MANUAL

INL/EXT-TODO
Revision 0
Printed November 13, 2022

RAVEN Plugins Manual

Paul W. Talbot, Congjian Wang

Prepared by
Idaho National Laboratory
Idaho Falls, Idaho 83415

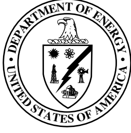
The Idaho National Laboratory is a multiprogram laboratory operated by
Battelle Energy Alliance for the United States Department of Energy
under DOE Idaho Operations Office. Contract DE-AC07-05ID14517.

Approved for unlimited release.

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

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

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



INL/EXT-TODO
Revision 0
Printed November 13, 2022

RAVEN Plugins Manual

Paul W. Talbot
Congjian Wang

Contents

1 Introduction 7
1.1 What are Plugins? 7
1.2 What isn't a Plugin? 7
1.3 How do I decide? 7
1.4 Supported Plugins 8
2 Making a New RAVEN Plugin 10
2.1 Plugin Structure 10
2.2 Plugin Entities 10
2.3 Common Methods for Plugin Entities 12
2.3.1 Method: `getInputSpecification` 12
2.3.2 Method: `__init__` 12
2.3.3 Method: `handleInput or _handleInput` 13
2.3.4 Method: `initialize` 13
2.4 Additional Libraries 15
2.5 Installing in RAVEN 15
2.6 Using the Plugin in RAVEN 15
2.7 Adding Testers 16
3 ExternalModel Plugins 17
3.1 ExternalModel Plugin Input 17
3.2 ExternalModel Plugin Creation 19
3.2.1 Method: `run` 19
3.2.2 Method: `createNewInput` 20
3.2.3 Method: `_readMoreXML` 21
3.2.4 Method: `initialize` 22
4 OutStreams Plot Plugins 23
4.1 Optional Methods 24
4.2 `run` method, required 24
4.3 initialization, optional 24
5 Code Plugins 25
5.0.1 Method: `_readMoreXML` 27
6 PostProcessor Plugins 29
Appendices 31
A Document Version Information 31
References 32

1 Introduction

RAVEN (Risk Analysis Virtual ENvironment) [1,2] is a framework for risk- and uncertainty-based stochastic analysis.

Occasionally, the applications of RAVEN become sufficiently varied and complex to warrant a special application to a particular suite of problems. These special applications may introduce new physics, new templates, new RAVEN entities, or any combination of the above. In this instance, it's worth considering whether developing a Plugin could be beneficial to you and the community.

1.1 What are Plugins?

Plugins are defined the RAVEN Software Quality Assurance documentation [2]. They are intended to allow development that does not fit in the scope of RAVENs mission to be completed in a way that is still captured under the umbrella of RAVEN-based tools.

1.2 What isn't a Plugin?

While Plugins have wide applicability to extending the functionaliy of RAVEN, there are some alternatives that may be more suitable for some developments.

A Plugin is not simple a RAVEN Template. RAVEN Templates are used to simplify complex RAVEN workflows for more restrictive but increasingly efficient workflow permutations. If the end result of development is only a new RAVEN Template, we recommend you contribute this as a new templated workflow rather than a separate Plugin.

A Plugin is not a set of models that do similar activities to what RAVEN already does. For example, if new development includes RAVEN postprocessors and RAVEN metrics for uncertainty analysis and clustering using a particular theory, this is well within RAVENs scope and should be directly contributed to the code so these additions can benefit the community directly without the Plugin interaction.

1.3 How do I decide?

Ultimately, the decision to develop a plugin is best discussed with the RAVEN core team. However, the following guidelines may help inform a good decision.

