MANUAL

INL/EXT-18-44465
Revision 1
Printed July 11, 2024

RAVEN User Guide

Andrea Alfonsi, Cristian Rabiti, Diego Mandelli, Joshua Cogliati, Congjian Wang, Paul W. Talbot, Jia Zhou, Pralhad Burli,Mohammad G. Abdo

Prepared by
Idaho National Laboratory
Idaho Falls, Idaho 83415

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

Idaho National Laboratory

INL/EXT-18-44465
Revision 1
Printed July 11, 2024

RAVEN User Guide

Andrea Alfonsi
Cristian Rabiti
Diego Mandelli
Joshua Cogliati
Congjian Wang
Paul W. Talbot
Jia Zhou
Pralhad Burli
Mohammad G. Abdo

Contents

1 Introduction

1.1 Project Background

The development of RAVEN started in 2012 when, within the Nuclear Energy Advanced Modeling and Simulation (NEAMS) program [1], 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 [2].

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 [3], any Multiphysics Object Oriented Simulation Environment-based App, etc.) or are currently ongoing for coupling RAVEN with several different software.

1.2 Acquiring and Installing RAVEN

RAVEN is supported on three separate computing platforms: Linux, OSX (Apple Macintosh), and Microsoft Windows. Currently, RAVEN is open-source and downloadable from RAVEN GitHub repository: `https://github.com/idaholab/raven`. New users should visit `https://github.com/idaholab/raven/wiki` or refer to the user manual [4] to get started with RAVEN. This typically involves the following steps:

- *Download RAVEN*
  You can download the source code of RAVEN from `https://github.com/idaholab/raven`.
- *Install RAVEN dependencies*
  Instructions are available from `https://github.com/idaholab/raven/wiki`, or the user manual [4].
- *Install RAVEN*
  Instructions are available from `https://github.com/idaholab/raven/wiki`, or the user manual [4].

- *Run RAVEN*
  If RAVEN is installed successfully, please run the regression tests to verify your installation:

```
      ./run_tests
```

  Normally there are skipped tests because either some of the codes are not available, or some of the test are not currently working. The output will explain why each is skipped. If all the tests pass, you are ready to run RAVEN. Now, open a terminal and use the following command (replace `<inputFileName.xml>` with your RAVEN input file):

```
      raven_framework <inputFileName.xml>
```

  where the `raven_framework` script can be found in the RAVEN folder. Alternatively, the `raven_framework.py` script contained in the folder "`raven`" can be directly used:

```
      python raven/raven_framework.py <inputFileName.xml>
```

- *Participate in RAVEN user communities*
  Join RAVEN mail lists to get help and updates of RAVEN: `https://groups.google.com/forum/#!forum/inl-raven-users`.

1.3 User Guide Formats

In order to highlight some parts of the user guide 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
```

- ***RAVEN XML input example:***

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

- ***Bash Commands:***

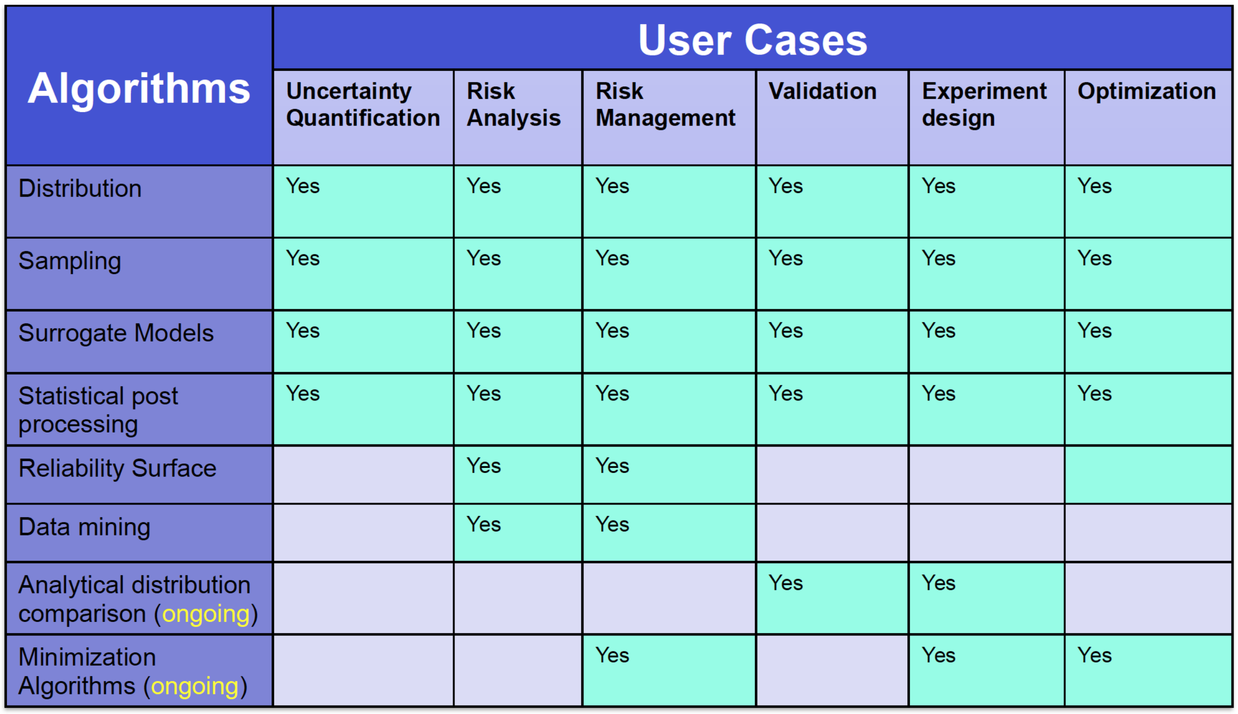
```
cd trunk/raven/
./raven_libs_script.sh
cd ../../
```

1.4 Capabilities of RAVEN

RAVEN [5] [6] [7] [8] 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 [9], currently under development at Idaho National Laboratory (INL). Now, RAVEN is not only a framework to perform DPRA but it is a flexible and multi-purpose uncertainty quantification, regression analysis, probabilistic risk assessment, data analysis and model optimization platform. Depending on the tasks to be accomplished and on the probabilistic characterization of the problem, RAVEN perturbs (e.g., Monte-Carlo, Latin hypercube, reliability surface search) the response of the system under consideration by altering its own parameters. The system is modeled by third party software (e.g., RELAP5-3D, MAAP5, BISON, etc.) and accessible to RAVEN either directly (software coupling) or indirectly (via input/output files). The data generated by the sampling process is analyzed using classical statistical and more advanced data mining approaches. RAVEN also manages the parallel dispatching (i.e. both on desktop/workstation and large High Performance Computing machines) of the software representing the physical model. RAVEN heavily relies on artificial intelligence algorithms to construct surrogate models of complex physical systems in order to perform uncertainty quantification, reliability analysis (limit state surface) and parametric studies.

The main capabilities of RAVEN, with brief descriptions, are summarized here, or one can check the Figure. 1. These capabilities may be used on their own or as building blocks to construct the sought workflow. In addition, RAVEN also provides some more sophisticated **Ensemble algorithms** such as **EnsembleForward**, **EnsembleModel** to combine the existing capabilites.

- **Sensitivity Analysis and Uncertainty Quantification**: Sensitivity analysis is a mathematical tool that can be used to identify the key sources of uncertainties. Uncertainty quantification is a process by which probabilistic information about system responses can be computed according to specified input parameter probability distributions. Available approaches in RAVEN include **Monte Carlo**, **Grid**, **Stratified (Latin hypercube)**, **Sparse Grid Collocation**, **Sobol**, **Adaptive Sparse Grid**, **Adaptive Sobol** and **BasicStatistics**.
- **Design of Experiments**: The design of experiments (DOE) is a powerful tool that can be used to explore the parameter space at a variety of experimental situations. It can be used



Algorithms	User Cases					
	Uncertainty Quantification	Risk Analysis	Risk Management	Validation	Experiment design	Optimization
Distribution	Yes	Yes	Yes	Yes	Yes	Yes
Sampling	Yes	Yes	Yes	Yes	Yes	Yes
Surrogate Models	Yes	Yes	Yes	Yes	Yes	Yes
Statistical post processing	Yes	Yes	Yes	Yes	Yes	Yes
Reliability Surface		Yes	Yes			
Data mining		Yes	Yes			
Analytical distribution comparison (ongoing)				Yes	Yes	
Minimization Algorithms (ongoing)			Yes		Yes	Yes

Figure 1. RAVEN Capabilities vs. Needs

to determine the relationship between input factors and the desired outputs. Available approaches in RAVEN include **Factorial Design** (i.e. General full factorial, 2-level fractional-factorial and Plackett-Burman) and **Response Surface Design** (i.e. Box-Behnken and Central composite algorithms).

- **Risk Mitigation or Model Optimization**: RAVEN uses the **Optimizer**, a powerful sampler-like entity that searches the input space to find minimum or maximum values of a reponse. Currently available optimizers include **Simultaneous Perturbation Stochastic Approximation (SPSA)**.

- **Risk Analysis**: Available approaches in RAVEN include **Dynamic Event Tree**, **Limit Surface Search**, **Hybrid Dynamic Event Tree**, **Adaptive Dynamic Event Tree**, **Adaptive Hybrid Dynamic Event Tree**, **Data Mining**, **Importance Rank**, **Safest Point**, and **Basic Statistics**.

- **Risk Management**: Available approaches in RAVEN include **Reduced order models**, approaches used for sensitivity and uncertainty analysis, and **Dynamic Event Tree** methods.

- **Validation**: Available approaches in RAVEN include **ROMs**, **Comparison Statistics** and **Validation Metrics**

In addition, RAVEN includes a number of related advanced capabilities. **Surrogate or Reduced order models (ROMs)** are mathematical model trained to predict a response of interest of a physical system. Typically, ROMs trade speed for accuracy representing a faster, rough estimate of the underlying systems. They can be used to explore the input parameter space for optimization or sensitivity and uncertainty studies. **Ensemble Model** is able to combine **Codes**, **External Models** and **ROMs**. It is intended to create a chain of models whose execution order is determined by the input/output relationships among them. If the relationships among the models evolve in a non-linear system, a Picard's iteration scheme is employed.

1.5 Components of RAVEN

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. For more information about XML, please click on the link: **XML tutorial**.
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.
- **`<VariableGroups>`**: Creates a collection of variables.
- **`<Optimizers>`**: Performs the driving of a specific goal function over the model for value optimization.
- **`<Metrics>`**: Calculate the distance values among points and histories.

- `<Steps>`: Combines other blocks to detail a step in the RAVEN workflow including I/O and computations to be performed.

Each of these components are explained in dedicated sections of the user manual [4], and can be used as building blocks to construct certain calculation flow, as shown in Figure. 2. In this guide, we will only show how to use these components to build the analysis flow, and we recommend the user to check the user manual [4] for the detailed descriptions.

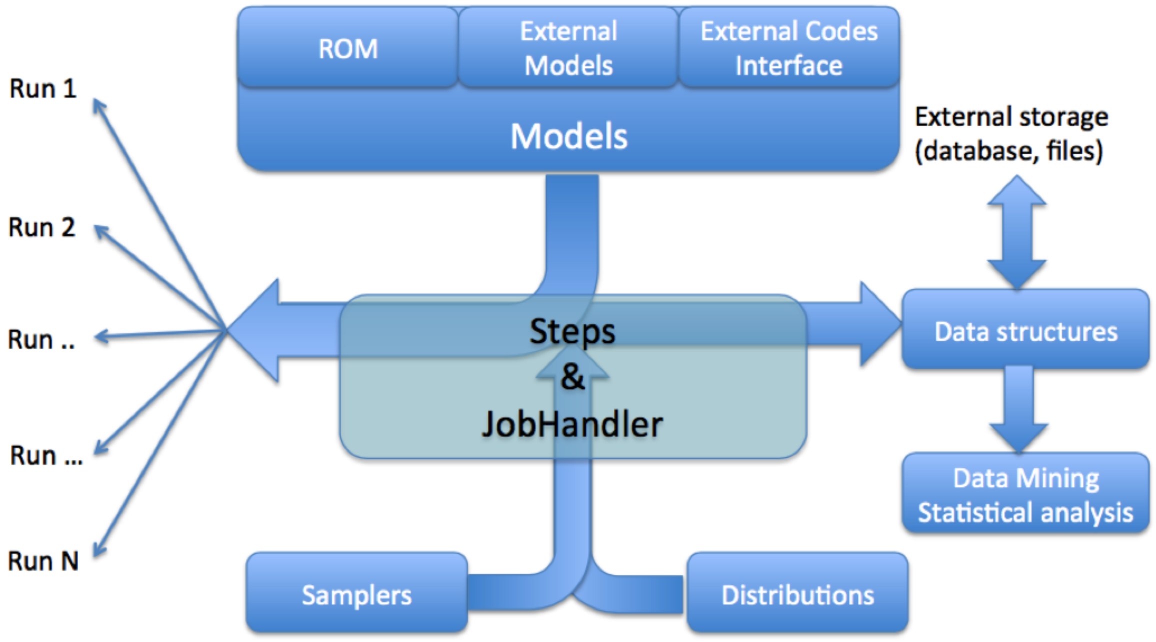


Figure 2. RAVEN structures

In addition, RAVEN allows the user to load any external input file that contains the required XML nodes into the RAVEN main input file, and provide the standard XML comments, using <!-- and -->. For example, one can use the following template to load the `<Distributions>` from file ‘Distributions.xml’.

```
<Simulation verbosity='all'>
  ...
  <!-- An Example Comment -->
  <Steps verbosity='debug'>
    ...
  </Steps>
  ...
  <ExternalXML node='Distributions'
     xmlToLoad='path_to_folder/Distributions.xml'/>
  ...
</Simulation>
```

RAVEN also allows the user to control the level of output to the user interface by using `verbosity` system. These settings can be declared globally as attributes in the `<Simulation>` node, or locally in each block node as shown in above template. The verbosity levels are

- `'silent'` - Only simulation-breaking errors are displayed.
- `'quiet'` - Errors as well as warnings are displayed.
- `'all'` (default) - Errors, warnings, and messages are displayed.
- `'debug'` - For developers. All errors, warnings, messages, and debug messages are displayed.

1.6 Code Interfaces of RAVEN

The procedure of coupling a new code/application with RAVEN is a straightforward process. 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. For example, for all the codes currently supported by RAVEN (e.g. RELAP-7, RELAP-5D, BISON, MAMMOTH, etc.), the coupling is performed through a Python interface that interprets the information coming from RAVEN and translates them into the input of the driven code. The couping procedure does not require modifying RAVEN itself. Instread, 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). In addition, RAVEN will manage concurrent executions of your simulations in parallel, whether on a local desktop or remote high-performance cluster.

Figure. 3 depicts the different APIs between RAVEN and the computational models, i.e. the **ROM**, **External Models** and **External Code** APIs.

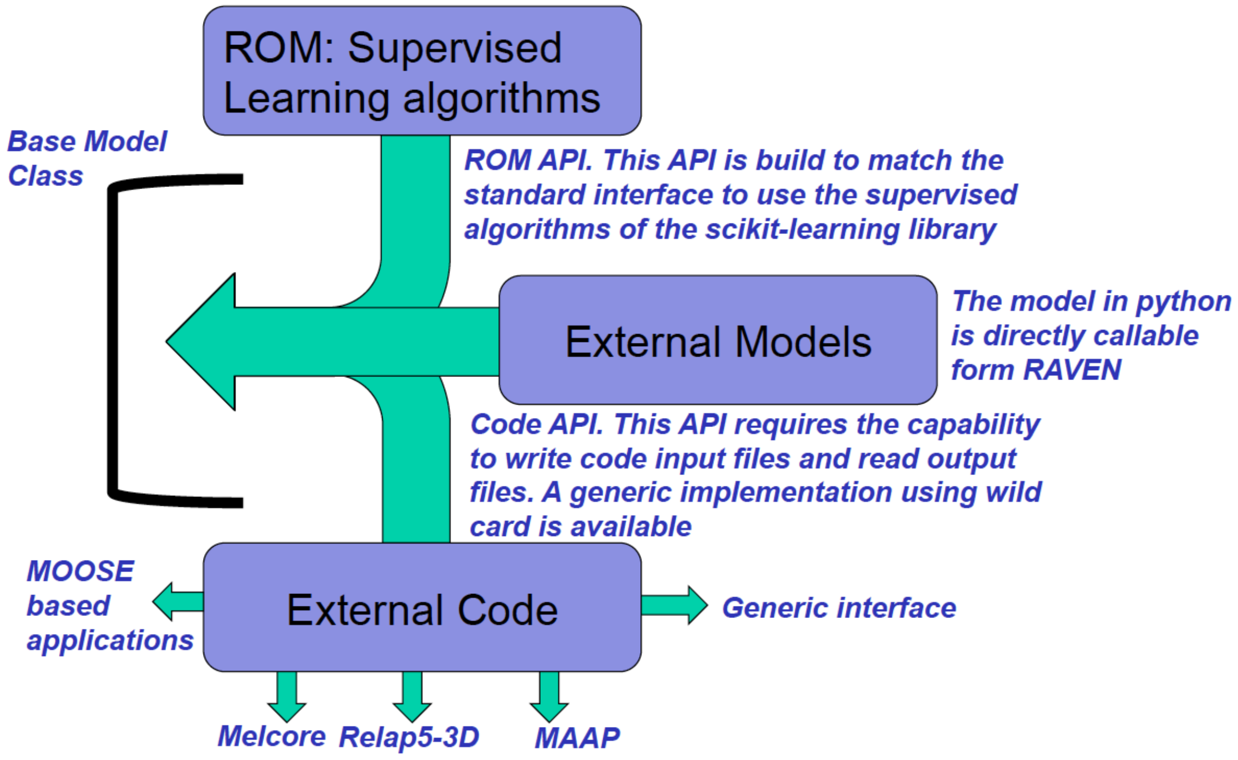

ROM: Supervised Learning algorithms
Base Model Class
ROM API. This API is build to match the standard interface to use the supervised algorithms of the scikit-learning library
External Models
The model in python is directly callable form RAVEN
Code API. This API requires the capability to write code input files and read output files. A generic implementation using wild card is available
MOOSE based applications
External Code
Generic interface
Melcore
Relap5-3D
MAAP

Figure 3. RAVEN Application Programming Interfaces

1.7 User Guide Organization

The goal of this document is to provide a set of detailed examples that can help the user to become familiar with the RAVEN code. 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. New users should consult the **RAVEN Tutorial** to get started.

- **RAVEN Tutorial**: section 2
- **Sampling Strategies**: section 3 and section 4
- **Restart**: section 5
- **Reduced Order Modeling**: section 6
- **Risk Analysis**: section 7
- **Data Mining**: section 8
- **Model Optimization**: section 9

2 RAVEN Tutorial

2.1 Example Model: Analytic Bateman

This section is intended for the new users to familiarize them with how to perform their studies through RAVEN. A simple example, conventionally called **AnalyticBateman**, has been developed. It solves a system of ordinary differential equations (ODEs), of the form:

$$\begin{cases} \frac{\mathrm{d}\mathbf{X}}{\mathrm{d}t} = \mathbf{S} - \mathbf{L} \\ \mathbf{X}(t=0) = \mathbf{X_0} \end{cases} \tag{1}$$

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 two ODEs as follows:

$$\begin{cases} \frac{\mathrm{d}x_1}{\mathrm{d}t} = \phi(t) \times \sigma_{x_1} - \lambda_{x_1} \times x_1(t) \\ \frac{\mathrm{d}x_2}{\mathrm{d}t} = \phi(t) \times \sigma_{x_2} - \lambda_{x_2} \times x_2(t) + x_1(t) \times \lambda_{x_1} \\ x_1(t=0) = x_1^0 \\ x_2(t=0) = 0.0 \end{cases} \tag{2}$$

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 as an example in this guide. All the files required for this system are located at "*raven/tests/framework/user_guide/physicalCode*".

raven/tests/framework/user_guide/physicalCode/analyticalbateman/Input.xml

```
<AnalyticalBateman>
  <totalTime>10</totalTime>

  <powerHistory>1 1 1</powerHistory>

  <flux>10000 10000 10000</flux>
  <stepDays>0 100 200 300</stepDays>

  <timeSteps>10 10 10</timeSteps>

  <nuclides>
    <A>
      <equationType>N1</equationType>
      <initialMass>1.0</initialMass>
      <decayConstant>0</decayConstant>
      <sigma>1</sigma>
      <ANumber>230</ANumber>
    </A>
    <B>
      <equationType>N2</equationType>
      <initialMass>1.0</initialMass>
      <decayConstant>0.00000005</decayConstant>
      <sigma>10</sigma>
      <ANumber>200</ANumber>
    </B>
    <C>
      <equationType>N3</equationType>
      <initialMass>1.0</initialMass>
      <decayConstant>0.000000005</decayConstant>
      <sigma>45</sigma>
      <ANumber>150</ANumber>
    </C>
    <D>
      <equationType>N4</equationType>
      <initialMass>1.0</initialMass>
      <decayConstant>0.00000008</decayConstant>
      <sigma>3</sigma>
      <ANumber>100</ANumber>
    </D>
  </nuclides>
</AnalyticalBateman>
```

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

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

2.2 Build RAVEN input: `<SingleRun>`

In this section, we will show the user how to use RAVEN to run a single instance of a driven code, and printing some variables. We will start to build a very simple RAVEN input, and this input file can be found at "*raven/tests/framework/user_guide/ravenTutorial/singleRun.xml*". From this process, we hope the user can get a better idea about RAVEN entities and learn how to build their own RAVEN inputs for their applications. In order to accomplish these tasks, the following procedures are needed:

1. **Set up the running environment**: `<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.). For the simple case, the **RunInfo** will look like:

   raven/tests/framework/user_guide/ravenTutorial/singleRun.xml

```
<Simulation>
  ...
  <RunInfo>
    <JobName>singleRun</JobName>
    <Sequence>single</Sequence>
    <WorkingDir>singleRunAnalysis</WorkingDir>
    <batchSize>1</batchSize>
  </RunInfo>
  ...
</Simulation>
```

In this specific case, only one step named `'single'` is going to be sequentially run using a single processor as defined by `<BatchSize>`. All the output files and temporary files will be dumped in the folder `'singleRunAnalysis'`.

2. **Provide the required files**: `<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. Each entry in the `<Files>` block is a tag with the file type. Files given through the input XML at this point are all `<Input>` type. Each `<Input>` node has a required attributes `name`. It does not need to be the actual filename, and it is the name by which RAVEN will use to identify the specific file. Other optional attributes are not directly used by RAVEN, and they are mainly used by the **CodeInterface**. More detailed information can be found in the user manual [4]. For the simple case, the **Files** will look like:

   raven/tests/framework/user_guide/ravenTutorial/singleRun.xml

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

   This RAVEN input file shows that the user will provide a file that is located at "*../commonFiles/referenceInput.xml*" with reference name `'referenceInput.xml'`. This file will be available for use via other RAVEN input blocks or entities. In this case, a relative path to the working directory specified via `<WorkingDir>` under node `<RunInfo>` is used.

3. **Link between RAVEN and driven code**: `<Models>`
   The **Models** entity represents the projection from the input to the output space. In other words, the Model entity can be seen as a transfer function between the input and output space. Currently, RAVEN defines the following sub-entities:

   - *Code*, represents the driven code, through external code interfaces (see [4])
   - *ExternalModel*, represents a physical or mathematical model that is directly implemented by the user in a Python module

- *ROM*, represents the Reduced Order Model, interfaced with several algorithms
- *HybridModel*, automatic/smart Entity to automatically choose between a ROM (or a set of them) and an High-Fidelity Model (e.g. ExternalModel, Code)
- *PostProcessor*, is used to perform action on data, such as computation of statistical moments, correlation matrices, etc.

For simplicity, only *Code* is used here for the demonstration, and the input block looks like:

`raven/tests/framework/user_guide/ravenTutorial/singleRun.xml`

```
<Simulation>
  ...
  <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" />
    </Code>
  </Models>
  ...
</Simulation>
```

As shown in the **`<Models>`** block, the subnodes defined for **`<Code>`** is equivalent to:

```
     python ../physicalCode/analyticalbateman/AnalyticalDplMain.py
```

with the requirement of extensions of input and output files, as defined via **`<clargs>`**, to be **`'.xml'`** and **`'.csv'`**, respectively. In this case, the **GenericCode** interface is employed. This interface is meant to handle a wide variety of generic codes that take straightforward input files and produce CSV files. **Note:** If a code contains cross-dependent data, the generic interface is not applicable. For more detailed information, the user can refer to section **Existing Interface** of the user manual [4].

4. **Container of input and output data**: **`<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 the following data types, each with a particular conceptual meaning:

   - *PointSet* is a collection of individual objects, each describing the state of the system at a certain point (e.g. in time). 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).

- *HistorySet* is a collection of individual objects, each describing the temporal evolution of the state of the system within a certain input domain. 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.
- *DataSet* is a collection of individual objects and a generalization of the previously described DataObjects, aimed to contain a mixture of data (scalars, arrays, etc.). The variables here stored can be independent (i.e. scalars) or dependent (arrays) on certain dimensions (e.g. time, coordinates, etc.). It can be considered a mapping between multiple sets of parameters in the input space (both dependent and/or independent) and the resulting sets of evolution in the output space (bothdependentand/orindependent).

The DataObjects represent the designeted 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.). For the simple case, the **<DataObjects>** block of RAVEN input is:

raven/tests/framework/user_guide/ravenTutorial/singleRun.xml

```
<Simulation>
  ...
  <DataObjects>
    <HistorySet name="history">
      <Input>InputPlaceHolder</Input>
      <Output>A,B,C,D,time</Output>
    </HistorySet>
  </DataObjects>
  ...
</Simulation>
```

<HistorySet> with a user-defined identifier (e.g. “history”) is used to collect the mass evolutions of four given isotopes, i.e. A, B, C, D. **<Input>** node is used to list the input parametes to which this data is connected. If there is no input data associated with this node, the **'InputPlaceHolder'** can be used. **<Output>** is used to list the output parameters to which this data is connected. Similarly, if there is no output data associated with this node, the **'OutputPlaceHolder'** can be used. This is mainly because both **<Input>** and **<Output>** nodes are required for all types of **DataObjects**.

5. **Print and plot input and output data**: **<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.

In this case, a simple `<OutStreams>` is used to output the mass evolutions of all four model variables into a CSV file with the name prefix "print_history".

raven/tests/framework/user_guide/ravenTutorial/singleRun.xml

```
<Simulation>
  ...
  <OutStreams>
    <Print name="print_history">
      <type>csv</type>
      <source>history</source>
    </Print>
  </OutStreams>
  ...
</Simulation>
```

6. **Control of executions**: `<Steps>`
The **Steps** entity is used to create a peculiar analysis flow via combining together different RAVEN entities. It is the location where all the defined entities get finally linked in order to perform a combined action on a certain *Model*. In order to perform this linking, each entity defined in the Step needs to "play" a role:

- *Input* represents the input of the step. The allowable input objects depend on the type of *Model* in this step.
- *Model* represents a physical or mathematical system or behavior. The object used in this role defines the allowable types of inputs and outputs usable in this step.
- *Output* defines where to collect the results of an action performed by the *Model*. It is generally one of the following types: **DataObjects**, **Databases**, or **OutStreams**.
- *Sampler* defines the sampling strategy to be used to probe the model. **Note:** When a sampling strategy is employed, the "variables" defined in the `<variable>` blocks are going to be directly placed in the output objects of type **DataObjects** and **Databases**.
- *Function* is an extremely importance role. It introduces the capability to perform pre or post processing of model inputs and outputs. Its specific behavior depends on the step is using it.
- *ROM* defines an acceleration reduced order model to use for a step.
- *SolutionExport*, represents the container of the eventual output of a step. It is the entity that is used to export the solution of a *Sampler* or post-processors.

Currently, RAVEN supports the following 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

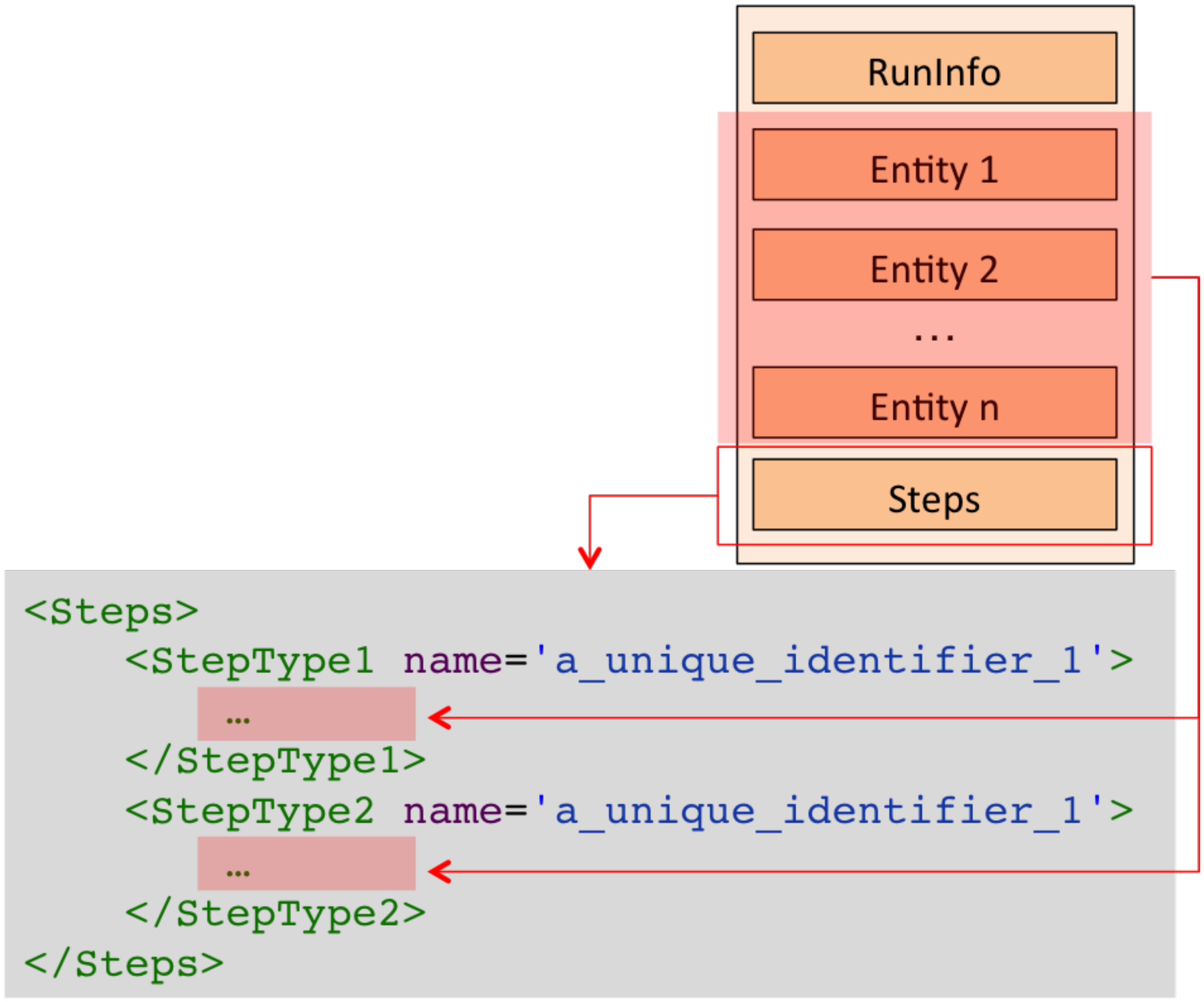
For this example, the **<SingleRun>** is used to assemble a calculation flow, i.e. perform a single action of a model.

raven/tests/framework/user_guide/ravenTutorial/singleRun.xml

```
<Simulation>
  ...
  <Steps>
    <SingleRun name="single">
      <Input class="Files" type="input">referenceInput.xml</Input>
      <Model class="Models" type="Code">testModel</Model>
      <Output class="DataObjects"
         type="HistorySet">history</Output>
      <Output class="OutStreams"
         type="Print">print_history</Output>
    </SingleRun>
  </Steps>
  ...
</Simulation>
```

The code “testModel” will be executed once, and the outputs will be collected into a **'DataObjects'** of type **HistorySet**. In addition, **'OutStreams'** is used to print the output data into a CSV file.

The core of the RAVEN calculation flow is the **Steps** system. The **Steps** is in charge of assembling different entities in RAVEN in order to perform a task defined by the kind of step being used (see Figure. 4).



