Grid Collocation sampler is here used.
- **<IOStep>** named “writeHistories”, used to 1) dump the “histories”, “samples” and “samplesROM” *DataObjects Entity* in a CSV file and 2) plot the data in the PNG file and on the screen.

As previously mentioned, Figure 17 shows the evolution of the outputs A, B, C, D under uncertainties. Figures 18 and 19 show the final responses of the sampling employed using the driven code and the ROM, respectively. As it can be seen, the constructed ROM can perfectly represent the response of the driven code. This example shows the potential of reduced order modeling, in general, and of the *GaussPolynomialRom*, in particular.

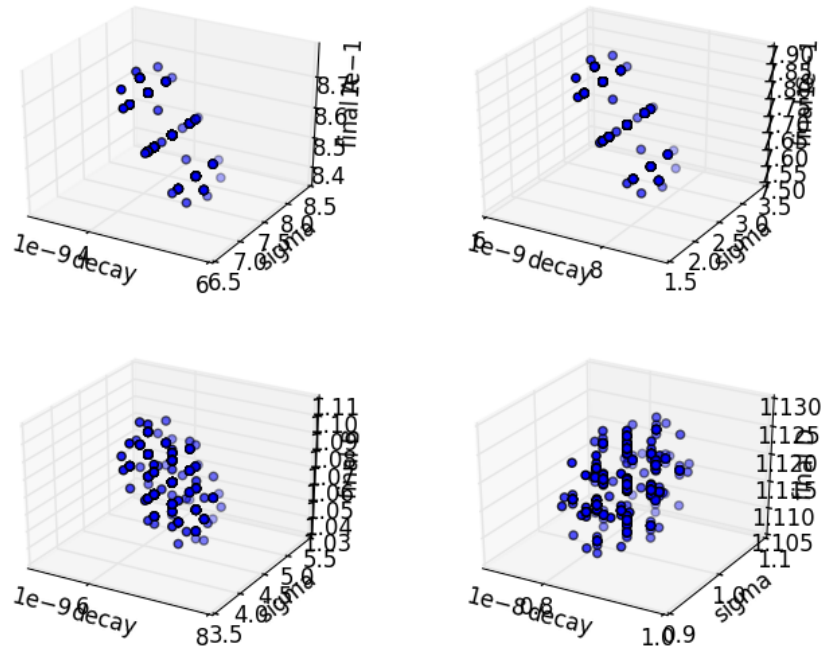


Figure 19. Plot of the samples generated by the Stratified sampling for variables A, B, C, D .

8 Adaptive Sampling Strategies

Performing UQ and Dynamic PRA can be challenging from a computational point of view. The *Forward* sampling strategies reported in the previous Section can lead to a large number of unnecessary evaluations of the physical model leading to an unacceptable resource expenses (CPU time). In addition, the *Forward* methodologies are not designed to leverage the information content that is extractable from the simulations already concluded.

To overcome these limitations, in RAVEN several adaptive algorithms are available:

1. *Limit Surface Search*
2. *Adaptive Dynamic Event Tree*
3. *Adaptive Hybrid Dynamic Event Tree*
4. *Adaptive Sparse Grid*
5. *Adaptive Sobol Decomposition.*

In this Section, only the first algorithm is going to be reported, explaining the theory behind it and providing RAVEN examples.

8.1 Limit Surface Search Method

The motivation behind the choice of adaptive sampling strategies is that numerical simulations are often computationally expensive, time consuming, and with a large number of uncertain parameters. Thus, exploring the space of all possible simulation outcomes is almost unfeasible using finite computing resources. During DPRA analysis, it is important to discover the relationship between a potentially large number of uncertain parameters and the response of a simulation using as few simulation trials as possible. This is a typical context where “goal” oriented sampling could be beneficial. The “Limit Surface Search method” is a scheme where few observations, obtained from the model run, are used to build a simpler and computationally faster mathematical representation of the model, ROM, also known as Surrogate Model (ROM or SM). The ROM (see Section 10.1) is then used to predict where further exploration of the input space could be most informative. This information is used to select new locations in the input space for which a code run is executed (see Figure 20). The new observations are used to update the ROM and this process iterates until, within a certain metric, it is converged.

In the case of the “Limit Surface (LS) Search method”, a ROM is used to determine the location in the input space where further observations are most informative to establish the location of the

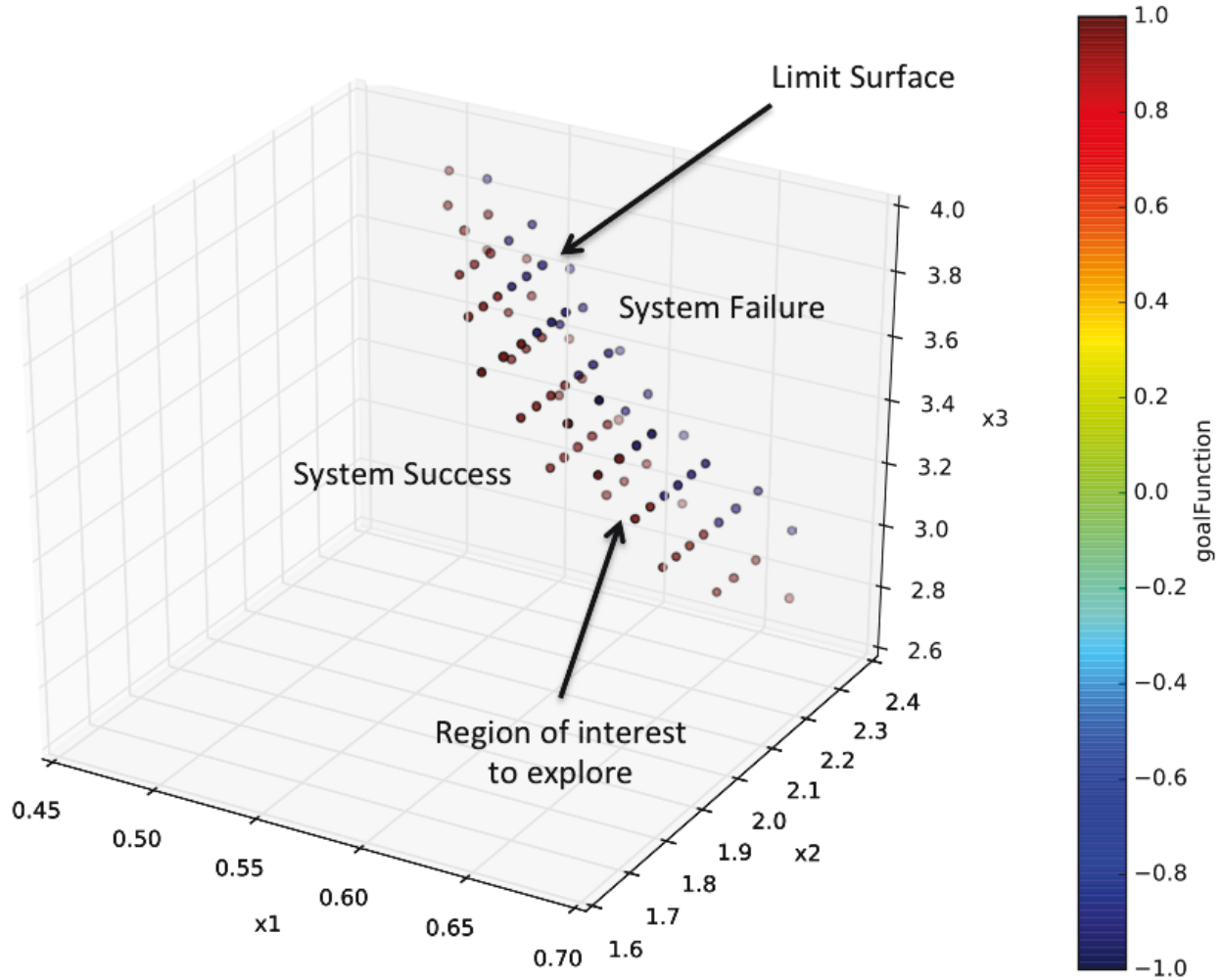


Figure 20. Example of limit surface in the uncertain space.

LS, then code runs are executed on those locations and the ROM updated. The process continues until the location of the LS is established within a certain tolerance.

8.1.1 Limit Surface Theory

To properly explain the LS concept and relative properties, it is necessary to analyze the idea behind the LSs, firstly, from a mathematical and, secondly, from a practical point of view. Consider a dynamic system that is represented in the phase space by the Eq. 1 in Section 3. The equation

can be rewritten as follows:

$$\frac{\partial \bar{\theta}}{\partial t} = \bar{H}(\bar{x}, \bar{p}, t) \quad (83)$$

where:

- $\bar{\theta}$ represents the coordinate of the system in the phase space
- (\bar{x}, \bar{p}, t) independent variables that are separated, respectively, in spatial, temporal, and parametric space (distinction between (\bar{x}, \bar{p}, t) is purely based on engineering considerations).

Now it is possible to introduce the concept of “goal” function, C . C is a binary function that, based on the response of the system, can assume the value 0 (false) to indicate that the system is properly available (e.g., system success) and 1 (true) to indicate that the system is not available (e.g., failure of the system):

$$C(\bar{\theta}, \bar{x}, \bar{p}, t) = C(H(\bar{x}, \bar{p}, t), \bar{x}, \bar{p}, t) = C(\bar{x}, \bar{p}, t) \quad (84)$$

Without loss of generality, let's assume that C does not depend on time (e.g. $C \leftarrow \int_{t_0}^{t_{end}} dt C(\bar{x}, \bar{p}, t)$):

$$C = C(\bar{x}, \bar{p}) \quad (85)$$

To simplify the mathematical description of the LS concept, it is possible to hypothesize that the equation describing the PDF time evolution of the system in the phase space is of type Gauss Codazzi (in its Liouville's derivation) [21], which allows ensuring that all the stochastic phenomena in the system are representable as PDFs in the uncertain domain (see Section 3). This allows combining the parametric space with the initial condition space:

$$\begin{aligned} (\bar{x}) &\leftarrow (\bar{x}, \bar{p}) \\ C(\bar{x}) &\leftarrow C(\bar{x}, \bar{p}) \end{aligned} \quad (86)$$

This assumption is rarely violated, for example for those systems that present an intrinsic stochastic behavior (e.g., the dynamic of a particle of dust in the air where it continuously and randomly interacts with the molecules of air that “move” with different velocities and in different and random directions). In most of the cases of interest in the safety analysis, the above mentioned assumption is correct. The full heuristic approach to the characterization of system stochastic behaviors is reported in Section 3.

Under the above simplifications, it is possible to identify the region of the input space (V) leading to a specific outcome of the “goal” function. For example, it can be defined the failure region V_F as the region of the input space where $C = 1$:

$$V_F = \{\forall \bar{x} | C(\bar{x}) = 1\} \quad (87)$$

The definition of the complementary of the failure region is obviously:

$$V_F^c = \{\forall \bar{x} | C(\bar{x}) = 0\} \quad (88)$$

Its boundary is the named LS:

$$L_S = \partial V_F^c = \partial \{ \forall \bar{x} | C(\bar{x}) = 1 \} \quad (89)$$

The identification of the LS location is necessary to identify boundary regions for which the system under consideration will or will not exceed certain FOMs (e.g., operative margins). The LS location is extremely important for design optimization and, in addition, its informative content can be used to analyze the system to characterize its behavior from a stochastic point of view. Consider $\bar{x} \in V$ and $\bar{x} \sim \bar{X}$, where \bar{x} is the random variate realization of the stochastic variable \bar{X} . If $pdf_{\bar{X}}(\bar{x})$ is the probability density function of \bar{X} , the failure probability of the system (P_F) is:

$$P_F = \int_V d\bar{x} C(\bar{x}) pdf_{\bar{X}}(\bar{x}) = \int_{V_F + V_F^c} d\bar{x} C(\bar{x}) pdf_{\bar{X}}(\bar{x}) \quad (90)$$

And, based on the definition given in Equations 86 and 87:

$$\int_{V_F} d\bar{x} pdf_{\bar{X}}(\bar{x}) \quad (91)$$

Equations 90 and 91 are summarized by stating that the system failure probability is equivalent to the probability of the system being in the uncertain subdomain (region of the input space) that leads to a failure pattern. This probability is equal to the probability-weighted hyper-volume that is surrounded by the LS (see Figure 21).

It is beneficial for better understanding to assess the LS concept through an example related to the safety of an Nuclear Power Plant (NPP). As an example, consider a station black out (SBO) scenario in an NPP. Suppose that the only uncertain parameters are:

- t_F : Temperature that would determine the failure of the fuel cladding
- rt_{DGs} : Recovery time of the diesel generators (DGs) that can guarantee, through the emergency core cooling system (ECCS), the removal of the decay heat.

And, the corresponding CDF (uniform) is:

$$t_F \sim pdf_{T_F}(T_F) = \begin{cases} 0 & \text{if } t_F < t_{F_{min}} \\ \frac{1}{(t_{F_{max}} - t_{F_{min}}) = \Delta t_F} & \text{if } t_F > t_{F_{max}} \\ 0 & \end{cases} \quad (92)$$

$$rt_{DGs} \sim pdf_{RT_{DGs}}(rt_{DGs}) = \begin{cases} 0 & \text{if } rt_{DGs} < rt_{DGs_{min}} \\ \frac{1}{(rt_{DGs_{max}} - rt_{DGs_{min}}) = \Delta rt_{DGs}} & \text{if } rt_{DGs} > rt_{DGs_{max}} \\ 0 & \end{cases} \quad (93)$$

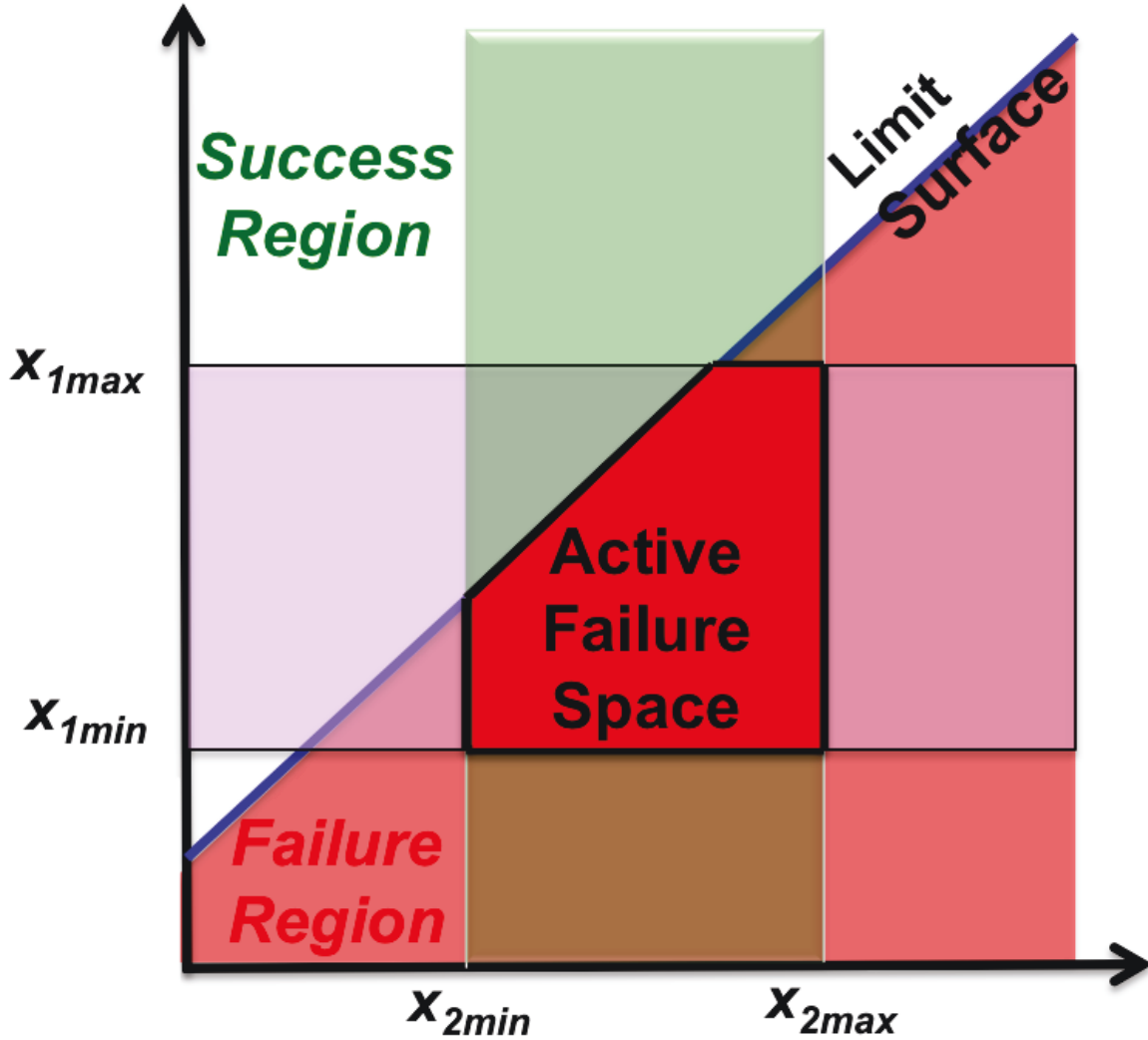


Figure 21. Example of limit surface probability of failure region.

For simplicity, assume that the clad temperature is a quadratic function of the DG recovery time in an SBO scenario:

$$t = t_0 + \alpha \times rt_{DGs}^2 \quad (94)$$

and that the $t_{Fmin} > t_0 + \alpha \times rt_{DGsmin}^2$ and $t_{Fmax} < t_0 + \alpha \times rt_{DGsmax}^2$. The LS, failure region, and active part of the failure region (failure region with non-zero probability) are illustrated, for example, in Figure 21 (in agreement with the above assumptions). In this case, the transition/failure

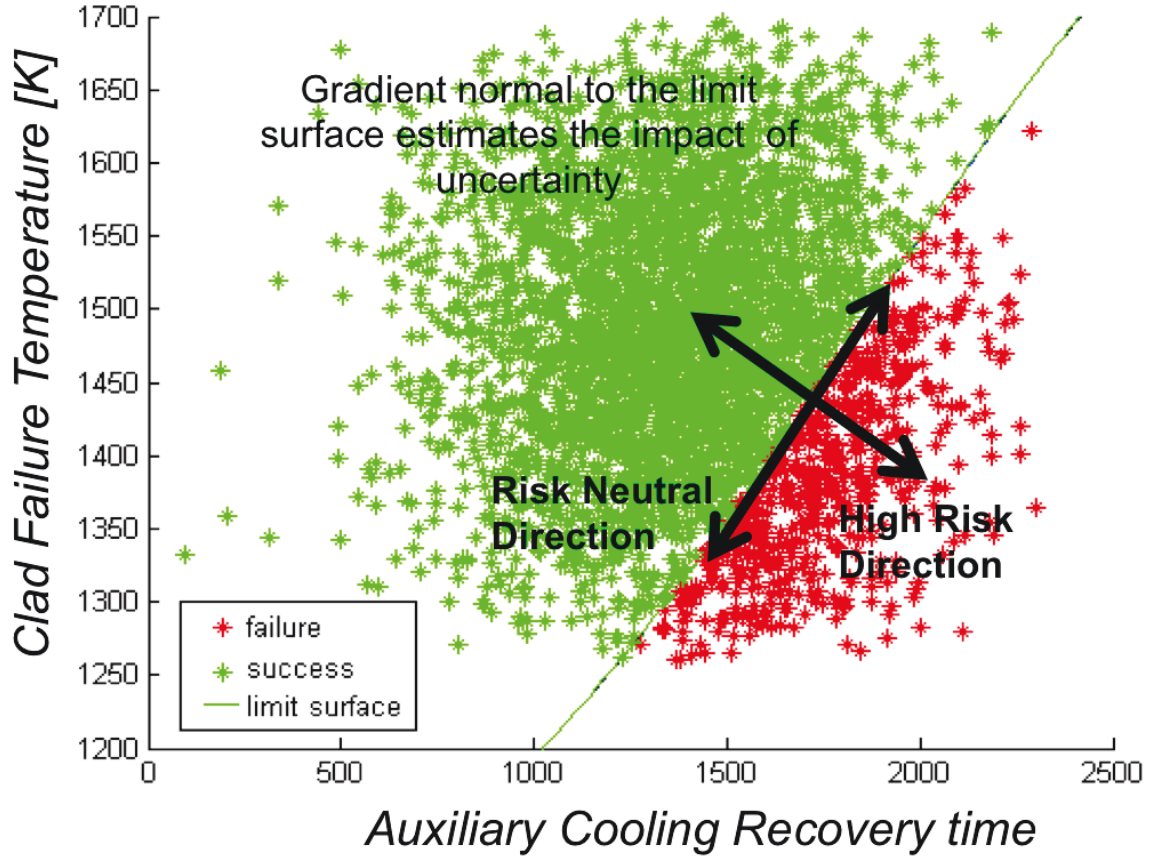


Figure 22. Example of limit surface highlighting the risk directions.

probability is evaluated as follows:

$$\begin{aligned}
 P_F &= \int_{V_F} d\bar{x} \, pdf_{\bar{X}}(\bar{x}) = \int_0^{+\infty} dt_F \, pdf_{T_F}(T_F) \int_{\sqrt{\frac{t_F-t_0}{\alpha}}}^{+\infty} drt_{DGs} \, pdf_{RT_{DGs}}(rt_{DGs}) = \\
 &= \int_{t_{Fmin}}^{t_{Fmax}} dt_F \frac{1}{t_{Fmax}-t_{Fmin}} \int_{\sqrt{\frac{t_F-t_0}{\alpha}}}^{rt_{DGsmax}} drt_{DGs} \frac{1}{rt_{DGsmax}-rt_{DGsmin}} = \\
 &= \frac{rt_{DGsmax}}{\Delta rt_{DGs}} + \frac{2\alpha}{3(\Delta rt_{DGs} \Delta t_F)} \left(3/2 \sqrt{\frac{t_{Fmin}-t_0}{\alpha}} - 3/2 \sqrt{\frac{t_{Fmax}-t_0}{\alpha}} \right)
 \end{aligned} \tag{95}$$

This simple example is useful to understand how the LS is defined in a practical application (that is analyzed numerically in the results Section) and how the hyper volume needs weighted with respect to the probability in the uncertain domain. An example of the computed LS is shown in Figure 22. In this figure the neutral and high risk directions are highlighted.