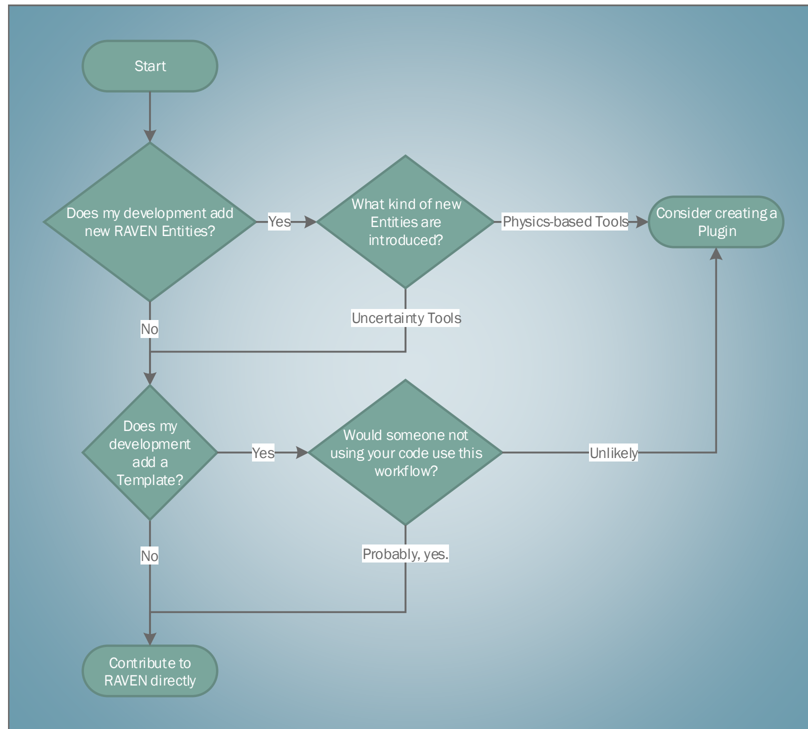


Figure 1. Choosing to Plugin Workflow

1.4 Supported Plugins

Officially supported RAVEN Plugins are a subset of all Plugins filtered by a particular set of characteristics. It is intended that all official Plugins are compatible with the latest developments in RAVEN as well as with each other. Additionally, official plugins contain continuous integration (CI) testing and provide the necessary software quality assurance (SQA) documentation to be included in RAVEN's SQA plan.

To be an officially supported RAVEN Plugin, the following prerequesites must be met:

- The appropriate SQA documentation for NQA-1 level 2 coverage under the RAVEN SQA must be in place for the Plugin.
- The Plugin must contain CI testing with coverage consistent with RAVEN SQA. This CI testing must be compatible with RAVEN's CI testing system.
- The Plugin must contain documentation explaining the use and options included in the Plugin's contents.

- If at any time a change in RAVEN causes a failure in the CI tests in the plugin, the plugin must be updated to be compatible with the new RAVEN.

If a Plugin fails any of these criteria, a grace period (usually a month) is provided along with notification to the Plugin developers. If the grace period expires and the Plugin does not meet the requirements, it may be removed from the official supported Plugins list, pending an update to the Plugin.

2 Making a New RAVEN Plugin

Creating a new plugin is a straightforward process. It involves setting up a repository, establishing a basic structure, and installing in RAVEN for testing.

2.1 Plugin Structure

The following directories and `__init__.py` must be present in the main directory of the plugin in order for RAVEN to read it correctly as shown in figure 2:

- `src`, where the entities for RAVEN to load are located;
- `doc`, where the documentation for the plugin and its entities is located.
- `tests`, where continuous integration tests are located;
- `__init__.py`, where the plugin entities need to be loaded.

Here is the example for `__init__.py` from the `raven/plugins/ExamplePlugin`:

```
### External Model
from ExamplePlugin.src import SumOfExponential
### ROM
from ExamplePlugin.src import LinearROM
### Outstreams
from ExamplePlugin.src import CorrelationPlot
### PostProcessors
from ExamplePlugin.src import testInterfacedPP
from ExamplePlugin.src import testInterfacedPP_PointSet
```

If the plugin developer wants to make his plugin an official supported plugin in RAVEN (by the submodule system), he needs to check the raven wiki under the "contribution" section), and this plugin needs to be placed in a folder (whatever name) located in (see figure 2 for example):

```
path/to/raven/plugins/
```

2.2 Plugin Entities

The procedure of adding a plugin entity for RAVEN is a straightforward process. The addition of the entity does not require modifying RAVEN itself. Instead, the developer creates a new Python

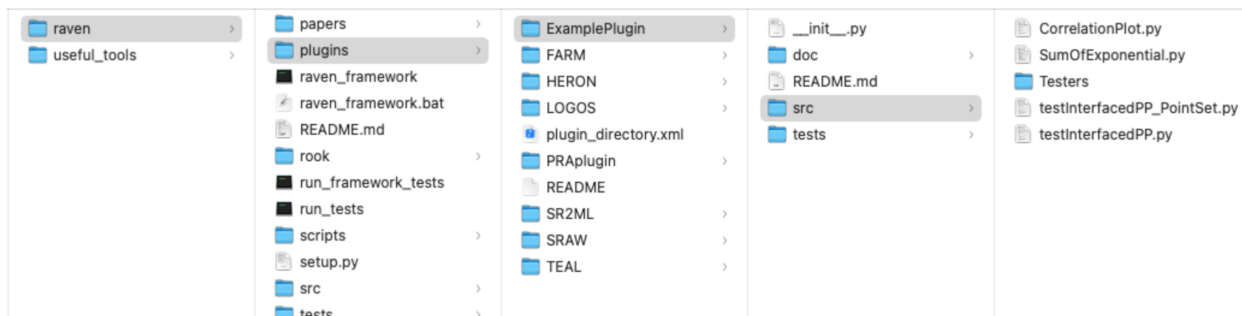


Figure 2. Plugins Location

module inherited from plugin base class that is going to be embedded in RAVEN at run-time (no need to introduce hard-coded statements). These RAVEN plugin base classes are located at `raven/ravenframework/PluginBaseClasses`:

```
ExternalModelPluginBase in ExternalModelPluginBase.py
PlotPlugin in OutStreamPlotPlugin.py
PostProcessorPluginBase in PostProcessorPluginBase.py
SupervisedLearningPlugin in SupervisedLearningPlugin.py
```