```
<Steps>
    <StepType1 name='a_unique_identifier_1'>
       ...
    </StepType1>
    <StepType2 name='a_unique_identifier_1'>
       ...
    </StepType2>
</Steps>
```

Figure 4. Example of the Steps **Entity** and its connection in the input file.

2.3 Build RAVEN Input: `<IOStep>`

The `<IOStep>` acts as a ”transfer network” among different RAVEN storing or streaming objects. The number of `<Input>` and `<Output>` is unlimited. This `<IOStep>` assumes one-to-one mapping, i.e. the first `<Input>` is going to be used for the first `<Output>`, etc. **Note:** If the `<Output>` nodes are class `'OutStreams'`, the user does not need to follow this assumption, since **OutStreams** objects are already linked to **DataObjects** in the relative RAVEN input block. The **IOStep** can be used to:

- construct/update a *Database* from a *DataObjects* object, and vice versa;

- construct/update a *Database* or a *DataObjects* object from *CSV* files contained in a directory;
- stream the content of a *Database* or a *DataObjects* out through an **OutStream** object;
- store/retrieve a *ROM* to/from an external *File* using Pickle module of Python.

The last function can be used to create and store mathematical model of fast solution trained to predict a response of interest of a physical system. This model can be recovered in other simulations or used to evaluate the response of a physical system in a Python program by the implementing of the Pickle module.

2.3.1 Perform input/output operations

In this case, we will use **`<IOStep>`** to stream the output data from the *DataObjects* out through the *OutStreams*. The **`<IOStep>`** block is shown as follows:

raven/tests/framework/user_guide/ravenTutorial/singleRunPlotAndPrint.xml

```
<Simulation>
  ...
  <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>
      <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>
  ...
</Simulation>
```

As shown in the **`<IOStep>`**, the input is a history set “history” that is previous generated by the **SingleRun** step. The data stored in the “history” will be printed and plotted via the **OutStreams**.The data object “history” is defined as follows:

raven/tests/framework/user_guide/ravenTutorial/singleRunPlotAndPrint.xml

```
<Simulation>
  ...
  <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>
  ...
</Simulation>
```

Note: If a *PointSet* data object is used to collect the temporal output data, only the data from the last time step will be stored in this data object. As demonstrated in this case, the output csv file with name "pointValues.csv" generated through the **OutStreams** in **SingleRun** step only contains the data for the last time step. This file can be found in the working directory specified by sub-node **<WorkingDir>** under node **<RunInfo>**.

As mentioned before, **OutStreams** can be used to plot the data stored in the data objects. The following input block demonstrates the use of **OutStreams** for plotting.

raven/tests/framework/user_guide/ravenTutorial/singleRunPlotAndPrint.xml

```
<Simulation>
  ...
  <OutStreams>
    <Print name="pointValues">
      <type>csv</type>
      <source>pointValues</source>
    </Print>
    <Print name="history">
      <type>csv</type>
      <source>history</source>
    </Print>
    <Plot name="historyPlot" overwrite="false" verbosity="debug">
      <plotSettings>
        <plot>
          <type>line</type>
          <x>history|Output|time</x>
          <y>history|Output|A</y>
          <kwargs>
            <color>blue</color>
          </kwargs>
        </plot>
        <plot>
```

```
          <type>line</type>
          <x>history|Output|time</x>
          <y>history|Output|B</y>
          <kwargs>
            <color>orange</color>
          </kwargs>
        </plot>
        <plot>
          <type>line</type>
          <x>history|Output|time</x>
          <y>history|Output|C</y>
          <kwargs>
            <color>green</color>
          </kwargs>
        </plot>
        <plot>
          <type>line</type>
          <x>history|Output|time</x>
          <y>history|Output|D</y>
          <kwargs>
            <color>red</color>
          </kwargs>
        </plot>
        <xlabel>time (s)</xlabel>
        <ylabel>evolution (kg)</ylabel>
      </plotSettings>
      <actions>
        <how>png</how>
        <title>
          <text> </text>
        </title>
        <figureProperties>
          <figsize>(8.,6.)</figsize>
          <dpi>100</dpi>
        </figureProperties>
      </actions>
    </Plot>
  </OutStreams>
  ...
</Simulation>
```

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

- *Print*: named "history" connected with the *DataObjects* **Entity** "history" (`<source>`) When this object 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 in the working directory.

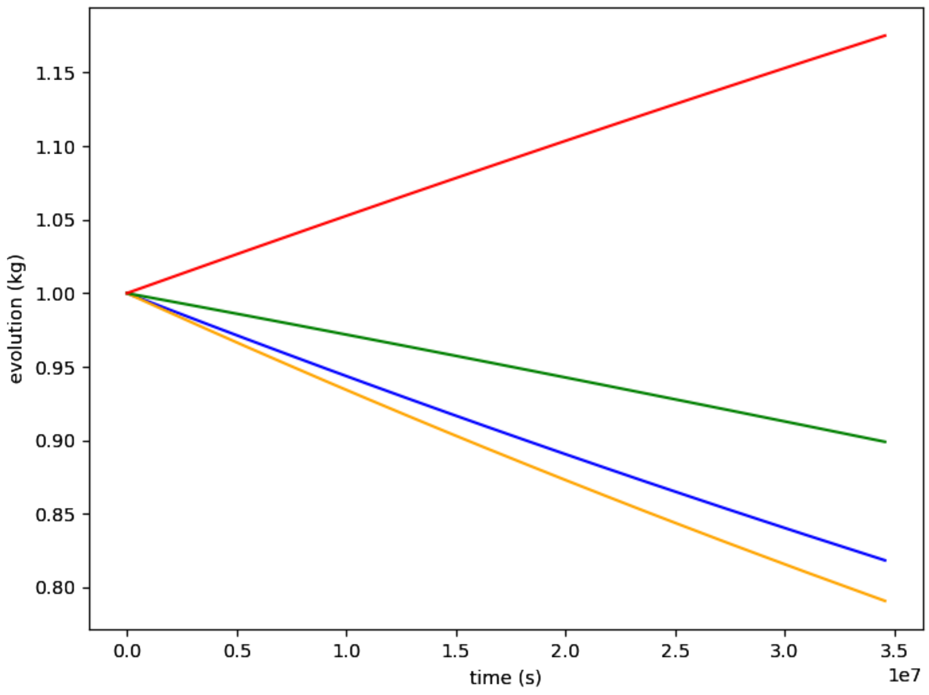


Figure 5. Plot of the history for variables A, B, C, D.

For examples of the numerical data produced by the OutStreams *Print*, see `history_0.csv` in the directory `raven/tests/framework/user_guide/ravenTutorial/ gold/singleRunPlot`
As previously mentioned, Figure 5 reports the four plots (four variables) drawn in the same picture.

2.3.2 Sub-plot and selectively printing.

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 `<IOStep>` needs to be modified as follows:

raven/tests/framework/user_guide/ravenTutorial/singleRunSubPlotsAndSelectivePrint.xml

```
<Simulation>
  ...
  <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>
      <Output class="DataObjects" type="HistorySet">history</Output>
    </SingleRun>
    <IOStep name="writehistory" pauseAtEnd="True">
      <Input class="DataObjects" type="PointSet">pointValues</Input>
      <Input class="DataObjects" type="HistorySet">history</Input>
      <Output class="OutStreams" type="Print">history</Output>
      <Output class="OutStreams" type="Plot">historyPlot</Output>
      <Output class="OutStreams" type="Print">pointValues</Output>
    </IOStep>
  </Steps>
  ...
</Simulation>
```

Note: as mentioned before, this `<IOStep>` does not need to follow the one-to-one mapping, since *OutStreams* are already linked to the *DataObjects*. And the *OutStreams* **Entity** in the input defined in the previous section needs to be modified as follows:

raven/tests/framework/user_guide/ravenTutorial/singleRunSubPlotsAndSelectivePrint.xml

```
<Simulation>
  ...
  <OutStreams>
    <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>
    <Plot 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>
          <kwargs>
            <color>blue</color>
          </kwargs>
          <gridLocation>
            <x>0</x>
            <y>0</y>
          </gridLocation>
        </plot>
        <plot>
          <type>line</type>
          <x>history|Output|time</x>
          <y>history|Output|B</y>
          <kwargs>
            <color>orange</color>
          </kwargs>
          <gridLocation>
            <x>1</x>
            <y>0</y>
          </gridLocation>
        </plot>
        <plot>
          <type>line</type>
          <x>history|Output|time</x>
          <y>history|Output|C</y>
          <kwargs>
            <color>green</color>
          </kwargs>
          <gridLocation>
            <x>0</x>
            <y>1</y>
          </gridLocation>
        </plot>
        <plot>
          <type>line</type>
          <x>history|Output|time</x>
          <y>history|Output|D</y>
          <kwargs>
            <color>red</color>
          </kwargs>
```

```
          <gridLocation>
            <x>1</x>
            <y>1</y>
          </gridLocation>
        </plot>
        <xlabel>time (s)</xlabel>
        <ylabel>evolution (kg)</ylabel>
      </plotSettings>
      <actions>
        <how>png</how>
        <title>
          <text> </text>
        </title>
      </actions>
    </Plot>
  </OutStreams>
  ...
</Simulation>
```

1. ***Print***: With respect to the *Print* nodes defined in the previous section, 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 along with time.

2. ***Plot***: Note that the *Plot* **Entity** does not differ much with respect to the one in previous section: 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 printed data will dump to the CSV file *history_0.csv*, and Figure 6 reports the four plots (four variables) drawn in the same picture.

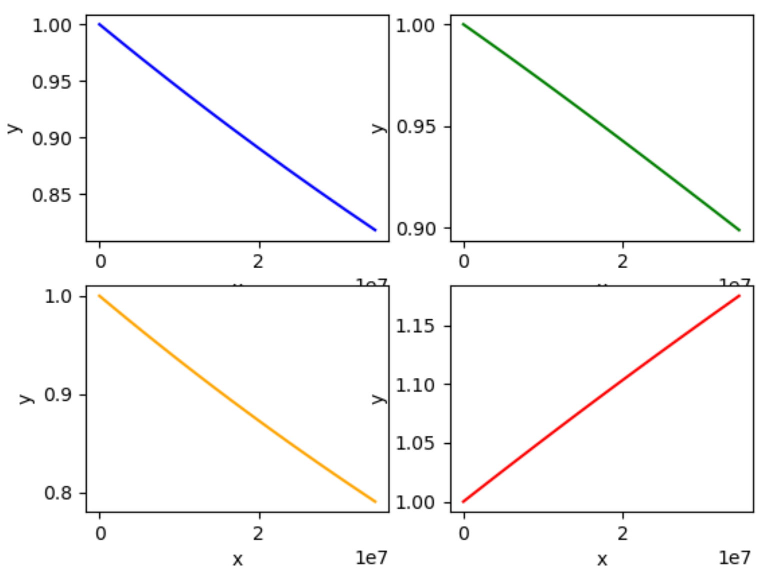


Figure 6. Subplot of the history for variables A, B, C, D.

2.4 Build RAVEN Input: `<MultiRun>`

The **MultiRun** step allows the user to assemble the calculation flow of an analysis that requires multiple "runs" of the same model. This step is used, for example, when the input (space) of the model needs to be perturbed by a particular sampling strategy. In the `<MultiRun>` input block, the user needs to specify the objects that need to be used for the different allowable roles. This step accepts the following roles:

- *Input*
- *Model*
- *Output*
- *Sampler*
- *Optimizer*
- *SolutionExport*

MultiRun is intended 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 MC sampling).

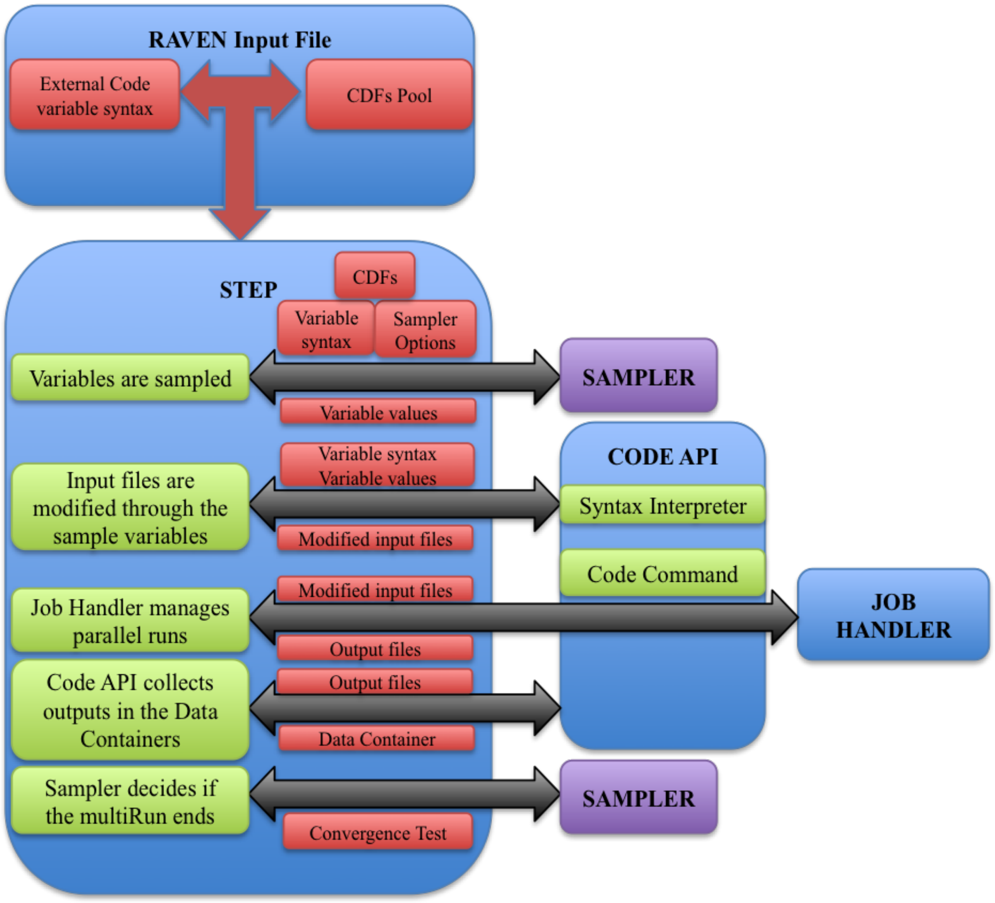


Figure 7. Calculation flow for a multi-run sampling

As shown in Figure 7, 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. 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 to store the data for later recovery.

In this section, we will show the user how to set up the *Sampler*, and employ **MultiRun** to execute all perturbed models. The use of *Optimizer* and *SolutionExport* will be introduced in another section. 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 sparse grid, 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.

The sampler is probably the most important entity in the RAVEN framework. It provides many different sampling strategies that can be used in almost all RAVEN related applications. In this section, we will only illustrate the simplest forward sampler, i.e. *Monte-Carlo*, to familarize the user with the use of sampler. Monte-Carlo method is one of the most-used methodologies in several mathematic disciplines. The theory of this method can be found in the RAVEN theory manual. In addition, we will continue to use the **AnalyticBateman** to illustrate the setup of **MultiRun** and **Samplers**. In order to accomplish these tasks, the following precedures or RAVEN entities are needed:

1. Set up the running environment: `<RunInfo>`
2. Provide the required files: `<Files>`
3. Link between RAVEN and dirven code: `<Models>`
4. Define probability distribution functions for inputs: `<Distributions>`
5. Set up a simple Monte-Carlo sampling for perturbing the input space: `<Samplers>`
6. Store the input and output data: `<DataObjects>`
7. Print and plot input and output data: `<OutStreams>`
8. Control multiple executions: `<Steps>`

In section 2.2, we have already discussed the use of **<RunInfo>**, **<Files>**, **<Models>**, **<DataObjects>**, **<OutStreams>**. For practice, the user can try to build these RAVEN entities by themselves, and refer to the complete input file located at *raven/tests/framework/user_guide/ ravenTutorial/MonteCarlo.xml*. In this section, we'd like to show the users how to set up **<Distributions>**, **<Samplers>** and **MultiRun** of **<Steps>**. RAVEN employs **<Distributions>** to define many different probability distribution functions (PDFs) that can be used to characterize the input parameters. One can consider the **<Distributions>** entity to be a container of all the stochastic representation 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.

For the **AnalyticBateman** example, two 1-D uniform distributions are defined:

- $sigma \sim \mathbb{U}(1, 10)$, used to model the uncertainties associated with the *sigma* Model variables;
- $decayConstant \sim \mathbb{U}(0.5e-8, 1e-8)$, used to model the uncertainties associated with the Model variable *decay constants*. Note that the same distribution can be re-used for multiple input variables, while still keeping those variables independent.

The following is the definition of **<Distributions>** block that is used for the **AnalyticBateman** problem:

raven/tests/framework/user_guide/ravenTutorial/MonteCarlo.xml

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

For uniform distributions, only **<lowerBound>** and **<upperBound>** are required. For other distributions, please refer to the RAVEN user manual.

As we already mentioned, we will employ Monte-Carlo sampling strategy to demonstrate **MultiRun**. To employ the Monte-Carlo sampling strategy, a **<MonteCarlo>** node needs to be defined. The user also needs to specify the variables that need to be sampled using **<variable>**. In addition, the setting for this sampler need to be specified in the **<samplerInit>** block. The only required sub-node **<limit>** is used to specify the number of Monte Carlo samples. The user can also use other optional sub-node to characterize their samplers. For this example, the **<Samplers>** block is:

raven/tests/framework/user_guide/ravenTutorial/MonteCarlo.xml

```
<Simulation>
  ...
  <Samplers>
    <MonteCarlo name="monteCarlo">
      <samplerInit>
        <limit>100</limit>
        <reseedEachIteration>True</reseedEachIteration>
        <initialSeed>0</initialSeed>
      </samplerInit>
      <variable name="sigma-A">
        <distribution>sigma</distribution>
      </variable>
      <variable name="decay-A">
        <distribution>decayConstant</distribution>
      </variable>
    </MonteCarlo>
  </Samplers>
  ...
</Simulation>
```

In this case, the Monte-Carlo method is employed on two model variables, each of which are listed by name and are associated with a distribution. Note that the *decay-* and *sigma-* variables are associated with the distributions $decayConstant$ and $sigma$, respectively. These variables and their values are passed to the model via the generic code interface. This requires the users to make some changes in their input files in order to accept these variables. For example, the input file of **AnalyticBateman** becomes:

raven/tests/framework/user_guide/ravenTutorial/commonFiles/referenceInput_generic_CI.xml

```
<AnalyticalBateman>
  ...
  <nuclides>
    <A>
      <equationType>N1</equationType>
      <initialMass>1.0</initialMass>
      <decayConstant>$RAVEN-decay-A|10$</decayConstant>
      <sigma>$RAVEN-sigma-A|10$</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>
  ...
</AnalyticalBateman>
```

As shown in this example, the values of nodes **`<sigma>`** and **`<decayConstant>`** are replaced with variables **`'$RAVEN-decay-A|10$'`** and **`'$RAVEN-sigma-A|10$'`**, respectively. **Note:** we use prefix **`'RAVEN-'`** + **`'variable names defined inside RAVEN input files'`** within **`'$ $'`** to define the RAVEN-editable input parameters. In other words, the RAVEN-editable input parameters is used to transfer the sampled values of RAVEN variables to input parameters of given code. This is the only way to connect the input parameters of code with variables of RAVEN if *Generic code interface* is employed. In addition, we use the wild-cards | to define the format of the value of the RAVEN-editable input parameters. In this case, the value that is going to be replaced by the generic code interface will be left-justified with a string length of 10 (e.g. "∣10"). Other formatting options can be found in the RAVEN user manual.

As we already mentioned, the *Generic code interface* requires that the codes need to return a CSV file with the input parameters and output parameters. The filename of the CSV file should includes:

- prefix: "out∼"
- filename: the base of input filename without extension
- extension: ".csv"

For this case, the input filename is "referenceInput_generic_CI.xml", thus the output CSV filename should be "out∼referenceInput_generic_CI.csv".

Once all the other entities are defined in the RAVEN input file, they must be combined in the **`<Steps>`** block, which dictates the workflow of RAVEN. For this case, two **`<Steps>`** are defined:

- **`<MultiRun>`** "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 specified 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** to a CSV file and 2) plot the data in the EPS file.

Figures 8 and 9 show the report generated by RAVEN of the evolution of the variable A and its final values, respectively.

raven/tests/framework/user_guide/ravenTutorial/MonteCarlo.xml

```
<Simulation>
  ...
  <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">samplesPlot_A</Output>
      <Output class="OutStreams" type="Plot">history_A</Output>
      <Output class="OutStreams" type="Print">histories</Output>
    </IOStep>
  </Steps>
  ...
</Simulation>
```

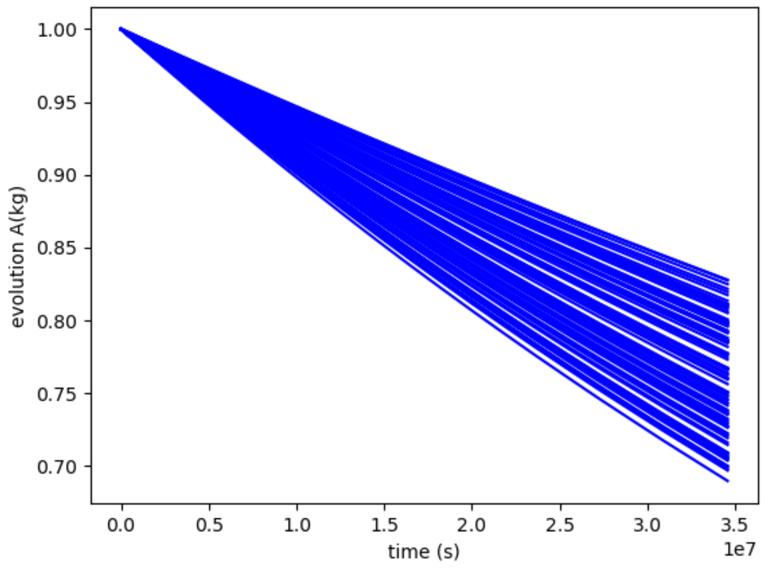


Figure 8. Plot of the histories generated by the Monte Carlo sampling for variable A.

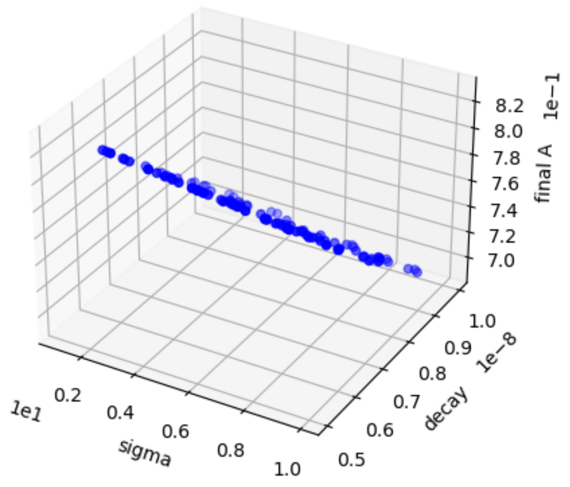


Figure 9. Plot of the samples generated by the MC sampling for variable A.

2.5 Build RAVEN Input: `<RomTrainer>`

The **RomTrainer** step type performs the training of a Reduced Order Model (ROM), and the specifications of this step must be defined within a `<RomTrainer>` block. ROMs, also known as a surrogate model, are used to lower the computational cost, reducing the number of needed points and prioritizing the area of the input space that needs to be explored when the simulations using the high-fidelity codes are very expensive. ROMs can be considered as an artificial representation of the link between the input and output spaces for a particular system.

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 reduced order model that is capable of representing the system behavior through a binary response (failure/success). Currently, RAVEN supports around 40 different ROM methodologies. All these supervised learning algorithms have been imported via an API from the Scikit-Learn 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.

In this section, the N-dimensional inverse weight method is employed to construct ROM to familarize the user with the use of ROMs. Inverse distance weighting (IDW) is a type of deterministic method for multivariate interpolation with a known scattered set of points. The assigned values to unknown points are calculated via a weighted average of the values available at the known points.
It is important to NOTE that RAVEN uses a Z-score normalization of the training data before constructing the *NDinvDistWeight* ROM:

$$\mathbf{X}' = \frac{(\mathbf{X} - \mu)}{\sigma} \tag{3}$$

2.5.1 How to train and output a ROM?

In general, the "training" is a process that use sampling of the physical model to improve the prediction capability of the ROM. As mentioned before, RAVEN provides lots of different sampling strategies, such as Monte Carlo, Grid, Stratified (e.g. LHS), and Stochastic Collocation methods. All of them can be used to train a ROM. In this section, we will continue to use **AnalyticBateman** to illustrate the setup of **RomTrainer**. As before, a simple step **MultiRun** with **Monte Carlo** sampler is employed to generate the data set that can be used to train the ROM. The full RAVEN input file can be found: *raven/tests/framework/user_guide/ravenTutorial/RomTrain.xml*. Because the setup of **MultiRun** is the same as previous example in section 2.4, the RAVEN input file about the **MultiRun** is not include in this section. However, the full precedures are listed here to make the user better understand the construction of ROM. The following precedures or RAVEN entities are needed:

- **MultiRun**: Monte Carlo sampling to generate the data set
  1. Set up the running environment: **<RunInfo>**;
  2. Provide the required files: **<Files>**;
  3. Link between RAVEN and dirven code: **<Code>**;
  4. Define probability distribution functions for inputs: **<Distributions>**;
  5. Set up a simple Monte-Carlo sampling for perturbing the input space: **<Samplers>**;
  6. Store the input and output data: **<DataObjects>**;
  7. Print and plot input and output data: **<OutStreams>**;
  8. Control multiple executions: **<MultiRun>**.
- **RomTrainer**: Train the ROM with given data set
  1. Specify the type of ROMs: **<ROM>**;
  2. Provide the data set for ROM training: **<DataObjects>**;
  3. Train the ROM: **<RomTrainer>**;
  4. Dump the ROM: **<IOStep>**

The specifications of the reduced order model must be defined within **<ROM>** XML block. This XML node accepts the following attributes:

- **name**, ***required string attribute***, user-defined identifier of this model.
- **subType**, ***required string attribute***, defines which of the sub-types should be used, choosing among the previously reported types.

In the **<ROM>** input block, the following XML sub-nodes are required, independent of the **subType** specified:

- **<Features>**, ***comma separated string, required field***, specifies the names of the features of this ROM. **Note:** These parameters are going to be requested for the training of this object;
- **<Target>**, ***comma separated string, required field***, contains a comma separated list of the targets of this ROM. These parameters are the Figures of Merit (FOMs) this ROM is supposed to predict. **Note:** These parameters are going to be requested for the training of this object.

For each sub-type specified in the attribute **subType**, additional sub-nodes may be required (Please check the RAVEN user manual for each ROM). In addition, if an **<HistorySet>** is provided in the training step, then a temporal ROM is created, i.e. a ROM that generates not a single value prediction of each element indicated in the **<Target>** block but its full temporal profile. In this section, a time-dependent ROM will be constructed.

In order to use **N-dimensional inverse distance weighting** ROM, the **<ROM>** attribute **subType** needs to be **'NDinvDistWeight'**. The following addition sub-node is also required.

- **<p>**, ***integer, required field***, must be greater than zero and represents the "power parameter". For the choice of value for **<p>**,it is necessary to consider the degree of smoothing desired in the interpolation/extrapolation, the density and distribution of samples being interpolated, and the maximum distance over which an individual sample is allowed to influence the surrounding ones (lower p means greater importance for points far away).

Based on previous RAVEN input file used in section 2.4, the **<ROM>**, **<RomTrain>** and **<IOStep>** are added. The **<ROM>** is used to describe the N-dimensional inverse distance weighting ROM:

raven/tests/framework/user_guide/ravenTutorial/RomTrain.xml

