MANUAL

INL/EXT-15-34123
Revision 7
Printed May 19, 2021

RAVEN User Manual

Cristian Rabiti, Andrea Alfonsi, Joshua Cogliati, Diego Mandelli, Robert Kinoshita, Sonat Sen, Congjian Wang, Paul W. Talbot, Daniel P. Maljovec, 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.

Approved for unlimited release.

Idaho National Laboratory

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

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

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

INL/EXT-15-34123
Revision 7
Printed May 19, 2021

RAVEN User Manual

Project Manager:
Cristian Rabiti
Principal Investigator and Technical Leader:
Andrea Alfonsi
Main Developers:
Andrea Alfonsi
Diego Mandelli
Joshua Cogliati
Congjian Wang
Paul W. Talbot
Daniel P. Maljovec
Robert Kinoshita
Mohammad G. Abdo
Former Developers:
Sonat Sen
Jun Chen
Contributors:
Alessandro Bandini (Post-Processor)
Ivan Rinaldi (documentation)
Claudia Picoco (new external code interface)
James B. Tompkins (new external code interface)
Matteo Donorio (new external code interface)
Fabio Giannetti (new external code interface)
Jia Zhou (conjugate gradient optimizer)

Contents

1 Introduction . . . 19
2 Manual Formats . . . 20
3 Installation . . . 21
3.1 Overview . . . 21
3.2 Linux Ubuntu Installation . . . 21
3.2.0.1 Optional LateX installation . . . 21
3.3 Mac OSX Installation . . . 22
3.3.1 Installing XCode Command Line Tools . . . 22
3.3.2 Installing XQuartz . . . 22
3.4 Microsoft Windows . . . 22
3.4.1 A Visual Guide . . . 23
3.4.2 GIT SCM for Windows . . . 23
3.4.3 Install Python Language and Package Support . . . 23
3.4.4 Compiler Installation and Configuration . . . 24
3.5 Conda: Python Dependencies . . . 24
3.6 Installing RAVEN . . . 24
3.6.1 Obtaining RAVEN Source Code . . . 25
3.6.2 Installing Python Libraries . . . 25
3.6.3 Compiling RAVEN . . . 26
3.6.4 Testing RAVEN . . . 26
3.6.5 Updating RAVEN . . . 27
3.6.6 In-use Testing . . . 27
4 Running RAVEN . . . 28
5 Raven Input Structure . . . 29
5.1 Comments . . . 29
5.2 Verbosity . . . 30
5.3 External Input Files . . . 31
6 RunInfo . . . 33
6.1 RunInfo: Input of Calculation Flow . . . 33
6.2 RunInfo: Input of Queue Modes . . . 36
6.3 RunInfo: Example Cluster Usage . . . 38
6.4 RunInfo: Advanced Users . . . 39
6.5 RunInfo: Examples . . . 42
7 Files . . . 43
8 VariableGroups . . . 44
9 Distributions . . . 46
9.1 1-Dimensional Probability Distributions . . . 46
9.1.1 1-Dimensional Continuous Distributions . . . 46
9.1.1.1 Beta Distribution . . . 47
9.1.1.2 Exponential Distribution . . . 48

9.1.1.3 Gamma Distribution 49
9.1.1.4 Laplace Distribution 50
9.1.1.5 Logistic Distribution 50
9.1.1.6 LogNormal Distribution 51
9.1.1.7 LogUniform Distribution 52
9.1.1.8 Normal Distribution 53
9.1.1.9 Triangular Distribution 54
9.1.1.10 Uniform Distribution 55
9.1.1.11 Weibull Distribution 55
9.1.1.12 Custom1D Distribution 56
9.1.2 1-Dimensional Discrete Distributions. 58
9.1.2.1 Bernoulli Distribution 58
9.1.2.2 Binomial Distribution 59
9.1.2.3 Geometric Distribution 59
9.1.2.4 Poisson Distribution 60
9.1.2.5 Categorical Distribution 61
9.1.2.6 Uniform Discrete Distribution 62
9.1.2.7 Markov Categorical Distribution 63
9.2 N-Dimensional Probability Distributions 64
9.2.1 MultivariateNormal Distribution 65
9.2.2 NDInverseWeight Distribution 67
9.2.3 NDCartesianSpline Distribution 69
10 Samplers 71
10.1 Forward Samplers 74
10.1.1 Monte Carlo 75
10.1.2 Grid 79
10.1.3 Sparse Grid Collocation 85
10.1.4 Sobol 91
10.1.5 Stratified 95
10.1.6 Response Surface Design 102
10.1.7 Factorial Design 108
10.1.8 Ensemble Forward Sampling strategy 114
10.1.9 Custom Sampling strategy 117
10.2 Dynamic Event Tree (DET) Samplers 120
10.2.1 Dynamic Event Tree 120
10.2.2 Hybrid Dynamic Event Tree 124
10.3 Adaptive Samplers 132
10.3.1 Limit Surface Search 133
10.3.2 Adaptive Monte Carlo 138
10.3.3 Adaptive Dynamic Event Tree 143
10.3.4 Adaptive Hybrid Dynamic Event Tree 149
10.3.5 Adaptive Sparse Grid 157

10.3.6 Adaptive Sobol Decomposition . . . 164
10.4 Markov Chain Monte Carlo . . . 168
10.4.1 Metropolis (Metropolis-Hastings Sampler) . . . 169
10.4.2 Adaptive Metropolis Sampler . . . 174
11 Optimizers . . . 179
11.1 GradientDescent . . . 179
11.2 SimulatedAnnealing . . . 188
11.3 GeneticAlgorithm . . . 195
12 DataObjects . . . 205
13 Databases . . . 209
13.1 NetCDF . . . 209
13.2 HDF5 . . . 210
14 OutStream system . . . 212
14.1 Defaults . . . 212
14.2 Default Printing system . . . 212
14.2.1 **DataObjects** Printing . . . 213
14.2.2 **ROM** Printing . . . 214
14.3 Default Plotting system . . . 215
14.3.1 Plot input structure . . . 215
14.3.1.1 “Actions” input block . . . 216
14.3.1.2 “plotSettings” input block . . . 222
14.3.1.2.1 Specifying What Values to Plot . . . 226
14.3.1.3 Predefined Plotting System: 2D/3D . . . 226
14.3.2 2D & 3D Scatter plot . . . 228
14.3.3 2D & 3D Line plot . . . 229
14.3.4 2D & 3D Histogram plot . . . 229
14.3.5 2D & 3D Stem plot . . . 231
14.3.6 2D Step plot . . . 232
14.3.7 2D Pseudocolor plot . . . 232
14.3.8 2D Contour or filledContour plots . . . 233
14.3.9 3D Surface Plot . . . 234
14.3.10 3D Wireframe Plot . . . 235
14.3.11 3D Tri-surface Plot . . . 236
14.3.12 3D Contour or filledContour plots . . . 237
14.3.13 DataMining plots . . . 238
14.3.14 Example XML input . . . 239
14.4 Specific Plots . . . 240
14.4.1 SamplePlot . . . 240
15 Models . . . 242
15.1 Code . . . 243
15.2 Dummy . . . 246
15.3 ROM . . . 247

15.3.1 NDspline 249
15.3.2 pickledROM 250
15.3.3 GaussPolynomialRom 251
15.3.4 HDMRRom 254
15.3.5 MSR 255
15.3.6 NDinvDistWeight 258
15.3.7 SciKitLearn 259
15.3.7.1 Linear Models 260
15.3.7.1.1 Linear Model: Automatic Relevance Determination Regression 260
15.3.7.1.2 Linear Model: Bayesian ridge regression 261
15.3.7.1.3 Linear Model: Elastic Net 263
15.3.7.1.4 Linear Model: Elastic Net CV 264
15.3.7.1.5 Linear Model: Least Angle Regression model 265
15.3.7.1.6 Linear Model: Cross-validated Least Angle Regression model 266
15.3.7.1.7 Linear Model trained with L1 prior as regularizer (aka the Lasso) 267
15.3.7.1.8 Lasso linear model with iterative fitting along a regularization path 268
15.3.7.1.9 Lasso model fit with Least Angle Regression 269
15.3.7.1.10 Cross-validated Lasso, using the LARS algorithm 270
15.3.7.1.11 Lasso model fit with Lars using BIC or AIC for model selection 271
15.3.7.1.12 Ordinary least squares Linear Regression 272
15.3.7.1.13 Logistic Regression 273
15.3.7.1.14 Multi-task Lasso model trained with L1/L2 mixed-norm as regularizer 274
15.3.7.1.15 Multi-task Elastic Net model trained with L1/L2 mixed-norm as regularizer 275
15.3.7.1.16 Orthogonal Mathching Pursuit model (OMP) 276
15.3.7.1.17 Cross-validated Orthogonal Mathching Pursuit model (OMP) 277
15.3.7.1.18 Passive Aggressive Classifier 278
15.3.7.1.19 Passive Aggressive Regressor 279
15.3.7.1.20 Perceptron 280
15.3.7.1.21 Linear least squares with l2 regularization 282
15.3.7.1.22 Classifier using Ridge regression 283
15.3.7.1.23 Ridge classifier with built-in cross-validation 284
15.3.7.1.24 Ridge regression with built-in cross-validation 285
15.3.7.1.25 Linear classifiers (SVM, logistic regression, a.o.) with SGD training 286

15.3.7.1.26 Linear model fitted by minimizing a regularized empirical loss with SGD . . . 289
15.3.7.2 Support Vector Machines . . . 291
15.3.7.2.1 Linear Support Vector Classifier . . . 291
15.3.7.2.2 C-Support Vector Classification . . . 293
15.3.7.2.3 Nu-Support Vector Classification . . . 294
15.3.7.2.4 Support Vector Regression . . . 296
15.3.7.3 Multi Class . . . 297
15.3.7.3.1 One-vs-the-rest (OvR) multiclass/multilabel strategy . . . 297
15.3.7.3.2 One-vs-one multiclass strategy . . . 298
15.3.7.3.3 Error-Correcting Output-Code multiclass strategy . . . 298
15.3.7.4 Naive Bayes . . . 299
15.3.7.4.1 Gaussian Naive Bayes . . . 300
15.3.7.4.2 Multinomial Naive Bayes . . . 300
15.3.7.4.3 Bernoulli Naive Bayes . . . 301
15.3.7.5 Neighbors . . . 302
15.3.7.5.1 K Neighbors Classifier . . . 303
15.3.7.5.2 Radius Neighbors Classifier . . . 304
15.3.7.5.3 K Neighbors Regressor . . . 306
15.3.7.5.4 Radius Neighbors Regressor . . . 307
15.3.7.5.5 Nearest Centroid Classifier . . . 308
15.3.7.6 Tree . . . 309
15.3.7.6.1 Decision Tree Classifier . . . 310
15.3.7.6.2 Decision Tree Regressor . . . 311
15.3.7.6.3 Extra Tree Classifier . . . 313
15.3.7.6.4 Extra Tree Regressor . . . 314
15.3.7.7 Gaussian Process . . . 315
15.3.7.8 Neural Network Models . . . 317
15.3.7.8.1 MLPClassifier . . . 318
15.3.7.8.2 MLPRegressor . . . 321
15.3.8 SyntheticHistory . . . 324
15.3.9 ARMA . . . 327
15.3.10 PolyExponential . . . 334
15.3.11 DMD . . . 337
15.3.12 TensorFlow-Keras Deep Neural Networks . . . 340
15.3.12.1 Activation Functions . . . 345
15.3.12.2 Initializer Functions . . . 345
15.3.12.3 Regularizer Functions . . . 346
15.3.12.4 Constraint Functions . . . 347
15.3.12.5 KerasMLPClassifier . . . 347
15.3.12.6 KerasConvNetClassifier . . . 350
15.3.12.7 KerasLSTMClassifier . . . 362

15.4 External Model . . . 367
15.4.1 Generic External Model . . . 369
15.4.1.1 Method: `def _readMoreXML` . . . 369
15.4.1.2 `def initialize` . . . 370
15.4.1.3 Method: `def createNewInput` . . . 372
15.4.1.4 Method: `def run` . . . 373
15.4.2 pickledModel . . . 373
15.5 PostProcessor . . . 374
15.5.1 BasicStatistics . . . 375
15.5.2 ComparisonStatistics . . . 381
15.5.3 ImportanceRank . . . 384
15.5.4 SafestPoint . . . 387
15.5.5 LimitSurface . . . 389
15.5.6 LimitSurfaceIntegral . . . 391
15.5.7 External . . . 393
15.5.8 TopologicalDecomposition . . . 396
15.5.9 DataMining . . . 398
15.5.9.1 SciKitLearn . . . 399
15.5.9.2 Gaussian mixture models . . . 400
15.5.9.2.1 GMM classifier . . . 400
15.5.9.2.2 Variational GMM Classifier (VBGMM) . . . 401
15.5.9.3 Clustering . . . 402
15.5.9.3.1 K-Means Clustering . . . 404
15.5.9.3.2 Mini Batch K-Means . . . 405
15.5.9.3.3 Affinity Propagation . . . 407
15.5.9.3.4 Mean Shift . . . 408
15.5.9.3.5 Spectral clustering . . . 409
15.5.9.3.6 DBSCAN Clustering . . . 411
15.5.9.3.7 Agglomerative Clustering . . . 411
15.5.9.3.8 Clustering performance evaluation . . . 413
15.5.9.4 Decomposing signals in components (matrix factorization problems) . . . 413
15.5.9.4.1 Principal component analysis (PCA) . . . 413
15.5.9.4.2 Truncated singular value decomposition . . . 419
15.5.9.4.3 Fast ICA . . . 419
15.5.9.5 Manifold learning . . . 420
15.5.9.5.1 Isomap . . . 421
15.5.9.5.2 Locally Linear Embedding . . . 422
15.5.9.5.3 Spectral Embedding . . . 424
15.5.9.5.4 Multi-dimensional Scaling (MDS) . . . 425
15.5.9.6 Scipy . . . 426
15.5.10 Discrete Risk Measures . . . 428

15.5.11 HistorySetDelay . . . 433
15.5.12 Interfaced . . . 434
15.5.12.1 Data Format . . . 436
15.5.12.2 Method: HStoPSOperator . . . 437
15.5.12.3 Method: HistorySetSampling . . . 439
15.5.12.4 Method: HistorySetSync . . . 440
15.5.12.5 Method: HistorySetSnapShot . . . 441
15.5.12.6 Method: HS2PS . . . 443
15.5.12.7 Method: TypicalHistoryFromHistorySet . . . 443
15.5.12.8 Method: dataObjectLabelFilter . . . 445
15.5.13 ParetoFrontier . . . 445
15.5.14 MCSImporter . . . 447
15.5.15 ETImporter . . . 449
15.5.16 FTImporter . . . 454
15.5.17 Metric . . . 458
15.5.18 CrossValidation . . . 460
15.5.18.1 SciKitLearn . . . 462
15.5.18.2 K-fold . . . 462
15.5.18.3 Stratified k-fold . . . 462
15.5.18.4 Label k-fold . . . 463
15.5.18.5 Leave-One-Out - LOO . . . 463
15.5.18.6 Leave-P-Out - LPO . . . 463
15.5.18.7 Leave-One-Label-Out - LOLO . . . 464
15.5.18.8 Leave-P-Label-Out . . . 464
15.5.18.9 ShuffleSplit . . . 465
15.5.18.10 Label-Shuffle-Split . . . 465
15.5.19 DataClassifier . . . 466
15.5.20 ValueDuration . . . 469
15.5.21 FastFourierTransform . . . 470
15.5.22 SampleSelector . . . 471
15.5.23 EconomicRatio . . . 472
15.6 EnsembleModel . . . 474
15.7 HybridModel . . . 478
15.8 LogicalModel . . . 483
16 Functions . . . 488
17 Metrics . . . 490
17.1 Paired Distance Metric . . . 491
17.1.1 Euclidean . . . 491
17.1.2 Cosine . . . 491
17.1.3 Manhattan . . . 492
17.1.4 Braycurtis . . . 492
17.1.5 Canberra . . . 492

17.1.6 Correlation . . . 493
17.1.7 Minkowski . . . 493
17.2 Regression Metric . . . 495
17.2.1 Explained variance score . . . 495
17.2.2 Mean absolute error . . . 495
17.2.3 Mean squared error . . . 496
17.2.4 R2 score . . . 496
17.3 Boolean Metric . . . 497
17.3.1 Dice . . . 497
17.3.2 Hamming . . . 498
17.3.3 Jaccard . . . 498
17.3.4 Kulsinski . . . 499
17.3.5 Rogerstanimoto . . . 499
17.3.6 Russellrao . . . 500
17.3.7 Sokalmichener . . . 500
17.3.8 Sokalsneath . . . 501
17.3.9 Yule . . . 501
17.4 Dynamic Time Warping . . . 503
17.5 CDFAreaDifference . . . 504
17.6 PDFCommonArea . . . 504
17.7 Pairwise Metric . . . 505
17.7.1 Polynomial . . . 505
17.7.2 additive_chi2 . . . 506
17.7.3 chi2 . . . 506
17.7.4 cosine_similarity . . . 506
17.7.5 laplacian . . . 507
17.7.6 linear . . . 507
17.7.7 rbf . . . 508
17.7.8 sigmoid . . . 508
17.7.9 Distance Based Metric . . . 508
18 Steps . . . 510
18.1 SingleRun . . . 511
18.2 MultiRun . . . 513
18.3 IOStep . . . 516
18.4 RomTrainer . . . 521
18.5 PostProcess . . . 522
19 Existing Interfaces . . . 524
19.1 Generic Interface . . . 524
19.2 RAVEN Interface . . . 527
19.2.1 ExternalXML and RAVEN interface . . . 532
19.3 RELAP5 Interface . . . 533
19.3.1 Sequence . . . 533

19.3.2 batchSize and mode . . . 533
19.3.3 RunInfo . . . 533
19.3.4 Files . . . 533
19.3.5 Models . . . 535
19.3.6 Distributions . . . 536
19.3.7 Samplers . . . 537
19.3.8 Steps . . . 540
19.3.9 Databases . . . 541
19.3.10 Modified Version of the Institute of Nuclear Safety System Incorporated (Japan) . . . 542
19.4 RELAP7 Interface . . . 542
19.4.1 Files . . . 543
19.4.2 Models . . . 543
19.4.3 Distributions . . . 543
19.4.4 Samplers . . . 544
19.5 MooseBasedApp Interface . . . 545
19.5.1 Files . . . 545
19.5.2 Models . . . 545
19.5.3 Distributions . . . 546
19.5.4 Samplers . . . 547
19.5.5 Steps . . . 549
19.5.6 Databases . . . 551
19.5.7 DataObjects . . . 551
19.5.8 OutStreams . . . 552
19.6 MooseVPP Interface . . . 552
19.7 OpenModelica Interface . . . 554
19.7.1 Files . . . 554
19.7.2 Models . . . 555
19.7.3 CSV Output . . . 556
19.8 Dymola Interface . . . 556
19.8.1 Files . . . 557
19.8.2 Models . . . 558
19.9 Mesh Generation Coupled Interfaces . . . 560
19.9.1 MooseBasedApp and Cubit Interface . . . 560
19.9.1.1 Files . . . 561
19.9.1.2 Models . . . 562
19.9.1.3 Distributions . . . 563
19.9.1.4 Samplers . . . 563
19.9.1.5 Steps,OutStreams,DataObjects . . . 564
19.9.1.6 File Cleanup . . . 564
19.9.2 MooseBasedApp and Bison Mesh Script Interface . . . 564
19.9.2.1 Files . . . 564

19.9.2.2 Models . . . 565
19.9.2.3 Distributions . . . 566
19.9.2.4 Samplers . . . 566
19.9.2.5 Steps,OutStreams,DataObjects . . . 567
19.9.2.6 File Cleanup . . . 567
19.10 Rattlesnake Interfaces . . . 567
19.10.1 Files . . . 568
19.10.1.1 Perturb Yak Multigroup Cross Section Libraries . . . 568
19.10.1.2 Perturb Instant format Cross Section Libraries . . . 570
19.10.2 Models . . . 571
19.10.3 Distributions . . . 571
19.10.3.1 Samplers . . . 571
19.10.4 Steps . . . 572
19.11 MAAP5 Interface . . . 573
19.11.1 RAVEN Input file . . . 573
19.11.1.1 Files . . . 573
19.11.1.2 Models . . . 573
19.11.1.3 Other blocks . . . 574
19.11.2 MAAP5 Input files . . . 574
19.11.2.1 MAAP5 include file . . . 575
19.11.2.2 MAAP5 input file . . . 575
19.11.2.3 MAAP5 PLOTFIL blocks . . . 577
19.12 MAMMOTH Interface . . . 577
19.12.1 Files . . . 578
19.12.2 Models . . . 579
19.12.3 Distributions . . . 579
19.12.3.1 Samplers . . . 579
19.12.4 Steps . . . 581
19.13 MELCOR Interface . . . 582
19.13.1 Sequence . . . 582
19.13.2 batchSize and mode . . . 582
19.13.3 RunInfo . . . 582
19.13.4 Files . . . 583
19.13.5 Models . . . 584
19.13.6 Distributions . . . 584
19.13.7 Samplers . . . 585
19.13.8 Steps . . . 586
19.13.9 Databases . . . 587
19.13.10 DataObjects . . . 588
19.14 SCALE Interface . . . 589
19.14.1 Models . . . 589
19.14.2 Files . . . 591

19.14.2.1 Output Files conversion 591
19.14.3 Samplers or Optimizers 593
19.15 CTF Interface 595
19.15.1 Sequence 595
19.15.2 batchSize and mode 595
19.15.3 RunInfo 595
19.15.4 Models 596
19.15.5 Files 596
19.15.5.1 Output Files Conversion 598
19.15.6 Distributions 601
19.15.7 Samplers 602
19.15.8 Steps 603
19.16 SAPHIRE Interface 605
19.16.1 Files 605
19.16.2 Models 605
19.16.3 Distributions 607
19.16.4 Samplers 607
19.16.5 Steps 611
19.17 PHISICS Interface 611
19.17.1 General Information 611
19.17.2 Files 611
19.17.3 Models 613
19.17.4 Distributions 614
19.17.5 Samplers 614
19.17.5.1 Decay constant variable 615
19.17.5.2 Fission yield variable 615
19.17.5.3 Number density variable 615
19.17.5.4 Fission Q-values variables 615
19.17.5.5 α decay variable 616
19.17.5.6 β^+ decay variable 616
19.17.5.7 β^{+*} decay variable 616
19.17.5.8 β decay variable 616
19.17.5.9 β^* decay variable 616
19.17.5.10 Internal transition decay variable 616
19.17.5.11 Cross section scaling factors 616
19.17.6 Steps 618
19.17.7 Additional Input 618
19.17.8 Output Files Conversion 620
19.18 PHISICS/RELAP5 Interface 621
19.18.1 General Information 621
19.18.2 Files 622
19.18.3 Models 622

19.18.4 Distributions . . . 623
19.18.5 Samplers . . . 623
19.18.6 Steps . . . 624
19.18.7 Additional Input . . . 624
19.18.8 Output Files Conversion . . . 624
19.19 Neutrino Interface . . . 625
19.19.1 Files . . . 625
19.19.2 Models . . . 626
19.19.3 Distributions . . . 626
19.19.4 Samplers . . . 627
19.19.5 Steps . . . 627
19.19.6 Output File Conversion . . . 627
19.19.7 Additional Information . . . 628
19.20 Prescient Interface . . . 628
19.20.1 General Information . . . 628
19.20.2 Sampler . . . 628
19.20.3 Files . . . 628
19.20.4 Models . . . 629
19.20.5 Output Files Conversion . . . 629
19.20.6 Installation of Libraries . . . 630
20 Advanced Users: How to couple a new code . . . 631
20.1 Pre-requisites. . . . 632
20.2 Code Interface Creation . . . 634
20.2.1 Method: `generateCommand` . . . 635
20.2.2 Method: `createNewInput` . . . 636
20.2.3 Method: `getInputExtension` . . . 637
20.2.4 Method: `initialize` . . . 638
20.2.5 Method: `finalizeCodeOutput` . . . 638
20.2.6 Method: `checkForOutputFailure` . . . 639
20.2.7 Method: `setRunOnShell` . . . 639
20.2.8 Method: `setCsvLoadUtil` . . . 640
20.3 Tools for Developing Code Interfaces . . . 640
20.3.1 File Objects . . . 640
21 Advanced Users: How to create a RAVEN ExternalModel plugin . . . 643
21.1 ExternalModel Plugin Input . . . 644
21.2 ExternalModel Plugin Creation . . . 644
21.2.1 Method: `run` . . . 645
21.2.2 Method: `createNewInput` . . . 646
21.2.3 Method: `_readMoreXML` . . . 647
21.2.4 Method: `initialize` . . . 648
22 Advanced Users: How and When to create a RAVEN Template . . . 650
22.1 When to use a RAVEN Template . . . 650

22.2 How to create a RAVEN Template 650
22.2.1 Templated Workflows 651
22.2.2 Template Class 652
22.2.3 Template Interface 653
22.3 Example 653
22.3.1 Example Templated Workflow 654
22.3.2 Example Template Class 654
22.3.3 Example Template Interface 655

Appendix

A Appendix: Example Primer 657
A.1 Example 1. 657
A.2 Example 2. 660
References 668

1 Introduction

RAVEN is a software framework able to perform parametric and stochastic analysis based on the response of complex system codes. The initial development was aimed at providing dynamic risk analysis capabilities to the thermohydraulic code RELAP-7, currently under development at Idaho National Laboratory (INL). Although the initial goal has been fully accomplished, RAVEN is now a multi-purpose stochastic and uncertainty quantification platform, capable of communicating with any system code.

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

The development of RAVEN started in 2012 when, within the Nuclear Energy Advanced Modeling and Simulation (NEAMS) program, the need to provide 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) program. RISMC is one of the pathways defined within the Light Water Reactor Sustainability (LWRS) program.

In the RISMC approach, the goal is not just to identify the frequency of an event potentially leading to a system failure, but the proximity (or lack thereof) to key safety-related events. Hence, the approach is interested in identifying and increasing the safety margins related to those events. A safety margin is a numerical value quantifying the probability that a safety metric (e.g. peak pressure in a pipe) is exceeded under certain conditions.

Most of the capabilities, implemented having 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, any MOOSE-based App, etc.) or are currently ongoing for coupling RAVEN with several different software. The aim of this document is to detail the input requirements for RAVEN focusing on the input structure.

2 Manual Formats

In order to highlight some parts of the Manual having a particular meaning (e.g. input structure, examples, terminal commands, etc.), specific formats have been used. In this sections all the formats with a specific meaning are reported:

- ***Python Coding:***

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

- ***XML input example:***

```
<MainXMLBlock>
  ...
  <anXMLnode name='anObjectName' anAttribute='aValue'>
     <aSubNode>body</aSubNode>
  </anXMLnode>
  ...
</MainXMLBlock>
```

- ***Bash Commands:***

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

3 Installation

3.1 Overview

The installation of the RAVEN code is a straightforward procedure; depending on the usage purpose and machine architecture, the installation process slightly differs.

In the following sections, the recommended installation procedure is outlined. For alternatives, we encourage checking the RAVEN wiki. The machines on which RAVEN is tested and developed, however, use the standard installation procedures outlined below.

The installation process will involve three steps:

- Installing prerequisites, which depends on your operating system;
- Installing conda;
- Installing RAVEN.

Depending on your operating system (Windows in section 3.4, MacOSX in section 3.3, Ubuntu Linux in section 3.2), follow the instructions for installing prerequisites, then continue with installing conda (section 3.5), and then installing RAVEN (section 3.6).

3.2 Linux Ubuntu Installation

The following instructions are for installing RAVEN on a Linux machine running Ubuntu 16.04 or greater. Some explanations of alternatives for other Linux distributions may be provided on the RAVEN wiki.

To install the prerequisite packages, the following terminal command should be executed (note this requires administrative privileges):

```
sudo apt-get install libtool git python3-dev swig g++
```

3.2.0.1 Optional LateX installation

Optionally, if you want to be able to edit and rebuild the manuals, you can install TEX Live and its related packages:

```
sudo apt-get install texlive-latex-base \
texlive-extra-utils texlive-latex-extra texlive-math-extra
```

Once the above are installed, proceed with installing conda (see section 3.5).

3.3 Mac OSX Installation

When using an Apple Macintosh computer, software dependencies are met by following steps:

- Install the XCode command line tools from Apple,
- Install the XQuartz X-Window system server,

3.3.1 Installing XCode Command Line Tools

The XCode command line tools package from Apple Computer provides the C++ compilers and git source code control tools needed to obtain and build RAVEN. It is freely available from the Apple store. In order to obtain it the following command should be launched in an open terminal:

```
xcode-select --install
```

3.3.2 Installing XQuartz

XQuartz is an implementation of the X Server for the Mac OSX operating system. XQuartz is freely available on the web and can be downloaded from the link `https://dl.bintray.com/xquartz/downloads/XQuartz-2.7.9.dmg`.
After downloaded, install the package.

With XCode and XQuartz installed, continue on to install conda (see section 3.5).

Note: While `gcc` and `git` are also required, they are installed by default in the OSX system.

3.4 Microsoft Windows

The process of establishing the required environment for Windows is notably more involved than the other two systems; however, it is straightforward. First, RAVEN has the following prerequisites on Windows:

- A system running a 64-bit version of Microsoft Windows. Installation and operation has been verified on Windows 7, 10, and Windows Server 2012 R2 Standard.
- At least 9 Gigabytes of available disk space:
  - 0.5 GB for GIT SCM, including supporting tools and git source code control
  - 1.5 GB for Python language and supporting packages
  - 1 GB for RAVEN framework
  - 5.0 GB for the Visual Studio compiler needed to build RAVEN

3.4.1 A Visual Guide

Note: An illustrated version of this procedure may be found on the RAVEN wiki.

3.4.2 GIT SCM for Windows

RAVEN currently works on Windows using basic tools freely available online. The first software to be downloaded and installed is **Git SCM** available at `https://gitforwindows.org/`.

1. Obtain the latest Git SCM for Windows installer from `https://gitforwindows.org/` and install it. Install Git Bash and have the installer add Git Bash to your Windows *PATH* environment variables. The *PATH* can be updated either automatically (allowing the Git SCM installer to update it for you) or manually (Systems Properties - Environment Variables - Edit Environment Variables).

3.4.3 Install Python Language and Package Support

1. Download the latest 64-bit installer for Windows Python 3 from `https://conda.io/miniconda.html` and install it.
2. The installer will ask whether Python should be installed for only the logged in user or for all users. Either option will work for RAVEN.
3. have the installer add *conda* to your Windows *PATH* environment variables. The *PATH* can be updated either automatically (allowing the *conda* installer to update it for you) or manually (Systems Properties - Environment Variables - Edit Environment Variables).
4. Check the installation of Python and coda locating and testing the Python installation. Open a Windows command prompt and enter the command "*where python*", which attempts to locate a the Python language interpreter in the current system path. This looks like:

```
C:\Users\USERID> where python
C:\Users\USERID\AppData\Local\Continuum\Miniconda3\python.exe
```

3.4.4 Compiler Installation and Configuration

1. Download and install Visual Studio. A C++ language compiler that supports C++11 features is needed to perform this step. Microsoft's Visual Studio Community Edition is free and available from `https://www.visualstudio.com/downloads/`.

   The current version (as of this writing) is 2017. The 2015 and 2017 versions have been successfully used to build RAVEN. Professional and Enterprise versions of these will also work. If one of these is already present on your system, it is not necessary to obtain another one. Note that because C++11 language features are required, the "Microsoft Visual C++ Compiler for Python 2.7 or 3.x" often used for building Python add-ons will **not** work.

   After downloading and running the Visual Studio installer, it will ask what features to install. For building RAVEN, "Desktop development with C++" is needed at a minimum. Installation of other Visual Studio features should be fine.

Once the compiler installation and configuration is complete, you are prepared to install the RAVEN libraries (see section 3.5).

3.5 Conda: Python Dependencies

The standard installation procedure for RAVEN includes using Miniconda (often simply referred to as *conda*) to install the Python libraries required to run RAVEN. If conda cannot be made available on an operating system, refer to the wiki (listed above) for alternatives. To install miniconda, follow the instructions for your operating system at `https://conda.io/miniconda.html`.

Note: RAVEN currently works with Python 2.7, but it is recommended that Python 3 be used, so unless you have a reason to use 2.7, we recommend installing the 64 bit Python 3 version of miniconda.

Once conda is installed, proceed to installing RAVEN itself (section 3.6).

3.6 Installing RAVEN

Once the RAVEN dependencies have been installed and conda is present (see section 3.1), the rest of RAVEN can be installed.