These classes can be loaded as follows in the Python modules created by the plugin developers:

```
from ravenframework.PluginBaseClasses.ExternalModelPluginBase import ExternalModelPluginBase
from ravenframework.PluginBaseClasses.OutStreamPlotPlugin import PlotPlugin
from ravenframework.PluginBaseClasses.PostProcessorPluginBase import PostProcessorPluginBase
from ravenframework.PluginBaseClasses.SupervisedLearningPlugin import SupervisedLearningPlugin
from ravenframework.PluginBaseClasses.CodePluginBase import CodePluginBase

class NewExternalModelPluginEntity(ExternalModelPluginBase):
  ...

class NewPlotPluginEntity(PlotPlugin):
  ...

class NewPostProcessorPluginEntity(PostProcessorPluginBase):
  ...

class NewROMPluginEntity(SupervisedLearningPlugin):
  ...

class NewCodePluginEntity(CodePluginBase):
  ...
```

The APIs for these classes can be either found in the following sections or directly in the modules themselves.

2.3 Common Methods for Plugin Entities

2.3.1 Method: getInputSpecification

RAVEN uses InputData module to describe data types, check and read XML inputs (see https://github.com/idaholab/raven/wiki/input-data for more detailed description). In order to use this feature, the plugin developers need to load InputData and InputTypes modules in the developed plugin entity class, i.e.,

```
from ravenframework.utils import InputData, InputTypes
```

And add a class method called getInputSpecification:

```
@classmethod
def getInputSpecification(cls):
  """
    Method to get a reference to a class that specifies the input
    data for class cls.
    @ In, None
    @ Out, specs, InputData.ParameterInput, class to use for
      specifying input of cls.
  """
  specs = super().getInputSpecification()
  specs.addParam("xmlAttributeName", param_type=InputTypes.StringType, required=True, default='no-
      default', descr='')
  subNode = InputData.parameterInputFactory('subnodeName', contentType=InputTypes.StringType,
      default='no-default', decr='')
  subNode.addParam("xmlSubNodeAttributeName", param_type=InputTypes.StringType, required=True,
      default='no-default', descr='')
  specs.addSub(subNode)
  return specs
```

Since the plugin entities inherit from RAVEN plugin base class, they can get the parent input specification, and use that as a start (i.e., specs = super().getInputSpecification()).

2.3.2 Method: __init__

As the constructor for Python classes, the __init__ method should be extended to define any instance variables used in the plugin class. If no instance variables are used, this may be ommitted.

Any call to __init__ must include a call to the parent's constructor, such as

```
def __init__(self):
  """ ... """
  super().__init__()
```

This assures access to basic RAVEN functionalities required to use the plugin.

2.3.3 Method: **handleInput or _handleInput**

This API is used by RAVEN to process the input specifications generated by `getInputSpecification`. The input to this method is the input specification class instance returned from `getInputSpecification`, with `parseNode` function from this class instance has been already called on the RAVEN XML node (See `https://github.com/idaholab/raven/wiki/input-data` for more detailed description).

```
def _handleInput(self, specs):
  """
    Function to handle the parameter input.
    @ In, specs, InputData.ParameterInput, the already-parsed input.
    @ Out, None
  """
  super()._handleInput(specs)
```

Depend on the APIs of entities, either `handleInput or _handleInput` can be used. Similar to other methods, any call to `handleInput or _handleInput` must include a call to the paraent's method using `super()`.

2.3.4 Method: **initialize**

The `initialize` method can be implemented in the plugin entities in order to initialize some variables needed by it. For example, it can be used to compute a quantity needed by the `run` method in **ExternalModel** plugin. RAVEN is going to call it at the initialization stage of each **Step**. RAVEN will communicate, through a set of method attributes, all the information that are generally needed to perform an initialization:

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

  - `NumThreads` – number of threads by run
  - `numProcByRun` – total number of core used by one run (number of threads by number of mpi)
  - `batchSize` – number of contemporaneous runs
  - `ParallelCommand` – the command that should be used to submit jobs in parallel (mpi)
  - `numNode` – number of nodes
  - `procByNode` – number of processors by node
  - `totalNumCoresUsed` – total number of cores used by driver
  - `queueingSoftware` – queueing software name
  - `stepName` – the name of the step currently running
  - `precommand` – added to the front of the command that is run
  - `postcommand` – added after the command that is run
  - `delSucLogFiles` – if a simulation (code run) has not failed, delete the relative log file (if True)
  - `deleteOutExtension` – if a simulation (code run) has not failed, delete the relative output files with the listed extension (comma separated list, for example: 'e,r,txt')
  - `mode` – running mode, curently the only mode supported is mpi (but custom modes can be created)
  - *expectedTime* – how long the complete input is expected to run
  - *logfileBuffer* – logfile buffer size in bytes

- `inputs`, a list of all the inputs that have been specified in the "Step" using this model.
- `initDict`, a dictionary with initialization options.

```
def initialize(self, runInfo, inputs, initDict=None):
  """
    Method to initialize this object before use in each Step
    @ In, runInfo, dict, dictionary of run info
          (e.g. working dir, etc)
    @ In, inputs, list, list of inputs
    @ In, initDict, dict, optional, dictionary with
          initialization options
    @ Out, None
  """
  super().initialize(runInfo, inputs, initDict)
```

2.4 Additional Libraries

If the plugin requires additional libraries, they can create the `dependencies.xml` file in the same manner as RAVEN's dependencies file. Only the libraries that are required by the plugin but not listed in `raven/dependencies.xml` are required to be added in the `pluginName/dependencies.xml` In this case, libraries will be added like they are for RAVEN itself, and a check will be performed to assure no base RAVEN (or other plugin) dependencies are modified.

