8 RunInfo

In the **RunInfo** block, the user specifies how the overall computation should be run. This block 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 (e.g. queueing system, if not PBS, etc.). In the following subsections, we explain all the keywords and how to use them in detail.

8.1 RunInfo: Input of Calculation Flow

This sub-section contains the information regarding the XML nodes used to define the settings of the calculation flow that is being performed through RAVEN:

- **<WorkingDir>**, ***string, required field***, specifies the absolute or relative (with respect to the location where the xml file is located) path to a directory that will store all the results of the calculations and where RAVEN looks for the files specified in the block **<Files>**. If `runRelative='True'` is used as an attribute, then it will be relative to where raven is run.
  Default: None

- **<RemoteRunCommand>**, ***string, optional field***, specifies the absolute or relative (with respect to the framework directory) path to a command that can be used on a remote machine to execute a command. The command is passed in as the environmental variable COMMAND.

  Default: raven_qsub_command.sh

- **<NodeParameter>**, ***string, optional field***, specifies the flag used to specify a node file for the MPIExec command. This will be followed by a file with the nodes that a single batch will run on.
  Default: -f

- **<MPIExec>**, ***string, optional field***, specifies the command used to run mpi. This will be followed by the **<NodeParameter>** and then the node file and then the code command.
  Default: mpiexec

- **<batchSize>**, ***integer, optional field***, specifies the number of parallel runs executed simultaneously (e.g., the number of driven code instances, e.g. RELAP5-3D, that RAVEN will spawn at the same time). Each parallel run will use `NumThreads * NumMPI` cores.
  Default: 1

- **<Sequence>**, ***comma separated string, required field***, is an ordered list of the step names that RAVEN will run (see Section 20).

- **<JobName>**, ***string, optional field***, specifies the name to use for the job when submitting to a pbs queue. Acceptable characters include alphanumerics as well as "-" and "_". If more than 15 characters are provided, RAVEN will truncate it using a hyphen between the first 10 and last 4 character, i.e., "1234567890abcdefgh" will be truncated to "1234567890-efgh".
  Default: raven_qsub

- **<printInput>**, ***string, optional field***, if provided, indicates RAVEN should print out a duplicate of the input file. If the provided text is **'false'**, or the node is not provided, then no duplicate will be printed. If the node is provided but no name specified, it will use the default name. Otherwise, the file will be written in the working directory as `name_provided.xml`.
  Default: duplicated_input.xml

- **<NumThreads>**, ***integer, optional field***, can be used to specify the number of threads RAVEN should associate when running the driven software. For example, if RAVEN is driving a code named "FOO," and this code has multi-threading support, this block is used to specify how many threads each instance of FOO should use (e.g. "`FOO --n-threads=N`" where `N` is the number of threads).
  Default: 1 (or None when the driven code does not have multi-threading support)

- **<NumMPI>**, ***integer, optional field***, can be used to specify the number of MPI CPUs RAVEN should associate when running the driven software. For example, if RAVEN is driving a code named "FOO," and this code has MPI support, this block specifies how many MPI CPUs each instance of FOO should use (e.g. "`mpiexec FOO -np N`" where `N` is the number of CPUs).
  Default: 1 (or None when the driven code does not have MPI support)

- **<totalNumCoresUsed>**, ***integer, optional field***, is the global number of CPUs RAVEN is going to use for performing the calculation. When the driven code has MPI and/or multi-threading support and the user specifies `NumThreads` > 1 and `NumMPI` > 1, then `totalNumCoresUsed` is set according to the following formula:
  `totalNumCoresUsed` = `NumThreads` $*$ `NumMPI` $*$ `batchSize`.
  Default: 1

- **<internalParallel>**, ***boolean, optional field***, is a boolean flag that controls the type of parallel implementation needs to be used for Internal Objects (e.g. ROMs, External Models, PostProcessors, etc.). If this flag is set to:

  - **False**, the internal parallelism is employed using multi-threading (i.e. 1 processor, multiple threads equal to the **<batchSize>**).
    Note: This "parallelism mode" runs multiple instances of the Model in a single processor. If the evaluation of the model is memory intensive (i.e. it uses a lot of memory) or computational intensive (i.e. a lot of computation operations evolving in a $CPUt \approx> 0.1\frac{sec}{evaluation}$) the single processor might get over-loaded determining a

degradation of performance. In such cases, the internal parallelism needs to be used (see the following);

- **True**, the internal parallelism is employed using a internally-developed multi-processor approach (i.e. **<batchSize>** processors, 1 single thread). This approach works for both Shared Memory Systems (e.g. PC, laptops, workstations, etc.) and Distributed Memory Machines (e.g. High Performance Computing Systems, etc.).
  Note: This "parallelism mode" runs multiple instances of the Model in multiple processors. Since the parallelism is employed in Python, some overhead is present. This "mode" needs to be used when:
  * the Model evaluation is memory intensive (i.e. the multi-threading approach will cause the over-load of a single processor);
  * the Model evaluation is computation intensive (i.e. $CPUt \approx> 0.1 \frac{sec}{evaluation}$).