8.1.1.1 Limit Surface Search Algorithm

The identification of the LS location is extremely challenging, depending on the particular physics/phenomena that are investigated. To identify the real location of the LS, the evaluation of system responses is needed, through the high-fidelity code (RELAP 7, RELAP5-3D, etc.), in the full domain of uncertainty (infinite number of combinations of uncertainties represented by the respective PDFs). Obviously, this is not a feasible approach, and a reasonable approximation is to locate the LS on a Cartesian N-D grid, in the uncertain domain.

In reality, the location of the LS is not exactly determined but rather bounded. The algorithm determines the set of grid nodes between which the transition 0/1 of the “goal” function happens. This set is also classified with respect to the value of the “goal” function. With reference to Figure 23, for example, green is used for grid nodes with a “goal” function that equals 0 and red when the “goal” function equals 1. Each evaluation of the “goal” function in one of the grid nodes implies the evaluation of the high-fidelity code (e.g. system simulator) for the corresponding set of entries in the uncertain space. As already mentioned, the evaluation of the high fidelity code is computationally expensive and, in order to identify the LS, one should appraise each point in the N-D grid covering the uncertainty space. Discretization depends on the accuracy requested by the user. In most cases, this approach is not feasible and, consequentially, the process needs to be accelerated using “predicting” methods that are represented by the employment of supervised learning algorithms (i.e., ROMs).

This approach is commonly referred to as an active learning process that ultimately results in training of a ROM of type classifier capable of predicting the outcome of the “goal” function for any given point of the uncertain space. In an active learning process, a supervised learning algorithm is combined with criteria to choose the next node in the N D grid that needs explored, using the high fidelity physical model. This process is repeated until, under a particular metric, the prediction capabilities of the supervised learning algorithm do not improve by further increasing the training set.

In more detail, the iterative scheme could be summarized through the following steps:

1. A limited number of points in the uncertain space $\{\bar{x}_k\}$ are selected via one of the forward sampling strategies (e.g., stratified or Monte Carlo)
2. The high fidelity code is used to compute the status of the system for the set of points in the input set: $\{\bar{\theta}(t)\}_k = H(\{\bar{x}\}_k, t)$.
3. The “goal” function is evaluated at the phase space coordinate of the system: $\{c\}_k = C(\{\bar{\theta}(t)\}_k)$
4. The set of pairs $\{(\bar{x}, c)_k\}$ are used to train a ROM of type classifier, $G(\{\bar{x}_k\})$
5. The ROM classifier is used to predict the values of the “goal” function for all the N nodes

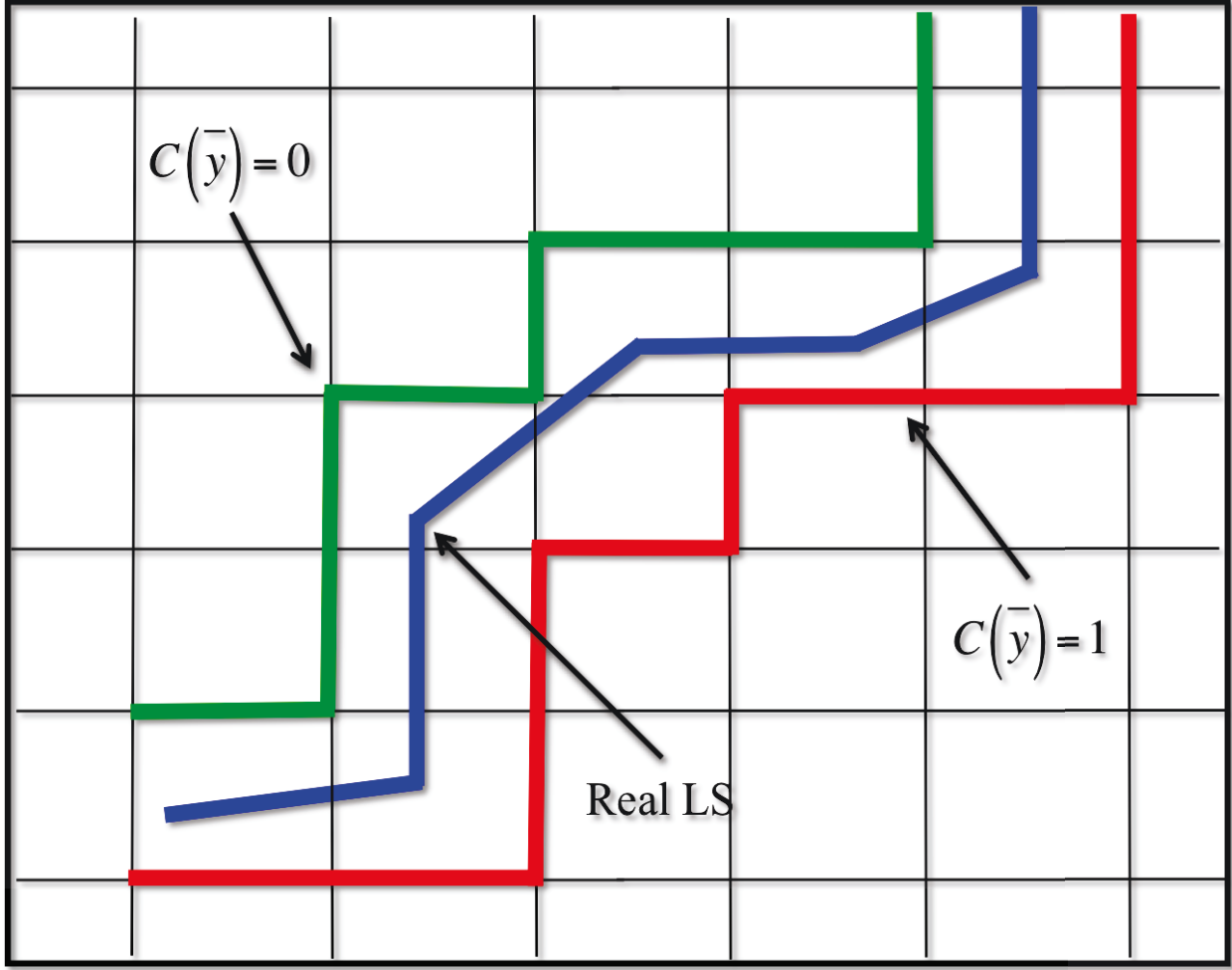


Figure 23. Example of limit surface search evaluation grid (where $\bar{y} = \bar{\theta}$).

of the N-D grid in the domain space:

$$\left(G\left(\{\bar{x}\}_j\right) \sim \{c\}_j, j = 1, \dots, N\right) \quad (96)$$

6. The values of the “goal” function are used to determine the LS location based on the change of values of $\{c\}_j$:

$$\{c\}_j \rightarrow \partial V_F \quad (97)$$

7. A new point is chosen to increase the training set and a new pair is generated
8. The procedure is repeated starting from Step 3 until convergence is achieved. The convergence is achieved when there are no changes in the location of the LS after a certain number

of consecutive iterations.

The iteration scheme is graphically shown in Figure 24. Note that there is an additional requirement

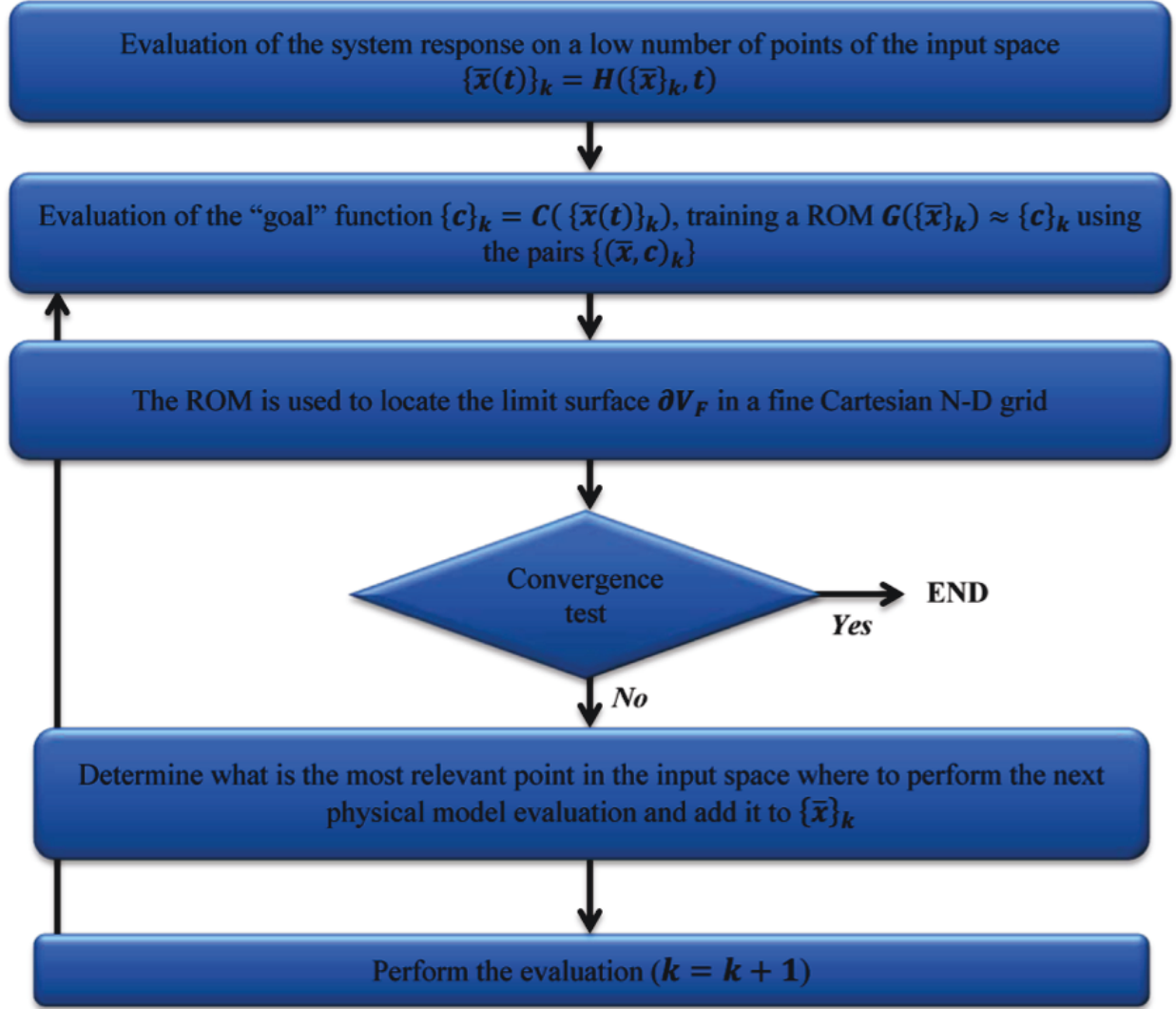


Figure 24. Limit surface search algorithm conceptual scheme.

regarding the LS search algorithm: the LS location has to stay constant for a certain number (user defined) of consecutive iterations. The reason for this choice is determined by the attempt to mitigate the effect of the build of non-linear bias in the searching pattern. Indeed, the searching algorithm might focus too much on a certain region of the LS while putting too few points in other zones and completely hiding undiscovered topological features of the LS. Regarding the strategy to choose the nodes on the N-D grid that needs evaluated in the iterative process for the LS

identification, it has been decided to employ a metric based on the distance between the predicted LS and the evaluations already performed. The points on the LS are ranked based on the distance from the closest training point already explored (the larger is the distance the higher is the score for the candidate point), and based on its persistence (the larger is the number of time the prediction of the “goal” function for that point have changed the higher is the score). Since this approach creates a queue of ranked candidates, it could be used also in the parallel implementation of the algorithm. When several training points are run in parallel, it is possible that the evaluation of one additional point does not alter dramatically the location of the LS. Consequently, it is possible that the candidate with the highest score is already being submitted for evaluation and possibly the simulation is not yet completed. In this case, to avoid submitting the same evaluation point twice, the algorithm searches among all the ranked candidates (in descending order) for the one that was not submitted for evaluation. Even if it is extremely unlikely that all the candidates were submitted, in this remote event, the method will choose the next point employing a Monte Carlo strategy.

8.1.1.2 Acceleration through Multi-grid Approach

The location of the LS, being a numerical iterative process, can be known given a certain tolerance. As already mentioned, the LS search is done by constructing an evaluation grid on which the acceleration ROM is inquired. The tolerance of the iterative process determines how the evaluation grid is discretized. Before addressing the acceleration scheme, it is important to introduce some concepts on the employed numerical process.

Assume that each of D dimensions of the uncertain domain is discretized with the same number of equally-spaced nodes N (see Figure 25), with discretization size indicated by h_i . Hence, the Cartesian grid contains N^D individual nodes, indexed through the multi-index vector $\bar{j} = (j_{i=1 \rightarrow D})$, $j_i \leq N \forall i$. Introducing the vectors $\bar{I} = (1, \dots, 1)$ and $\bar{N} = (N, \dots, N)$, the “goal” function is expressed on this N- D grid as:

$$C(\bar{x}) = \sum_{\bar{j}=\bar{I}}^{\bar{N}} \varphi_{\bar{j}}(\bar{x}) C(\bar{x}_{\bar{j}}) \quad (98)$$

where $\varphi_{\bar{j}}$ is the characteristic function of the hyper-volume $\Omega_{\bar{j}}$ surrounding the node $\bar{x}_{\bar{j}}$:

$$\varphi_{\bar{j}}(\bar{x}) = \begin{cases} 1 & \text{if } \bar{x} \in \Omega_{\bar{j}} \\ 0 & \text{if } \bar{x} \notin \Omega_{\bar{j}} \end{cases} \quad (99)$$

where:

$$\Omega_{\bar{j}} = \prod_{i=1}^D \left[x_{j_i} - \frac{h_i}{2}, x_{j_i} + \frac{h_i}{2} \right] \quad (100)$$

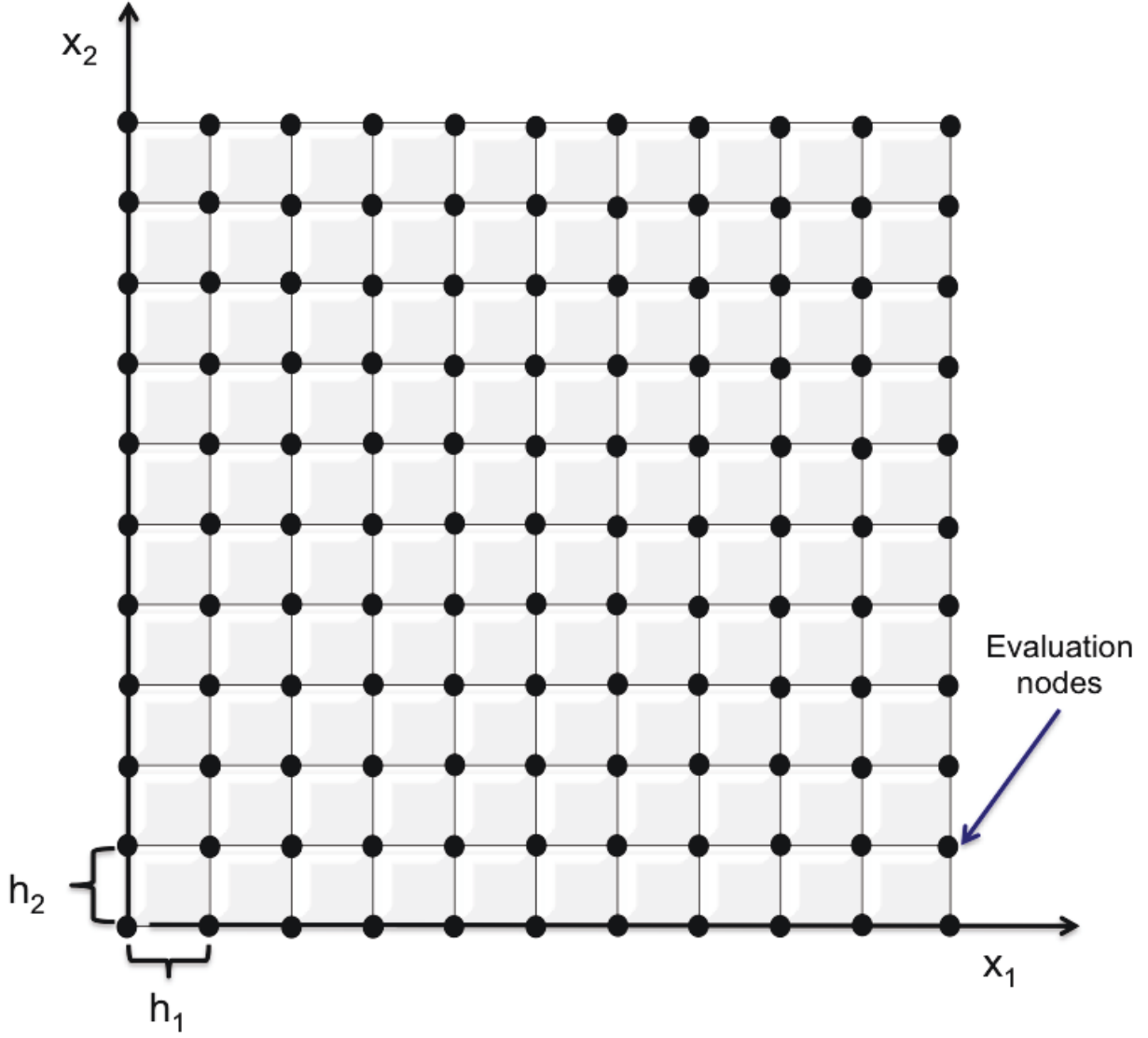


Figure 25. Discretization grid.

The probability of the uncertain parameters is expressed as:

$$pdf_{\bar{X}}(\bar{x}) = \sum_{\bar{j}=\bar{I}}^{\bar{N}} \varphi_{\bar{j}}(\bar{x}) pdf_{\bar{X}}(\bar{x}_{\bar{j}}) \quad (101)$$

Following the approach briefly explained in Section 8.1.1, the probability of the event (e.g., failure) could be expressed as:

$$P_F = \left(\prod_{i=1}^D h_i \right) \sum_{\bar{j}=\bar{I}}^{\bar{N}} pdf_{\bar{X}}(\bar{x}_{\bar{j}}) C(\bar{x}_{\bar{j}}) \quad (102)$$

Under certain assumptions, the concept of active hyper-volume V_A as the region of the input space identified by the support of the uncertain parameters' probability density functions $pdf_{\bar{X}}(\bar{x})$ could be introduced; Equation 102 is recast, using a Taylor expansion, as follows:

$$P_F = \int_V C(\bar{x}) pdf_{\bar{X}}(\bar{x}) d\bar{x} = \int_{V_A} C(\bar{x}) \left[\sum_{\bar{j}=\bar{I}}^{\bar{N}} \varphi_{\bar{j}}(\bar{x}) \left(pdf_{\bar{X}}(\bar{x}_{\bar{j}}) + \sum_{i=1}^D \frac{\partial pdf_{\bar{X}}}{\partial x_i} \Big|_{\bar{x}_{\bar{j}}} (x_i - x_{j_i}) \right) \right] d\bar{x} \quad (103)$$

And, considering the evaluation grid as:

$$P_F = \sum_{\substack{\bar{j}=\bar{I} \\ \bar{x}_{\bar{j}} \in V_A}}^{\bar{N}} \int_{\bar{x}_{\bar{j}}-\bar{h}/2}^{\bar{x}_{\bar{j}}+\bar{h}/2} C(\bar{x}) \left[\sum_{\bar{j}=\bar{I}}^{\bar{N}} \varphi_{\bar{j}}(\bar{x}) \left(pdf_{\bar{X}}(\bar{x}_{\bar{j}}) + \sum_{i=1}^D \frac{\partial pdf_{\bar{X}}}{\partial x_i} \Big|_{\bar{x}_{\bar{j}}} (x_i - x_{j_i}) \right) \right] d\bar{x} \quad (104)$$

At this point, it is possible to label, in the active hyper-volume, the sub-domain identified by the nodes where the “goal” function $C(\bar{x})$ changes its value (the frontier nodes between the region where $C(\bar{x}) = 1$ and $C(\bar{x}) = 0$) $V_A \cap V_{\partial V_F}$.