2.5 Installing in RAVEN

Use the installation script in `raven/scripts/install_plugins.py`.

```
raven/scripts/install_plugins.py -s /abs/path/to/pluginName
```

Replacing `pluginName` with the path to your plugin and the name of the directory, such as `/user/projects/raven/plugins/pluginName`. Use the absolute path to your new plugin to avoid any navigation problems. If installing an officially-supported plugin that you do not plan on modifying, the following command can be run (using TEAL as the example plugin):

```
raven/scripts/install_plugins.py -s TEAL
```

Note the path was eliminated. This will initialize (or update) the official plugin in `raven/plugins` with the official submoduled version. To install all officially-supported plugins, the shortcut option `-a` or `--all` can be used:

```
raven/scripts/install_plugins.py -a
```

At this stage, RAVEN will import all the plugins within that directory and perform some error checking.

This process automatically registers the plugin in the plugin directory, and informs the plugin about RAVEN

2.6 Using the Plugin in RAVEN

Once registered/installed, the new plugin entity (i.e., inherited from ExternalModel, ROM, PostProcessor) can be used in RAVEN input file by using corresponding RAVEN entity and its **subType** to load the plugin entity. The **subType** is defined by your plugin name and your plugin entity name separated by '.'. For example, if your plugin external model class is named "myPluginModel", you can access an external model in the RAVEN input as

```
<Models>
  ...
  <ExternalModel name='myName'
     subType='myPluginName.myPluginModel'>
    ...
  </ExternalModel>
  ...
</Models>
```

2.7 Adding Testers

RAVEN automatically provides a testing harness for automated regression testing. This includes a variety of *testers*, such as CSV checkers and XML checkers.

If a plugin requires additional testers for regression testing, they can be added to the plugin and loaded by RAVEN's test harness at testing time.

Any new testers should be added under a folder named `Testers` in the `src` directory of the plugin. For example, for a plugin named `examplePlugin` and a tester named `myNewTester`:

```
  /path/to/examplePlugin/src/Testers/myNewTester.py
```

Any discovered testers will be made available to the `tests` files used by the RAVEN regression test system; for example:

```
[Tests]
  [./aTestForMyPlugin]
    type = myNewTester
    input = my_test_input.xml
    csv = 'TestWorkingDir/results.csv'
  [../]
[]
```

For more information on inheriting from and creating new testers, see the RAVEN regression system documentation.

Entities in RAVEN that are ready for new strategies via Plugins are described in the following sections.

3 ExternalModel Plugins

The procedure of adding a plugin for the ExternalModel is a straightforward process. At the initialization stage, RAVEN imports all the Plugins that are contained in this directory and performs some preliminary cross-checks.
It is important to notice that the name of class in the Plugin module is the one the user needs to specify when the new plugin needs to be used. For example, if the Plugin module contains the class "NewPlugin", the *subType* in the **<ExternalModel>** block will be "pluginName.NewPlugin":

```
class NewPlugin(ExternalModelPluginBase):
  ...
```

```
  ...
  <ExternalModel name='whatever'
     subType='pluginName.NewPlugin'>
   ...
  </ExternalModel>
  ...
</Models>
```

In the following sub-sections, a step-by-step procedure for creating a new ExternalModel plugin is outlined.

3.1 ExternalModel Plugin Input

When a new ExternalModel plugin is developed, its RAVEN input is almost identical to the general ExternalModel entity (see **??**). The specifications of an ExternalModel Plugin must be defined within the XML block **<ExternalModel>**. This XML node needs to contain the attributes:

- **name**, ***required string attribute***, user-defined name of this External Model. **Note:** As with the other objects, this is the name that can be used to refer to this specific entity from other input blocks in the XML.
- **subType**, ***required string attribute***, must be equal to the plugin type, e.g., **PluginName.PluginModuleName**. For example, TEAL.CashFlow. **Note:** In case a plugin is requested (through the **subType** attribute) the attribute **ModuleToLoad** must not be inputted.

In order to make the RAVEN code aware of the variables the user is going to manipulate/use in her/his ExternalModel Plugin, the variables need to be specified in the **<ExternalModel>** input block. The user needs to input, within this block, only the variables that RAVEN needs to be aware of (i.e. the variables are going to directly be used by the Plugin) and not the local variables

that the ExternalModel Plugin developer does not want to, for example, store in a RAVEN internal object. These variables are specified within a **<inputs>** and **<outputs>** or **<variables>** blocks:

- **<variables>**, ***string, required parameter***. Comma-separated list of variable names. Each variable name needs to match a variable used/defined in the external python model. **Note:** this node is is deprecated and will be removed in future releas of RAVEN in favor of the two following nodes **<inputs>** and **<outputs>**

- **<inputs>**, ***string, required parameter***. Comma-separated list of input variable names. Each variable name needs to match a variable used/defined in the external python model.