The installation of RAVEN involves the following steps:

- Obtain the source code,
- Install the prerequisite Python libraries using conda,
- Compile

3.6.1 Obtaining RAVEN Source Code

RAVEN is hosted publicly as a `Git` repo on `GitHub` and can be viewed at `https://github.com/idaholab/raven/wiki`. In the event that access to `GitHub` is impossible, contact the user list and other arrangements may be possible. In general, however, using the git repository assures the most consistent usage and update process.

To clone RAVEN, navigate in a terminal to the desired destination, for example `/projects`. Then run the commands

```
git clone https://github.com/idaholab/raven.git
cd raven
git submodule update --init
```

This will obtain RAVEN as well as other submodules that RAVEN uses. In the future, whenever we declare a path starting with `raven/`, we refer to the cloned directory.

3.6.2 Installing Python Libraries

RAVEN depends heavily on Python, and uses conda to maintain a separate environment to prevent conflicts with other Python library installations. This separate environment is called `raven_libraries`.

In order to establish this environment, navigate to `raven`, then

- **Any unix-based systems (e.g. Macintosh, Linux, etc.)**:

```
cd scripts
./establish_conda_env.sh --install
```

- **Windows**:

```
cd scripts
bash.exe establish_conda_env.sh --install
```

Assure that there are no errors in this process, then continue to compiling RAVEN.

Note: If `conda` is not installed in the default location, then the path to the conda definitions needs to be provided, for example

- **Any unix-based systems (e.g. Macintosh, Linux, etc.)**:

```
cd scripts
./establish_conda_env.sh --install
   --conda-defs /path/to/miniconda3/etc/profile.d/conda.sh
```

- **Windows**:

```
cd scripts
bash.exe establish_conda_env.sh --install
  --conda-defs \path/\to\miniconda3\etc\profile.d\conda.sh
```

replacing `/path/to` with the install path for `conda`.

3.6.3 Compiling RAVEN

Once Python libraries are established and the source code present, navigate to `raven` and run

- **Any unix-based systems (e.g. Macintosh, Linux, etc.)**:

```
./build_raven
```

- **Windows**:

```
bash.exe build_raven
```

This will compile several dependent libraries. This step has the highest potential for revealing problems in the operating system setup, particularly for Windows. See troubleshooting on the RAVEN wikifor help sorting out difficulties.

3.6.4 Testing RAVEN

To test the installation of RAVEN, navigate to `raven`, then run the command

- **Any unix-based systems (e.g. Macintosh, Linux, etc.)**:

```
../run_tests -j2
```

- **Windows**:

```
bash.exe ./run_tests -j2
```

where `-j2` signifies running with 2 processors. If more processors are available, this can be increased, but using all or more than all of the available processes can slow down the testing dramatically. This command runs RAVEN's regression tests, analytic tests, and unit tests. The number of tests changes frequently as the code's needs change, and the time taken to run the tests depends strongly on the number of processors and processor speed.

At the end of the tests, a number passed, skipped, and failing will be reported. Having some skipped tests is expected; RAVEN has many tests that apply only to particular configurations or codes that are not present on all machines. However, no tests should fail; if there are problems, consult the troubleshooting section on the RAVEN wiki.

3.6.5 Updating RAVEN

RAVEN updates frequently, and new features are added while bugs are fixed on a regular basis. To update RAVEN, navigate to `raven`, then run the commands

- **Any unix-based systems (e.g. Macintosh, Linux, etc.)**:

```
git pull
./scripts/establish_conda_env.sh --install
./build_raven
```

- **Windows**:

```
git pull
bash.exe scripts/establish_conda_env.sh --install
bash.exe build_raven
```

3.6.6 In-use Testing

Whenever RAVEN is installed on a new computer or whenever there is a significant change to the operating system, in-use tests shall be conducted. Acceptable performance of RAVEN shall be confirmed by running the installation tests as described in 3.6.4.

4 Running RAVEN

The RAVEN code is a blend of C++, C, and Python software. The entry point resides on the Python side and is accessible via a command line interface. After following the instructions in the previous Section, RAVEN is ready to be used. The `raven_framework` script is in the raven folder. To run RAVEN, open a terminal and use the following command (replace `<inputFileName.xml>` with your RAVEN input file):

- **Any unix-based systems (e.g. Macintosh, Linux, etc.)**:

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

- **Windows**:

```
bash.exe raven_framework <inputFileName.xml>
```

Using `raven_framework` is the recommended way to run RAVEN. In the event bypassing the typical environment loading and checks is desired, it can also be run via the `Driver.py` script using python, with the input file as argument. However, this is not recommended, as it will use whatever default versions of Python and other libraries are discovered, rather than the matching libraries set up during installation.

Note: For Windows systems, we provided a convienient Batch script (`raven_framework.bat`) for running RAVEN avoiding to interact with the Windows command line terminal. More info on how to use it can be found in the RAVEN RAVEN wiki, section *Running RAVEN* (`https://github.com/idaholab/raven/wiki/runningRAVEN`).

5 Raven Input Structure

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

- **`<Simulation>`**: The root node containing the entire input, all of the following blocks fit inside the *Simulation* block.
- **`<RunInfo>`**: Specifies the calculation settings (number of parallel simulations, etc.).
- **`<Files>`**: Specifies the files to be used in the calculation.
- **`<Distributions>`**: Defines distributions needed for describing parameters, etc.
- **`<Samplers>`**: Sets up the strategies used for exploring an uncertain domain.
- **`<Optimizers>`**: Sets up the strategies used for minimizing/maximizing an objective function.
- **`<DataObjects>`**: Specifies internal data objects used by RAVEN.
- **`<Databases>`**: Lists the HDF5 databases used as input/output to a RAVEN run.
- **`<OutStreams>`**: Visualization and Printing system block.
- **`<Models>`**: Specifies codes, ROMs, post-processing analysis, etc.
- **`<Functions>`**: Details interfaces to external user-defined functions and modules. the user will be building and/or running.
- **`<Steps>`**: Combines other blocks to detail a step in the RAVEN workflow including I/O and computations to be performed.

Each of these blocks are explained in dedicated sections in the following chapters.

5.1 Comments

Comments may be included in the RAVEN input using standard XML comments, using `<!--` and `-->` as shown in the example below.

```
<Simulation>
  ...
  <!-- An Example Comment -->
  <Samplers>
  ...
```

Comments may be placed anywhere *except* before the **<Simulation>** node or after the **</Simulation>** node. Comments outside the root node will cause errors in maintaining input file compatability. Additionally, comments must completely surround any nodes they comment out. Comments are intended to completely remove blocks of code, or to add readability. For instance, the following is INCORRECT usage:

```
  <!--<Assembler> -->
  <!--</Assembler> -->
```

and the following is compatible usage for a code block:

```
  <!--<Samplers>
    <Monte Carlo name='mc'>
      ...
    </Monte Carlo>
    ...
  </Samplers> -->
```

5.2 Verbosity

Each block within RAVEN also makes use of a **verbosity** system, which allows a user to control the level of output to the user interface. These settings are declared globally as attributes in the **<Simulation>** node, and locally in each block. 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.

Examples of verbosity usage are included in many examples throughout this manual.

At the **<Simulation>** node, the following global variables can be set:

- **verbosity**, optional string, determines the global verbosity level. Defaults to **'all'**.
- **printTimeStamps**, optional boolean, determines whether time stamps will be added to printed messages. Defaults to true.
- **color**, optional boolean, determines whether ANSI color tags will be used in printed messages. Defaults to false.
- **profile**, optional comma-separated list, enables time profiling of parts of RAVEN. Options include **'jobs'**. Default is no profiling.

5.3 External Input Files

The **<ExternalXML>** node defines external input file (XML format) that can be used to replace any XML nodes under **<Simulation>** in the RAVEN input file. This node allows a user to load any external input file that contains the required XML nodes into the RAVEN input file. Each **<ExternalXML>** node has the following attributes:

- **node**, ***required string attribute***, user-defined XML node of RAVEN input file.
- **xmlToLoad**, ***required string attribute***, file name with its absolute or relative path. Note: if a relative path is specified, it must be relative with respect to the RAVEN input file.

For example, if the file `Models.xml` contain the required RAVEN input XML node **<Models>**, the RAVEN input file might appear as:

```
<Simulation>
  ...
  <Steps>
    ...
  </Steps>
  ...
  <ExternalXML node='Models'
     xmlToLoad='external_input/Models.xml'/>
  ...
</Simulation>
```

Another example, if the file `MultiRun.xml` contain the required RAVEN input XML node **<MultiRun>** under node **<Steps>**, the RAVEN input file might appear as:

```
<Simulation>
  ...
  <Steps>
```

```
    ...
    <ExternalXML node='MultiRun'
       xmlToLoad='external_input/MultiRun.xml'/>
    ...
  </Steps>
  ...
</Simulation>
```

6 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.

6.1 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

- **`<maxQueueSize>`**, ***integer, optional field***, specifies the number of parallel runs that can be staged for running simultaneously. The RAVEN architecture is inherently multithreaded

where a job queue is continuously monitored by a job handling thread. New jobs are added to this queue as they become available from the main thread of execution. Since the main thread is also responsible for collecting the results of previously finished jobs, it is possible that faster jobs may complete before the main thread can replenish the queue. By increasing this value, you are allowing RAVEN to consume more memory in order to stage more jobs, placing them in a pending job queue, with the benefit that slower job collection times will be masked as the job handler will flush the complete jobs and run whatever is available on the pending queue. With smaller values, RAVEN will consume less memory staging jobs, but there is potential that the job processing thread may be starved of jobs and waste parallel cycles as the code degrades to serially waiting for the main thread to complete collecting finished jobs. Where **`<batchSize>`** represents the number of jobs running, **`<maxQueueSize>`** represents the total number of jobs running plus the queued jobs. Values of **`<maxQueueSize>`** less than **`<batchSize>`** will be ignored. By default, **`<maxQueueSize>`** will be equal to **`<batchSize>`**.

- **`<Sequence>`**, ***comma separated string, required field***, is an ordered list of the step names that RAVEN will run (see Section 18).

- **`<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). **Note:** This flag is only active for Models of type "Code".
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. **Note:** This flag is only active for Models of type "Code".
Default: False

6.2 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>** command (default: `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 cores needed can be specified manually with the **<coresneeded>**. This is directly used in the qsub command select statement.
* The max memory needed can be specified with the **<memory>** XML node. This will be used in the qsub command select statement.
* The placement can be specified with the **<place>** XML node. This will be used in the qsub place statement.
* There is a "mpilegacy" mode. This probably will be removed in the future. In this mode 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 inter-

nal 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 6.4 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.)

6.3 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
```

6.4 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 6.2, 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 remoteRunCommand(self, runInfoDict):
    # If it returns a dictionary, then run the command in args
    # Example: {"args":["ssh","remotehost","raven_framework"]}
    # Note that this command needs to be able to tell when it
    # is running remotely, and then return None at that point
    return None

  def modifyInfo(self, runInfoDict):
    # modifyInfo 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.
    # In order to change them, return a dictionary with new values.
    # Those new values will be used.
    return {}

  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 providing a remote command for running RAVEN. 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 remoteRunCommand 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 `modifyInfo` method and returning a new dictionary with new values. The `runInfoDict` in the simulation is passed in.

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.

6.5 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`).

7 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.
- **subDirectory**, ***optional string attribute***, sub-directory that should be created in the perturbation process. The file specified in the body of the XML node should be located in the **subDirectory** under the **workingDir** specified in the **<RunInfo>** XML block (i.e. $workingDir/subDirectory$). If specified, the file will be placed in the sub-directory. For example, in a *MultiRun* step, the file will be copied into $workingDir/stepName/\%counter\%/subDirectory$, where $workingDir$ is the working directory specified in the *RunInfo* XML block, $stepName$ is the name of the step, $/\%counter\%$ is the realization identifier (e.g. 1,2, etc.) and $subDirectory$ is the sub-directory here specified. If not provided, defaults to an empty string.

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>
    <Input name='fileInSubDir' type=''
       subDirectory="theSubDirectory">theFileInTheSubDir.inp</Input>
  </Files>
   ...
</Simulation>
```

8 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. Additionally, set operations can be used to construct variable groups from other variable groups, by listing them in node text along with the operation to perform. 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 set operations can be performed by separating them with commas in the text of the group node, whether they be variable groups or single variables. In the event a circular dependency loop is detected, an error will be raised. VariableGroups are evaluated in the order of entries listed in their node text.

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.

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">x_first,-x1,+ x4,+x5</Group>
  <Group name="union"      >x_odd,+x_even</Group>
```

```
    <Group name="complement">x_odd,-x_first</Group>
    <Group name="intersect" >x_even,^x_first</Group>
    <Group name="sym_diff"  >x_odd,% x_first</Group>
  </VariableGroups>
  ...
  <DataObjects>
    <PointSet name="dataset">
      <Input>union</Input>
      <Output>y_group</Output>
    </PointSet>
  </DataObjects>
  ...
</Simulation>
```

9 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.

9.1 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.

9.1.1 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>
```

```
</Distributions>
```

Each distribution has a pre-defined, default support (domain) based on its definition, however these domains can be shifted/stretched using the appropriate **<low>** and **<high>** parameters where applicable, and/or truncated using the nodes in the example above, namely **<lowerBound>** and **<upperBound>**. For example, the Normal distribution domain is $[-\infty, +\infty]$, and thus cannot be shifted or stretched, as it is already unbounded, but can be truncated. RAVEN currently provides support for 13 1-Dimensional distributions. In the following paragraphs, all the input requirements are reported and commented.

9.1.1.1 Beta Distribution

The **Beta** distribution is parameterized by two positive shape parameters, denoted by α and β, that appear as exponents of the random variable. Its default support (domain) is $x \in [0, 1]$. The distribution domain can be changed, specifying new boundaries, to fit the user's needs. The user can specify a **Beta** distribution in two ways. The standard is to provide the parameters **<low>**, **<high>**, **<alpha>**, and **<beta>**. Alternatively, to approximate a normal distribution that falls to 0 at the endpoints, the user may provide the parameters **<low>**, **<high>**, and **<peakFactor>**. The peak factor is a value between 0 and 1 that determines the peakedness of the distribution. At 0 it is dome-like ($\alpha = \beta = 4$) and at 1 it is very strongly peaked around the mean ($\alpha = \beta = 100$). A reasonable approximation to a Gaussian normal is a peak factor of 0.5.

The specifications of this distribution must be defined within a **<Beta>** XML block. This XML node accepts one attribute:

- **name**, ***required string attribute***, user-defined name of this distribution. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.

This distribution can be initialized with the following children nodes:

- Standard initialization:
  - **<alpha>**, ***float, conditional required parameter***, first shape parameter. If specified, **<beta>** must also be inputted and **<peakFactor>** can not be specified.
  - **<beta>**, ***float, conditional required parameter***, second shape parameter. If specified, **<alpha>** must also be inputted and **<peakFactor>** can not be specified.
  - **<low>**, ***float, optional parameter***, lower domain boundary.
    Default: 0.0
  - **<high>**, ***float, optional parameter***, upper domain, boundary.
    Default: 1.0

- Alternative initialization:
  - **<peakFactor>**, ***float, optional parameter***, alternative to specifying **<alpha>** and **<beta>**. Acceptable values range from 0 to 1.
  - **<low>**, ***float, optional parameter***, lower domain boundary.
    Default: 0.0
  - **<high>**, ***float, optional parameter***, upper domain, boundary.
    Default: 1.0

Example:

```
<Distributions>
  ...
  <Beta name='aUserDefinedName'>
     <low>aFloatValue</low>
     <high>aFloatValue</high>
     <alpha>aFloatValue</alpha>
     <beta>aFloatValue</beta>
  </Beta>
  <Beta name='aUserDefinedName2'>
     <low>aFloatValue</low>
     <high>aFloatValue</high>
     <peakFactor>aFloatValue</peakFactor>
  </Beta>
  ...
</Distributions>
```

9.1.1.2 Exponential Distribution

The **Exponential** distribution has a default support of $x \in [0, +\infty)$.

The specifications of this distribution must be defined within an **<Exponential>** XML block. This XML node accepts one attribute:

- **name**, ***required string attribute***, user-defined name of this distribution. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.

This distribution can be initialized with the following child node:

- **<lambda>**, ***float, required parameter***, rate parameter.

- **<low>**, ***float, optional parameter***, lower domain boundary.
  Default: 0.0

Example:

```
<Distributions>
  ...
  <Exponential name='aUserDefinedName'>
    <lambda>aFloatValue</lambda>
    <low>aFloatValue</low>
  </Exponential>
  ...
</Distributions>
```

9.1.1.3 Gamma Distribution

The **Gamma** distribution is a two-parameter family of continuous probability distributions. The common exponential distribution and χ-squared distribution are special cases of the gamma distribution. Its default support is $x \in [0, +\infty]$.

The specifications of this distribution must be defined within a **<Gamma>** XML block. This XML node accepts one attribute:

- **name**, ***required string attribute***, user-defined name of this distribution. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.

This distribution can be initialized with the following children nodes:

- **<alpha>**, ***float, required parameter***, shape parameter.
- **<beta>**, ***float, optional parameter***, 1/scale or the inverse scale parameter.
  Default: 1.0
- **<low>**, ***float, optional parameter***, lower domain boundary.
  Default: 0.0

Example:

```
<Distributions>
  ...
  <Gamma name='aUserDefinedName'>
    <alpha>aFloatValue</alpha>
```

```
    <beta>aFloatValue</beta>
    <low>aFloatValue</low>
  </Gamma>
  ...
</Distributions>
```

9.1.1.4 Laplace Distribution

The **Laplace** distribution is a two-parameter continuous probability distribution. It is the distribution of the differences between two independent random variables with identical exponential distributions. Its default support is $x \in (-\infty, +\infty)$.

The specifications of this distribution must be defined within a **<Laplace>** XML block. This XML node accepts one attribute:

- **name**, ***required string attribute***, user-defined name of this distribution. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.

This distribution can be initialized with the following children nodes:

- **<location>**, ***float, required parameter***, determines the location or shift of the distribution.
- **<scale>**, ***float, required parameter***, must be greater than 0, and determines how spread out the distribution is.

Example:

```
<Distributions>
  ...
  <Laplace name='aUserDefinedName'>
    <location>aFloatValue</location>
    <scale>aFloatValue</scale>
  </Laplace>
  ...
</Distributions>
```

9.1.1.5 Logistic Distribution

The **Logistic** distribution is similar to the normal distribution with a CDF that is an instance of a logistic function ($Cdf(x) = \frac{1}{1+e^{-\frac{(x-location)}{scale}}}$). It resembles the normal distribution in shape but has

heavier tails (higher kurtosis). Its default support is $x \in [-\infty, +\infty]$.

The specifications of this distribution must be defined within a **<Logistic>** XML block. This XML node accepts one attribute:

- **name**, ***required string attribute***, user-defined name of this distribution. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.

This distribution can be initialized with the following children nodes:

- **<location>**, ***float, required parameter***, the distribution mean.
- **<scale>**, ***float, required parameter***, scale parameter that is proportional to the standard deviation ($\sigma^2 = \frac{1}{3}\pi^2 scale^2$).

Example:

```
<Distributions>
  ...
  <Logistic name='aUserDefinedName'>
    <location>aFloatValue</location>
    <scale>aFloatValue</scale>
  </Logistic>
  ...
</Distributions>
```

9.1.1.6 LogNormal Distribution

The **LogNormal** distribution is a distribution with the logarithm of the random variable being normally distributed. Its default support is $x \in [0, +\infty]$.

The specifications of this distribution must be defined within a **<LogNormal>** XML block. This XML node accepts one attribute:

- **name**, ***required string attribute***, user-defined name of this distribution. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.

This distribution can be initialized with the following children nodes:

- **<mean>**, ***float, required parameter***, the log of the distribution mean or expected value.
- **<sigma>**, ***float, required parameter***, standard deviation.

- **<low>**, ***float, optional parameter***, lower domain boundary.
  Default: 0.0

Note: The **<mean>** and **<sigma>** listed above are NOT the mean and standard deviation of the distribution; they are the mean and standard deviation of the log of the distribution. Using the following notation:

- μ_ℓ: the μ parameter of the lognormal distribution, which RAVEN expects in the **<mean>** node;
- σ_ℓ: the σ parameter of the lognormal distribution, which RAVEN expects in the **<sigma>** node;
- M: the user-desired mean value of the distribution;
- S: the user-desired standard deviation of the distribution;

a conversion is defined to translate from mean M and standard deviation S into the parameters RAVEN expects:

$$\mu_\ell = \log\left(\frac{M}{\sqrt{1+\frac{S^2}{M^2}}}\right), \tag{1}$$

$$\sigma_\ell = \sqrt{\log 1 + \frac{S^2}{M^2}}. \tag{2}$$

Example:

```
<Distributions>
  ...
  <LogNormal name='aUserDefinedName'>
    <mean>aFloatValue</mean>
    <sigma>aFloatValue</sigma>
    <low>aFloatValue</low>
  </LogNormal>
  ...
</Distributions>
```

9.1.1.7 LogUniform Distribution

The **LogNormal** distribution is a distribution associated to a variable $y = h(x) = e^x$ where variable x is uniform distributed. This distribution supports not only the case $y = h(x) = e^x$ (natural case) but also the case where $y = h(x) = 10^x$ (decimal case).

Its default support is $x \in [h(lowerBound), h(upperBound)]$.

The specifications of this distribution must be defined within a **<LogUniform>** XML block. This XML node accepts one attribute:

- **name**, ***required string attribute***, user-defined name of this distribution. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.

This distribution can be initialized with the following children nodes:

- **<lowerBound>**, ***float, required parameter***, domain lower boundary.
- **<upperBound>**, ***float, required parameter***, domain upper boundary.
- **<base>**, ***string, required parameter***, case type (decimal or natural).

Example:

```
<Distributions>
  ...
  <LogUniform name="x_dist">
    <upperBound>1.0</upperBound>
    <lowerBound>3.0</lowerBound>
    <base>natural</base>
  </LogUniform>
  ...
</Distributions>
```

9.1.1.8 Normal Distribution

The **Normal** distribution is an extremely useful continuous distribution. Its utility is due to the central limit theorem, which states that, under mild conditions, the mean of many random variables independently drawn from the same distribution is distributed approximately normally, irrespective of the form of the original distribution. Its default support is $x \in [-\infty, +\infty]$.

The specifications of this distribution must be defined within a **<Normal>** XML block. This XML node accepts one attribute:

- **name**, ***required string attribute***, user-defined name of this distribution. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.

This distribution can be initialized with the following children nodes:

- **<mean>**, ***float, required parameter***, the distribution mean or expected value.
- **<sigma>**, ***float, required parameter***, the standard deviation.

Example:

```
<Distributions>
  ...
  <Normal name='aUserDefinedName'>
    <mean>aFloatValue</mean>
    <sigma>aFloatValue</sigma>
  </Normal>
  ...
</Distributions>
```

9.1.1.9 Triangular Distribution

The **Triangular** distribution is a continuous distribution that has a triangular shape for its PDF. Like the uniform distribution, upper and lower limits are "known," but a "best guess," of the mode or center point is also added. It has been recommended as a "proxy" for the beta distribution. Its default support is $x \in [min, max]$.

The specifications of this distribution must be defined within a **<Triangular>** XML block. This XML node accepts one attribute:

- **name**, ***required string attribute***, user-defined name of this distribution. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.

This distribution can be initialized with the following children nodes:

- **<apex>**, ***float, required parameter***, peak location
- **<min>**, ***float, required parameter***, domain lower boundary.
- **<max>**, ***float, required parameter***, domain upper boundary.

Example:

```
<Distributions>
  ...
  <Triangular name='aUserDefinedName'>
    <apex>aFloatValue</apex>
```

```
    <min>aFloatValue</min>
    <max>aFloatValue</max>
  </Triangular>
  ...
</Distributions>
```

9.1.1.10 Uniform Distribution

The **Uniform** distribution is a continuous distribution with a rectangular-shaped PDF. It is often used where the distribution is only vaguely known, but upper and lower limits are known. Its default support is $x \in [lower, upper]$.

The specifications of this distribution must be defined within a **`<Uniform>`** XML block. This XML node accepts one attribute:

- **`name`**, ***required string attribute***, user-defined name of this distribution. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.

This distribution can be initialized with the following children nodes:

- **`<lowerBound>`**, ***float, required parameter***, domain lower boundary.
- **`<upperBound>`**, ***float, required parameter***, domain upper boundary.

Note: Since the Uniform distribution is a rectangular-shaped PDF, the truncation does not have any effect; this is the reason why the children nodes are the ones generally used for truncated distributions. Example:

```
<Distributions>
  ...
  <Uniform name='aUserDefinedName'>
    <lowerBound>aFloatValue</lowerBound>
    <upperBound>aFloatValue</upperBound>
  </Uniform>
  ...
</Distributions>
```

9.1.1.11 Weibull Distribution

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

rate is:

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

Its default support is $x \in [0, +\infty)$.

The specifications of this distribution must be defined within a **<Weibull>** XML block. This XML node accepts one attribute:

- **name**, ***required string attribute***, user-defined name of this distribution. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.

This distribution can be initialized with the following children nodes:

- **<k>**, ***float, required parameter***, shape parameter.
- **<lambda>**, ***float, required parameter***, scale parameter.
- **<low>**, ***float, optional parameter***, lower domain boundary.
  Default: 0.0

Example:

```
<Distributions>
  ...
  <Weibull name='aUserDefinedName'>
    <lambda>aFloatValue</lambda>
    <k>aFloatValue</k>
    <low>aFloatValue</low>
  </Weibull>
  ...
</Distributions>
```

9.1.1.12 Custom1D Distribution

The **Custom1D** distribution is a custom continuous distribution that can be initialized from a dataObject generated by RAVEN. This distribution cannot be initialized from a dataObject directly but through a .csv file. This file must contain the values of either cdf or pdf of the random variable sampled along the range of the desired random variable. In the distribution block of the RAVEN input file, the user needs to specify which file (including its working directory) needs to be used

to initialize the distribution. In addition, the user is required to specify which type (cdf or pdf) or values are contained in the file and also the IDs of both the random variable and cdf/pdf. Thus the csv file contains a set of points that samples the function $pdf(x)$ or $cdf(x)$ for several values of the stochastic variable x. The user needs to specify which variable IDs correspond to x and $pdf(x)$ (or $cdf(x)$). The distribution create a fourth order spline interpolation from the provided input points. Note that the support of this distribution is set between the minimum and maximum values of the random variable which are specified in the distribution input file.

Refer to the test example ($tests/framework/test_distributionCustom1D.xml$) for more clarification.

The specifications of this distribution must be defined within a **<Custom1D>** XML block. This XML node accepts one attribute:

- **name**, ***required string attribute***, user-defined name of this distribution. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.

This distribution can be initialized with the following children nodes:

- **<dataFilename>**, ***string, required parameter***, file name to be used to initialize the distribution.
- **<workingDir>**, ***string, optional parameter***, relative working directory that contains the input file.
- **<functionType>**, ***string, required parameter***, type of initialization values specified in the input file (pdf or cdf).
- **<variableID>**, ***string, required parameter***, ID of the variable contained in the input file.
- **<functionID>**, ***string, required parameter***, ID of the function associated to the variableID contained in the input file.

Example:

```
<Distributions>
  ...
    <Custom1D name="pdf_custom">
      <dataFilename>PointSetFile2_dump.csv</dataFilename>
      <functionID>pdf_values</functionID>
      <variableID>x</variableID>
      <functionType>pdf</functionType>
      <workingDir>custom1D/</workingDir>
    </Custom1D>
    <Custom1D name="cdf_custom">
```

```
      <dataFilename>PointSetFile3_dump.csv</dataFilename>
      <functionID>cdf_values</functionID>
      <variableID>x</variableID>
      <functionType>cdf</functionType>
      <workingDir>custom1D/</workingDir>
    </Custom1D>
  ...
</Distributions>
```

The example above initializes two distributions from two .csv files. For example, the first distribution retrieves the pdf values, located in the column with label pdf_values, for several locations of the variable located in the column with label x in the file $PointSetFile2_dump.csv$.

9.1.2 1-Dimensional Discrete Distributions.

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

9.1.2.1 Bernoulli Distribution

The **Bernoulli** distribution is a discrete distribution of the outcome of a single trial with only two results, 0 (failure) or 1 (success), with a probability of success p. It is the simplest building block on which other discrete distributions of sequences of independent Bernoulli trials can be based. Basically, it is the binomial distribution (k = 1, p) with only one trial. Its default support is $k \in 0, 1$.

The specifications of this distribution must be defined within a **`<Bernoulli>`** XML block. This XML node accepts one attribute:

- **`name`**, ***required string attribute***, user-defined name of this distribution. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.

This distribution can be initialized with the following child node:

- **`<p>`**, ***float, required parameter***, probability of success.

Example:

```
<Distributions>
  ...
```

```
  <Bernoulli name='aUserDefinedName'>
    <p>aFloatValue</p>
  </Bernoulli>
  ...
</Distributions>
```

9.1.2.2 Binomial Distribution

The **Binomial** distribution is the discrete probability distribution of the number of successes in a sequence of n independent yes/no experiments, each of which yields success with probability p. Its default support is $k \in 0, 1, 2, ..., n$.

The specifications of this distribution must be defined within a **<Binomial>** XML block. This XML node accepts one attribute:

- **name**, ***required string attribute***, user-defined name of this distribution. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.

This distribution can be initialized with the following children nodes:

- **<p>**, ***float, required parameter***, probability of success.
- **<n>**, ***integer, required parameter***, number of experiments.

Example:

```
<Distributions>
  ...
  <Binomial name='aUserDefinedName'>
    <n>aIntegerValue</n>
    <p>aFloatValue</p>
  </Binomial>
  ...
</Distributions>
```

9.1.2.3 Geometric Distribution

The **Geometric** distribution is a one-parameter discrete probability distribution. The distribution uses the probability p that trial will be successful. The geometric distribution gives the probability of observing k trials before the first success. Its support is $k \in 0, 1, 2, ..., n$.

The specifications of this distribution must be defined within a **<Geometric>** XML block.

This XML node accepts one attribute:

- **name**, ***required string attribute***, user-defined name of this distribution. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.

This distribution can be initialized with the following children nodes:

- **<p>**, ***float, required parameter***, the success fraction for the trials.

Example:

```
<Distributions>
  ...
  <Geometric name='aUserDefinedName'>
    <p>aFloatValue</p>
  </Geometric>
  ...
</Distributions>
```

9.1.2.4 Poisson Distribution

The **Poisson** distribution is a discrete probability distribution that expresses the probability of a given number of events occurring in a fixed interval of time and/or space if these events occur with a known average rate and independently of the time since the last event. Its default support is $k \in 1, 2, 3, 4, ...$

The specifications of this distribution must be defined within a **<Poisson>** XML block. This XML node accepts one attribute:

- **name**, ***required string attribute***, user-defined name of this distribution. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.

This distribution can be initialized with the following child node:

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

Example:

```
<Distributions>
  ...
  <Poisson name='aUserDefinedName'>
    <mu>aFloatValue</mu>
  </Poisson>
  ...
</Distributions>
```

9.1.2.5 Categorical Distribution

The **Categorical** distribution is a discrete distribution that describes the result of a random variable that can have K possible outcomes. The probability of each outcome is separately specified. The possible outcomes must be only numerical values (either integer or float numbers). No string can be assigned to any outcome. There is not necessarily an underlying ordering of these outcomes, but labels are assigned in describing the distribution (in the range 1 to K). The specifications of this distribution must be defined within a `<Categorical>` XML block. This XML node accepts one attribute:

- `name`, ***required string attribute***, user-defined name of this distribution. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.

This distribution can be initialized with the following child node:

- `<state>`, ***float, required parameter***, probability for outcome 1
  - `outcome`, ***float, required parameter***, outcome value.
- `<state>`, ***float, required parameter***, probability for outcome 2
  - `outcome`, ***float, required parameter***, outcome value.
- ...
- `<state>`, ***float, required parameter***, probability for outcome K
  - `outcome`, ***float, required parameter***, outcome value.

Example:

```
<Distributions>
  ...
    <Categorical name='testCategorical'>
```

```
        <state outcome="10">0.1</state>
        <state outcome="20">0.2</state>
        <state outcome="50">0.15</state>
        <state outcome="60">0.4</state>
        <state outcome="90">0.15</state>
    </Categorical>
  ...
</Distributions>
```

9.1.2.6 Uniform Discrete Distribution

The **UniformDiscrete** distribution is a discrete distribution which describes a random variable that can have N values having equal probability value. This distribution allows the user to choose two kinds of sampling strategies: with or without replacement. In case the "without replacement" strategy is used, the distribution samples from the set of specified N values reduced by the previously sampled values. After, the sampler has generated values for all variables, the distribution is resetted (i.e., the set of values that can be sampled is returned to N). In case the "with replacement" strategy is used, the distribution samples always from the complete set of specified N values.