Consequently, it is possible to identify the sub-domains in which the “goal” function $C(\bar{x})$ is equal to 0 ($V_A \cap V_{\partial V_{C(\bar{x})=0}} \notin V_A \cap V_{\partial V_F}$):

$$\sum_{\substack{\bar{j}=\bar{I} \\ \bar{x}_{\bar{j}} \in V_A \cap V_{C(\bar{x})=0}}}^{\bar{N}} \int_{\bar{x}_{\bar{j}}-\bar{h}/2}^{\bar{x}_{\bar{j}}+\bar{h}/2} C(\bar{x}) \left(pdf_{\bar{X}}(\bar{x}_{\bar{j}}) + \sum_{i=1}^D \frac{\partial pdf_{\bar{X}}}{\partial x_i} \Big|_{\bar{x}_{\bar{j}}} (x_i - x_{j_i}) \right) d\bar{x} \quad (105)$$

in which the “goal” function $C(\bar{x})$ is equal to 1 ($V_A \cap V_{\partial V_{C(\bar{x})=1}} \notin V_A \cap V_{\partial V_F}$):

$$\begin{aligned} & \sum_{\substack{\bar{j}=\bar{I} \\ \bar{x}_{\bar{j}} \in V_A \cap V_{C(\bar{x})=1}}}^{\bar{N}} \int_{\bar{x}_{\bar{j}}-\bar{h}/2}^{\bar{x}_{\bar{j}}+\bar{h}/2} C(\bar{x}) \left(pdf_{\bar{X}}(\bar{x}_{\bar{j}}) + \sum_{i=1}^D \frac{\partial pdf_{\bar{X}}}{\partial x_i} \Big|_{\bar{x}_{\bar{j}}} (x_i - x_{j_i}) \right) d\bar{x} = \\ & = \sum_{\substack{\bar{j}=\bar{I} \\ \bar{x}_{\bar{j}} \in V_A \cap V_{C(\bar{x})=1}}}^{\bar{N}} \int_{\bar{x}_{\bar{j}}-\bar{h}/2}^{\bar{x}_{\bar{j}}+\bar{h}/2} \left(pdf_{\bar{X}}(\bar{x}_{\bar{j}}) + \sum_{i=1}^D \frac{\partial pdf_{\bar{X}}}{\partial x_i} \Big|_{\bar{x}_{\bar{j}}} (x_i - x_{j_i}) \right) d\bar{x} \end{aligned} \quad (106)$$

Equation 104 is now expressed as:

$$\begin{aligned}
P_F = & \sum_{\substack{\bar{j} = \bar{I} \\ \bar{x}_{\bar{j}} \in V_A \cap V_{C(\bar{x})=1}}}^{\bar{N}} \left(\prod_{i=1}^D h_i \right) pdf_{\bar{X}}(\bar{x}_{\bar{j}}) + O(h^{N+1}) + \\
& + \sum_{\substack{\bar{j} = \bar{I} \\ \bar{x}_{\bar{j}} \in V_A \cap V_{\partial V_f}}}^{\bar{N}} \int_{\bar{x}_{\bar{j}} - \bar{h}/2}^{\bar{x}_{\bar{j}} + \bar{h}/2} C(\bar{x}) \left(pdf_{\bar{X}}(\bar{x}_{\bar{j}}) + \sum_{i=1}^D \frac{\partial pdf_{\bar{X}}}{\partial x_i} \Big|_{\bar{x}_{\bar{j}}} (x_i - x_{j_i}) \right) d\bar{x}
\end{aligned} \tag{107}$$

As inferred from Equation 107, the process is bounded if the surface area-to-volume ratio (amount of surface area per unit volume) is in favor of the volume:

$$\sum_{\substack{\bar{j} = \bar{I} \\ \bar{x}_{\bar{j}} \in V_A \cap V_{C(\bar{x})=1}}}^{\bar{N}} \left(\prod_{i=1}^D h_i \right) pdf_{\bar{X}}(\bar{x}_{\bar{j}}) \gg \sum_{\substack{\bar{j} = \bar{I} \\ \bar{x}_{\bar{j}} \in V_A \cap V_{\partial V_f}}}^{\bar{N}} \left| \int_{\bar{x}_{\bar{j}} - \bar{h}/2}^{\bar{x}_{\bar{j}} + \bar{h}/2} pdf_{\bar{X}}(\bar{x}_{\bar{j}}) d\bar{x} \right| \tag{108}$$

If the grid is built in the transformed space of probability (i.e., replacing the measure $d\bar{x}$ with $d\bar{\mu} pdf_{\bar{X}}(\bar{x}_{\bar{j}})$) the condition expressed in Equation 108 is reduced:

$$number\ nodes \in V_A \cap V_{C(\bar{x})=1} \gg number\ nodes \in V_A \cap V_{\partial V_F} \tag{109}$$

This means that error is bounded by the total probability contained in the cells on the frontier of the LS.

Based on this derivation, it is clear how important it is to keep the content of the total probability on the frontier of the LS as low as possible, and simultaneously, increase the importance of the volume of the failure/event region as much as possible (to improve the surface area-to-volume ratio).

To do that, the step size in probability should be significantly reduced ($h_i^p \rightarrow 0^+$). Even if this is theoretically feasible, it is computational inapplicable. To approach a similar result, it is possible to learn from other numerical methods that use the technique of adaptive meshing for the resolution of the partial differential equation system (e.g., finite element methods).

For this reason, an acceleration scheme was designed and developed employing a multi-grid approach. The main idea, it is to recast the iterative process in two different sub-sequential steps. Firstly, performing the LS search on a coarse evaluation grid, and once converged, adaptively refining the cells that lie on the frontier of the LS ($V_A \cap V_{\partial V_F}$) and, consequentially, converging on the new refined grid.

The iteration scheme is graphically shown in Figure 26. In more detail, the iterative scheme could be summarized through the following steps:

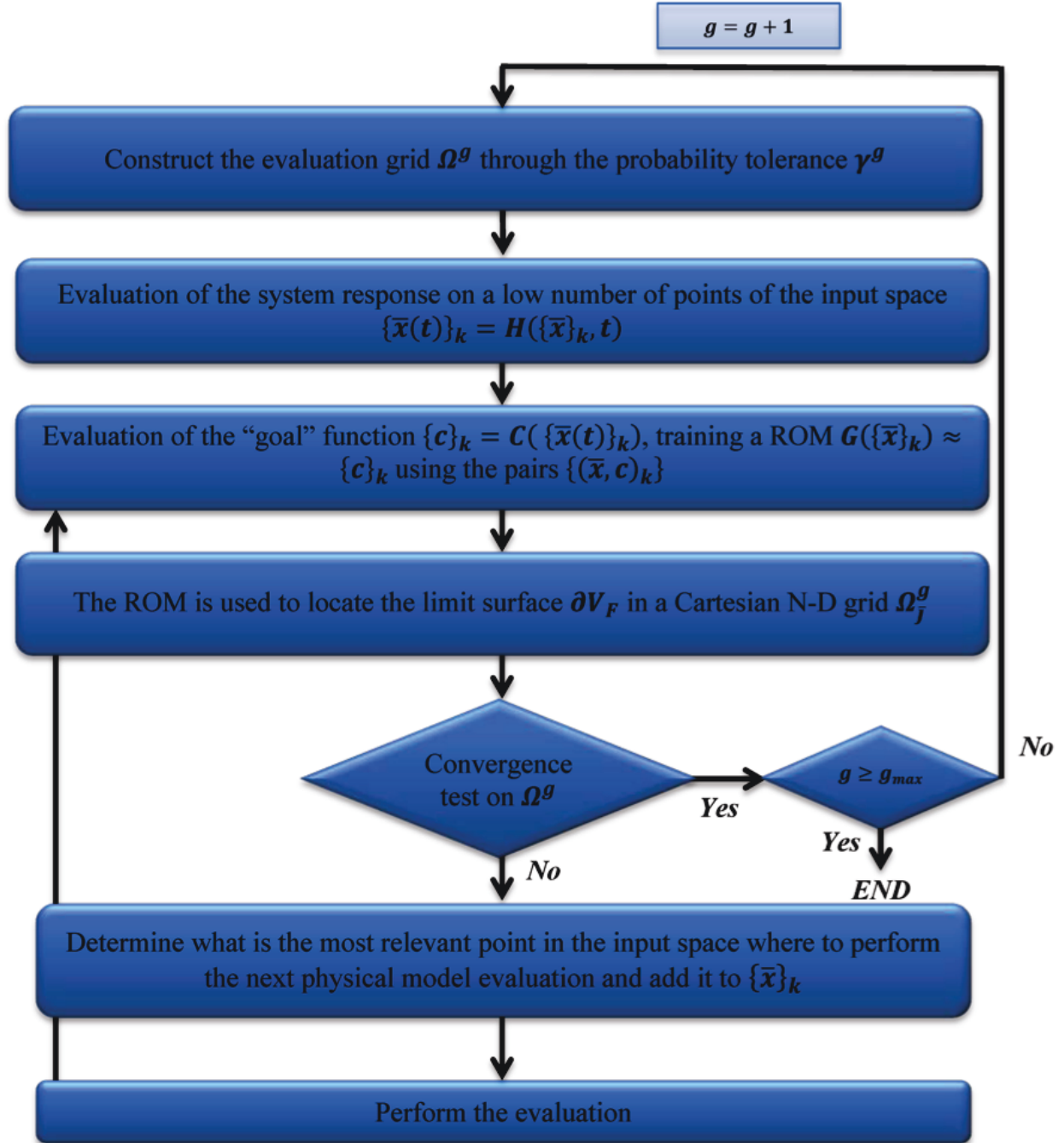


Figure 26. Multi-grid limit surface search scheme.

1. The user specifies two tolerances in probability (*CDF*) : $\gamma^{g=1}$ for the initial coarse grid and $\gamma^{g=2}$ for the refined grid, where $\gamma^{g=1} > \gamma^{g=2}$;

2. Following Equation 100, the initial coarse evaluation grid Ω^1 is constructed ($N^{g=1}$ total nodes). The discretization of this grid is done to have cells with a content of probability equal to $\gamma^{g=1}$.
3. A limited number of points in the uncertain space $\{\bar{x}_k\}$ are selected via one of the forward sampling strategies (e.g., stratified or Monte Carlo).
4. The high fidelity code is used to compute the status of the system for the set of points in the input set: $\{\bar{\theta}(t)\}_k = H(\{\bar{x}\}_k, t)$.
5. The “goal” function is evaluated at the phase space coordinate of the system: $\{c\}_k = C(\{\bar{\theta}(t)\}_k)$.
6. The set of pairs $\{(\bar{x}, c)_k\}$ are used to train a ROM of type classifier, $G(\{\bar{x}_k\})$.
7. The ROM classifier is used to predict the values of the “goal” function for all the $N^{g=1}$ nodes of the N-D grid in the domain space:

$$\left(G(\{\bar{x}\}_j) \sim \{c\}_j, j = 1, \dots, N^{g=1}\right) \quad (110)$$

8. The values of the “goal” function are used to determine the LS location based on the change of values of $\{c\}_j$:

$$\{c\}_j \rightarrow \partial V_F \quad (111)$$

9. A new point is chosen to increase the training set and a new pair is generated.
10. The procedure is repeated starting from Step 5 until convergence is achieved on grid Ω^g . The convergence is reached when there are no changes in the location of the LS after a certain number of consecutive iterations (user defined).
11. When the convergence is achieved on the coarse grid $\Omega^{g=1}$, all the cells that lie on the frontier of the LS ($V_A \cap V_{\partial V_F}$) are refined to contain an amount of probability equal to $\gamma^{g=2}$.
12. Steps 7 through 9 are performed based on the new refined grid. Finally, the process starts again by performing Steps 5 through 10, until the convergence is achieved in the refined grid.

As shown in Figure 26, the algorithm consists in searching the location of the LS proceeding with subsequential refinement of the sub-domain, in the active space, that contains the LS. In this way, the computational burden is kept as low as possible. In addition, another advantage of this approach is that, since the refinement grid represents a constrained domain, the sub-sequential ROM training process can be regularized, since the LS between an iteration and the other can move, at maximum, within the refinement domain.

8.1.2 Limit Surface Search sampling through RAVEN

The goal of this Section is to learn how to:

1. Set up a LS Search sampling for efficiently perturb a driven code
2. Use the LS Integral Post-processor for computing the probability of failure of the system subject to the same “goal” function
3. Plot the obtained LS.

In order to accomplish these tasks, the following RAVEN **Entities** (XML blocks in the input files) are defined:

1. *RunInfo*:

```
<RunInfo>
  <JobName>LSsearch</JobName>
  <Sequence>
    sample,computeLSintegral,writeHistories
  </Sequence>
  <WorkingDir>LSsearch</WorkingDir>
  <batchSize>8</batchSize>
</RunInfo>
```

As shown in Section 3.2, the *RunInfo* **Entity** is intended to set up the analysis that the user wants to perform. In this specific case, three steps (**<Sequence>**) are sequentially run using eight processors (**<batchSize>**).

2. *Files*:

```
<Files>
  <Input name="referenceInput.xml"
    type="input">referenceInput.xml</Input>
  <Input name="LSintegral.csv"
    type="">LSintegral.csv</Input>
</Files>
```

Since the driven code uses a single input file, in this Section the original input is placed. As detailed in the user manual the attribute **name** represents the alias that is going to be used in all the other input blocks in order to refer to this file.

In addition the output file used in **<Sequence>** *computeLSintegral* is here inputted.

3. *Models*:

```

<Models>
  <Code name="testModel" subType="GenericCode">
    <executable>
      ../physicalCode/analyticalbateman/AnalyticalDplMain.py
    </executable>
    <clargs arg="python" type="prepend"/>
    <clargs arg="" extension=".xml" type="input"/>
    <clargs arg="" extension=".csv" type="output"/>
    <prepend>python</prepend>
  </Code>
  <ROM name="AccelerationROM" subType="SciKitLearn">
    <Features>sigma-A,decay-A</Features>
    <Target>goalFunction</Target>
    <SKLtype>neighbors|KNeighborsClassifier</SKLtype>
    <n_neighbors>1</n_neighbors>
  </ROM>
  <PostProcessor name="integralLS"
    subType="LimitSurfaceIntegral">
    <tolerance>0.001</tolerance>
    <integralType>MonteCarlo</integralType>
    <seed>20021986</seed>
    <target>goalFunction</target>
    <variable name="sigma-A">
      <distribution>sigmaA</distribution>
    </variable>
    <variable name="decay-A">
      <distribution>decayConstantA</distribution>
    </variable>
  </PostProcessor>
</Models>

```

As mentioned above, the goal of this example is the employment of an efficient sampling strategy, having as goal the determination of the failure of a system.

In addition to the previously explained Code model, the ROM of type *SciKitLearn* is here specified. The ROM will be used in the adaptive sampling strategy *LimitSurfaceSearch* in order to accelerate the convergence of the method. As it can be seen, a nearest neighbor classifier is used, targeting only two uncertainties $\sigma - A$ and $\text{decay} - A$.

For the computation of the probability of failure (see the following), a Post-Processor (PP) of type *LimitSurfaceIntegral* is here specified. This PP performs an integral of the LS generated by the adaptive sampling technique.

4. Distributions:


```

<Distributions>
  <Uniform name="sigmaA">
    <lowerBound>0</lowerBound>
    <upperBound>1000</upperBound>
  </Uniform>
  <Uniform name="decayConstantA">
    <lowerBound>0.00000001</lowerBound>
    <upperBound>0.0000001</upperBound>
  </Uniform>
</Distributions>

```

In the Distributions XML Section, the stochastic model for the uncertainties treated by the LS search sampling are reported. In this case two distributions are defined:

- $\sigma A \sim \mathbb{U}(0, 1000)$, used to model the uncertainty associated with the Model *sigma-A*
- $\text{decayConstant}A \sim \mathbb{U}(1e - 8, 1e - 7)$, used to model the uncertainty associated with the Model *decay-A*.

5. Samplers:

```

<Samplers>
  <LimitSurfaceSearch name="LSsearchSampler">
    <ROM class="Models"
      type="ROM">AccelerationROM</ROM>
    <Function class="Functions"
      type="External">goalFunction</Function>
    <TargetEvaluation class="DataObjects"
      type="PointSet">samples</TargetEvaluation>
    <Convergence forceIteration="False" limit="50000"
      persistence="20" weight="CDF">0.00001</Convergence>
    <variable name="sigma-A">
      <distribution>sigmaA</distribution>
    </variable>
    <variable name="decay-A">
      <distribution>decayConstantA</distribution>
    </variable>
  </LimitSurfaceSearch>
</Samplers>

```

In order to employ the LS search sampling strategy, a **<LimitSurfaceSearch>** node needs to be inputted. As it can be seen from above, each variable is associated to a different distribution defined in the **<Distributions>** block. In addition, the *AccelerationROM*

<ROM> is inputted. As already mentioned, this ROM (of type classifier) is used to accelerate the convergence of the LS Search method. In addition, the goal function *goalFunction* and the *samples* are here reported.

For this example, a convergence criterion of $1.0e - 5$ is set. To reach such a confidence with a Monte-Carlo, millions of samples would be needed.

6. Functions:

```
<Functions>
  <External file="goalFunction" name="goalFunction">
    <variable>A</variable>
  </External>
</Functions>
```

As already mentioned, the LS search sampling strategy uses a goal function in order to identify the regions of the uncertain space that are more informative. The *goalFunction* used for this example is reported below. As it can be seen, if the final response *A* is \leq of 0.3 , the system is considered to be in a “safe” condition.

```
def __residuumSign(self):
    returnValue = 1.0
    if self.A <= 0.3:
        returnValue = -1.0
    return returnValue
```

7. DataObjects:

```
<DataObjects>
  <PointSet name="limitSurface">
    <Input>sigma-A, decay-A</Input>
    <Output>goalFunction</Output>
  </PointSet>
  <PointSet name="samples">
    <Input>sigma-A, decay-A</Input>
    <Output>A, B, C, D, time</Output>
  </PointSet>
  <HistorySet name="histories">
    <Input>sigma-A, decay-A</Input>
    <Output>A, B, C, D, time</Output>
  </HistorySet>
</DataObjects>
```

In this block, three *DataObjects* are defined: 1) PointSet named “samples” used to collect the final outcomes of the code, 2) HistorySet named “histories” in which the full time responses

of the variables A, B, C, D are going to be stored, 3) PointSet named “limitSurface” used to export the LS location (in the uncertain space) during the employment of the sampling strategy.

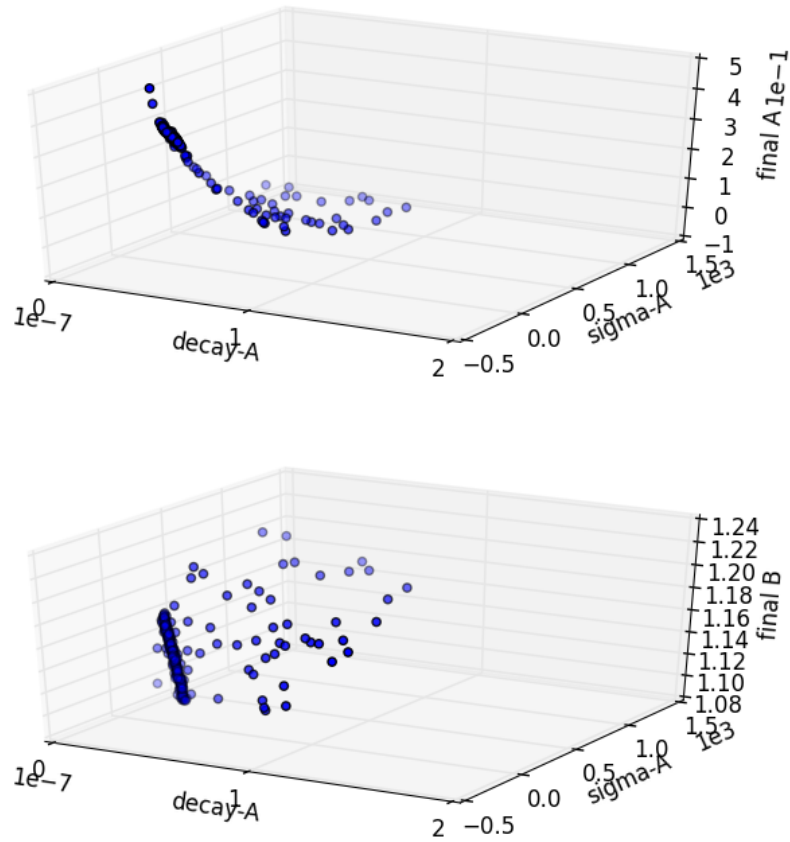


Figure 27. Plot of the samples generated by the LS search sampling for variables A, B .

8. Steps:

```
<Steps>
  <MultiRun name="sample">
    <Input class="Files"
      type="input">referenceInput.xml</Input>
```

```

<Model          class="Models"
  type="Code">testModel</Model>
<Sampler        class="Samplers"
  type="LimitSurfaceSearch">LSsearchSampler</Sampler>
<SolutionExport class="DataObjects"
  type="PointSet">limitSurface</SolutionExport>
<Output         class="DataObjects"
  type="PointSet">samples</Output>
<Output         class="DataObjects"
  type="HistorySet">histories</Output>
</MultiRun>
<PostProcess name="computeLSintegral">
  <Input  class="DataObjects" type="PointSet"
    >limitSurface</Input>
  <Model  class="Models"
    type="PostProcessor">integralLS</Model>
  <Output class="DataObjects" type="PointSet"
    >limitSurface</Output>
  <Output class="Files"      type=""
    >LSintegral.csv</Output>
</PostProcess>
<IOStep name="writeHistories" pauseAtEnd="True">
  <Input  class="DataObjects"
    type="HistorySet">histories</Input>
  <Input  class="DataObjects"
    type="PointSet">samples</Input>
  <Input  class="DataObjects"
    type="PointSet">limitSurface</Input>
  <Output class="OutStreams"
    type="Plot">samplesPlot3D</Output>
  <Output class="OutStreams"
    type="Plot">historyPlot</Output>
  <Output class="OutStreams"
    type="Print">samples</Output>
  <Output class="OutStreams"
    type="Plot">limitSurfacePlot</Output>
  <Output class="OutStreams"
    type="Print">histories</Output>
</IOStep>
</Steps>

```

Finally, all the previously defined **Entities** can be combined in the `<Steps>` block. As inferable, three `<Steps>` have been inputted:

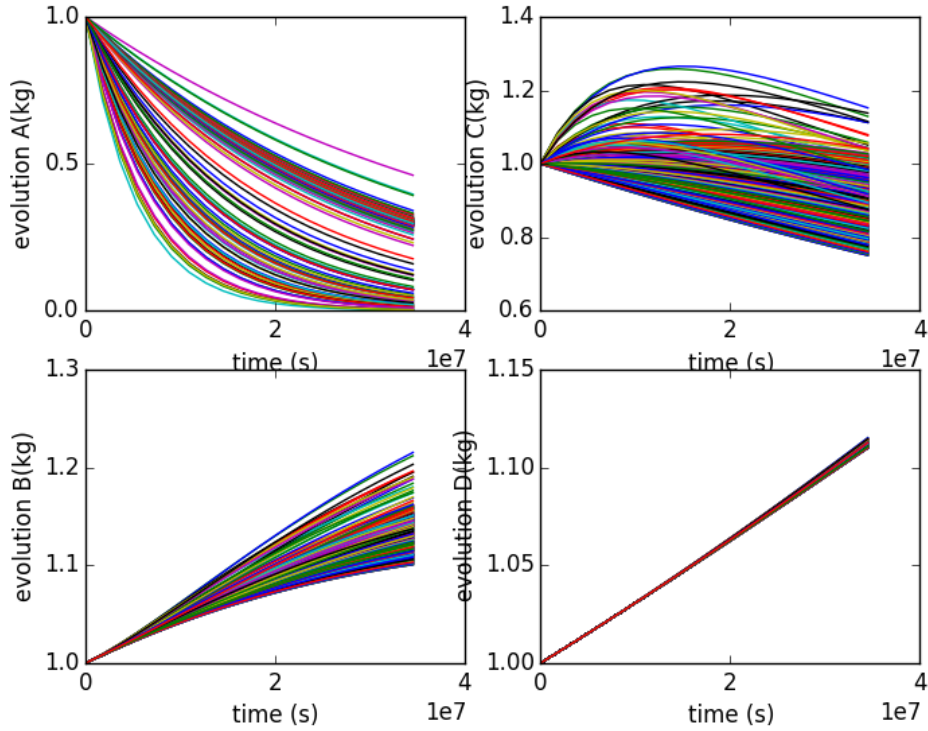


Figure 28. Plot of the histories generated by the LS search method for variables A, B, C, D .