```
<Simulation>
  ...
  <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" />
    </Code>
    <ROM name="rom" subType="NDinvDistWeight">
      <Features>sigma-A,decay-A</Features>
      <Target>A, time</Target>
      <p>3</p>
    </ROM>
  </Models>
  ...
</Simulation>
```

The inputs and outputs of **AnalyticBateman** is used to generate the data set. The ROM will be constructed considering two features (*sigma-A and decay-A*) and two targets (*A and time*). **Note:** the *time* is treated as target in ROM construction.

Then, the **<RomTrain>** and **<IOStep>** are used to construct ROM on the fly or dump the ROM into a file (i.e. pickled ROM), respectively.

raven/tests/framework/user_guide/ravenTutorial/RomTrain.xml

```
<Simulation>
  ...
  <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="HistorySet">histories</Output>
    </MultiRun>
    <RomTrainer name="trainROM">
      <Input class="DataObjects" type="HistorySet">histories</Input>
      <Output class="Models" type="ROM">rom</Output>
    </RomTrainer>
    <IOStep name="dumpROM">
      <Input class="Models" type="ROM">rom</Input>
      <Output class="Files" type="">rom_inv</Output>
    </IOStep>
    <IOStep name="writeHistories" pauseAtEnd="True">
      <Input class="DataObjects" type="HistorySet">histories</Input>
      <Output class="OutStreams" type="Print">histories</Output>
    </IOStep>
  </Steps>
  ...
</Simulation>
```

The `'HistorySet'` generated by the **"sample"** step is used as input of the **"trainROM"** step. The output of the **"trainROM"** is "rom" which is predefined in **`<ROM>`**. If a `'PointSet'` is provided as input, the output "rom" will be time-independent. The **"dumpROM"** step is used to serialize the "rom" to file (pickled). In this example, the file is identified in **`<Files>`**.

raven/tests/framework/user_guide/ravenTutorial/RomTrain.xml

```
<Simulation>
  ...
  <Files>
    <Input name="referenceInput.xml" type="input">
          ../commonFiles/referenceInput_generic_CI.xml
      </Input>
    <Input name="rom_inv" type="">inverseRom.pk</Input>
  </Files>
  ...
</Simulation>
```

A pickled file with name "inverseRom.pk" will be generated in the working directory. This pickled ROM can be reused by RAVEN to perform additional analysis, and we will introduce this capability in the following section. Since we have four different steps to execute, the **`<RunInfo>`** block is modified:

raven/tests/framework/user_guide/ravenTutorial/RomTrain.xml

```
<Simulation>
  ...
  <RunInfo>
    <JobName>RomTrain</JobName>
    <Sequence>sample, trainROM, dumpROM, writeHistories</Sequence>
    <WorkingDir>ROM</WorkingDir>
    <batchSize>1</batchSize>
  </RunInfo>
  ...
</Simulation>
```

The **<Sequence>** is used to provide an ordered list of the step names that RAVEN will run.

2.5.2 How to load and sample a ROM?

In previous section, we have shown that RAVEN can be used to train a ROM and output the ROM to a pickled file. In this section, we will show the user how to load the pickled ROM, and how to reuse it inside RAVEN environment. In general, a **<ROM>** with subtype **'pickledROM'** is used to hold the place of the ROM that will be loaded from file. The notation for this ROM is much less than a typical ROM; it only requires a name and its subtype.

Example: For this example the ROM has already been created and trained in another RAVEN run, then pickled to a file called `rom_pickle.pk`. In the example, the file is identified in **<Files>**, the model is defined in **<Models>**, and the model loaded in **<Steps>**.

```
<Simulation>
  ...
  <Files>
    <Input name="rompk" type="">rom_pickle.pk</Input>
  </Files>
  ...
  <Models>
    ...
    <ROM name="myRom" subType="pickledROM"/>
    ...
  </Models>
  ...
  <Steps>
    ...
    <IOStep name="loadROM">
      <Input class="Files" type="">rompk</Input>
      <Output class="Models" type="ROM">myRom</Output>
    </IOStep>
    ...
  </Steps>
  ...
```

```
</Simulation>
```

Note: When loading ROMs from file, RAVEN will not perform any checks on the expected inputs or outputs of a ROM; it is expected that a user knows at least the I/O of a ROM before trying to use it as a model. However, RAVEN does require that pickled ROMs be trained before pickling in the first place.

Initially, a pickled ROM is not usable. It can not be trained or sampled; attempting to do so will raise an error. An `<IOStep>` is used to load the ROM from file, at which point the ROM will have all the same characteristics as when it was pickled in a previous RAVEN run. Take **AnalyticBateman** for example, the pickled ROM "inverseRom.pk" is generated in previous section and copied to the *commonFiles* folder, RAVEN use the **Files** object to track the pickled ROM file.

raven/tests/framework/user_guide/ravenTutorial/RomLoad.xml

```
<Simulation>
  ...
  <Files>
    <Input name="rom_inv" type="">../commonFiles/inverseRom.pk</Input>
  </Files>
  ...
</Simulation>
```

In this example, the subtype `'pickledROM'` of `<ROM>` is used since the hyper-parameters of the ROM can not be changed once the ROM is loaded from a pickled (serialized) file.

raven/tests/framework/user_guide/ravenTutorial/RomLoad.xml

```
<Simulation>
  ...
  <Models>
    <ROM name="rom" subType="pickledROM" />
  </Models>
  ...
</Simulation>
```

Two data objects are defined: 1) a **HistorySet** named "inputPlaceHolder" used as a place-holder input for the ROM sampling step, 2) a **HistorySet** named "histories" used to store the ROM responses from Monte Carlo samples.

raven/tests/framework/user_guide/ravenTutorial/RomLoad.xml

```
<Simulation>
  ...
  <DataObjects>
```

```
    <PointSet name="inputPlaceHolder">
      <Input>sigma-A,decay-A</Input>
      <Output>OutputPlaceHolder</Output>
    </PointSet>
    <HistorySet name="histories">
      <Input>sigma-A,decay-A</Input>
      <Output>A, time</Output>
      <options>
        <pivotParameter>time</pivotParameter>
      </options>
    </HistorySet>
  </DataObjects>
  ...
</Simulation>
```

Note: In this example, a time-dependent ROM trained in previous case is used here. `'time'` is identified as **Output**. The sub-node `<pivotParameter>` can be used to define the pivot variable (e.g. time) that is non-decreasing in the input HistorySet.

As mentioned before, the `<IOStep>` is used to load the pickled ROM. In addition, the `<MultiRun>` is used to sample the ROM using Monte Carlo method. Another `<IOStep>` is used to output the responses of ROM into a CSV file. Figure 10 shows the different evolutions of the variable A for all 10 samples.

raven/tests/framework/user_guide/ravenTutorial/RomLoad.xml

```
<Simulation>
  ...
  <Steps>
    <IOStep name="loadROM">
      <Input class="Files" type="">rom_inv</Input>
      <Output class="Models" type="ROM">rom</Output>
    </IOStep>
    <MultiRun name="sampleROM">
      <Input class="DataObjects" type="PointSet">inputPlaceHolder</Input>
      <Model class="Models" type="ROM">rom</Model>
      <Sampler class="Samplers" type="MonteCarlo">monteCarlo</Sampler>
      <Output class="DataObjects" type="HistorySet">histories</Output>
    </MultiRun>
    <IOStep name="writeHistories" pauseAtEnd="True">
      <Input class="DataObjects" type="HistorySet">histories</Input>
      <Output class="OutStreams" type="Plot">historyROMPlot</Output>
      <Output class="OutStreams" type="Print">historiesROM</Output>
    </IOStep>
  </Steps>
  ...
</Simulation>
```

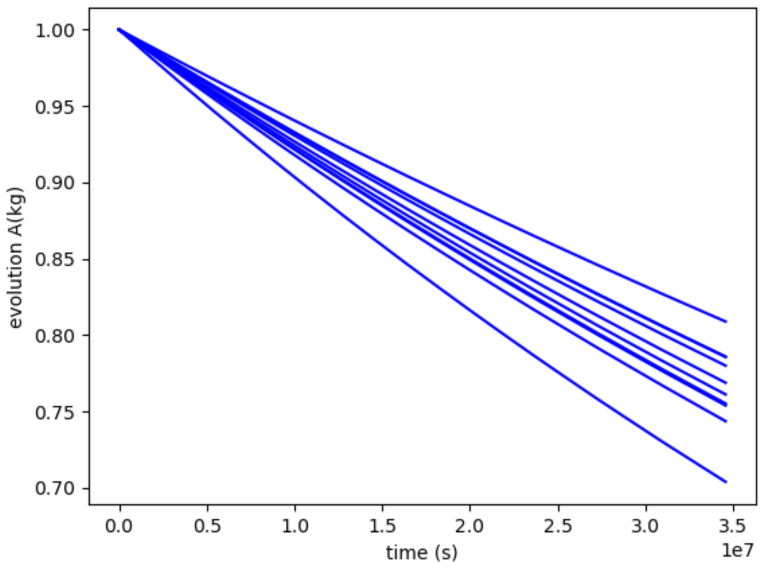


Figure 10. Plot of the histories generated by the Monte Carlo sampling of pickled ROM for variable A (10 samples).

Finally, ROMs are generally not constructed for all possible inputs, geometries or representing all outputs. However, it is possible to build a ROM of faster solution with respect to the original set. The accuracy in the prediction can be obtained by further training. Figure 11 shows the general workflow for ROM construction.

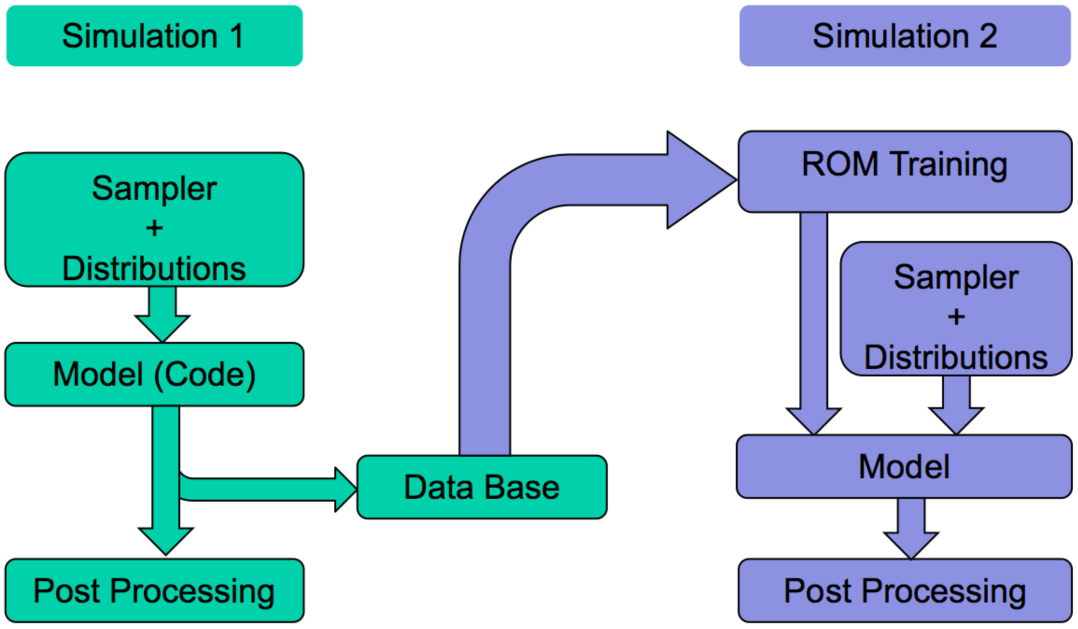


Figure 11. Workflow for ROM construction

2.6 Build RAVEN Input: `<PostProcess>`

The **PostProcess** step is used to post-process data or manipulate RAVEN entities. It is intended to perform a single action that is employed by a **Model** of type **PostProcessor**. The specification of this type of step is defined within a `<PostProcess>` XML block. As for the other objects, the attribute `name` is required and is used to refer to this specific entity in the `<RunInfo>` block under the `<Sequence>` node.

In the `<PostProcess>` input block, the user needs to specify the objects needed for the different allowable roles. This step requires the following roles:

- `<Input>`: accepts **Files**, **DataObjects** or **Databases**.
- `<Model>`: Only `'Models'` and `'PostProcessor'` can be assigned to the node's attributes `class` and `type`, respectively.
- `<Output>`: accepts **Files**, **DataObjects**, **Databases** or **OutStreams**.

As mentioned before, only the model with type **PostProcessor** is allowed for this step. A post-processor can be considered as an action performed on a set of data or other type of objects. Most of the post-processors contained in RAVEN, employ a mathematical operation on the data given as **"Input"**. Currently, the following **PostProcessor** are available in RAVEN:

- **BasicStatistics**
- **ComparisonStatistics**
- **ImportanceRank**
- **SafestPoint**
- **LimitSurface**
- **LimitSurfaceIntegral**
- **External**
- **TopologicalDecomposition**
- **RavenOutput**
- **DataMining**

One can use the node attribute `subType` to select which of the post-processors to be used. As with other objects, the attribute `name` is always required so that other RAVEN input XML blocks can use this name to refer to this specific entity. In addition, each post-processor may require extra sub-nodes, and the user can refer to the RAVEN user manual for the detailed specifications.

In this example, the **BasicStatistics** post-processor is used to demonstrate the **PostProcess** step. **BasicStatistics** is a container of the algorithms to compute many of the most important statistical quantities. Both **PointSet** and **HistorySet** can be accepted to compute the static statistics and dynamic statistics, respectively. In case an **HistorySet** is provided as **Input**, the user need to define the **pivotParameter**, and sometimes the user need to synchronize the **HistorySet** first via the **Interfaced** post-processor of type **HistorySetSync**.

raven/tests/framework/user_guide/ravenTutorial/PostProcess.xml

```
<Simulation>
  ...
  <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" />
    </Code>
    <PostProcessor name="statisticalAnalysis" subType="BasicStatistics">
      <pivotParameter>time</pivotParameter>
      <skewness prefix="skew">A</skewness>
      <variationCoefficient prefix="vc">A</variationCoefficient>
      <percentile prefix="percentile">A</percentile>
      <expectedValue prefix="mean">A</expectedValue>
      <kurtosis prefix="kurt">A</kurtosis>
      <median prefix="median">A</median>
      <maximum prefix="max">A</maximum>
      <minimum prefix="min">A</minimum>
      <samples prefix="samp">A</samples>
      <variance prefix="var">A</variance>
      <sigma prefix="sigma">A</sigma>
      <NormalizedSensitivity prefix="nsen">
        <targets>A</targets>
        <features>sigma-A,decay-A</features>
      </NormalizedSensitivity>
      <sensitivity prefix="sen">
        <targets>A</targets>
        <features>sigma-A,decay-A</features>
      </sensitivity>
      <pearson prefix="pear">
        <targets>A</targets>
        <features>sigma-A,decay-A</features>
      </pearson>
      <covariance prefix="cov">
        <targets>A</targets>
        <features>sigma-A,decay-A</features>
      </covariance>
      <VarianceDependentSensitivity prefix="vsen">
        <targets>A</targets>
        <features>sigma-A,decay-A</features>
      </VarianceDependentSensitivity>
    </PostProcessor>
  </Models>
  ...
</Simulation>
```

In this example, all the metrics of **BasicStatistics** will be computed for the response A.

The **<Files>** will be used to include all input and output files. In this example, a single input file for the driven code and two output files of the **PostProcess** step are defined here. As shown in the following, two output files are defined for this case study to store the static statistics and dynamic statistics information. The `'time'` is used as the `<pivotParameter>`.

raven/tests/framework/user_guide/ravenTutorial/PostProcess.xml

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

As before, all defined RAVEN entities are combined in the `<Steps>` block.

raven/tests/framework/user_guide/ravenTutorial/PostProcess.xml

```
<Simulation>
  ...
  <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="statAnalysis_1">
      <Input class="DataObjects" type="PointSet">samplesMC</Input>
      <Model class="Models" type="PostProcessor">statisticalAnalysis</Model>
      <Output class="DataObjects" type="PointSet">statisticalAnalysis_basicStatPP</Output>
      <Output class="OutStreams" type="Print">statisticalAnalysis_basicStatPP_dump</Output>
    </PostProcess>
    <PostProcess name="statAnalysis_2">
      <Input class="DataObjects" type="HistorySet">histories</Input>
      <Model class="Models" type="PostProcessor">statisticalAnalysis</Model>
      <Output class="DataObjects" type="HistorySet">statisticalAnalysis_basicStatPP_time</Output>
      <Output class="OutStreams" type="Print">statisticalAnalysis_basicStatPP_time_dump</Output>
    </PostProcess>
  </Steps>
  ...
</Simulation>
```

In this case, three steps have been defined:

- `<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 Monte Carlo sampling strategy.
- `<PostProcess>` named "statAnalysis_1", is used to compute all the statistical moments and FOMs based on the data obtained through the sampling strategy. As it can be noticed, the `'PointSet'` "samplesMC" is used as input to compute the static statistics.

- **<PostProcess>** named “statAnalysis_2”, is used to compute all the statistical moments and FOMs based on the data obtained through the sampling strategy. As it can be noticed, the **'HistorySet'** “histories” is used as input to compute the dynamic statistics.

3 Forward Sampling Strategies

In order to perform UQ and dynamic probabilistic risk assessment (DPRA), a sampling strategy needs to be employed. The sampling strategy perturbs 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 in RAVEN as ***Forward*** samplers. ***Forward*** samplers include all the strategies that simply perform the sampling of the input space. These strategies sample 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 2.4, RAVEN has several different ***Forward*** samplers:

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

In addition, RAVEN posses 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 interactions with increasing complexity (*Sobol* sampler).

In the following subsections, we provide examples of input files in RAVEN using the method, with explanatory commentary.

3.1 Monte-Carlo sampling through RAVEN

The Monte-Carlo method is one of the most-used methodologies in several mathematic disciplines. In this section, we will explain the techniques for employing this methodology in RAVEN, and we recommend the user to read the theory manual to explore the theory of the method. 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 RAVEN DataObjects (HistorySets and PointSets)

3. Print the contents of DataObjects to file

4. Generate plots of the sampling results.

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

1. ***RunInfo***:

   raven/tests/framework/user_guide/ForwardSamplingStrategies/forwardSamplingMontecarlo.xml

   ```
   <Simulation>
     ...
     <RunInfo>
       <JobName>RunDir/MonteCarlo</JobName>
       <Sequence>sample,writeHistories</Sequence>
       <WorkingDir>RunDir/MonteCarlo</WorkingDir>
       <batchSize>1</batchSize>
     </RunInfo>
     ...
   </Simulation>
   ```

   As discussed in Section 2.2, the *RunInfo* **Entity** sets up the analysis that the user wants to perform. The number of steps specified in (`<Sequence>`) are sequentially run using the number of processors assigned in (`<batchSize>`). Note that the `<JobName>` is not required, but is useful in identifying the input file.

2. ***Models***:

   raven/tests/framework/user_guide/ForwardSamplingStrategies/forwardSamplingMontecarlo.xml

   ```
   <Simulation>
     ...
     <Models>
       <ExternalModel subType="" name="projectile" ModuleToLoad="../../AnalyticModels/projectile.py">
         <variables>x,y,v0,angle,r,t,timeOption</variables>
       </ExternalModel>
     </Models>
     ...
   </Simulation>
   ```

   The Model used in this example is the **Projectile** external model, which is defined in section 2.1.

3. ***Distributions***:

   raven/tests/framework/user_guide/ForwardSamplingStrategies/forwardSamplingMontecarlo.xml

```xml
<Simulation>
  ...
  <Distributions>
    <Normal name="vel_dist">
      <mean>30</mean>
      <sigma>5</sigma>
      <lowerBound>1</lowerBound>
      <upperBound>60</upperBound>
    </Normal>
    <Uniform name="angle_dist">
      <lowerBound>5</lowerBound>
      <upperBound>85</upperBound>
    </Uniform>
  </Distributions>
  ...
</Simulation>
```

In the `<Distributions>` block, the stochastic model for the uncertainties treated by the `<Sampler>` is defined. In this case two distributions are defined:

- $vel\_dist \sim \mathbb{N}(30, 5)$, used to model the uncertainties associated with the *velocity*;
- $angle\_dist \sim \mathbb{U}(5, 85)$, used to model the uncertainties associated with the *angle*.

4. ***Samplers***:

`raven/tests/framework/user_guide/ForwardSamplingStrategies/forwardSamplingMontecarlo.xml`

```xml
<Simulation>
  ...
  <Samplers>
    <MonteCarlo name="monteCarlo">
      <samplerInit>
        <limit>500</limit>
        <reseedEachIteration>True</reseedEachIteration>
        <initialSeed>0</initialSeed>
      </samplerInit>
      <variable name="v0">
        <distribution>vel_dist</distribution>
      </variable>
      <variable name="angle">
        <distribution>angle_dist</distribution>
      </variable>
      <constant name="x0">0</constant>
      <constant name="y0">0</constant>
      <constant name="timeOption">0</constant>
    </MonteCarlo>
  </Samplers>
  ...
</Simulation>
```

To employ the Monte-Carlo sampling strategy, a `<MonteCarlo>` node needs to be defined. The number of samples is defined within this node. The Monte-Carlo method is employed on model variables listed by name and are associated with a distribution.

5. ***DataObjects***:

raven/tests/framework/user_guide/ForwardSamplingStrategies/forwardSamplingMontecarlo.xml

```
<Simulation>
  ...
  <DataObjects>
    <PointSet name="samples">
      <Input>v0,angle</Input>
      <Output>r,t</Output>
    </PointSet>
    <PointSet name="dummyIN">
      <Input>v0,angle</Input>
      <Output>OutputPlaceHolder</Output>
    </PointSet>
    <HistorySet name="histories">
      <Input>v0,angle</Input>
      <Output>x,y,r,t</Output>
      <options>
        <pivotParameter> t </pivotParameter>
      </options>
    </HistorySet>
  </DataObjects>
  ...
</Simulation>
```

In this block, three *DataObjects* are defined to store results: 1) a PointSet named "samples", 2) a PointSet named "dummyIN" 3) a 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 in the DataObject to the outputs listed in the `<Output>` node. Furthermore, since we use an external model that does not have any input file, we define a pointset named "dummyIN" that is used as a dummy input in the multirun step.

6. ***OutStreams***:

raven/tests/framework/user_guide/ForwardSamplingStrategies/forwardSamplingMontecarlo.xml