The specifications of this distribution must be defined within a **`<Uniform Discrete>`** XML block. This XML node accepts one attribute:

- **`name`**, ***required string attribute***, user-defined name of this distribution. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.

This distribution can be initialized with the following child node:

- **`<lowerBound>`**, ***integer, required parameter***, lower bound.
- **`<upperBound>`**, ***integer, required parameter***, upper bound.
- **`<nPoints>`**, ***integer, optional parameter***, number of points between lower and upper bound
- **`<strategy>`**, ***string, required parameter***, type of sampling strategy (withReplacement or withoutReplacement).

Example:

```
<Distributions>
  ...
    <UniformDiscrete name="UD_dist">
```

```
      <lowerBound>3</lowerBound>
      <upperBound>8</upperBound>
      <strategy>orderedWithReplacement</strategy>
    </UniformDiscrete>
  ...
</Distributions>
```

9.1.2.7 Markov Categorical Distribution

The **MarkovCategorical** distribution is a specific discrete categorical distribution describes a random variable that can have K possible outcomes, based on the steady state probabilities provided by Markov model.

- **`<transition>`**, ***float, optional field***, the transition matrix of given Markov model.
- **`<dataFile>`**, ***string, optional xml node***. The path for the given data file, i.e. the transition matrix. In this node, the following attribute should be specified:
  - **`fileType`**, ***string, optional field***, the type of given data file, default is 'csv'.

  Note: Either **`<transition>`** or **`<dataFile>`** is required to provide the transition matrix.
- **`<workingDir>`**, ***string, optional field***, the path of working directory
- **`<state>`**, ***required xml node***. The output from this state indicates the probability for outcome 1. In this node, the following attribute should be specified:
  - **`outcome`**, ***float, required field***, outcome value.
  - **`index`**, ***integer, required field***, the index of steady state probabilities corresponding to the transition matrix.
- **`<state>`**, ***required xml node***. The output from this state indicates the probability for outcome 2. In this node, the following attribute should be specified:
  - **`outcome`**, ***float, required field***, outcome value.
  - **`index`**, ***integer, required field***, the index of steady state probabilities corresponding to the transition matrix.
- ...
- **`<state>`**, ***required xml node***. The output from this state indicates the probability for outcome K. In this node, the following attribute should be specified:

- `outcome`, ***float, required field***, outcome value.
- `index`, ***integer, required field***, the index of steady state probabilities corresponding to the transition matrix.

Example:

```
<Simulation>
 ...
  <Distributions>
    ...
    <MarkovCategorical name="x_dist">
        <!--dataFile fileType='csv'>transitionFile</dataFile-->
        <transition>
            -1.1   0.8   0.7
            0.8    -1.4  0.2
            0.3    0.6   -0.9
        </transition>
        <state outcome='1' index='1'/>
        <state outcome='2' index='2'/>
        <state outcome='4' index='3'/>
    </MarkovCategorical>
    ...
  </Distributions>
 ...
</Simulation>
```

9.2 N-Dimensional Probability Distributions

The group of N-Dimensional distributions allow the user to model stochastic dependences between parameters. Thus instead of using N distributions for N parameters, the user can define a single distribution lying in a N-Dimensional space. The following N-Dimensional Probability Distributions are available within RAVEN:

- MultivariateNormal: Multivariate normal distribution (see Section 9.2.1)
- NDInverseWeight: ND Inverse Weight interpolation distribution (see Section 9.2.2)
- NDCartesianSpline: ND spline interpolation distribution (see Section 9.2.3)

For NDInverseWeight and NDCartesianSpline distributions, the user provides the sampled values of either CDF or PDF of the distribution. The sampled values can be scattered distributed (for NDInverseWeight) or over a cartesian grid (for NDCartesianSpline).

The user could specify, for each N-Dimensional distribution, the parameters of the random number generator function:

- **`<initialGridDisc>`**, ***positive integer, optional field***, user-defined initial grid discretization. This parameter specifies the number of discretizations that need to be performed, initially, for each Dimension in order to find N-Dimensional coordinate that corresponds to the CDF represented by a random number (0-1);
- **`<tolerance>`**, ***float, optional field***, user-defined tolerance in order to find the N-D coordinates corresponding to a random number. This tolerance is expressed in terms of CDF.

in the **`<samplerInit>`** block defined in sampler block **`<samplerInit>`** (see Section 10).

9.2.1 MultivariateNormal Distribution

the multivariate normal distribution or multivariate Gaussian distribution, is a generalization of the one-dimensional (univariate) normal distribution to higher dimensions. The multivariate normal distribution is often used to describe, at least approximately, any set of (possibly) correlated real-valued random variables each of which clusters around a mean value. The multivariate normal distribution of a k-dimensional random vector $\mathbf{x} = [x_1, x_2, \ldots, x_k]$ can be written in the following notation: $\mathbf{x} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ with with k-dimensional mean vector

$$\boldsymbol{\mu} = [E[x_1], E[x_2], \ldots, E[x_k]]$$

and $k \times k$ covariance matrix

$$\boldsymbol{\Sigma} = [Cov[x_i, x_j]], i = 1, 2, \ldots, k; j = 1, 2, \ldots, k$$

The probability distribution function for this distribution is the following:

$$f_{\mathbf{x}}(x_1, \ldots, x_k) = \frac{1}{\sqrt{(2\pi)^k |\boldsymbol{\Sigma}|}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^{\mathrm{T}} \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu})\right),$$

The specifications of this distribution must be defined within the xml block **`<MultivariateNormal>`**. This XML node needs to contain the attributes:

- **`name`**, ***required string attribute***, user-defined identifier of this multivariate normal distribution. **Note:** As with other objects, this is the name that can be used to refer to this specific entity from other input XML blocks.
- **`method`**, ***required string attribute***, defines which method is used to generate the multivariate normal distribution. The only allowable methods are **`'spline'`** and **`'pca'`**.

In RAVEN the MultivariateNormal distribution can be initialized through the following keywords:

- **<mu>**, list of mean values of each dimension
- **<covariance>**, list of element values in the covariance matrix. There are two types of **<covariance>**, based on the **type**:
  - **type**, ***string, optional field***, specifies the type of covariance, the default **type** is **'abs'**. Possible values for **type** are **'abs'** and **'rel'**. **Note:** **'abs'** indicates the covariance is a normal covariance matrix, while **'rel'** indicates the covariance is a relative covariance matrix. In addition, method **'pca'** can be combined with both types, and method **'spline'** only accept the type **'abs'**
- **<transformation>**, ***XML node, optional field***, option to enable input parameter transformation using principal component analysis (PCA) approach. If this node is provided, PCA will be used to compute the principal components of input covariance matrix. The subnode **<rank>** is used to indicate the number of principal components that will be used for the input transformation. The content will specify one attribute:
  - **<rank>**, ***positive integer, required field***, user-defined dimensionality reduction.

Example:

```
<Distributions>
  ...
    <MultivariateNormal name='MultivariateNormal_test'
       method='spline'>
        <mu>0.0 60.0</mu>
        <covariance>
        1.0 0.7
        0.7 1.0
        </covariance>
    </MultivariateNormal>
    <MultivariateNormal name='MultivariateNormal_abs'
       method='pca'>
        <mu>0.0 60.0</mu>
        <covariance type='abs'>
        1.0 0.7
        0.7 1.0
        </covariance>
    </MultivariateNormal>
    <MultivariateNormal name='MultivariateNormal_rel'
       method='pca'>
        <mu>0.0 60.0</mu>
        <covariance type='rel'>
        1.0 0.7
```

```
        0.7 1.0
        </covariance>
    </MultivariateNormal>
  ...
</Distributions>
```

In the following, we defined a distribution with a transformation node using PCA method. The number of principal components is defined in **<rank>**. In this distribution, PCA is employed to restruct the multivariate normal distribution. In addition, the size of uncorrelated variables is also determined by **<rank>**.

```
<Distributions>
  ...
    <MultivariateNormal name='MultivariateNormal_test'
       method='pca'>
        <mu>0.0 10.0 20.0</mu>
        <covariance type="abs">
           1.0   0.7   -0.2
           0.7   1.0   0.4
           -0.2  0.4   1.0
        </covariance>
        <transformation>
          <rank>2</rank>
        </transformation>
    </MultivariateNormal>
  ...
</Distributions>
```

9.2.2 NDInverseWeight Distribution

The NDInverseWeight distribution creates a N-Dimensional distribution given a set of points scattered distributed. These points sample the PDF of the original distribution. Distribution values (PDF or CDF) are calculated using the inverse weight interpolation scheme.

The specifications of this distribution must be defined within a **<NDInverseWeight>** XML block. This XML node accepts the following attributes:

- **name**, ***required string attribute***, user-defined name of this distribution. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.

In RAVEN the NDInverseWeight distribution can be initialized through the following nodes:

- **<p>**, ***float, required parameter***, power parameter. Greater values of p assign greater influence to values closest to the interpolated point.
- **<data_filename>**, ***string, required parameter***, name of the data file containing scattered values (file type '.txt').
  - **type**, ***required string attribute***, indicates if the data in indicated file is PDF or CDF.
- **<working_dir>**, ***string, required parameter***, folder location of the data file

Example:

```
<Distributions>
  ...
  <NDInverseWeight name='...'>
        <p>...</p>
        <dataFilename type='...'>...</dataFilename>
        <workingDir>...</workingDir>
  </NDInverseWeight>
  ...
</Distributions>
```

Each data entry contained in data_filename is listed row by row and must be listed as follows:

- number of dimensions
- number of sampled points
- ND coordinate of each sampled point
- value of each sampled point

As an example, the following shows the data entries contained in data_filename for a 3-dimensional data set that contained two sampled CDF values: ([0.0,0.0,0.0], 0.1) and ([1.0, 1.0,0.0], 0.8)

Example scattered data file:

```
3
2
0.0
0.0
0.0
1.0
1.0
```

```
0.0
0.1
0.8
```

9.2.3 NDCartesianSpline Distribution

The NDCartesianSpline distribution creates a N-Dimensional distribution given a set of points regularly distributed on a cartesian grid. These points sample the PDF of the original distribution. Distribution values (PDF or CDF) are calculated using the ND spline interpolation scheme.

The specifications of this distribution must be defined within a **<NDCartesianSpline>** XML block. This XML node accepts the following attributes:

- **name**, ***required string attribute***, user-defined name of this distribution. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.

In RAVEN the NDCartesianSpline distribution can be initialized through the following nodes:

- **<data_filename>**, ***string, required parameter***, name of the data file containing scattered values (file type '.txt').
  - **type**, ***required string attribute***, indicates if the data in indicated file is PDF or CDF.
- **<working_dir>**, ***string, required parameter***, folder location of the data file

Example:

```
<Distributions>
  ...
  <NDCartesianSpline name='...'>
        <dataFilename type='...'>...</dataFilename>
        <workingDir></workingDir>
  </NDCartesianSpline>
  ...
</Distributions>
```

Each data entry contained in data_filename is listed row by row and must be listed as follows:

- number of dimensions
- number of discretization for each dimension

- discretization values for each dimension
- value of each sampled point

As an example, the following shows the data entries contained in data_filename for a 2-dimensional CDF data set on the following grid (x, y):

- first dimension (x): -0.5, 0.5
- first dimension (y): 1.0 2.0 3.0

Example scattered data file:

```
2
2
3
-0.5
0.5
1.0
2.0
3.0
CDF value of (-0.5,1.0)
CDF value of (+0.5,1.0)
CDF value of (-0.5,2.0)
CDF value of (+0.5,2.0)
CDF value of (-0.5,3.0)
CDF value of (+0.5,3.0)
```

10 Samplers

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

- **Forward** (see Section 10.1)
- **Dynamic Event Tree (DET)** (see Section 10.2)
- **Adaptive** (see Section 10.3)

Before analyzing each sampler in detail, it is important to mention that each type has a similar syntax to input the variables to be "sampled". In the example below, the variable **'variableName'** is going to be sampled by the Sampler **'whatever'** using the distribution named **'aDistribution'**.

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

As reported in section 19, the variable naming syntax, for external driven codes, depends on the way the "code interface" has been implemented. For example, if the code has an input structure like the one reported below (YAML), the variable name might be **'I-Level|II-Level|variable'**. In this way, the relative code interface (and input parser) will know which variable needs to be perturbed and the "recipe" to access it. As reported in 19, its syntax is chosen by the developer of

the "code interface" and is implemented in the interface only (no modifications are needed in the RAVEN code).

Example YAML based Input:

```
[I-Level]
  [./II-Level]
    variable = xxx
  [../]
[]
```

Example XML block to define the variables and associated distributions:

```
<variable name='I-Level|II-Level|variable'>
  <distribution>exampleDistribution</distribution>
</variable>
```

If the variable is associated to a multi-dimensional ND distribution, it is needed to specify which dimension of the ND distribution is associated to such variable. An example is shown below: the variable "variableX" is associated to the third dimension of the ND distribution "ND-distribution".

```
<variable name='variableX'>
     <distribution dim='3'>NDdistribution</distribution>
</variable>
```

For most codes, it is prudent that there are no redundant inputs; however there are cases where this is not reality. For example, if there is a variable **'inner_radius'** and a variable **'outer_radius'**, there may be a third variable **'thickness'** that is actually derived from the previous two, as **'thickness'** = **'outer_radius'** - **'inner_radius'**. RAVEN supports this type of redundant input through a Function entity. In this case, instead of a **<distribution>** node in the **<variable>** block, there is a **<function>** node, specifying the name of the function (defined in the **<Functions>** block). In order to work properly, this function must have a method named "evaluate" that returns a single python float object. In this way, multiple variables can be associated with the same function. For example,

```
...
<Functions>
  <External name='torus_calcs' file='torus_calcs.py'>
    <variable>outer_radius</variable>
    <variable>inner_radius</variable>
  </External>
<Functions>
...
```

```
<Samplers>
  <WhatEverSampler name='myExampleSampler'>
    <variable name='inner_radius'>
      <distribution>inner_dist</distribution>
    </variable>
    <variable name='outer_radius'>
      <distribution>outer_dist</distribution>
    </variable>
    <variable name='thickness'>
      <function>torus_calcs</function>
    </variable>
  </WhatEverSampler>
</Samplers>
```

The corresponding function file **'torus_calcs.py'** needs the following method:

```
def evaluate(self):
  return self.outer_radius - self.inner_radius
```

The **'thickness'** parameter will still be treated as an input for the sake of csv printing and DataObjects storage.
Note: It is important to notice that if the user use variables with no-Python compatible names (e.g. parenthesis, etc.), the **<alias>** system needs to be used to alias the variables.

In the sampler class a special node exists: the **<sampler_init>** node. This node contains specific parameters that characterize each particular sampler. In addition, **<sampler_init>** might contain the information regarding the random generator function for each N-Dimensional distribution (specified in the **<dist_init>** node):

- initial_grid_disc
- tolerance

An example of **<dist_init>** node is provided below:

```
<distInit>
    <distribution name= 'ND_dist_name'>
         <initialGridDisc>5</initialGridDisc>
          <tolerance>0.2</tolerance>
     </distribution>
  </distInit>
```

In the **<sampler_init>** node it is possible to add also the subnode **<globalGrid>**. The **<globalGrid>** can be used in two cases:

- 1D distributions: an identical grid that is associated to several distributions
- ND distribution: a grid associated to a single ND distribution. This is the case when a stratified sampling is performed on the CDF of an ND distribution: the **<globalGrid>** is shared among the variables associated to the Nd distribution

10.1 Forward Samplers

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

- **Monte Carlo (MC)**
- **Stratified**
- **Grid Based**
- **Sparse Grid Collocation**
- **Sobol Decomposition**
- **Response Surface Design of Experiment**
- **Factorial Design of Experiment**
- **Ensemble Forward Sampling strategy**
- **Custom Sampling strategy**

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

10.1.1 Monte Carlo

The **Monte-Carlo** sampling approach is one of the most well-known and widely used approaches to perform exploration of the input space. The main idea behind MonteCarlo sampling is to randomly perturb the input space according to uniform or parameter-based probability density functions.

The specifications of this sampler must be defined within a **`<MonteCarlo>`** XML block. This XML node accepts the following attributes:

- **name**, *required string attribute*, user-defined name of this Sampler. N.B. As for the other objects, this is the name that can be used to refer to this specific entity from other input blocks (xml);

In the **MonteCarlo** input block, the user needs to specify the variables need to be sampled. As already mentioned, these variables are inputted within consecutive xml blocks called **`<variable>`**. In addition, the settings for this sampler need to be specified in the **`<samplerInit>`** XML block:

- **`<samplerInit>`**, ***XML node, required parameter***. In this xml-node,the following xml sub-nodes need to be specified:
  - **`<limit>`**, ***integer,required field***, number of MonteCarlo samples needs to be generated;
  - **`<initialSeed>`**, ***integer, optional field***, initial seeding of random number generator
  - **`<reseedEachIteration>`**, ***boolean/string(case insensitive), optional field***, perform a re-seeding for each sample generated (True values = True, yes, y, t).
    Default: False;
  - **`<distInit>`**, ***integer, optional field***, in this node the user specifies the initialization of the random number generator function for each N-Dimensional Probability Distributions (see Section 9.2).
  - **`<samplingType>`**, ***string, optional field***, sub-type of sampling
    Default: None. the user can choose to perform a Monte-Carlo sampling where the location of the samples in the input space is uniformly distributed and not generated accordingly to the specific set of distributions. This can be specificed in the **`<samplingType>`** with the kewyword "uniform". This option works only if all the distributions have an upper and lower bound specified (i.e., **`<lowerBound>`** and **`<upperBound>`**). Allowed fields for this node are "None" and "uniform".

- **`<variable>`**, ***XML node, required parameter*** can specify the following attribute:
  - **`name`**, ***required string attribute***, user-defined name of this variable.

- **shape**, ***comma-separated integers, optional field***, determines the number of samples and shape of samples to be taken. For example, **shape**="2,3" will provide a 2 by 3 matrix of values, while **shape**="10" will produce a vector of 10 values. Omitting this optional attribute will result in a single scalar value instead. Each of the values in the matrix or vector will be the same as the single sampled value. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

This **<variable>** recognizes the following child nodes:

- **<distribution>**, ***string, required field***, name of the distribution that is associated to this variable. Its name needs to be contained in the **<Distributions>** block explained in Section 9. In addition, if NDDistribution is used, the attribute **dim** is required. **Note:** Alternatively, this node must be omitted if the **<function>** node is supplied.
- **<function>**, ***string, required field***, name of the function that defines the calculation of this variable from other distributed variables. Its name needs to be contained in the **<Functions>** block explained in Section 16. This function must implement a method named "evaluate". **Note:** Alternatively, this node must be ommitted if the **<distribution>** node is supplied.

- **<constant>**, ***XML node, optional parameter*** the user is able to input variables that need to be kept constant. For doing this, as many **<constant>** nodes as needed can be input. There are options for setting a constant. To simply set the constant's value, the body of the node contains the constant value, and the **<constant>** node has the following attributes:

  - **name**, ***required string attribute***, user-defined name of this constant.
  - **shape**, ***comma-separated integers, optional field***, determines the shape of samples of the constant value. For example, **shape**="2,3" will shape the values into a 2 by 3 matrix, while **shape**="10" will shape into a vector of 10 values. Unlike the **<variable>**, the constant requires each value be entered; the number of required values is equal to the product of the **shape**. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

  Alternatively, the constant value can be read from a DataObject that has been identified as a **<ConstantSource>** for this Sampler. In this case, the body of the **<constant>** node is the name of the variable that needs to be read from the **<ConstantSource>**, and the **<constant>** node has the following additional attributes in addition to the those above:

  - **source**, ***required string attribute***, the name of the DataObject containing the value to be used for this constant. This must be the name of one of the **<ConstantSource>** DataObjects.
  - **index**, ***optional integer attribute***, the index of the realization in the source DataObject that contains the value to use for the constant.
    Default: the last entry

By way of example, consider the following Sampler definition. The constant will be named **'C'** in the Sampler, and its value is taken from the DataObject **'MyConstant'**, which is identified in the **<ConstantSource>** node. To find the value of the constant in **'MyConstant'**, the Sampler will look at the realization with index **'3'** for the value of variable **'A'** to use as the constant value.

```
<Samplers>
  <WhatEverSampler name='whatever'>
    <ConstantSource class='DataObjects'
       type='PointSet'>MyConstants</ConstantSource>
    <constant name='C' source='MyConstants'
       index='3'>A</constant>
  </WhatEverSampler>
</Samplers>
```

If the input parameters are correlated, the **MonteCarlo** sampling approach can be also used if the user specified a multivariate distributions inside the **<Distributions>** (see Section 9.2). Furthermore, if the covariance matrix is provided and the input parameters is assumed to have the multivariate normal distribution, one can also use **MonteCarlo** approach to sample the input parameters in the transformed space (aka subspace, reduced space). If this is the case, the user needs to provide additional information, i.e. the **<transformation>** under **<MultivariateNormal>** of **<Distributions>** (more information can be found in Section 9.2). In addition, the node **<variablesTransformation>** is also required for **MonteCarlo** sampling. This node is used to tranform the variables specified by **<latentVariables>** in the transformed space of input into variables spefified by **<manifestVariables>** in the input space. The variables listed in **<latentVariables>** should be predefined in **<variable>**, and the variables listed in **<manifestVariables>** are used by the **<Models>**.

- **<variablesTransformation>**, ***optional field***. this XML node accepts one attribute:
  - **distribution**, ***required string attribute***, the name for the distribution defined in the XML node **<Distributions>**. This attribute indicates the values of **<manifestVariables>** are drawn from **distribution**.

  In addition, this XML node also accepts three childen nodes:
  - **<latentVariables>**, ***comma separated string, required field***, user-defined latent variables that are used for the variables transformation. All the variables listed under this node should be also mentioned in **<variable>**.
  - **<manifestVariables>**, ***comma separated string, required field***, user-defined manifest variables that can be used by the **model**.

- **`<manifestVariablesIndex>`**, ***comma separated string, optional field***, user-defined manifest variables indices paired with **`<manifestVariables>`**. These indices indicate the position of manifest variables associated with multivariate normal distribution defined in the XML node **`<Distributions>`**. The indices should be postive integer. If not provided, the code will use the positions of manifest variables listed in **`<manifestVariables>`** as the indices.
- **`<method>`**, ***string, required field***, the method that is used for the variables transformation. The currently available method is '**pca**'.

Assembler Objects These objects are either required or optional depending on the functionality of the MonteCarlo Sampler. The objects must be listed with a rigorous syntax that, except for the XML node tag, is common among all the objects. Each of these nodes must contain 2 attributes that are used to identify them within the simulation framework:

- **`class`**, ***required string attribute***, the main "class" of the listed object. For example, it can be **`'Models'`**, **`'Functions'`**, etc.
- **`type`**, ***required string attribute***, the object identifier or sub-type. For example, it can be **`'ROM'`**, **`'External'`**, etc.

The **MonteCarlo** approach requires or optionally accepts the following object types:

- **`<Restart>`**, ***string, optional field***, the body of this XML node must contain the name of an appropriate **DataObject** defined in the **`<DataObjects>`** block (see Section 12). It is used as a "restart" tool, where it accepts pre-existing solutions in the PointSet instead of recalculating solutions.

The following node is an additional option when a restart DataObject is provided:

- **`<restartTolerance>`**, ***float, optional field***, the body of this XML node must contain a valid floating point value. If a **`<Restart>`** node is supplied for this **`<Sampler>`**, this node offers a way to determine how strictly matching points are determined. Given a point in the input space, if that point is within a relative Euclidean distance (equal to the tolerance) of a restart point, the nearest restart point will be used.
  Default: 1e-14

- **`<ConstantSource>`**, ***string, optional field***, the body of this XML node must contain the name of an appropriate **DataObject** defined in the **`<DataObjects>`** block (see Section 12). It is used as a source from which constants can take values.

Example:

```
<Samplers>
  ...
  <MonteCarlo name='MCname'>
    <samplerInit>
      <limit>10</limit>
      <initialSeed>200286</initialSeed>
      <reseedEachIteration>false</reseedEachIteration>
      <distInit>
        <distribution name= 'ND_InverseWeight_P'>
          <initialGridDisc>10</initialGridDisc>
          <tolerance>0.2</tolerance>
        </distribution>
      </distInit>
    </samplerInit>
    <variable name='var1'>
      <distribution>aDistributionNameDefinedInDistributionBlock
      </distribution>
    </variable>
    <Restart class='DataObjects' type='PointSet'>data</Restart>
  </MonteCarlo>
  ...
</Samplers>
  ...
  <PointSet name="data">
    <Input>var1</Input>
    <Output>ans</Output>
  </PointSet>
  ...
```

10.1.2 Grid

The **Grid** sampling approach is probably the simplest exploration approach that can be employed to explore an uncertain domain. The idea is to construct an N-dimensional grid where each dimension is represented by one uncertain variable. This approach performs the sampling at each node of the grid. The sampling of the grid consists in evaluating the answer of the system under all possible combinations among the different variables' values with respect to a predefined discretization metric. In RAVEN two discretization metrics are available: 1) cumulative distribution function, and 2) value. Thus, the grid meshing can be input via probability or variable values. Regarding the N-dimensional distributions, the user can specify for each dimension the type of grid to be used

(i.e., value or CDF). Note the discretization of the CDF, only for the grid sampler, is performed on the marginal distribution for the specific variable considered.

The specifications of this sampler must be defined within a **<Grid>** XML block. This XML node accepts one attribute:

- **name**, ***required string attribute***, user-defined name of this sampler. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.

In the **<Grid>** input block, the user needs to specify the variables to sample. As already mentioned, these variables are specified within consecutive **<variable>** XML blocks:

- **<variable>**, ***XML node, required parameter*** can specify the following attribute:
  - **name**, ***required string attribute***, user-defined name of this variable.
  - **shape**, ***comma-separated integers, optional field***, determines the number of samples and shape of samples to be taken. For example, **shape**="2,3" will provide a 2 by 3 matrix of values, while **shape**="10" will produce a vector of 10 values. Omitting this optional attribute will result in a single scalar value instead. Each of the values in the matrix or vector will be the same as the single sampled value. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

  This **<variable>** recognizes the following child nodes:

  - **<distribution>**, ***string, required field***, name of the distribution that is associated to this variable. Its name needs to be contained in the **<Distributions>** block explained in Section 9. In addition, if NDDistribution is used, the attribute **dim** is required. **Note:** Alternatively, this node must be omitted if the **<function>** node is supplied.
  - **<function>**, ***string, required field***, name of the function that defines the calculation of this variable from other distributed variables. Its name needs to be contained in the **<Functions>** block explained in Section 16. This function must implement a method named "evaluate". **Note:** Alternatively, this node must be ommitted if the **<distribution>** node is supplied.
  - **<grid>**, ***space separated floats, required field***, the content of this XML node depends on the definition of the associated attributes:
    * **type**, ***required string attribute***, user-defined discretization metric type: 1) **'CDF'**, the grid will be specified based on cumulative distribution function probability thresholds, and 2) **'value'**, the grid will be provided using variable values.
    * **construction**, ***required string attribute***, how the grid needs to be constructed, independent of its type (i.e. **'CDF'** or **'value'**).

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

* `construction='equal'`. The grid is going to be constructed equally-spaced (`type='value'`) or equally probable (`type='CDF'`). This construction type requires the definition of additional attributes:
  - `steps`, ***required integer attribute***, number of equally spaced/probable discretization steps.

  This construction type requires that the content of the `<grid>` node represents the lower and upper bounds (either in probability or value). Two values need to be specified; the lowest one will be considered as the $lowerBound$, the largest, the $upperBound$. The lower and upper bounds are checked against the associated `<distribution>` bounds. If one or both of them falls outside the distribution's bounds, the code will raise an error. The $stepSize$ is determined as follows: $stepSize = (upperBound - lowerBound)/steps$
* `construction='custom'`. The grid will be directly specified by the user. No additional attributes are needed. This construction type requires that the `<grid>` node contains the actual mesh bins. For example, if the grid `type` is `'CDF'`, in the body of `<grid>`, the user will specify the CDF probability thresholds (nodalization in probability). All the bins are checked against the associated `<distribution>` bounds. If one or more of them falls outside the distribution's bounds, the code will raise an error.

Note: The `<grid>` node is only required if a `<distribution>` node is supplied. In the case of a `<function>` node, no grid information is requested.

- `<constant>`, ***XML node, optional parameter*** the user is able to input variables that need to be kept constant. For doing this, as many `<constant>` nodes as needed can be input. There are options for setting a constant. To simply set the constant's value, the body of the node contains the constant value, and the `<constant>` node has the following attributes:
  - `name`, ***required string attribute***, user-defined name of this constant.
  - `shape`, ***comma-separated integers, optional field***, determines the shape of samples of the constant value. For example, `shape`="2,3" will shape the values into a 2 by 3 matrix, while `shape`="10" will shape into a vector of 10 values. Unlike the `<variable>`, the constant requires each value be entered; the number of required values is equal to the product of the `shape`. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

Alternatively, the constant value can be read from a DataObject that has been identified as a `<ConstantSource>` for this Sampler. In this case, the body of the `<constant>` node is the name of the variable that needs to be read from the `<ConstantSource>`, and the `<constant>` node has the following additional attributes in addition to the those above:

- **source**, ***required string attribute***, the name of the DataObject containing the value to be used for this constant. This must be the name of one of the **<ConstantSource>** DataObjects.
- **index**, ***optional integer attribute***, the index of the realization in the source DataObject that contains the value to use for the constant.
  Default: the last entry

By way of example, consider the following Sampler definition. The constant will be named **'C'** in the Sampler, and its value is taken from the DataObject **'MyConstant'**, which is identified in the **<ConstantSource>** node. To find the value of the constant in **'MyConstant'**, the Sampler will look at the realization with index **'3'** for the value of variable **'A'** to use as the constant value.

```
<Samplers>
  <WhatEverSampler name='whatever'>
    <ConstantSource class='DataObjects'
       type='PointSet'>MyConstants</ConstantSource>
    <constant name='C' source='MyConstants'
       index='3'>A</constant>
  </WhatEverSampler>
</Samplers>
```

If the input parameters are correlated, the **Grid** sampling approach can be also used if the user specified a multivariate distributions inside the **<Distributions>** (see Section 9.2). Furthermore, if the covariance matrix is provided and the input parameters is assumed to have the multivariate normal distribution, one can also use **Grid** approach to sample the input parameters in the transformed space (aka subspace, reduced space). This means one creates the grids of variables listed by **<latentVariables>** in the transformed space. If this is the case, the user needs to provide additional information, i.e. the **<transformation>** under **<MultivariateNormal>** of **<Distributions>** (more information can be found in Section 9.2). In addition, the node **<variablesTransformation>** is also required for **Grid** sampling. This node is used to tranform the variables specified by **<latentVariables>** in the transformed space of input into variables spefified by **<manifestVariables>** in the input space. The variables listed in **<latentVariables>** should be predefined in **<variable>**, and the variables listed in **<manifestVariables>** are used by the **<Models>**.

- **<variablesTransformation>**, ***optional field***. this XML node accepts one attribute:
  - **distribution**, ***required string attribute***, the name for the distribution defined in the XML node **<Distributions>**. This attribute indicates the values of **<manifestVariables>** are drawn from **distribution**.

  In addition, this XML node also accepts three childen nodes:

- **<latentVariables>**, ***comma separated string, required field***, user-defined latent variables that are used for the variables transformation. All the variables listed under this node should be also mentioned in **<variable>**.
- **<manifestVariables>**, ***comma separated string, required field***, user-defined manifest variables that can be used by the **model**.
- **<manifestVariablesIndex>**, ***comma separated string, optional field***, user-defined manifest variables indices paired with **<manifestVariables>**. These indices indicate the position of manifest variables associated with multivariate normal distribution defined in the XML node **<Distributions>**. The indices should be postive integer. If not provided, the code will use the positions of manifest variables listed in **<manifestVariables>** as the indices.
- **<method>**, ***string, required field***, the method that is used for the variables transformation. The currently available method is '**pca**'.

Assembler Objects These objects are either required or optional depending on the functionality of the Grid Sampler. The objects must be listed with a rigorous syntax that, except for the XML node tag, is common among all the objects. Each of these nodes must contain 2 attributes that are used to identify them within the simulation framework:

- **class**, ***required string attribute***, the main "class" of the listed object. For example, it can be **'Models'**, **'Functions'**, etc.
- **type**, ***required string attribute***, the object identifier or sub-type. For example, it can be **'ROM'**, **'External'**, etc.