- **<outputs>**, ***string, required parameter***. Comma-separated list of output variable names. Each variable name needs to match a variable used/defined in the external python model.

In addition, if the user wants to use the alias system, the following XML block can be used:

```
<Models>
 ...
 <ExternalModel name='myName' subType='myPluginName.myPluginModel'>
   ...
   <inputs>ExternalModelInputVariableNameList</inputs>
   <outputs>ExternalModelOutputVariableNameList</outputs>
   <alias variable='RavenAliasForInput' type='input'>ExternalModelInputVariableName</alias>
   <alias variable='RavenAliasForOutput' type='output'>ExternalModelOutputVariableName</alias>
 </ExternalModel>
 ...
</Models>
```

The description of the alias is:

- **<alias>** ***string, optional field*** specifies alias for any variable of interest in the input or output space for the model. In the body of this node the user specifies the name of the variable that the model is going to use (during its execution). The actual alias, usable throughout the RAVEN input, is instead defined in the **variable** attribute of this tag.
  The user can specify aliases for both the input and the output space. As sanity check, RAVEN requires an additional required attribute **type**. This attribute can be either “input” or “output”. **Note:** The user can specify as many aliases as needed.

When the Plugin variables are defined, at run time, RAVEN initializes them and tracks their values during the simulation. Each variable defined in the **<ExternalModel>** block is available in the Plugin class (in each implemented method) as the object “container” that “acts” as a Python “self”. For example,

```
  def run (self, container, inputs):
    print(container.variableA)
```

3.2 ExternalModel Plugin Creation

As already mentioned, RAVEN imports all the "ExternalModel Plugins" at run-time. In order to make RAVEN able to drive a newer ExternalModel plugin, the developer needs to code a Python class containing few methods (with strict syntax) that are called by RAVEN during the simulation. Every new "ExternalModel Plugin" must inherit from a RAVEN base class named $ExternalModelPluginBase$:

```
class NewPlugin(ExternalModelPluginBase):
  ...
```

This base class is needed by RAVEN to identify in the plugins folder which class must be considered as an "ExternalModel Plugin".
In addition, when loading an "ExternalModel Plugin", RAVEN expects to find, in the class representing the plugin, the following required methods:

```
from ravenframework.PluginBaseClasses.ExternalModelPluginBase import ExternalModelPluginBase
class NewPlugin(ExternalModelPluginBase):
  def run (self, container, Inputs)
```

In addition, the following optional methods can be specified:

```
from ravenframework.PluginBaseClasses.ExternalModelPluginBase import ExternalModelPluginBase
class NewPlugin(ExternalModelPluginBase):
  ...
  def createNewInput(self, container, inputs, samplerType, **Kwargs)
  def _readMoreXML(self, container, xmlNode)
  def initialize(self,container, runInfo, inputs)
```

In the following sub-sections all the methods are fully explained, providing examples.

3.2.1 Method: `run`

```
def run (self, container, Inputs)
```

As stated previously, the only method that *must* be present in an ExternalModel Plugin is the **run** function. In this function, the plugin developer needs to implement the algorithm that RAVEN will execute. The **`run`** method is generally called after having inquired the "createNewInput" method (either the internal RAVEN one or the one implemented by the plugin developer). The only two attributes this method is going to receive are a Python list of inputs (the inputs coming from the `createNewInput` method) and a "self-like" object named "container". If the user wants RAVEN to collect the results of this method, the outcomes of interest need to be stored in the above mentioned "container" object. **Note:** RAVEN is trying to collect the values of the variables listed only in the **<ExternalModel>** XML block. In the following an example is reported:

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

3.2.2 Method: `createNewInput`

```
def createNewInput(self, container, inputs, samplerType, **Kwargs)
```

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

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

Note: If the "Step" that is using this Model has as input(s) an object of main class type "DataObjects" (see Section **??**), the internal "createNewInput" method is going to convert it in a dictionary of values. Here we present an example:

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

3.2.3 Method: _readMoreXML

```
def _readMoreXML(self, container, xmlNode)
```

As already mentioned, the **_readMoreXML** method can be implemented by the ExternalModel Plugin developer if the XML input that belongs to this ExternalModel plugin needs to be extended to contain other information. The read information needs to be stored in the "self-like" object "container" in order to be available to all the other methods (e.g. if the developer needs to add a couple of newer XML nodes with information needed by the algorithm implemented in the "run" method). If this method is implemented in the **ExternalModel**, RAVEN is going to call it when the node **<ExternalModel>** is found parsing the XML input file. The method receives from RAVEN an attribute of type "xml.etree.ElementTree", containing all the sub-nodes and attribute of the XML block **<ExternalModel>**.