Default: False

- **<precommand>**, ***string, optional field***, specifies a command that needs to be inserted before the actual command that is used to run the external model (e.g., `mpiexec -n 8 precommand ./externalModel.exe (...)`). Note that the precommand as well as the postcommand are ONLY applied to execution commands flagged as "parallel" within the code interface.
  Default: None

- **<postcommand>**, ***string, optional field***, specifies a command that needs to be appended after the actual command that is used to run the external model (e.g., `mpiexec -n 8 ./externalModel.exe (...) postcommand`). Note that the postcommand as well as the precommand are ONLY applied to execution commands flagged as "parallel" within the code interface.
  Default: None

- **<clusterParameters>**, ***string, optional field***, specifies extra parameters to be used with the cluster submission command. For example, if qsub is used to submit a command, then these parameters will be used as extra parameters with the qsub command. This can be repeated multiple times as needed and they will all be passed to the cluster submission command.
  Default: None

- **<MaxLogFileSize>**, ***integer, optional field***. specifies the maximum size of the log file in bytes. Every time RAVEN drives a code/software, it creates a logfile of the code's screen output.
  Default: ∞
  *(**Note:** This flag is not implemtend yet.)*

- **<deleteOutExtension>**, ***comma separated string, optional field***, specifies, if a run of an external model has not failed, which output files should be deleted by their extension (e.g., **<deleteOutExtension>**`txt,pdf`**</deleteOutExtension>** will delete all generated txt and pdf files).
  Default: None

- **<delSucLogFiles>**, ***boolean, optional field***, when True and the run of an external model has not failed (return code = 0), deletes the associated log files.
  Default: False

8.2 RunInfo: Input of Queue Modes

In this sub-section, all of the keywords (XML nodes) for setting the queue system are reported.

- **<mode>**, ***string, optional field***, can specify which kind of protocol the parallel enviroment should use. RAVEN currently supports one pre-defined "mode":

  – **mpi**: this "mode" uses `mpiexec` to distribute the running program; more information regarding this protocol can be found in [1]. Mode "MPI" can either generate a `qsub` command or can execute on selected nodes. In order to make the "mpi" mode generate a `qsub` command, an additional keyword (xml sub-node) needs to be specified:

    * If RAVEN is executed in the HEAD node of an HPC system using [2], the user needs to input a sub-node, **<runQSUB>**, right after the specification of the mpi mode (i.e. **<mode>**`mpi`**<runQSUB/></mode>**). If the keyword is provided, RAVEN generates a `qsub` command, instantiates itself, and submits itself to the queue system.
    * If the user decides to execute RAVEN from an "interactive node" (a certain number of nodes that have been reserved in interactive PBS mode), RAVEN, using the "mpi" system, is going to utilize the reserved resources (CPUs and nodes) to distribute the jobs, but, will not generate a `qsub` command.

    When the user decides to run in "mpi" mode without making RAVEN generate a `qsub` command, different options are available:

    * If the user decides to run on the local machine (either in local desktop/workstation or a remote machine), no additional keywords are needed (i.e. **<mode>**`mpi`**</mode>**).
    * If the user is running on multiple nodes, the node ids have to be specified:
      · the node ids can be specified in an external text file (node ids separated by blank space). This file needs to be provided in the XML node **<mode>**, introducing a sub-node named **<nodefile>** (e.g. **<mode>**`mpi`**<nodefile>**`/tmp/nodes`**</nodefile></mode>**).

- the node ids can be contained in an enviromental variable (node ids separated by blank space). This variable needs to be provided in the **<mode>** XML node, introducing a sub-node named **<nodefileenv>** (e.g. **<mode>**mpi**<nodefileenv>**NODEFILE**</nodefileenv></mode>**).
- If none of the above options are used, RAVEN will attempt to find the nodes' information in the enviroment variable `PBS_NODEFILE`.