The **Grid** approach requires or optionally accepts the following object types:

- **<Restart>**, ***string, optional field***, the body of this XML node must contain the name of an appropriate **DataObject** defined in the **<DataObjects>** block (see Section 12). It is used as a "restart" tool, where it accepts pre-existing solutions in the PointSet instead of recalculating solutions.

The following node is an additional option when a restart DataObject is provided:

- **<restartTolerance>**, ***float, optional field***, the body of this XML node must contain a valid floating point value. If a **<Restart>** node is supplied for this **<Sampler>**, this node offers a way to determine how strictly matching points are determined. Given a point in the input space, if that point is within a relative Euclidean distance (equal to the tolerance) of a restart point, the nearest restart point will be used.
  Default: 1e-14

- **<ConstantSource>**, ***string, optional field***, the body of this XML node must contain the name of an appropriate **DataObject** defined in the **<DataObjects>** block (see Section 12). It is used as a source from which constants can take values.

Example:

```
<Samplers>
  ...
  <Grid name='Gridname'>
    <variable name='var1'>
      <distribution>aDistributionNameDefinedInDistributionBlock1
      </distribution>
      <grid type='value' construction='equal' steps='100' >0.2
         10</grid>
    </variable>
    <variable name='var2'>
      <distribution>aDistributionNameDefinedInDistributionBlock2
      </distribution>
      <grid type='CDF' construction='equal' steps='5' >0.2
         0.8</grid>
    </variable>
    <variable name='var3'>
      <distribution>aDistributionNameDefinedInDistributionBlock3
      </distribution>
      <grid type='value' construction='equal' steps='100' >0.2
         21.0</grid>
    </variable>
    <variable name='var4'>
      <distribution>aDistributionNameDefinedInDistributionBlock4
      </distribution>
      <grid type='CDF' construction='equal' steps='5' >0.2
         1.0</grid>
    </variable>
    <variable name='var5'>
      <distribution>aDistributionNameDefinedInDistributionBlock5
      </distribution>
      <grid type='value' construction='custom'>0.2 0.5
         10.0</grid>
    </variable>
    <variable name='var6'>
      <distribution>aDistributionNameDefinedInDistributionBlock6
      </distribution>
      <grid type='CDF' construction='custom'>0.2 0.5 1.0</grid>
```

```
    </variable>
    <Restart class='DataObjects' type='PointSet'>data</Restart>
    <restartTolerance>1e-6</restartTolerance>
  </Grid>
  ...
</Samplers>
  ...
  <PointSet name="data">
    <Input>var1,var2,var3,var4,var5,var6</Input>
    <Output>ans</Output>
  </PointSet>
  ...
```

Note: A restart example is included here but is not necessary in general.

10.1.3 Sparse Grid Collocation

Sparse Grid Collocation 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 you construct an N-dimensional grid, with each uncertain variable providing an axis. Along each axis, the points of evaluation correspond to quadrature points necessary to integrate polynomials (see 15.3.3). In the simplest (and most naive) case, a N-Dimensional 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 specifications of this sampler must be defined within a `<SparseGridCollocation>` XML block. .

- `name`, ***required string attribute***, user-defined name of this sampler. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.
- `parallel`, ***optional string attribute***, option to disable parallel construction of the sparse grid. Because of increasing computational expense with increasing input space dimension, RAVEN will default to parallel construction of the sparse grid.
- `outfile`, ***optional string attribute***, option to allow the generated sparse grid points and weights to be printed to a file with the given name.
  Default: True

In the `<SparseGridCollocation>` input block, the user needs to specify the variables to

sample. As already mentioned, these variables are specified within consecutive **<variable>** XML blocks:

- **<variable>**, ***XML node, required parameter*** can specify the following attribute:
  - **name**, ***required string attribute***, user-defined name of this variable.
  - **shape**, ***comma-separated integers, optional field***, determines the number of samples and shape of samples to be taken. For example, **shape**="2,3" will provide a 2 by 3 matrix of values, while **shape**="10" will produce a vector of 10 values. Omitting this optional attribute will result in a single scalar value instead. Each of the values in the matrix or vector will be the same as the single sampled value. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

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

  - **<distribution>**, ***string, required field***, name of the distribution that is associated to this variable. Its name needs to be contained in the **<Distributions>** block explained in Section 9. In addition, if NDDistribution is used, the attribute **dim** is required. **Note:** Alternatively, this node must be omitted if the **<function>** node is supplied.
  - **<function>**, ***string, required field***, name of the function that defines the calculation of this variable from other distributed variables. Its name needs to be contained in the **<Functions>** block explained in Section 16. This function must implement a method named "evaluate". **Note:** Alternatively, this node must be ommitted if the **<distribution>** node is supplied.

- **<constant>**, ***XML node, optional parameter*** the user is able to input variables that need to be kept constant. For doing this, as many **<constant>** nodes as needed can be input. There are options for setting a constant. To simply set the constant's value, the body of the node contains the constant value, and the **<constant>** node has the following attributes:
  - **name**, ***required string attribute***, user-defined name of this constant.
  - **shape**, ***comma-separated integers, optional field***, determines the shape of samples of the constant value. For example, **shape**="2,3" will shape the values into a 2 by 3 matrix, while **shape**="10" will shape into a vector of 10 values. Unlike the **<variable>**, the constant requires each value be entered; the number of required values is equal to the product of the **shape**. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

  Alternatively, the constant value can be read from a DataObject that has been identified as a **<ConstantSource>** for this Sampler. In this case, the body of the **<constant>** node is the name of the variable that needs to be read from the **<ConstantSource>**, and the **<constant>** node has the following additional attributes in addition to the those above:

- `source`, ***required string attribute***, the name of the DataObject containing the value to be used for this constant. This must be the name of one of the `<ConstantSource>` DataObjects.
- `index`, ***optional integer attribute***, the index of the realization in the source DataObject that contains the value to use for the constant.
  Default: the last entry

By way of example, consider the following Sampler definition. The constant will be named `'C'` in the Sampler, and its value is taken from the DataObject `'MyConstant'`, which is identified in the `<ConstantSource>` node. To find the value of the constant in `'MyConstant'`, the Sampler will look at the realization with index `'3'` for the value of variable `'A'` to use as the constant value.

```
<Samplers>
  <WhatEverSampler name='whatever'>
    <ConstantSource class='DataObjects'
       type='PointSet'>MyConstants</ConstantSource>
    <constant name='C' source='MyConstants'
       index='3'>A</constant>
  </WhatEverSampler>
</Samplers>
```

Because of the tight coupling between the Sampler and the ROM in stochastic collocation for generalized polynomial chaos, the Sampler needs access to the ROM via the assembler do determine the polynomials, quadratures, and importance weights to use in each dimension (see 15.3.3).

Assembler Objects These objects are either required or optional depending on the functionality of the SparseGridCollocation Sampler. The objects must be listed with a rigorous syntax that, except for the XML node tag, is common among all the objects. Each of these nodes must contain 2 attributes that are used to identify them within the simulation framework:

- `class`, ***required string attribute***, the main "class" of the listed object. For example, it can be `'Models'`, `'Functions'`, etc.
- `type`, ***required string attribute***, the object identifier or sub-type. For example, it can be `'ROM'`, `'External'`, etc.

The **SparseGridCollocation** approach requires or optionally accepts the following object types:

- `<ROM>`, ***string, required field***, the body of this XML node must contain the name of an appropriate ROM defined in the `<Models>` block (see Section 15.3).

- **<Restart>**, ***string, optional field***, the body of this XML node must contain the name of an appropriate **DataObject** defined in the **<DataObjects>** block (see Section 12). It is used as a "restart" tool, where it accepts pre-existing solutions in the PointSet instead of recalculating solutions.

The following node is an additional option when a restart DataObject is provided:

- **<restartTolerance>**, ***float, optional field***, the body of this XML node must contain a valid floating point value. If a **<Restart>** node is supplied for this **<Sampler>**, this node offers a way to determine how strictly matching points are determined. Given a point in the input space, if that point is within a relative Euclidean distance (equal to the tolerance) of a restart point, the nearest restart point will be used.
  Default: 1e-14

- **<ConstantSource>**, ***string, optional field***, the body of this XML node must contain the name of an appropriate **DataObject** defined in the **<DataObjects>** block (see Section 12). It is used as a source from which constants can take values.

```
Example:
<Samplers>
  ...
  <SparseGridCollocation name="mySG" parallel="0">
    <variable name="x1">
      <distribution>myDist1</distribution>
    </variable>
    <variable name="x2">
      <distribution>myDist2</distribution>
    </variable>
    <ROM class = 'Models' type = 'ROM' >SCROM</ROM>
    <Restart class = 'DataObjects' type = 'PointSet' >solns</Restart>
  </SparseGridCollocation>
  ...
</Samplers>
  ...
  <PointSet name="solns">
    <Input>x1,x2</Input>
    <Output>y</Output>
  </PointSet>
  ...
```

In general, **SparseGridCollocation** requires uncorrelated input parameters. If the input parameters are correlated, one can transform the correlated parameters into uncorrelated parameters; the **SparseGridCollocation** can also be used with the uncorrelated parameters in the transformed space. Like in the **Grid** sampler, if the covariance matrix is provided and the input parameters

are assumed to have the multivariate normal distribution, the **SparseGridCollocation** can be used. This means one creates the sparse grids of variables listed by **`<latentVariables>`** in the transformed space. If this is the case, the user needs to provide additional information, i.e. the **`<transformation>`** under **`<MultivariateNormal>`** of **`<Distributions>`** (more information can be found in Section 9.2). In addition, the node **`<variablesTransformation>`** is also required for **SparseGridCollocation** sampler. This node is used to tranform the variables specified by **`<latentVariables>`** in the transformed space of input into variables spefified by **`<manifestVariables>`** in the input space. The variables listed in **`<latentVariables>`** should be predefined in **`<variable>`**, and the variables listed in **`<manifestVariables>`** are used by the **`<Models>`**.

- **`<variablesTransformation>`**, ***optional field***. this XML node accepts one attribute:
  - **`distribution`**, ***required string attribute***, the name for the distribution defined in the XML node **`<Distributions>`**. This attribute indicates the values of **`<manifestVariables>`** are drawn from **`distribution`**.

  In addition, this XML node also accepts three childen nodes:
  - **`<latentVariables>`**, ***comma separated string, required field***, user-defined latent variables that are used for the variables transformation. All the variables listed under this node should be also mentioned in **`<variable>`**.
  - **`<manifestVariables>`**, ***comma separated string, required field***, user-defined manifest variables that can be used by the **`model`**.
  - **`<manifestVariablesIndex>`**, ***comma separated string, optional field***, user-defined manifest variables indices paired with **`<manifestVariables>`**. These indices indicate the position of manifest variables associated with multivariate normal distribution defined in the XML node **`<Distributions>`**. The indices should be postive integer. If not provided, the code will use the positions of manifest variables listed in **`<manifestVariables>`** as the indices.
  - **`<method>`**, ***string, required field***, the method that is used for the variables transformation. The currently available method is '**pca**'.

```
...
<Models>
    ...
    <ExternalModel ModuleToLoad="lorentzAttractor_noK"
       name="PythonModule" subType="">
        <variables>sigma,rho,beta,x,y,z,time,z0,y0,z0</variables>
    </ExternalModel>
    <ROM name="SCROM" subType="GaussPolynomialRom">
        <Target>and</Target>
```

```
        <Features>x1,y1,z1</Features>
        <IndexSet>TensorProduct</IndexSet>
        <PolynomialOrder>1</PolynomialOrder>
    </ROM>
    ...
</Models>

<Distributions>
    ...
    <MultivariateNormal name='MVNDist' method='pca'>
        <transformation>
            <rank>3</rank>
        </transformation>
        <mu>0.0 1.0 2.0</mu>
        <covariance type="abs">
            1.0       0.6      -0.4
            0.6       1.0      0.2
            -0.4      0.2      0.8
        </covariance>
    </MultivariateNormal>
    ...
</Distributions>

<Samplers>
  ...
  <SparseGridCollocation name='SC'>
        <variable name='x0'>
            <distribution dim='1'>MVNDist</distribution>
        </variable>
        <variable name='y0'>
            <distribution dim='2'>MVNDist</distribution>
        </variable>
        <variable name='z0'>
            <distribution dim='3'>MVNDist</distribution>
        </variable>
        <variablesTransformation model="PythonModule">
            <latentVariables>x1,y1,z1</latentVariables>
            <manifestVariables>x0,y0,z0</manifestVariables>
            <method>pca</method>
        </variablesTransformation>
        <ROM class = 'Models' type = 'ROM' >SCROM</ROM>
        <Restart class="DataObjects"
```

```
            type="PointSet">solns</Restart>
  </SparseGridCollocation>
  ...
</Samplers>
  ...
  <PointSet name="solns">
    <Input>x0,y0,z0</Input>
    <Output>ans</Output>
  </PointSet>
  ...
```

10.1.4 Sobol

The **Sobol** sampler uses high-density model reduction (HDMR) a.k.a. Sobol decomposition to approximate a function as the sum of increasing-complexity interactions. At its lowest level (order 1), it treats the function as a sum of the reference case plus a functional of each input dimesion separately. At order 2, it adds functionals to consider the pairing of each dimension with each other dimension. The benefit to this approach is considering several functions of small input cardinality instead of a single function with large input cardinality. This allows reduced order models like generalized polynomial chaos (see 15.3.3) to approximate the functionals accurately with few computations runs. This Sobol sampler uses the associated HDMRRom (see 15.3.4) to determine at what points the input space need be evaluated. Since Sobol sampler relies on SparseGridCollocation, it is also compatible with multivariate normal distribution objects. The **`<Sobol>`** node supports the following attributes:

- **`name`**, ***required string attribute***, user-defined name of this sampler. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.
- **`parallel`**, ***optional string attribute***, option to disable parallel construction of the sparse grid. Because of increasing computational expense with increasing input space dimension, RAVEN will default to parallel construction of the sparse grid.
  Default: True

In the **`<Sobol>`** input block, the user needs to specify the variables to sample. As already mentioned, these variables are specified within consecutive **`<variable>`** XML blocks:

- **`<variable>`**, ***XML node, required parameter*** can specify the following attribute:
  - **`name`**, ***required string attribute***, user-defined name of this variable.

- **shape**, ***comma-separated integers, optional field***, determines the number of samples and shape of samples to be taken. For example, **shape**="2,3" will provide a 2 by 3 matrix of values, while **shape**="10" will produce a vector of 10 values. Omitting this optional attribute will result in a single scalar value instead. Each of the values in the matrix or vector will be the same as the single sampled value. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

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

- **<distribution>**, ***string, required field***, name of the distribution that is associated to this variable. Its name needs to be contained in the **<Distributions>** block explained in Section 9. In addition, if NDDistribution is used, the attribute **dim** is required. **Note:** Alternatively, this node must be omitted if the **<function>** node is supplied.
- **<function>**, ***string, required field***, name of the function that defines the calculation of this variable from other distributed variables. Its name needs to be contained in the **<Functions>** block explained in Section 16. This function must implement a method named "evaluate". **Note:** Alternatively, this node must be ommitted if the **<distribution>** node is supplied.

- **<constant>**, ***XML node, optional parameter*** the user is able to input variables that need to be kept constant. For doing this, as many **<constant>** nodes as needed can be input. There are options for setting a constant. To simply set the constant's value, the body of the node contains the constant value, and the **<constant>** node has the following attributes:

  - **name**, ***required string attribute***, user-defined name of this constant.
  - **shape**, ***comma-separated integers, optional field***, determines the shape of samples of the constant value. For example, **shape**="2,3" will shape the values into a 2 by 3 matrix, while **shape**="10" will shape into a vector of 10 values. Unlike the **<variable>**, the constant requires each value be entered; the number of required values is equal to the product of the **shape**. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

  Alternatively, the constant value can be read from a DataObject that has been identified as a **<ConstantSource>** for this Sampler. In this case, the body of the **<constant>** node is the name of the variable that needs to be read from the **<ConstantSource>**, and the **<constant>** node has the following additional attributes in addition to the those above:

  - **source**, ***required string attribute***, the name of the DataObject containing the value to be used for this constant. This must be the name of one of the **<ConstantSource>** DataObjects.
  - **index**, ***optional integer attribute***, the index of the realization in the source DataObject that contains the value to use for the constant.
    Default: the last entry

By way of example, consider the following Sampler definition. The constant will be named **'C'** in the Sampler, and its value is taken from the DataObject **'MyConstant'**, which is identified in the **<ConstantSource>** node. To find the value of the constant in **'MyConstant'**, the Sampler will look at the realization with index **'3'** for the value of variable **'A'** to use as the constant value.

```
<Samplers>
  <WhatEverSampler name='whatever'>
    <ConstantSource class='DataObjects'
       type='PointSet'>MyConstants</ConstantSource>
    <constant name='C' source='MyConstants'
       index='3'>A</constant>
  </WhatEverSampler>
</Samplers>
```

Like the **SparseGridCollocation**, if multivariate normal distribution is provided, the following node need to be specified:

- **<variablesTransformation>**, ***optional field***. this XML node accepts one attribute:
  - **distribution**, ***required string attribute***, the name for the distribution defined in the XML node **<Distributions>**. This attribute indicates the values of **<manifestVariables>** are drawn from **distribution**.

  In addition, this XML node also accepts three childen nodes:
  - **<latentVariables>**, ***comma separated string, required field***, user-defined latent variables that are used for the variables transformation. All the variables listed under this node should be also mentioned in **<variable>**.
  - **<manifestVariables>**, ***comma separated string, required field***, user-defined manifest variables that can be used by the **model**.
  - **<manifestVariablesIndex>**, ***comma separated string, optional field***, user-defined manifest variables indices paired with **<manifestVariables>**. These indices indicate the position of manifest variables associated with multivariate normal distribution defined in the XML node **<Distributions>**. The indices should be postive integer. If not provided, the code will use the positions of manifest variables listed in **<manifestVariables>** as the indices.
  - **<method>**, ***string, required field***, the method that is used for the variables transformation. The currently available method is '**pca**'.

Because of the tight coupling between the Sobol sampler and the HDMRRom, the Sampler needs access to the ROM via the assembler do determine the polynomials, quadratures, Sobol order, and importance weights to use in each dimension (see 15.3.4).

Assembler Objects These objects are either required or optional depending on the functionality of the Sobol Sampler. The objects must be listed with a rigorous syntax that, except for the XML node tag, is common among all the objects. Each of these nodes must contain 2 attributes that are used to identify them within the simulation framework:

- **`class`**, ***required string attribute***, the main "class" of the listed object. For example, it can be **`'Models'`**, **`'Functions'`**, etc.
- **`type`**, ***required string attribute***, the object identifier or sub-type. For example, it can be **`'ROM'`**, **`'External'`**, etc.

The **Sobol** approach requires or optionally accepts the following object types:

- **`<ROM>`**, ***string, required field***, the body of this XML node must contain the name of an appropriate ROM defined in the **`<Models>`** block (see Section 15.3).

- **`<Restart>`**, ***string, optional field***, the body of this XML node must contain the name of an appropriate **DataObject** defined in the **`<DataObjects>`** block (see Section 12). It is used as a "restart" tool, where it accepts pre-existing solutions in the PointSet instead of recalculating solutions.

The following node is an additional option when a restart DataObject is provided:

- **`<restartTolerance>`**, ***float, optional field***, the body of this XML node must contain a valid floating point value. If a **`<Restart>`** node is supplied for this **`<Sampler>`**, this node offers a way to determine how strictly matching points are determined. Given a point in the input space, if that point is within a relative Euclidean distance (equal to the tolerance) of a restart point, the nearest restart point will be used.
  Default: 1e-14

- **`<ConstantSource>`**, ***string, optional field***, the body of this XML node must contain the name of an appropriate **DataObject** defined in the **`<DataObjects>`** block (see Section 12). It is used as a source from which constants can take values.

```
Example:
<Samplers>
  ...
  <Sobol name="mySobol" parallel="0">
    <variable name="x1">
      <distribution>myDist1</distribution>
```

```
    </variable>
    <variable name="x2">
      <distribution>myDist2</distribution>
    </variable>
    <ROM class = 'Models' type = 'ROM' >myHDMR</ROM>
    <Restart class="DataObjects" type="PointSet">solns</Restart>
  </Sobol>
  ...
</Samplers>
  ...
  <PointSet name="solns">
    <Input>x1,y2</Input>
    <Output>ans</Output>
  </PointSet>
  ...
```

10.1.5 Stratified

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

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

Then simple random sampling or systematic sampling is applied within each stratum. It is worthwhile to note that the well-known Latin hypercube sampling represents a specialized version of the stratified approach, when the domain strata are constructed in equally-probable CDF bins.

The specifications of this sampler must be defined within a **`<Stratified>`** XML block. This XML node accepts one attribute:

- **`name`**, ***required string attribute***, user-defined name of this sampler. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.

In the **`<Stratified>`** input block, the user needs to specify the variables to sample. As already mentioned, these variables are specified within consecutive **`<variable>`** XML blocks:

- **`<variable>`**, ***XML node, required parameter*** can specify the following attribute:

- **name**, ***required string attribute***, user-defined name of this variable.
- **shape**, ***comma-separated integers, optional field***, determines the number of samples and shape of samples to be taken. For example, **shape**="2,3" will provide a 2 by 3 matrix of values, while **shape**="10" will produce a vector of 10 values. Omitting this optional attribute will result in a single scalar value instead. Each of the values in the matrix or vector will be the same as the single sampled value. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

This **<variable>** recognizes the following child nodes:

- **<distribution>**, ***string, required field***, name of the distribution that is associated to this variable. Its name needs to be contained in the **<Distributions>** block explained in Section 9. In addition, if NDDistribution is used, the attribute **dim** is required. **Note:** Alternatively, this node must be omitted if the **<function>** node is supplied.
- **<function>**, ***string, required field***, name of the function that defines the calculation of this variable from other distributed variables. Its name needs to be contained in the **<Functions>** block explained in Section 16. This function must implement a method named "evaluate". **Note:** Alternatively, this node must be ommitted if the **<distribution>** node is supplied.
- **<grid>**, ***space separated floats, required field***, the content of this XML node depends on the definition of the associated attributes:
  * **type**, ***required string attribute***, user-defined discretization metric type: 1) **'CDF'**, the grid will be specified based on cumulative distribution function probability thresholds, and 2) **'value'**, the grid will be provided using variable values.
  * **construction**, ***required string attribute***, how the grid needs to be constructed, independent of its type (i.e. **'CDF'** or **'value'**).

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

  * **construction='equal'**. The grid is going to be constructed equally-spaced (**type='value'**) or equally probable (**type='CDF'**). This construction type requires the definition of additional attributes:
    · **steps**, ***required integer attribute***, number of equally spaced/probable discretization steps.

    This construction type requires that the content of the **<grid>** node represents the lower and upper bounds (either in probability or value). Two values need to be specified; the lowest one will be considered as the $lowerBound$, the largest, the $upperBound$. The lower and upper bounds are checked against the associated **<distribution>** bounds. If one or both of them falls outside the distribution's bounds, the code will raise an error. The $stepSize$ is determined as follows:
    $stepSize = (upperBound - lowerBound)/steps$

* `construction='custom'`. The grid will be directly specified by the user. No additional attributes are needed. This construction type requires that the `<grid>` node contains the actual mesh bins. For example, if the grid `type` is `'CDF'`, in the body of `<grid>`, the user will specify the CDF probability thresholds (nodalization in probability). All the bins are checked against the associated `<distribution>` bounds. If one or more of them falls outside the distribution's bounds, the code will raise an error.

Note: The `<grid>` node is only required if a `<distribution>` node is supplied. In the case of a `<function>` node, no grid information is requested.

- `<constant>`, ***XML node, optional parameter*** the user is able to input variables that need to be kept constant. For doing this, as many `<constant>` nodes as needed can be input. There are options for setting a constant. To simply set the constant's value, the body of the node contains the constant value, and the `<constant>` node has the following attributes:

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

  – `shape`, ***comma-separated integers, optional field***, determines the shape of samples of the constant value. For example, `shape`="2,3" will shape the values into a 2 by 3 matrix, while `shape`="10" will shape into a vector of 10 values. Unlike the `<variable>`, the constant requires each value be entered; the number of required values is equal to the product of the `shape`. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

  Alternatively, the constant value can be read from a DataObject that has been identified as a `<ConstantSource>` for this Sampler. In this case, the body of the `<constant>` node is the name of the variable that needs to be read from the `<ConstantSource>`, and the `<constant>` node has the following additional attributes in addition to the those above:

  – `source`, ***required string attribute***, the name of the DataObject containing the value to be used for this constant. This must be the name of one of the `<ConstantSource>` DataObjects.

  – `index`, ***optional integer attribute***, the index of the realization in the source DataObject that contains the value to use for the constant.
  Default: the last entry

  By way of example, consider the following Sampler definition. The constant will be named `'C'` in the Sampler, and its value is taken from the DataObject `'MyConstant'`, which is identified in the `<ConstantSource>` node. To find the value of the constant in `'MyConstant'`, the Sampler will look at the realization with index `'3'` for the value of variable `'A'` to use as the constant value.

```
    <Samplers>
      <WhatEverSampler name='whatever'>
```

```
        <ConstantSource class='DataObjects'
           type='PointSet'>MyConstants</ConstantSource>
        <constant name='C' source='MyConstants'
           index='3'>A</constant>
      </WhatEverSampler>
    </Samplers>
```

In addition, the settings for this sampler need to be specified in the **`<samplerInit>`** XML block:

- **`<samplerInit>`**, ***XML node, required parameter***. In this xml-node,the following xml sub-nodes need to be specified:
  - **`<initialSeed>`**, ***integer, optional field***, initial seeding of random number generator
  - **`<distInit>`**, ***integer, optional field***, in this node the user specifies the initialization of the random number generator function for each N-Dimensional Probability Distributions (see Section 9.2).

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

Assembler Objects These objects are either required or optional depending on the functionality of the Stratified Sampler. The objects must be listed with a rigorous syntax that, except for the XML node tag, is common among all the objects. Each of these nodes must contain 2 attributes that are used to identify them within the simulation framework:

- **`class`**, ***required string attribute***, the main "class" of the listed object. For example, it can be **`'Models'`**, **`'Functions'`**, etc.
- **`type`**, ***required string attribute***, the object identifier or sub-type. For example, it can be **`'ROM'`**, **`'External'`**, etc.

The **Stratified** approach requires or optionally accepts the following object types:

- **`<Restart>`**, ***string, optional field***, the body of this XML node must contain the name of an appropriate **DataObject** defined in the **`<DataObjects>`** block (see Section 12). It is used as a "restart" tool, where it accepts pre-existing solutions in the PointSet instead of recalculating solutions.

The following node is an additional option when a restart DataObject is provided:

- **<restartTolerance>**, ***float, optional field***, the body of this XML node must contain a valid floating point value. If a **<Restart>** node is supplied for this **<Sampler>**, this node offers a way to determine how strictly matching points are determined. Given a point in the input space, if that point is within a relative Euclidean distance (equal to the tolerance) of a restart point, the nearest restart point will be used.
  Default: 1e-14

- **<ConstantSource>**, ***string, optional field***, the body of this XML node must contain the name of an appropriate **DataObject** defined in the **<DataObjects>** block (see Section 12). It is used as a source from which constants can take values.

Example:

```
<Samplers>
  ...
  <Stratified name='StratifiedName'>
    <variable name='var1'>
      <distribution>aDistributionNameDefinedInDistributionBlock1
      </distribution>
      <grid type='CDF' construction='equal' steps='5' >0.2
         0.8</grid>
    </variable>
    <variable name='var2'>
      <distribution>aDistributionNameDefinedInDistributionBlock2
      </distribution>
      <grid type='value' construction='equal' steps='100' >0.2
         21.0</grid>
    </variable>
    <variable name='var3'>
      <distribution>aDistributionNameDefinedInDistributionBlock3
      </distribution>
      <grid type='CDF' construction='custom'>0.2 0.5 1.0</grid>
    </variable>
  </Stratified>
  ...
</Samplers>
```

For N-dimensional (ND) distributions, there are two different approahes to perform the stratified sampling. In the first approach, the subgroups is determined by the joint CDF of given multivariate distributions. If this approach is used, the sampling is performed on a grid on a CDF, while the user is required to specify the same CDF grid for all the dimensions of the ND distribution. This is possible by defining a **<globalGrid>** node and associate such **<globalGrid>** to each variable belonging to the ND distribution as follows.

```
<Samplers>
  ...
  <Stratified name='StratifiedName'>
        <variable name='x0'>
            <distribution
               dim='1'>ND_InverseWeight_P</distribution>
            <grid type='globalGrid'>name_grid1</grid>
        </variable>
        <variable name='y0,z0'>
            <distribution
               dim='2'>ND_InverseWeight_P</distribution>
            <grid type='globalGrid'>name_grid1</grid>
        </variable>
        <globalGrid>
            <grid name='name_grid1' type='CDF'
               construction='custom'>0.1 1.0 0.2</grid>
        </globalGrid>
  </Stratified>
  ...
</Samplers>
...
```

The second approach is different than the first approach. Like in the **Grid** sampling, if the covariance matrix is provided and the input parameters is assumed to have the multivariate normal distribution, one can also use **Stratified** approach to sample the input parameters in the transformed space (aka subspace, reduced space). This means one creates the grids of variables listed by **`<latentVariables>`** in the transformed space. If this is the case, the user needs to provide additional information, i.e. the **`<transformation>`** under **`<MultivariateNormal>`** of **`<Distributions>`** (more information can be found in Section 9.2). In addition, the node **`<variablesTransformation>`** is also required for **Stratified** sampler. This node is used to tranform the variables specified by **`<latentVariables>`** in the transformed space of input into variables spefified by **`<manifestVariables>`** in the input space. The variables listed in **`<latentVariables>`** should be predefined in **`<variable>`**, and the variables listed in **`<manifestVariables>`** are used by the **`<Models>`**. In addition, **`<globalGrid>`** will be not used for approach.

- **`<variablesTransformation>`**, ***optional field***. this XML node accepts one attribute:
  - **`distribution`**, ***required string attribute***, the name for the distribution defined in the XML node **`<Distributions>`**. This attribute indicates the values of **`<manifestVariables>`** are drawn from **`distribution`**.

In addition, this XML node also accepts three childen nodes:

- **<latentVariables>**, ***comma separated string, required field***, user-defined latent variables that are used for the variables transformation. All the variables listed under this node should be also mentioned in **<variable>**.
- **<manifestVariables>**, ***comma separated string, required field***, user-defined manifest variables that can be used by the **model**.
- **<manifestVariablesIndex>**, ***comma separated string, optional field***, user-defined manifest variables indices paired with **<manifestVariables>**. These indices indicate the position of manifest variables associated with multivariate normal distribution defined in the XML node **<Distributions>**. The indices should be postive integer. If not provided, the code will use the positions of manifest variables listed in **<manifestVariables>** as the indices.
- **<method>**, ***string, required field***, the method that is used for the variables transformation. The currently available method is '**pca**'.

```
...
<Models>
    ...
    <ExternalModel ModuleToLoad="lorentzAttractor_noK"
       name="PythonModule" subType="">
        <variables>sigma,rho,beta,x,y,z,time,z0,y0,z0</variables>
    </ExternalModel>
    ...
</Models>

<Distributions>
    ...
    <MultivariateNormal name='MVNDist' method='pca'>
        <transformation>
            <rank>3</rank>
        </transformation>
        <mu>0.0 1.0 2.0</mu>
        <covariance type="abs">
            1.0       0.6      -0.4
            0.6       1.0       0.2
            -0.4      0.2       0.8
        </covariance>
    </MultivariateNormal>
    ...
</Distributions>
```

```
<Samplers>
  ...
  <Stratified name='StratifiedName'>
        <variable name='x0'>
            <distribution dim='1'>MVNDist</distribution>
            <grid type='CDF' construction='equal' steps='3'>0.1
               0.9</grid>
        </variable>
        <variable name='y0'>
            <distribution dim='2'>MVNDist</distribution>
            <grid type='value' construction='equal'
               steps='3'>0.1 0.9</grid>
        </variable>
        <variable name='z0'>
            <distribution dim='3'>MVNDist</distribution>
            <grid type='CDF' construction='equal' steps='3'>0.2
               0.8</grid>
        </variable>
        <variablesTransformation model="PythonModule">
            <latentVariables>x1,y1,z1</latentVariables>
            <manifestVariables>x0,y0,z0</manifestVariables>
            <method>pca</method>
        </variablesTransformation>
  </Stratified>
  ...
</Samplers>
...
```

10.1.6 Response Surface Design

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

- **Box-Behnken**: This methodology aims to achieve the following goals:
  - Each factor, or independent variable, is placed at one of three equally spaced values, usually coded as -1, 0, +1. (At least three levels are needed for the following goal);

- The design should be sufficient to fit a quadratic model, that is, one squared term per factor and the products of any two factors;
- The ratio of the number of experimental points to the number of coefficients in the quadratic model should be reasonable (in fact, their designs keep it in the range of 1.5 to 2.6);
- The estimation variance should more or less depend only on the distance from the center (this is achieved exactly for the designs with 4 and 7 factors), and should not vary too much inside the smallest (hyper)cube containing the experimental points.