```
<Simulation>
  ...
  <OutStreams>
    <Print name="samples">
      <type>csv</type>
      <source>samples</source>
    </Print>
    <Print name="histories">
      <type>csv</type>
      <source>histories</source>
    </Print>
    <Plot name="historyPlot" overwrite="false" verbosity="debug">
      <plotSettings>
        <gridSpace>2 1</gridSpace>
        <plot>
          <type>scatter</type>
          <x>histories|Input|v0</x>
          <y>histories|Output|r</y>
          <kwargs>
            <color>blue</color>
          </kwargs>
          <gridLocation>
            <x>0</x>
            <y>0</y>
          </gridLocation>
          <xlabel>velocity</xlabel>
          <ylabel>range</ylabel>
        </plot>
        <plot>
          <type>scatter</type>
          <x>histories|Input|angle</x>
          <y>histories|Output|r</y>
          <kwargs>
            <color>orange</color>
          </kwargs>
          <gridLocation>
            <x>1</x>
            <y>0</y>
          </gridLocation>
          <xlabel>angle</xlabel>
          <ylabel>range</ylabel>
        </plot>
      </plotSettings>
      <actions>
        <how>png</how>
        <title>
          <text> </text>
        </title>
      </actions>
    </Plot>
    <Plot name="samplesPlot3D" overwrite="false" verbosity="debug">
      <plotSettings>
        <gridSpace>2 1</gridSpace>
        <plot>
          <type>scatter</type>
          <x>samples|Input|v0</x>
          <y>samples|Input|angle</y>
          <z>samples|Output|r</z>
          <c>blue</c>
          <gridLocation>
            <x>0</x>
            <y>0</y>
          </gridLocation>
          <xlabel>velocity</xlabel>
          <ylabel>angle</ylabel>
          <zlabel>range</zlabel>
        </plot>
        <plot>
          <type>scatter</type>
          <x>samples|Input|v0</x>
          <y>samples|Input|angle</y>
          <z>samples|Output|t</z>
          <c>orange</c>
          <gridLocation>
            <x>1</x>
            <y>0</y>
          </gridLocation>
          <xlabel>velocity</xlabel>
          <ylabel>angle</ylabel>
          <zlabel>time</zlabel>
        </plot>
      </plotSettings>
      <actions>
        <how>png</how>
        <title>
          <text> </text>
        </title>
      </actions>
    </Plot>
  </OutStreams>
  ...
</Simulation>
```

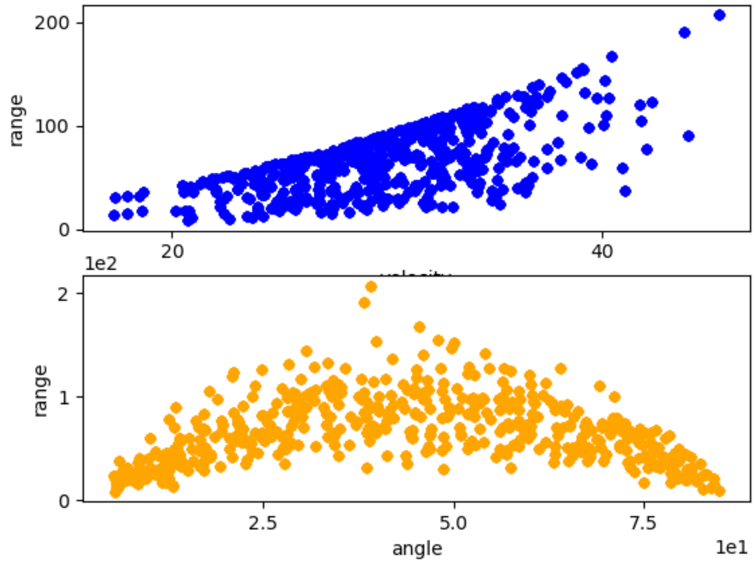


Figure 12. Plot of the histories generated by the Monte Carlo sampling.

To see the results of the simulation, **`<OutStreams>`** are included in the input. In this block, both OutStream types are used:

- *Print*:
  - "samples" connected with the *DataObjects* **Entity** "samples" (**`<source>`**)
  - "histories" connected with the *DataObjects* **Entity** "histories" (**`<source>`**)

  Note that in RAVEN, multiple entities can have the same name, as it takes a class, a type, and a name to uniquely identify a RAVEN object. When the two OutStream objects are used, all the information contained in the linked *DataObjects* are going to be exported in CSV files (**`<type>`**).
- *Plot*:
  - "historiesPlot" connected with the *DataObjects* **Entity** "histories". This plot shows the variable $range$ with respect to the input variables $velocity$ and $angle$.
  - "samplesPlot3D" connected with the *DataObjects* **Entity** "samples". This plot shows the variables $range, time$ with respect to the input variables $velocity$ and $angle$.

Note that both plots use gridded subplots. Two plots are placed in each of the figures.

7. ***Steps***:

raven/tests/framework/user_guide/ForwardSamplingStrategies/forwardSamplingMontecarlo.xml

```
<Simulation>
  ...
  <Steps>
    <MultiRun name="sample">
      <Input class="DataObjects" type="PointSet">dummyIN</Input>
      <Model class="Models" type="ExternalModel">projectile</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>
  ...
</Simulation>
```

Once all the other entities are defined in the RAVEN input file, they must be combined in the **<Steps>** block, which dictates the workflow of RAVEN. For this case, two **<Steps>** are defined:

- **<MultiRun>** "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 specified 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** to a CSV file and 2) plot the data in the EPS file.

Figures 12 and 13 show the report generated by RAVEN.

3.2 Grid sampling through RAVEN

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. The goal of this section is to show how to:

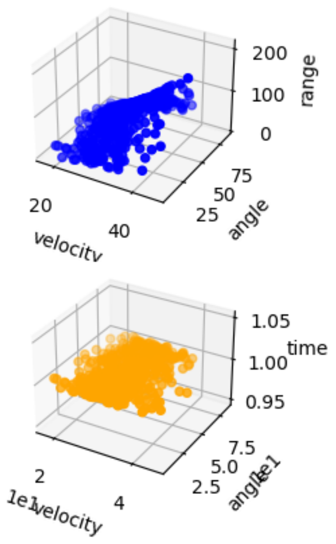


Figure 13. Plot of the samples generated by the MC sampling.

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***:

   raven/tests/framework/user_guide/ForwardSamplingStrategies/forwardSamplingGrid.xml

```
<Simulation>
  ...
  <RunInfo>
    <JobName>RunDir/Grid</JobName>
    <Sequence>sample,writeHistories</Sequence>
    <WorkingDir>RunDir/Grid</WorkingDir>
    <batchSize>1</batchSize>
  </RunInfo>
  ...
</Simulation>
```

As shown in Section 2.2, the *RunInfo* **Entity** is intended to set up the desired analysis. The number of steps specified in (**<Sequence>**) are sequentially run using the number of processors assigned in (**<batchSize>**).

2. ***Models***:
   raven/tests/framework/user_guide/ForwardSamplingStrategies/forwardSamplingGrid.xml

```
<Simulation>
  ...
  <Models>
    <ExternalModel subType="" name="projectile" ModuleToLoad="../../AnalyticModels/projectile.py">
      <variables>x,y,v0,angle,r,t,timeOption</variables>
    </ExternalModel>
  </Models>
  ...
</Simulation>
```

   The Model here is represented by the **Projectile**, which already dumps its output file in a CSV format (standard format that RAVEN can read).

3. ***Distributions***:

   raven/tests/framework/user_guide/ForwardSamplingStrategies/forwardSamplingGrid.xml

```
<Simulation>
  ...
  <Distributions>
    <Normal name="vel_dist">
      <mean>30</mean>
      <sigma>5</sigma>
      <lowerBound>1</lowerBound>
      <upperBound>60</upperBound>
    </Normal>
    <Uniform name="angle_dist">
      <lowerBound>5</lowerBound>
      <upperBound>85</upperBound>
    </Uniform>
  </Distributions>
  ...
</Simulation>
```

   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:

   - $vel\_dist \sim \mathbb{N}(30, 5)$, used to model the uncertainties associated with the *velocity*;
   - $angle\_dist \sim \mathbb{U}(5, 85)$, used to model the uncertainties associated with the *angle*.

4. ***Samplers***:

raven/tests/framework/user_guide/ForwardSamplingStrategies/forwardSamplingGrid.xml

```
<Simulation>
  ...
  <Samplers>
    <Grid name="grid">
      <variable name="v0">
        <distribution>vel_dist</distribution>
        <grid construction="equal" steps="2" type="CDF">.1 0.85</grid>
      </variable>
      <variable name="angle">
        <distribution>angle_dist</distribution>
        <grid construction="equal" steps="3" type="CDF">0.15  0.9</grid>
      </variable>
      <constant name="x0">0</constant>
      <constant name="y0">0</constant>
      <constant name="timeOption">0</constant>
    </Grid>
  </Samplers>
  ...
</Simulation>
```

To employ the Grid sampling strategy, a **<Grid>** node needs to be specified. As shown above, in each variable section, the **<grid>** is defined.

5. ***DataObjects***:

raven/tests/framework/user_guide/ForwardSamplingStrategies/forwardSamplingGrid.xml

```
<Simulation>
  ...
  <DataObjects>
    <PointSet name="samples">
      <Input>v0,angle</Input>
      <Output>r,t</Output>
    </PointSet>
    <PointSet name="dummyIN">
      <Input>v0,angle</Input>
      <Output>OutputPlaceHolder</Output>
    </PointSet>
    <HistorySet name="histories">
      <Input>v0,angle</Input>
      <Output>x,y,r,t</Output>
      <options>
        <pivotParameter> t </pivotParameter>
      </options>
    </HistorySet>
  </DataObjects>
  ...
</Simulation>
```

In this block, three *DataObjects* are defined to store results: 1) a PointSet named “samples”, 2) a PointSet named “dummyIN” 3) a 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. As described earlier as well, since we use an external model that does not have any input file, we define a pointset named “dummyIN” that is used as a dummy input in the multirun step.

6. ***OutStreams***:

raven/tests/framework/user_guide/ForwardSamplingStrategies/forwardSamplingGrid.xml

```
<Simulation>
  ...
  <OutStreams>
    <Print name="samples">
      <type>csv</type>
      <source>samples</source>
    </Print>
    <Print name="histories">
      <type>csv</type>
      <source>histories</source>
    </Print>
    <Plot name="historyPlot" overwrite="false" verbosity="debug">
      <plotSettings>
        <gridSpace>2 1</gridSpace>
        <plot>
          <type>scatter</type>
          <x>histories|Input|v0</x>
          <y>histories|Output|r</y>
          <kwargs>
            <color>blue</color>
          </kwargs>
          <gridLocation>
            <x>0</x>
            <y>0</y>
          </gridLocation>
          <xlabel>velocity</xlabel>
          <ylabel>range</ylabel>
        </plot>
        <plot>
          <type>scatter</type>
          <x>histories|Input|angle</x>
          <y>histories|Output|r</y>
          <kwargs>
            <color>orange</color>
          </kwargs>
          <gridLocation>
            <x>1</x>
            <y>0</y>
          </gridLocation>
          <xlabel>angle</xlabel>
          <ylabel>range</ylabel>
        </plot>
      </plotSettings>
      <actions>
        <how>png</how>
        <title>
          <text> </text>
        </title>
      </actions>
    </Plot>
    <Plot name="samplesPlot3D" overwrite="false" verbosity="debug">
      <plotSettings>
        <gridSpace>2 1</gridSpace>
        <plot>
          <type>scatter</type>
          <x>samples|Input|v0</x>
          <y>samples|Input|angle</y>
          <z>samples|Output|r</z>
          <c>blue</c>
          <gridLocation>
            <x>0</x>
            <y>0</y>
          </gridLocation>
          <xlabel>velocity</xlabel>
          <ylabel>angle</ylabel>
          <zlabel>range</zlabel>
        </plot>
        <plot>
          <type>scatter</type>
          <x>samples|Input|v0</x>
          <y>samples|Input|angle</y>
          <z>samples|Output|t</z>
          <c>orange</c>
          <gridLocation>
            <x>1</x>
            <y>0</y>
          </gridLocation>
          <xlabel>velocity</xlabel>
          <ylabel>angle</ylabel>
          <zlabel>time</zlabel>
        </plot>
      </plotSettings>
      <actions>
        <how>png</how>
        <title>
          <text> </text>
        </title>
      </actions>
    </Plot>
  </OutStreams>
  ...
</Simulation>
```

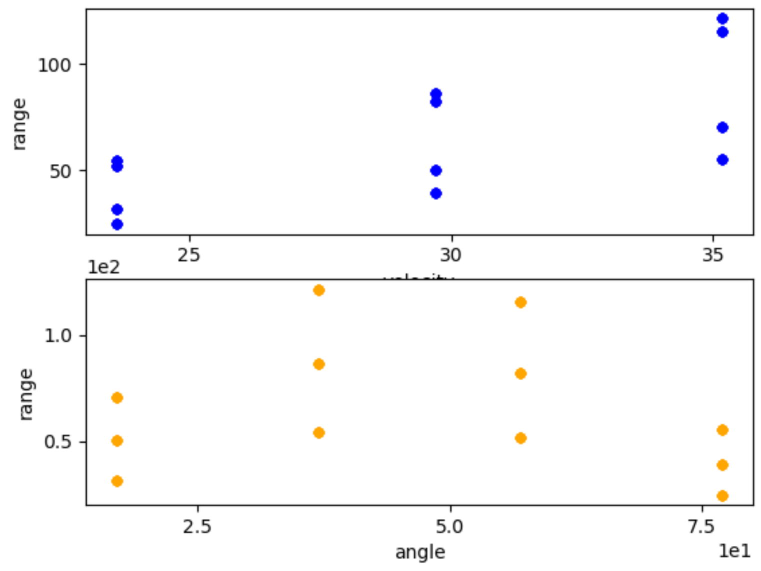


Figure 14. Plot of the histories generated by the Grid sampling.

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** "histories". This plot shows the variable $range$ with respect to the input variables $velocity$ and $angle$.
  - named "samplesPlot3D" connected with the *DataObjects* **Entity** "samples". This plot shows the variables $range, time$ with respect to the input variables $velocity$ and $angle$.

7. ***Steps***:

raven/tests/framework/user_guide/ForwardSamplingStrategies/forwardSamplingGrid.xml

```
<Simulation>
  ...
  <Steps>
    <MultiRun name="sample">
      <Input class="DataObjects" type="PointSet">dummyIN</Input>
      <Model class="Models" type="ExternalModel">projectile</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>
  ...
</Simulation>
```

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 14 and 15 display the report generated by RAVEN.

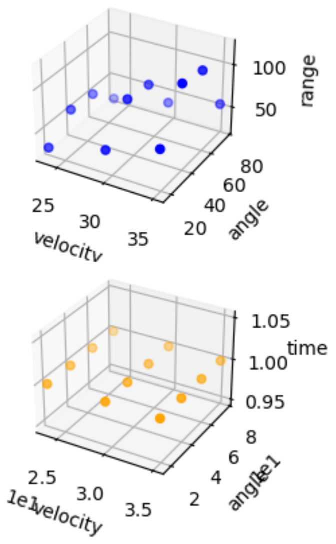


Figure 15. Plot of the samples generated by the Grid sampling for variables A, B, C, D.

3.3 Stratified sampling through RAVEN

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 16, the Stratified sampling approach is exemplified.
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.

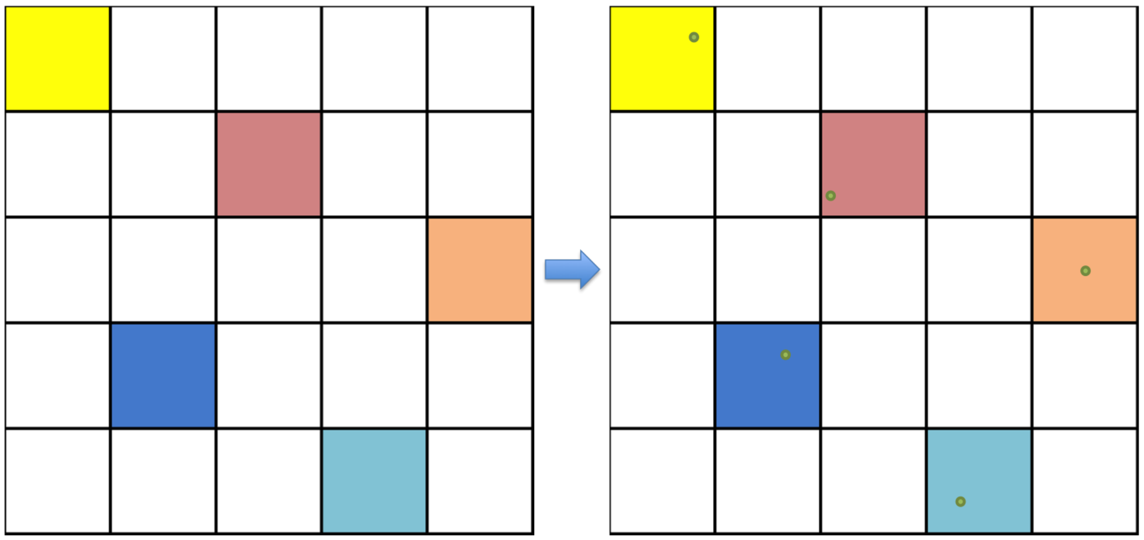


Figure 16. Example of Stratified sampling approach.

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

1. ***RunInfo***:

   raven/tests/framework/user_guide/ForwardSamplingStrategies/forwardSamplingStratified.xml

```
<Simulation>
  ...
  <RunInfo>
    <JobName>RunDir/Stratified</JobName>
    <Sequence>sample,writeHistories</Sequence>
    <WorkingDir>RunDir/Stratified</WorkingDir>
    <batchSize>1</batchSize>
  </RunInfo>
  ...
</Simulation>
```

   As explained earlier, the *RunInfo* **Entity** is intended to set up the analysis that the user wants to perform. The number of steps specified in (`<Sequence>`) are sequentially run using the number of processors assigned in (`<batchSize>`).

2. ***Models***:

   raven/tests/framework/user_guide/ForwardSamplingStrategies/forwardSamplingStratified.xml

```
<Simulation>
  ...
  <Models>
    <ExternalModel subType="" name="projectile" ModuleToLoad="../../AnalyticModels/projectile.py">
      <variables>x,y,v0,angle,r,t,timeOption</variables>
    </ExternalModel>
  </Models>
  ...
</Simulation>
```

The Model here is represented by the **Projectile**, which already dumps its output file in a CSV format (standard format that RAVEN can read).

3. ***Distributions***:

   raven/tests/framework/user_guide/ForwardSamplingStrategies/forwardSamplingStratified.xml

```
<Simulation>
  ...
  <Distributions>
    <Normal name="vel_dist">
      <mean>30</mean>
      <sigma>5</sigma>
      <lowerBound>1</lowerBound>
      <upperBound>60</upperBound>
    </Normal>
    <Uniform name="angle_dist">
      <lowerBound>5</lowerBound>
      <upperBound>85</upperBound>
    </Uniform>
  </Distributions>
  ...
</Simulation>
```

   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:

   - $vel\_dist \sim \mathbb{N}(30, 5)$, used to model the uncertainties associated with the *velocity*;
   - $angle\_dist \sim \mathbb{U}(5, 85)$, used to model the uncertainties associated with the *angle*.

4. ***Samplers***:

   raven/tests/framework/user_guide/ForwardSamplingStrategies/forwardSamplingStratified.xml

```
<Simulation>
  ...
  <Samplers>
    <Stratified name="stratified">
      <samplerInit>
        <initialSeed>42</initialSeed>
      </samplerInit>
      <variable name="v0">
        <distribution>vel_dist</distribution>
        <grid construction="equal" steps="100" type="CDF">.1 0.85</grid>
      </variable>
      <variable name="angle">
        <distribution>angle_dist</distribution>
        <grid construction="equal" steps="100" type="CDF">0.15
           0.9</grid>
      </variable>
      <constant name="x0">0</constant>
      <constant name="y0">0</constant>
      <constant name="timeOption">0</constant>
    </Stratified>
  </Samplers>
  ...
</Simulation>
```

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.

5. ***DataObjects***:

raven/tests/framework/user_guide/ForwardSamplingStrategies/forwardSamplingStratified.xml

```
<Simulation>
  ...
  <DataObjects>
    <PointSet name="samples">
      <Input>v0,angle</Input>
      <Output>r,t</Output>
    </PointSet>
    <PointSet name="dummyIN">
      <Input>v0,angle</Input>
      <Output>OutputPlaceHolder</Output>
    </PointSet>
    <HistorySet name="histories">
      <Input>v0,angle</Input>
      <Output>x,y,r,t</Output>
      <options>
        <pivotParameter> t </pivotParameter>
      </options>
    </HistorySet>
  </DataObjects>
  ...
```

```
</Simulation>
```

In this block, two *DataObjects* are defined: 1) a PointSet named “samples”, 2) a PointSet named “dummyIN” 3) a 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. Since we use an external model that does not have any input file, we define a pointset named “dummyIN” that is used as a dummy input in the multirun step.

6. ***OutStreams***:

raven/tests/framework/user_guide/ForwardSamplingStrategies/forwardSamplingStratified.xml

```
<Simulation>
  ...
  <OutStreams>
    <Print name="samples">
      <type>csv</type>
      <source>samples</source>
    </Print>
    <Print name="histories">
      <type>csv</type>
      <source>histories</source>
    </Print>
    <Plot name="historyPlot" overwrite="false" verbosity="debug">
      <plotSettings>
        <gridSpace>2 1</gridSpace>
        <plot>
          <type>scatter</type>
          <x>histories|Input|v0</x>
          <y>histories|Output|r</y>
          <kwargs>
            <color>blue</color>
          </kwargs>
          <gridLocation>
            <x>0</x>
            <y>0</y>
          </gridLocation>
          <xlabel>velocity</xlabel>
          <ylabel>range</ylabel>
        </plot>
        <plot>
          <type>scatter</type>
          <x>histories|Input|angle</x>
          <y>histories|Output|r</y>
          <kwargs>
            <color>orange</color>
          </kwargs>
          <gridLocation>
            <x>1</x>
            <y>0</y>
          </gridLocation>
          <xlabel>angle</xlabel>
          <ylabel>range</ylabel>
        </plot>
      </plotSettings>
      <actions>
        <how>png</how>
        <title>
          <text> </text>
        </title>
      </actions>
    </Plot>
    <Plot name="samplesPlot3D" overwrite="false" verbosity="debug">
      <plotSettings>
        <gridSpace>2 1</gridSpace>
        <plot>
          <type>scatter</type>
          <x>samples|Input|v0</x>
          <y>samples|Input|angle</y>
          <z>samples|Output|r</z>
          <c>blue</c>
          <gridLocation>
            <x>0</x>
            <y>0</y>
          </gridLocation>
          <xlabel>velocity</xlabel>
          <ylabel>angle</ylabel>
          <zlabel>range</zlabel>
        </plot>
        <plot>
          <type>scatter</type>
          <x>samples|Input|v0</x>
          <y>samples|Input|angle</y>
          <z>samples|Output|t</z>
          <c>orange</c>
          <gridLocation>
            <x>1</x>
            <y>0</y>
          </gridLocation>
          <xlabel>velocity</xlabel>
```

```
          <ylabel>angle</ylabel>
          <zlabel>time</zlabel>
        </plot>
      </plotSettings>
      <actions>
        <how>png</how>
        <title>
          <text> </text>
        </title>
      </actions>
    </Plot>
  </OutStreams>
  ...
</Simulation>
```

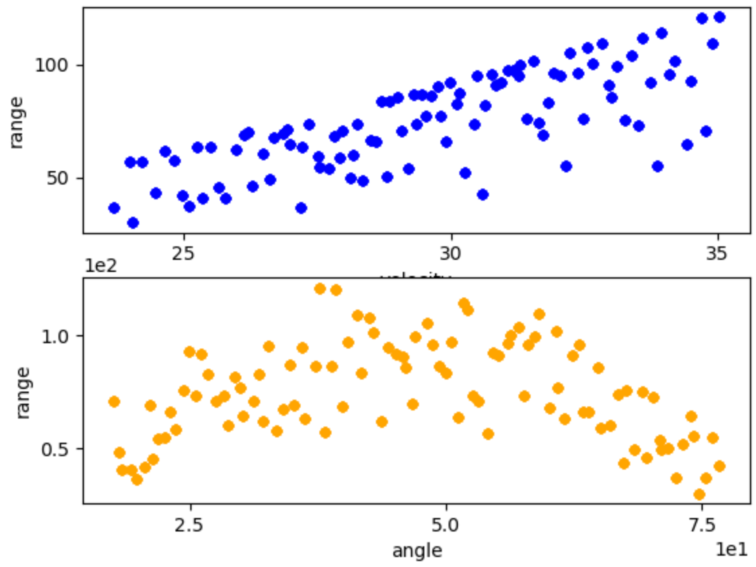


Figure 17. Plot of the histories generated by the Stratified sampling.

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** "histories". This plot shows the variable $range$ with respect to the input variables $velocity$ and $angle$.
  - named "samplesPlot3D" connected with the *DataObjects* **Entity** "samples". This plot shows the variables $range, time$ with respect to the input variables $velocity$ and $angle$.

7. ***Steps***:

raven/tests/framework/user_guide/ForwardSamplingStrategies/forwardSamplingStratified.xml

```
<Simulation>
  ...
  <Steps>
    <MultiRun name="sample">
      <Input class="DataObjects" type="PointSet">dummyIN</Input>
      <Model class="Models" type="ExternalModel">projectile</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>
  ...
</Simulation>
```

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 17 and **??** display the report generated by RAVEN.

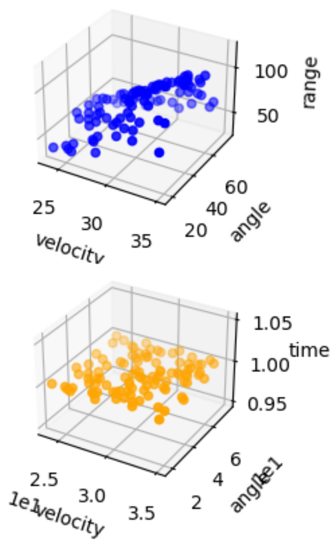


Figure 18. Plot of the samples generated by the Stratified sampling.

3.4 Sparse Grid Collocation sampling through RAVEN

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 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.
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***:

   raven/tests/framework/user_guide/ForwardSamplingStrategies/forwardSamplingSparseGrid.xml

```
<Simulation>
  ...
  <RunInfo>
    <JobName>RunDir/SparseGrid</JobName>
    <WorkingDir>RunDir/SparseGrid</WorkingDir>
    <Sequence>sample,train,validateModel,validateROM,rom_stats,output_print,output_plot</Sequence>
    <batchSize>3</batchSize>
  </RunInfo>
  ...
</Simulation>
```

   AThe *RunInfo* **Entity** is intended to set up the analysis that the user wants to perform. The steps listed in (**`<Sequence>`**) are going to be sequentially run using the number of processors specified in (**`<batchSize>`**). The first two steps build the ROM (**`'sample'`**, **`'train'`**), the next two validate the ROM against the original Code Model (**`'validateModel'`**, **`'validateROM'`**), **`'rom_stats'`** stores ROM-related information into DataObject, and the last two produce plots and print data (**`'output_print'`**, **`'output_plot'`**).

2. ***Models***:

   raven/tests/framework/user_guide/ForwardSamplingStrategies/forwardSamplingSparseGrid.xml

```
<Simulation>
  ...
  <Models>
    <ExternalModel subType="" name="projectile" ModuleToLoad="../../AnalyticModels/projectile.py">
      <variables>x,y,v0,angle,r,t,timeOption</variables>
    </ExternalModel>
    <ROM name="rom" subType="GaussPolynomialRom">
      <Target>r,t</Target>
      <Features>
        v0,angle
      </Features>
      <pivotParameter> t </pivotParameter>
      <IndexSet>TotalDegree</IndexSet>
      <PolynomialOrder>2</PolynomialOrder>
      <Interpolation poly="Legendre" quad="Legendre" weight="1">v0</Interpolation>
      <Interpolation poly="Legendre" quad="Legendre" weight="1">angle</Interpolation>
    </ROM>
  </Models>
  ...
</Simulation>
```

   The goal of this example is the generation of a GaussPolynomialRom for subsequent usage. In addition to the previously explained External model, the ROM of type *GaussPolynomial-Rom* is specified here. The ROM is generated through a Sparse Grid Collocation sampling strategy. Note that the **`<Interpolation>`** nodes are not required, but are included for the sake of demonstration.

3. ***Distributions***:

   raven/tests/framework/user_guide/ForwardSamplingStrategies/forwardSamplingSparseGrid.xml

```xml
<Simulation>
  ...
  <Distributions>
    <Normal name="vel_dist">
      <mean>30</mean>
      <sigma>5</sigma>
      <lowerBound>1</lowerBound>
      <upperBound>60</upperBound>
    </Normal>
    <Uniform name="angle_dist">
      <lowerBound>5</lowerBound>
      <upperBound>85</upperBound>
    </Uniform>
  </Distributions>
  ...
</Simulation>
```

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

- $vel\_dist \sim \mathbb{N}(30, 5)$, used to model the uncertainties associated with the *velocity*;
- $angle\_dist \sim \mathbb{U}(5, 85)$, used to model the uncertainties associated with the *angle*.

4. ***Samplers***:

raven/tests/framework/user_guide/ForwardSamplingStrategies/forwardSamplingSparseGrid.xml

```xml
<Simulation>
  ...
  <Samplers>
    <MonteCarlo name="mc">
      <samplerInit>
        <limit>100</limit>
        <initialSeed>42</initialSeed>
        <reseedEachIteration>True</reseedEachIteration>
      </samplerInit>
      <variable name="v0">
        <distribution>vel_dist</distribution>
      </variable>
      <variable name="angle">
        <distribution>angle_dist</distribution>
      </variable>
      <constant name="x0">0</constant>
      <constant name="y0">0</constant>
      <constant name="timeOption">0</constant>
    </MonteCarlo>
    <SparseGridCollocation name="SG">
      <variable name="v0">
        <distribution>vel_dist</distribution>
      </variable>
      <variable name="angle">
        <distribution>angle_dist</distribution>
      </variable>
      <constant name="x0">0</constant>
      <constant name="y0">0</constant>
      <constant name="timeOption">0</constant>
      <ROM class="Models" type="ROM">rom</ROM>
    </SparseGridCollocation>
  </Samplers>
  ...
</Simulation>
```

In order to employ the Sparse Grid Collocation sampling strategy, a **<SparseGridCollocation>** node needs to be defined. As can be seen from above, each variable is associated with a different distribution, defined in the **<Distributions>** block. In addition, the *GaussPolynomialRom* **<ROM>** is linked to the **<SparseGridCollocation>** sampler. Because this sampler is used exclusively to build the ROM, some of the parameters of the ROM are needed by the sampler, and this connection makes that communication possible. The setting of this ROM (e.g. polynomial order, Index set method, etc.) determines how the Stochastic Collocation Method is employed.

Additionally, a **<MonteCarlo>** sampler is set up for validating the ROM against the original Code. The random number generation seed (**<initialSeed>**) is specified and set to reset on each use (**<reseedEachIteration>**) so that the Monte Carlo sampler can be used to compare the ROM against the original model. We use 100 samples (**<limit>**) to sample the ROM and the model, and then print and plot both data sets to compare them.

5. ***DataObjects***:

raven/tests/framework/user_guide/ForwardSamplingStrategies/forwardSamplingSparseGrid.xml