* The mpi exec can be forced to run on one shared memory node with the **<NoSplitNode>**. If this is present, the splitting apart of the batches will put each batch on one shared memory node. Without **<NoSplitNode>**, they can be split across nodes. There is an option **maxOnNode** which puts at most **maxOnNode** number of mpi processes on one node. **<NoSplitNode>** can cause processes to not be placed, so **<NoSplitNode>** should not be used unless needed. If limiting the number of mpi processes on one node is desired without forcing them to only run on one node, **<LimitNode>** can be used. Both **<NoSplitNode>** and **<LimitNode>** can have a **noOverlap** which prevents multiple batches from running on a single node.

In addition, this flag activates the remote (PBS) execution of internal Models (e.g. ROMs, ExternalModels, PostProcessors, etc.). If this node is not present, the internal Models are run using a multi-threading approach (i.e. master processor, multiple parallel threads)

- **<CustomMode>**, ***xml node, optional field***, is an xml node where "advanced" users can implement newer "modes." Please refer to sub-section 8.4 for advanced users.

- **<queueingSoftware>**, ***string, optional field***. RAVEN has support for the PBS queueing system. If the platform provides a different queueing system, the user can specify its name here (e.g., PBS PROFESSIONAL, etc.).
  Default: PBS PROFESSIONAL

- **<expectedTime>**, ***colum separated string, optional field (mpi or custom mode)***, specifies the time the whole calculation is expected to last. The syntax of this node is *hours:minutes:seconds* (e.g. `40:10:30` equals 40 hours, 10 minutes, 30 seconds). After this period of time, the HPC system will automatically stop the simulation (even if the simulation is not completed). It is preferable to rationally overestimate the needed time.
  Default: 10:00:00 (10 hours.)

8.3 RunInfo: Example Cluster Usage

For this example, we have a PBSPro cluster, and there are thousands of node, and each node has 4 processors that share memory. There are a couple different ways this can be used. One way is to use interactive mode and have a RunInfo block:

```
<RunInfo>
    <WorkingDir>.</WorkingDir>
    <Sequence>FirstMRun</Sequence>
    <batchSize>3</batchSize>
    <NumThreads>4</NumThreads>
    <mode>mpi</mode>
    <NumMPI>2</NumMPI>
</RunInfo>
```

Then the commands can be used:

```
#Note: select=NumMPI*batchSize, ncpus=NumThreads
qsub -l select=6:ncpus=4:mpiprocs=1 -l walltime=10:00:00 -I
#wait for processes to be allocated and interactive shell to start

#Switch to the correct directory
cd $PBS_O_WORKDIR

#Load the module with the raven libraries
module load raven-devel-gcc

#Start Raven
python  ../../framework/Driver.py test_mpi.xml
```

Alternatively, RAVEN can be asked to submit the qsub directory. With this, the RunInfo is:

```
<RunInfo>
    <WorkingDir>.</WorkingDir>
    <Sequence>FirstMQRun</Sequence>
    <batchSize>3</batchSize>
    <NumThreads>4</NumThreads>
    <mode>
      mpi
      <runQSUB/>
    </mode>
    <NumMPI>2</NumMPI>
    <expectedTime>10:00:00</expectedTime>
</RunInfo>
```

In this case, the command run from the cluster submit node:

```
python ../../framework/Driver.py test_mpiqsub_local.xml
```

8.4 RunInfo: Advanced Users

This sub-section addresses some customizations of the running environment that are possible in RAVEN. Firstly, all the keywords reported in the previous sections can be pre-defined by the user in an auxiliary XML input file. Every time RAVEN gets instantiated (i.e. the code is run), it looks for an optional file, named "`default_runinfo.XML`" contained in the "`\home\username\.raven\`" directory (i.e. "`\home\username\.raven\default_runinfo.XML`"). This file (same syntax as the RunInfo block defined in the general input file) will be used for defining default values for the data in the RunInfo block. In addition to the keywords defined in the previous sections, in the **`<RunInfo>`** node, an additional keyword can be defined:

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

As already mentioned, this file is read to define default data for the RunInfo block. This means that all the keywords defined here will be overridden by any values specified in the actual RAVEN input file.
In section 8.2, it is explained how RAVEN can handle the queue and parallel systems. If the currently available "modes" are not suitable for the user's system (workstation, HPC system, etc.), it is possible to define a custom "mode" modifying the **`<RunInfo>`** block as follows:

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

The file field can use %BASE_WORKING_DIR% and %FRAMEWORK_DIR% to specify the location of the file with respect to the base working directory or the framework directory.