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

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

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

All the parameters, needed for setting up the algorithms reported above, must be defined within a **`<ResponseSurfaceDesign>`** block. This XML node accepts one attribute:

- **`name`**, ***required string attribute***, user-defined name of this sampler. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.

In the **`<ResponseSurfaceDesign>`** input block, the user needs to specify the variables to sample. As already mentioned, these variables are specified within consecutive **`<variable>`** XML blocks:

- **`<variable>`**, ***XML node, required parameter*** can specify the following attribute:
  - **`name`**, ***required string attribute***, user-defined name of this variable.

- **shape**, ***comma-separated integers, optional field***, determines the number of samples and shape of samples to be taken. For example, **shape**="2,3" will provide a 2 by 3 matrix of values, while **shape**="10" will produce a vector of 10 values. Omitting this optional attribute will result in a single scalar value instead. Each of the values in the matrix or vector will be the same as the single sampled value. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

This **<variable>** recognizes the following child nodes:

- **<distribution>**, ***string, required field***, name of the distribution that is associated to this variable. Its name needs to be contained in the **<Distributions>** block explained in Section 9. In addition, if NDDistribution is used, the attribute **dim** is required. **Note:** Alternatively, this node must be omitted if the **<function>** node is supplied.
- **<function>**, ***string, required field***, name of the function that defines the calculation of this variable from other distributed variables. Its name needs to be contained in the **<Functions>** block explained in Section 16. This function must implement a method named "evaluate". **Note:** Alternatively, this node must be ommitted if the **<distribution>** node is supplied.
- **<grid>**, ***space separated floats, required field***, the content of this XML node depends on the definition of the associated attributes:
  * **type**, ***required string attribute***, user-defined discretization metric type: 1) **'CDF'**, the grid will be specified based on cumulative distribution function probability thresholds, and 2) **'value'**, the grid will be provided using variable values.
  * **construction**, ***required string attribute***, how the grid needs to be constructed, independent of its type (i.e. **'CDF'** or **'value'**).

  Based on the **construction** type, the content of the **<grid>** XML node and the requirements for other attributes change. In this case, only the following is available:

  * **construction='custom'**. The grid will be directly specified by the user. This construction type requires that the **<grid>** node contains the actual mesh bins. For example, if the grid **type** is **'CDF'**, in the body of **<grid>**, the user will specify the CDF probability thresholds (nodalization in probability). All the bins are checked against the associated **<distribution>** bounds. If one or more of them falls outside the distribution's bounds, the code will raise an error. No additional attributes are needed.

  Note: The **<grid>** node is only required if a **<distribution>** node is supplied. In the case of a **<function>** node, no grid information is requested. **Note:** Only the construction "custom" is available. In the **<grid>** body only the lower and upper bounds can be inputted (2 numbers only).

- **<constant>**, ***XML node, optional parameter*** the user is able to input variables that need to be kept constant. For doing this, as many **<constant>** nodes as needed can be input.

There are options for setting a constant. To simply set the constant's value, the body of the node contains the constant value, and the **<constant>** node has the following attributes:

- **name**, ***required string attribute***, user-defined name of this constant.
- **shape**, ***comma-separated integers, optional field***, determines the shape of samples of the constant value. For example, **shape**="2,3" will shape the values into a 2 by 3 matrix, while **shape**="10" will shape into a vector of 10 values. Unlike the **<variable>**, the constant requires each value be entered; the number of required values is equal to the product of the **shape**. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

Alternatively, the constant value can be read from a DataObject that has been identified as a **<ConstantSource>** for this Sampler. In this case, the body of the **<constant>** node is the name of the variable that needs to be read from the **<ConstantSource>**, and the **<constant>** node has the following additional attributes in addition to the those above:

- **source**, ***required string attribute***, the name of the DataObject containing the value to be used for this constant. This must be the name of one of the **<ConstantSource>** DataObjects.
- **index**, ***optional integer attribute***, the index of the realization in the source DataObject that contains the value to use for the constant.
  Default: the last entry

By way of example, consider the following Sampler definition. The constant will be named **'C'** in the Sampler, and its value is taken from the DataObject **'MyConstant'**, which is identified in the **<ConstantSource>** node. To find the value of the constant in **'MyConstant'**, the Sampler will look at the realization with index **'3'** for the value of variable **'A'** to use as the constant value.

```
<Samplers>
  <WhatEverSampler name='whatever'>
    <ConstantSource class='DataObjects'
       type='PointSet'>MyConstants</ConstantSource>
    <constant name='C' source='MyConstants'
       index='3'>A</constant>
  </WhatEverSampler>
</Samplers>
```

- **<ResponseSurfaceDesignSettings>**, ***required***, In this sub-node, the user needs to specify different settings depending on the algorithm being used:
  - **<algorithmType>**, ***string, required field***, this XML node will contain the name of the algorithm to be used. Based on the chosen algorithm, other nodes need to be defined:

* **<algorithmType>**BoxBehnken**<algorithmType/>**. If Box-Behnken is specified, the following additional node is recognized:
  - **<ncenters>**, ***integer, optional field***, the number of center points to include in the box. If this parameter is not specified, then a pre-determined number of points are automatically included.
    Default: Automatic Generation.

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

    orthogonal for orthogonal design.

    rotatable for rotatable design.

    Default: orthogonal.
  - **<face>**, ***string, optional field***, in this node, the user defines how faces should be constructed. Three options are available:

    circumscribed for circumscribed facing

    inscribed for inscribed facing

    faced for faced facing.

    Default: circumscribed.

  Note: In order to employ the "Central Composite" design, at least 2 variables must be used.

Furthermore, if the covariance matrix is provided and the input parameters are assumed to have a multivariate normal distribution, one can use **ResponseSurfaceDesign** approach to sample the input parameters in the transformed space (aka subspace, reduced space). In this case, the user needs to provide additional information, i.e. the **<transformation>** under **<MultivariateNormal>** of **<Distributions>** (more information can be found in Section 9.2). In addition, the node **<variablesTransformation>** is also required for **ResponseSurfaceDesign** sampling. This node is used to tranform the variables specified by **<latentVariables>** in the transformed space of input into variables spefified by **<manifestVariables>** in the input space. The variables listed in **<latentVariables>** should be predefined in **<variable>**, and the variables listed in **<manifestVariables>** are used by the **<Models>**.

- **`<variablesTransformation>`**, ***optional field***. this XML node accepts one attribute:
  - **`distribution`**, ***required string attribute***, the name for the distribution defined in the XML node **`<Distributions>`**. This attribute indicates the values of **`<manifestVariables>`** are drawn from **`distribution`**.

  In addition, this XML node also accepts three childen nodes:
  - **`<latentVariables>`**, ***comma separated string, required field***, user-defined latent variables that are used for the variables transformation. All the variables listed under this node should be also mentioned in **`<variable>`**.
  - **`<manifestVariables>`**, ***comma separated string, required field***, user-defined manifest variables that can be used by the **`model`**.
  - **`<manifestVariablesIndex>`**, ***comma separated string, optional field***, user-defined manifest variables indices paired with **`<manifestVariables>`**. These indices indicate the position of manifest variables associated with multivariate normal distribution defined in the XML node **`<Distributions>`**. The indices should be postive integer. If not provided, the code will use the positions of manifest variables listed in **`<manifestVariables>`** as the indices.
  - **`<method>`**, ***string, required field***, the method that is used for the variables transformation. The currently available method is '**pca**'.

Example:

```
<Samplers>
  ...
    <ResponseSurfaceDesign name='BoxBehnkenRespDesign'>
        <ResponseSurfaceDesignSettings>
            <algorithmType>BoxBehnken</algorithmType>
            <ncenters>1</ncenters>
        </ResponseSurfaceDesignSettings>
        <variable name='var1' >
            <distribution >Gauss1</distribution>
           <grid type='CDF' construction='custom'  >0.2
              0.8</grid>
        </variable>
        <!-- N.B. at least 3 variables need to inputted
                in order to employ this algorithm
         -->
    </ResponseSurfaceDesign>
    <ResponseSurfaceDesign name='CentralCompositeRespDesign'>
        <ResponseSurfaceDesignSettings>
            <algorithmType>CentralComposite</algorithmType>
```

```
            <centers>1, 2</centers>
            <alpha>orthogonal</alpha>
            <face>circumscribed</face>
        </ResponseSurfaceDesignSettings>
        <variable name='var4' >
            <distribution >Gauss1</distribution>
            <grid type='CDF' construction='custom'  >0.2
               0.8</grid>
        </variable>
        <!-- N.B. at least 2 variables need to inputted
                in order to employ this algorithm
         -->
    </ResponseSurfaceDesign>
    <ResponseSurfaceDesign name='transformedSpaceSampling'>
        <ResponseSurfaceDesignSettings>
            <algorithmType>BoxBehnken</algorithmType>
            <ncenters>1</ncenters>
        </ResponseSurfaceDesignSettings>
        <variable name='var1' >
            <distribution >Gauss1</distribution>
           <grid type='CDF' construction='custom'  >0.2
              0.8</grid>
        </variable>
        ...
        <variablesTransformation model="givenModel">
          <latentVariables>var1,...</latentVariables>
          <manifestVariables>...</manifestVariables>
          <method>pca</method>
        </variablesTransformation>
    </ResponseSurfaceDesign>
  ...
</Samplers>
```

10.1.7 Factorial Design

The **Factorial Design** method is an important method to determine the effects of multiple variables on a response. A factorial design can reduce the number of samples one has to perform by studying multiple factors simultaneously. Additionally, it can be used to find both main effects (from each independent factor) and interaction effects (when both factors must be used to explain the outcome). A factorial design tests all possible conditions. Because factorial designs can lead to a large number of trials, which can become expensive and time-consuming, they are best used for

small numbers of variables with only a few domain discretizations (1 to 3). Factorial designs work well when interactions between variables are strong and important and where every variable contributes significantly. RAVEN currently employs three different algorithms that can be classified within this family of techniques:

- **General Full Factorial** explores the input space by investigating all possible combinations of a set of factors (variables).
- **2-Level Fractional-Factorial** consists of a carefully chosen subset (fraction) of the experimental runs of a full factorial design. The subset is chosen so as to exploit the sparsity-of-effects principle exposing information about the most important features of the problem studied, while using a fraction of the effort of a full factorial design in terms of experimental runs and resources.
- **Plackett-Burman** identifies the most important factors early in the experimentation phase when complete knowledge about the system is usually unavailable. It is an efficient screening method for identifying the active factors (variables) using as few samples as possible. In Plackett-Burman designs, main effects have a complicated confounding relationship with two-factor interactions. Therefore, these designs should be used to study main effects when it can be assumed that two-way interactions are negligible.

All the parameters needed for setting up the algorithms reported above must be defined within a **`<FactorialDesign>`** block. This XML node accepts one attribute:

- **`name`**, ***required string attribute***, user-defined name of this sampler. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.

In the **`<FactorialDesign>`** input block, the user needs to specify the variables to sample. As already mentioned, these variables are specified within consecutive **`<variable>`** XML blocks:

- **`<variable>`**, ***XML node, required parameter*** can specify the following attribute:
  - **`name`**, ***required string attribute***, user-defined name of this variable.
  - **`shape`**, ***comma-separated integers, optional field***, determines the number of samples and shape of samples to be taken. For example, **`shape`**="2,3" will provide a 2 by 3 matrix of values, while **`shape`**="10" will produce a vector of 10 values. Omitting this optional attribute will result in a single scalar value instead. Each of the values in the matrix or vector will be the same as the single sampled value. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

  This **`<variable>`** recognizes the following child nodes:

- **`<distribution>`**, ***string, required field***, name of the distribution that is associated to this variable. Its name needs to be contained in the **`<Distributions>`** block explained in Section 9. In addition, if NDDistribution is used, the attribute **`dim`** is required. **Note:** Alternatively, this node must be omitted if the **`<function>`** node is supplied.
- **`<function>`**, ***string, required field***, name of the function that defines the calculation of this variable from other distributed variables. Its name needs to be contained in the **`<Functions>`** block explained in Section 16. This function must implement a method named "evaluate". **Note:** Alternatively, this node must be ommitted if the **`<distribution>`** node is supplied.
- **`<grid>`**, ***space separated floats, required field***, the content of this XML node depends on the definition of the associated attributes:
  * **`type`**, ***required string attribute***, user-defined discretization metric type: 1) **`'CDF'`**, the grid will be specified based on cumulative distribution function probability thresholds, and 2) **`'value'`**, the grid will be provided using variable values.
  * **`construction`**, ***required string attribute***, how the grid needs to be constructed, independent of its type (i.e. **`'CDF'`** or **`'value'`**).

  Based on the **`construction`** type, the content of the **`<grid>`** XML node and the requirements for other attributes change:

  * **`construction='equal'`**. The grid is going to be constructed equally-spaced (**`type='value'`**) or equally probable (**`type='CDF'`**). This construction type requires the definition of additional attributes:
    · **`steps`**, ***required integer attribute***, number of equally spaced/probable discretization steps.

    This construction type requires that the content of the **`<grid>`** node represents the lower and upper bounds (either in probability or value). Two values need to be specified; the lowest one will be considered as the $lowerBound$, the largest, the $upperBound$. The lower and upper bounds are checked against the associated **`<distribution>`** bounds. If one or both of them falls outside the distribution's bounds, the code will raise an error. The $stepSize$ is determined as follows: $stepSize = (upperBound - lowerBound)/steps$
  * **`construction='custom'`**. The grid will be directly specified by the user. No additional attributes are needed. This construction type requires that the **`<grid>`** node contains the actual mesh bins. For example, if the grid **`type`** is **`'CDF'`**, in the body of **`<grid>`**, the user will specify the CDF probability thresholds (nodalization in probability). All the bins are checked against the associated **`<distribution>`** bounds. If one or more of them falls outside the distribution's bounds, the code will raise an error.

  Note: The **`<grid>`** node is only required if a **`<distribution>`** node is supplied. In the case of a **`<function>`** node, no grid information is requested.

- **<constant>**, ***XML node, optional parameter*** the user is able to input variables that need to be kept constant. For doing this, as many **<constant>** nodes as needed can be input. There are options for setting a constant. To simply set the constant's value, the body of the node contains the constant value, and the **<constant>** node has the following attributes:

  - **name**, ***required string attribute***, user-defined name of this constant.
  - **shape**, ***comma-separated integers, optional field***, determines the shape of samples of the constant value. For example, **shape**="2,3" will shape the values into a 2 by 3 matrix, while **shape**="10" will shape into a vector of 10 values. Unlike the **<variable>**, the constant requires each value be entered; the number of required values is equal to the product of the **shape**. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

  Alternatively, the constant value can be read from a DataObject that has been identified as a **<ConstantSource>** for this Sampler. In this case, the body of the **<constant>** node is the name of the variable that needs to be read from the **<ConstantSource>**, and the **<constant>** node has the following additional attributes in addition to the those above:

  - **source**, ***required string attribute***, the name of the DataObject containing the value to be used for this constant. This must be the name of one of the **<ConstantSource>** DataObjects.
  - **index**, ***optional integer attribute***, the index of the realization in the source DataObject that contains the value to use for the constant.
    Default: the last entry

  By way of example, consider the following Sampler definition. The constant will be named **'C'** in the Sampler, and its value is taken from the DataObject **'MyConstant'**, which is identified in the **<ConstantSource>** node. To find the value of the constant in **'MyConstant'**, the Sampler will look at the realization with index **'3'** for the value of variable **'A'** to use as the constant value.

```
<Samplers>
  <WhatEverSampler name='whatever'>
    <ConstantSource class='DataObjects'
       type='PointSet'>MyConstants</ConstantSource>
    <constant name='C' source='MyConstants'
       index='3'>A</constant>
  </WhatEverSampler>
</Samplers>
```

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

- **<algorithmType>**, ***string, required field***, specifies the algorithm to be used. Based on the chosen algorithm, other nodes may be defined:
  - **<algorithmType>**`full`**<algorithmType/>**. Full factorial design. If `full` is specified, no additional nodes are necessary.
    Note: The full factorial design does not have any limitations on the number of discretization bins that can be used in the **<grid>** XML node for each **<variable>** specified.
  - **<algorithmType>**`2levelFract`**<algorithmType/>**. Two-level Fractional-Factorial design. If
    `2levelFract` is specified, the following additional nodes must be specified:
    * **<gen>**, ***space separated strings, required field***, specifies the confounding mapping. For instance, in this block the user defines the decisions on a fraction of the full-factorial by allowing some of the factor main effects to be compounded with other factor interaction effects. This is done by defining an alias structure that defines, symbolically, these interactions. These alias structures are written like "C = AB" or "I = ABC", or "AB = CD", etc. These define how a column is related to the others.
    * **<genMap>**, ***space separated strings, required field***, defines the mapping between the **<gen>** symbolic aliases and the variables that have been inputted in the **<FactorialDesign>** main block.

    Note: The Two-levels Fractional-Factorial design is limited to 2 discretization bins in the **<grid>** node for each **<variable>**.
  - **<algorithmType>**`pb`**<algorithmType/>**. Plackett-Burman design. If `pb` is specified, no additional nodes are necessary.
    Note: The Plackett-Burman design does not have any limitations on the number of discretization bins allowed in the **<grid>** node for each **<variable>**.

Example:

```
<Samplers>
  ...
  <FactorialDesign name='fullFactorial'>
    <FactorialSettings>
      <algorithmType>full</algorithmType>
    </FactorialSettings>
    <variable name='var1' >
      <distribution>aDistributionNameDefinedInDistributionBlock1
      </distribution>
      <grid type='value' construction='custom' >0.02 0.03
         0.5</grid>
    </variable>
```

```
    <variable name='var2' >
      <distribution>aDistributionNameDefinedInDistributionBlock2
      </distribution>
      <grid type='CDF' construction='custom'>0.5 0.7 1.0</grid>
    </variable>
  </FactorialDesign>
  <FactorialDesign name='2levelFractFactorial'>
    <FactorialSettings>
      <algorithmType>2levelFract</algorithmType>
      <gen>a,b,ab</gen>
      <genMap>var1,var2,var3</genMap>
    </FactorialSettings>
    <variable name='var1' >
      <distribution>aDistributionNameDefinedInDistributionBlock3
      </distribution>
      <grid type='value' construction='custom' >0.02 0.5</grid>
    </variable>
    <variable name='var2' >
      <distribution>aDistributionNameDefinedInDistributionBlock
      </distribution>
      <grid type='CDF' construction='custom'>0.5 1.0</grid>
    </variable>
    <variable name='var3'>
      <distribution>aDistributionNameDefinedInDistributionBlock5
      </distribution>
      <grid type='value' upperBound='4' construction='equal'
         steps='1'>0.5</grid>
    </variable>
  </FactorialDesign>
  <FactorialDesign name='pbFactorial'>
    <FactorialSettings>
      <algorithmType>pb</algorithmType>
    </FactorialSettings>
    <variable name='var1' >
      <distribution>aDistributionNameDefinedInDistributionBlock6
      </distribution>
      <grid type='value' construction='custom' >0.02 0.5</grid>
    </variable>
    <variable name='VarGauss2' >
      <distribution>aDistributionNameDefinedInDistributionBlock7
      </distribution>
      <grid type='CDF' construction='custom'>0.5 1.0</grid>
```

```
    </variable>
  </FactorialDesign>
  ...
</Samplers>
```

10.1.8 Ensemble Forward Sampling strategy

The **Ensemble Forward** sampling approach allows the user to combine multiple Forward sampling strategies into one single strategy. For example, it can happen that a variable is more suitable for a particular sampling strategy (e.g. a stochastic event modeled with a Monte Carlo approach) and a second variable is more suitable for another sampling method (e.g. because part of a parametric space modeled with a Grid-based approach). The specifications of this sampler must be defined within a **`<EnsembleForward>`** XML block. This XML node accepts the following attributes:

- **name**, *required string attribute*, user-defined name of this Sampler. N.B. As for the other objects, this is the name that can be used to refer to this specific entity from other input blocks (xml);

In the **EnsembleForward** input block, the user needs to specify the sampling strategies that he wants to combine together.
Currently, only the following strategies can be combined:

- **`<MonteCarlo>`**
- **`<Grid>`**
- **`<Stratiefied>`**
- **`<FactorialDesign>`**
- **`<ResponseSurfaceDesign>`**
- **`<CustomSampler>`**

For each of the above samplers, the input specifications can be found in the relative sections.

Example:

```
<Samplers>
  ...
    <EnsembleForward name="testEnsembleForward">
        <MonteCarlo name = "theMC">
            <samplerInit> <limit>4</limit> </samplerInit>
            <variable name="sigma">
```

```
            <distribution>norm</distribution>
        </variable>
    </MonteCarlo>
    <Grid name = "theGrid">
        <variable name="x0">
            <distribution>unif</distribution>
            <grid construction="custom" type="value">0.02
               0.5 0.6</grid>
        </variable>
    </Grid>
    <Stratified name = "theStratified">
        <variable name="z0">
            <distribution>tri</distribution>
            <grid construction="equal" steps="2"
               type="CDF">0.2 0.8</grid>
        </variable>
        <variable name="y0">
            <distribution>unif</distribution>
            <grid construction="equal" steps="2"
               type="value">0.5 0.8</grid>
        </variable>
    </Stratified>
    <ResponseSurfaceDesign name = "theRSD">
        <ResponseSurfaceDesignSettings>
            <algorithmType>CentralComposite</algorithmType>
            <centers>1,2</centers>
            <alpha>orthogonal</alpha>
            <face>circumscribed</face>
        </ResponseSurfaceDesignSettings>
        <variable name="rho">
            <distribution>unif</distribution>
            <grid construction="custom" type="CDF">0.0
               1.0</grid>
        </variable>
        <variable name="beta">
            <distribution>tri</distribution>
            <grid construction="custom" type="value">0.1
               1.5</grid>
        </variable>
    </ResponseSurfaceDesign>
  </EnsembleForward>
...
```

```
</Samplers>
```

Care should be used when using deterministic random seeds for EnsembleForward sampling. The EnsembleForward sample will ignore any seeds set in any of its subset samplers; however, the global random seed can be set by adding a **<samplerInit>** block with the **<initialSeed>** block therein, with an integer value providing the seed. For example,

```
<Samplers>
  ...
  <EnsembleForward name='testEnsembleForward'>
    <samplerInit>
      <initialSeed>42</initialSeed>
    </samplerInit>
  ...
   </EnsembleForward>
  ...
</Samplers>
```

Because RAVEN has a single global random number generator, this will set the seed for the full calculation when the Step containing a run using this ForwardSampler is begun.

Note also variables that are defined from functions, as well as constants, need to be defined outside the samplers of the ensemble sampler. An example is shown below.

Example:

```
<Samplers>
  <EnsembleForward name='testEnsembleForward'>
    <variable name='x3'>
        <function>funct1</function>
    </variable>
    <variable name='x4,x5'>
        <function>funct2</function>
    </variable>
    <constant name='pi'>3.14159</constant>
    <MonteCarlo name='notNeeded'>
      <samplerInit>
        <limit>3</limit>
      </samplerInit>
      <variable name='x1'>
        <distribution>norm</distribution>
      </variable>
    </MonteCarlo>
    <Grid name='notNeeded'>
```

```
        <variable name='x2'>
          <distribution>unif</distribution>
          <grid construction='custom' type='value'>0.02
             0.6</grid>
        </variable>
      </Grid>
    </EnsembleForward>
  </Samplers>
```

In this example note that:

- variables $x1$ and $x2$ are generated by the two samplers (Monte-Carlo and Grid respectively)
- variable $x3$ is generated from the function $funct1$
- variables $x4$ and $x5$ are generated from the function $funct2$
- variables $x3$, $x4$ and $x5$ are defined outside the Monte-Carlo and Grid

10.1.9 Custom Sampling strategy

The **Custom** sampling approach allows the user to specify a predefined set of coordinates (in the input space) that RAVEN should use to inquire the model. For example, the user can provide a CSV file containing a list of samples that RAVEN should use. The specifications of this sampler must be defined within a **`<CustomSampler>`** XML block. This XML node accepts the following attributes:

- **name**, *required string attribute*, user-defined name of this Sampler. N.B. As for the other objects, this is the name that can be used to refer to this specific entity from other input blocks (xml);

In the **CustomSampler** input block, the user needs to specify the variables need to be sampled. As already mentioned, these variables are inputted within consecutive XML blocks called **`<variable>`**. Note that if any variables are dependent on other dimensions (e.g. "time"), the dependent dimensions need to be listed as variables as well.

In addition, the **`<Source>`** from which the samples need to be retrieved needs to be specified:

- **`<variable>`**, ***XML node, required parameter*** can specify the following attribute:
  - **`name`**, ***required string attribute***, user-defined name of this variable.
  - **`nameInSource`**, ***optional string attribute***, name of the variable to read from in **`<Source>`**.

    Default: Same as **`name`**.

- `shape`, ***comma-separated integers, optional field***, determines the number of samples and shape of samples to be taken. For example, `shape`="2,3" will provide a 2 by 3 matrix of values, while `shape`="10" will produce a vector of 10 values. Omitting this optional attribute will result in a single scalar value instead. Each of the values in the matrix or vector will be the same as the single sampled value. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

- `<Source>`, ***XML node, required parameter*** will specify the following attributes:
  - `class`, ***required string attribute***, class entity of the source where the samples need to be retrieved from. It can be either **Files** or **DataObjects**.
  - `type`, ***required string attribute***, type of the source withing the previously explained "class". If `class` is **Files**, this attribute needs to be kept empty; otherwise it must be one of the **DataSet** objects: PointSet, HistorySet, or DataSet.
    Note: If the `<Source>` `class` is **Files**, the File needs to be a standard CSV file, specified in the `<Files>` XML block in the RAVEN input.
    In addition, it is important to notice that if in the `<Source>` the **PointProbability** and **ProbabilityWeight** quantities are not found, the samples are assumed to come from a MonteCarlo (from a statistical post-processing prospective).

- `<index>`, ***comma-separated integer, optional parameter*** indexes to use from the `<Source>`. If provided, then only the listed indexes will be used. Indexes are zero-based; that is, the first realization is indexed at 0, the second at 1, and so forth. Default is for all indices in the source to be used.

- `<constant>`, ***XML node, optional parameter*** the user is able to input variables that need to be kept constant. For doing this, as many `<constant>` nodes as needed can be input. There are options for setting a constant. To simply set the constant's value, the body of the node contains the constant value, and the `<constant>` node has the following attributes:
  - `name`, ***required string attribute***, user-defined name of this constant.
  - `shape`, ***comma-separated integers, optional field***, determines the shape of samples of the constant value. For example, `shape`="2,3" will shape the values into a 2 by 3 matrix, while `shape`="10" will shape into a vector of 10 values. Unlike the `<variable>`, the constant requires each value be entered; the number of required values is equal to the product of the `shape`. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

  Alternatively, the constant value can be read from a DataObject that has been identified as a `<ConstantSource>` for this Sampler. In this case, the body of the `<constant>` node is the name of the variable that needs to be read from the `<ConstantSource>`, and the `<constant>` node has the following additional attributes in addition to the those above:

- `source`, ***required string attribute***, the name of the DataObject containing the value to be used for this constant. This must be the name of one of the `<ConstantSource>` DataObjects.
- `index`, ***optional integer attribute***, the index of the realization in the source DataObject that contains the value to use for the constant.
  Default: the last entry

By way of example, consider the following Sampler definition. The constant will be named `'C'` in the Sampler, and its value is taken from the DataObject `'MyConstant'`, which is identified in the `<ConstantSource>` node. To find the value of the constant in `'MyConstant'`, the Sampler will look at the realization with index `'3'` for the value of variable `'A'` to use as the constant value.

```
<Samplers>
  <WhatEverSampler name='whatever'>
    <ConstantSource class='DataObjects'
       type='PointSet'>MyConstants</ConstantSource>
    <constant name='C' source='MyConstants'
       index='3'>A</constant>
  </WhatEverSampler>
</Samplers>
```

Example:

Table 1: samples.csv

y	x	z	PointProbability	ProbabilityWeight
0.725675246	0.031099304	0.984988317	0.1	0.2
0.565949127	0.028589754	1.13186372	0.1	0.2
0.72567754	0.031099304	0.967209238	0.1	0.2
0.565951633	0.028589754	1.111431662	0.1	0.2
0.725968307	0.031100307	0.98498835	0.1	0.2

```
<Samplers>
  ...
  <Samplers>
    <CustomSampler name="customSamplerDataObject">
      <Source   class="DataObjects"
         type="PointSet">outCustomSamplerFromFile</Source>
      <variable name="x"/>
      <variable name="y"/>
      <variable name="z"/>
```

```
    </CustomSampler>
  </Samplers>
  <Samplers>
    <CustomSampler name="customSamplerFile">
      <Source   class="Files"  type="">samples.csv</Source>
      <variable name="x"/>
      <variable name="y"/>
      <variable name="z"/>
    </CustomSampler>
  </Samplers>
  ...
</Samplers>
```

10.2 Dynamic Event Tree (DET) Samplers

The **Dynamic Event Tree** methodologies are designed to take the timing of events explicitly into account, which can become very important especially when uncertainties in complex phenomena are considered. Hence, the main idea of this methodology is to let a system code determine the pathway of an accident scenario within a probabilistic environment. In this family of methods, a continuous monitoring of the system evolution in the phase space is needed. In order to use the DET-based methods, the generic driven code needs to have, at least, an internal trigger system and, consequently, a "restart" capability. In the RAVEN framework, 4 different DET samplers are available:

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

The ADET and the AHDET methodologies represent a hybrid between the DET/HDET and adaptive sampling approaches. For this reason, its input requirements are reported in the Adaptive Samplers' section (10.3).

10.2.1 Dynamic Event Tree

The **Dynamic Event Tree** sampling approach is a sampling strategy that is designed to take the timing of events, in transient/accident scenarios, explicitly into account. From an application point

of view, an N-Dimensional grid is built on the CDF space. A single simulation is spawned and a set of triggers is added to the system code control logic. Every time a trigger is activated (one of the CDF thresholds in the grid is exceeded), a new set of simulations (branches) is spawned. Each branch carries its conditional probability. In the RAVEN code, the triggers are defined by specifying a grid using a predefined discretization metric in the mode input space. RAVEN provides two discretization metrics: 1) CDF, and 2) value. Thus, the trigger thresholds can be entered either in probability or value space.

The specifications of this sampler must be defined within a **`<DynamicEventTree>`** XML block. This XML node accepts the following attributes:

- **`name`**, ***required string attribute***, user-defined name of this sampler. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.
- **`printEndXmlSummary`**, ***optional string/boolean attribute***, controls the dumping of a "summary" of the DET performed into an external XML.
  Default: False.
- **`maxSimulationTime`**, ***optional float attribute***, this attribute controls the maximum "mission" time of the simulation underneath.
  Default: None.

In the **`<DynamicEventTree>`** input block, the user needs to specify the variables to sample. As already mentioned, these variables are specified within consecutive **`<variable>`** XML blocks:

- **`<variable>`**, ***XML node, required parameter*** can specify the following attribute:
  - **`name`**, ***required string attribute***, user-defined name of this variable.
  - **`shape`**, ***comma-separated integers, optional field***, determines the number of samples and shape of samples to be taken. For example, **`shape`**="2,3" will provide a 2 by 3 matrix of values, while **`shape`**="10" will produce a vector of 10 values. Omitting this optional attribute will result in a single scalar value instead. Each of the values in the matrix or vector will be the same as the single sampled value. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

  This **`<variable>`** recognizes the following child nodes:

  - **`<distribution>`**, ***string, required field***, name of the distribution that is associated to this variable. Its name needs to be contained in the **`<Distributions>`** block explained in Section 9. In addition, if NDDistribution is used, the attribute **`dim`** is required. **Note:** Alternatively, this node must be omitted if the **`<function>`** node is supplied.