Example XML:

```
<Models>
  ...
  <ExternalModel name='AnExtModule' subType='NewPlugin'>
    <inputs>sigma,rho</inputs>
    <outputs>outcome</outputs>
    <!--
      here we define other XML nodes RAVEN does not read
         automatically.
      We need to implement, in the external model Plugin class
         the _readMoreXML
      method
    -->
    <newNodeWeNeedToRead>
      whatNeedsToBeRead
    </newNodeWeNeedToRead>
  </ExternalModel>
  ...
</Models>
```

Corresponding Python function:

```
def _readMoreXML(self, container, xmlNode):
  # the xmlNode is passed in by RAVEN framework
  # <newNodeWeNeedToRead> is unknown (in the RAVEN framework)
  # we have to read it on our own
  # get the node
  ourNode = xmlNode.find('newNodeWeNeedToRead')
```

```
  # get the information in the node
  container.ourNewVariable = ourNode.text
  # end function
```

3.2.4 Method: **initialize**

```
def initialize(self, container, runInfo, inputs)
```

The **initialize** method can be implemented in the **ExternalModel** Plugin in order to initialize some variables needed by it. This function accepts similar inputs to **initialize** method in section 2.3.4, except the `container` input variable. As all the others method in the ExternalModel Plugin, the information *must* be stored in the "self-like" object "container". If this method is implemented in the **ExternalModel** Plugin, RAVEN is going to call it at the initialization stage of each "Step" (see RAVEN User Manual Steps section).

In the following an example is reported:

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

4 OutStreams Plot Plugins

New plots that are specific to particular applications are possible through **<OutStreams>** **<Plot>** plugins.

These plotting plugins should inherit from the `PlotPlugin` base class defined in

```
raven/framework/PluginBaseClasses/OutStreamPlotPlugin.py
```

which sets up the plotting tool to be found when RAVEN runs. When loading an "OutStreams Plot Plugin", RAVEN expects to find, in the class representing the plugin, the following required methods:

```
from ravenframework.PluginBaseClasses.OutStreamPlotPlugin import PlotPlugin
class NewPlugin(PlotPlugin):
  ...
  def run(self):
    """
      Generate the plot
      @ In, None
      @ Out, None
    """
```

In addition, the following optional methods can be specified:

```
from ravenframework.PluginBaseClasses.OutStreamPlotPlugin import PlotPlugin
class NewPlugin(PlotPlugin):
  ...
  def __init__(self):
    """
      Constructor.
      @ In, None
      @ Out, None
    """
  ...
  @classmethod
  def getInputSpecification(cls):
    """
      Define the acceptable user inputs for this class.
      @ In, None
      @ Out, specs, InputData.ParameterInput,
    """
  ...
  def handleInput(self, spec):
    """
      Reads in data from the input file
      @ In, spec, InputData.ParameterInput, input information
      @ Out, None
    """
  ...
  def initialize(self, stepEntities):
    """
      Set up plotter for each run
      @ In, stepEntities, dict, entities from the Step
      @ Out, None
    """
```

A good example of the `PlotPlugin` can be found in the RAVEN ExamplePlugin, found at

```
raven/plugins/ExamplePlugin/src/CorrelationPlot.py
```

In the following sub-sections, these methods are explained in detail.

4.1 Optional Methods

The explanation for `__init__`, `getInputSpecification` and `handleInput` methods can be found in 2.3.2, 2.3.1, 2.3.3, respectively. All of these methods require a call to `super` to function as expected.

4.2 `run` method, required

The `run` method is the primary execution method for the PlotPlugin. Here, whatever data handling and plotting mechanics will be executed. Note that `run` does not receive any inputs; often, the source for the data to be plotted will be identified in the `initialize` method.

The `run` method can perform many actions including data manipulation, creation of `matplotlib` figures and axes, saving figures to file, and so forth. In the end, `run` should not return anything.

4.3 initialization, optional

The aptly-named `initialize` method is used at the start of every RAVEN **<Step>** to prepare for execution. This method has been simplified and it only accepts one input variable `stepEntities`. A common task for this method is to find the source of the data to plot. To make this process easier, RAVEN provides a `self.findSource` method that can search the input dictionary provided to `initialize` and find a **<DataObject>** by string name. In the following an example is reported:

```
def initialize(self, stepEntities):
  """
    Set up plotter for each run
    @ In, stepEntities, dict, entities from the Step
    @ Out, None
  """
  src = self.findSource('DataObjectName', stepEntities)
```

5 Code Plugins

The procedure of creating a new code/application plugin with RAVEN is a straightforward process. The plugin is performed through a Python interface that interprets the information coming from RAVEN and translates them into the input of the driven code plugin. This procedure does not require modifying RAVEN itself. Instead, the developer creates a new Python Code Plugin entity that is going to be embedded in RAVEN at run-time (no need to introduce hard-coded coupling statements).