- **<MultiRun>** named “sample”, used to run the multiple instances of the driven code and collect the outputs in the two *DataObjects*. As it can be seen, the **<Sampler>** is inputted to communicate to the *Step* that the driven code needs to be perturbed through the LS search sampling strategy;
- **<PostProcess>** named “computeLSIntegral”, used to compute the probability of failure of the system based on the LS generated employing the LS search strategy. This probability is computed integrating the LS with a Monte-Carlo method.
- **<IOStep>** named “writeHistories”, used to 1) export the “histories” and “samples” *DataObjects* **Entity** in a CSV file and 2) plot the data and the Limit Surface in PNG files and on the screen.

Figure 28 shows the evolution of the outputs A, B, C, D under uncertainties. Figure 27 shows the final responses of A and B of the sampling employed using the driven code. Figure 29 shows the limit surface for this particular example. Only 367 samples were needed in order to reach the full convergence.

The integration of the LS determines a probability of failure of $3.45e - 2$.

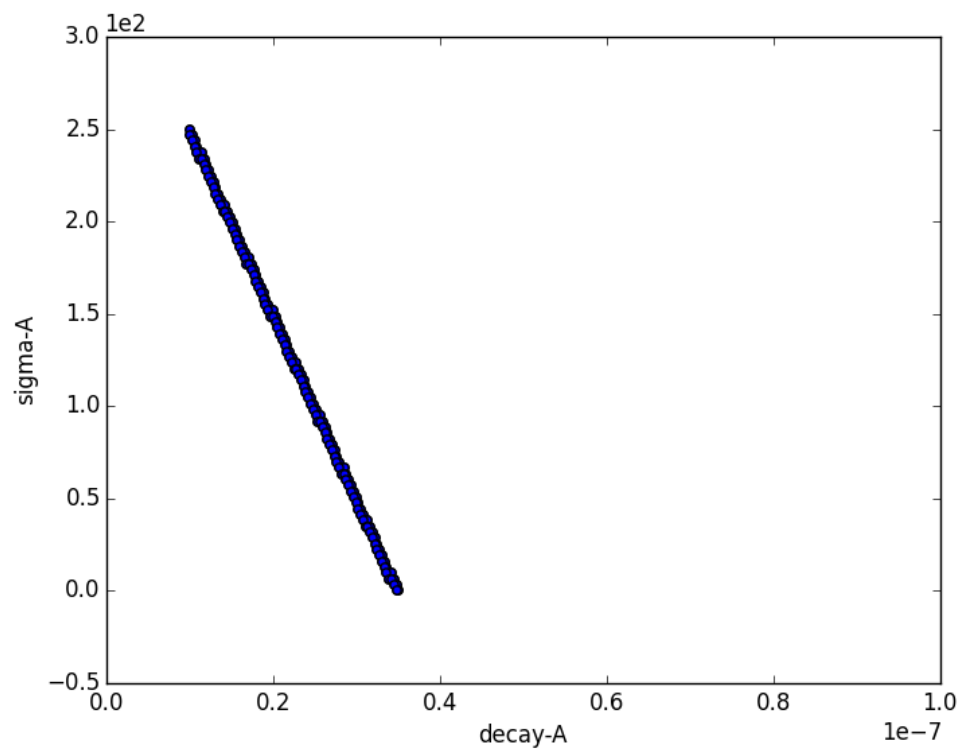


Figure 29. Limit Surface generated by the LS search method.

9 Sampling from Restart

In some instances, there are existing solutions stored that are useful to a new sampling calculation. For example, if a Monte Carlo run collects 1000 runs, then later the user decides to expand to 1500 runs, the original 1000 should not be wasted. In this case, it is desirable to restart sampling.

All **<Sampler>** entities in RAVEN accept the **<Restart>** node, which allows the user to provide a **<DataObject>** from which sampling can draw. The way each sampler interacts with this restart data is dependent on the sampling strategy.

Random sampling strategies, such as the **<MonteCarlo>** and **<Stratified>** samplers, increment the random number generator by the number of samples in the restart data, then continue sampling as normal.

Grid-based sampling strategies, such as **<Grid>**, **<SparseGridCollocation>**, and **<Sobol>**, require specific sampling points. As each required point in the input space is determined, the sampler will check the restart data for a match. If a match is found, the corresponding output values are used instead of sampling the **<Model>** for that point in the input space. In order to determine a match, all of the values in the restart point must be within a relative tolerance of the corresponding point required by the sampler. While RAVEN has a default tolerance of 1e-15, the user can adjust this tolerance using the **<restartNode>** node in the **<Sampler>** block.

In order to demonstrate this restart method, we include here an example of restarting a **<Grid>** sampler. This example runs a simple example Python code from the command line using the **<GenericCode>** interface. Within the run the following steps occur:

1. A grid is sampled that includes only the endpoints in each dimension.
2. The results of the first sampling are written to file.
3. The results in the CSV are read back in to a new **<DataObject>** called 'restart'.
4. A second, more dense grid is sampled that requires the points of the first sampling, plus several more. The results are added both to the original **<DataObject>** as well as a new one, for demonstration purposes.
5. The results of only the new sampling can be written to CSV because we added the second data object in the last step.
6. Lastly, the complete **<DataObject>** is written to file, including both the original and more dense sampling.

By looking at the contents of `GRIDdump1.csv`, `GRIDdump2.csv`, and `GRIDdump3.csv`, the progressive construction of the data object becomes clear. `GRIDdump1.csv` contains only

a few samples corresponding to the endpoints of the distributions. GRIDdump3.csv contains all the points necessary to include the midpoints of the distributions as well as the endpoints. GRIDdump2.csv contains only those points that were not already obtained in the first sampling, but still needed for the more dense sampling.

raven/tests/framework/Samplers/Restart/Truncated/grid.xml

```
<?xml version="1.0" ?>
<Simulation verbosity="silent">
  <RunInfo>
    <WorkingDir>grid</WorkingDir>
    <Sequence>make1,print1,load,make2,print2,print3</Sequence>
    <batchSize>1</batchSize>
  </RunInfo>

  <Files>
    <Input name="inp" type="">input_truncated.i</Input>
    <Input name="csv" type="">dump1.csv</Input>
  </Files>

  <Steps>
    <MultiRun name="make1">
      <Input class="Files" type="Input">inp</Input>
      <Model class="Models" type="Code">code</Model>
      <Sampler class="Samplers" type="Grid">1</Sampler>
      <Output class="DataObjects" type="PointSet">solns</Output>
    </MultiRun>
    <MultiRun name="make2">
      <Input class="Files" type="Input">inp</Input>
      <Model class="Models" type="Code">code</Model>
      <Sampler class="Samplers" type="Grid">2</Sampler>
      <Output class="DataObjects" type="PointSet">solns</Output>
      <Output class="DataObjects" type="PointSet">solns2</Output>
    </MultiRun>
    <IOStep name="print1">
      <Input class="DataObjects" type="PointSet">solns</Input>
      <Output class="OutStreams" type="Print">dump1</Output>
    </IOStep>
    <IOStep name="load">
      <Input class="Files" type="">csv</Input>
      <Output class="DataObjects" type="PointSet">restart</Output>
    </IOStep>
    <IOStep name="print2">
      <Input class="DataObjects" type="PointSet">solns2</Input>
      <Output class="OutStreams" type="Print">dump2</Output>
    </IOStep>
    <IOStep name="print3">
      <Input class="DataObjects" type="PointSet">solns</Input>
      <Output class="OutStreams" type="Print">dump3</Output>
    </IOStep>
  </Steps>
</Simulation>
```



```

</Steps>

<Distributions>
  <Uniform name="u1">
    <lowerBound>1.123456789012345</lowerBound>
    <upperBound>2</upperBound>
  </Uniform>
  <Uniform name="u2">
    <lowerBound>2.123456789012345</lowerBound>
    <upperBound>3</upperBound>
  </Uniform>
</Distributions>

<Samplers>
  <Grid name="1">
    <variable name="x">
      <distribution>u1</distribution>
      <grid construction="equal" steps="1" type="CDF">0.0 1.0</grid>
    </variable>
    <variable name="y">
      <distribution>u2</distribution>
      <grid construction="equal" steps="1" type="CDF">0.0 1.0</grid>
    </variable>
  </Grid>
  <Grid name="2">
    <variable name="x">
      <distribution>u1</distribution>
      <grid construction="equal" steps="2" type="CDF">0.0 1.0</grid>
    </variable>
    <variable name="y">
      <distribution>u2</distribution>
      <grid construction="equal" steps="2" type="CDF">0.0 1.0</grid>
    </variable>
    <Restart class="DataObjects" type="PointSet">restart</Restart>
    <restartTolerance>1e-3</restartTolerance>
  </Grid>
</Samplers>

<Models>
  <Code name="code" subType="GenericCode">
    <executable>../../../../AnalyticModels/AnalyticCodes/truncated_output.py</executable>
    <clargs arg="python" type="prepend"/>
    <clargs arg="-i" extension=".i" type="input"/>
    <clargs arg="-o" type="output"/>
    <prepend>python</prepend>
  </Code>
</Models>

<DataObjects>
  <PointSet name="dummyIN">
    <Input>x, y</Input>
  </PointSet>
</DataObjects>

```

```

    <Output>OutputPlaceholder</Output>
  </PointSet>
  <PointSet name="solns">
    <Input>x,y</Input>
    <Output>a,b</Output>
  </PointSet>
  <PointSet name="restart">
    <Input>x,y</Input>
    <Output>a,b</Output>
  </PointSet>
  <PointSet name="solns2">
    <Input>x,y</Input>
    <Output>a,b</Output>
  </PointSet>
</DataObjects>

<OutStreams>
  <Print name="dump1">
    <type>csv</type>
    <source>solns</source>
  </Print>
  <Print name="dump2">
    <type>csv</type>
    <source>solns2</source>
  </Print>
  <Print name="dump3">
    <type>csv</type>
    <source>solns</source>
  </Print>
</OutStreams>

</Simulation>

```

10 Reduced Order Modeling

The development of high-fidelity codes, for thermal-hydraulic systems and integrated multi-physics, has undergone a significant acceleration in the last years. Multi-physics codes simulate multiple physical models or multiple simultaneous physical phenomena, in a integrated solving environment. Multi-physics typically solves coupled systems of partial differential equations, generally characterized by several different geometrical and time scales.

The new multi-physics codes are characterized by remarkable improvements in the approximation of physics (high approximation order and reduced use of empirical correlations). This greater fidelity is generally accompanied by a greater computational effort (calculation time increased). This peculiarity is an obstacle in the application of computational techniques of quantification of uncertainty and risk associated with the operation of particular industrial plant (e.g., a nuclear reactor).

A solution to this problem is represented by the usage of highly effective sampling strategies. Sometimes also these approaches is not enough in order to perform a comprehensive UQ and PRA analysis. In these cases the help of reduced order modeling is essential.

RAVEN has support of several different ROMs, such as:

1. *Nearest Neighbors approaches*
2. *Support Vector Machines*
3. *Inverse Weight regressors*
4. *Spline regressors* , etc.

In this section only few of them are going to be analyzed, explaining the theory behind it by way of applied RAVEN examples.

A ROM, also known a surrogate model, is a mathematical representation of a system, used to predict a FOM of a physical system.

The “training” is a process of setting the internal parameters of the ROM from a set of samples generated the physical model, .e., the high-fidelity simulator (RELAP-7, RELAP5 3D, PHISICS, etc.),

Two characteristics of these models are generally assumed (even if exceptions are possible):

1. The higher the number of realizations in the training sets, the higher is the accuracy of the prediction performed by the ROM is. This statement is true for most of the cases, although

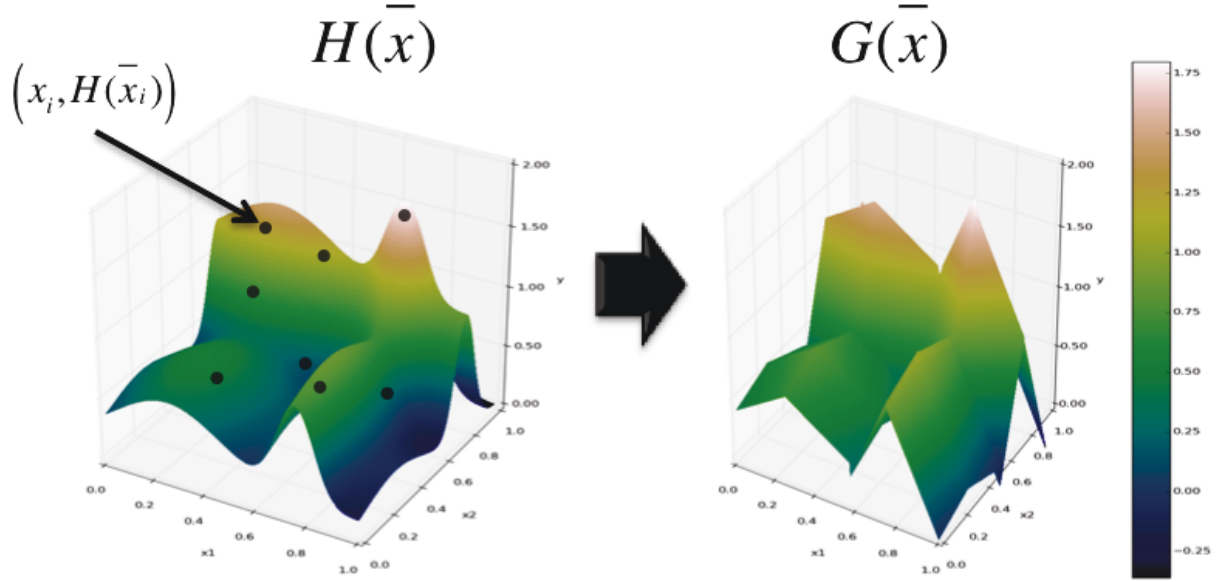


Figure 30. Example of reduced order model representation of physical system (regression).

some ROMs might be subject to the over-fitting issues. The over-fitting phenomenon is not analyzed in this thesis, since its occurrence highly depends on the algorithm type, and, hence, the problem needs to be analyzed for all the large number of ROM types available;

2. The smaller the size of the input (uncertain) domain with respect to the variability of the system response, the more likely the ROM is able to represent the system response space.

10.1 Reduced Order Modeling: Theory

To provide a very simple idea of a ROM, assume that the final response space of a physical system is governed by the transfer function $H(\bar{x})$ (see Section 3.1), which, from a practical point of view, represents the outcome of the system based on the initial conditions \bar{x} . Now, sample the domain of variability of the initial conditions \bar{x} to create a set of N realizations of the input and response space $((\bar{x}_i, H(\bar{x}_i)), i = 1, N)$, named “training” set. Based on the data set generated, it is possible to construct a mathematical representation $G(\bar{x} : \bar{x}_i)$ of the real system $H(\bar{x})$, which will approximate its response (see Figure 30):

$$G(\bar{x}) : \bar{x}_i \rightarrow G(\bar{x}_i) \cong H(\bar{x}_i) \quad (112)$$

The ROMs reported above are generally named “regressors”, among which all the most common data fitting algorithms are found (e.g., least square for construction of linear models).

An important class of ROMs for the work presented here after is the one containing the so called “classifiers”. A classifier is a ROM that is capable of representing the system behavior from a binary point of view (e.g., event happened/not happened or failure/success). It is a model (set of equations) that identifies to which category an object belongs in the feature (input) space. Referring to the example that brought to Equation 112, a classifier can be formally represented as follows (see Figure 31):

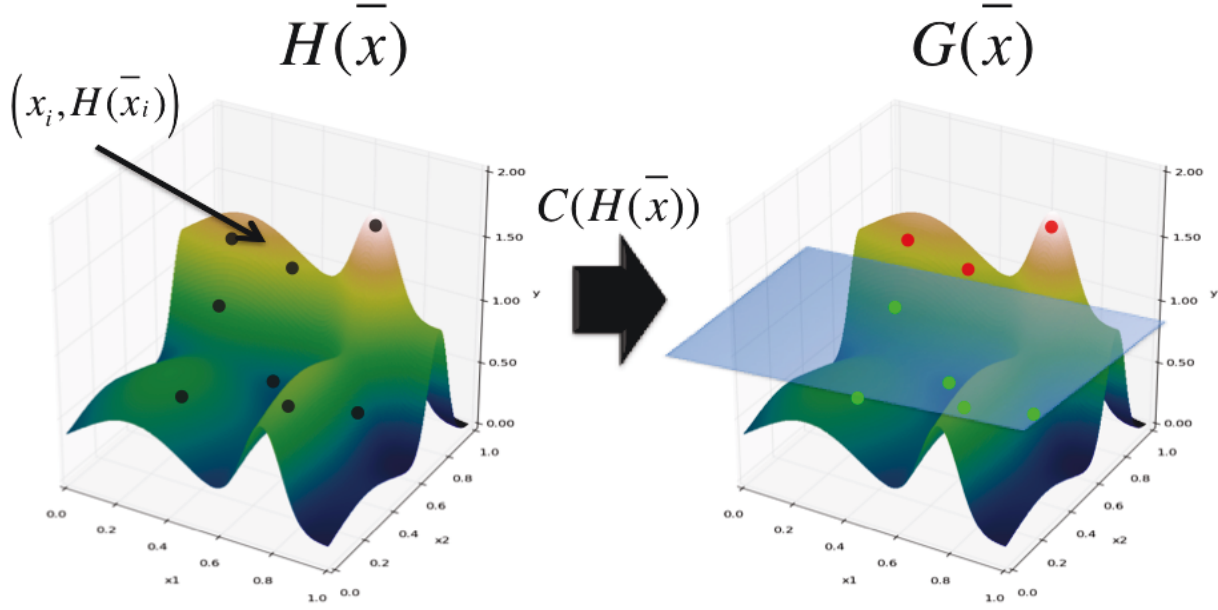


Figure 31. Example of reduced order model representation of physical system (classifier).

$$G(\bar{x}) : \bar{x}_i \rightarrow G(\bar{x}_i) \cong C(H(\bar{x}_i)) \quad (113)$$

The function $C(H(\bar{x}_i) = \bar{\theta})$ is the so called “goal” function that is able to recast the response of the system $H(\bar{x}_i)$ into a binary form (e.g., failure/success). As an example, referring to Figure 31, the “goal” function would be:

$$C(\bar{\theta}) = \begin{cases} 1 & \text{if } \bar{\theta} > 1.0 \\ 0 & \text{if } \bar{\theta} \leq 1.0 \end{cases} \quad (114)$$

Hence, the ROM of type classifier $G(\bar{x})$ will operate in the space transformed through the “goal” function $C(\bar{\theta})$.

The classifiers and regressors can be categorized into two main classes:

- Model-based algorithms
- Data-based algorithms

In the first class, the created ROM aims to approximate the response of the system as a function of the input parameters. These algorithms construct a functional representation of the system. Examples of such ROM type are Support Vector Machines (SVMs), Kriging-based regressors, discriminant-based models, and polynomial chaos.

On the other side, data-based algorithms do not build a response- function-based ROM but classify or predict the response of the system from the neighborhood graph constructed from the training data, without any dependencies on a particular prediction model. These algorithms directly build a neighborhood structure as the ROM (e.g., a relaxed Gabriel graph) on the initial training data. Examples of such ROM type are nearest neighbors and decision trees.

10.1.1 Gaussian Process Models

Gaussian Processes (GPs) [22] are algorithms that extend multivariate Gaussian distributions to infinite dimensionality. A Gaussian process generates a data set located throughout some domain such that any finite subset of the range follows a multivariate Gaussian distribution. Now, the n observations in an arbitrary data set, $y = y_1, \dots, y_n$, can always be imagined as a single point sampled from some multivariate (n -variate) Gaussian distribution. What relates one observation to another in such cases is just the covariance function, $k(x, x')$. A popular choice is the squared exponential:

$$k(x, x') = \sigma_f^2 \exp \left[\frac{-(x - x')^2}{2l^2} \right] \quad (115)$$

where the maximum allowable covariance is defined as σ_f^2 ; this should be high for functions that cover a broad range on the y axis. If $x \simeq x'$, then $k(x, x')$ approach this maximum meaning $f(x)$ is very correlated to $f(x')$. On the other hand, if x is very distant from x' , then $k(x, x') \simeq 0$ (i.e., the two points cannot see each other. So, for example, during interpolation at new x values, distant observations will have negligible effect). How much effect this separation has will depend on the length parameter l . Each observation y can be thought of as related to an underlying function $f(x)$ through a Gaussian noise model:

$$y = f(x) + N(0, \sigma_n^2) \quad (116)$$

The new kernel function can be written as:

$$k(x, x') = \sigma_f^2 \exp \left[\frac{-(x - x')^2}{2l^2} \right] + \sigma_n^2 \delta(x, x') \quad (117)$$

So given n observations y , the objective is to predict the value y_* at the new point x_* . This process is performed by following this sequence of steps:

1. Calculate three matrices:

$$K = \begin{bmatrix} k(x_1, x_1) & \dots & k(x_1, x_n) \\ \vdots & \ddots & \vdots \\ k(x_n, x_1) & \dots & k(x_n, x_n) \end{bmatrix} \quad (118)$$

$$K_* = [k(x_*, x_1) \quad \dots \quad k(x_*, x_n)] \quad (119)$$

$$K_{**} = k(x_*, x_*) \quad (120)$$

2. The basic assumption of GPM is that:

$$\begin{bmatrix} y \\ y_* \end{bmatrix} = \mathcal{N}(0, \begin{bmatrix} K & K_*^T \\ K_* & K_{**} \end{bmatrix}) \quad (121)$$

3. The estimate \bar{y}_* for y_* is the mean of this distribution

$$\bar{y}_* = K_* K^{-1} y \quad (122)$$

4. The uncertainty associated to the estimate \bar{y}_* can be expressed in terms of variance of y_* :

$$\text{var}(y_*) = K_{**} - k_* K^{-1} K_*^T \quad (123)$$

10.1.2 Support Vector Machines

The Support Vector Machine (SVM) [11] classifier is a methodology that aims to determine the optimal separation hyperplane between data sets having different labels. The training data consist of N data points (x_i, y_i) $i = 1, \dots, N$ where $x_i \in \mathbb{R}^M$ and $y_i \in -1, 1$. Assuming a linear property of the hyperplane then its definition is:

$$\{x : f(x) = x^T \beta + \beta_0 = 0\} \quad (124)$$

where β is a unit vector.