- **`<function>`**, ***string, required field***, name of the function that defines the calculation of this variable from other distributed variables. Its name needs to be contained in the **`<Functions>`** block explained in Section 16. This function must implement a method named "evaluate". **Note:** Alternatively, this node must be ommitted if the **`<distribution>`** node is supplied.
- **`<grid>`**, ***space separated floats, required field***, the content of this XML node depends on the definition of the associated attributes:
  * **`type`**, ***required string attribute***, user-defined discretization metric type: 1) **`'CDF'`**, the grid will be specified based on cumulative distribution function probability thresholds, and 2) **`'value'`**, the grid will be provided using variable values.
  * **`construction`**, ***required string attribute***, how the grid needs to be constructed, independent of its type (i.e. **`'CDF'`** or **`'value'`**).

  Based on the **`construction`** type, the content of the **`<grid>`** XML node and the requirements for other attributes change:

  * **`construction='equal'`**. The grid is going to be constructed equally-spaced (**`type='value'`**) or equally probable (**`type='CDF'`**). This construction type requires the definition of additional attributes:
    · **`steps`**, ***required integer attribute***, number of equally spaced/probable discretization steps.

    This construction type requires that the content of the **`<grid>`** node represents the lower and upper bounds (either in probability or value). Two values need to be specified; the lowest one will be considered as the $lowerBound$, the largest, the $upperBound$. The lower and upper bounds are checked against the associated **`<distribution>`** bounds. If one or both of them falls outside the distribution's bounds, the code will raise an error. The $stepSize$ is determined as follows: $stepSize = (upperBound - lowerBound)/steps$
  * **`construction='custom'`**. The grid will be directly specified by the user. No additional attributes are needed. This construction type requires that the **`<grid>`** node contains the actual mesh bins. For example, if the grid **`type`** is **`'CDF'`**, in the body of **`<grid>`**, the user will specify the CDF probability thresholds (nodalization in probability). All the bins are checked against the associated **`<distribution>`** bounds. If one or more of them falls outside the distribution's bounds, the code will raise an error.

  Note: The **`<grid>`** node is only required if a **`<distribution>`** node is supplied. In the case of a **`<function>`** node, no grid information is requested.

- **`<constant>`**, ***XML node, optional parameter*** the user is able to input variables that need to be kept constant. For doing this, as many **`<constant>`** nodes as needed can be input. There are options for setting a constant. To simply set the constant's value, the body of the node contains the constant value, and the **`<constant>`** node has the following attributes:
  - **`name`**, ***required string attribute***, user-defined name of this constant.

- **shape**, ***comma-separated integers, optional field***, determines the shape of samples of the constant value. For example, **shape**="2,3" will shape the values into a 2 by 3 matrix, while **shape**="10" will shape into a vector of 10 values. Unlike the **<variable>**, the constant requires each value be entered; the number of required values is equal to the product of the **shape**. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

Alternatively, the constant value can be read from a DataObject that has been identified as a **<ConstantSource>** for this Sampler. In this case, the body of the **<constant>** node is the name of the variable that needs to be read from the **<ConstantSource>**, and the **<constant>** node has the following additional attributes in addition to the those above:

- **source**, ***required string attribute***, the name of the DataObject containing the value to be used for this constant. This must be the name of one of the **<ConstantSource>** DataObjects.
- **index**, ***optional integer attribute***, the index of the realization in the source DataObject that contains the value to use for the constant.
  Default: the last entry

By way of example, consider the following Sampler definition. The constant will be named **'C'** in the Sampler, and its value is taken from the DataObject **'MyConstant'**, which is identified in the **<ConstantSource>** node. To find the value of the constant in **'MyConstant'**, the Sampler will look at the realization with index **'3'** for the value of variable **'A'** to use as the constant value.

```
<Samplers>
  <WhatEverSampler name='whatever'>
    <ConstantSource class='DataObjects'
       type='PointSet'>MyConstants</ConstantSource>
    <constant name='C' source='MyConstants'
       index='3'>A</constant>
  </WhatEverSampler>
</Samplers>
```

Example:

```
<Samplers>
  ...
  <DynamicEventTree name='DETname'>
    <variable name='var1'>
      <distribution>aDistributionNameDefinedInDistributionBlock1
         </distribution>
      <grid type='value' construction='equal' steps='100' >1.0
         201.0</grid>
```

```
    </variable>
    <variable name='var2'>
      <distribution>aDistributionNameDefinedInDistributionBlock2
         </distribution>
      <grid type='CDF' construction='equal' steps='5'>0 1</grid>
    </variable>
    <variable name='var3'>
      <distribution>aDistributionNameDefinedInDistributionBlock3
         </distribution>
      <grid type='value' construction='equal' steps='10' >11.0
         21.0</grid>
    </variable>
    <variable name='var4'>
      <distribution>aDistributionNameDefinedInDistributionBlock4
         </distribution>
      <grid type='CDF' construction='equal' steps='5' >0.0
         1.0</grid>
    </variable>
    <variable name='var5'>
      <distribution>aDistributionNameDefinedInDistributionBlock5
         </distribution>
      <grid type='value' construction='custom'>0.2 0.5
         10.0</grid>
    </variable>
    <variable name='var6'>
      <distribution>aDistributionNameDefinedInDistributionBlock6
         </distribution>
      <grid type='CDF' construction='custom'>0.2 0.5 1.0</grid>
    </variable>
  </DynamicEventTree>
  ...
</Samplers>
```

10.2.2 Hybrid Dynamic Event Tree

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

uncertainties can be sampled through the following strategies:

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

From a practical point of view, the user defines the parameters that need to be sampled by one or more different approaches. The HDET module samples those parameters creating an N-dimensional grid characterized by all the possible combinations of the input space coordinates coming from the different sampling strategies. Each coordinate in the input space represents a separate and parallel standard DET exploration of the uncertain domain. The HDET methodology allows the user to explore the uncertain domain employing the best approach for each variable kind. The addition of a grid sampling strategy among the usable approaches allows the user to perform a discrete parametric study under aleatory and epistemic uncertainties.

Regarding the input requirements, the HDET sampler is a "sub-type" of the **`<DynamicEventTree>`** sampler. For this reason, its specifications must be defined within a **`<DynamicEventTree>`** block. This XML node accepts the following attributes:

- **`name`**, ***required string attribute***, user-defined name of this sampler. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.
- **`printEndXmlSummary`**, ***optional string/boolean attribute***, controls the dumping of a "summary" of the DET performed into an external XML.
  Default: False.
- **`maxSimulationTime`**, ***optional float attribute***, this attribute controls the maximum "mission" time of the simulation underneath.
  Default: None.

In the **`<DynamicEventTree>`** input block, the user needs to specify the variables to sample. As already mentioned, these variables are specified within consecutive **`<variable>`** XML blocks:

- **`<variable>`**, ***XML node, required parameter*** can specify the following attribute:
  - **`name`**, ***required string attribute***, user-defined name of this variable.

- **shape**, ***comma-separated integers, optional field***, determines the number of samples and shape of samples to be taken. For example, **shape**="2,3" will provide a 2 by 3 matrix of values, while **shape**="10" will produce a vector of 10 values. Omitting this optional attribute will result in a single scalar value instead. Each of the values in the matrix or vector will be the same as the single sampled value. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

This **<variable>** recognizes the following child nodes:

- **<distribution>**, ***string, required field***, name of the distribution that is associated to this variable. Its name needs to be contained in the **<Distributions>** block explained in Section 9. In addition, if NDDistribution is used, the attribute **dim** is required. **Note:** Alternatively, this node must be omitted if the **<function>** node is supplied.
- **<function>**, ***string, required field***, name of the function that defines the calculation of this variable from other distributed variables. Its name needs to be contained in the **<Functions>** block explained in Section 16. This function must implement a method named "evaluate". **Note:** Alternatively, this node must be ommitted if the **<distribution>** node is supplied.
- **<grid>**, ***space separated floats, required field***, the content of this XML node depends on the definition of the associated attributes:
  * **type**, ***required string attribute***, user-defined discretization metric type: 1) **'CDF'**, the grid will be specified based on cumulative distribution function probability thresholds, and 2) **'value'**, the grid will be provided using variable values.
  * **construction**, ***required string attribute***, how the grid needs to be constructed, independent of its type (i.e. **'CDF'** or **'value'**).

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

  * **construction='equal'**. The grid is going to be constructed equally-spaced (**type='value'**) or equally probable (**type='CDF'**). This construction type requires the definition of additional attributes:
    · **steps**, ***required integer attribute***, number of equally spaced/probable discretization steps.

    This construction type requires that the content of the **<grid>** node represents the lower and upper bounds (either in probability or value). Two values need to be specified; the lowest one will be considered as the $lowerBound$, the largest, the $upperBound$. The lower and upper bounds are checked against the associated **<distribution>** bounds. If one or both of them falls outside the distribution's bounds, the code will raise an error. The $stepSize$ is determined as follows: $stepSize = (upperBound - lowerBound)/steps$
  * **construction='custom'**. The grid will be directly specified by the user. No additional attributes are needed. This construction type requires that the **<grid>**

node contains the actual mesh bins. For example, if the grid `type` is `'CDF'`, in the body of `<grid>`, the user will specify the CDF probability thresholds (nodalization in probability). All the bins are checked against the associated `<distribution>` bounds. If one or more of them falls outside the distribution's bounds, the code will raise an error.

Note: The `<grid>` node is only required if a `<distribution>` node is supplied. In the case of a `<function>` node, no grid information is requested.

- `<constant>`, ***XML node, optional parameter*** the user is able to input variables that need to be kept constant. For doing this, as many `<constant>` nodes as needed can be input. There are options for setting a constant. To simply set the constant's value, the body of the node contains the constant value, and the `<constant>` node has the following attributes:
  - `name`, ***required string attribute***, user-defined name of this constant.
  - `shape`, ***comma-separated integers, optional field***, determines the shape of samples of the constant value. For example, `shape`="2,3" will shape the values into a 2 by 3 matrix, while `shape`="10" will shape into a vector of 10 values. Unlike the `<variable>`, the constant requires each value be entered; the number of required values is equal to the product of the `shape`. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

  Alternatively, the constant value can be read from a DataObject that has been identified as a `<ConstantSource>` for this Sampler. In this case, the body of the `<constant>` node is the name of the variable that needs to be read from the `<ConstantSource>`, and the `<constant>` node has the following additional attributes in addition to the those above:

  - `source`, ***required string attribute***, the name of the DataObject containing the value to be used for this constant. This must be the name of one of the `<ConstantSource>` DataObjects.
  - `index`, ***optional integer attribute***, the index of the realization in the source DataObject that contains the value to use for the constant.
    Default: the last entry

  By way of example, consider the following Sampler definition. The constant will be named `'C'` in the Sampler, and its value is taken from the DataObject `'MyConstant'`, which is identified in the `<ConstantSource>` node. To find the value of the constant in `'MyConstant'`, the Sampler will look at the realization with index `'3'` for the value of variable `'A'` to use as the constant value.

```
    <Samplers>
      <WhatEverSampler name='whatever'>
        <ConstantSource class='DataObjects'
           type='PointSet'>MyConstants</ConstantSource>
```

```
        <constant name='C' source='MyConstants'
           index='3'>A</constant>
      </WhatEverSampler>
    </Samplers>
```

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

- **type**, ***required string attribute***, type of pre-sampling strategy to be used. Available options are **'MonteCarlo'**, **'Grid'**, and **'Stratified'**.

Independent of the type of "pre-sampler" that has been specified, the **<HybridSampler>** must contain the variables that need to be sampled. As already mentioned, these variables are specified within consecutive **<variable>** XML blocks:

- **<variable>**, ***XML node, required parameter*** can specify the following attribute:
  - **name**, ***required string attribute***, user-defined name of this variable.
  - **shape**, ***comma-separated integers, optional field***, determines the number of samples and shape of samples to be taken. For example, **shape**="2,3" will provide a 2 by 3 matrix of values, while **shape**="10" will produce a vector of 10 values. Omitting this optional attribute will result in a single scalar value instead. Each of the values in the matrix or vector will be the same as the single sampled value. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

  This **<variable>** recognizes the following child nodes:

  - **<distribution>**, ***string, required field***, name of the distribution that is associated to this variable. Its name needs to be contained in the **<Distributions>** block explained in Section 9. In addition, if NDDistribution is used, the attribute **dim** is required. **Note:** Alternatively, this node must be omitted if the **<function>** node is supplied.
  - **<function>**, ***string, required field***, name of the function that defines the calculation of this variable from other distributed variables. Its name needs to be contained in the **<Functions>** block explained in Section 16. This function must implement a method named "evaluate". **Note:** Alternatively, this node must be ommitted if the **<distribution>** node is supplied.

- **<constant>**, ***XML node, optional parameter*** the user is able to input variables that need to be kept constant. For doing this, as many **<constant>** nodes as needed can be input. There are options for setting a constant. To simply set the constant's value, the body of the node contains the constant value, and the **<constant>** node has the following attributes:

  - **name**, ***required string attribute***, user-defined name of this constant.
  - **shape**, ***comma-separated integers, optional field***, determines the shape of samples of the constant value. For example, **shape**="2,3" will shape the values into a 2 by 3 matrix, while **shape**="10" will shape into a vector of 10 values. Unlike the **<variable>**, the constant requires each value be entered; the number of required values is equal to the product of the **shape**. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

  Alternatively, the constant value can be read from a DataObject that has been identified as a **<ConstantSource>** for this Sampler. In this case, the body of the **<constant>** node is the name of the variable that needs to be read from the **<ConstantSource>**, and the **<constant>** node has the following additional attributes in addition to the those above:

  - **source**, ***required string attribute***, the name of the DataObject containing the value to be used for this constant. This must be the name of one of the **<ConstantSource>** DataObjects.
  - **index**, ***optional integer attribute***, the index of the realization in the source DataObject that contains the value to use for the constant.
    Default: the last entry

  By way of example, consider the following Sampler definition. The constant will be named **'C'** in the Sampler, and its value is taken from the DataObject **'MyConstant'**, which is identified in the **<ConstantSource>** node. To find the value of the constant in **'MyConstant'**, the Sampler will look at the realization with index **'3'** for the value of variable **'A'** to use as the constant value.

```
<Samplers>
  <WhatEverSampler name='whatever'>
    <ConstantSource class='DataObjects'
       type='PointSet'>MyConstants</ConstantSource>
    <constant name='C' source='MyConstants'
       index='3'>A</constant>
  </WhatEverSampler>
</Samplers>
```

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

- **type**, ***required string attribute***, user-defined discretization metric type:
  - **'CDF'**, the grid is going to be specified based on the cumulative distribution function probability thresholds
  - **'value'**, the grid is going to be provided using variable values.
- **construction**, ***required string attribute***, how the grid needs to be constructed, independent of its type (i.e. **'CDF'** or **'value'**).

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

- **construction='equal'**. The grid is going to be constructed equally-spaced (**type='value'**) or equally probable (**type='CDF'**). This construction type requires the definition of additional attributes:
  - **steps**, ***required integer attribute***, number of equally spaced/probable discretization steps.

  This construction type requires that the content of the **<grid>** node represents the lower and upper bounds (either in probability or value). Two values need to be specified; the lowest one will be considered as the $lowerBound$, the largest, the $upperBound$. The lower and upper bounds are checked against the associated **<distribution>** bounds. If one or both of them falls outside the distribution's bounds, the code will raise an error. The $stepSize$ is determined as follows:
  $stepSize = (upperBound - lowerBound)/steps$
- **construction='custom'**. The grid will be directly specified by the user. No additional attributes are needed. This construction type requires that the **<grid>** node contains the actual mesh bins. For example, if the grid **type** is **'CDF'**, in the body of **<grid>**, the user will specify the CDF probability thresholds (nodalization in probability). All the bins are checked against the associated **<distribution>** bounds. If one or more of them falls outside the distribution's bounds, the code will raise an error.

Example:

```
<Samplers>
  ...
  <DynamicEventTree name='HybridDETname' print_end_XML="True">
    <HybridSampler type='MonteCarlo' limit='2'>
      <variable name='var1' >
        <distribution>aDistributionNameDefinedInDistributionBlock1
          </distribution>
```

```
    </variable>
    <variable name='var2' >
      <distribution>aDistributionNameDefinedInDistributionBlock2
         </distribution>
      <grid type='CDF' construction='equal' steps='1'
         lowerBound='0.1'>0.1</grid>
    </variable>
  </HybridSampler>
  <HybridSampler type='Grid'>
    <!-- Point sampler way (directly sampling the variable) -->
    <variable name='var3' >
      <distribution>aDistributionNameDefinedInDistributionBlock3
         </distribution>
      <grid type='CDF' construction='equal' steps='1'
         lowerBound='0.1'>0.1</grid>
    </variable>
    <variable name='var4' >
      <distribution>aDistributionNameDefinedInDistributionBlock4
         </distribution>
      <grid type='CDF' construction='equal' steps='1'
         lowerBound='0.1'>0.1</grid>
    </variable>
  </HybridSampler>
  <HybridSampler type='Stratified'>
    <!-- Point sampler way (directly sampling the variable )
       -->
    <variable name='var5' >
      <distribution>aDistributionNameDefinedInDistributionBlock5
         </distribution>
      <grid type='CDF' construction='equal' steps='1'
         lowerBound='0.1'>0.1</grid>
    </variable>
    <variable name='var6' >
      <distribution>aDistributionNameDefinedInDistributionBlock6
         </distribution>
      <grid type='CDF' construction='equal' steps='1'
         lowerBound='0.1'>0.1</grid>
    </variable>
  </HybridSampler>
  <!-- DYNAMIC EVENT TREE INPUT (it goes outside an inner
     block like HybridSamplerSettings) -->
    <Distribution name='dist7'>
```

```
        <distribution>aDistributionNameDefinedInDistributionBlock7
           </distribution>
        <grid type='CDF' construction='custom'>0.1 0.8</grid>
      </Distribution>
  </DynamicEventTree>
  ...
</Samplers>
```

10.3 Adaptive Samplers

The Adaptive Samplers' family provides the possibility to perform smart sampling (also known as adaptive sampling) as an alternative to classical "Forward" techniques. The motivation is that system simulations are often computationally expensive, time-consuming, and high dimensional with respect to the number of input parameters. Thus, exploring the space of all possible simulation outcomes is infeasible using finite computing resources. During simulation-based probabilistic risk analysis, it is important to discover the relationship between a potentially large number of input parameters and the output of a simulation using as few simulation trials as possible.

The description above characterizes a typical context for performing adaptive sampling where a few observations are obtained from the simulation, a reduced order model (ROM) is built to represent the simulation space, and new samples are selected based on the model constructed. The reduced order model (see section 15.3) is then updated based on the simulation results of the sampled points. In this way, an attempt is made to gain the most information possible with a small number of carefully selected sample points, limiting the number of expensive trials needed to understand features of the system space.

Currently, RAVEN provides support for the following adaptive algorithms:

- Limit Surface Search
- Adaptive Monte Carlo
- Adaptive Dynamic Event Tree
- Adaptive Hybrid Dynamic Event Tree
- Adaptive Sparse Grid
- Adaptive Sobol Decomposition

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

10.3.1 Limit Surface Search

The **Limit Surface Search** approach is an advanced methodology that employs a smart sampling around transition zones that determine a change in the status of the system (limit surface). To perform such sampling, RAVEN uses ROMs for predicting, in the input space, the location(s) of these transitions, in order to accelerate the exploration of the input space in proximity of the limit surface.

The specifications of this sampler must be defined within an **`<LimitSurfaceSearch>`** XML block. This XML node accepts one attribute:

- **`name`**, ***required string attribute***, user-defined name of this sampler. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.

In the **`<LimitSurfaceSearch>`** input block, the user needs to specify the variables to sample. As already mentioned, these variables are specified within consecutive **`<variable>`** XML blocks:

- This **`<variable>`** recognizes the following child nodes:
  - **`<distribution>`**, ***string, required field***, name of the distribution that is associated to this variable. Its name needs to be contained in the **`<Distributions>`** block explained in Section 9. In addition, if NDDistribution is used, the attribute **`dim`** is required. **Note:** Alternatively, this node must be omitted if the **`<function>`** node is supplied.
  - **`<function>`**, ***string, required field***, name of the function that defines the calculation of this variable from other distributed variables. Its name needs to be contained in the **`<Functions>`** block explained in Section 16. This function must implement a method named "evaluate". **Note:** Alternatively, this node must be ommitted if the **`<distribution>`** node is supplied.

In addition to the **`<variable>`** nodes, the main XML node **`<Adaptive>`** needs to contain two supplementary sub-nodes:

- **`<Convergence>`**, ***float, required field***, Convergence tolerance. The meaning of this tolerance depends on the definition of other attributes that might be defined in this XML node:
  - **`limit`**, ***optional integer attribute***, the maximum number of adaptive samples (iterations).
    Default: infinite.

- **forceIteration**, ***optional boolean attribute***, this attribute controls if at least a number of iterations equal to **limit** must be performed.
  Default: False.
- **weight**, ***optional string attribute (case insensitive)***, defines on what the convergence check needs to be performed.
  * **'CDF'**, the convergence is checked in terms of probability (Cumulative Distribution Function). From a practical point of view, this means that full uncertain domain is discretized in a way that the probability volume of each cell is going to be equal to the tolerance specified in the body of the node **<Convergence>**
  * **'value'**, the convergence is checked on the hyper-volume in terms of variable values.From a practical point of view, this means that full uncertain domain is discretized in a way that the "volume" fraction of each cell is going to be equal to the tolerance specified in the body of the node **<Convergence>**. In other words, each cell volume is going to be equal to the total volume times the tolerance.

  Default: CDF.
- **persistence**, ***optional integer attribute***, offers an additional convergence check. It represents the number of times the computed error needs to be below the inputted tolerance before convergence is reported.
  Default: 5.
- **subGridTol**, ***optional float attribute***, this attribute is used to activate the multi-grid approach (adaptive meshing) of the constructed evaluation grid (see attribute **weight**). In case this attribute is specified, the final grid discretization (cell's "volume content" aka convergence confidence) is represented by the value here specified. The sampler converges on the initial coarse grid, defined by the tolerance specified in the body of the node **<Convergence>**. When the Limit Surface has been identified on the coarse grid, the sampler starts refining the grid until the "volume content" of each cell is equal to the value specified in this attribute (Multi-grid approach).
  Default: None.

In summary, this XML node contains the information that is needed in order to control this sampler's convergence criterion.

- **<batchStrategy>**, ***string, optional field***, defines how points should be selected within a batch of size n where n is given by the **<maxBatchSize>** parameter below. Four options are available:
  - **'none'** If this is specified then the **<maxBatchSize>** parameter below will be ignored and the functionality will replicate the LimitSurfaceSearch, in that the limit surface will be rebuilt and the points will be re-scored after each trial is completed.
  - **'naive'** The top n candidates will be queued for adaptive sampling before retraining the limit surface and re-scoring the new candidate set.

- **'maxP'** The topology of the limit surface given the scoring function values will be decomposed and the top n highest topologically persistent features (local maxima) will be queued for adaptive sampling before retraining and re-scoring the new candidate set.
- **'maxV'** The topology of the limit surface given the scoring function values will be decomposed and the top n highest topological features (local maxima) will be queued for adaptive sampling before retraining and re-scoring the new candidate set.

Default: none.

- **<maxBatchSize>**, ***integer, optional field***, specifies the number of points to select for adaptive sampling before retraining the limit surface and re-scoring the candidates. This is the equivalent of the n parameter used in the **<batchStrategy>** description.
Default: 1.

- **<scoring>**, ***string, optional field***, defines the scoring function to use on the candidate limit surface points in order to select the next adaptive point. Two options are available:
  - **'distance'** will scoring the candidate points by their distance to the closest realized point, in this way preference is given to unexplored regions of the limit surface.
  - **'distancePersistence'** augments the distance above by multiplying it with the inverse persistence of a candidate point which measures how many times the label of the candidate point has changed throughout the lifespan of the algorithm.

Default: distancePersistence.

- **<simplification>**, ***float in the range [0,1], optional field***, specifies the percent of the scoring function range (on the candidate set) as the amount of topological simplification to do before extracting the topological features from the candidate set (local maxima). This only applies when the **<batchStrategy>** is set to **'maxP'** or **'maxV'**. Thus, one may end up with a batch size less than that specified by **<maxBatchSize>**.
Default: 0.

- **<thickness>**, ***positive integer, optional field***, specifies how much the limit surface should be expanded (in terms of grid distance) when constructing a candidate set. A value of 1 implies only the points bounding the limit surface.
Default: 1.

- **<threshold>**, ***float in the range [0,1], optional field***, once the candidates have been ranked and selected, before queueing them for adaptive sampling, this value is used to threshold any points whose score is less than this percentage of the scoring function range (on the candidate set). Thus, one may end up with a batch size less than that specified by **<maxBatchSize>**.
Default: 0

- **Assembler Objects** These objects are either required or optional depending on the functionality of the LimitSurfaceSearch Sampler. The objects must be listed with a rigorous syntax that, except for the XML node tag, is common among all the objects. Each of these nodes must contain 2 attributes that are used to identify them within the simulation framework:

  - `class`, ***required string attribute***, the main "class" of the listed object. For example, it can be `'Models'`, `'Functions'`, etc.
  - `type`, ***required string attribute***, the object identifier or sub-type. For example, it can be `'ROM'`, `'External'`, etc.

  The **LimitSurfaceSearch** approach requires or optionally accepts the following object types:

  - `<Function>`, ***string, required field***, the body of this XML block needs to contain the name of an external function object defined within the `<Functions>` main block (see Section 16). This object represents the boolean function that defines the transition boundaries. This function must implement a method called `__residuumSign(self)`, that returns either -1 or 1, depending on the system conditions (see Section 16.
  - `<ROM>`, ***, string, optional field***, if used, the body of this XML node must contain the name of a ROM defined in the `<Models>` block (see Section 15.3). The ROM here specified is going to be used as "acceleration model" to speed up the convergence of the sampling strategy. The `<Target>` XML node in the ROM input block (within the `<Models>` section) needs to match the name of the goal `<Function>` (e.g. if the goal function is named "transitionIdentifier", the `<Target>` of the ROM needs to report the same name: `<Target>`**transitionIdentifier**`<Target>`).
  - `<TargetEvaluation>`, ***string, required field***, represents the container where the system evaluations are stored. From a practical point of view, this XML node must contain the name of a data object defined in the `<DataObjects>` block (see Section 12). The object here specified must be input as `<Output>` in the Steps that employ this sampling strategy. The Limit Surface Search sampling accepts "DataObjects" of type "PointSet" only.

Example:

```
<Samplers>
  ...
  <LimitSurfaceSearch name='LSSName'>
    <ROM class='Models' type='ROM'>ROMname</ROM>
    <Function class='Functions' type='External'
       >FunctionName</Function>
    <TargetEvaluation class='DataObjects'
       type='PointSet'>DataName</TargetEvaluation>
    <Convergence limit='3000'  forceIteration='False'
       weight='CDF'  subGridTol='1e-4' persistence='5'>
```

```
      1e-2
    </Convergence>
    <variable name='var1'>
      <distribution>aDistributionNameDefinedInDistributionBlock1
         </distribution>
    </variable>
    <variable name='var2'>
      <distribution>aDistributionNameDefinedInDistributionBlock2
         </distribution>
    </variable>
    <variable name='var3'>
      <distribution>aDistributionNameDefinedInDistributionBlock3
         </distribution>
    </variable>
  </LimitSurfaceSearch>
  ...
</Samplers>
```

Batch sampling Example:

```
<Samplers>
  ...
  <LimitSurfaceSearch name='LSBSName'>
    <ROM class='Models' type='ROM'>ROMname</ROM>
    <Function class='Functions' type='External'
       >FunctionName</Function>
    <TargetEvaluation class='DataObjects'
       type='PointSet'>DataName</TargetEvaluation>
    <Convergence limit='3000'  forceIteration='False'
       weight='CDF'  subGridTol='1e-4' persistence='5'>
      1e-2
    </Convergence>
    <scoring>distancePersistence</scoring>
    <batchStrategy>maxP</batchStrategy>
    <thickness>1</thickness>
    <maxBatchSize>4</maxBatchSize>
    <variable name='var1'>
      <distribution>aDistributionNameDefinedInDistributionBlock1
         </distribution>
    </variable>
    <variable name='var2'>
      <distribution>aDistributionNameDefinedInDistributionBlock2
         </distribution>
```

```
    </variable>
    <variable name='var3'>
      <distribution>aDistributionNameDefinedInDistributionBlock3
         </distribution>
    </variable>
  </LimitSurfaceSearch>
  ...
</Samplers>
```

Associated External Python Module:

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

10.3.2 Adaptive Monte Carlo

The **<AdaptiveMonteCarlo>** approach is an extension of the **<MonteCarlo>** sampler. However, instead of having a predefined number of samples, the **<AdaptiveMonteCarlo>** sampler continues sampling until the standard error of all the desired metrics are less than the specified tolerance.

The specifications of this sampler must be defined within an **<AdaptiveMonteCarlo>** XML block.

- **name**, ***required string attribute***, user-defined name of this sampler. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.

In the **<AdaptiveMonteCarlo>** input block, the user needs to specify the variables to sample. As already mentioned, these variables are specified within consecutive **<variable>** XML blocks:

- **<variable>**, ***XML node, required parameter*** can specify the following attribute:
  - **name**, ***required string attribute***, user-defined name of this variable.
  - **shape**, ***comma-separated integers, optional field***, determines the number of samples and shape of samples to be taken. For example, **shape**="2,3" will provide a 2 by 3

matrix of values, while **shape**="10" will produce a vector of 10 values. Omitting this optional attribute will result in a single scalar value instead. Each of the values in the matrix or vector will be the same as the single sampled value. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

This **<variable>** recognizes the following child nodes:

- **<distribution>**, ***string, required field***, name of the distribution that is associated to this variable. Its name needs to be contained in the **<Distributions>** block explained in Section 9. In addition, if NDDistribution is used, the attribute **dim** is required. **Note:** Alternatively, this node must be omitted if the **<function>** node is supplied.
- **<function>**, ***string, required field***, name of the function that defines the calculation of this variable from other distributed variables. Its name needs to be contained in the **<Functions>** block explained in Section 16. This function must implement a method named "evaluate". **Note:** Alternatively, this node must be ommitted if the **<distribution>** node is supplied.

- **<constant>**, ***XML node, optional parameter*** the user is able to input variables that need to be kept constant. For doing this, as many **<constant>** nodes as needed can be input. There are options for setting a constant. To simply set the constant's value, the body of the node contains the constant value, and the **<constant>** node has the following attributes:

  - **name**, ***required string attribute***, user-defined name of this constant.
  - **shape**, ***comma-separated integers, optional field***, determines the shape of samples of the constant value. For example, **shape**="2,3" will shape the values into a 2 by 3 matrix, while **shape**="10" will shape into a vector of 10 values. Unlike the **<variable>**, the constant requires each value be entered; the number of required values is equal to the product of the **shape**. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

  Alternatively, the constant value can be read from a DataObject that has been identified as a **<ConstantSource>** for this Sampler. In this case, the body of the **<constant>** node is the name of the variable that needs to be read from the **<ConstantSource>**, and the **<constant>** node has the following additional attributes in addition to the those above:

  - **source**, ***required string attribute***, the name of the DataObject containing the value to be used for this constant. This must be the name of one of the **<ConstantSource>** DataObjects.
  - **index**, ***optional integer attribute***, the index of the realization in the source DataObject that contains the value to use for the constant.
    Default: the last entry

By way of example, consider the following Sampler definition. The constant will be named **'C'** in the Sampler, and its value is taken from the DataObject **'MyConstant'**, which is identified in the **<ConstantSource>** node. To find the value of the constant in **'MyConstant'**, the Sampler will look at the realization with index **'3'** for the value of variable **'A'** to use as the constant value.

```
<Samplers>
  <WhatEverSampler name='whatever'>
    <ConstantSource class='DataObjects'
       type='PointSet'>MyConstants</ConstantSource>
    <constant name='C' source='MyConstants'
       index='3'>A</constant>
  </WhatEverSampler>
</Samplers>
```

In addition to the **<variable>** nodes, the main **<AdaptiveMonteCarlo>** node needs to contain the following supplementary sub-nodes:

- **<Convergence>** recognizes the following child nodes:
  - **<limit>**, ***integer required field***, the maximum number of adaptive samples (iterations).
    Default: infinite.
  - **<forceIteration>**, ***boolean optional field***, this attribute controls if at least a number of iterations equal to **limit** must be performed.
    Default: False.
  - **<persistence>**, ***integer optional field***, offers an additional convergence check. It represents the number of times the computed error needs to be below the inputted tolerance before convergence is reported.
    Default: 5.
  - **<"metric">**, ***comma separated string list, required field***, specifications for the aggregate metrics on which **<AdaptiveMonteCarlo>** will attempt to converge. The name of each node is the requested metric. The text of the node is a comma-separated list of the parameters for which the metric should be calculated. See the example below.