The python file should define a class that inherits from `Simulation.SimulationMode` of the RAVEN framework and overrides the necessary functions. Generally, `modifySimulation` will be overridden to change the precommand or postcommand parts which will be added before and after the executable command. An example Python class is given below with the functions that can and should be overridden:

```
import Simulation
class NewMode(Simulation.SimulationMode):
```

```
def doOverrideRun(self):
  # If doOverrideRun is true, then use runOverride instead of
  # running the simulation normally.
  return True

def runOverride(self):
  # this can completely override the Simulation's run method
  # If implemented this method should call simulation.run somehow,
  # possibly very indirectly
  pass

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

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

RAVEN's Job Handler module controls the creation and execution of individual code runs. Essentially, the SimulationMode class may be used when it is necessary to customize that behavior. First, it allows overriding how Simulation runs. This first method can be used if for example RAVEN needs to be run on a different machine such as a head node of a computer cluster. In such a case, a runOverride function can be created that causes RAVEN to be instantiated on the cluster head node (in cases where that is different than the computer where the user is currently working). Secondly, (and usually easier when this is sufficient) the SimulationMode class allows modifying the various run info parameters before the code is run.

For modification of the run info parameters, generally the two most important are precommand and postcommand. They are placed in front and back before running the code. So for example if precommand is 'mpiexec -n 3' and postcommand is '–number-threads=4' and the code command is 'runIt' then the full command would be: 'mpiexec -n 3 runIt –number-threads=4' The precommand and postcommand are used for any run type that is 'parallel', but not for 'serial' codes. They can be modified by overriding the `modifySimulation` method and assigning to the `runInfoDict` in the simulation passed in when the class is created.

To help with these commands, there are several variables that are substituted in before running the command. These are:

%INDEX% Contains the zero-based index in list of running jobs. Note that this is stable for the life of the job. After the job finishes, this is reused. An example use would be if there were four cpus and the batch size was four, the %INDEX% could be used to determine which cpu to run on.

%INDEX1% Contains the one-based index in the list of running jobs, same as %INDEX%+1

%CURRENT_ID% zero-based id for the job handler. This starts as 0, and increases for each job the job handler starts.

%CURRENT_ID1% one-based id for the job handler, same as %CURRENT_ID%+1

%SCRIPT_DIR% Expands to the full path of the script directory (raven/scripts)

%FRAMEWORK_DIR% Expands to the full path of the framework directory (raven/framework)

%WORKING_DIR% Expands to the working directory where the input is

%BASE_WORKING_DIR% Expands to the base working directory given in RunInfo. This will likely be a parent of WORKING_DIR

%METHOD% Expands to the environmental variable $METHOD

%NUM_CPUS% Expands to the number of cpus to use per single batch. This is NumThreads in the XML file.

The final joining of the commands and substituting the variables is done in the JobHandler class.

8.5 RunInfo: Examples

Here we present a few examples using different components of the RunInfo node:

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

<Files>
```

```
    <Input name='lorentzAttractor.py'
       type=''>lorentzAttractor.py</Input>
</Files>
```

This examples specifies the working directory (`WorkingDir`) where the necessary file (`Files`) is located and to run a series of 100 (`batchSize`) Monte-Carlo calculations (`Sequence`). MPI mode (`mode`) is used along with 4 threads (`NumThreads`) and 2 MPI processes per run (`NumMPI`).

9 Files

The **<Files>** block defines any files that might be needed within the RAVEN run. This could include inputs to the Model, pickled ROM files, or CSV files for postprocessors, 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 the following attributes:

- **name**, ***required string attribute***, user-defined name of the file. This does not need to be the actual filename; this is the name by which RAVEN will identify the file. **Note:** As with other objects, this name can be used to refer to this specific entity from other input blocks in the XML.
- **type**, ***optional string attribute***, a type label for this file. While RAVEN does not directly make use of file types, they are available in the CodeInterface as identifiers. If not provided, the type will be stored as python `None` type.
- **perturbable**, ***optional boolean attribute***, flag to indicate whether a file can be perturbed or not. RAVEN does not directly use this attribute, but it is availabe in the CodeInterface. If not provided, defaults to True.