```
<Simulation>
  ...
  <DataObjects>
    <PointSet name="inputPlaceholder">
      <Input>
        v0,angle
      </Input>
      <Output>OutputPlaceHolder</Output>
    </PointSet>
    <PointSet name="samplesModel">
      <Input>
        v0,angle
      </Input>
      <Output>r,t</Output>
    </PointSet>
    <PointSet name="samplesROM">
      <Input>
        v0,angle
      </Input>
      <Output>r,t</Output>
    </PointSet>
    <HistorySet name="histories">
      <Input>
        v0,angle
      </Input>
      <Output>x,y,r,t</Output>
      <options>
        <pivotParameter> t </pivotParameter>
      </options>
    </HistorySet>
    <DataSet name="rom_stats" />
  </DataObjects>
  ...
</Simulation>
```

In this block, five *DataObjects* are defined: 1) a PointSet named "inputPlaceholder" used as a placeholder input for the ROM sampling step, 2) a PointSet named "samplesModel" to store the Code responses from Monte Carlo samples, 3) a PointSet named "samplesROM" to store the ROM responses from Monte Carlo samples, 4) a HistorySet named "histories" used to collect the samples needed to train the ROM, and 5) a DataSet named "rom_stats" to store information from the ROM.

6. ***OutStreams***:

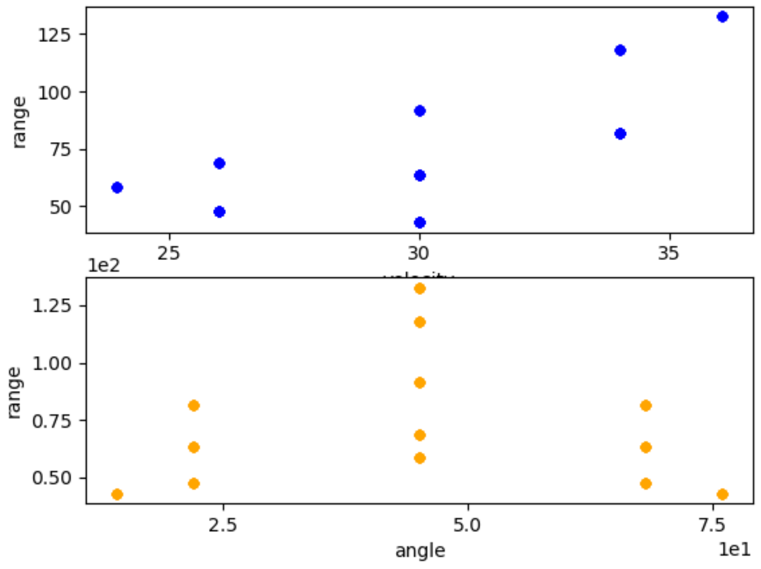


Figure 19. Plot of the training samples generated by the SparseGridCollocation sampling for variables A, B, C, D.

raven/tests/framework/user_guide/ForwardSamplingStrategies/forwardSamplingSparseGrid.xml

```
<Simulation>
  ...
  <OutStreams>
    <Print name="samplesModel">
      <type>csv</type>
      <source>samplesModel</source>
    </Print>
    <Print name="samplesROM">
      <type>csv</type>
      <source>samplesROM</source>
    </Print>
    <Print name="histories">
      <type>csv</type>
      <source>histories</source>
    </Print>
    <Print name="rom_output">
      <type>csv</type>
      <source>rom_stats</source>
    </Print>
    <Plot name="historyPlot" overwrite="false" verbosity="debug">
      <plotSettings>
        <gridSpace>2 1</gridSpace>
        <plot>
          <type>scatter</type>
          <x>histories|Input|v0</x>
          <y>histories|Output|r</y>
          <kwargs>
            <color>blue</color>
          </kwargs>
          <gridLocation>
            <x>0</x>
            <y>0</y>
          </gridLocation>
          <xlabel>velocity</xlabel>
          <ylabel>range</ylabel>
        </plot>
        <plot>
          <type>scatter</type>
          <x>histories|Input|angle</x>
          <y>histories|Output|r</y>
          <kwargs>
            <color>orange</color>
          </kwargs>
          <gridLocation>
            <x>1</x>
            <y>0</y>
          </gridLocation>
          <xlabel>angle</xlabel>
          <ylabel>range</ylabel>
        </plot>
      </plotSettings>
      <actions>
        <how>png</how>
        <title>
          <text> </text>
        </title>
      </actions>
    </Plot>
    <Plot name="samplesModelPlot3D" overwrite="false" verbosity="debug">
      <plotSettings>
        <gridSpace>2 1</gridSpace>
        <plot>
          <type>scatter</type>
          <x>samplesModel|Input|v0</x>
          <y>samplesModel|Input|angle</y>
          <z>samplesModel|Output|r</z>
          <c>blue</c>
          <gridLocation>
            <x>0</x>
            <y>0</y>
          </gridLocation>
          <xlabel>velocity</xlabel>
          <ylabel>angle</ylabel>
          <zlabel>range</zlabel>
        </plot>
        <plot>
          <type>scatter</type>
          <x>samplesModel|Input|v0</x>
          <y>samplesModel|Input|angle</y>
          <z>samplesModel|Output|t</z>
          <c>orange</c>
          <gridLocation>
            <x>1</x>
            <y>0</y>
          </gridLocation>
          <xlabel>velocity</xlabel>
          <ylabel>angle</ylabel>
          <zlabel>time</zlabel>
        </plot>
      </plotSettings>
      <actions>
        <how>png</how>
        <title>
          <text> </text>
        </title>
      </actions>
    </Plot>
    <Plot name="samplesROMPlot3D" overwrite="false" verbosity="debug">
      <plotSettings>
        <gridSpace>2 1</gridSpace>
        <plot>
          <type>scatter</type>
          <x>samplesROM|Input|v0</x>
          <y>samplesROM|Input|angle</y>
          <z>samplesROM|Output|r</z>
          <c>blue</c>
          <gridLocation>
            <x>0</x>
            <y>0</y>
          </gridLocation>
          <xlabel>velocity</xlabel>
          <ylabel>angle</ylabel>
          <zlabel>range</zlabel>
        </plot>
        <plot>
          <type>scatter</type>
          <x>samplesROM|Input|v0</x>
          <y>samplesROM|Input|angle</y>
          <z>samplesROM|Output|t</z>
          <c>orange</c>
          <gridLocation>
            <x>1</x>
            <y>0</y>
          </gridLocation>
          <xlabel>velocity</xlabel>
          <ylabel>angle</ylabel>
          <zlabel>time</zlabel>
        </plot>
      </plotSettings>
      <actions>
        <how>png</how>
        <title>
          <text> </text>
        </title>
      </actions>
    </Plot>
  </OutStreams>
  ...
</Simulation>
```

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

- *Print*:
  - named "samplesModel" connected with the *DataObjects* **Entity** "samplesModel" (**<source>**)
  - named "samplesROM" connected with the *DataObjects* **Entity** "samplesROM" (**<source>**)
  - named "histories" connected with the *DataObjects* **Entity** "histories" (**<source>**)
  - named "rom_output" connected with the *ROM* **Entity** "rom" (**<source>**).

  When these objects get used, all the information contained in the linked *DataObjects* are going to be exported in ether CSV files for DataObjects or XML files for ROMs (**<type>**).
- *Plot*:
  - named "historyPlot" connected with the *DataObjects* **Entity** "histories". This plots the variable $range$ with respect to the input variables $velocity$ and $angle$.
  - named "samplesModelPlot3D" connected with the *DataObjects* **Entity** "samplesModel". This plot will draw the variables $range, time$ as Monte Carlo sampled on the Code.
  - named "samplesROMPlot3D" connected with the *DataObjects* **Entity** "samplesROM". This plot will draw the variables $range, time$ as Monte Carlo sampled on the ROM.

7. ***Steps***:

raven/tests/framework/user_guide/ForwardSamplingStrategies/forwardSamplingSparseGrid.xml

```
<Simulation>
  ...
  <Steps>
    <MultiRun name="sample">
      <Input class="DataObjects" type="PointSet">inputPlaceholder</Input>
      <Model class="Models" type="ExternalModel">projectile</Model>
      <Sampler class="Samplers" type="SparseGridCollocation">SG</Sampler>
      <Output class="DataObjects" type="HistorySet">histories</Output>
    </MultiRun>
    <RomTrainer name="train">
      <Input class="DataObjects" type="HistorySet">histories</Input>
      <Output class="Models" type="ROM">rom</Output>
    </RomTrainer>
    <MultiRun name="validateModel">
      <Input class="DataObjects" type="PointSet">inputPlaceholder</Input>
      <Model class="Models" type="ExternalModel">projectile</Model>
      <Sampler class="Samplers" type="MonteCarlo">mc</Sampler>
      <Output class="DataObjects" type="PointSet">samplesModel</Output>
    </MultiRun>
    <MultiRun name="validateROM">
      <Input class="DataObjects" type="PointSet">inputPlaceholder</Input>
      <Model class="Models" type="ROM">rom</Model>
      <Sampler class="Samplers" type="MonteCarlo">mc</Sampler>
      <Output class="DataObjects" type="PointSet">samplesROM</Output>
    </MultiRun>
    <IOStep name="rom_stats">
      <Input class="Models" type="ROM">rom</Input>
      <Output class="DataObjects" type="DataSet">rom_stats</Output>
    </IOStep>
    <IOStep name="output_print">
      <Input class="DataObjects" type="HistorySet">histories</Input>
      <Input class="DataObjects" type="PointSet">samplesModel</Input>
      <Input class="DataObjects" type="PointSet">samplesROM</Input>
      <Input class="DataObjects" type="DataSet">rom_stats</Input>
      <Output class="OutStreams" type="Print">samplesModel</Output>
      <Output class="OutStreams" type="Print">samplesROM</Output>
      <Output class="OutStreams" type="Print">histories</Output>
      <Output class="OutStreams" type="Print">rom_output</Output>
    </IOStep>
    <IOStep name="output_plot" pauseAtEnd="True">
      <Input class="DataObjects" type="HistorySet">histories</Input>
      <Input class="DataObjects" type="PointSet">samplesModel</Input>
      <Input class="DataObjects" type="PointSet">samplesROM</Input>
      <Output class="OutStreams" type="Plot">historyPlot</Output>
      <Output class="OutStreams" type="Plot">samplesModelPlot3D</Output>
      <Output class="OutStreams" type="Plot">samplesROMPlot3D</Output>
    </IOStep>
  </Steps>
  ...
</Simulation>
```

Finally, the previously defined **Entities** can be combined in the `<Steps>` block. The following `<Steps>` have been defined:

- `<MultiRun>` named “sample”, used to run the training samples of the driven code and collect the outputs in the *DataObjects*. The `<Sampler>` is specified to communicate

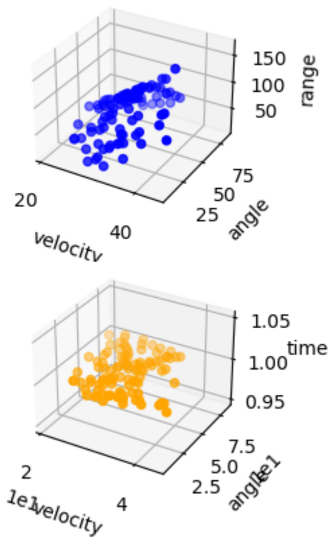


Figure 20. Plot of validation samples generated by Monte Carlo sampling on the Code.

to the *Step* that the driven code needs to be sampled through the Sparse Grid Collocation sampling strategy;

- **`<RomTrainer>`** named "train", 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 "sampleModel", used to run the Monte Carlo perturbed samples of the original Model for use in verification. The results are collected in the *samplesModel DataObjects*.
- **`<MultiRun>`** named "sampleROM", used to run the Monte Carlo perturbed samples of the previously constructed ROM for use in verificaiton. The results are collected in the *samplesROM DataObjects*.
- **`<IOStep>`** named "rom_stas", used to dump rom-related information into *DataObject* ,
- **`<IOStep>`** named "output_print", used to dump the "histories", "samplesModel" and "samplesROM" *DataObjects* **Entity** in a CSV file,
- **`<IOStep>`** named "output_plot", used to plot the data and store it in the PNG file and on the screen.

As it can be seen, the constructed ROM can accurately represent the response of the driven code.

This example shows the potential of reduced order modeling, in general, and of the *GaussPolynomialRom*, in particular.

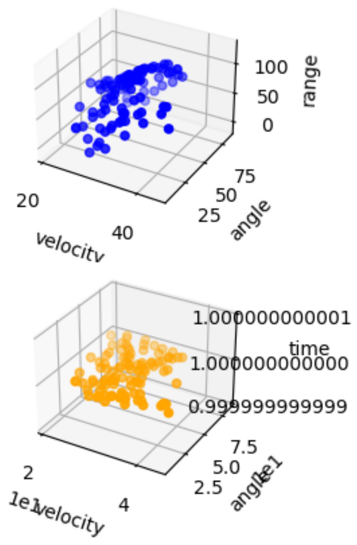


Figure 21. Plot of validation samples generated by Monte Carlo sampling on the ROM.

4 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, we will only show how to use the first algorithm.

4.1 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***:

   `raven/tests/framework/user_guide/AdaptiveSamplingStrategies/adaptiveSamplingLSsearch.xml`

```
<Simulation>
  ...
  <RunInfo>
    <JobName>LSsearch</JobName>
    <Sequence>sample,computeLSintegral,writeHistories</Sequence>
    <WorkingDir>LSsearch</WorkingDir>
    <batchSize>1</batchSize>
  </RunInfo>
  ...
</Simulation>
```

As shown in Section 2.2, 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 1 processor (**<batchSize>**).

2. ***Files***:

raven/tests/framework/user_guide/AdaptiveSamplingStrategies/adaptiveSamplingLSsearch.xml

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

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***:

raven/tests/framework/user_guide/AdaptiveSamplingStrategies/adaptiveSamplingLSsearch.xml

```
<Simulation>
  ...
  <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" />
    </Code>
    <ROM name="AccelerationROM" subType="KNeighborsClassifier">
      <Features>sigma-A,decay-A</Features>
      <Target>goalFunction</Target>
      <algorithm>brute</algorithm>
      <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 class="Distributions" type="Uniform">sigmaA</distribution>
      </variable>
      <variable name="decay-A">
        <distribution class="Distributions" type="Uniform">decayConstantA</distribution>
      </variable>
      <outputName>EventProbability</outputName>
    </PostProcessor>
  </Models>
  ...
</Simulation>
```

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 $decay - A$.
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***:

raven/tests/framework/user_guide/AdaptiveSamplingStrategies/adaptiveSamplingLSsearch.xml

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

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:

- $sigmaA \sim \mathbb{U}(0, 1000)$, used to model the uncertainty associated with the Model *sigma-A*

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

5. ***Samplers***:

raven/tests/framework/user_guide/AdaptiveSamplingStrategies/adaptiveSamplingLSsearch.xml

```
<Simulation>
  ...
  <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>
  ...
</Simulation>
```

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***:

raven/tests/framework/user_guide/AdaptiveSamplingStrategies/adaptiveSamplingLSsearch.xml

```
<Simulation>
  ...
  <Functions>
    <External file="goalFunction" name="goalFunction">
      <variables>A</variables>
    </External>
  </Functions>
  ...
</Simulation>
```

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 $<=$ of 0.1 , the system is considered to be in a “safe” condition.

```
def __residuumSign(self):
  returnValue = 1.0
  if self.A  <= 0.1:
    returnValue = -1.0
  return returnValue
```

7. ***DataObjects***:

   raven/tests/framework/user_guide/AdaptiveSamplingStrategies/adaptiveSamplingLSsearch.xml

```
<Simulation>
  ...
  <DataObjects>
    <PointSet name="limitSurface">
      <Input>sigma-A,decay-A</Input>
      <Output>goalFunction</Output>
    </PointSet>
    <PointSet name="limitSurfaceIntegral">
      <Input>sigma-A,decay-A</Input>
      <Output>goalFunction,EventProbability</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>
  ...
</Simulation>
```

   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.

8. ***OutStreams***:

   raven/tests/framework/user_guide/AdaptiveSamplingStrategies/adaptiveSamplingLSsearch.xml

```
<Simulation>
  ...
  <OutStreams>
    <Print name="samples">
      <type>csv</type>
      <source>samples</source>
    </Print>
    <Print name="histories">
      <type>csv</type>
      <source>histories</source>
    </Print>
    <Print name="LSintegral">
      <type>csv</type>
      <source>limitSurfaceIntegral</source>
    </Print>
    <Plot name="historyPlot" overwrite="false" verbosity="debug">
      <plotSettings>
        <gridSpace>2 2</gridSpace>
        <plot>
          <type>line</type>
          <x>histories|Output|time</x>
          <y>histories|Output|A</y>
          <kwargs>
            <color>blue</color>
          </kwargs>
          <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>
          <kwargs>
            <color>orange</color>
          </kwargs>
          <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>
          <kwargs>
            <color>green</color>
          </kwargs>
          <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>
          <kwargs>
            <color>red</color>
          </kwargs>
          <gridLocation>
            <x>1</x>
            <y>1</y>
          </gridLocation>
          <xlabel>time (s)</xlabel>
          <ylabel>evolution D(kg)</ylabel>
        </plot>
      </plotSettings>
      <actions>
        <how>png</how>
        <title>
          <text> </text>
        </title>
      </actions>
    </Plot>
    <Plot name="limitSurfacePlot" overwrite="false" verbosity="debug">
      <plotSettings>
        <plot>
          <type>scatter</type>
          <x>limitSurface|Input|decay-A</x>
          <y>limitSurface|Input|sigma-A</y>
        </plot>
        <xlabel>decay-A</xlabel>
        <ylabel>sigma-A</ylabel>
      </plotSettings>
      <actions>
        <how>png</how>
        <range>
          <xmin>0.00000000</xmin>
          <xmax>0.0000001</xmax>
        </range>
        <title>
          <text> </text>
        </title>
      </actions>
    </Plot>
    <Plot name="samplesPlot3D" overwrite="false" verbosity="debug">
      <plotSettings>
        <gridSpace>2 1</gridSpace>
        <plot>
```

```
          <type>scatter</type>
          <x>samples|Input|decay-A</x>
          <y>samples|Input|sigma-A</y>
          <z>samples|Output|A</z>
          <c>blue</c>
          <gridLocation>
            <x>0</x>
            <y>0</y>
          </gridLocation>
          <xlabel>decay-A</xlabel>
          <ylabel>sigma-A</ylabel>
          <zlabel>final A</zlabel>
        </plot>
        <plot>
          <type>scatter</type>
          <x>samples|Input|decay-A</x>
          <y>samples|Input|sigma-A</y>
          <z>samples|Output|B</z>
          <c>blue</c>
          <gridLocation>
            <x>1</x>
            <y>0</y>
          </gridLocation>
          <xlabel>decay-A</xlabel>
          <ylabel>sigma-A</ylabel>
          <zlabel>final B</zlabel>
        </plot>
      </plotSettings>
      <actions>
        <how>png</how>
        <title>
          <text> </text>
        </title>
      </actions>
    </Plot>
  </OutStreams>
  ...
</Simulation>
```

Several out streams are included in this workflow, two for printing and three for plotting:

- "samples", which writes the validation sample contents of the **'samples'** PointSet DataObject to a CSV file,
- "histories", which writes the sampling contents of the **'histories'** HistorySet DataObject to a set of connected CSV files,
- "historyPlot", which plots the evolution of the samples taken,
- "limitSurfacePlot", which plots the limit surface discovered by the PostProcessor,
- "samplesPlot3D", which plots the final state of the samples taken against the figures of merit.

The plots demonstrate how visualization of three-dimensional data, time-dependent data, and limit surfaces can be realized using RAVEN.

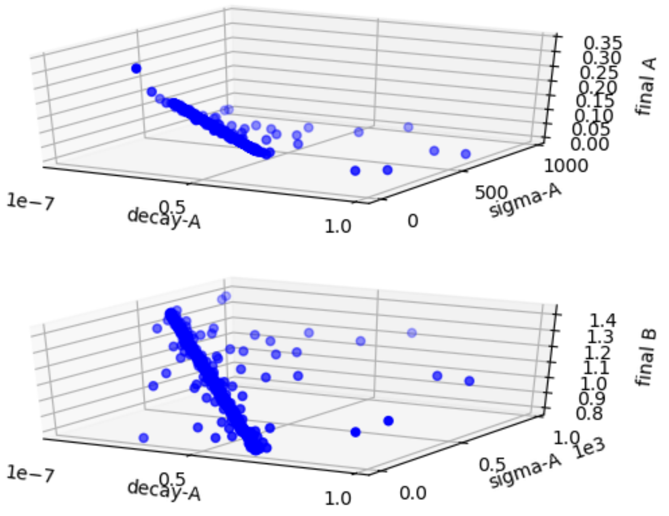


Figure 22. Plot of the samples generated by the LS search sampling for variables A, B.

9. ***Steps***:

raven/tests/framework/user_guide/AdaptiveSamplingStrategies/adaptiveSamplingLSsearch.xml

```
<Simulation>
  ...
  <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">limitSurfaceIntegral</Output>
      <Output class="OutStreams" type="Print">LSintegral</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>
  ...
</Simulation>
```

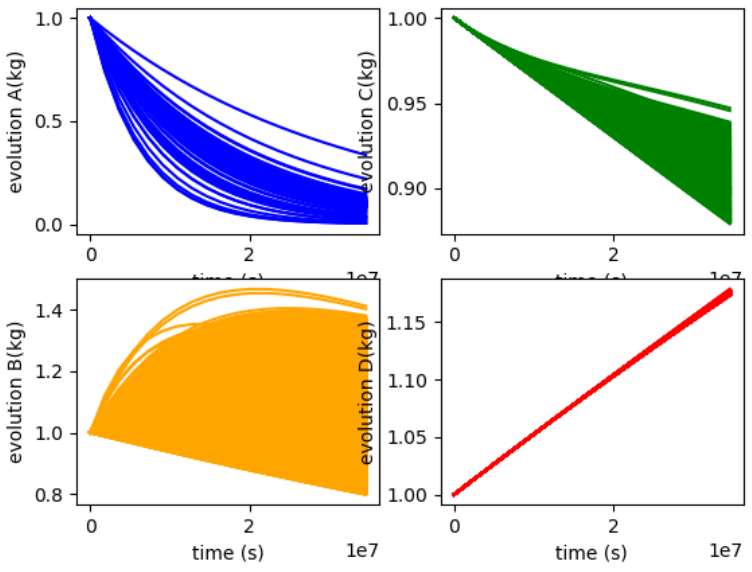


Figure 23. Plot of the histories generated by the LS search method for variables A, B, C, D.

Finally, all the previously defined **Entities** can be combined in the **`<Steps>`** block. As inferable, three **`<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 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 23 shows the evolution of the outputs A, B, C, D under uncertainties. Figure 22 shows the final responses of A and B of the sampling employed using the driven code. Figure 24 shows the limit surface for this particular example. Only 399 samples were needed in order to reach the full convergence.
The integration of the LS determines a probability of failure of $1.79e-1$.

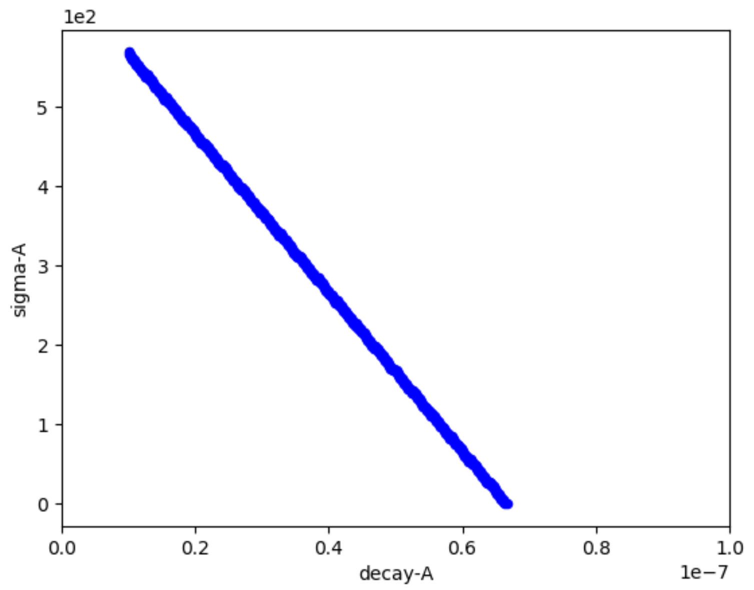


Figure 24. Limit Surface generated by the LS search method.

5 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
<Simulation verbosity="debug">
  <RunInfo>
    <WorkingDir>grid</WorkingDir>
    <Sequence>makeCoarse,printCoarse,load,makeRestart,printRestart</Sequence>
    <batchSize>1</batchSize>
  </RunInfo>

  <TestInfo>
    <name>framework/Samplers/Restart/Truncated/Grid</name>
    <author>talbpaul</author>
    <created>2016-04-05</created>
    <classesTested>Samplers.Grid</classesTested>
    <description>
      This is similar to the restart tests in the parent directory, but in this one we test the use of the
      restartTolerance to recover restart points from a code that produces finite precision when reporting input
      values.  As with the other restart tests, "coarse" returns a 1 and "fine" returns a 2.
    </description>
  </TestInfo>

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

  <Steps>
    <MultiRun name="makeCoarse">
      <Input class="Files" type="Input">inp</Input>
      <Model class="Models" type="Code">coarse</Model>
      <Sampler class="Samplers" type="Grid">coarse</Sampler>
      <Output class="DataObjects" type="PointSet">coarse</Output>
    </MultiRun>
    <MultiRun name="makeRestart">
      <Input class="Files" type="Input">inp</Input>
      <Model class="Models" type="Code">fine</Model>
      <Sampler class="Samplers" type="Grid">fine</Sampler>
      <Output class="DataObjects" type="PointSet">fine</Output>
    </MultiRun>
    <IOStep name="printCoarse">
      <Input class="DataObjects" type="PointSet">coarse</Input>
      <Output class="OutStreams" type="Print">coarse</Output>
    </IOStep>
    <IOStep name="load">
      <Input class="Files" type="">csv</Input>
      <Output class="DataObjects" type="PointSet">restart</Output>
    </IOStep>
    <IOStep name="printRestart">
      <Input class="DataObjects" type="PointSet">fine</Input>
      <Output class="OutStreams" type="Print">fine</Output>
    </IOStep>
  </Steps>

  <Distributions>
    <Uniform name="u1">
      <lowerBound>0.123456789012345</lowerBound>
      <upperBound>1</upperBound>
    </Uniform>
    <Uniform name="u2">
      <lowerBound>10.123456789012345</lowerBound>
      <upperBound>11</upperBound>
    </Uniform>
  </Distributions>

  <Samplers>
    <Grid name="coarse">
      <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="fine">
      <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>5e-3</restartTolerance>
    </Grid>
  </Samplers>

  <Models>
    <Code name="coarse" subType="GenericCode">
      <executable>model_1.py</executable>
      <clargs arg="python" type="prepend" />
      <clargs arg="-i" extension=".i" type="input" />
      <clargs arg="-o" type="output" />
      <prepend>python</prepend>
    </Code>
    <Code name="fine" subType="GenericCode">
      <executable>model_2.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="coarse">
      <Input>x,y</Input>
      <Output>a</Output>
    </PointSet>
    <PointSet name="restart">
      <Input>x,y</Input>
      <Output>a</Output>
    </PointSet>
    <PointSet name="fine">
      <Input>x,y</Input>
      <Output>a</Output>
    </PointSet>
  </DataObjects>

  <OutStreams>
    <Print name="coarse">
      <type>csv</type>
      <source>coarse</source>
      <what>input,output</what>
    </Print>
    <Print name="fine">
      <type>csv</type>
      <source>fine</source>
      <what>input,output</what>
    </Print>
  </OutStreams>
</Simulation>
```

In order to restart an existing file that failed for some reason, the data will need to be listed in the **<Files>** section and a **<IOStep>** that loads that input into a different data object will be needed to be added to the **<Steps>** section. After that a **<Restart>** node can be added.

raven/tests/framework/Samplers/Restart/test_restart_Grid_part2.xml

```
<Simulation>
  ...
  <Files>
    <Input name="old_solns" type="">coarse.csv</Input>
  </Files>
  ...
</Simulation>
```

raven/tests/framework/Samplers/Restart/test_restart_Grid_part2.xml

```
<Simulation>
  ...
  <Steps>
    <IOStep name="load_old">
      <Input class="Files" type="">old_solns</Input>
      <Output class="DataObjects" type="PointSet">solns_orig</Output>
    </IOStep>
    <MultiRun name="makeCoarse">
      <Input class="DataObjects" type="PointSet">dummyIN</Input>
      <Model class="Models" type="ExternalModel">coarsemod</Model>
      <Sampler class="Samplers" type="Grid">coarse</Sampler>
      <Output class="DataObjects" type="PointSet">solns</Output>
    </MultiRun>
    <IOStep name="print">
      <Input class="DataObjects" type="PointSet">solns</Input>
      <Output class="OutStreams" type="Print">coarse2</Output>
    </IOStep>
  </Steps>
  ...
</Simulation>
```

raven/tests/framework/Samplers/Restart/test_restart_Grid_part2.xml

```
<Simulation>
  ...
  <Samplers>
    <Grid name="coarse">
      <variable name="x1">
        <distribution>u1</distribution>
        <grid construction="equal" steps="2" type="CDF">0.0 1.0</grid>
      </variable>
      <variable name="x2">
        <distribution>u2</distribution>
        <grid construction="equal" steps="2" type="CDF">0.0 1.0</grid>
      </variable>
      <Restart class="DataObjects" type="PointSet">solns_orig</Restart>
    </Grid>
  </Samplers>
  ...
</Simulation>
```

6 Reduced Order Modeling through RAVEN

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 (increased calculation time). 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.