    <AdaptiveMonteCarlo> will attempt to converge the standard errors of the requested metrics. Currently the metrics available are:

    * **expectedValue**: expected value or mean
    * **median**: median
    * **variance**: variance

* **sigma**: standard deviation
* **skewness**: skewness
* **kurtosis**: excess kurtosis (also known as Fisher's kurtosis)

The nodes containing metrics need to contain the following attributes:

* `prefix`, ***required string attribute***, user-defined prefix for the given **metric**. For scalar quantifies, RAVEN will define a variable with name defined as: "prefix" + "_" + "parameter name". For example, if we define "mean" as the prefix for **expectedValue**, and parameter "x", then variable "mean_x" will be defined by RAVEN. For matrix quantities, RAVEN will define a variable with name defined as: "prefix" + "_" + "target parameter name" + "_" + "feature parameter name". For example, if we define "sen" as the prefix for **sensitivity**, target "y" and feature "x", then variable "sen_y_x" will be defined by RAVEN. **Note:** These variable will be used by RAVEN for the internal calculations. It is also accessible by the user through **DataObjects** and **OutStreams**.
* `tol`, ***required float attribute***, convergence tolerance for the standard error of the metric.

RAVEN will define a variable with name defined as: "prefix for given **metric**" + "_ste_" + "parameter name" to store the standard error of the given **metric** with respect to the given parameter. This variable needs to be included in the **`<TargetEvaluation>`** **`<DataObject>`** which is an output of the **`<Step>`** in which the **`<AdaptiveMonteCarlo>`** is used. This variable is also available for output to the **`<SolutionExport>`** **`<DataObjec>`**.

Note: When defining the metrics to use, it is possible to have multiple nodes with the same name. For example, if a problem has inputs $X1$, and $X2$, and the responses are $Y1$, $Y2$, it is possible that the desired metrics are the **`<sigma>`** of $Y1$,and $Y2$ on same tolerance, and **`<expectedValue>`** of $Y1$,and $Y2$ on different tolerance. The first has the parameters $Y1, Y2$ in the same node with one tolerance attribute, while the second need to divide into two nodes. One has target $Y1$ and another one has target $Y2$ instead. This could reduce some computation effort in problems with many responses or inputs. An example of this is shown below.

In summary, the **`<convergence>`** node contains the information that is needed in order to control the **`<AdaptiveMonteCarlo>`** sampler's convergence criteria.

- **`<initialSeed>`**, ***integer, optional field***, initial seeding of random number generator for Monte Carlo sampler. By default, RAVEN uses an internal static seed.
  Default: 20021986

- **Assembler Objects** These objects are either required or optional depending on the functionality of the AdaptiveMonteCarlo Sampler. The objects must be listed with a rigorous syntax

that, except for the XML node tag, is common among all the objects. Each of these nodes must contain 2 attributes that are used to identify them within the simulation framework:

- **class**, ***required string attribute***, the main "class" of the listed object. For example, it can be **'Models'**, **'Functions'**, etc.
- **type**, ***required string attribute***, the object identifier or sub-type. For example, it can be **'ROM'**, **'External'**, etc.

The **AdaptiveMonteCarlo** approach requires or optionally accepts the following object types:

- **<TargetEvaluation>**, ***string, required field***, represents the container where the system evaluations are stored. From a practical point of view, this XML node must contain the name of a data object defined in the **<DataObjects>** block (see Section 12). The adaptive sampling accepts "DataObjects" of type "PointSet" only.

Example:

```
<Samplers>
  ...
  <AdaptiveMonteCarlo name = 'AdaptiveName'>
    <TargetEvaluation class = 'DataObjects' type =
       'PointSet'>DataName</TargetEvaluation>
    <Convergence>
      <forceIteration>False</forceIteration>
      <limit>30</limit>
      <persistence>6</persistence>
      <expectedValue prefix="mean"
         tol="1e-1">y1,y2</expectedValue>
      <sigma prefix="sigma" tol="6e-2">y1</sigma>
      <sigma prefix="sigma" tol="5e-2">y2</sigma>
    </Convergence>
    <variable name = 'var1'>
        <distribution>
         aDistributionNameDefinedInDistributionBlock1
        </distribution>
    </variable>
    <variable name = 'var2'>
        <distribution>
          aDistributionNameDefinedInDistributionBlock2
        </distribution>
    </variable>
    <variable name = 'var3'>
```

```
        <distribution>
          aDistributionNameDefinedInDistributionBlock3
        </distribution>
    </variable>
  </AdaptiveMonteCarlo>
  ...
</Samplers>
```

10.3.3 Adaptive Dynamic Event Tree

The **Adaptive Dynamic Event Tree** approach is an advanced methodology employing a smart sampling around transition zones that determine a change in the status of the system (limit surface), using the support of a Dynamic Event Tree methodology. The main idea of the application of the previously explained adaptive sampling approach to the DET comes from the observation that the DET, when evaluated from a limit surface perspective, is intrinsically adaptive. For this reason, it appears natural to use the DET approach to perform a goal-function oriented pre-sampling of the input space.

RAVEN uses ROMs for predicting, in the input space, the location(s) of these transitions, in order to accelerate the exploration of the input space in proximity of the limit surface.

The specifications of this sampler must be defined within an **`<AdaptiveDynamicEventTree>`** XML block. This XML node accepts one attribute:

- **`name`**, ***required string attribute***, user-defined name of this sampler. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.
- **`printEndXmlSummary`**, ***optional string/boolean attribute***, this attribute controls the dumping of a "summary" of the DET performed in to an external XML.
  Default: False.
- **`maxSimulationTime`**, ***optional float attribute***, this attribute controls the maximum "mission" time of the simulation underneath.
  Default: None.
- **`mode`**, ***optional string attribute***, controls when the adaptive search needs to begin. Two options are available:
  - **`'post'`**, if this option is activated, the sampler first performs a standard Dynamic Event Tree analysis. At end of it, it uses the outcomes to start the adaptive search in conjunction with the DET support.

- **'online'**, if this option is activated, the adaptive search starts at the beginning, during the initial standard Dynamic Event Tree analysis. Whenever a transition is detected, the **Adaptive Dynamic Event Tree** starts its goal-oriented search using the DET as support;

Default: post.

- **updateGrid**, ***optional boolean attribute***, if true, each adaptive request is going to update the meshing of the initial DET grid.
  Default: True.

In the **<AdaptiveDynamicEventTree>** input block, the user needs to specify the variables to sample. As already mentioned, these variables are specified within consecutive **<variable>** XML blocks:

- **<variable>**, ***XML node, required parameter*** can specify the following attribute:
  - **name**, ***required string attribute***, user-defined name of this variable.
  - **shape**, ***comma-separated integers, optional field***, determines the number of samples and shape of samples to be taken. For example, **shape**="2,3" will provide a 2 by 3 matrix of values, while **shape**="10" will produce a vector of 10 values. Omitting this optional attribute will result in a single scalar value instead. Each of the values in the matrix or vector will be the same as the single sampled value. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

  This **<variable>** recognizes the following child nodes:

  - **<distribution>**, ***string, required field***, name of the distribution that is associated to this variable. Its name needs to be contained in the **<Distributions>** block explained in Section 9. In addition, if NDDistribution is used, the attribute **dim** is required. **Note:** Alternatively, this node must be omitted if the **<function>** node is supplied.
  - **<function>**, ***string, required field***, name of the function that defines the calculation of this variable from other distributed variables. Its name needs to be contained in the **<Functions>** block explained in Section 16. This function must implement a method named "evaluate". **Note:** Alternatively, this node must be ommitted if the **<distribution>** node is supplied.
  - **<grid>**, ***space separated floats, required field***, the content of this XML node depends on the definition of the associated attributes:
    * **type**, ***required string attribute***, user-defined discretization metric type: 1) **'CDF'**, the grid will be specified based on cumulative distribution function probability thresholds, and 2) **'value'**, the grid will be provided using variable values.

* **construction**, ***required string attribute***, how the grid needs to be constructed, independent of its type (i.e. **'CDF'** or **'value'**).

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

* **construction='equal'**. The grid is going to be constructed equally-spaced (**type='value'**) or equally probable (**type='CDF'**). This construction type requires the definition of additional attributes:
  · **steps**, ***required integer attribute***, number of equally spaced/probable discretization steps.

  This construction type requires that the content of the **<grid>** node represents the lower and upper bounds (either in probability or value). Two values need to be specified; the lowest one will be considered as the $lowerBound$, the largest, the $upperBound$. The lower and upper bounds are checked against the associated **<distribution>** bounds. If one or both of them falls outside the distribution's bounds, the code will raise an error. The $stepSize$ is determined as follows:
  $stepSize = (upperBound - lowerBound)/steps$
* **construction='custom'**. The grid will be directly specified by the user. No additional attributes are needed. This construction type requires that the **<grid>** node contains the actual mesh bins. For example, if the grid **type** is **'CDF'**, in the body of **<grid>**, the user will specify the CDF probability thresholds (nodalization in probability). All the bins are checked against the associated **<distribution>** bounds. If one or more of them falls outside the distribution's bounds, the code will raise an error.

Note: The **<grid>** node is only required if a **<distribution>** node is supplied. In the case of a **<function>** node, no grid information is requested.

- **<constant>**, ***XML node, optional parameter*** the user is able to input variables that need to be kept constant. For doing this, as many **<constant>** nodes as needed can be input. There are options for setting a constant. To simply set the constant's value, the body of the node contains the constant value, and the **<constant>** node has the following attributes:
  - **name**, ***required string attribute***, user-defined name of this constant.
  - **shape**, ***comma-separated integers, optional field***, determines the shape of samples of the constant value. For example, **shape**="2,3" will shape the values into a 2 by 3 matrix, while **shape**="10" will shape into a vector of 10 values. Unlike the **<variable>**, the constant requires each value be entered; the number of required values is equal to the product of the **shape**. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

Alternatively, the constant value can be read from a DataObject that has been identified as a **<ConstantSource>** for this Sampler. In this case, the body of the **<constant>** node is the name of the variable that needs to be read from the **<ConstantSource>**, and the **<constant>** node has the following additional attributes in addition to the those above:

- **source**, ***required string attribute***, the name of the DataObject containing the value to be used for this constant. This must be the name of one of the **<ConstantSource>** DataObjects.
- **index**, ***optional integer attribute***, the index of the realization in the source DataObject that contains the value to use for the constant.
  Default: the last entry

By way of example, consider the following Sampler definition. The constant will be named **'C'** in the Sampler, and its value is taken from the DataObject **'MyConstant'**, which is identified in the **<ConstantSource>** node. To find the value of the constant in **'MyConstant'**, the Sampler will look at the realization with index **'3'** for the value of variable **'A'** to use as the constant value.

```
<Samplers>
  <WhatEverSampler name='whatever'>
    <ConstantSource class='DataObjects'
       type='PointSet'>MyConstants</ConstantSource>
    <constant name='C' source='MyConstants'
       index='3'>A</constant>
  </WhatEverSampler>
</Samplers>
```

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

- **<Convergence>**, ***float, required field***, Convergence tolerance. The meaning of this tolerance depends on the definition of other attributes that might be defined in this XML node:
  - **limit**, ***optional integer attribute***, the maximum number of adaptive samples (iterations).
    Default: infinite.
  - **forceIteration**, ***optional boolean attribute***, this attribute controls if at least a number of iterations equal to **limit** must be performed.
    Default: False.
  - **weight**, ***optional string attribute (case insensitive)***, defines on what the convergence check needs to be performed.
    * **'CDF'**, the convergence is checked in terms of probability (Cumulative Distribution Function). From a practical point of view, this means that full uncertain domain is discretized in a way that the probability volume of each cell is going to be equal to the tolerance specified in the body of the node **<Convergence>**

* **'value'**, the convergence is checked on the hyper-volume in terms of variable values.From a practical point of view, this means that full uncertain domain is discretized in a way that the "volume" fraction of each cell is going to be equal to the tolerance specified in the body of the node **<Convergence>**. In other words, each cell volume is going to be equal to the total volume times the tolerance.

Default: CDF.

- **persistence**, ***optional integer attribute***, offers an additional convergence check. It represents the number of times the computed error needs to be below the inputted tolerance before convergence is reported.
  Default: 5.
- **subGridTol**, ***optional float attribute***, this attribute is used to activate the multi-grid approach (adaptive meshing) of the constructed evaluation grid (see attribute **weight**). In case this attribute is specified, the final grid discretization (cell's "volume content" aka convergence confidence) is represented by the value here specified. The sampler converges on the initial coarse grid, defined by the tolerance specified in the body of the node **<Convergence>**. When the Limit Surface has been identified on the coarse grid, the sampler starts refining the grid until the "volume content" of each cell is equal to the value specified in this attribute (Multi-grid approach).
  Default: None.

In summary, this XML node contains the information that is needed in order to control this sampler's convergence criterion.

- **Assembler Objects** These objects are either required or optional depending on the functionality of the AdaptiveDynamicEventTree Sampler. The objects must be listed with a rigorous syntax that, except for the XML node tag, is common among all the objects. Each of these nodes must contain 2 attributes that are used to identify them within the simulation framework:

  - **class**, ***required string attribute***, the main "class" of the listed object. For example, it can be **'Models'**, **'Functions'**, etc.
  - **type**, ***required string attribute***, the object identifier or sub-type. For example, it can be **'ROM'**, **'External'**, etc.

  The **AdaptiveDynamicEventTree** approach requires or optionally accepts the following object types:

  - **<Function>**, ***string, required field***, the body of this XML block needs to contain the name of an external function object defined within the **<Functions>** main block (see Section 16). This object represents the boolean function that defines the transition boundaries. This function must implement a method called `__residuumSign(self)`, that returns either -1 or 1, depending on the system conditions (see Section 16.

- **<ROM>**, , ***string, optional field***, if used, the body of this XML node must contain the name of a ROM defined in the **<Models>** block (see Section 15.3). The ROM here specified is going to be used as "acceleration model" to speed up the convergence of the sampling strategy. The **<Target>** XML node in the ROM input block (within the **<Models>** section) needs to match the name of the goal **<Function>** (e.g. if the goal function is named "transitionIdentifier", the **<Target>** of the ROM needs to report the same name: **<Target>transitionIdentifier<Target>**).
- **<TargetEvaluation>**, ***string, required field***, represents the container where the system evaluations are stored. From a practical point of view, this XML node must contain the name of a data object defined in the **<DataObjects>** block (see Section 12). The adaptive sampling accepts "DataObjects" of type "PointSet" only.

Example:

```
<Samplers>
  ...
  <AdaptiveDynamicEventTree name = 'AdaptiveName'>
    <ROM class = 'Models' type = 'ROM'ROMname</ROM>
    <Function class = 'Functions' type =
       'External'>FunctionName</Function>
    <TargetEvaluation class = 'DataObjects' type =
       'PointSet'>DataName</TargetEvaluation>
    <Convergence limit = '3000' subGridTol= '0.001'
       forceIteration = 'False' weight = 'CDF'
       subGriTol='''1e-5' persistence = '5'>
      1e-2
    </Convergence>
    <variable name = 'var1'>
        <distribution>
         aDistributionNameDefinedInDistributionBlock1
        </distribution>
        <grid type='CDF' construction='custom'>0.1 0.8</grid>
    </variable>
    <variable name = 'var2'>
        <distribution>
          aDistributionNameDefinedInDistributionBlock2
        </distribution>
        <grid type='CDF' construction='custom'>0.1 0.8</grid>
    </variable>
    <variable name = 'var3'>
        <distribution>
          aDistributionNameDefinedInDistributionBlock3
        </distribution>
```

```
        <grid type='CDF' construction='custom'>0.1 0.8</grid>
    </variable>
  </AdaptiveDynamicEventTree>
  ...
</Samplers>
```

Associated External Python Module:

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

10.3.4 Adaptive Hybrid Dynamic Event Tree

The **Adaptive Hybrid Dynamic Event Tree** approach is an advanced methodology employing a smart sampling around transition zones that determine a change in the status of the system (limit surface), using the support of the Hybrid Dynamic Event Tree methodology. Practically, this methodology represents a conjunction between the previously described Adaptive DET and the Hybrid DET method for the treatment of the epistemic variables.

Regarding the input requirements, the AHDET sampler is a "sub-type" of the **`<AdaptiveDynamicEventTree>`** sampler. For this reason, its specifications must be defined within a **`<AdaptiveDynamicEventTree>`** block.

The specifications of this sampler must be defined within an **`<AdaptiveDynamicEventTree>`** XML block. This XML node accepts one attribute:

- **`name`**, ***required string attribute***, user-defined name of this sampler. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.
- **`printEndXmlSummary`**, ***optional string/boolean attribute***, this attribute controls the dumping of a "summary" of the DET performed in to an external XML.
  Default: False.
- **`maxSimulationTime`**, ***optional float attribute***, this attribute controls the maximum "mission" time of the simulation underneath.
  Default: None.
- **`mode`**, ***optional string attribute***, controls when the adaptive search needs to begin. Two options are available:

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

Default: post.

- **updateGrid**, ***optional boolean attribute***, if true, each adaptive request is going to update the meshing of the initial DET grid.
  Default: True.

In the **<AdaptiveDynamicEventTree>** input block, the user needs to specify the variables to sample. As already mentioned, these variables are specified within consecutive **<variable>** XML blocks:

- **<variable>**, ***XML node, required parameter*** can specify the following attribute:
  - **name**, ***required string attribute***, user-defined name of this variable.
  - **shape**, ***comma-separated integers, optional field***, determines the number of samples and shape of samples to be taken. For example, **shape**="2,3" will provide a 2 by 3 matrix of values, while **shape**="10" will produce a vector of 10 values. Omitting this optional attribute will result in a single scalar value instead. Each of the values in the matrix or vector will be the same as the single sampled value. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

  This **<variable>** recognizes the following child nodes:

  - **<distribution>**, ***string, required field***, name of the distribution that is associated to this variable. Its name needs to be contained in the **<Distributions>** block explained in Section 9. In addition, if NDDistribution is used, the attribute **dim** is required. **Note:** Alternatively, this node must be omitted if the **<function>** node is supplied.
  - **<function>**, ***string, required field***, name of the function that defines the calculation of this variable from other distributed variables. Its name needs to be contained in the **<Functions>** block explained in Section 16. This function must implement a method named "evaluate". **Note:** Alternatively, this node must be ommitted if the **<distribution>** node is supplied.
  - **<grid>**, ***space separated floats, required field***, the content of this XML node depends on the definition of the associated attributes:

* **type**, ***required string attribute***, user-defined discretization metric type: 1) **'CDF'**, the grid will be specified based on cumulative distribution function probability thresholds, and 2) **'value'**, the grid will be provided using variable values.
* **construction**, ***required string attribute***, how the grid needs to be constructed, independent of its type (i.e. **'CDF'** or **'value'**).

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

* **construction='equal'**. The grid is going to be constructed equally-spaced (**type='value'**) or equally probable (**type='CDF'**). This construction type requires the definition of additional attributes:
  · **steps**, ***required integer attribute***, number of equally spaced/probable discretization steps.

  This construction type requires that the content of the **<grid>** node represents the lower and upper bounds (either in probability or value). Two values need to be specified; the lowest one will be considered as the $lowerBound$, the largest, the $upperBound$. The lower and upper bounds are checked against the associated **<distribution>** bounds. If one or both of them falls outside the distribution's bounds, the code will raise an error. The $stepSize$ is determined as follows: $stepSize = (upperBound - lowerBound)/steps$
* **construction='custom'**. The grid will be directly specified by the user. No additional attributes are needed. This construction type requires that the **<grid>** node contains the actual mesh bins. For example, if the grid **type** is **'CDF'**, in the body of **<grid>**, the user will specify the CDF probability thresholds (nodalization in probability). All the bins are checked against the associated **<distribution>** bounds. If one or more of them falls outside the distribution's bounds, the code will raise an error.

Note: The **<grid>** node is only required if a **<distribution>** node is supplied. In the case of a **<function>** node, no grid information is requested.

- **<constant>**, ***XML node, optional parameter*** the user is able to input variables that need to be kept constant. For doing this, as many **<constant>** nodes as needed can be input. There are options for setting a constant. To simply set the constant's value, the body of the node contains the constant value, and the **<constant>** node has the following attributes:

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

  – **shape**, ***comma-separated integers, optional field***, determines the shape of samples of the constant value. For example, **shape**="2,3" will shape the values into a 2 by 3 matrix, while **shape**="10" will shape into a vector of 10 values. Unlike the **<variable>**, the constant requires each value be entered; the number of required values is equal to the product of the **shape**. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

Alternatively, the constant value can be read from a DataObject that has been identified as a **<ConstantSource>** for this Sampler. In this case, the body of the **<constant>** node is the name of the variable that needs to be read from the **<ConstantSource>**, and the **<constant>** node has the following additional attributes in addition to the those above:

- **source**, ***required string attribute***, the name of the DataObject containing the value to be used for this constant. This must be the name of one of the **<ConstantSource>** DataObjects.
- **index**, ***optional integer attribute***, the index of the realization in the source DataObject that contains the value to use for the constant.
  Default: the last entry

By way of example, consider the following Sampler definition. The constant will be named **'C'** in the Sampler, and its value is taken from the DataObject **'MyConstant'**, which is identified in the **<ConstantSource>** node. To find the value of the constant in **'MyConstant'**, the Sampler will look at the realization with index **'3'** for the value of variable **'A'** to use as the constant value.

```
<Samplers>
  <WhatEverSampler name='whatever'>
    <ConstantSource class='DataObjects'
       type='PointSet'>MyConstants</ConstantSource>
    <constant name='C' source='MyConstants'
       index='3'>A</constant>
  </WhatEverSampler>
</Samplers>
```

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

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

* **'CDF'**, the convergence is checked in terms of probability (Cumulative Distribution Function). From a practical point of view, this means that full uncertain domain is discretized in a way that the probability volume of each cell is going to be equal to the tolerance specified in the body of the node **<Convergence>**
* **'value'**, the convergence is checked on the hyper-volume in terms of variable values.From a practical point of view, this means that full uncertain domain is discretized in a way that the "volume" fraction of each cell is going to be equal to the tolerance specified in the body of the node **<Convergence>**. In other words, each cell volume is going to be equal to the total volume times the tolerance.

Default: CDF.

– **persistence**, ***optional integer attribute***, offers an additional convergence check. It represents the number of times the computed error needs to be below the inputted tolerance before convergence is reported.
Default: 5.

– **subGridTol**, ***optional float attribute***, this attribute is used to activate the multi-grid approach (adaptive meshing) of the constructed evaluation grid (see attribute **weight**). In case this attribute is specified, the final grid discretization (cell's "volume content" aka convergence confidence) is represented by the value here specified. The sampler converges on the initial coarse grid, defined by the tolerance specified in the body of the node **<Convergence>**. When the Limit Surface has been identified on the coarse grid, the sampler starts refining the grid until the "volume content" of each cell is equal to the value specified in this attribute (Multi-grid approach).
Default: None.

In summary, this XML node contains the information that is needed in order to control this sampler's convergence criterion.

- **Assembler Objects** These objects are either required or optional depending on the functionality of the AdaptiveDynamicEventTree Sampler. The objects must be listed with a rigorous syntax that, except for the XML node tag, is common among all the objects. Each of these nodes must contain 2 attributes that are used to identify them within the simulation framework:

  – **class**, ***required string attribute***, the main "class" of the listed object. For example, it can be **'Models'**, **'Functions'**, etc.

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

The **AdaptiveDynamicEventTree** approach requires or optionally accepts the following object types:

- **<Function>**, ***string, required field***, the body of this XML block needs to contain the name of an external function object defined within the **<Functions>** main block (see Section 16). This object represents the boolean function that defines the transition boundaries. This function must implement a method called `__residuumSign(self)`, that returns either -1 or 1, depending on the system conditions (see Section 16.
- **<ROM>**, ***, string, optional field***, if used, the body of this XML node must contain the name of a ROM defined in the **<Models>** block (see Section 15.3). The ROM here specified is going to be used as "acceleration model" to speed up the convergence of the sampling strategy. The **<Target>** XML node in the ROM input block (within the **<Models>** section) needs to match the name of the goal **<Function>** (e.g. if the goal function is named "transitionIdentifier", the **<Target>** of the ROM needs to report the same name: **<Target>transitionIdentifier<Target>**).
- **<TargetEvaluation>**, ***string, required field***, represents the container where the system evaluations are stored. From a practical point of view, this XML node must contain the name of a data object defined in the **<DataObjects>** block (see Section 12). The adaptive sampling accepts "DataObjects" of type "PointSet" only.

As it can be noticed, the basic specifications of the Adaptive Hybrid Dynamic Event Tree method are consistent with the ones for the ADET methodology. In order to activate the **Adaptive Hybrid Dynamic Event Tree** sampler, the main **<AdaptiveDynamicEventTree>** block needs to contain an additional sub-node called **<HybridSampler>**. This sub-node needs to contain the following attribute:

- **type**, ***required string attribute***, type of pre-sampling strategy to be used. Up to now only one option is available:
  - **'LimitSurface'**. With this option, the epistemic variables here listed are going to be part of the LS search. This means that the discretization of the domain of these variables is determined by the **<Convergece>** node.

Independent of the type of HybridSampler that has been specified, the **<HybridSampler>** must contain the variables that need to be sampled. As already mentioned, these variables are specified within consecutive **<variable>** XML blocks:

- **<variable>**, ***XML node, required parameter*** can specify the following attribute:
  - **name**, ***required string attribute***, user-defined name of this variable.
  - **shape**, ***comma-separated integers, optional field***, determines the number of samples and shape of samples to be taken. For example, **shape**="2,3" will provide a 2 by 3 matrix of values, while **shape**="10" will produce a vector of 10 values. Omitting this optional attribute will result in a single scalar value instead. Each of the values in the

matrix or vector will be the same as the single sampled value. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

This **<variable>** recognizes the following child nodes:

- **<distribution>**, ***string, required field***, name of the distribution that is associated to this variable. Its name needs to be contained in the **<Distributions>** block explained in Section 9. In addition, if NDDistribution is used, the attribute **dim** is required. **Note:** Alternatively, this node must be omitted if the **<function>** node is supplied.
- **<function>**, ***string, required field***, name of the function that defines the calculation of this variable from other distributed variables. Its name needs to be contained in the **<Functions>** block explained in Section 16. This function must implement a method named "evaluate". **Note:** Alternatively, this node must be ommitted if the **<distribution>** node is supplied.

- **<constant>**, ***XML node, optional parameter*** the user is able to input variables that need to be kept constant. For doing this, as many **<constant>** nodes as needed can be input. There are options for setting a constant. To simply set the constant's value, the body of the node contains the constant value, and the **<constant>** node has the following attributes:

  - **name**, ***required string attribute***, user-defined name of this constant.
  - **shape**, ***comma-separated integers, optional field***, determines the shape of samples of the constant value. For example, **shape**="2,3" will shape the values into a 2 by 3 matrix, while **shape**="10" will shape into a vector of 10 values. Unlike the **<variable>**, the constant requires each value be entered; the number of required values is equal to the product of the **shape**. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

  Alternatively, the constant value can be read from a DataObject that has been identified as a **<ConstantSource>** for this Sampler. In this case, the body of the **<constant>** node is the name of the variable that needs to be read from the **<ConstantSource>**, and the **<constant>** node has the following additional attributes in addition to the those above:

  - **source**, ***required string attribute***, the name of the DataObject containing the value to be used for this constant. This must be the name of one of the **<ConstantSource>** DataObjects.
  - **index**, ***optional integer attribute***, the index of the realization in the source DataObject that contains the value to use for the constant.
    Default: the last entry

  By way of example, consider the following Sampler definition. The constant will be named **'C'** in the Sampler, and its value is taken from the DataObject **'MyConstant'**, which is identified in the **<ConstantSource>** node. To find the value of the constant in **'MyConstant'**,

the Sampler will look at the realization with index **'3'** for the value of variable **'A'** to use as the constant value.

```
    <Samplers>
      <WhatEverSampler name='whatever'>
        <ConstantSource class='DataObjects'
           type='PointSet'>MyConstants</ConstantSource>
        <constant name='C' source='MyConstants'
           index='3'>A</constant>
      </WhatEverSampler>
    </Samplers>
```

Example:

```
<Samplers>
  ...
  <AdaptiveDynamicEventTree name = 'AdaptiveName'>
    <ROM class = 'Models' type = 'ROM'ROMname</ROM>
    <Function class = 'Functions' type =
       'External'>FunctionName</Function>
    <TargetEvaluation class = 'DataObjects' type =
       'PointSet'>DataName</TargetEvaluation>
    <Convergence limit = '3000' subGridTol= '0.001'
       forceIteration = 'False' weight = 'CDF'
       subGriTol='''1e-5' persistence = '5'>
      1e-2
    </Convergence>
    <HybridSampler type='LimitSurface'>
       <variable name = 'epistemicVar1'>
          <distribution>
            aDistributionNameDefinedInDistributionBlock1
          </distribution>
      </variable>
       <variable name = 'epistemicVar2'>
          <distribution>
            aDistributionNameDefinedInDistributionBlock2
          </distribution>
      </variable>
    </HybridSampler>
    <variable name = 'var1'>
        <distribution>
         aDistributionNameDefinedInDistributionBlock3
        </distribution>
```

```
        <grid type='CDF' construction='custom'>0.1 0.8</grid>
    </variable>
    <variable name = 'var2'>
        <distribution>
          aDistributionNameDefinedInDistributionBlock4
        </distribution>
        <grid type='CDF' construction='custom'>0.1 0.8</grid>
    </variable>
    <variable name = 'var3'>
        <distribution>
          aDistributionNameDefinedInDistributionBlock5
        </distribution>
        <grid type='CDF' construction='custom'>0.1 0.8</grid>
    </variable>

  </AdaptiveDynamicEventTree>
  ...
</Samplers>
```

Associated External Python Module:

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

10.3.5 Adaptive Sparse Grid

The **Adaptive Sparse Grid** approach is an advanced methodology that employs an intelligent search for the most suitable sparse grid quadrature to characterize a model. To perform such sampling, RAVEN adaptively builds an index set and generates sparse grids in a similar manner to Sparse Grid Collocation samplers. In each iterative step, the adaptive index set determines the next possible quadrature orders to add in each dimension, and determines the index set point that would offer the largest impact to one of the convergence metrics. This process continues until the total impact of all the potential index set points is less than tolerance. For many models, this function converges after fewer runs than a traditional Sparse Grid Collocation sampling. However, it should be noted that this algorithm fails in the event that the partial derivative of the response surface with respect to any single input dimension is zero at the origin of the input domain. For example, the adaptive algorithm fails for the model $f(x) = x \cdot y$.

The specifications of this sampler must be defined within an **`<Adaptive Sparse Grid>`** XML block. This XML node accepts one attribute:

- **`name`**, ***required string attribute***, user-defined name of this sampler. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.

In the **`<Adaptive Sparse Grid>`** input block, the user needs to specify the variables to sample. As already mentioned, these variables are specified within consecutive **`<variable>`** XML blocks:

- **`<variable>`**, ***XML node, required parameter*** can specify the following attribute:
  - **`name`**, ***required string attribute***, user-defined name of this variable.
  - **`shape`**, ***comma-separated integers, optional field***, determines the number of samples and shape of samples to be taken. For example, **`shape`**="2,3" will provide a 2 by 3 matrix of values, while **`shape`**="10" will produce a vector of 10 values. Omitting this optional attribute will result in a single scalar value instead. Each of the values in the matrix or vector will be the same as the single sampled value. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

  This **`<variable>`** recognizes the following child nodes:

  - **`<distribution>`**, ***string, required field***, name of the distribution that is associated to this variable. Its name needs to be contained in the **`<Distributions>`** block explained in Section 9. In addition, if NDDistribution is used, the attribute **`dim`** is required. **Note:** Alternatively, this node must be omitted if the **`<function>`** node is supplied.
  - **`<function>`**, ***string, required field***, name of the function that defines the calculation of this variable from other distributed variables. Its name needs to be contained in the **`<Functions>`** block explained in Section 16. This function must implement a method named "evaluate". **Note:** Alternatively, this node must be ommitted if the **`<distribution>`** node is supplied.
- **`<constant>`**, ***XML node, optional parameter*** the user is able to input variables that need to be kept constant. For doing this, as many **`<constant>`** nodes as needed can be input. There are options for setting a constant. To simply set the constant's value, the body of the node contains the constant value, and the **`<constant>`** node has the following attributes:
  - **`name`**, ***required string attribute***, user-defined name of this constant.