At the initialization stage, RAVEN imports all the code plugin entity that are contained in `plugin/src` directory and performs some preliminary cross-checks.
It is important to notice that the name of class in the plugin entity module is the one the user needs to specify when the new code plugin needs to be used. For example, if the new plugin module contains the class "NewCode", the *subType* in the **<Code>** block will be **PluginName.NewCode**:

```
from ravenframework.PluginBaseClasses.CodePluginBase import CodePluginBase
class NewCode(CodePluginBase):
  ...
```

```
    ...
    <Code name='whatever' subType='PluginName.NewCode'>
     ...
    </Code>
    ...
  </Models>
```

When loading an "Code Plugin", RAVEN expects to find, in the class representing the plugin, the following required methods:

```
from ravenframework.PluginBaseClasses.CodePluginBase import CodePluginBase
class NewCode(CodePluginBase):
  ...
  def generateCommand(self, inputFiles, executable, clargs=None, fargs=None, preExec=None):
    """
      See base class. Collects all the clargs and the executable to produce the command-line call
          .
      Returns tuple of commands and base file name for run.
      Commands are a list of tuples, indicating parallel/serial and the execution command to use.
      @ In, inputFiles, list, List of input files (lenght of the list depends on the number of
          inputs have been added in the Step is running this code)
      @ In, executable, string, executable name with absolute path (e.g. /home/path_to_executable
          /code.exe)
      @ In, clargs, dict, optional, dictionary containing the command-line flags the user can
          specify in the input (e.g. under the node < Code >< clargstype =0 input0arg =0
          i0extension =0 .inp0/ >< /Code >)
      @ In, fargs, dict, optional, a dictionary containing the axuiliary input file variables the
           user can specify in the input (e.g. under the node < Code >< clargstype =0 input0arg
          =0 aux0extension =0 .aux0/ >< /Code >)
      @ In, preExec, string, optional, a string the command that needs to be pre-executed before
          the actual command here defined
      @ Out, returnCommand, tuple, tuple containing the generated command. returnCommand[0] is
          the command to run the code (string), returnCommand[1] is the name of the output root
```

```
    """
  def createNewInput(self, currentInputFiles, oriInputFiles, samplerType, **Kwargs):
    """
      Generate a new Projectile input file (txt format) from the original, changing parameters
      as specified in Kwargs['SampledVars']. In addition, it creaes an additional input file
          including the vector data to be
      passed to Dymola.
      @ In, currentInputFiles, list, list of current input files (input files from last this
          method call)
      @ In, oriInputFiles, list, list of the original input files
      @ In, samplerType, string, Sampler type (e.g. MonteCarlo, Adaptive, etc. see manual
          Samplers section)
      @ In, Kwargs, dictionary, kwarded dictionary of parameters. In this dictionary there is
          another dictionary called "SampledVars"
           where RAVEN stores the variables that got sampled (e.g. Kwargs['SampledVars'] => {'var1
               ':10,'var2':40})
      @ Out, newInputFiles, list, list of newer input files, list of the new input files (
          modified and not)
    """
```

In addition, the following optional methods can be specified:

```
from ravenframework.PluginBaseClasses.CodePluginBase import CodePluginBase
class NewCode(CodePluginBase):
  ...
  def __init__(self):
    """
      Constructor
      @ In, None
      @ Out, None
    """
  def initialize(self, runInfo, oriInputFiles):
    """
      Method to initialize the run of a new step
      @ In, runInfo, dict, dictionary of the info in the <RunInfo> XML block
      @ In, oriInputFiles, list, list of the original input files
      @ Out, None
    """
  def _readMoreXML(self, xmlNode):
    """
      Function to read the portion of the xml input that belongs to this specialized class and
      initialize some members based on inputs. This can be overloaded in specialized code
          interface in order
      to read specific flags
      @ In, xmlNode, xml.etree.ElementTree.Element, Xml element node
      @ Out, None
    """
  def getInputExtension(self):
    """
      Return a tuple of possible file extensions for a simulation initialization file (e.g.,
          input.i).
      @ In, None
      @ Out, validExtensions, tuple, tuple of valid extensions
    """
  def finalizeCodeOutput(self, command, output, workingDir):
    """
      Called by RAVEN to modify output files (if needed) so that they are in a proper form.
      In this case, the default .mat output needs to be converted to .csv output, which is the
      format that RAVEN can communicate with.
      @ In, command, string, the command used to run the just ended job
      @ In, output, string, the Output name root
      @ In, workingDir, string, current working dir
      @ Out, output, string, optional, present in case the root of the output file gets changed
          in this method.
```

```
    """
  def checkForOutputFailure(self, output, workingDir):
    """
      This method is called by RAVEN at the end of each run if the return code is == 0.
      This method needs to be implemented for the codes that, if the run fails, return a return
          code that is 0
      This can happen in those codes that record the failure of the job (e.g. not converged, etc
          .) as normal termination (returncode == 0)
      This method can be used, for example, to parse the output file looking for a special
          keyword that testifies that a particular job got failed
      (e.g. in RELAP5 would be the keyword "********")
      @ In, output, string, the Output name root
      @ In, workingDir, string, current working dir
      @ Out, failure, bool, True if the job is failed, False otherwise
    """
```

The explanation for all above required and optional methods except the `_readMoreXML` can be found in RAVEN user manual section **Advanced Users: How to couple a new code**. We recommand the plugin developers take a look at the RAVEN user manual for the detailed information and learn how to use these methods. In the following subsection, the `_readMoreXML` method is explained.