The SVM parameters β and β_0 are determined by solving this optimization problem:

$$\begin{cases} \min_{\beta, \beta_0} \|\beta\| \\ \text{subject to } y_i(x_i^T \beta + \beta_0) \geq 1, \quad i = 1, \dots, N \end{cases} \quad (125)$$

Once the SVM parameters β and β_0 are determined then the classification of a new point \bar{x} is given by:

$$G(\bar{x}) = \text{sign}(\bar{x}^T \beta + \beta_0) \quad (126)$$

10.1.3 KNN Classifier and KNR Regressor

The K Nearest Neighbor algorithm [23] (KNN) is a non-parametric method used for both regression and classification. The only input parameter is the variable K which indicates the number of neighbors to be considered in the classification/regression process. The special case where the class is predicted to be the class of the closest training sample (i.e. when $K = 1$) is called the nearest neighbor algorithm. In binary (two class) classification problems, it is helpful to choose k to be an odd number as this avoids tied votes. The output depends on whether KNN is used for classification or regression:

- In KNN classification, the output is a class membership. An object is classified by a majority vote of its neighbors, with the object being assigned to the class most common among its K nearest neighbors (K is a positive integer, typically small). If $K = 1$, then the object is simply assigned to the class of that single nearest neighbor.
- In KNN regression, the output is the property value for the object. This value is the average of the values of its K nearest neighbors.

Both for classification and regression, it can be useful to assign weight to the contributions of the neighbors, so that the nearer neighbors contribute more to the average than the more distant ones. For example, a common weighting scheme consists in giving each neighbor a weight of $1/d$, where d is the distance to the neighbor.

10.1.4 Multi-Dimensional Interpolation

This section covers the methods that have been implemented in the CROW statistical library:

- Shepard's Method (see Section 10.1.4.1)
- Multi-Dimensional Spline method (see Section 10.1.4.2).

These two methods are interpolation methods that can be used in any dimension. In RAVEN they are employed in two major applications:

1. ROMs
2. Multi-dimensional distributions.

For both applications, given a set of N data points (x_i, u_i) $i = 1, \dots, N$ where x_i are the coordinate in the input space $D \subset \mathbb{R}^M$ and $u_i \in \mathbb{R}$ is the outcome, the methods predicts the outcome \tilde{u} for a new coordinate $\tilde{x} \in \mathbb{R}^n$.

10.1.4.1 Shepard's Method

The Shepard interpolator [24] is also known as Inverse Distance Weighting (IDW) interpolator. The starting point is a set of N data points (x_i, u_i) for $i = 1, \dots, N$. The Inverse-Weight interpolator can be represented as a function $f_{IDW}(x)$ that, given a new coordinate in the input space x , generates a prediction on u such that

$$u : x \in \mathbb{R}^M \rightarrow f_{IDW}(x) \in \mathbb{R} \quad (127)$$

based on the distance $d(x, x_i)$ in the euclidean space between x and x_i .

Such prediction $u = f_{IDW}(x)$ is performed by summing all data points x_i $i = 1, \dots, N$ weighted by a weighting parameter $w_i(x)$ as follows:

$$f_{IDW}(x) = \begin{cases} \sum_{i=1}^N w(x_i) u_i & \text{if } d(x, x_i) \neq 0 \\ u_i & \text{if } d(x, x_i) = 0 \end{cases} \quad (128)$$

where

$$w(x_i) = \frac{w_i}{\sum_{i=1}^N w_i} \quad (129)$$

and

$$w_i = \left(\frac{1}{d(x, x_i)} \right)^p \quad (130)$$

Large values of p assign greater weight w_i to data points x_i closest to x , with the result turning into a mosaic of tiles (i.e., Voronoi diagram) with nearly constant interpolated value.

10.1.4.2 Multi-Dimensional Spline

The Multi-Dimensional Spline (MDS) [25] is a method that requires the sampled points x_i to be lying in multi-dimensional cartesian grid. A generic grid Δ_m for each dimension m will be indicated as follows:

$$\Delta_m = \{x_{0_m}, x_{1_m}, \dots, x_{p_m}\} \text{ for } m = 1, \dots, M \quad (131)$$

This methods construct a M -dimensional cubic spline so that, given a coordinate in the input space $x = (x_1, x_2, \dots, x_M)$, generates a prediction on u such that

$$u : x \in \mathbb{R}^M \rightarrow f_{MDS}(x) \in \mathbb{R} \quad (132)$$

where

$$f_{MDS}(x) = \sum_{i_1=1}^{p_1+3} \sum_{i_2=1}^{p_2+3} \dots \sum_{i_M=1}^{p_M+3} c_{i_1, i_2, \dots, i_p} \prod_{m=1}^M u_{i_j}(x_m) \quad (133)$$

where

$$u_{i_j}(x_m) = \Phi \left(\frac{x_m - x_{0_m}}{h_j} + 2 - i_j \right) \quad (134)$$

The cubic kernel $\Phi(t)$ is defined as:

$$\Phi(t) = \begin{cases} (2 - |t|)^3 & 1 \leq |t| \leq 2 \\ 4 - 6|t|^2 + 3|t|^3 & |t| \leq 1 \\ 0 & \text{elsewhere} \end{cases} \quad (135)$$

The set of $\prod_{m=1}^M (p_m + 3)$ coefficients c_{i_1, i_2, \dots, i_p} is determined when the interpolator is initialized.

10.2 Reduced Order Modeling through RAVEN

The goals of this section are about learning how to:

1. Set up a sampling strategy to construct multiple ROMs, perturbing a driven code
2. Train the different ROMs with the data-set obtained by the applied sampling strategy;
3. Use the same sampling strategy, perturbing the ROMs
4. Plot the responses of the driven code and ROMs, respectively.

In order to accomplish these tasks, the following RAVEN **Entities** (XML blocks in the input files) need to be defined:

1. *RunInfo*:

```
<RunInfo>
  <JobName>Chapter-IX/ROMConstruction</JobName>
  <Sequence>
    sample, trainROMGaussianProcess, trainROMsvm,
    trainROMinverse, sampleROMGaussianProcess,
    sampleROMInverse, sampleROMsvm, writeHistories
  </Sequence>
  <WorkingDir>ROMConstruction</WorkingDir>
  <batchSize>12</batchSize>
</RunInfo>
```

As in the other examples, the the *RunInfo* **Entity** is intended to set up the analysis sequence that needs to be performed. In this specific case, eight steps (**<Sequence>**) are going to be sequentially run using eight processors (**<batchSize>**).

In the first step, the original physical model is going to be sampled. The obtained results are going to be used to train three different ROMs. These ROMs are sampled by the same strategy used in the first step in order to compare the ROMs' responses with the ones coming from the original physical model.

2. Files:

```
<Files>
  <Input name="referenceInput.xml"
    type="input">referenceInput.xml</Input>
</Files>
```

Since the driven code uses a single input file, the original input is placed in this section. As detailed in the user manual the attribute `name` represents the alias that is going to be used in all the other input blocks in order to refer to this file.

3. Models:

```
<Models>
  <Code name="testModel" subType="GenericCode">
    <executable>
      ../physicalCode/analyticalbateman/AnalyticalDplMain.py
    </executable>
    <clargs arg="python" type="prepend"/>
    <clargs arg="" extension=".xml" type="input"/>
    <clargs arg="" extension=".csv" type="output"/>
    <prepend>python</prepend>
  </Code>
  <ROM name="ROMGaussianProcess" subType="SciKitLearn">
    <Features>sigma-A, sigma-B, decay-A, decay-B</Features>
    <Target>A, B</Target>
    <SKLtype>GaussianProcess|GaussianProcess</SKLtype>
    <regr>quadratic</regr>
    <theta0>10</theta0>
  </ROM>
  <ROM name="ROMsvm" subType="SciKitLearn">
    <Features>sigma-A, sigma-B, decay-A, decay-B</Features>
    <Target>A, B</Target>
    <SKLtype>svm|SVR</SKLtype>
    <kernel>rbf</kernel>
    <C>50.0</C>
    <tol>0.000001</tol>
  </ROM>
  <ROM name="ROMinverse" subType="NDinvDistWeight">
    <Features>sigma-A, sigma-B, decay-A, decay-B</Features>
    <Target>A, B</Target>
    <p>3</p>
  </ROM>
</Models>
```

As mentioned above, the goal of this example is the employment of a sampling strategy in order to construct multiple types of ROMs.

Indeed, in addition to the previously explained Code model, three different ROMs (GP, SVM and IDW) are here specified. The ROMs will be constructed (“trained”) through the data-set generated by the sampling of the physical model. Once trained, they are going to be used in place of the original physical model.

As it can be seen, the ROMs will be constructed considering four features ($\sigma-A$, $\sigma-B$, $\text{decay}-A$, and $\text{decay}-B$) and two targets (A and B).

4. *Distributions:*

```
<Distributions>
  <Uniform name="sigma">
    <lowerBound>0</lowerBound>
    <upperBound>1000</upperBound>
  </Uniform>
  <Uniform name="decayConstant">
    <lowerBound>0.00000001</lowerBound>
    <upperBound>0.0000001</upperBound>
  </Uniform>
</Distributions>
```

In the Distributions XML section, the stochastic model for the uncertainties are reported. In this case two distributions are defined:

- $\sigma \sim \mathbb{U}(0, 1000)$, used to model the uncertainties associated with the Model $\sigma-A$ and $\sigma-B$;
- $\text{decayConstant} \sim \mathbb{U}(1e-8, 1e-7)$, used to model the uncertainties associated with the Model $\text{decay}-A$ and $\text{decay}-B$.

5. *Samplers:*

```
<Samplers>
  <Grid name="grid">
    <variable name="sigma-A">
      <distribution>sigma</distribution>
      <grid construction="equal" steps="5" type="CDF">0.01
        0.99</grid>
    </variable>
    <variable name="decay-A">
      <distribution>decayConstant</distribution>
      <grid construction="equal" steps="5" type="CDF">0.01
        0.99</grid>
    </variable>
    <variable name="sigma-B">
```

```

        <distribution>sigma</distribution>
        <grid construction="equal" steps="5"
            type="CDF">0.01 0.99</grid>
    </variable>
    <variable name="decay-B">
        <distribution>decayConstant</distribution>
        <grid construction="equal" steps="5"
            type="CDF">0.01 0.99</grid>
    </variable>
</Grid>
</Samplers>

```

To obtain the data-set through which the ROMs are going to be constructed, a *Grid* sampling approach is here employed.

6. *DataObjects*:

```

<DataObjects>
  <PointSet name="samples">
    <Input>sigma-A, sigma-B, decay-A, decay-B</Input>
    <Output>A, B, C, D</Output>
  </PointSet>
  <HistorySet name="histories">
    <Input>sigma-A, sigma-B, decay-A, decay-B</Input>
    <Output>A, B, C, D, time</Output>
  </HistorySet>
  <PointSet name="inputPlaceholder">
    <Input>sigma-A, sigma-B, decay-A, decay-B</Input>
    <Output>OutputPlaceholder</Output>
  </PointSet>
  <PointSet name="samplesGP">
    <Input>sigma-A, sigma-B, decay-A, decay-B</Input>
    <Output>A, B</Output>
  </PointSet>
  <PointSet name="samplesInverse">
    <Input>sigma-A, sigma-B, decay-A, decay-B</Input>
    <Output>A, B</Output>
  </PointSet>
  <PointSet name="samplesSVM">
    <Input>sigma-A, sigma-B, decay-A, decay-B</Input>
    <Output>A, B</Output>
  </PointSet>
</DataObjects>

```

In this block, six *DataObjects* are defined: 1) PointSet named “samples” used to collect the final outcomes of the code, 2) HistorySet named “histories” in which the full time responses of the variables A, B, C, D are going to be stored, 3) PointSet named “inputPlaceholder” used in the role of **<Input>** for the ROMs sampling; 4) PointSet named “samplesGP” used to collect the final outcomes (sampling) of the GP ROM; 5) PointSet named “samplesInverse” used to collect the final outcomes (sampling) of the IDW ROM; 6) PointSet named “samplesSVM” used to collect the final outcomes (sampling) of the SVM ROM.

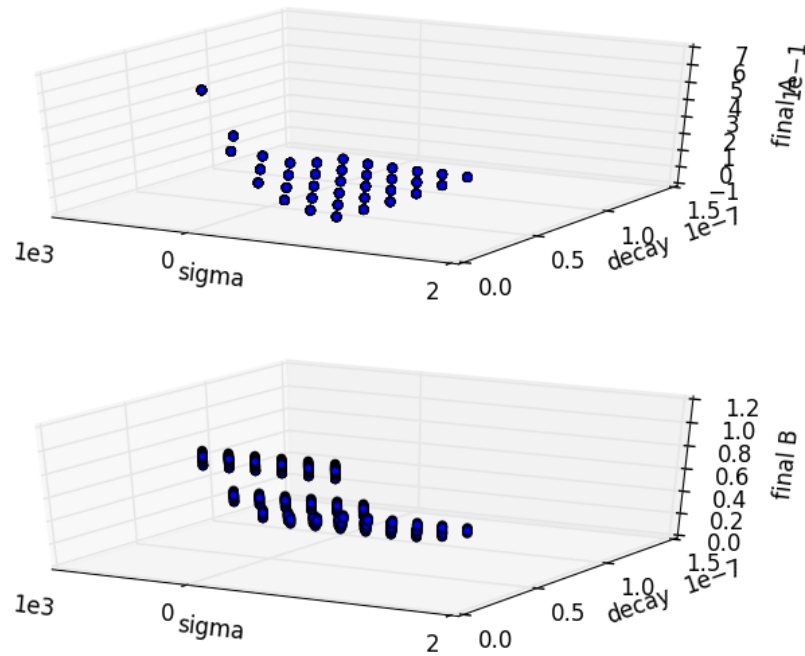


Figure 32. Plot of the samples generated by the Grid sampling for variables A, B .

7. Steps:

```
<Steps>
  <MultiRun name="sample">
    <Input class="Files"
      type="input">referenceInput.xml</Input>
    <Model class="Models" type="Code">testModel</Model>
    <Sampler class="Samplers" type="Grid">grid</Sampler>
```

```

    <Output class="DataObjects"
      type="PointSet">samples</Output>
    <Output class="DataObjects"
      type="HistorySet">histories</Output>
  </MultiRun>
  <MultiRun name="sampleROMGaussianProcess">
    <Input class="DataObjects"
      type="PointSet">inputPlaceholder</Input>
    <Model class="Models"
      type="ROM">ROMGaussianProcess</Model>
    <Sampler class="Samplers" type="Grid">grid</Sampler>
    <Output class="DataObjects"
      type="PointSet">samplesGP</Output>
  </MultiRun>
  <MultiRun name="sampleROMInverse">
    <Input class="DataObjects"
      type="PointSet">inputPlaceholder</Input>
    <Model class="Models" type="ROM">ROMInverse</Model>
    <Sampler class="Samplers" type="Grid">grid</Sampler>
    <Output class="DataObjects"
      type="PointSet">samplesInverse</Output>
  </MultiRun>
  <MultiRun name="sampleROMsvm">
    <Input class="DataObjects"
      type="PointSet">inputPlaceholder</Input>
    <Model class="Models" type="ROM">ROMsvm</Model>
    <Sampler class="Samplers" type="Grid">grid</Sampler>
    <Output class="DataObjects"
      type="PointSet">samplesSVM</Output>
  </MultiRun>
  <RomTrainer name="trainROMGaussianProcess">
    <Input class="DataObjects"
      type="PointSet">samples</Input>
    <Output class="Models"
      type="ROM">ROMGaussianProcess</Output>
  </RomTrainer>
  <RomTrainer name="trainROMsvm">
    <Input class="DataObjects"
      type="PointSet">samples</Input>
    <Output class="Models" type="ROM">ROMsvm</Output>
  </RomTrainer>
  <RomTrainer name="trainROMInverse">

```

```

    <Input class="DataObjects"
      type="PointSet">samples</Input>
    <Output class="Models" type="ROM">ROMinverse</Output>
  </RomTrainer>
  <IOStep name="writeHistories" pauseAtEnd="True">
    <Input class="DataObjects"
      type="HistorySet">histories</Input>
    <Input class="DataObjects"
      type="PointSet">samples</Input>
    <Input class="DataObjects"
      type="PointSet">samplesGP</Input>
    <Input class="DataObjects"
      type="PointSet">samplesInverse</Input>
    <Input class="DataObjects"
      type="PointSet">samplesSVM</Input>
    <Output class="OutStreams"
      type="Plot">samplesPlot3D</Output>
    <Output class="OutStreams"
      type="Plot">samplesPlot3DROMgp</Output>
    <Output class="OutStreams"
      type="Plot">samplesPlot3DROMsvm</Output>
    <Output class="OutStreams"
      type="Plot">samplesPlot3DROMinverse</Output>
    <Output class="OutStreams"
      type="Plot">historyPlot</Output>
    <Output class="OutStreams"
      type="Print">samples</Output>
    <Output class="OutStreams"
      type="Print">histories</Output>
  </IOStep>
</Steps>

```

Finally, all the previously defined **Entities** can be combined in the **<Steps>** block. As inferable, eight **<Steps>** have been inputted:

- **<MultiRun>** named “sample”, used to run the multiple instances of the driven code and collect the outputs in the two *DataObjects*. As it can be seen, the **<Sampler>** is inputted to communicate to the *Step* that the driven code needs to be perturbed through the Grid sampling strategy;
- **<RomTrainer>** named “trainROMGaussianProcess”, used to construct (“train”) the GP ROM, based on the data-set generated in the “sample” **Step**;
- **<RomTrainer>** named “trainROMsvm”, used to construct (“train”) the SVM ROM, based on the data-set generated in the “sample” **Step**;

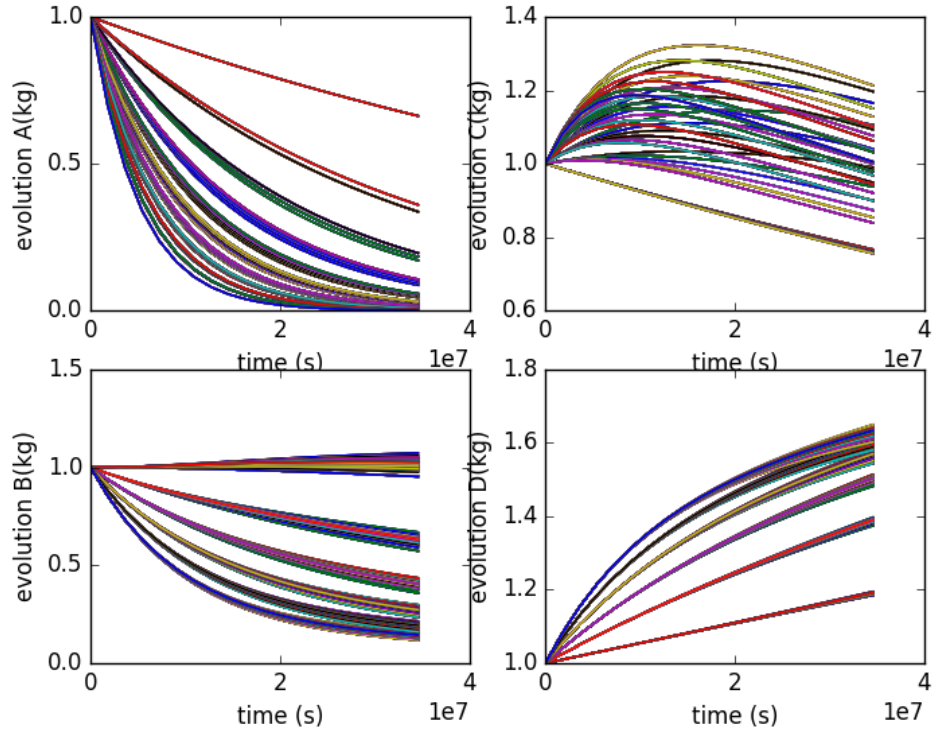


Figure 33. Plot of the histories generated by the Grid method for variables A, B, C, D .

- **<RomTrainer>** named “trainROMinverse”, used to construct (“train”) the IDW ROM, based on the data-set generated in the “sample” Step;
- **<MultiRun>** named “sampleROMGaussianProcess”, used to run the multiple instances of the previously constructed GP ROM and collect the outputs in the PointSet *DataObject*. As it can be seen, the same **<Sampler>** used for perturbing the original model is here used.
- **<MultiRun>** named “sampleROMsvm”, used to run the multiple instances of the previously constructed Support Vector Machine ROM and collect the outputs in the PointSet *DataObject*. As it can be seen, the same **<Sampler>** used for perturbing the original model is here used.
- **<MultiRun>** named “sampleROMInverse”, used to run the multiple instances of the previously constructed Inverse Distance Weight ROM and collect the outputs in the PointSet *DataObject*. As it can be seen, the same **<Sampler>** used for perturbing the original model is here used.
- **<IOStep>** named “writeHistories”, used to 1) export the “histories” and “samples”

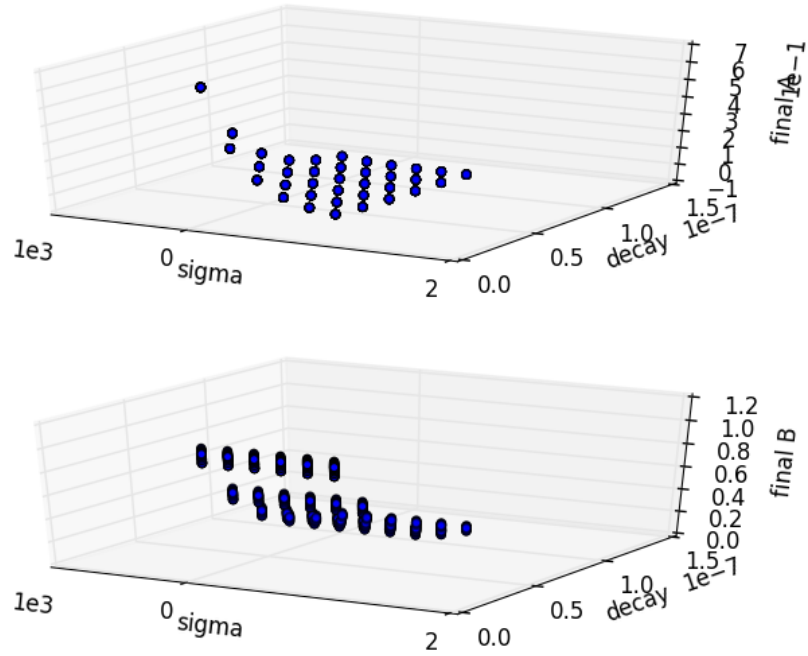


Figure 34. Plot of the samples generated by the Grid sampling applied on the Gaussian Process ROM for variables A, B

DataObjects **Entity** in a CSV file and 2) plot the responses of the sampling performed on the physical model, GP ROM, SVM ROM and IDW ROM in PNG files and on the screen.