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, i.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 some ROMs might be subject to the over-fitting issues. The over-fitting phenomenon is not analyzed here, 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

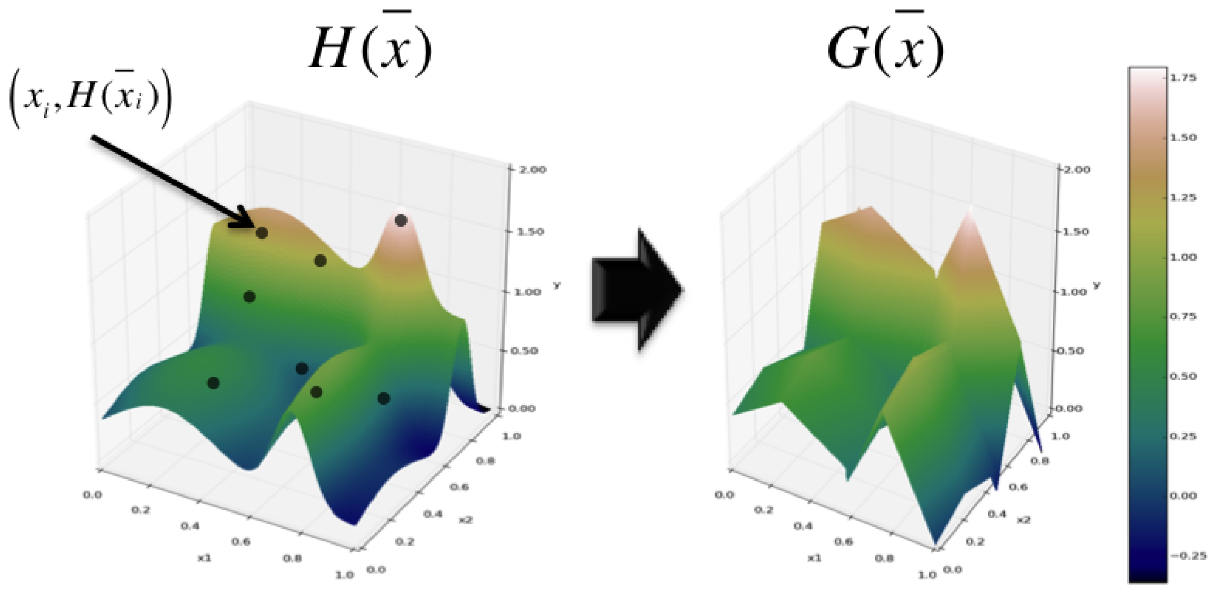


Figure 25. Example of reduced order model representation of physical system (regression).

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.

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***:

   `raven/tests/framework/user_guide/ReducedOrderModeling/reducedOrderModeling.xml`

```
<Simulation>
  ...
  <RunInfo>
    <JobName>ROMConstruction</JobName>
    <Sequence>
        sample,trainROMGaussianProcess,trainROMsvm,
        trainROMinverse,sampleROMGaussianProcess,
        sampleROMInverse,sampleROMsvm,writeHistories
    </Sequence>
    <WorkingDir>ROMConstruction</WorkingDir>
    <batchSize>3</batchSize>
  </RunInfo>
  ...
</Simulation>
```

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

2. ***Models***:
raven/tests/framework/user_guide/ReducedOrderModeling/reducedOrderModeling.xml

```
<Simulation>
  ...
  <Models>
    <ExternalModel ModuleToLoad="../../../AnalyticModels/projectile.py" name="projectile" subType="">
      <variables>v0,angle,r,t,x,y,timeOption</variables>
    </ExternalModel>
    <ROM name="ROMGaussianProcess" subType="GaussianProcess">
      <Features>v0,angle</Features>
      <Target>r,t</Target>
    </ROM>
    <ROM name="ROMsvm" subType="SVR">
      <Features>v0,angle</Features>
      <Target>r,t</Target>
      <kernel>rbf</kernel>
      <C>50.0</C>
      <tol>0.000001</tol>
    </ROM>
    <ROM name="ROMinverse" subType="NDinvDistWeight">
      <Features>v0,angle</Features>
      <Target>r,t</Target>
      <p>3</p>
    </ROM>
  </Models>
  ...
</Simulation>
```

As mentioned earlier, the goal of this example is the employment of a sampling strategy in order to construct multiple types of ROMs.
Indeed, in addition to an External 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 External model. Once trained, they are going to be used in place of the original model.
As it can be seen, the ROMs will be constructed considering two features ($v0,\ and angle,$) and two targets ($r\ and\ t$).

3. ***Distributions***:

raven/tests/framework/user_guide/ReducedOrderModeling/reducedOrderModeling.xml

```
<Simulation>
  ...
  <Distributions>
    <Normal name="vel_dist">
      <mean>30</mean>
      <sigma>5</sigma>
      <lowerBound>1</lowerBound>
      <upperBound>60</upperBound>
    </Normal>
    <Uniform name="angle_dist">
      <lowerBound>5</lowerBound>
      <upperBound>85</upperBound>
    </Uniform>
  </Distributions>
  ...
</Simulation>
```

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

- $vel\_dist \sim \mathbb{N}(30, 5)$, used to model the uncertainties associated with the *velocity*;
- $angle\_dist \sim \mathbb{U}(5, 85)$, used to model the uncertainties associated with the *angle*.

4. ***Samplers***:

raven/tests/framework/user_guide/ReducedOrderModeling/reducedOrderModeling.xml

```
<Simulation>
  ...
  <Samplers>
    <MonteCarlo name="my_mc">
      <samplerInit>
        <limit>500</limit>
        <initialSeed>42</initialSeed>
      </samplerInit>
      <variable name="v0">
```

```
        <distribution>vel_dist</distribution>
      </variable>
      <variable name="angle">
        <distribution>angle_dist</distribution>
      </variable>
      <constant name="x0">0</constant>
      <constant name="y0">0</constant>
      <constant name="timeOption">1</constant>
    </MonteCarlo>
  </Samplers>
  ...
</Simulation>
```

To obtain the data-set on which the data mining algorithms are going to be applied, a *MonteCarlo* sampling approach is employed here.

5. ***DataObjects***:

raven/tests/framework/user_guide/ReducedOrderModeling/reducedOrderModeling.xml

```
<Simulation>
  ...
  <DataObjects>
    <PointSet name="inputPlaceHolder">
      <Input>v0,angle</Input>
      <Output>OutputPlaceHolder</Output>
    </PointSet>
    <PointSet name="samples">
      <Input>v0,angle</Input>
      <Output>r,t</Output>
    </PointSet>
    <PointSet name="samplesGP">
      <Input>v0,angle</Input>
      <Output>r,t</Output>
    </PointSet>
    <PointSet name="samplesInverse">
      <Input>v0,angle</Input>
      <Output>r,t</Output>
    </PointSet>
    <PointSet name="samplesSVM">
      <Input>v0,angle</Input>
      <Output>r,t</Output>
    </PointSet>
    <HistorySet name="histories">
```

```
      <Input>v0,angle</Input>
      <Output>x,y,r,t</Output>
      <options>
        <pivotParameter>t</pivotParameter>
      </options>
    </HistorySet>
  </DataObjects>
  ...
</Simulation>
```

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 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 Gaussian Process (GP) ROM; 5) PointSet named "samplesInverse" used to collect the final outcomes (sampling) of the Inverse Distance Weighting (IDW) ROM; 6) PointSet named "samplesSVM" used to collect the final outcomes (sampling) of the Support Vector Machine (SVM) ROM.

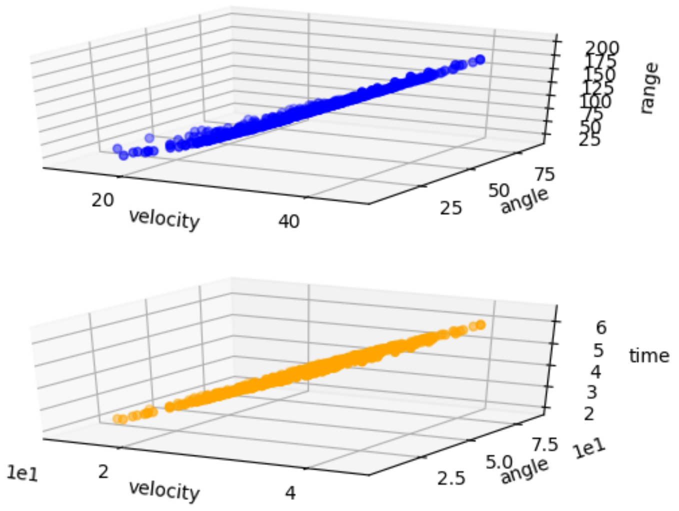


Figure 26. Plot of the samples generated by the Monte Carlo sampling

6. ***OutStreams***:

raven/tests/framework/user_guide/ReducedOrderModeling/reducedOrderModeling.xml

```
<Simulation>
  ...
  <OutStreams>
    <Print name="samples">
      <type>csv</type>
      <source>samples</source>
    </Print>
    <Print name="histories">
      <type>csv</type>
      <source>histories</source>
    </Print>
    <Plot name="historyPlot" overwrite="false" verbosity="debug">
      <plotSettings>
        <gridSpace>2 1</gridSpace>
        <plot>
          <type>scatter</type>
          <x>histories|Input|v0</x>
          <y>histories|Output|r</y>
          <kwargs>
            <color>blue</color>
          </kwargs>
          <gridLocation>
            <x>0</x>
            <y>0</y>
          </gridLocation>
          <xlabel>velocity</xlabel>
          <ylabel>range</ylabel>
        </plot>
        <plot>
          <type>scatter</type>
          <x>histories|Input|angle</x>
          <y>histories|Output|r</y>
          <kwargs>
            <color>orange</color>
          </kwargs>
          <gridLocation>
            <x>1</x>
            <y>0</y>
          </gridLocation>
          <xlabel>angle</xlabel>
          <ylabel>range</ylabel>
        </plot>
      </plotSettings>
      <actions>
        <how>png</how>
        <title>
          <text> </text>
        </title>
      </actions>
    </Plot>
    <Plot name="samplesPlot3D" overwrite="false" verbosity="debug">
      <plotSettings>
        <gridSpace>2 1</gridSpace>
        <plot>
          <type>scatter</type>
          <x>samples|Input|v0</x>
          <y>samples|Input|angle</y>
          <z>samples|Output|r</z>
          <c>blue</c>
          <gridLocation>
            <x>0</x>
            <y>0</y>
          </gridLocation>
          <xlabel>velocity</xlabel>
          <ylabel>angle</ylabel>
          <zlabel>range</zlabel>
        </plot>
        <plot>
          <type>scatter</type>
          <x>samples|Input|v0</x>
          <y>samples|Input|angle</y>
          <z>samples|Output|t</z>
          <c>orange</c>
          <gridLocation>
            <x>1</x>
            <y>0</y>
          </gridLocation>
          <xlabel>velocity</xlabel>
          <ylabel>angle</ylabel>
          <zlabel>time</zlabel>
        </plot>
      </plotSettings>
      <actions>
        <how>png</how>
        <title>
          <text> </text>
        </title>
      </actions>
    </Plot>
    <Plot name="samplesPlot3DROMgp" overwrite="false" verbosity="debug">
      <plotSettings>
        <gridSpace>2 1</gridSpace>
        <plot>
          <type>scatter</type>
          <x>samples|Input|v0</x>
          <y>samples|Input|angle</y>
          <z>samples|Output|r</z>
          <c>blue</c>
          <gridLocation>
            <x>0</x>
            <y>0</y>
          </gridLocation>
          <xlabel>velocity</xlabel>
          <ylabel>angle</ylabel>
          <zlabel>range</zlabel>
        </plot>
        <plot>
          <type>scatter</type>
          <x>samples|Input|v0</x>
          <y>samples|Input|angle</y>
          <z>samples|Output|t</z>
          <c>orange</c>
          <gridLocation>
            <x>1</x>
            <y>0</y>
          </gridLocation>
          <xlabel>velocity</xlabel>
          <ylabel>angle</ylabel>
          <zlabel>time</zlabel>
        </plot>
      </plotSettings>
      <actions>
        <how>png</how>
        <title>
          <text> </text>
        </title>
      </actions>
    </Plot>
```

```
<Plot name="samplesPlot3DROMsvm" overwrite="false" verbosity="debug">
  <plotSettings>
    <gridSpace>2 1</gridSpace>
    <plot>
      <type>scatter</type>
      <x>samples|Input|v0</x>
      <y>samples|Input|angle</y>
      <z>samples|Output|r</z>
      <c>blue</c>
      <gridLocation>
        <x>0</x>
        <y>0</y>
      </gridLocation>
      <xlabel>velocity</xlabel>
      <ylabel>angle</ylabel>
      <zlabel>range</zlabel>
    </plot>
    <plot>
      <type>scatter</type>
      <x>samples|Input|v0</x>
      <y>samples|Input|angle</y>
      <z>samples|Output|t</z>
      <c>orange</c>
      <gridLocation>
        <x>1</x>
        <y>0</y>
      </gridLocation>
      <xlabel>velocity</xlabel>
      <ylabel>angle</ylabel>
      <zlabel>time</zlabel>
    </plot>
  </plotSettings>
  <actions>
    <how>png</how>
    <title>
      <text> </text>
    </title>
  </actions>
</Plot>
<Plot name="samplesPlot3DROMinverse" overwrite="false" verbosity="debug">
  <plotSettings>
    <gridSpace>2 1</gridSpace>
    <plot>
      <type>scatter</type>
      <x>samples|Input|v0</x>
      <y>samples|Input|angle</y>
      <z>samples|Output|r</z>
      <c>blue</c>
      <gridLocation>
        <x>0</x>
        <y>0</y>
      </gridLocation>
      <xlabel>velocity</xlabel>
      <ylabel>angle</ylabel>
      <zlabel>range</zlabel>
    </plot>
    <plot>
      <type>scatter</type>
      <x>samples|Input|v0</x>
      <y>samples|Input|angle</y>
      <z>samples|Output|t</z>
      <c>orange</c>
      <gridLocation>
        <x>1</x>
        <y>0</y>
```

```
          </gridLocation>
          <xlabel>velocity</xlabel>
          <ylabel>angle</ylabel>
          <zlabel>time</zlabel>
        </plot>
      </plotSettings>
      <actions>
        <how>png</how>
        <title>
          <text> </text>
        </title>
      </actions>
    </Plot>
  </OutStreams>
  ...
</Simulation>
```

This model makes use of two Print OutStreams and five Plot OutStreams:

- "samples," which writes the contents of the point-wise training samples to CSV,
- "histories," which writes the contents of the history-wise training samples to linked CSVs,
- "historyPlot," which plots the evolution of the training samples,
- "samplesPlot3D," which plots the final state of the training samples with relation to the outputs of interest,
- "samplesPlot3DROMgp," which plots the validation samples of the Gaussian Process ROM,
- "samplesPlot3DROMsvm," which plots the validation samples of the Support-Vector Machine ROM,
- "samplesPlot3Dinverse," which plots the validation samples of the multidimensional Inverse Weight ROM.

The 3D plots of the samples as well as the ROM samples can be used as a view-norm validation of the ROMs.

7. ***Steps***:

raven/tests/framework/user_guide/ReducedOrderModeling/reducedOrderModeling.xml

```
<Simulation>
  ...
  <Steps>
    <MultiRun name="sample">
      <Input class="DataObjects" type="PointSet">inputPlaceHolder</Input>
      <Model class="Models" type="ExternalModel">projectile</Model>
      <Sampler class="Samplers" type="MonteCarlo">my_mc</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="MonteCarlo">my_mc</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="MonteCarlo">my_mc</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="MonteCarlo">my_mc</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>
  ...
</Simulation>
```

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**;
- **`<RomTrainer>`** named "trainROMinverse", used to construct ("train") the IDW ROM, based on the data-set generated in the "sample" **Step**;

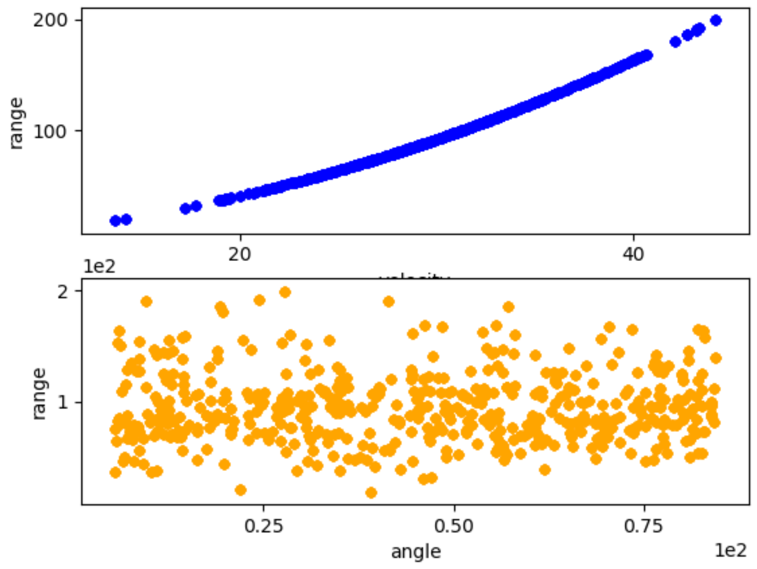


Figure 27. Plot of the histories generated by the Monte Carlo method

- **`<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" *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 27 shows the range r for different velocity and angle. Figure 26 shows the final responses of the sampling employed using the driven code.

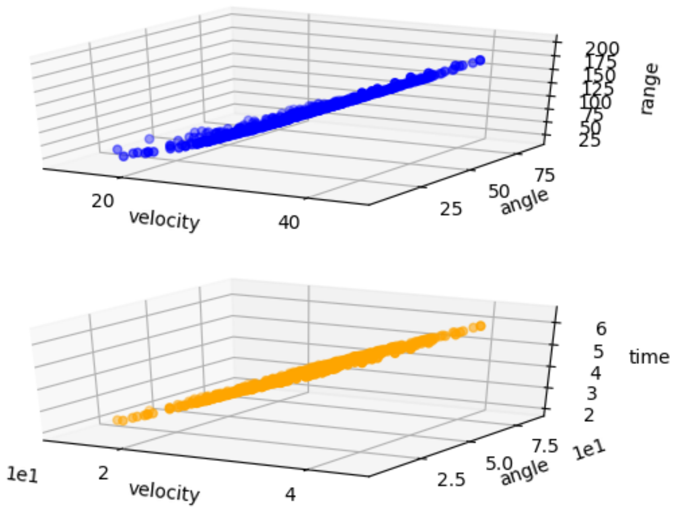


Figure 28. Plot of the samples generated by the Monte Carlo sampling applied on the Gaussian Process ROM

Figures 28, 29 and 30 show the final responses 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 26).

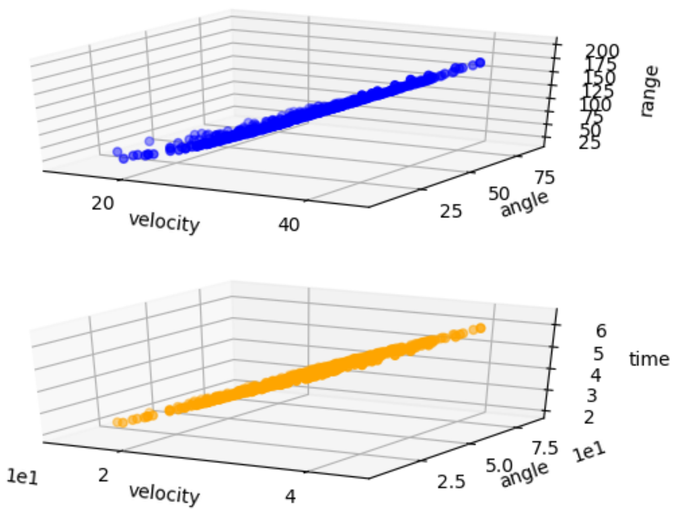


Figure 29. Plot of the samples generated by the Monte Carlo sampling applied on the Support Vector Machine ROM

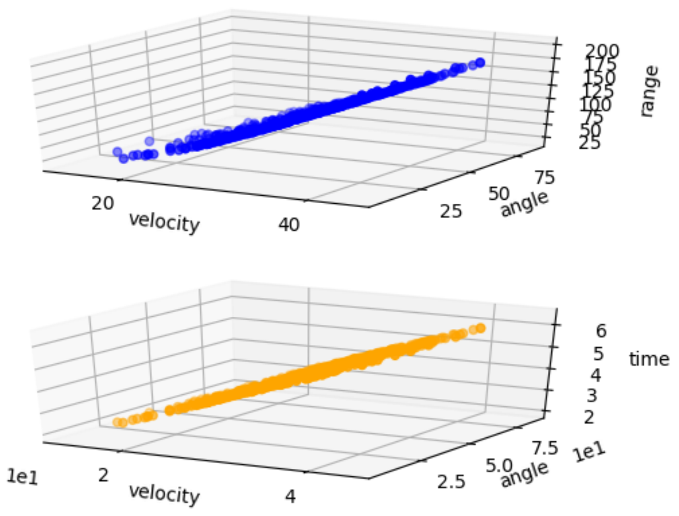


Figure 30. Plot of the samples generated by the Monte Carlo sampling applied on the Inverse Distance Weight ROM

7 Statistical Analysis through RAVEN

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*.

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***:

   raven/tests/framework/user_guide/StatisticalAnalysis/statisticalAnalysis.xml

```
<Simulation>
  ...
  <RunInfo>
    <JobName>StatisticalAnalysis</JobName>
    <Sequence>sampleMC,statisticalAnalysisMC</Sequence>
    <WorkingDir>StatisticalAnalysis</WorkingDir>
    <batchSize>3</batchSize>
  </RunInfo>
  ...
</Simulation>
```

   As shown in the other examples, the *RunInfo* **Entity** is intended to set up the desired analysis. The number of steps specified in (`<Sequence>`) are sequentially run, two steps in this specific case, using the number of processors assigned in (`<batchSize>`).
   In the first step, the original physical model is sampled. The obtained results are analyzed with the Statistical Post-Processor.

2. ***Models***:
   raven/tests/framework/user_guide/StatisticalAnalysis/statisticalAnalysis.xml

```
<Simulation>
  ...
  <Models>
    <ExternalModel ModuleToLoad="../../../AnalyticModels/projectile.py" name="projectile" subType="">
      <variables>v0,angle,r,t,x,y,timeOption</variables>
    </ExternalModel>
    <PostProcessor name="statisticalAnalysis" subType="BasicStatistics" verbosity="debug">
      <skewness prefix="skew">r,t</skewness>
      <variationCoefficient prefix="vc">r,t</variationCoefficient>
      <percentile prefix="percentile">r,t</percentile>
      <expectedValue prefix="mean">r,t</expectedValue>
      <kurtosis prefix="kurt">r,t</kurtosis>
      <median prefix="median">r,t</median>
      <maximum prefix="max">r,t</maximum>
      <minimum prefix="min">r,t</minimum>
      <samples prefix="samp">r,t</samples>
      <variance prefix="var">r,t</variance>
      <sigma prefix="sigma">r,t</sigma>
      <NormalizedSensitivity prefix="nsen">
        <targets>r,t</targets>
        <features>v0,angle</features>
      </NormalizedSensitivity>
      <sensitivity prefix="sen">
        <targets>r,t</targets>
        <features>v0,angle</features>
      </sensitivity>
      <pearson prefix="pear">
        <targets>r,t</targets>
        <features>v0,angle</features>
      </pearson>
      <covariance prefix="cov">
        <targets>r,t</targets>
        <features>v0,angle</features>
      </covariance>
      <VarianceDependentSensitivity prefix="vsen">
        <targets>r,t</targets>
        <features>v0,angle</features>
      </VarianceDependentSensitivity>
    </PostProcessor>
  </Models>
  ...
</Simulation>
```

The goal of this example is to show how the principal statistical FOMs can be computed through RAVEN.
We use an External model and specify a Post-Processor model (BasicStatistics). The post-process step is performed on all the output FOMs used in this example ($randt$).

3. ***Distributions***:

raven/tests/framework/user_guide/StatisticalAnalysis/statisticalAnalysis.xml

```
<Simulation>
  ...
  <Distributions>
    <Normal name="vel_dist">
      <mean>30</mean>
      <sigma>5</sigma>
      <lowerBound>1</lowerBound>
      <upperBound>60</upperBound>
    </Normal>
    <Uniform name="angle_dist">
      <lowerBound>5</lowerBound>
      <upperBound>85</upperBound>
    </Uniform>
  </Distributions>
  ...
</Simulation>
```

In the Distributions XML section, the stochastic model for the uncertainties are reported. In this case 2 distributions are defined:

- $vel\_dist \sim \mathbb{N}(30, 5)$, used to model the uncertainties associated with the *velocity*;
- $angle\_dist \sim \mathbb{U}(5, 85)$, used to model the uncertainties associated with the *angle*.

4. ***Samplers***:

raven/tests/framework/user_guide/StatisticalAnalysis/statisticalAnalysis.xml

```
<Simulation>
  ...
  <Samplers>
    <MonteCarlo name="my_mc">
      <samplerInit>
        <limit>1000</limit>
        <initialSeed>42</initialSeed>
      </samplerInit>
      <variable name="v0">
        <distribution>vel_dist</distribution>
      </variable>
      <variable name="angle">
        <distribution>angle_dist</distribution>
      </variable>
      <constant name="x0">0</constant>
      <constant name="y0">0</constant>
      <constant name="timeOption">1</constant>
    </MonteCarlo>
  </Samplers>
  ...
</Simulation>
```

In order to obtain the data-set on which the data mining algorithms are going to be applied, a *MonteCarlo* sampling approach is employed here.