5.0.1 Method: _readMoreXML

The **_readMoreXML** method can be implemented by the Code plugin developer if the XML input that belongs to this Code plugin needs to be extended to contain other information. If this method is implemented in the **NewCodePlugin**, RAVEN is going to call it when the node **<Code>** is found parsing the XML input file. The method receives from RAVEN an attribute of type "xml.etree.ElementTree", containing all the sub-nodes and attribute of the XML block **<Code>**.

Example XML:

```
<Models>
  ...
  <Code name='newCode'
     subType='NewPluginName.NewCodePluginEntity'>
    <!--
      here we define other XML nodes RAVEN does not read
         automatically.
      We need to implement, in the code Plugin class the
         _readMoreXML
      method
    -->
    <newNodeWeNeedToRead>
      whatNeedsToBeRead
    </newNodeWeNeedToRead>
  </ExternalModel>
```

```
  ...
</Models>
```

6 PostProcessor Plugins

Similar to other entities plugins, the PostProcessor plugin accepts RAVEN DataObject(s) as input(s), perform additional operations on the data stored in the DataObject(s), and save the calculations results into a new RAVEN DataObject. This procedure does not require modifying RAVEN itself, and the developed plugin entity is going to be embedded in RAVEN at run-time.

At the initialization stage, RAVEN imports all the `PostProcessor` plugin entity that are contained in `plugin/src` directory and performs some preliminary cross-checks.
It is important to notice that the name of class in the plugin entity module is the one the user needs to specify when the new postprocessor plugin entity needs to be used. For example, if the new plugin module contains the class "NewPostProcessor", the *subType* in the **<PostProcessor>** block will be **PluginName.NewPostProcessor**:

```
from ravenframework.PluginBaseClasses.PostProcessorPluginBase import PostProcessorPluginBase
class NewPostProcessor(PostProcessorPluginBase):
  ...
```

```
    ...
    <PostProcessor name='whatever'
       subType='PluginName.NewPostProcessor'>
     ...
    </PostProcessor>
    ...
  </Models>
```

When loading an "PostProcessor Plugin entities", RAVEN expects to find, in the class representing the plugin, the following required methods:

```
from ravenframework.PluginBaseClasses.CodePluginBase import CodePluginBase
class NewCode(CodePluginBase):
  ...
  def run(self,inputIn):
    """
     This method to perform operations on the data from input DataObject(s)
     @ In, inputIn, dict, dictionary which contains the data inside the input DataObject
     @ Out, outputDict, dict, the output dictionary, passing through HistorySet info
    """
```

Where the input `inputIn` for **run** method has the following format:

```
inputIn = {'Data':listData, 'Files':listOfFiles},
listData has the following format if 'xrDataset' is passed to self.setInputDataType('xrDataset')
(listOfInputVars, listOfOutVars, xr.Dataset)
Otherwise listData has the following format: (listOfInputVars, listOfOutVars, DataDict) with
DataDict is a dictionary that has the format:
   dataDict['dims'] = dict {varName:independentDimensions}
   dataDict['metadata'] = dict {metaVarName:metaVarValue}
   dataDict['type'] = str TypeOfDataObject
   dataDict['inpVars'] = list of input variables
```

```
  dataDict['outVars'] = list of output variables
  dataDict['numberRealization'] = int SizeOfDataObject
  dataDict['name'] = str DataObjectName
  dataDict['metaKeys'] = list of meta variables
  dataDict['data'] = dict {varName: varValue(1-D or 2-D numpy array)}
```

In addition, the following optional methods can be specified:

```
from ravenframework.PluginBaseClasses.CodePluginBase import CodePluginBase
class NewCode(CodePluginBase):
  ...
  @classmethod
  def getInputSpecification(cls):
    """
      Method to get a reference to a class that specifies the input data for
      class cls.
      @ In, cls, the class for which we are retrieving the specification
      @ Out, inputSpecification, InputData.ParameterInput, class to use for
        specifying input of cls.
    """
  def __init__(self):
    """
      Constructor
      @ In, None
      @ Out, None
    """
  def initialize(self, runInfo, inputs, initDict=None):
    """
      Method to initialize the DataClassifier post-processor.
      @ In, runInfo, dict, dictionary of run info (e.g. working dir, etc)
      @ In, inputs, list, list of inputs
      @ In, initDict, dict, optional, dictionary with initialization options
      @ Out, None
    """
  def _handleInput(self, paramInput):
    """
      Function to handle the parameter input.
      @ In, paramInput, ParameterInput, the already parsed input.
      @ Out, None
    """
```

The explanation for all above optional methods can be found in section 2.3.

Appendices

A Document Version Information

tag_number_23-2004-g63d74f3a6
a7f92ad87c217540ff980c7ccb7e5b2a5c7d0f58 Congjian Wang
Wed, 9 Nov 2022 22:00:37 -0700

References

[1] Rabiti, Alfonsi, Mandelli, Cogliati, and Kinoshita, "Raven, a new software for dynamic risk analysis," in *PSAM 12 Probabilistic Safety Assessment and Management*, (Honolulu, Hawaii), June 2014.

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

v1.28

INL
Idaho National Laboratory