Figure 33 shows the evolution of the outputs A, B, C, D under uncertainties. Figure 32 shows the final responses of A and B of the sampling employed using the driven code.

Figures 34, 35 and 36 show the final responses of A and B of the sampling employed using the Gaussian Process, Support Vector Machines and Inverse Distance Weight ROMs, respectively. It can be clearly noticed that the responses of the ROMs perfectly match the outcomes coming from the original model (see Figure 32).

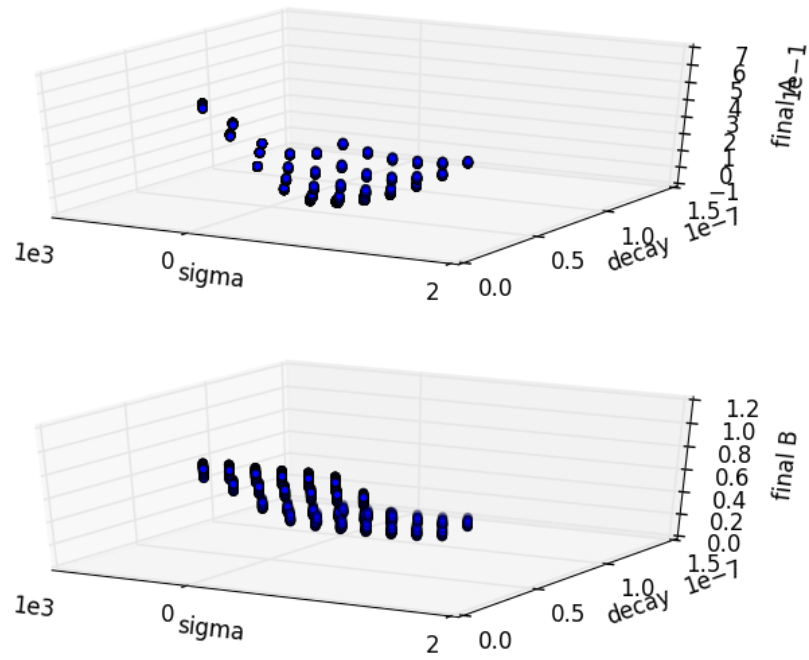


Figure 35. Plot of the samples generated by the Grid sampling applied on the Support Vector Machine ROM for variables A , B

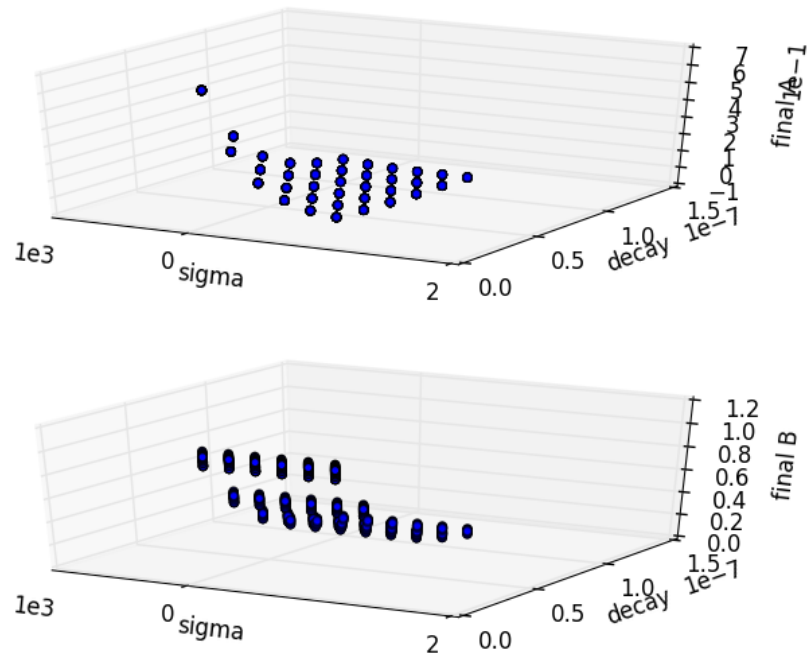


Figure 36. Plot of the samples generated by the Grid sampling applied on the Inverse Distance Weight ROM for variables A , B

11 Statistical Analysis

In order to perform a complete analysis of a system under uncertainties, it is crucial to be able to compute all the statistical moments of one or even multiple FOMs. In addition, it is essential to identify the correlation among different FOMs toward a specific input space.

RAVEN is able to compute the most important statistical moments: such as:

1. *Expected Value*
2. *Standard Deviation*
3. *Variance*
4. *variationCoefficient*
5. *Skewness*
6. *Kurtosis*
7. *Median*
8. *Percentile.*

In addition, RAVEN fully supports the computation of all of the statistical moments defined to “measure” the correlation among variables/parameters/FOMs:

1. *Covariance matrix*
2. *Normalized Sensitivity matrix*
3. *Variance Dependent Sensitivity matrix*
4. *Sensitivity matrix*
5. *Pearson matrix.*

In this section these features are analyzed, by showing the theory behind it using a set of RAVEN examples.

11.1 Statistical Analysis Theory

One of the most assessed ways to investigate the impact of the intrinsic variation of the input space is through the computation of statistical moments and linear correlation among variables/parameters/FOMs.

As shown in Section 7, RAVEN employs several different sampling methodologies to explore the response of a model subject to uncertainties. In order to correctly compute the statistical moments a weight-based approach is used. Each *Sampler* in RAVEN associate to each “sample” (i.e. realization in the input/uncertain space) a **weight** to represent the *importance* of the particular combination of input values from a statistical point of view (e.g., reliability weights). These weights are used in subsequential steps in order to compute the previously listed statistical moments and correlation metrics.

In the following subsections, the formulation of these statistical moments is reported.

11.1.1 Expected Value

The expected value represents one of the most fundamental metrics in probability theory: it represents a measurement of the center of the distribution (mean) of the random variable. From a practical point of view, the expected value of a discrete random variable is the probability-weighted average of all possible values of the subjected variable. Formally, the expected value of a random variable X :

$$\begin{aligned}\mathbb{E}(X) &= \mu = \sum_{x \in \mathcal{X}} x \text{pdf}_X(x) && \text{if } X \text{ discrete} \\ \mathbb{E}(X) &= \mu = \int_{x \in \mathcal{X}} x \text{pdf}_X(x) && \text{if } X \text{ continuous}\end{aligned}\tag{136}$$

In RAVEN, the expected value (i.e. first central moment) is computed as follows:

$$\begin{aligned}\mathbb{E}(X) &= \mu \approx \bar{x} = \frac{1}{n} \sum_{i=1}^n x_i && \text{if random sampling} \\ \mathbb{E}(X) &= \mu \approx \bar{x} = \frac{1}{V_1} \sum_{i=1}^n w_i x_i && \text{otherwise}\end{aligned}\tag{137}$$

where:

- w_i is the weight associated with the sample i
- n are the total number of samples
- $V_1 = \sum_{i=1}^n w_i$.

11.1.2 Standard Deviation and Variance

The variance (σ^2) and standard deviation (σ) of X are both measures of the spread of the distribution of the random variable about the mean. Simplistically, the variance measures how far a set of realizations of a random variable are spread out. The standard deviation is the square root of the variance. The standard deviation has the same unit of the original data, and hence is comparable to deviations from the mean.

Formally:

$$\begin{aligned}\sigma^2(X) &= \mathbb{E}([X - \mathbb{E}(X)]^2) = \int_{x \in \mathcal{X}} (x - \mu)^2 pdf(x) dx && \text{if } X \text{ continuous} \\ \sigma^2(X) &= \mathbb{E}([X - \mathbb{E}(X)]^2) = \sum_{x \in \mathcal{X}} (x - \mu)^2 pdf(x) && \text{if } X \text{ discrete} \\ \sigma(X) &= \mathbb{E}([X - \mathbb{E}(X)]) = \sqrt{\sigma^2(X)}\end{aligned}\tag{138}$$

In RAVEN, variance (i.e., second central moment) and standard deviation are computed as follows:

$$\begin{aligned}\mathbb{E}([X - \mathbb{E}(X)]^2) &\approx m_2 = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2 && \text{if random sampling} \\ \mathbb{E}([X - \mathbb{E}(X)]^2) &\approx m_2 = \frac{1}{V_1} \sum_{i=1}^n w_i (x_i - \bar{x})^2 && \text{otherwise} \\ \mathbb{E}([X - \mathbb{E}(X)]^2) &\approx s = \sqrt{m_2}\end{aligned}\tag{139}$$

where:

- w_i is the weight associated with the sample i
- n are the total number of samples
- $V_1 = \sum_{i=1}^n w_i$.

RAVEN performs an additional correction of variance to obtain an unbiased estimation with respect to the sample-size [26]:

$$\begin{aligned}\mathbb{E}([X - \mathbb{E}(X)]^2) &\approx M_2 = \frac{n}{n-1} m_2 && \text{if random sampling} \\ \mathbb{E}([X - \mathbb{E}(X)]^2) &\approx M_2 = \frac{V_1^2}{V_1^2 - V_2} m_2 && \text{otherwise}\end{aligned}\tag{140}$$

$$S = \sqrt{M_2}\tag{141}$$

where:

- w_i is the weight associated with the sample i

- n are the total number of samples
- $V_1 = \sum_{i=1}^n w_i^2$.

It is important to notice that S is not an unbiased estimator.

11.1.3 Skewness

The Skewness is a measure of the asymmetry of the distribution of a real-valued random variable about its mean. Negative skewness indicates that the tail on the left side of the distribution is longer or fatter than the right side. Positive skewness indicates that the tail on the right side is longer or fatter than the left side. From a practical point of view, the skewness is useful to identify distortion of the random variable with respect to the Normal distribution function.

Formally,

$$\gamma_1 = \mathbb{E} \left[\left(\frac{X - \mu}{\sigma} \right)^3 \right] = \frac{\mathbb{E} [(X - \mu)^3]}{(\mathbb{E} [(X - \mu)^2])^{3/2}} \quad (142)$$

In RAVEN, the skewness is computed as follows:

$$\begin{aligned} \mathbb{E} \left[\left(\frac{X - \mu}{\sigma} \right)^3 \right] &\approx \frac{m_3}{m_2^{3/2}} = \frac{\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^3}{\left(\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2 \right)^{3/2}} && \text{if random sampling} \\ \mathbb{E} \left[\left(\frac{X - \mu}{\sigma} \right)^3 \right] &\approx \frac{m_3}{m_2^{3/2}} = \frac{\frac{1}{V_1} \sum_{i=1}^n w_i \times (x_i - \bar{x})^3}{\left(\frac{1}{V_1} \sum_{i=1}^n w_i \times (x_i - \bar{x})^2 \right)^{3/2}} && \text{otherwise} \end{aligned} \quad (143)$$

where:

- w_i is the weight associated with the sample i
- n are the total number of samples
- $V_1 = \sum_{i=1}^n w_i$.

RAVEN performs an additional correction of skewness to obtain an unbiased estimation with respect to the sample-size [26]:

$$\begin{aligned} \mathbb{E} \left[\left(\frac{X - \mu}{\sigma} \right)^3 \right] &\approx \frac{M_3}{M_2^{3/2}} = \frac{n^2}{(n-1)(n-2)} m_3 \times \frac{1}{\left(\frac{n}{n-1} m_2 \right)^{3/2}} && \text{if random sampling} \\ \mathbb{E} \left[\left(\frac{X - \mu}{\sigma} \right)^3 \right] &\approx \frac{M_3}{M_2^{3/2}} = \frac{V_1^3}{V_1^3 - 3V_1V_2 + 2V_3} m_3 \times \frac{1}{\left(\frac{V_1^2}{V_1^2 - V_2} m_2 \right)^{3/2}} && \text{otherwise} \end{aligned} \quad (144)$$

where:

- w_i is the weight associated with the sample i
- n are the total number of samples
- $V_1 = \sum_{i=1}^n w_i^2$
- $V_2 = \sum_{i=1}^n w_i^2$
- $V_3 = \sum_{i=1}^n w_i^3$.

11.1.4 Excess Kurtosis

The Kurtosis [27] is the degree of peakedness of a distribution of a real-valued random variable. In a similar way to the concept of skewness, kurtosis describes the shape of the distribution. The Kurtosis is defined in order to obtain a value of 0 for a Normal distribution. If it is greater than zero, it indicates that the distribution is high peaked; If it is smaller than zero, it testifies that the distribution is flat-topped.

Formally, the Kurtosis can be expressed as follows:

$$\gamma_2 = \frac{\mathbb{E}[(X - \mu)^4]}{(\mathbb{E}[(X - \mu)^2])^2} \quad (145)$$

In RAVEN, the kurtosis (excess) is computed as follows:

$$\begin{aligned} \frac{\mathbb{E}[(X - \mu)^4]}{(\mathbb{E}[(X - \mu)^2])^2} &\approx \frac{m_4 - 3m_2^2}{m_2^2} = \frac{\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^4 - 3 \left(\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2 \right)^2}{\left(\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2 \right)^2} && \text{if random sampling} \\ \\ \frac{\mathbb{E}[(X - \mu)^4]}{(\mathbb{E}[(X - \mu)^2])^2} &\approx \frac{m_4 - 3m_2^2}{m_2^2} = \frac{\frac{1}{V_1} \sum_{i=1}^n w_i \times (x_i - \bar{x})^4 - 3 \left(\frac{1}{V_1} \sum_{i=1}^n w_i \times (x_i - \bar{x})^2 \right)^2}{\left(\frac{1}{V_1} \sum_{i=1}^n w_i \times (x_i - \bar{x})^2 \right)^2} && \text{otherwise} \end{aligned} \quad (146)$$

where:

- w_i is the weight associated with the sample i
- n are the total number of samples
- $V_1 = \sum_{i=1}^n w_i$.

RAVEN performs an additional correction of kurtosis (excess) to obtain an unbiased estimation with respect to the sample-size [26]:

$$\begin{aligned} \frac{\mathbb{E}[(X-\mu)^4]}{(\mathbb{E}[(X-\mu)^2])^2} &\approx \frac{M_4-3M_2^2}{M_2^2} = \frac{n^2(n+1)}{(n-1)(n-2)(n-3)}m_4 - \frac{3n^2}{(n-2)(n-3)}m_2^2 && \text{if random sampling} \\ \frac{\mathbb{E}[(X-\mu)^4]}{(\mathbb{E}[(X-\mu)^2])^2} &\approx \frac{M_4-3M_2^2}{M_2^2} = \frac{V_1^2(V_1^4-4V_1V_3+3V_2^2)}{(V_1^2-V_2)(V_1^4-6V_1^2V_2+8V_1V_3+3V_2^2-6V_4)}m_4 - \\ &\quad \frac{3V_1^2(V_1^4-2V_1^2V_2+4V_1V_3-3V_2^2)}{(V_1^2-V_2)(V_1^4-6V_1^2V_2+8V_1V_3+3V_2^2-6V_4)}m_2^2 && \text{otherwise} \end{aligned} \quad (147)$$

where:

- w_i is the weight associated with the sample i
- n are the total number of samples
- $V_1 = \sum_{i=1}^n w_i^2$
- $V_2 = \sum_{i=1}^n w_i^2$
- $V_3 = \sum_{i=1}^n w_i^3$
- $V_4 = \sum_{i=1}^n w_i^4$.

11.1.5 Median

The median of the distribution of a real-valued random variable is the number separating the higher half from the lower half of all the possible values. The median of a finite list of numbers can be found by arranging all the observations from lowest value to highest value and picking the middle value.

Formally, the median m can be cast as the number that satisfy the following relation:

$$P(X \leq m) = P(X \geq m) = \int_{-\infty}^m pdf(x)dx = \frac{1}{2} \quad (148)$$

11.1.6 Percentile

A percentile (or a centile) is a measure indicating the value below which a given percentage of observations in a group of observations fall.

11.1.7 Covariance and Correlation Matrices

Simplistically, the Covariance is a measure of how much two random variables variate together. In other words, It represents a measurement of the correlation, in terms of variance, among different variables. If the greater values of one variable mainly correspond with the greater values of the other variable, and the same holds for the lesser values (i.e., the variables tend to show similar behavior) the covariance is positive. In the opposite case, when the greater values of one variable mainly correspond to the lesser values of the other (i.e., the variables tend to show opposite behavior) the covariance is negative. Formally, the Covariance can be expressed as

$$\Sigma(\mathbf{X}, \mathbf{Y}) = \mathbb{E} \left[(\mathbf{X} - \mathbb{E}[\mathbf{X}]) (\mathbf{Y} - \mathbb{E}[\mathbf{Y}])^T \right] \quad (149)$$

Based on the previous equation, in RAVEN each entry of the Covariance matrix is computed as follows:

$$\begin{aligned} \mathbb{E}[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])] &\approx \frac{1}{n} \sum_{i=1}^n (x_i - \mu_x)(y_i - \mu_y) && \text{if random sampling} \\ \mathbb{E}[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])] &\approx \frac{1}{V_1} \sum_{i=1}^n w_i \times (x_i - \mu_x)(y_i - \mu_y) && \text{otherwise} \end{aligned} \quad (150)$$

where:

- w_i is the weight associated with the sample i
- n are the total number of samples
- $V_1 = \sum_{i=1}^n w_i$.

The correlation matrix (Pearson product-moment correlation coefficient matrix) can be obtained through the Covariance matrix, as follows:

$$\Gamma(\mathbf{X}, \mathbf{Y}) = \frac{\Sigma(\mathbf{X}, \mathbf{Y})}{\sigma_x \sigma_y} \quad (151)$$

As it can be seen, The correlation between X and Y is the covariance of the corresponding standard scores.

11.1.8 Variance-Dependent Sensitivity Matrix

The variance dependent sensitivity matrix is the matrix of the sensitivity coefficients that show the relationship of the individual uncertainty component to the standard deviation of the reported value for a test item.

Formally:

$$\Lambda = \Sigma(\mathbf{X}, \mathbf{Y}) v c^{-1}(\mathbf{Y}) \quad (152)$$

where:

- $vc^{-1}(\mathbf{Y})$ is the inverse of the covariance of the input space.

11.2 Statistical Analysis through RAVEN

The goals of this section is to show how to:

1. Set up a sampling strategy to perform a final statistical analysis perturbing a driven code
2. Compute all the statistical moments and correlation/covariance metrics.

In order to accomplish these tasks, the following RAVEN **Entities** (XML blocks in the input files) need to be defined:

1. *RunInfo*:

```
<RunInfo>
  <JobName>Chapter-X/StatisticalAnalysis</JobName>
  <Sequence>
    sampleMC,statisticalAnalysisMC
  </Sequence>
  <WorkingDir>StatisticalAnalysis</WorkingDir>
  <batchSize>40</batchSize>
</RunInfo>
```

As shown in the other examples, the *RunInfo* **Entity** is intended to set up the desired analysis . In this specific case, two steps (**<Sequence>**) are sequentially run using forty processors (**<batchSize>**).

In the first step, the original physical model is sampled. The obtained results are analyzed with the Statistical Post-Processor.

2. *Files*:

```
<Files>
  <Input name="referenceInput.xml"
    type="input">referenceInput.xml</Input>
  <Input name="output_stat_analysis_mc.csv"
    type="">output_stat_analysis_mc.csv</Input>
</Files>
```

Since the driven code uses a single input file, in this section the original input is placed. As detailed in the user manual the attribute **name** represents the alias that is going to be used in all the other input blocks in order to refer to this file.

In addition, the output file of the *PostProcess* **Step** is here defined (CSV).

3. Models:

```
<Models>
  <Code name="testModel" subType="GenericCode">
    <executable>
      ../physicalCode/analyticalbateman/AnalyticalDplMain.py
    </executable>
    <clargs arg="python" type="prepend"/>
    <clargs arg="" extension=".xml" type="input"/>
    <clargs arg="" extension=".csv" type="output"/>
    <prepend>python</prepend>
  </Code>
  <PostProcessor name="statisticalAnalysis"
    subType="BasicStatistics">
    <what>all</what>
    <parameters>
      sigma-A, sigma-B, decay-A, decay-B, A, B, C, D
    </parameters>
  </PostProcessor>
</Models>
```

The goal of this example is to show how the principal statistical FOMs can be computed through RAVEN.

Indeed, in addition to the previously explained Code model, a Post-Processor model (Basic-Statistics) is here specified. Note that the post-process step is performed on all the variables/parameters used in this example ($\sigma - A$, $\sigma - B$, $\text{decay} - A$, $\text{decay} - B$, A , B , C and D).

4. Distributions:

```
<Distributions>
  <Uniform name="sigma">
    <lowerBound>0</lowerBound>
    <upperBound>1000</upperBound>
  </Uniform>
  <Uniform name="decayConstant">
    <lowerBound>0.00000001</lowerBound>
    <upperBound>0.0000001</upperBound>
  </Uniform>
</Distributions>
```

In the Distributions XML section, the stochastic models for the uncertainties are reported. In this case 2 distributions are defined:

- $\sigma \sim \mathbb{U}(0, 1000)$, used to model the uncertainties associated with the Model σ -A and σ -B
- $\text{decayConstant} \sim \mathbb{U}(1e-8, 1e-7)$, used to model the uncertainties associated with the Model decay -A and decay -B.

5. Samplers:

```
<Samplers>
  <MonteCarlo name="mc">
    <samplerInit>
      <limit>1200</limit>
    </samplerInit>
    <variable name="sigma-A">
      <distribution>sigma</distribution>
    </variable>
    <variable name="decay-A">
      <distribution>decayConstant</distribution>
    </variable>
    <variable name="sigma-B">
      <distribution>sigma</distribution>
    </variable>
    <variable name="decay-B">
      <distribution>decayConstant</distribution>
    </variable>
  </MonteCarlo>
</Samplers>
```

In order to obtain the data-set through which the statistical FOMs need to be computed, a *MonteCarlo* sampling approach is here employed.

6. DataObjects:

```
<DataObjects>
  <PointSet name="samplesMC">
    <Input>sigma-A, sigma-B, decay-A, decay-B</Input>
    <Output>A, B, C, D</Output>
  </PointSet>
  <HistorySet name="histories">
    <Input>sigma-A, sigma-B, decay-A, decay-B</Input>
    <Output>A, B, C, D, time</Output>
  </HistorySet>
</DataObjects>
```

In this block, two *DataObjects* are defined: 1) PointSet named “samplesMC” used to collect the final outcomes of the code, 2) HistorySet named “histories” in which the full time responses of the variables A, B, C, D are going to be stored.