For example, if the files `templateInput.i, materials.i, history.i, mesh.e` are required to run a Model, the **<Files>** block might appear as:

```
  ...
  <Files>
    <Input name='main' type='maininput'>templateInput.i</Input>
    <Input name='mat'  type='mtlinput' >materials.i</Input>
    <Input name='hist' type='histinput'>history.i</Input>
    <Input name='mesh' type='mesh'
       perturbable='false'>mesh.e</Input>
  </Files>
   ...
</Simulation>
```

10 VariableGroups

The **<VariableGroups>** block is an optional input for the convenience of the user. It allows the possibility of creating a collection of variables instead of re-listing all the variables in places throughout the input file, such as DataObjects, ROMs, and ExternalModels. Each entry in the **<VariableGroups>** block has a distinct name and list of each constituent variable in the group. Alternatively, set operations can be used to construct variable groups from other variable groups. In this case, the dependent groups and the base group on which operations should be performed must be listed. The following types of set operations are included in RAVEN:

- +, Union, the combination of all variables in the **'base'** set and listed set,
- −, Complement, the relative complement of the listed set in the **'base'** set,
- ^, Intersection, the variables common to both the **'base'** and listed set,
- %, Symmetric Difference, the variables in only either the **'base'** or listed set, but not both.

Multiple operations can be performed by separating them with commas in the text of the group node. In the event the listed set is a single variable, it will be treated like a set with a single entry.

When using the variable groups in a node, they can be listed alone or as part of a comma-separated list. The variable group name will only be substituted in the text of nodes, not attributes or tags.

Each **<Group>** node has the following attributes:

- **name**, ***required string attribute***, user-defined name of the group. This is the identifier that will be used elsewhere in the RAVEN input.
- **dependencies**, ***optional comma-separated string attribute***, the other variable groups on which this group is dependent for construction. If listed, all entries in the text of this node must be proceeded by one of the set operators above. Defaults to an empty string.
- **base**, ***optional string attribute***, the starting set for constructing a dependent variable group. This attribute is required if any dependencies are listed. Set operations are performed by performing the chosen operation on the variable group listed in this attribute along with the listed variable group. No default.

An example of constructing and using variable groups is listed here. The variable groups **'x_odd'**, **'x_even'**, **'x_first'**, and **'y_group'** are constructed independently, and the remainder are examples of other operations.

```
...
<VariableGroups>
  <Group name="x_odd"  >x1,x3,x5</Group>
```

```
    <Group name="x_even" >x2,x4,x6</Group>
    <Group name="x_first">x1,x2,x3</Group>
    <Group name="y_group">y1,y2</Group>
    <Group name="add_remove" dependencies="x_first"
      base="x_first">-x1,+ x4,+x5</Group>
    <Group name="union"      dependencies="x_odd,x_even"
      base="x_odd">+x_even</Group>
    <Group name="complement" dependencies="x_odd,x_first"
      base="x_odd">-x_first</Group>
    <Group name="intersect"  dependencies="x_even,x_first"
      base="x_even">^x_first</Group>
    <Group name="sym_diff"   dependencies="x_odd,x_first"
      base="x_odd">% x_first</Group>
  </VariableGroups>
  ...
  <DataObjects>
    <PointSet name="dataset">
      <Input>union</Input>
      <Output>y_group</Output>
    </PointSet>
  </DataObjects>
  ...
</Simulation>
```

11 Distributions

RAVEN provides support for several probability distributions. Currently, the user can choose among several 1-dimensional distributions and N-dimensional ones, either custom or multidimensional normal.

The user will specify the probability distributions, that need to be used during the simulation, within the **<Distributions>** XML block:

```
<Simulation>
   ...
  <Distributions>
    <!-- All the necessary distributions will be listed here -->
  </Distributions>
  ...
</Simulation>
```

In the next two sub-sections, the input requirements for all of the distributions are reported.

11.1 1-Dimensional Probability Distributions

This sub-section is organized in two different parts: 1) continuous 1-D distributions and 2) discrete 1-D distributions. These two paragraphs cover all the requirements for using the different distribution entities.

11.1.1 1-Dimensional Continuous Distributions

In this paragraph all the 1-D distributions currently available in RAVEN are reported.

Firstly, all the probability distributions functions in the code can be truncated by using the following keywords:

```
<Distributions>
   ...
   <aDistributionType>
      ...
      <lowerBound>aFloatValue</lowerBound>
      <upperBound>aFloatValue</upperBound>
      ...
   </aDistributionType>
```