5. ***DataObjects***:

raven/tests/framework/user_guide/StatisticalAnalysis/statisticalAnalysis.xml

```
<Simulation>
  ...
  <DataObjects>
    <PointSet name="samples">
      <Input>v0,angle</Input>
      <Output>r,t</Output>
    </PointSet>
    <PointSet name="dummyIN">
      <Input>v0,angle</Input>
      <Output>OutputPlaceHolder</Output>
    </PointSet>
    <PointSet name="statisticalAnalysis_basicStatPP">
      <Output>statisticalAnalysis_vars</Output>
    </PointSet>
    <HistorySet name="histories">
      <Input>v0,angle</Input>
      <Output>x,y,t</Output>
      <options>
        <pivotParameter>t</pivotParameter>
      </options>
    </HistorySet>
  </DataObjects>
  ...
</Simulation>
```

In this block, three *DataObjects* are defined: 1) PointSet named "samples" used to collect the final outcomes of the code, 2) PointSet named "dummyIN" used as a placeholder for the *Multirun* step, 3) HistorySet named "histories" in which the full time responses of the variables x, y, t are going to be stored.

6. ***Steps***:

raven/tests/framework/user_guide/StatisticalAnalysis/statisticalAnalysis.xml

```
<Simulation>
  ...
  <Steps>
    <MultiRun name="sampleMC">
      <Input class="DataObjects" type="PointSet">dummyIN</Input>
      <Model class="Models" type="ExternalModel">projectile</Model>
      <Sampler class="Samplers" type="MonteCarlo">my_mc</Sampler>
      <Output class="DataObjects" type="PointSet">samples</Output>
      <Output class="DataObjects" type="HistorySet">histories</Output>
    </MultiRun>
    <PostProcess name="statisticalAnalysisMC">
      <Input class="DataObjects" type="PointSet">samples</Input>
      <Model class="Models" type="PostProcessor">statisticalAnalysis</Model>
      <Output class="DataObjects" type="PointSet">statisticalAnalysis_basicStatPP</Output>
      <Output class="OutStreams" type="Print">statisticalAnalysis_basicStatPP_dump</Output>
    </PostProcess>
  </Steps>
  ...
</Simulation>
```

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 MonteCarlo 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 2-6 show all the results of the *PostProcess* step.

Table 2. Computed Moments and Cumulants

Computed Quantities	**r**	**t**
expected value	65.88	3.94
median	61.74	4.12
variance	1022.01	3.53
sigma	31.97	1.89
variation coefficient	0.48	0.48
skewness	0.55	-0.03
kurtosis	-0.01	-0.96
percentile 5%	20.21	0.85
percentile 95%	122.83	6.90

Table 3. Covariance matrix.

Covariance	**r**	**t**
velocity	95.36	3.29
angle	25.29	40.42

Table 4. Correlation matrix

Correlation	**r**	**t**
velocity	0.61	0.36
angle	0.03	0.92

Table 5. Variance Dependent Sensitivity matrix

Variance Sensitivity	**r**	**t**
velocity	-1.69	0.08
angle	-3.31	0.07

Table 6. Sensitivity matrix

Sensitivity (I/O)	**r**	**t**
velocity	3.95	0.12
angle	0.01	0.07

8 Data Mining through RAVEN

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.

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***:

   `raven/tests/framework/user_guide/DataMining/dataMiningAnalysis.xml`

   ```
   <Simulation>
     ...
     <RunInfo>
       <JobName>dataMiningAnalysis</JobName>
       <WorkingDir>dataMiningAnalysis</WorkingDir>
       <Sequence>sampleMC,kmeans,pca</Sequence>
       <batchSize>3</batchSize>
     </RunInfo>
     ...
   </Simulation>
   ```

   The *RunInfo* **Entity** is intended to set up the analysis sequence that needs to be performed. The number of steps specified in (`<Sequence>`) are sequentially run using the number of processors assigned in (`<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. ***Models***:
   raven/tests/framework/user_guide/DataMining/dataMiningAnalysis.xml

```
<Simulation>
  ...
  <Models>
    <ExternalModel ModuleToLoad="../../AnalyticModels/projectile.py" name="projectile" subType="">
      <variables>x,y,v0,angle,r,t,timeOption</variables>
    </ExternalModel>
    <PostProcessor name="KMeans1" subType="DataMining">
      <KDD labelFeature="klabels" lib="SciKitLearn">
        <SKLtype>cluster|KMeans</SKLtype>
        <Features>r,t</Features>
        <n_clusters>3</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>r,t</Features>
        <SKLtype>decomposition|PCA</SKLtype>
        <n_components>2</n_components>
      </KDD>
    </PostProcessor>
  </Models>
  ...
</Simulation>
```

   The goal of this example is to show how the data mining algorithms in RAVEN can be useful to analyze large data set.
   In addition to an External model, two Post-Processor models ($DataMining|cluster|KMeans$ and $DataMining|decomposition|PCA$) are specified. Note that the post-processing is performed on all the output FOMs used in this example ($randt$).

3. ***Distributions***:

   raven/tests/framework/user_guide/DataMining/dataMiningAnalysis.xml

```
<Simulation>
  ...
  <Distributions>
    <Normal name="vel_dist">
      <mean>30</mean>
      <sigma>5</sigma>
      <lowerBound>1</lowerBound>
      <upperBound>60</upperBound>
    </Normal>
    <Uniform name="angle_dist">
      <lowerBound>5</lowerBound>
      <upperBound>85</upperBound>
    </Uniform>
  </Distributions>
  ...
</Simulation>
```

In the Distributions XML section, the stochastic model for the uncertainties are reported. In this case 2 distributions are defined:

- $vel\_dist \sim \mathbb{N}(30, 5)$, used to model the uncertainties associated with the *velocity*;
- $angle\_dist \sim \mathbb{U}(5, 85)$, used to model the uncertainties associated with the *angle*.

4. ***Samplers***:

   raven/tests/framework/user_guide/DataMining/dataMiningAnalysis.xml

```xml
<Simulation>
  ...
  <Samplers>
    <MonteCarlo name="my_mc">
      <samplerInit>
        <limit>1000</limit>
        <initialSeed>42</initialSeed>
      </samplerInit>
      <variable name="v0">
        <distribution>vel_dist</distribution>
      </variable>
      <variable name="angle">
        <distribution>angle_dist</distribution>
      </variable>
      <constant name="x0">0</constant>
      <constant name="y0">0</constant>
      <constant name="timeOption">1</constant>
    </MonteCarlo>
  </Samplers>
  ...
</Simulation>
```

   In order to obtain the data-set on which the data mining algorithms are going to be applied, a *MonteCarlo* sampling approach is employed here.

5. ***DataObjects***:

   raven/tests/framework/user_guide/DataMining/dataMiningAnalysis.xml

```xml
<Simulation>
  ...
  <DataObjects>
    <PointSet name="samples">
      <Input>v0,angle</Input>
      <Output>r,t</Output>
    </PointSet>
    <PointSet name="kmeansSamples">
      <Input>v0,angle</Input>
      <Output>r,t,klabels</Output>
    </PointSet>
    <PointSet name="pcaSamples">
      <Input>v0,angle</Input>
      <Output>r,t,klabels,PCA1Dimension1,PCA1Dimension2</Output>
    </PointSet>
    <PointSet name="dummyIN">
      <Input>v0,angle</Input>
      <Output>OutputPlaceHolder</Output>
    </PointSet>
    <HistorySet name="histories">
      <Input>v0,angle</Input>
      <Output>x,y,t</Output>
      <options>
        <pivotParameter>t</pivotParameter>
      </options>
    </HistorySet>
  </DataObjects>
  ...
</Simulation>
```

In this block, three *DataObjects* are defined: 1) PointSet named "samples" used to collect the final outcomes of the code, 2) PointSet named "dummyIN" used as a placeholder for the *Multirun* step, 3) HistorySet named "histories" in which the full time responses of the variables x, y, t are going to be stored.

6. ***OutStreams***:

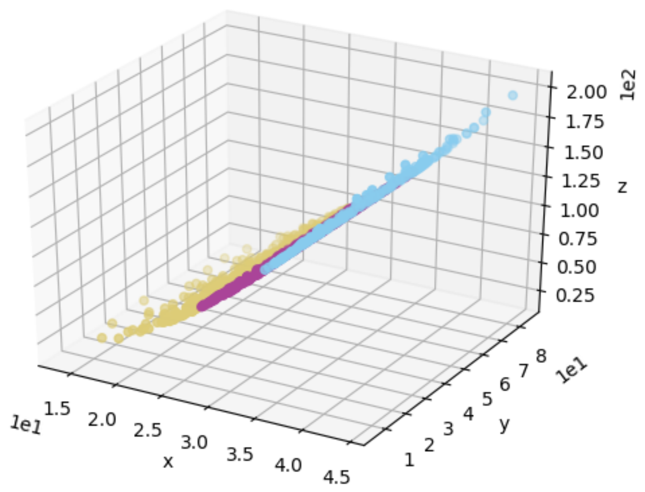


Figure 31. K-means clustering on original dataset.

raven/tests/framework/user_guide/DataMining/dataMiningAnalysis.xml

```
<Simulation>
  ...
  <OutStreams>
    <Print name="samplesDump">
      <type>csv</type>
      <source>kmeansSamples</source>
      <what>input,output,metadata|klabels</what>
    </Print>
    <Plot name="PlotKMeans1" overwrite="false">
      <plotSettings>
        <gridSpace>2 1</gridSpace>
        <plot>
          <type>dataMining</type>
          <SKLtype>cluster</SKLtype>
          <clusterLabels>kmeansSamples|Output|klabels</clusterLabels>
          <noClusters>3</noClusters>
          <x>kmeansSamples|Input|angle</x>
          <y>kmeansSamples|Output|r</y>
          <xlabel>angle</xlabel>
          <ylabel>range</ylabel>
          <gridLocation>
            <x>0</x>
            <y>0</y>
          </gridLocation>
          <range>
            <xmin>0</xmin>
            <xmax>100</xmax>
            <ymin>0</ymin>
            <ymax>200</ymax>
          </range>
        </plot>
        <plot>
          <type>dataMining</type>
          <SKLtype>cluster</SKLtype>
          <clusterLabels>kmeansSamples|Output|klabels</clusterLabels>
          <noClusters>3</noClusters>
          <x>kmeansSamples|Input|v0</x>
          <y>kmeansSamples|Output|r</y>
          <xlabel>velocity</xlabel>
          <ylabel>range</ylabel>
          <gridLocation>
            <x>1</x>
            <y>0</y>
          </gridLocation>
          <range>
            <xmin>0</xmin>
            <xmax>60</xmax>
            <ymin>0</ymin>
            <ymax>200</ymax>
          </range>
        </plot>
      </plotSettings>
      <actions>
        <how>png</how>
        <title>
          <text> </text>
        </title>
      </actions>
    </Plot>
    <Plot name="PlotLabels" overwrite="false">
      <plotSettings>
        <plot>
          <type>dataMining</type>
          <SKLtype>cluster</SKLtype>
          <clusterLabels>kmeansSamples|Output|klabels</clusterLabels>
          <noClusters>3</noClusters>
```

```
          <x>kmeansSamples|Input|v0</x>
          <y>kmeansSamples|Input|angle</y>
          <z>kmeansSamples|Output|r</z>
          <xlabel>velocity</xlabel>
          <ylabel>angle</ylabel>
          <zlabel>range</zlabel>
          <range>
            <xmin>0</xmin>
            <xmax>60</xmax>
            <ymin>0</ymin>
            <ymax>100</ymax>
            <zmin>0</zmin>
            <zmax>200</zmax>
          </range>
        </plot>
      </plotSettings>
      <actions>
        <how>png</how>
        <title>
          <text> </text>
        </title>
      </actions>
    </Plot>
    <Plot name="PlotPCA1" overwrite="false">
      <plotSettings>
        <plot>
          <type>dataMining</type>
          <SKLtype>cluster</SKLtype>
          <clusterLabels>pcaSamples|Output|klabels</clusterLabels>
          <noClusters>3</noClusters>
          <x>pcaSamples|Output|PCA1Dimension1</x>
          <y>pcaSamples|Output|PCA1Dimension2</y>
        </plot>
      </plotSettings>
      <actions>
        <how>png</how>
        <title>
          <text> </text>
        </title>
      </actions>
    </Plot>
  </OutStreams>
  ...
</Simulation>
```

This workflow uses one Print OutStream and three Plot OutStreams:

- "samplesDump", which writes the original sample set with the additional columns from the PostProcess steps,
- "PlotKMeans1", which plots the samples against the Figures of Merit with coloring according to the KMeans clustering,
- "PlotLabels", which plots the samples and colors them according to the KMeans clustering,
- "PlotPCA1," which plots the surrogate principal component dimensions and their associated clustering.

Note that a special kind of plot, the "dataMining" **<type>**, has been implemented to simplify plotting complicated results using RAVEN, and is used in all three of the plots in this workflow. Also note the use of the **<range>** block to define the data range of the plots created.

7. ***Steps***:

raven/tests/framework/user_guide/DataMining/dataMiningAnalysis.xml

```
<Simulation>
  ...
  <Steps>
    <MultiRun name="sampleMC">
      <Input class="DataObjects" type="PointSet">dummyIN</Input>
      <Model class="Models" type="ExternalModel">projectile</Model>
      <Sampler class="Samplers" type="MonteCarlo">my_mc</Sampler>
      <Output class="DataObjects" type="PointSet">samples</Output>
      <Output class="DataObjects" type="HistorySet">histories</Output>
    </MultiRun>
    <PostProcess name="kmeans" pauseAtEnd="True">
      <Input class="DataObjects" type="PointSet">samples</Input>
      <Model class="Models" type="PostProcessor">KMeans1</Model>
      <Output class="DataObjects" type="PointSet">kmeansSamples</Output>
      <Output class="OutStreams" type="Plot">PlotKMeans1</Output>
      <Output class="OutStreams" type="Plot">PlotLabels</Output>
    </PostProcess>
    <PostProcess name="pca" pauseAtEnd="True">
      <Input class="DataObjects" type="PointSet">kmeansSamples</Input>
      <Model class="Models" type="PostProcessor">PCA1</Model>
      <Output class="OutStreams" type="Print">samplesDump</Output>
      <Output class="DataObjects" type="PointSet">pcaSamples</Output>
      <Output class="OutStreams" type="Plot">PlotPCA1</Output>
    </PostProcess>
  </Steps>
  ...
</Simulation>
```

Finally, all the previously defined **Entities** can be combined in the `<Steps>` block; 3 `<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*.The `<Sampler>` is inputted to communicate to the *Step* that the driven code needs to be perturbed through the MonteCarlo 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 MonteCarlo 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 31 shows the clustering on the input space and the range, coloring according to the KMeans clustering,.
Figure 32 shows the clustering on the range-angle and range-velocity plots respectively.
Figure 33 shows the PCA decomposition on the data set.

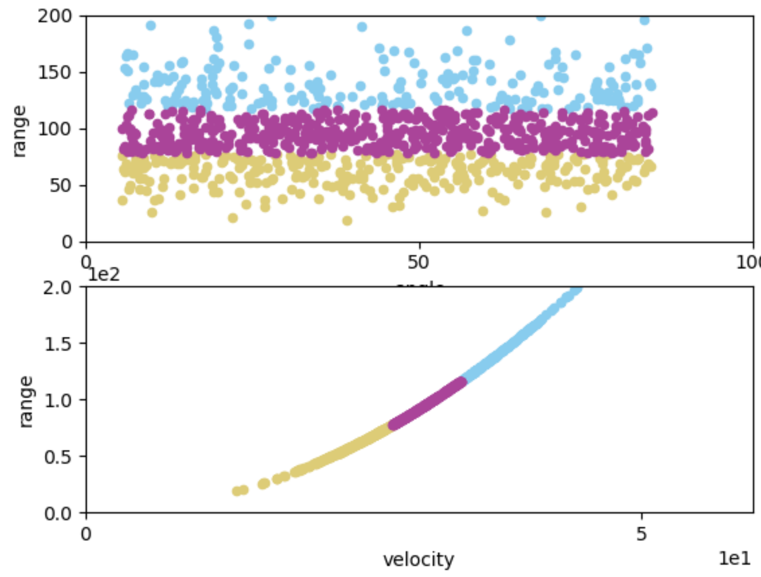


Figure 32. K-means clustering on projected parameters.

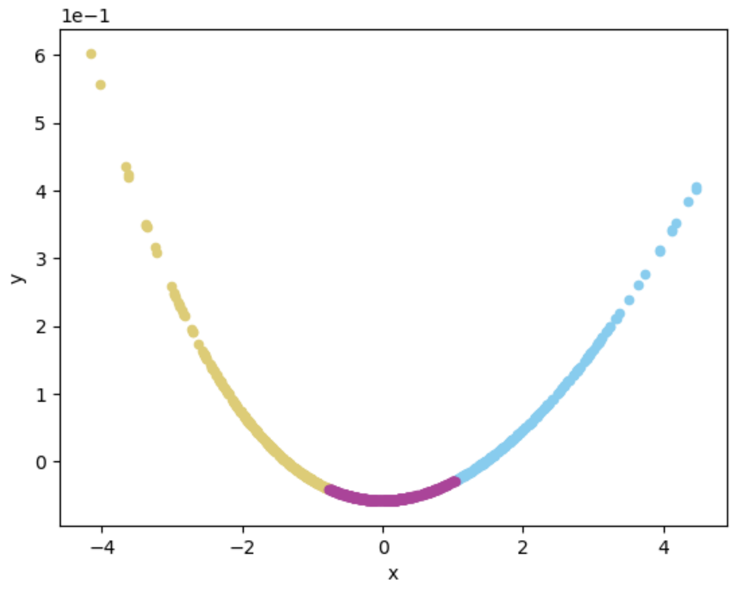


Figure 33. Principal Component Analysis.

9 Model Optimization

When analyzing the range of values obtainable by a model, frequently a key question is "what set of parameters result in the best response value?" To answer this question, RAVEN uses the **`<Optimizer>`**, a powerful sampler-like entity that searches the input space to find minimum or maximum values of a response.

In the remainder of this section, we will explore how to use the optimizer using a simple analytic problem, with a two-dimensional input space and single response of interest. After getting used to running with the optimizer, we will add increasing complexity, including changing adaptive step sizes, initial conditions, parallel trajectories, input space subdivision, input space constraints, and response constraints.

To demonstrate the operation of the Optimizer entities in RAVEN, the model we consider is the Beale function, which is documented in the analytic tests for RAVEN and replicated here:

- Function: $f(x, y) = (1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.625 - x + xy^3)^2$
- Domain: $-4.5 \leq x, y \leq 4.5$
- Global Minimum: $f(3, 0.5) = 0$

The two inputs are the variables x and y, and the response is a value we'll assign to ans, short for "answer". The model is an external model in RAVEN, and can be found at

```
raven/tests/framework/AnalyticModes/optimizing/beale.py.
```

The function's values are distributed as in Fig. 34, with red indicating high values and blue indicating low values. The objective is to minimize the function.

Note that throughout this example we use the SPSA optimizer by way of demonstration, since it is the first advanced algorithm for optimization included in RAVEN; many of the options and parameters apply to other optimizers, and details can be found in the RAVEN user manual.

9.1 Introduction: The Optimizer Input

As with other entities, the Optimizer gets its own XML block in the RAVEN input. Here's an example of an input for a SPSA optimizer named **`'opter'`**:

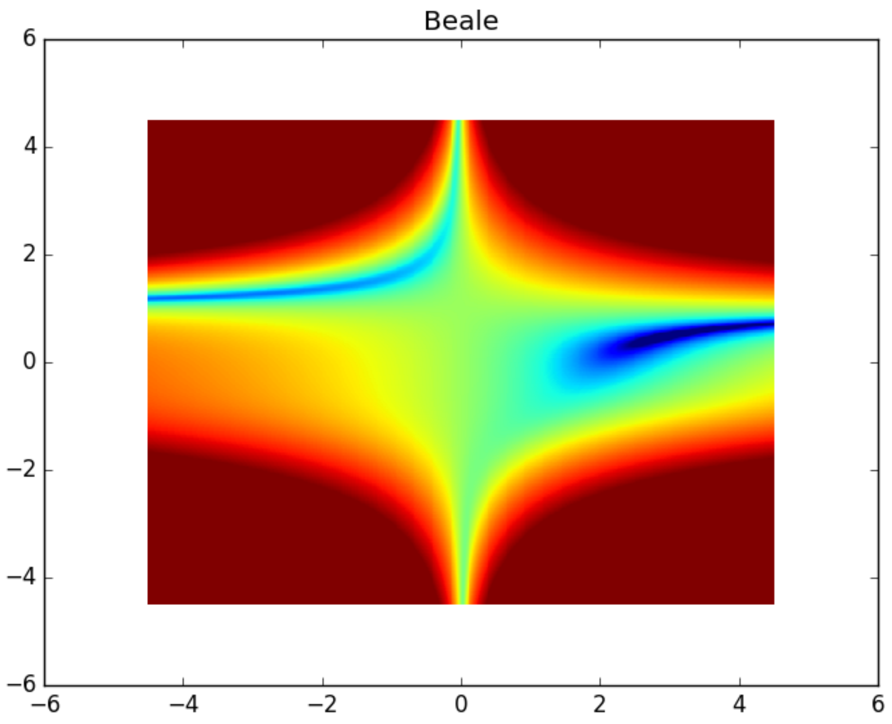


Figure 34. Plot of Beale's function for Optimization

raven/tests/framework/user_guide/optimizing/simple.xml

```
<Simulation>
  ...
  <Optimizers>
    <GradientDescent name="opter">
      <objective>ans</objective>
      <variable name="x">
        <distribution>beale_domain</distribution>
        <initial>0</initial>
      </variable>
      <variable name="y">
        <distribution>beale_domain</distribution>
        <initial>0</initial>
      </variable>
      <TargetEvaluation class="DataObjects" type="PointSet">optOut</TargetEvaluation>
      <samplerInit>
        <limit>5000</limit>
        <initialSeed>1234</initialSeed>
      </samplerInit>
      <gradient>
        <SPSA />
      </gradient>
      <stepSize>
        <GradientHistory />
      </stepSize>
      <acceptance>
        <Strict />
      </acceptance>
      <convergence>
        <gradient>1e-1</gradient>
      </convergence>
    </GradientDescent>
  </Optimizers>
  ...
</Simulation>
```

This is the smallest amount of input needed to run an optimization problem, with the exception that we include the **<initialSeed>** to maintain consistent results. Note the required blocks included to define the optimizer:

- **<objective>**, which is where you indicate the variable for which you want to find the minimum (or, if you change the default, maximum). As listed here, we want to minimize the value of `ans` given a range of possible values for `x` and `y`.
- **<variable>**, which is where you can define the input space variables, one for each of these nodes. Declaring a variable here informs the optimizer that you want it to find the optimal value for this variable, along with the other variables declared in their own blocks. Note this follows the same pattern as any other **<Sampler>**, including a **<distribution>** node to describe the domain of the variable. For **<GradientDescent>**, the shape of the distribution is not significant unless performing other advanced optimizations (such as optimization at risk). Nominally, this distribution simply defines the acceptable range of the variable, making the **<Uniform>** distribution a common choice. The distribution sets the upper and lower bounds of the variable, which will give the optimizer some general expectations for finding the optimal point; it will never try to sample a value smaller than the lower bound or larger than the upper bound. In the example we define variables *x* and *y* as our input variables, and both of them coincidentally range between -4.5 and 4.5. We set the initial values for both variables to 0 through the **<initial>** block, which is required in most cases; the exception is when a preconditioner sets them in mulitlevel optimization, but we're not concerned with that feature for this example.
- **<TargetEvaluation>**, which declares the DataObject that the optimization search evaluations are going to be placed in. All of the optimal points found as part of the optimization, as well as any other points evaluated as part of the algorithm, are placed in this object so the optimizer can retrieve this information later. When this data object is defined, it is critical that the objective variable is defined in the output space, and the input variables in the input space, so the optimizer can collect the results of its sampling. The data object type should be "PointSet" for this data object. In this example, we use the self-descriptive *optOut* data object.
- **<samplerInit>**, which contains initialization parameters for the optimization algorithm. In this case, we set an **<initialSeed>** to 1234 just to maintain consistent results. We

also set the maximum number of model evaluations through the **<limit>** node. We don't expect to need all these runs, but in case the optimizer is struggling, we set this cutoff to prevent the code running ad infinitum.

- **<gradient>**, which defines the gradient approximation algorithm to use within the gradient descent algorithm. In this case, we simply indicate that we want to use **<SPSA>**, and need no additional inputs.
- **<stepSize>**, which defines how we should control the step size during the gradient descent algorithm. There are several algorithms to choose from; in this case, we choose **<GradientHistory>**, which uses the scalar product between successive steps to determine what step to take in the search algorithm. A bigger growth factor results in traversing the input space more quickly, but converging more slowly. A bigger shrink factor results in collapsing to a minimum more quickly, converging quickly but possibly falling into local minima. We're using the default growth (1.25) and shrink (1.15) factors here.
- **<acceptance>**, which determines the algorithm by which we decide whether to accept a potential new optimal point during the gradient descent algorithm. In this case we use **<Strict>**, which indicates any potential new optimal points in the search that are not preferrable to the previously-found optimal point are discarded, and the search continues from the previously-found optimal point in the search.
- **<convergence>**, which informs the searching algorithm of when to decide it has found the optimal point within a sufficient tolerance. There are several stopping criteria; in this case, we use the local value of the **<gradient>**, which we want to be at most 0.1.

The other critical blocks in this input are as follows:

9.1.1 Models

raven/tests/framework/user_guide/optimizing/simple.xml

```xml
<Simulation>
  ...
  <Models>
    <ExternalModel ModuleToLoad="../../../../framework/AnalyticModels/optimizing/beale" name="beale" subType="">
      <inputs>x,y</inputs>
      <outputs>ans</outputs>
    </ExternalModel>
  </Models>
  ...
</Simulation>
```

Note that we define the external model with the name **'beale'** and provide a path to the analytic model itself. This model is set up with the `run` method that allows RAVEN to run the model. We also list all our input/output variables, *x, y*, and *ans*.

9.1.2 Data Objects

raven/tests/framework/user_guide/optimizing/simple.xml

```
<Simulation>
  ...
  <DataObjects>
    <PointSet name="placeholder" />
    <PointSet name="optOut">
      <Input>x,y</Input>
      <Output>ans</Output>
    </PointSet>
    <PointSet name="opt_export">
      <Input>trajID</Input>
      <Output>iteration,x,y,ans</Output>
    </PointSet>
  </DataObjects>
  ...
</Simulation>
```

We have three data objects:

- **'placeholder'**, which is necessary to define the input to our external model in the Steps (the external model doesn't use any input file, so we just use a placeholder here);
- **'optOut'**, which will hold all of the samples taken by our optimizer (optimal candidates, gradient evaluation points, rejected points, etc); and
- **'opt_export'**, which will hold the actual solution path taken by our optimizer. We store the path travelled by the optimization algorithm as successive samples, with `iteration` keeping track of the optimization steps taken. Note especially how the input of **'opt_export'** is set to `trajID`, which is a special keyword for the Optimizer trajectory tracking, as is the output variable `iteration`. There are several other special keyword outputs that can be written to the Solution Export data object, that can be found in the user manual.

9.1.3 Out Streams

raven/tests/framework/user_guide/optimizing/simple.xml

```
<Simulation>
  ...
  <OutStreams>
```

```
    <Print name="opt_export">
      <type>csv</type>
      <source>opt_export</source>
    </Print>
  </OutStreams>
  ...
</Simulation>
```

Here we define the way to print the output of our optimization algorithm. There's not much to note, except that we'll be printing the optimization path as a CSV.

9.1.4 Steps

raven/tests/framework/user_guide/optimizing/simple.xml

```
<Simulation>
  ...
  <Steps>
    <MultiRun name="optimize">
      <Input class="DataObjects" type="PointSet">placeholder</Input>
      <Model class="Models" type="ExternalModel">beale</Model>
      <Optimizer class="Optimizers" type="GradientDescent">opter</Optimizer>
      <SolutionExport class="DataObjects" type="HistorySet">opt_export</SolutionExport>
      <Output class="DataObjects" type="PointSet">optOut</Output>
    </MultiRun>
    <IOStep name="print">
      <Input class="DataObjects" type="HistorySet">opt_export</Input>
      <Output class="OutStreams" type="Print">opt_export</Output>
    </IOStep>
  </Steps>
  ...
</Simulation>
```

Here we put it all together into a work flow that RAVEN can follow. We only need two steps: one to optimize, and one to print out the results. To actually perform the optimization, we need a MultiRun step, which we cleverly name `'optimize'`. For input we take the placeholder data object *placeholder*, which sets up the input space of the model we defined, *beal*. Where a **`<Sampler>`** would normally go, we include the **`<Optimizer>`** we defined earlier. We output to the same data object we indicated in the Optimizer's **`<TargetEvaluation>`** node. Finally, we note specifically the use of the **`<SolutionExport>`** node. The data object defined in this node is where the Optimizer will write the optimization path history, with the final entry being the last step taken by the optimizer. The IOStep is unremarkable, used simply to write out the optimization path history to file.