7. Steps:

```
<Steps>
  <MultiRun name="sampleMC">
    <Input class="Files"
      type="input">referenceInput.xml</Input>
    <Model class="Models"
      type="Code">testModel</Model>
    <Sampler class="Samplers"
      type="MonteCarlo">mc</Sampler>
    <Output class="DataObjects"
      type="PointSet">samplesMC</Output>
    <Output class="DataObjects"
      type="HistorySet">histories</Output>
  </MultiRun>
  <PostProcess name="statisticalAnalysisMC">
    <Input class="DataObjects"
      type="PointSet">samplesMC</Input>
    <Model class="Models"
      type="PostProcessor">statisticalAnalysis</Model>
    <Output class="Files"
      type="">output_stat_analysis_mc.csv</Output>
  </PostProcess>
</Steps>
```

Finally, all the previously defined **Entities** can be combined in the `<Steps>` block. As inferable, 2 `<Steps>` have been inputted:

- `<MultiRun>` named “sampleMC”, used to run the multiple instances of the driven code and collect the outputs in the two *DataObjects*. As it can be seen, the `<Sampler>` is inputted to communicate to the *Step* that the driven code needs to be perturbed through the Grid sampling strategy.
- `<PostProcess>` named “statisticalAnalysisMC”, used compute all the statistical moments and FOMs based on the data obtained through the sampling strategy. As it can be noticed, the `<Output>` of the “sampleMC” *Step* is the `<Input>` of the “statisticalAnalysisMC” *Step*.

Tables 9-13 show all the results of the *PostProcess* step.

Table 9. Computed Moments and Cumulants.

Computed Quantities	A	B	C	D	decay-A	decay-B	sigma-A	sigma-B
<i>expected value</i>	5.97E-02	3.97E-01	9.82E-01	1.50E+00	5.57E-08	5.61E-08	5.07E+02	4.73E+02
<i>median</i>	2.45E-02	3.06E-01	9.89E-01	1.54E+00	5.73E-08	5.62E-08	5.11E+02	4.70E+02
<i>variance</i>	8.19E-03	6.00E-02	1.19E-02	1.49E-02	7.00E-16	6.83E-16	8.52E+04	8.64E+04
<i>sigma</i>	9.05E-02	2.45E-01	1.09E-01	1.22E-01	2.64E-08	2.61E-08	2.92E+02	2.94E+02
<i>variation coefficient</i>	1.52E+00	6.17E-01	1.11E-01	8.15E-02	4.75E-01	4.66E-01	5.75E-01	6.21E-01
<i>skewness</i>	2.91E+00	9.88E-01	-1.49E-01	-9.64E-01	-6.25E-02	-5.75E-02	-2.18E-02	7.62E-02
<i>kurtosis</i>	9.56E+00	-1.12E-01	-6.98E-01	-1.50E-01	-1.24E+00	-1.21E+00	-1.21E+00	-1.20E+00
<i>percentile 5%</i>	2.87E-03	1.48E-01	7.89E-01	1.24E+00	1.42E-08	1.45E-08	5.08E+01	2.97E+01
<i>percentile 95%</i>	2.51E-01	9.19E-01	1.16E+00	1.63E+00	9.54E-08	9.48E-08	9.59E+02	9.49E+02

Table 10. Covariance matrix.

Covariance	A	B	C	D	decay-A	decay-B	sigma-A	sigma-B
A	8.19E-03	-1.11E-03	-3.09E-03	-1.13E-04	-1.28E-09	5.14E-11	-1.49E+01	-3.74E-01
B	-1.11E-03	6.00E-02	2.26E-03	-2.96E-02	-7.80E-11	-6.02E-09	7.00E+00	-1.47E+00
C	-3.09E-03	2.26E-03	1.19E-02	7.15E-04	-1.44E-09	-4.11E-12	2.63E+01	3.19E-01
D	-1.13E-04	-2.96E-02	7.15E-04	1.49E-02	-1.21E-10	3.01E-09	1.12E+00	8.01E-01
decay-A	-1.28E-09	-7.80E-11	-1.44E-09	-1.21E-10	7.00E-16	-1.73E-17	-1.26E-07	2.07E-07
decay-B	5.14E-11	-6.02E-09	-4.11E-12	3.01E-09	-1.73E-17	6.83E-16	-1.86E-07	3.91E-08
sigma-A	-1.49E+01	7.00E+00	2.63E+01	1.12E+00	-1.26E-07	-1.86E-07	8.52E+04	1.79E+03
sigma-B	-3.74E-01	-1.47E+00	3.19E-01	8.01E-01	2.07E-07	3.91E-08	1.79E+03	8.64E+04

Table 11. Correlation matrix.

Correlation	A	B	C	D	decay-A	decay-B	sigma-A	sigma-B
A	1.00E+00	-5.02E-02	-3.13E-01	-1.03E-02	-5.35E-01	2.17E-02	-5.63E-01	-1.40E-02
B	-5.02E-02	1.00E+00	8.47E-02	-9.90E-01	-1.20E-02	-9.41E-01	9.80E-02	-2.04E-02
C	-3.13E-01	8.47E-02	1.00E+00	5.37E-02	-4.98E-01	-1.44E-03	8.25E-01	9.96E-03
D	-1.03E-02	-9.90E-01	5.37E-02	1.00E+00	-3.75E-02	9.43E-01	3.14E-02	2.23E-02
decay-A	-5.35E-01	-1.20E-02	-4.98E-01	-3.75E-02	1.00E+00	-2.50E-02	-1.64E-02	2.67E-02
decay-B	2.17E-02	-9.41E-01	-1.44E-03	9.43E-01	-2.50E-02	1.00E+00	-2.44E-02	5.08E-03
sigma-A	-5.63E-01	9.80E-02	8.25E-01	3.14E-02	-1.64E-02	-2.44E-02	1.00E+00	2.08E-02
sigma-B	-1.40E-02	-2.04E-02	9.96E-03	2.23E-02	2.67E-02	5.08E-03	2.08E-02	1.00E+00

Table 12. Variance Dependent Sensitivity matrix.

Variance Sensitivity	A	B	C	D	decay-A	decay-B	sigma-A	sigma-B
A	1.00E+00	-1.36E-01	-3.77E-01	-1.38E-02	-1.56E-07	6.27E-09	-1.82E+03	-4.56E+01
B	-1.86E-02	1.00E+00	3.77E-02	-4.94E-01	-1.30E-09	-1.00E-07	1.17E+02	-2.45E+01
C	-2.60E-01	1.90E-01	1.00E+00	6.01E-02	-1.21E-07	-3.46E-10	2.21E+03	2.68E+01
D	-7.60E-03	-1.99E+00	4.80E-02	1.00E+00	-8.11E-09	2.02E-07	7.51E+01	5.37E+01
decay-A	-1.83E+06	-1.11E+05	-2.05E+06	-1.73E+05	1.00E+00	-2.47E-02	-1.81E+08	2.96E+08
decay-B	7.52E+04	-8.82E+06	-6.02E+03	4.40E+06	-2.53E-02	1.00E+00	-2.72E+08	5.72E+07
sigma-A	-1.75E-04	8.22E-05	3.08E-04	1.32E-05	-1.48E-12	-2.19E-12	1.00E+00	2.10E-02
sigma-B	-4.33E-06	-1.70E-05	3.69E-06	9.27E-06	2.40E-12	4.52E-13	2.07E-02	1.00E+00

Table 13. Sensitivity matrix.

Sensitivity (I/O)	decay-A	decay-B	sigma-A	sigma-B
A	3.83E-06	-1.78E-04	-2.07E+04	-1.86E+06
B	-1.36E-05	6.28E-05	-8.80E+06	-3.14E+05
C	2.17E-06	3.05E-04	2.64E+04	-2.00E+06
D	6.96E-06	2.25E-05	4.40E+06	-6.19E+04

12 RAVEN Theory by way of Examples: Data Mining

Data mining is the computational process of discovering patterns in large data sets (“big data”) involving methods at the intersection of artificial intelligence, machine learning, statistics, and database systems. The overall goal of the data mining process is to extract information from a data set and transform it into an understandable structure for further use.

RAVEN has support of several different data mining algorithms, such as:

1. *Hierarchical methodologies*
2. *K-Means*
3. *Mean-Shift*, etc.

In this section only few algorithms will be analyzed, explaining the theory behind them by way of applied RAVEN examples.

12.1 Data Mining Theory

12.1.1 Clustering

A loose definition of clustering is the process of organizing objects into groups whose members are, in some way, similar. Therefore, a cluster is a collection of objects that are similar to each other and are dissimilar to the objects belonging to other clusters [28, 29].

The similarity criterion is distance. Two or more objects belong to the same cluster if they are “close” according to a specified distance. The approach of using distance metrics to clustering is called distance-based clustering and is used in this work.

The notion of distance implies that the data points lay in a metric space [30]:

Definition 1 (Metric Space) *A metric space is a space X provided with a function $d: f : X \times X \rightarrow \mathbb{R}$ satisfying the following properties $\forall \mathbf{x}, \mathbf{y} \in X$:*

- $d(\mathbf{x}, \mathbf{y}) \geq 0$
- $d(\mathbf{x}, \mathbf{y}) = d(\mathbf{y}, \mathbf{x})$
- $d(\mathbf{x}, \mathbf{y}) \leq d(\mathbf{x}, \mathbf{z}) + d(\mathbf{z}, \mathbf{y})$

The function $d(\mathbf{x}, \mathbf{y})$ is usually called the distance function. In a 2-dimensional Euclidean space (\mathbb{R}^2), the distance between points can be calculated using the Pythagorean theorem which is the direct application of the Euclidean distance and is a special case of the most general Minkowski distance $d_2(\mathbf{x}, \mathbf{y}) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$ between two points $\mathbf{x} = (x_1, x_2)$ and $\mathbf{y} = (y_1, y_2)$ in \mathbb{R}^2 .

In the literature [30], it is possible to find several types of distances other than the Euclidean and the Minkowski distance as shown in Table 14. The approach of using distance metrics is called distance-based clustering and will be used in this dissertation.

Table 14. Summary of the commonly used measures [30].

Measure	Form
Minkowski distance	$d_n(\mathbf{x}, \mathbf{y}) = \left(\sum_{k=1}^{\delta} x_k - y_k ^n \right)^{\frac{1}{n}}$
Euclidean distance	$d_2(\mathbf{x}, \mathbf{y}) = \left(\sum_{k=1}^{\delta} x_k - y_k ^2 \right)^{\frac{1}{2}}$
Taxicab distance	$d_1(\mathbf{x}, \mathbf{y}) = \sum_{k=1}^{\delta} x_k - y_k $
Supremum distance	$d_0(\mathbf{x}, \mathbf{y}) = \max_k x_k - y_k $
Mahalanobis distance	$d_M(\mathbf{x}, \mathbf{y}) = (\mathbf{x} - \mathbf{y})^T S^{-1} (\mathbf{x} - \mathbf{y})$

From a mathematical viewpoint, the concept of clustering [28] aims to find a partition $\mathbf{C} = \{C_1, \dots, C_l, \dots, C_L\}$ of the set of I scenarios $\mathbf{X} = \{\mathbf{x}_1, \dots, \mathbf{x}_i, \dots, \mathbf{x}_I\}$ where each scenario \mathbf{x}_i is represented as a δ -dimensional vector. Each C_l ($l = 1, \dots, L$) is called a cluster. The partition \mathbf{C} of \mathbf{X} is given as follows¹:

$$\begin{cases} \mathbf{C}_l \neq \emptyset, l = 1, \dots, L \\ \bigcup_{l=1}^L \mathbf{C}_l = \mathbf{X} \end{cases} \quad (153)$$

¹In most clustering algorithms each scenario belongs to only one cluster. However this is not always the case. In fuzzy clustering methodologies [31] a scenario may be allowed to belong to more than one cluster with a degree of membership $u_{i,j} \in [0, 1]$ which represents the member coefficient of the j scenario for the i^{th} cluster and satisfies the following properties:

$$\sum_{i=1}^K u_{i,j} = 1, \text{ and } \sum_{j=1}^N u_{i,j} < N, \forall j$$

12.1.2 Hierarchical Methodologies

These methodologies organize the data set into a hierarchical structure according to a proximity matrix. Each element $d(i, j)$ of this matrix contains the distance between the i^{th} and the j^{th} cluster center. The final results of this technique is a tree commonly called a dendrogram. This kind of representation has the advantages of providing a very informative description and visualization of the data structure even for high values of dimensionality.

The procedure to determine the dendrogram for a data set of I points in an δ -dimensional space is the following:

1. Start the analysis with a set of I clusters (i.e., each point is considered as a cluster).
2. Determine the proximity matrix M (dimension: $I \times I$): $M(i, j) = d(\mathbf{x}_i, \mathbf{x}_j)$ where \mathbf{x}_i and \mathbf{x}_j are the position of the i^{th} and the j^{th} cluster.
3. For each point p find the closest neighbor q from the proximity matrix M
4. Combine the points p and q
5. Repeat Steps 2, 3 and 4 until all the points of the data set are in the same cluster

The advantage of this kind of algorithm is the nice visualization of the results that show the underlying structure of the data set. However, the computational complexity for most of the hierarchical algorithm is of the order of $\mathcal{O}(I^2)$ (where I is the number of points in the data set).

12.1.3 K-Means

K -Means clustering algorithms belong to the more general family of Squared Error algorithms. The goal is to partition I data points \mathbf{x}_i ($i = 1, \dots, I$) into K clusters in which each data point maps to the cluster with the nearest mean. The stopping criterion is to find the global minimum of the error squared function χ defined as:

$$\chi = \sum_{i=1}^K \sum_{\mathbf{x}_j \in C_i} |\mathbf{x}_j - \boldsymbol{\mu}_i|^2 \quad (154)$$

where $\boldsymbol{\mu}_i$ is the centroid (i.e., the center) of the cluster C_i .

The procedure to determine the centroids $\boldsymbol{\mu}_i$ of K clusters (C_1, \dots, C_K) is the following:

1. Start with a set of K random centroids distributed in the state space

2. Assign each pattern to the the closest centroid
3. Determine the new K centroids according to the point-centroid membership

$$\mu_i = \frac{1}{N_i} \sum_{\mathbf{x}_j \in C_i} \mathbf{x}_j \quad (155)$$

where N_i corresponds to the number of of data points in the i^{th} cluster.

4. Repeat Steps 2 and 3 until convergence is met (i.e., until a minima of the χ function is reached)

K -Means algorithm is one of the most popular and used methodologies also due to the fact that is very straightforward to implement and the computational time is directly proportional to the cardinality of data points (i.e., $\mathcal{O}(I)$ where I is the number of data points). The main disadvantage is that the algorithm is sensitive to the choice of the initial partition and may converge to a local minimum of the error squared function [32]. Another disadvantage of this algorithm is that is only able to identify clusters having spherical or ellipsoidal geometry. Thus, K -Means is not able to identify clusters of points having arbitrary shapes. Moreover, the number of cluster K to be obtained is specified by the user prior the clustering process.

12.1.4 Mean-Shift

The Mean-Shift algorithm [33] is a non-parametric iterative procedure that can be used to assign each point to one cluster center through a set of local averaging operations [33]. The local averaging operations provide empirical cluster centers within the locality and define the vector which denotes the direction of increase for the underlying unknown density function.

The underlying idea is to treat each point \mathbf{x}_i ($i = 1, \dots, I$) of the dataset as an empirical probability distribution function using kernel $K(\mathbf{x}) : \mathbb{R}^{M \cdot K} \rightarrow \mathbb{R}$. This multivariate kernel density resides in a multidimensional space where regions with high data density (i.e., modes) correspond to local maxima of the density estimate $f_I(\mathbf{x})$ [34] defined by:

$$f_I(\mathbf{x}) = \frac{1}{Ih^d} \sum_{i=1}^I K\left(\frac{\mathbf{x} - \mathbf{x}_i}{h}\right), \quad (156)$$

where $\mathbf{x} \in \mathbb{R}^{M \cdot K}$ and h is often referred as the bandwidth associated with the kernel.

The kernel in Equation 156 serves as a weighting function [34] associated with each data point and is expressed as:

$$K(\mathbf{x}) = c_k k(\|\mathbf{x}\|^2) \quad (157)$$

where $k(x) : [0, \infty] \rightarrow \mathbb{R}$ is referred as the *kernel profile* and c_k is a normalization constant. The profile satisfies the following properties:

- $k(x)$ is non negative
- $k(x)$ is non increasing (i.e., $k(a) \geq k(b)$ if $a < b$)
- $k(x)$ is piecewise continuous and $\int_0^\infty k(x) dx < \infty$

In order to estimate the data points with highest probability from an initial estimate (i.e., the modes of $f_I(\mathbf{x})$), consider the gradient of the density function $\nabla_{\mathbf{x}} f_I(\mathbf{x}) = 0$ [33] where

$$\begin{aligned} \nabla_{\mathbf{x}} f_I(\mathbf{x}) &= \frac{2c_k}{Ih^{d+2}} \sum_{i=1}^I (\mathbf{x} - \mathbf{x}_i) k' \left(\left\| \frac{\mathbf{x} - \mathbf{x}_i}{h} \right\|^2 \right) \\ &= \underbrace{\frac{2c_k}{Ih^{d+2}} \left(\sum_{i=1}^I g \left(\left\| \frac{\mathbf{x} - \mathbf{x}_i}{h} \right\|^2 \right) \right)}_A \underbrace{\left(\frac{\sum_{i=1}^I \mathbf{x} g \left(\left\| \frac{\mathbf{x} - \mathbf{x}_i}{h} \right\|^2 \right)}{\sum_{i=1}^I g \left(\left\| \frac{\mathbf{x} - \mathbf{x}_i}{h} \right\|^2 \right)} - \mathbf{x} \right)}_B, \end{aligned} \quad (158)$$

which points in the direction of the increase in kernel density estimate. The kernel $K(\mathbf{x})$ is also referred to as the shadow of $G(\mathbf{x}) = c_g g(\|\mathbf{x}\|^2)$ [35] where c_g , similar to c_k , is a normalization constant and $g(x)$ is the derivative of $k(x)$ over x , i.e., $g(x) = k'(x)$. In the equation above, the first term denoted as A is a scalar proportional to the density estimate computed with the kernel $G(\mathbf{x})$ and does not provide information regarding where the mode resides. Unlike A , the vector quantity B , which is the second term in the equation above, is difference between the weighted mean

$$m(\mathbf{x}) = \frac{\sum_{i=1}^I \mathbf{x} g \left(\left\| \frac{\mathbf{x} - \mathbf{x}_i}{h} \right\|^2 \right)}{\sum_{i=1}^I g \left(\left\| \frac{\mathbf{x} - \mathbf{x}_i}{h} \right\|^2 \right)}. \quad (159)$$

and the initial estimate \mathbf{x} . This term points in the direction of local increase in density using kernel $G(\mathbf{x})$, hence provides a means to find the mode of the density. Note that all points used to compute a particular mode are considered to reside in the same cluster.

Since each data point \mathbf{x}_i (or scenario) is considered as an empirical probability distribution function, this consideration allows to include in the scenario clustering analysis also the possible uncertainty associated with each scenario.

12.1.5 DBSCAN

The Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm views clusters as areas of high density of data points. The data points in the low-density areas are seen as noise and border points, which are actually separating the clusters. Clusters found by DBSCAN can be any shape because of this approach. The main element of the DBSCAN algorithm is the concept of core samples, which are samples that are in areas of high density. Therefore, a cluster

is a set of core samples, each close to each other (measured by some distance measure) and a set of non-core samples that are close to a core sample (but are not themselves core samples). There are two parameters to the algorithm: *min_samples* and *eps*. Higher *min_samples* or lower *eps* indicate higher density necessary to form a cluster. A cluster is a set of core samples, that can be built by recursively by taking a core sample, finding all of its neighbors that are core samples, finding all of their neighbors that are core samples, and so on. A cluster also has a set of non-core samples, which are samples that are neighbors of a core sample in the cluster but are not themselves core samples; these are on the borders of a cluster. The DBSCAN algorithm finds core samples of high density and expands clusters from them. It is good for data, which contains clusters of similar density.

12.1.6 Dimensionality Reduction

The dimensionality δ of each data point (i.e., each scenario) is equal to the product of the number of variables (i.e., M) chosen to represent each scenario multiplied by the number of times each variable has been sampled. In order to reduce the computational time due to the high data dimensionality, the use of dimensionality reduction techniques was to reduce the number of variables M^2 .

The raw data generated by DET methodologies contain the temporal behavior of a vast set of variables (e.g., temperature, pressure). These variables are often heavily correlated and, consequently, the information contained in the set of M variables comprising the full state space can be condensed to a set of N variables where $N < M$. The objective of the dimensionality reduction process is to determine those N variables by finding the correlations among the original M variables³.

Linear algorithms, such as PCA [36] or multidimensional scaling (MDS) [37], have the advantage that they are easier to implement but they can only identify linear correlation among variables. On the other hand, methodologies such as Local Linear Embedding [38] and ISOMAP [39] are more computationally intensive but they are able to identify non-linear correlations.

Dimensionality reduction is the process of finding a bijective mapping function \mathfrak{F}

$$\mathfrak{F} : \mathbb{R}^D \mapsto \mathbb{R}^d \text{ (where } d < D \text{)} \quad (160)$$

which maps the data points from the D -dimensional space into a reduced d -dimensional space (i.e. embedding on a manifold) in such a way that the distances between each point and its neighbors are preserved. In our applications $D = M + 1$, i.e. M state variables plus time t .

²Other possible options are to reduce the number of sample instants K or to observe the local properties of the covariance matrix S .

³Note that those N variables are not necessarily a subset of the original M variables but, more likely, a combination of those M variables.

12.1.7 Dimensionality Reduction: Linear Algorithms

This section describes the two most important algorithms for dimensionality reduction:

1. PCA (see Section 12.1.8), and,
2. MDS (see Section 12.1.9).

12.1.8 Principal Component Analysis (PCA)

The main idea behind PCA [36] is to perform a linear mapping of the data set onto a lower dimensional space such that the variance of the data in the low-dimensional representation is maximized.

This is accomplished by determining the eigenvectors and their corresponding eigenvalues of the data covariance matrix⁴ S . The eigenvectors that correspond to the largest eigenvalues (i.e., the principal components) can be used as a set of basis functions. Thus, the original space is reduced to the space spanned by a few eigenvectors.

The algorithm is very straightforward to implement but, on the other hand, PCA is not able to identify non-linear correlations of more complex data sets.

12.1.9 Multidimensional Scaling (MDS)

Multidimensional scaling [37] is a popular technique used to analyze the properties of data sets. The scope of this methodology is to find a set of dimensions that preserve distances between data points.

This is performed by:

1. Creating dissimilarity matrix $D = [d_{ij}]$ where d_{ij} is the distance between two points x_i and x_j .
2. Finding the hyper-plane that preserves the dissimilarity matrix D (i.e., the *nearness* of points)

As in PCA analysis, the algorithm can be easily implemented but it is not able to identify non-linear correlations of more complex data sets.

⁴Given a data set in form of a vector Z , rows correspond to data dimensions (D) and columns correspond to data observations (Λ), the covariance matrix S is determined as: $S = \frac{1}{\Lambda-1} Z'Z$.

12.2 Data Mining through RAVEN

The goals of this section are about learning how to:

1. Set up a sampling strategy to apply clustering algorithms, perturbing a driven code
2. Analyze the data using clustering algorithms.

To accomplish these tasks, the following RAVEN **Entities** (XML blocks in the input files) need to be defined:

1. *RunInfo*:

```
<RunInfo>
  <JobName>Chapter-XI/dataMiningAnalysis</JobName>
  <Sequence>
    sampleMC,kmeans,pca
  </Sequence>
  <WorkingDir>dataMiningAnalysis</WorkingDir>
  <batchSize>40</batchSize>
</RunInfo>
```

The the *RunInfo* **Entity** is intended to set up the analysis sequence that needs to be performed. In this specific case, two steps (<Sequence>) are sequentially run using forty processors (<batchSize>).

In the first step, the original physical model is going to be sampled. The obtained results are going to be analyzed with data mining algorithms.

2. *Files*:

```
<Files>
  <Input name="referenceInput.xml"
    type="input">referenceInput.xml</Input>
</Files>
```

Since the driven code uses a single input file, in this section the original input is placed. The attribute **name** represents the alias that is going to be used in all the other input blocks in order to refer to this file.

3. *Models*:

```
<Models>
  <Code name="testModel" subType="GenericCode">
    <executable>
```

```

../physicalCode/analyticalbateman/AnalyticalDplMain.py
</executable>
<clargs arg="python" type="prepend"/>
<clargs arg="" extension=".xml" type="input"/>
<clargs arg="_" extension=".csv" type="output"/>
<prepend>python</prepend>
</Code>
<PostProcessor name="KMeans1" subType="DataMining">
  <KDD lib="SciKitLearn">
    <SKLtype>cluster|KMeans</SKLtype>
    <Features>A,B,C,D</Features>
    <n_clusters>2</n_clusters>
    <tol>1E-10</tol>
    <random_state>1</random_state>
    <init>k-means++</init>
    <precompute_distances>True</precompute_distances>
  </KDD>
</PostProcessor>
<PostProcessor name="PCA1" subType="DataMining">
  <KDD lib="SciKitLearn">
    <Features>A,B,C,D</Features>
    <SKLtype>decomposition|PCA</SKLtype>
    <n_components>2</n_components>
  </KDD>
</PostProcessor>
</Models>

```

The goal of this example is to show how the data mining algorithms in RAVEN can be useful to analyze large data set.

Indeed, in addition to the previously explained Code model, two Post-Processor models (*DataMining|cluster|KMeans* and *DataMining|decomposition|PCA*) are here specified. Note that the post-processing is performed on all the output FOMs used in this example (*A*, *B*, *C* and *D*).

4. *Distributions*:

```

<Distributions>
  <Uniform name="sigma">
    <lowerBound>0</lowerBound>
    <upperBound>1000</upperBound>
  </Uniform>
  <Uniform name="decayConstant">
    <lowerBound>0.00000001</lowerBound>
    <upperBound>0.0000001</upperBound>

```

```
</Uniform>
</Distributions>
```

In the Distributions XML section, the stochastic model for the uncertainties are reported. In this case 2 distributions are defined:

- $\sigma \sim \mathbb{U}(0, 1000)$, used to model the uncertainties associated with the Model σ -A and σ -B;
- $\text{decayConstant} \sim \mathbb{U}(1e-8, 1e-7)$, used to model the uncertainties associated with the Model decay -A and decay -B.

5. Samplers:

```
<Grid name="grid">
  <variable name="sigma-A">
    <distribution>sigma</distribution>
    <grid construction="equal" steps="9" type="CDF">0.01
      0.99</grid>
  </variable>
  <variable name="decay-A">
    <distribution>decayConstant</distribution>
    <grid construction="equal" steps="9" type="CDF">0.01
      0.99</grid>
  </variable>
  <variable name="sigma-B">
    <distribution>sigma</distribution>
    <grid construction="equal" steps="9"
      type="CDF">0.01 0.99</grid>
  </variable>
</Grid>
```

In order to obtain the data-set on which the data mining algorithms are going to be applied, a *Grid* sampling approach is here employed.

6. DataObjects:

```
<DataObjects>
  <PointSet name="samplesMC">
    <Input>sigma-A, sigma-B, decay-A, decay-B</Input>
    <Output>A, B, C, D</Output>
  </PointSet>
  <HistorySet name="histories">
    <Input>sigma-A, sigma-B, decay-A, decay-B</Input>
    <Output>A, B, C, D, time</Output>
  </HistorySet>
</DataObjects>
```

```

</HistorySet>
</DataObjects>

```

In this block, two *DataObjects* are defined: 1) PointSet named “samplesMC” used to collect the final outcomes of the code, 2) HistorySet named “histories” in which the full time responses of the variables A, B, C, D are going to be stored.

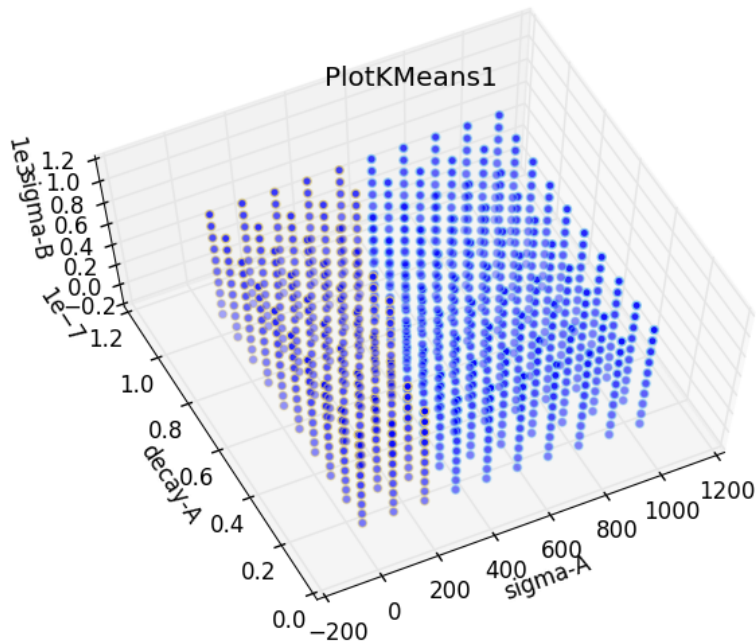


Figure 37. K-means clustering on original dataset.

7. Steps:

```

<Steps>
  <MultiRun name="sampleMC">
    <Input class="Files"
      type="input">referenceInput.xml</Input>
    <Model class="Models"
      type="Code">testModel</Model>
    <Sampler class="Samplers" type="Grid">grid</Sampler>
    <Output class="DataObjects"
      type="PointSet">samplesMC</Output>
  </MultiRun>
</Steps>

```

```

    <Output class="DataObjects"
      type="HistorySet">histories</Output>
  </MultiRun>
  <PostProcess name="kmeans" pauseAtEnd="True">
    <Input class="DataObjects"
      type="PointSet">samplesMC</Input>
    <Model class="Models"
      type="PostProcessor">KMeans1</Model>
    <Output class="DataObjects"
      type="PointSet">samplesMC</Output>
    <Output class="OutStreams"
      type="Plot">PlotKMeans1</Output>
    <Output class="OutStreams" type="Plot">PlotAll</Output>
    <Output class="OutStreams"
      type="Print">samplesMCDump</Output>
  </PostProcess>
  <PostProcess name="pca" pauseAtEnd="True">
    <Input class="DataObjects"
      type="PointSet">samplesMC</Input>
    <Model class="Models" type="PostProcessor">PCA1</Model>
    <Output class="DataObjects"
      type="PointSet">samplesMC</Output>
    <Output class="OutStreams"
      type="Plot">PlotPCA1</Output>
  </PostProcess>
</Steps>

```

Finally, all the previously defined **Entities** can be combined in the **<Steps>** block; 3 **<Steps>** have been inputted:

- **<MultiRun>** named “sampleMC”, used to run the multiple instances of the driven code and collect the outputs in the two *DataObjects*. The **<Sampler>** is inputted to communicate to the *Step* that the driven code needs to be perturbed through the Grid sampling strategy;
- **<PostProcess>** named “kmeans”, used to analyze the data obtained through the sampling strategy. In this step, a K-Means algorithm is going to be employed, plotting the clustering results; *Step* that the driven code needs to be perturbed through the Grid sampling strategy;
- **<PostProcess>** named “pca”, used to analyze the data obtained through the sampling strategy. In this Step, a PCA algorithm is going to be employed, plotting the decomposition results.

Figure 37 shows the clustering on the original input space.

Figure 38 shows the clustering on the projected input space. It can be noticed, that the algorithm fully capture the fact that the parameter $\sigma - B$ does not impact the response A (being completely independent).

Figure 39 shows the PCA decomposition on the data set.

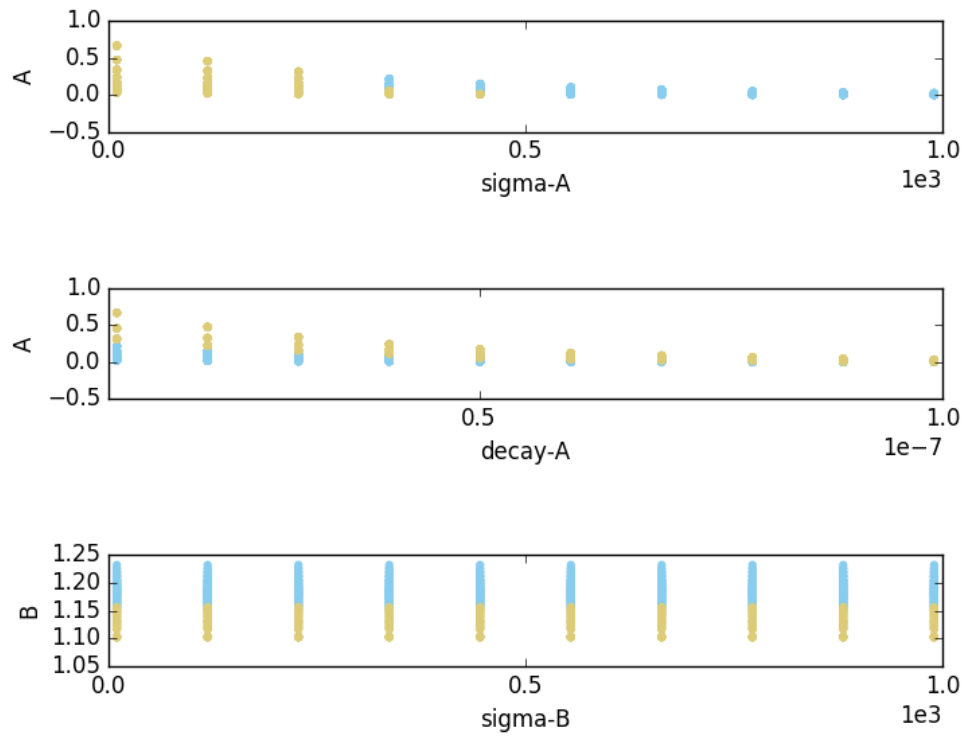


Figure 38. K-means clustering on projected parameters.

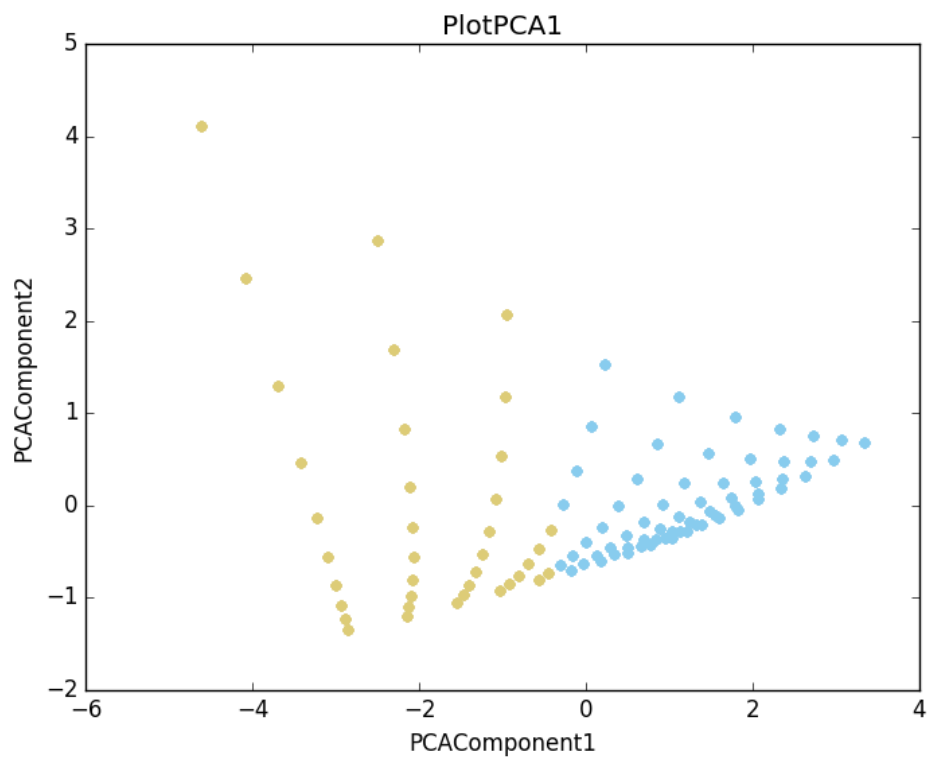


Figure 39. Principal Component Analysis.

Appendices

A Running RAVEN

The RAVEN code is a blend of C++, C, and Python languages. The entry point resides on the Python side and is accessible via a command line interface. After following the instructions in the previous Section, RAVEN is ready to be used. The RAVEN driver is contained in the folder “raven/framework.” To run RAVEN, open a terminal and use the following command (replace `inputFileName.xml` with your RAVEN input file):

```
python raven/framework/Driver.py inputFileName.xml
```

Alternatively, the `raven_framework` script can be used. In this case, the command is:

```
raven_framework inputFileName.xml
```


References

- [1] A. Alfonsi, C. Rabiti, D. Mandelli, J. Cogliati, and B. Kinoshita, “Raven as a tool for dynamic probabilistic risk assessment: Software overview,” in *Proceedings of International Conference of mathematics and Computational Methods Applied to Nuclear Science and Engineering (M&C 2013), Sun Valley (Idaho)*, pp. 1247–1261, 2013.
- [2] A. Alfonsi, C. Rabiti, D. Mandelli, J. Cogliati, B. Kinoshita, and A. Naviglio, “Dynamic event tree analysis through raven,” in *Proceedings of ANS PSA 2013 International Topical Meeting on Probabilistic Safety Assessment and Analysis, Columbia (South Carolina)*, 2013.
- [3] C. Rabiti, A. Alfonsi, D. Mandelli, J. Cogliati, and R. Kinoshita, “Deployment and overview of raven capabilities for a probabilistic risk assessment demo for a pwr station blackout,” Tech. Rep. INL/EXT-13-29510, Idaho National Laboratory (INL), 2013.
- [4] A. Alfonsi, C. Rabiti, D. Mandelli, J. Cogliati, and B. Kinoshita, “Raven and dynamic probabilistic risk assessment: Software overview,” in *Proceedings of ESREL European Safety and Reliability Conference (ESREL 2014), Wrocklaw (Poland)*, 2014.
- [5] D. Anders, R. Berry, D. Gaston, R. Martineau, J. Peterson, H. Zhang, H. Zhao, and L. Zou, “Relap-7 level 2 milestone report: Demonstration of a steady state single phase pwr simulation with relap-7,” Tech. Rep. INL/EXT-12-25924, Idaho National Laboratory (INL), 2012.
- [6] “Neams: The nuclear energy advanced modeling and simulation program,” Tech. Rep. ANL/NE-13/5.
- [7] “Light water reactor sustainability program integrated program plan,” Tech. Rep. INL-EXT-11-23452, April 2013.
- [8] R.-D. development team, “Relap5/mod3.3 code manual,” tech. rep., Idaho National Laboratory, October 2015.
- [9] A. Alfonsi, C. Rabiti, D. Mandelli, J. Cogliati, B. Kinoshita, and A. Naviglio, “Hybrid dynamic event tree sampling strategy in raven code,” in *Proceedings of ANS PSA 2015 International Topical Meeting on Probabilistic Safety Assessment and Analysis, Sun Valley (Idaho)*, 2015.
- [10] A. Alfonsi, C. Rabiti, D. Mandelli, J. Cogliati, and B. Kinoshita, “Adaptive dynamic event tree in raven code,” in *Proceedings American Nuclear Society 2014 Winter Meeting Nuclear-The Foundation of Clean Energy, Anaheim, CA, (2014)*, 2014.
- [11] C. J. C. Burges, “A tutorial on support vector machines for pattern recognition,” *Data Mining and Knowledge Discovery*, vol. 2, pp. 121–167, June 1998.
- [12] P. et al., “Scikit-learn: Machine learning in python,” *Journal of Machine Learning Research*, pp. 2825–2830, 2011.

- [13] C. Rabiti, A. Alfonsi, J. Cogliati, D. Mandelli, R. Kinoshita, and S. Sen, “Raven user manual,” tech. rep., Idaho National Laboratory, 2015.
- [14] J. Devooght and C. Smidts, “Probabilistic reactor dynamics - i: The theory of continuous event trees,” *Nuclear Science and Engineering*, vol. 111, pp. 229–240, 1992.
- [15] Wiener, “The homogeneous chaos,” *American Journal of Mathematics*, vol. 60, pp. 897–936, 1938.
- [16] Askey and Wilson, “Some basic hypergeometric orthogonal polynomials that generalize jacobi polynomials,” *Memoirs of the American Mathematical Society*, vol. 54, pp. 1–55, 1985.
- [17] Xiu and Karniadakis, “The wiener–askey polynomial chaos for stochastic differential equations,” *SIAM Journal on Scientific Computing*, vol. 24, no. 2, pp. 619–644, 2002.
- [18] Babuska, Nobile, and Tempone, “A stochastic collocation method for elliptic partial differential equations with random input data,” *SIAM Journal on Numerical Analysis*, vol. 45, 2007.
- [19] Novak and Ritter, “The curse of dimension and a universal method for numerical integration,” in *Multivariate approximation and splines* (G. Nürnberger, J. Schmidt, and G. Walz, eds.), vol. 125 of *ISNM International Series of Numerical Mathematics*, pp. 177–187, Birkhäuser Basel, 1997.
- [20] Smolyak, “Quadrature and interpolation formulas for tensor products of certain classes of functions,” in *Dokl. Akad. Nauk SSSR*, vol. 4, p. 123, 1963.
- [21] C. Rabiti, D. M. A. Alfonsi, J. Cogliati, and B. Kinoshita, “Mathematical framework for the analysis of dynamic stochastic systems with the raven code,” in *Proceedings of International Conference of mathematics and Computational Methods Applied to Nuclear Science and Engineering (M&C 2013), Sun Valley (Idaho)*, pp. 320–332, 2013.
- [22] C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005.
- [23] N. S. Altman, “An introduction to kernel and nearest-neighbor nonparametric regression,” *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992.
- [24] D. Shepard, “A two-dimensional interpolation function for irregularly-spaced data,” in *Proceedings of the 1968 23rd ACM National Conference*, ACM 1968, (New York, NY, USA), pp. 517–524, ACM, 1968.
- [25] C. Habermann and F. Kindermann, “Multidimensional spline interpolation: Theory and applications,” *Computational Economics*, vol. 30, no. 2, pp. 153–169, 2007.
- [26] L. Rimoldini, “Weighted skewness and kurtosis unbiased by sample size,” tech. rep., Astrophysical Observatory of the University of Geneva, April 2013.

- [27] A. M. and S. I. A, *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Dover, 9th ed., 1972.
- [28] X. Rui and Ii, “Survey of clustering algorithms,” *IEEE Transactions on Neural Networks*, vol. 16, pp. 645–678, May 2005.
- [29] D. Mandelli, A. Yilmaz, T. Aldemir, K. Metzroth, and R. Denning, “Scenario clustering and dynamic probabilistic risk assessment,” *Reliability Engineering & System Safety*, vol. 115, pp. 146–160, 2013.
- [30] B. Mendelson, *Introduction to Topology*. New York (NY), USA: Dover Publications, 1990.
- [31] E. Zio and F. D. Maio, “Processing dynamic scenarios from a reliability analysis of a nuclear power plant digital instrumentation and control system,” *Annals of Nuclear Energy*, vol. 36, pp. 1386–1399, 2009.
- [32] A. K. Jain, K. Dubes, and C. Richard, *Algorithms for clustering data*. Upper Saddle River, NJ (USA): Prentice-Hall, Inc., 1988.
- [33] K. Fukunaga and L. Hostetler, “The estimation of the gradient of a density function, with applications in pattern recognition,” *IEEE Transactions on Information Theory*, vol. 21, no. 1, pp. 32–40, 1975.
- [34] T. Cacoullos, “Estimation of a multivariate density,” *Annals of the Institute of Statistical Mathematics*, vol. 18, no. 1, pp. 179–189, 1966.
- [35] Y. A. Sheikh, E. Khan, and T. Kanade, “Mode-seeking by medoidshifts,” in *Eleventh IEEE International Conference on Computer Vision (ICCV 2007)*, no. 1, October 2007.
- [36] I. T. Jolliffe, *Principal Component Analysis*. Springer, second ed., October 2002.
- [37] I. Borg and P. Groenen, *Modern Multidimensional Scaling: Theory and Applications*. Springer-Verlag New York, 2005.
- [38] S. T. Roweis and L. K. Saul, “Nonlinear dimensionality reduction by locally linear embedding,” *Science*, vol. 290, pp. 2323–2328, 2000.
- [39] J. B. Tenenbaum, V. de Silva, and J. C. Langford, “A global geometric framework for nonlinear dimensionality reduction,” *Science*, vol. 290, pp. 2319–2323, 2000.