9.1.5 Conclusion

After reviewing the components (don't forget the RunInfo block!), you can run this example and see the results. In particular, we can view the final results of the optimizer in `Simple/opt_export_0.csv`. Note that `opt_export` is the name of the **`<Print>`** OutStream we defined in the input file.

When we open the file (preferably in a CSV reader such a spreadsheet viewer), we see a CSV with several headers, the outputs defined in the data object in the input file: *trajID*, *iteration*, *x*, *y*, and *ans* (not necessarily in that order). *x*, *y*, and *ans* are the values of the variable at each optimization iteration, while *iteration* gives the sequential order of the optimization iteration. *trajID* is the trajectory identifying number; since we are only using one trajectory, this identifier is simply 0.

We can see there's only one line of data in the ouput CSV, showing the final solution discovered by the optimization algorithm. If we look at the line, we converged around $f(2.7, 0.42) = 0.0199$ in 40 steps, which is okay but still a little ways from the analytic optimal point $f(3, 0.5) = 0$. If we look at the output from the run, we can look at the last time RAVEN was "Checking convergence for Trajectory 0". Below that statement, there are a series of convergence criteria and their respeective statuses. We can see our convergence criteria requested through the input file (`gradient`, whose final accepted value is 0.0857) as well as the `same point` convergence criteria, which helps determine if the optimal solution is at a boundary even though other conditions have not converged.

We can see that the reason we converged at the end is the `gradient`, which means the relative change in the gradient of *ans* was sufficiently small between steps to cause convergence. Clearly, we claimed convergence prematurely because of the low value required in the optimizer input. Because these convergence criteria are very problem-specific, one set parameters will not work best for all problems.

We can improve this result by changing convergence parameters as well as step size growth and shrink factors, all of which can be found in the user manual, and many of which we'll discuss in the rest of this section. Feel free to experiment with these values, and see their affect on the solution discovered.

9.2 Increasing verbosity

We saw in the previous section that the output stored in `Simple/opt_export.csv` only includes the final optimal solution, and minimal information about that point. We can increase the output to see the entire path traversed by adding a few parameters in the input file.

The first parameter to add is in **`<Optimizers>`** **`<GradientDescent>`** **`<samplerInit>`**. By adding the node **`<writeSteps>`** with the value **`'every'`**, we can see the full path taken by the optimizer from initial point to final accepted solution.

Further, the optimizer has some special variables that can be use in the **<SolutionExport>** **<DataObject>** to print additional information to the CSV. For example, the special variable `accepted` will tell us, for each point in the optimization path, what the result of that point is. For SPSA, these acceptance notes can be one of the following:

- `first`, or the initial point at which the optimization search begins;
- `accepted`, if the new proposed point is sufficiently improved to be accepted as a new optimal point in the search;
- `rejected`, if the new proposed point is *not* sufficiently improved and therefore rejected;
- `rerun`, indicating the search algorithm returned to an old optimal point after rejecting a proposed optimal point; and
- `final`, which shows that the point listed is the final accepted and converged optimal point.

9.3 Initial Conditions and Parallel Trajectories

Notice we set the optimization search to start at $(0, 0)$. You can change this initial value through the **<initial>** block within the **<variable>** definition nodes.

Furthermore, RAVEN offers the possibility to run multiple optimization paths in parallel. Because many (perhaps most) optimization techniques get stuck in local minima, using multiple paths (or *trajectories* as they are called in RAVEN) increases the likelihood that one of the trajectories will find the global minimum point. You can request multiple trajectories by providing a variety of initial conditions in the **<initial>** block, as shown in this Optimizer example:

`raven/tests/framework/user_guide/optimizing/multiple_trajectories.xml`

```
<Simulation>
  ...
  <Optimizers>
    <GradientDescent name="opter">
      <objective>ans</objective>
      <variable name="x">
        <distribution>beale_domain</distribution>
        <initial>-2,-2,2,2</initial>
      </variable>
      <variable name="y">
        <distribution>beale_domain</distribution>
        <initial>-2,2,-2,2</initial>
      </variable>
      <TargetEvaluation class="DataObjects" type="PointSet">optOut</TargetEvaluation>
      <samplerInit>
        <limit>5000</limit>
        <initialSeed>1234</initialSeed>
        <writeSteps>every</writeSteps>
      </samplerInit>
      <gradient>
        <SPSA />
      </gradient>
      <stepSize>
        <GradientHistory>
          <growthFactor>1.2</growthFactor>
          <shrinkFactor>1.1</shrinkFactor>
        </GradientHistory>
      </stepSize>
      <acceptance>
        <Strict />
      </acceptance>
      <convergence>
        <gradient>1e-1</gradient>
      </convergence>
    </GradientDescent>
  </Optimizers>
  ...
</Simulation>
```

Note that the ordered pairs are split across the **<initial>** nodes, so that the first trajectory will start as a point made up of all the first entries, the second trajectory starts at all the second entries, and et cetera. In this case, we've requested starting points at (-2,-2), (-2,2), (2,-2), and (2,2). This (and defining a new working directory in the **<RunInfo>** block) is the only input change between the original file and this one.

When run, we can see the results in the working directory `MultipleTraj`. There, we see the same files as for the base case, plus `opt_export` files 0-3. These are produced because we've clustered the outputs by `trajID` in the **<OutStreams>** definition. Each of these corresponds to the path that one of the initial points started at, as you can see at the top of each of these CSV files. We can see that trajectory 2 (who started at (2,-2)) ended close to the analytic optimal point, while trajectory 1 was far from it.

In the screen output from the RAVEN run, you can see the final summary shows the status of each trajectory. Under *Trajectory Results* we see trajectories 1-3 all converged with different optimal values, while trajectory 0 is marked as *following 1*. This means at some point Trajectory 0 started following the same path as Trajectory 1 already moved along, so Trajectory 0 was terminated as a result to save computational resources.

Finally, we see the optimal point selected was Trajectory 2 with a function value of roughly 3.7e-3 at (3.15, 0.54).

9.4 Adjusting Adaptive Steps

As we've seen, some of the optimization paths are struggling to converge to meaningful optimal solutions. One way to improve this is to tinker with the convergence tolerances as shown in the user manual. Another is to change the step size modifications used as part of the search process, which we discuss in this section. First, we briefly discuss how the SPSA chooses its step size, so we can make informed choices about what parameters to use.

Because SPSA is a gradient-based method, it operates by starting at a particular point, estimating the gradient at that point, then taking a step in the opposite direction of the gradient in order to follow a downhill path. It adaptively chooses how long of a step to take based on its history. If the gradient is in the same direction twice in a row, the algorithm assumes there's further to travel, so increases its step size multiplicatively by the *growthFactor*, which we had defaulted to 1.25. If, on the other hand, the gradient switches directions, then the step size is divided by the *shrinkFactor*, which we had defaulted to 1.15. This means that by default, if the gradient keeps going in the same direction, you always increase your step size by 25%, while if you're bouncing back and forth in a valley, the step size is reduced by 15% at each iteration.

By way of note, in higher dimensions, the actual growth or shrink multiplier is scaled by a dot product between the two previous gradients, with a max of the grain growth factor when the dot product is 1 (exactly aligned) and a minimum of grain shrink factor when the dot product is -1 (exactly opposite). This means if the gradient is at right angles with the past gradient, then the step size remains unchanged (dot product is 0).

There are some additional considerations for the step size change, as well. If the algorithm takes a step, then discovers the new point has a worse response value than the point it's at, it will reject the new point, re-evaluate the gradient, and flag the step size to be divided by the gain shrink factor. Because of this, if the gain shrink factor is too large, false convergence can be obtained when the algorithm struggles to find a new downhill point to move to. As a result, in practice it is often beneficial to have a gain shrink factor that is smaller than the gain growth factor.

For this new example, we use gain growth factor of 1.25 (meaning when the gradient continues in the same direction our step grows by 25% of its old value) and a gain shrink factor of 1.1 (meaning when the gradient flips directions our step size shrinks to 90% of its old value). We add this to the base case (`simple.xml`) to get:

`raven/tests/framework/user_guide/optimizing/step_size.xml`

```
<Simulation>
  ...
  <Optimizers>
    <GradientDescent name="opter">
      <objective>ans</objective>
      <variable name="x">
        <distribution>beale_domain</distribution>
        <initial>0</initial>
      </variable>
      <variable name="y">
        <distribution>beale_domain</distribution>
        <initial>0</initial>
      </variable>
      <TargetEvaluation class="DataObjects" type="PointSet">optOut</TargetEvaluation>
      <samplerInit>
        <limit>5000</limit>
        <initialSeed>1234</initialSeed>
      </samplerInit>
      <gradient>
        <SPSA />
      </gradient>
      <stepSize>
```

```
        <GradientHistory>
          <growthFactor>1.25</growthFactor>
          <shrinkFactor>1.1</shrinkFactor>
        </GradientHistory>
      </stepSize>
      <acceptance>
        <Strict />
      </acceptance>
      <convergence>
        <gradient>1e-1</gradient>
      </convergence>
    </GradientDescent>
  </Optimizers>
  ...
</Simulation>
```

Note the definition of the gain growth and shrink factors in the **<convergence>** block. Reviewing the output file StepSize, we can see more steps were taken than the case using default step sizes, but the final solution was $f(2.75, 0.430) = 0.013$ in 40 iterations, which is closer to the analytical solution of $f(3, 0.5) = 0$ than the original case using the same number of iterations.

It is often challenging to find the best gain growth and shrink factors, and these can have a very significant impact on the speed and accuracy of the convergence process. Too large a shrink factor results in poor resolution of valleys, while too small a shrink factor results in many unnecessary evaluations of the model.

9.5 Functional Constraints

Sometimes an optimization problem has a constrained input space, possibly where there is a trade-off between two inputs. In this event, RAVEN allows the user to define a *constraint* function, which will cause RAVEN to treat this constraint as it would a boundary condition.

For example, we will introduce a void in the input where we reject inputs. This void is defined by rejecting all samples within $(x-1)^2 + y^2 < 1$. We'll also include the modified step growth and shrink parameters discussed in section 9.4.

To include a constraint function, we first have to define it in the RAVEN input as a **<Function>** entity:

raven/tests/framework/user_guide/optimizing/constrain.xml

```
<Simulation>
  ...
  <Functions>
    <External file="./constraint" name="constraint">
      <variables>x,y</variables>
    </External>
  </Functions>
  ...
</Simulation>
```

Note that the file ./Constrain/constraint.py is located relative to the working directory. Currently, external functions are always Python files. In that file, note that the only method is constrain, which is RAVEN's keyword to find the constraint function. RAVEN will pass in a self object, which will have the function variables defined in the **<Functions>** input available as members. The method constrain then returns a boolean which is True if the evaluation does not violate the constraint, or False if the constraint is violated.

To attach the constraint to the optimizer, simply add it as an assembled **<Function>**:

raven/tests/framework/user_guide/optimizing/constrain.xml

```
<Simulation>
  ...
  <Optimizers>
    <GradientDescent name="opter">
      <objective>ans</objective>
      <variable name="x">
        <distribution>beale_domain</distribution>
        <initial>0</initial>
      </variable>
      <variable name="y">
        <distribution>beale_domain</distribution>
        <initial>0</initial>
      </variable>
      <TargetEvaluation class="DataObjects" type="PointSet">optOut</TargetEvaluation>
      <samplerInit>
        <limit>5000</limit>
        <initialSeed>1234</initialSeed>
        <writeSteps>every</writeSteps>
      </samplerInit>
      <gradient>
        <SPSA />
      </gradient>
      <stepSize>
        <GradientHistory />
      </stepSize>
      <acceptance>
        <Strict />
      </acceptance>
      <convergence>
        <gradient>1e-1</gradient>
      </convergence>
      <Constraint class="Functions" type="External">constraint</Constraint>
    </GradientDescent>
  </Optimizers>
  ...
</Simulation>
```

After running, looking through the path followed by trajectory 0 shows that instead of following the path from section 9.4, the path moves to lower y values before swinging back up toward the optimal point.

10 EnsembleModel

In most RAVEN MultiRun steps, a sampler provides a variety of inputs to a model, whose outputs then produce a set of responses that can be used in many ways. However, sometimes a single model is insufficient to produce the response we want. When this happens, the EnsembleModel opens up a new vista of simulation options.

The EnsembleModel is a powerful tool for chaining multiple models together, with outputs of some models becoming inputs for others. Whether this is adding a preprocessing or postprocessing model, or linking multiple physics-based models together, the EnsembleModel provides the functionality to see many models together as a single model in a MultiRun step.

Note that the EnsembleModel combines multiple models for *each sample*; that is, the whole ensemble of models is run for each sample taken by RAVEN. If instead you want to postprocess a batch of sampled data, try the PostProcessor models instead.

10.1 Introduction: The EnsembleModel

The key to the EnsembleModel is careful definition of the inputs and outputs of each model. When an EnsembleModel is defined in a RAVEN run, RAVEN will automatically scan the inputs and outputs of each model and determine the right order to evaluate the models in (or *graph*).

10.2 Example: ballistics and impact

As a basic example, consider two physics models. The first is a ballistics code, used to determine the kinetic energy E of a ball when it hits the ground, with a path depending on its initial height y_0, initial velocity v_0, mass m and initial angle θ_0. We'll assume we're near the Earth's surface and aside from the initial launch height, we're launching over a flat surface. This can be represented as a functional $E(y_0, v_0, m, \theta_0)$.

The second physics model is an impact code that estimates the diameter of the crater D made when a ball impacts. With a few simplifying assumptions (D is under 1 km, the impact is near Earth's surface, the impact is in dry sand with fixed density), size D can be determined as a function of the ball's kinetic energy when impacting E, its mass m, and its radius r: $D(E, m, r)$.

We can see that the input E for the impact model is calculated by the ballistics code. In RAVEN the EnsembleModel will automatically detect this, and run the ballistics code before the impact code in each iteration.

In general, the EnsembleModel also has some finite capacity for resolving circular dependencies using Picard iteration. See the manual for more information on this capability.

Setting up an EnsembleModel can be more complex than other models, so we'll walk through the input using our example outlined above. For our purposes, we're going to let RAVEN perturb the codes using the GenericCode interface. We'll call the ballistics code `ballistics.py` with keyword input file `ballistics_input.txt` and similarly the impact code `impact.py` with keyword input file `impact_input.txt`.

The RAVEN input file for this example is located at

```
raven/tests/framework/user_guide/EnsembleModel/basic.xml
```

with run directory `run_basic`. The models and inputs are in the run directory.

Now we turn our attention to the RAVEN input file. We will discuss **<DataObjects>**, **<Files>**, **<Models>**, and **<Steps>** in detail; the rest are typical usage and need no special attention.

10.2.1 DataObjects

The key to a successful EnsembleModel is setting up the DataObject inputs and outputs. Each model has a **<TargetEvaluation>** DataObject that specifies what the inputs and outputs are for that model. In addition, any **<Output>** DataObjects in the **<MultiRun>** step can collect any or all of the variables used throughout the EnsembleModel calculation, either as inputs or outputs.

In this example, our convention is to name the **<TargetEvaluation>** DataObjects as **'model_data'**, replacing **'model'** with the model name. Additionally, we add a DataObject to store the final results of the **<MultiRun>** step:

raven/tests/framework/user_guide/EnsembleModel/basic.xml

```
<Simulation>
  ...
  <DataObjects>
    <PointSet name="ballistics_data">
      <Input>y0,v0,ang,m</Input>
      <Output>E</Output>
    </PointSet>
    <PointSet name="impact_data">
      <Input>E,m,r</Input>
      <Output>D</Output>
    </PointSet>
    <PointSet name="final_results">
```

```
      <Input>y0,v0,ang,m,r</Input>
      <Output>E,D</Output>
    </PointSet>
  </DataObjects>
  ...
</Simulation>
```

In this example, we see three DataObjects. **'ballistics_data'** determines the inputs and outputs of the `ballistics.py` model, so for its input space we have `y0,v0,ang,m`, which correspond to y_0, v_0, θ_0, m; for the output, we have `E` (for E).

The second DataObject, **'impact_data'**, delineates the inputs and outputs of the `impact.py` model, so for its input space we have `E,m,r`, while in the output space we have `D`. RAVEN will use these first two DataObjects to map the order in which these two models should be run (first ballistics, then impact) and transfer data from one to the next.

The third DataObject, **'final_results'**, can contain any information we want it to. In this case, we want to collect all the variables in their original spaces, so we take `y0,v0,ang,m,r` as inputs and `E,D` as outputs.

10.2.2 Files

Since we know both of our models take input files to set the variable values, we inform RAVEN about the template input files and give those files RAVEN names in the **<Files>** block. In this example, our input file for `ballistics.py` is `ballistics_template.txt`, which we simply name **'ballistics_input'**. Similarly, for `impact.py` the input file is `impact_template.txt`, which we name **'impact_input'**:

raven/tests/framework/user_guide/EnsembleModel/basic.xml

```
<Simulation>
  ...
  <Files>
    <Input name="ballistics_input">ballistics_template.txt</Input>
    <Input name="impact_input">impact_template.txt</Input>
  </Files>
  ...
</Simulation>
```

We will pause to note that the file `ballistics_template.txt` is different than the file `ballistics_input.py`, and similarly for the impact code. In the template file, we've replaced variable values with the `$RAVEN-$` wildcard, since we will be using the Generic Code model for these two models. Having a template input file may not be required for all code interfaces, but it is required for the Generic interface. See more about this interface in the Models section and in the user manual.

10.2.3 Models

Once we've specified the DataObjects and files, we are prepared to set up the **<Models>** block. To set up an EnsembleModel, we define each sub-module by itself as if it were a stand-alone RAVEN model, and then combine them by defining an **<EnsembleModel>**:

raven/tests/framework/user_guide/EnsembleModel/basic.xml

```
<Simulation>
  ...
  <Models>
    <Code name="ballistics" subType="GenericCode">
      <executable>run_basic/ballistics.py</executable>
      <clargs arg="python" type="prepend" />
      <clargs arg="-i" extension=".txt" type="input" />
      <clargs arg="-o" type="output" />
    </Code>
    <Code name="impact" subType="GenericCode">
      <executable>run_basic/impact.py</executable>
      <clargs arg="python" type="prepend" />
      <clargs arg="-i" extension=".txt" type="input" />
      <clargs arg="-o" type="output" />
    </Code>
    <EnsembleModel name="ballistic_and_impact" subType="">
      <Model class="Models" type="Code">ballistics
        <Input class="Files" type="">ballistics_input</Input>
        <TargetEvaluation class="DataObjects" type="PointSet">ballistics_data</TargetEvaluation>
      </Model>
      <Model class="Models" type="Code">impact
        <Input class="Files" type="">impact_input</Input>
        <TargetEvaluation class="DataObjects" type="PointSet">impact_data</TargetEvaluation>
      </Model>
    </EnsembleModel>
  </Models>
  ...
</Simulation>
```

The first model we will look at is the definition of the **'ballistics'** model. We note that is uses the **'GenericCode'** subType, so it will use the interface as described in the user manual, where wildcards in the input file are replaced with RAVEN's sampled values. The **<clargs>** (command line arguments) include the **'python'** command, the **'-i'** flag to specify the input file, and the **'-o'** flag to specify the output file. These nodes are specific to the Generic code interface and will not apply to all codes or models. The **'Impact'** model is defined in a similar manner.

With both Models defined, we can construct the **<EnsembleModel>**, which we've named **'ballistic_and_impact'** to describe the codes that are paired (you can name this whatever you want). There is no **subType** for the **<EnsembleModel>** currently, so we leave it blank.

Within the **<EnsembleModel>** node there are two **<Model>** nodes listed, which will inform the EnsembleModel which models need coupling. The order listed does not matter; the EnsembleModel will sort out the order based on the **<TargetEvaluation>** DataObject of each model.

The first **<Model>** we list within the **<EnsembleModel>** is the ballistics model. Note that the `class` and `type` are used along with the name to find the right model. Note also the name of the model goes in the *text* of the `<Model>` node, right after the first closing angle bracket. The subnodes of the `'ballistics'` model are the `<Input>`, which points to the input template input file we defined in the `<Files>` block; and the `<TargetEvaluation>`, which points to the `<PointSet>` we defined in the `<DataObjects>` block. These are critical to allowing the EnsembleModel both to run the `'ballistics'` model correctly, as well as the entire ensemble in a sensible manner.

In general, some models (such as a `<ROM>` or `<ExternalModel>`) do not require any files as inputs. In this case, the `<Input>` for an input-less model should be a placeholder DataObject, usually with the same input space as the `<TargetEvaluation>` DataObject, and a placeholder output. For more information, see examples running ExternalModels in these guides as well as the user manual.

After the `'ballistics'` model, we see a similar structure pointing to the `'impact'` model. The only differences between this model specification and the ballistics model specificatoin are the names of the `<Input>` file and `<TargetEvaluation>` DataObject.

With the two models defined, and the EnsembleModel assembled, no more work is required to prepare these two models to run. More complicated systems may require additional nodes in the EnsembleModel; see the user manual for details.

10.2.4 Steps

Finally, to put all the pieces together, we consider the `<Steps>` node:

raven/tests/framework/user_guide/EnsembleModel/basic.xml

```
<Simulation>
  ...
  <Steps>
    <MultiRun name="sample">
      <Input class="Files" type="">ballistics_input</Input>
      <Input class="Files" type="">impact_input</Input>
      <Model class="Models" type="EnsembleModel">ballistic_and_impact</Model>
      <Sampler class="Samplers" type="Grid">grid</Sampler>
      <Output class="DataObjects" type="PointSet">final_results</Output>
    </MultiRun>
    <IOStep name="print">
      <Input class="DataObjects" type="PointSet">final_results</Input>
      <Output class="OutStreams" type="Print">results</Output>
    </IOStep>
  </Steps>
  ...
</Simulation>
```

We take two steps in this simulation, one `<MultiRun>` called `'sample'` to sample the EnsembleModel on a grid, and one `<IOStep>` to print the results. These are constructed just as they would be with any other model; as far as the `<Steps>` are concerned, the EnsembleModel is like any other model. Note that we include the two template input files as **<Input>** nodes for the sampling step.

We do not discuss the **<Distributions>**, **<Samplers>**, or **<OutStreams>** here. These operate in general as they would in any other RAVEN input file. We will note, however, that the **<Grid>** sampler named **'grid'** samples all the inputs that are needed by either of the submodels, unless the inputs are provided by another model. In this case, that means we need to sample y_0, θ_0, v_0, m, r but we do not sample E since it is calculated by the **'ballistics'** model.

We can find all the sampled and calculated values from both of the models in the output produced by this run, found at

```
raven/tests/framework/user_guide/EnsembleModel/run_basic/results.csv
```

10.3 ExternalModel in EnsembleModel

There are a few details that can make handling **<ExternalModel>** models challenging in EnsembleModel. We'll cover a few of these to help smooth over some potential bumps.

10.3.1 Input Placeholder DataObject

Firstly, as noted above, both ExternalModel and ROM differ from Code models in a significant way: they (usually) do not require any input files. So what do you put as the **<Input>** for an ExternalModel or ROM? To maintain consistency, we leave the **<Input>** node in place. Instead of specifying a file however, a placeholder DataObject is used instead. A placeholder DataObject has variables listed in its input, but for output either the **<Output>** node is omitted, or the special keyword `OutputPlaceHolder` is used instead. For example, see the excerpt from a regression test:

raven/tests/framework/ensembleModelTests/index_input_output.xml

```
<Simulation>
  ...
  <DataObjects>
    <PointSet name="first_in">
      <Input>scalar1</Input>
      <Output>OutputPlaceHolder</Output>
    </PointSet>
    <PointSet name="second_in">
      <Input>scalar2</Input>
      <Output>OutputPlaceHolder</Output>
    </PointSet>
    <HistorySet name="step_out">
      <Input>scalar1,scalar2</Input>
      <Output>hist1,hist2</Output>
      <options>
        <pivotParameter>t</pivotParameter>
      </options>
    </HistorySet>
    <DataSet name="first_target">
      <Input>scalar1</Input>
      <Output>hist1</Output>
      <Index var="t">hist1</Index>
    </DataSet>
    <DataSet name="second_target">
      <Input>scalar2,hist1</Input>
      <Output>hist2</Output>
      <Index var="t">hist1,hist2</Index>
    </DataSet>
  </DataObjects>
  ...
</Simulation>
```

The inputs **'first_in'** and **'second_in'** are placeholder DataObject, with a scalar input variable for the **<Input>** and placeholder OutputPlaceHolder as **<Output>**. It is always a **<PointSet>** type of DataObject. The key `OutputPlaceHolder` is recognized specially in RAVEN and translates into an empty output space. This type of DataObject is the only suitable input for an ExternalModel or ROM without an input file.

10.3.2 DataSet

Because of the complexity of EnsembleModel, it is possible to have mixed scalars and vectors as variable inputs to a particular model in the Ensemble. Furthermore, it is possible to have multiple vector variables that depend on different independent variables (for example, "time" and "space"). To accommodate this, the **<DataSet>** DataObject can be used. More can be found on this data structure in the RAVEN user manual.

10.3.3 Scalar Variables

Because of the flexibility of the EnsembleModel, and to maintain a level of consistency across all variables, scalar variables in the EnsembleModel are stored as Numpy arrays with length 1. In general this should be transparent for most operations; however, occasionally it may be required to access a scalar variable by accessing it at index 0. For example, see the variable "scalar2" in a model used in the regression tests for the EnsembleModel:

`raven/tests/framework/ensembleModelTests/IndexInputOutput/model2.py`

```
import numpy as np

def run(obj,dct):
  obj.t #just making sure it's available here
  obj.hist2 = obj.hist1 + obj.scalar2[0]
```

10.3.4 Independent Variables

One feature of ExternalModel is their capacity to take variables exactly as they are from RAVEN, without going through file IO. As a result, EnsembleModel frequently contain models that produce pivot-dependent data that can be used as inputs for other models in the Ensemble. Several terms aptly describe these variables on which other variables depend, including "independent variables," "indexes", and "pivots". In general, we refer to them as "indexes". While the index is often "time" or something similar, it can be any monotonically-increasing variable.

However, when it is stored in RAVEN, the indexes are not stored like other variables, because it is a coordinate axes that supports other variables. This allows RAVEN to do a great many flexible operations internally. However, it also means that any time you want to pass an index variable from one model to an ExternalModel, it can only come attached to another variable that depends on that input.

For example, consider two models. In the first model, a path followed by a charged particle is traced out in time. The outputs of the first model are time t, lateral position x, and vertical position y. The second model calculates the energy of the particle at each point in time; however, the second model is written such that it does not require any specific time values (it just reads them from the input).

To run these two models in Ensemble configuration, **`<TargetEvaluation>`** DataObject of the second model must request one of the time-dependent variables from the first model, x or y. If t is directly requested, it will not be made available in the second ExternalModel. Additionally, in the **`<Model>`** definition, all variables (dependent or index) should be listed. In this example, when defining the model, t should be listed as well as x in the **`<variables>`** node in the second **`<Model>`** definition.

For an example of time dependent data passing, consider the regression test example at

```
raven/tests/framework/ensembleModelTests/index_intput_output.xml
```

Appendices

A Document Version Information

6a89f3b0f6f89e625fca6969b6f8548c6e7eb3c5 Paul Talbot Thu, 11 Jul 2024 10:56:49 -0600

References

[1] "Neams: The nuclear energy advanced modeling and simulation program," Tech. Rep. ANL/NE-13/5.

[2] "Light water reactor sustainability program integrated program plan," Tech. Rep. INL-EXT-11-23452, April 2013.

[3] R.-D. development team, "Relap5/mod3.3 code manual," tech. rep., Idaho National Laboratory, October 2015.

[4] C. Rabiti, A. Alfonsi, J. Cogliati, D. Mandelli, R. Kinoshita, and S. Sen, "Raven user manual," tech. rep., Idaho National Laboratory, 2015.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

v1.28

Idaho National Laboratory