- **shape**, ***comma-separated integers, optional field***, determines the shape of samples of the constant value. For example, **shape**="2,3" will shape the values into a 2 by 3 matrix, while **shape**="10" will shape into a vector of 10 values. Unlike the **<variable>**, the constant requires each value be entered; the number of required values is equal to the product of the **shape**. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

Alternatively, the constant value can be read from a DataObject that has been identified as a **<ConstantSource>** for this Sampler. In this case, the body of the **<constant>** node is the name of the variable that needs to be read from the **<ConstantSource>**, and the **<constant>** node has the following additional attributes in addition to the those above:

- **source**, ***required string attribute***, the name of the DataObject containing the value to be used for this constant. This must be the name of one of the **<ConstantSource>** DataObjects.
- **index**, ***optional integer attribute***, the index of the realization in the source DataObject that contains the value to use for the constant.
  Default: the last entry

By way of example, consider the following Sampler definition. The constant will be named **'C'** in the Sampler, and its value is taken from the DataObject **'MyConstant'**, which is identified in the **<ConstantSource>** node. To find the value of the constant in **'MyConstant'**, the Sampler will look at the realization with index **'3'** for the value of variable **'A'** to use as the constant value.

```
<Samplers>
  <WhatEverSampler name='whatever'>
    <ConstantSource class='DataObjects'
       type='PointSet'>MyConstants</ConstantSource>
    <constant name='C' source='MyConstants'
       index='3'>A</constant>
  </WhatEverSampler>
</Samplers>
```

In addition to the **<variable>** nodes, the main XML node **<AdaptiveSparseGrid>** needs to contain the following supplementary sub-nodes:

- **<Convergence>**, ***float, required field***, Convergence tolerance. The meaning of this tolerance depends on the **target** attribute of this node.
  - **target**, ***required string attribute***, the metric for convergence. The following metrics are available: **'variance'**, which converges the sparse quadrature integration of the second moment of the model.

- **maxPolyOrder**, ***optional integer attribute***, limits the maximum size equivalent polynomial for any one dimension.
  Default: 10.
- **persistence**, ***optional integer attribute***, defines the number of index set points that are required to be found before calculation can exit. Setting this to a higher value can help if the adaptive process is not finding significant indices on its own.
  Default: 2.

In summary, this XML node contains the information that is needed in order to control this sampler's convergence criterion.

- **<convergenceStudy>**, ***optional node***, if included, triggers writing state points at particular numbers of model solves for the purpose of a convergence study. The study is performed by writing XML output files as described in the OutStreams for ROMs at the state points requested, using **'all'** as the requested **<what>** values. The state points are identified when a certain number of model runs is passed, as specified by the **<runStatePoints>** node. This node has the following sub-nodes to define its parameters:
  - **<runStatePoints>**, ***list of integers, required node***, lists the number of model runs at which state points should be written. Note that these will be written when the requested number of runs is met or passed, so the actual value is often somewhat more than the requested value, and the exact value will be listed in the XML output.
  - **<baseFilename>**, ***string, optional node***, if specified determines the base file name for the state point outputs. If not specified, defaults to **'out_'**.
  - **<pickle>**, ***no text, optional node***, if this node is included, serialized (pickled) versions of the ROM at each of the run states is also created in the working directory, with the format `<baseFilename><numRuns>.pk`, such as `out_100.pk`.
- **<logFile>**, ***optional node***, if included, the log file onto which the adaptive step progress can be printed. The log includes the values of included polynomial coefficients as well as the expected impacts of polynomial coefficients not yet included. This is different from the convergenceStudy print, which will give statistical moments at certain steps.
- **<maxRuns>**, ***optional node***, if included, the adaptive sampler will track the number of computational solves necessary to construct the associated GaussPolynomialROM. If at any point the number of solves exceeds the value given, it will not initiate any additional solves, and will exit when existing solves finish.

Assembler Objects These objects are either required or optional depending on the functionality of the Adaptive Sparse Grid Sampler. The objects must be listed with a rigorous syntax that, except for the XML node tag, is common among all the objects. Each of these nodes must contain 2 attributes that are used to identify them within the simulation framework:

- `class`, ***required string attribute***, the main "class" of the listed object. For example, it can be `'Models'`, `'Functions'`, etc.
- `type`, ***required string attribute***, the object identifier or sub-type. For example, it can be `'ROM'`, `'External'`, etc.

The **Adaptive Sparse Grid** approach requires or optionally accepts the following object types:

- `<ROM>`, ***string, required field***, the body of this XML node must contain the name of an appropriate ROM defined in the `<Models>` block (see Section 15.3).

- `<TargetEvaluation>`, ***string, required field***, represents the container where the system evaluations are stored. From a practical point of view, this XML node must contain the name of a data object defined in the `<DataObjects>` block (see Section 12). The Adaptive Sparse Grid sampling accepts "DataObjects" of type "PointSet" only.

Example:

```
<Samplers>
  ...
  <AdaptiveSparseGrid name="ASG" verbosity='debug'>
    <Convergence target='coeffs'>1e-2</Convergence>
    <variable name="x1">
      <distribution>UniDist</distribution>
    </variable>
    <variable name="x2">
      <distribution>UniDist</distribution>
    </variable>
    <ROM class = 'Models' type = 'ROM'>gausspolyrom</ROM>
    <TargetEvaluation class = 'DataObjects' type =
       'PointSet'>solns</TargetEvaluation>
  </AdaptiveSparseGrid>
  ...
</Samplers>
```

Like in the **SparseGridCollocation** sampler, if the covariance matrix is provided and the input parameters are assumed to have the multivariate normal distribution, the **AdaptiveSparseGrid** can be also used. This means one creates the sparse grids of variables listed by `<latentVariables>` in the transformed space. If this is the case, the user needs to provide additional information, i.e. the `<transformation>` under `<MultivariateNormal>` of `<Distributions>` (more information can be found in Section 9.2). In addition, the node `<variablesTransformation>`

is also required for **AdaptiveSparseGrid** sampler. This node is used to tranform the variables specified by **`<latentVariables>`** in the transformed space of input into variables spefified by **`<manifestVariables>`** in the input space. The variables listed in **`<latentVariables>`** should be predefined in **`<variable>`**, and the variables listed in **`<manifestVariables>`** are used by the **`<Models>`**.

- **`<variablesTransformation>`**, ***optional field***. this XML node accepts one attribute:
  - **`distribution`**, ***required string attribute***, the name for the distribution defined in the XML node **`<Distributions>`**. This attribute indicates the values of **`<manifestVariables>`** are drawn from **`distribution`**.

  In addition, this XML node also accepts three childen nodes:
  - **`<latentVariables>`**, ***comma separated string, required field***, user-defined latent variables that are used for the variables transformation. All the variables listed under this node should be also mentioned in **`<variable>`**.
  - **`<manifestVariables>`**, ***comma separated string, required field***, user-defined manifest variables that can be used by the **`model`**.
  - **`<manifestVariablesIndex>`**, ***comma separated string, optional field***, user-defined manifest variables indices paired with **`<manifestVariables>`**. These indices indicate the position of manifest variables associated with multivariate normal distribution defined in the XML node **`<Distributions>`**. The indices should be postive integer. If not provided, the code will use the positions of manifest variables listed in **`<manifestVariables>`** as the indices.
  - **`<method>`**, ***string, required field***, the method that is used for the variables transformation. The currently available method is '**pca**'.

```
...
<Models>
    ...
    <ExternalModel ModuleToLoad="lorentzAttractor_noK"
       name="PythonModule" subType="">
        <variables>sigma,rho,beta,x,y,z,time,x0,y0,z0</variables>
    </ExternalModel>
    <ROM name="gausspolyrom" subType="GaussPolynomialRom">
        <Target>ans</Target>
        <Features>x1,y1,z1</Features>
        <IndexSet>TensorProduct</IndexSet>
        <PolynomialOrder>1</PolynomialOrder>
    </ROM>
    ...
```

```
</Models>

<Distributions>
    ...
    <MultivariateNormal name='MVNDist' method='pca'>
        <transformation>
            <rank>3</rank>
        </transformation>
        <mu>0.0 1.0 2.0</mu>
        <covariance type="abs">
            1.0       0.6      -0.4
            0.6       1.0      0.2
            -0.4      0.2      0.8
        </covariance>
    </MultivariateNormal>
    ...
</Distributions>

<Samplers>
  ...
  <AdaptiveSparseGrid name='ASC'>
        <variable name='x0'>
            <distribution dim='1'>MVNDist</distribution>
        </variable>
        <variable name='y0'>
            <distribution dim='2'>MVNDist</distribution>
        </variable>
        <variable name='z0'>
            <distribution dim='3'>MVNDist</distribution>
        </variable>
        <variablesTransformation model="PythonModule">
            <latentVariables>x1,y1,z1</latentVariables>
            <manifestVariables>x0,y0,z0</manifestVariables>
            <method>pca</method>
        </variablesTransformation>
        <ROM class = 'Models' type = 'ROM'>gausspolyrom</ROM>
        <TargetEvaluation class = 'DataObjects' type =
           'PointSet'>solns</TargetEvaluation>
  </AdaptiveSparseGrid>
  ...
</Samplers>
...
```

10.3.6 Adaptive Sobol Decomposition

The **Adaptive Sobol Decomposition** approach is an advanced methodology that decomposes an uncertainty space into subsets and adaptively includes the most influential ones. For example, for a response function $f(a, b, c)$, the full list of subsets include $(a), (b), (c), (a, b), (a, c), (b, c), (a, b, c)$. A Gauss Polynomial ROM is constructed for each included subset using the Adaptive Sparse Grid sampler. The importance of each subset is estimated based on the importance of preceding subsets; that is, the impact of (a, b) on the representation of f is estimated using the impact of (a) and (b). Because of the excellent performance of Gauss Polynomial ROMs for small-dimension spaces, this sampler used to construct an HDMR ROM can be very efficient. Note that the ROM specified for this sampler *must* be an HDMRRom specified in the Models block.

The specifications of this sampler must be defined within an **`<Adaptive Sobol>`** XML block. This XML node accepts one attribute:

- **`name`**, ***required string attribute***, user-defined name of this sampler. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.

In the **`<Adaptive Sobol>`** input block, the user needs to specify the variables to sample. As already mentioned, these variables are specified within consecutive **`<variable>`** XML blocks:

- **`<variable>`**, ***XML node, required parameter*** can specify the following attribute:
  - **`name`**, ***required string attribute***, user-defined name of this variable.
  - **`shape`**, ***comma-separated integers, optional field***, determines the number of samples and shape of samples to be taken. For example, **`shape`**="2,3" will provide a 2 by 3 matrix of values, while **`shape`**="10" will produce a vector of 10 values. Omitting this optional attribute will result in a single scalar value instead. Each of the values in the matrix or vector will be the same as the single sampled value. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

  This **`<variable>`** recognizes the following child nodes:

  - **`<distribution>`**, ***string, required field***, name of the distribution that is associated to this variable. Its name needs to be contained in the **`<Distributions>`** block explained in Section 9. In addition, if NDDistribution is used, the attribute **`dim`** is required. **Note:** Alternatively, this node must be omitted if the **`<function>`** node is supplied.
  - **`<function>`**, ***string, required field***, name of the function that defines the calculation of this variable from other distributed variables. Its name needs to be contained in

the **<Functions>** block explained in Section 16. This function must implement a method named "evaluate". **Note:** Alternatively, this node must be ommitted if the **<distribution>** node is supplied.

- **<constant>**, ***XML node, optional parameter*** the user is able to input variables that need to be kept constant. For doing this, as many **<constant>** nodes as needed can be input. There are options for setting a constant. To simply set the constant's value, the body of the node contains the constant value, and the **<constant>** node has the following attributes:

  - **name**, ***required string attribute***, user-defined name of this constant.
  - **shape**, ***comma-separated integers, optional field***, determines the shape of samples of the constant value. For example, **shape**="2,3" will shape the values into a 2 by 3 matrix, while **shape**="10" will shape into a vector of 10 values. Unlike the **<variable>**, the constant requires each value be entered; the number of required values is equal to the product of the **shape**. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

  Alternatively, the constant value can be read from a DataObject that has been identified as a **<ConstantSource>** for this Sampler. In this case, the body of the **<constant>** node is the name of the variable that needs to be read from the **<ConstantSource>**, and the **<constant>** node has the following additional attributes in addition to the those above:

  - **source**, ***required string attribute***, the name of the DataObject containing the value to be used for this constant. This must be the name of one of the **<ConstantSource>** DataObjects.
  - **index**, ***optional integer attribute***, the index of the realization in the source DataObject that contains the value to use for the constant.
    Default: the last entry

  By way of example, consider the following Sampler definition. The constant will be named **'C'** in the Sampler, and its value is taken from the DataObject **'MyConstant'**, which is identified in the **<ConstantSource>** node. To find the value of the constant in **'MyConstant'**, the Sampler will look at the realization with index **'3'** for the value of variable **'A'** to use as the constant value.

```
<Samplers>
  <WhatEverSampler name='whatever'>
    <ConstantSource class='DataObjects'
       type='PointSet'>MyConstants</ConstantSource>
    <constant name='C' source='MyConstants'
       index='3'>A</constant>
  </WhatEverSampler>
</Samplers>
```

In addition to the **<variable>** nodes, the main XML node **<AdaptiveSobol>** needs to contain the following supplementary sub-nodes:

- **<Convergence>**, ***required node***, Convergence properties. This node contains the following properties that can be set by sub-nodes:
  - **<relTolerance>**, ***required float***, the relative tolerance to converge. This will compare to the estimate of subset polynomial errors and additional subset polynomials over the variance of the expansion so far to determine convergence.
  - **<maxRuns>**, ***optional integer field***, a limit for the number of model calls. Once this limit is reached, no additional subsets will be generated or considered; however, existing subsets will continue to be trained. If not specified, no limit on solves is imposed.
  - **<maxSobolOrder>**, ***optional integer field***, the largest polynomials orders to use in subset GaussPolynomialRom objects. If specified, polynomial indices with a value larger than the value given will be rejected during adaptive construction.
  - **<progressParam>**, ***optional float field***, a favoritism parameter ranging between 0 and 2. At 0, the algorithm will always prefer adding polynomials to adding new subsets in the HDMR expansion. At 2, the opposite is true. Default is 1.
  - **<logFile>**, ***optional string field***, a file to which adaptive progress is recorded. If specified, each adaptive step will trigger printing progress to the file given, including the estimated error at the step, the next adaptive step to take, the coefficient of each polynomial within each gPC expansion, and the actual and expected Sobol sensitivities of each HDMR subset. Default is no printing.
  - **<subsetVerbosity>**, ***optional string field***, the verbosity for components constructed during the adaptive HDMR process. Options are *silent*, *quiet*, *all*, or *debug*, in order of verbosity. If an invalid entry is provided, will resort to default. Default is *quiet*.

  In summary, this XML node contains the information that is needed in order to control this sampler's convergence criterion.

- **<convergenceStudy>**, ***optional node***, if included, triggers writing state points at particular numbers of model solves for the purpose of a convergence study. The study is performed by writing XML output files as described in the OutStreams for ROMs at the state points requested, using **'all'** as the requested **<what>** values. The state points are identified when a certain number of model runs is passed, as specified by the **<runStatePoints>** node. This node has the following sub-nodes to define its parameters:
  - **<runStatePoints>**, ***list of integers, required node***, lists the number of model runs at which state points should be written. Note that these will be written when the requested number of runs is met or passed, so the actual value is often somewhat more than the requested value, and the exact value will be listed in the XML output.

- **<baseFilename>**, ***string, optional node***, if specified determines the base file name for the state point outputs. If not specified, defaults to **'out_'**.
- **<pickle>**, ***no text, optional node***, if this node is included, serialized (pickled) versions of the ROM at each of the run states is also created in the working directory, with the format `<baseFilename><numRuns>.pk`, such as `out_100.pk`.

Like the **Sobol**, if multivariate normal distribution is provided, the following node need to be specified:

- **<variablesTransformation>**, ***optional field***. this XML node accepts one attribute:
  - **distribution**, ***required string attribute***, the name for the distribution defined in the XML node **<Distributions>**. This attribute indicates the values of **<manifestVariables>** are drawn from **distribution**.

  In addition, this XML node also accepts three childen nodes:
  - **<latentVariables>**, ***comma separated string, required field***, user-defined latent variables that are used for the variables transformation. All the variables listed under this node should be also mentioned in **<variable>**.
  - **<manifestVariables>**, ***comma separated string, required field***, user-defined manifest variables that can be used by the **model**.
  - **<manifestVariablesIndex>**, ***comma separated string, optional field***, user-defined manifest variables indices paired with **<manifestVariables>**. These indices indicate the position of manifest variables associated with multivariate normal distribution defined in the XML node **<Distributions>**. The indices should be postive integer. If not provided, the code will use the positions of manifest variables listed in **<manifestVariables>** as the indices.
  - **<method>**, ***string, required field***, the method that is used for the variables transformation. The currently available method is '**pca**'.

Assembler Objects These objects are either required or optional depending on the functionality of the AdaptiveSobol Sampler. The objects must be listed with a rigorous syntax that, except for the XML node tag, is common among all the objects. Each of these nodes must contain 2 attributes that are used to identify them within the simulation framework:

- **class**, ***required string attribute***, the main "class" of the listed object. For example, it can be **'Models'**, **'Functions'**, etc.
- **type**, ***required string attribute***, the object identifier or sub-type. For example, it can be **'ROM'**, **'External'**, etc.

The **AdaptiveSobol** approach requires or optionally accepts the following object types:

- **<ROM>**, ***string, required field***, the body of this XML node must contain the name of an appropriate ROM defined in the **<Models>** block (see Section 15.3).

- **<TargetEvaluation>**, ***string, required field***, represents the container where the system evaluations are stored. From a practical point of view, this XML node must contain the name of a data object defined in the **<DataObjects>** block (see Section 12). The Adaptive Sobol sampling accepts "DataObjects" of type "PointSet" only.

Example:

```
<Samplers>
  ...
  <AdaptiveSobol name="AS" verbosity='debug'>
    <Convergence>
      <relTolerance>1e-5</relTolerance>
      <maxRuns>150</maxRuns>
      <maxSobolOrder>3</maxSobolOrder>
      <progressParam>1</progressParam>
      <logFile>progress.txt</logFile>
      <subsetVerbosity>silent</subsetVerbosity>
    </Convergence>
    <variable name="x1">
      <distribution>UniDist</distribution>
    </variable>
    <variable name="x2">
      <distribution>UniDist</distribution>
    </variable>
    <ROM class = 'Models' type = 'ROM'>hdmrrom</ROM>
    <TargetEvaluation class = 'DataObjects' type =
       'PointSet'>solns</TargetEvaluation>
  </AdaptiveSobol>
  ...
</Samplers>
```

10.4 Markov Chain Monte Carlo

The Markov chain Monte Carlo (MCMC) is a Sampler entity in the RAVEN framework. It provides enormous scope for realistic statistical modeling. MCMC is essentially Monte Carlo integration using Markov chain. Bayesians, and sometimes also frequentists, need to integrate over possibly high-dimensional probability distributions to make inference about model parameters or to make

predictions. Bayesians need to integrate over the posterior distributions of model parameters given the data, and frequentists may need to integrate over the distribution of observables given parameter values. Monte Carlo integration draws samples from the required distribution, and then forms samples averages to approximate expectations. MCMC draws these samples by running a cleverly constructed Markov chain for a long time. There are a large number of MCMC algorithms, and popular families include Gibbs sampling, Metropolis-Hastings, slice sampling, Hamiltonian Monte Carlo, and many others. Regardless of the algorithm, the goal in Bayesian inference is to maximize the unnormalized joint posterior distribution and collect samples of the target distributions, which are marginal posterior distributions, later to be used for inference.

10.4.1 Metropolis (Metropolis-Hastings Sampler)

The Metropolis-Hastings (MH) algorithm is a MCMC method for obtaining a sequence of random samples from a probability distribution from which direct sampling is difficult. This sequence can be used to approximate the distribution or to compute an integral. It simulates from a probability distribution by making use of the full joint density function and (independent) proposal distributions for each of the variables of interest.

The specifications of this sampler must be defined within an **`<Metropolis>`** XML block. This XML node accepts one attribute:

- **`name`**, ***required string attribute***, user-defined name of this sampler. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.

In the **`<Metropolis>`** input block, the user needs to specify the variables to sample. As already mentioned, these variables are specified within consecutive **`<variable>`** XML blocks:

- **`<variable>`**, ***XML node, required parameter*** can specify the following attribute:
  - **`name`**, ***required string attribute***, user-defined name of this variable.
  - **`shape`**, ***comma-separated integers, optional field***, determines the number of samples and shape of samples to be taken. For example, **`shape`**="2,3" will provide a 2 by 3 matrix of values, while **`shape`**="10" will produce a vector of 10 values. Omitting this optional attribute will result in a single scalar value instead. Each of the values in the matrix or vector will be the same as the single sampled value. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

  This **`<variable>`** recognizes the following child nodes:

- **<distribution>**, ***string, required field***, name of the distribution that is associated to this variable. Its name needs to be contained in the **<Distributions>** block explained in Section 9. In addition, if NDDistribution is used, the attribute **dim** is required. **Note:** Alternatively, this node must be omitted if the **<function>** node is supplied.
- **<function>**, ***string, required field***, name of the function that defines the calculation of this variable from other distributed variables. Its name needs to be contained in the **<Functions>** block explained in Section 16. This function must implement a method named "evaluate". **Note:** Alternatively, this node must be ommitted if the **<distribution>** node is supplied.
- **<initial>**, ***float, optional field***, specified the initial value for given variable.
- **<proposal>**, ***Assembler Object***, specifies the proposal distribution for this variable. **Note:** We only allow one-dimensional symmetric distribution to be the proposal distribution. This node must contain the following two attributes:
  * **class**, ***required string attribute***, the main "class" of the listed object. Only "Distributions" is allowed.
  * **type**, ***required string attribute***, the object identifier or sub-type.
- **<probabilityFunction>**, ***Assembler Object***, specifies the prior distribution function.. This node must contain the following two attributes:
  * **class**, ***required string attribute***, the main "class" of the listed object. Only "Functions" is allowed.
  * **type**, ***required string attribute***, the object identifier or sub-type. Only "External" is allowed.

Note: For MCMC sampler, we only allow "continuous" distributions as input to **<variable>**. In addition, we allow the user to provide their defined prior distribution through the **<probabilityFunct** In this case, the "pdf" method needs to be defined in the external function. For example:

```
def pdf(self):
  """
    Method required for "probabilityFunction" used by MCMC sampler
    that is used to define the prior probability function
    @ In, None
    @ Out, priorPDF, float, the prior pdf value
  """
  priorPDF = 1/(1-self.rho**2)**(3/2)
  return priorPDF
```

- **<constant>**, ***XML node, optional parameter*** the user is able to input variables that need to be kept constant. For doing this, as many **<constant>** nodes as needed can be input. There are options for setting a constant. To simply set the constant's value, the body of the node contains the constant value, and the **<constant>** node has the following attributes:

- **name**, ***required string attribute***, user-defined name of this constant.
- **shape**, ***comma-separated integers, optional field***, determines the shape of samples of the constant value. For example, **shape**="2,3" will shape the values into a 2 by 3 matrix, while **shape**="10" will shape into a vector of 10 values. Unlike the **<variable>**, the constant requires each value be entered; the number of required values is equal to the product of the **shape**. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

Alternatively, the constant value can be read from a DataObject that has been identified as a **<ConstantSource>** for this Sampler. In this case, the body of the **<constant>** node is the name of the variable that needs to be read from the **<ConstantSource>**, and the **<constant>** node has the following additional attributes in addition to the those above:

- **source**, ***required string attribute***, the name of the DataObject containing the value to be used for this constant. This must be the name of one of the **<ConstantSource>** DataObjects.
- **index**, ***optional integer attribute***, the index of the realization in the source DataObject that contains the value to use for the constant.
  Default: the last entry

By way of example, consider the following Sampler definition. The constant will be named **'C'** in the Sampler, and its value is taken from the DataObject **'MyConstant'**, which is identified in the **<ConstantSource>** node. To find the value of the constant in **'MyConstant'**, the Sampler will look at the realization with index **'3'** for the value of variable **'A'** to use as the constant value.

```
<Samplers>
  <WhatEverSampler name='whatever'>
    <ConstantSource class='DataObjects'
       type='PointSet'>MyConstants</ConstantSource>
    <constant name='C' source='MyConstants'
       index='3'>A</constant>
  </WhatEverSampler>
</Samplers>
```

In the **Metropolis** input block, the user needs to specify the variables need to be sampled. As already mentioned, these variables are inputted within consecutive xml blocks called **<variable>**. In addition, the settings for this sampler need to be specified in the **<samplerInit>** XML block:

- **<samplerInit>**, ***required field***. In this xml-node, the following xml sub-nodes need to be specified:

- **<limit>**, ***integer, required field***, number of Metropolis samples needs to be generated;
- **<initialSeed>**, ***integer, optional field***, initial seeding of random number generator;
- **<burnIn>**, ***integer, optional field***, specifies the number of initial samples that would be discarded.
  Default: 0
- **<tune>**, ***bool, optional field***, indicates whether to tune the scaling parameter of proposal distributions or not;
  Default: 'True'
- **<tuneInterval>**, ***integer, optional field***, the number of sample steps for each tuning of scaling parameter;
  Default: 100

In addition to the **<variable>** nodes, the main XML node **<Metropolis>** needs to contain the following supplementary sub-nodes:

- **<likelihood>**, ***string, required node***, the output from the user provided likelihood function This node accept one attribute:
  - **log**, ***bool, optional field***, indicates whether the the log likelihood value is provided or not. When True, the code expects to receive the log likehood value.
    Default: 'False'
- **Assembler Objects** These objects are either required or optional depending on the functionality of the Metropolis Sampler. The objects must be listed with a rigorous syntax that, except for the XML node tag, is common among all the objects. Each of these nodes must contain 2 attributes that are used to identify them within the simulation framework:
  - **class**, ***required string attribute***, the main "class" of the listed object. For example, it can be **'Models'**, **'Functions'**, etc.
  - **type**, ***required string attribute***, the object identifier or sub-type. For example, it can be **'ROM'**, **'External'**, etc.

  The **Metropolis** approach requires or optionally accepts the following object types:

  - **<ConstantSource>**, ***string, optional field***, the body of this XML node must contain the name of an appropriate **DataObject** defined in the **<DataObjects>** block (see Section 12). It is used as a source from which constants can take values.

- **<TargetEvaluation>**, ***string, required field***, represents the container where the system evaluations are stored. From a practical point of view, this XML node must contain the name of a data object defined in the **<DataObjects>** block (see Section 12). The object here specified must be input as **<Output>** in the Steps that employ this sampling strategy. The Metropolis sampling accepts "DataObjects" of type "PointSet" only.

- **<Restart>**, ***string, optional field***, the body of this XML node must contain the name of an appropriate **DataObject** defined in the **<DataObjects>** block (see Section 12). It is used as a "restart" tool, where it accepts pre-existing solutions in the PointSet instead of recalculating solutions.

The following node is an additional option when a restart DataObject is provided:

- **<restartTolerance>**, ***float, optional field***, the body of this XML node must contain a valid floating point value. If a **<Restart>** node is supplied for this **<Sampler>**, this node offers a way to determine how strictly matching points are determined. Given a point in the input space, if that point is within a relative Euclidean distance (equal to the tolerance) of a restart point, the nearest restart point will be used.
  Default: 1e-14

Example:

```
<Samplers>
  ...
  <Metropolis name="Metropolis">
    <samplerInit>
      <limit>1000</limit>
      <initialSeed>070419</initialSeed>
      <tune>10</tune>
    </samplerInit>
    <likelihood log="False">zout</likelihood>
    <variable name="xin">
      <distribution>normal</distribution>
      <initial>0</initial>
      <proposal class="Distributions"
         type="Normal">normal</proposal>
    </variable>
    <variable name="yin">
      <distribution>normal</distribution>
      <initial>0</initial>
      <proposal class="Distributions"
         type="Normal">normal</proposal>
```

```
      <!-- <proposal>normal</proposal> -->
    </variable>
    <TargetEvaluation class="DataObjects"
       type="PointSet">outSet</TargetEvaluation>
  </Metropolis>
  ...
</Samplers>
```

10.4.2 Adaptive Metropolis Sampler

The search for improved proposal distributions of Metropolis sampler is often done manually, through trial and error, though this can be difficult especially in high dimensions. An alternative approach is adaptive Metropolis, which asks the computer to automatically "learn" better parameter values "on the fly".

The specifications of this sampler must be defined within an **`<AdaptiveMetropolis>`** XML block. This XML node accepts one attribute:

- **`name`**, ***required string attribute***, user-defined name of this sampler. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.

In the **`<AdaptiveMetropolis>`** input block, the user needs to specify the variables to sample. As already mentioned, these variables are specified within consecutive **`<variable>`** XML blocks:

- **`<variable>`**, ***XML node, required parameter*** can specify the following attribute:
  - **`name`**, ***required string attribute***, user-defined name of this variable.
  - **`shape`**, ***comma-separated integers, optional field***, determines the number of samples and shape of samples to be taken. For example, **`shape`**="2,3" will provide a 2 by 3 matrix of values, while **`shape`**="10" will produce a vector of 10 values. Omitting this optional attribute will result in a single scalar value instead. Each of the values in the matrix or vector will be the same as the single sampled value. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

  This **`<variable>`** recognizes the following child nodes:

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

explained in Section 9. In addition, if NDDistribution is used, the attribute `dim` is required. **Note:** Alternatively, this node must be omitted if the `<function>` node is supplied.

- `<function>`, ***string, required field***, name of the function that defines the calculation of this variable from other distributed variables. Its name needs to be contained in the `<Functions>` block explained in Section 16. This function must implement a method named "evaluate". **Note:** Alternatively, this node must be ommitted if the `<distribution>` node is supplied.
- `<initial>`, ***float, optional field***, specified the initial value for given variable.
- `<proposal>`, ***Assembler Object***, specifies the proposal distribution for this variable. **Note:** We only allow one-dimensional symmetric distribution to be the proposal distribution. This node must contain the following two attributes:
  * `class`, ***required string attribute***, the main "class" of the listed object. Only "Distributions" is allowed.
  * `type`, ***required string attribute***, the object identifier or sub-type.
  * `dim`, ***positive integer, optional attribute***, required for multivariate normal proposal distribution, indicates the dimension within the multivariate normal distribution that corresponds to this variable.
- `<probabilityFunction>`, ***Assembler Object***, specifies the prior distribution function.. This node must contain the following two attributes:
  * `class`, ***required string attribute***, the main "class" of the listed object. Only "Functions" is allowed.
  * `type`, ***required string attribute***, the object identifier or sub-type. Only "External" is allowed.

THIS NODE IS CURRENTLY NOT ALLOWED FOR ADAPTIVE METROPOLIS SAMPLER.

Note: For this sampler, we only allow "continuous" distributions as input to `<variable>`.

- `<constant>`, ***XML node, optional parameter*** the user is able to input variables that need to be kept constant. For doing this, as many `<constant>` nodes as needed can be input. There are options for setting a constant. To simply set the constant's value, the body of the node contains the constant value, and the `<constant>` node has the following attributes:
  - `name`, ***required string attribute***, user-defined name of this constant.
  - `shape`, ***comma-separated integers, optional field***, determines the shape of samples of the constant value. For example, `shape`="2,3" will shape the values into a 2 by 3 matrix, while `shape`="10" will shape into a vector of 10 values. Unlike the `<variable>`, the constant requires each value be entered; the number of required values is equal to the product of the `shape`. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

Alternatively, the constant value can be read from a DataObject that has been identified as a **<ConstantSource>** for this Sampler. In this case, the body of the **<constant>** node is the name of the variable that needs to be read from the **<ConstantSource>**, and the **<constant>** node has the following additional attributes in addition to the those above:

- **source**, ***required string attribute***, the name of the DataObject containing the value to be used for this constant. This must be the name of one of the **<ConstantSource>** DataObjects.
- **index**, ***optional integer attribute***, the index of the realization in the source DataObject that contains the value to use for the constant.
  Default: the last entry

By way of example, consider the following Sampler definition. The constant will be named **'C'** in the Sampler, and its value is taken from the DataObject **'MyConstant'**, which is identified in the **<ConstantSource>** node. To find the value of the constant in **'MyConstant'**, the Sampler will look at the realization with index **'3'** for the value of variable **'A'** to use as the constant value.

```
<Samplers>
  <WhatEverSampler name='whatever'>
    <ConstantSource class='DataObjects'
       type='PointSet'>MyConstants</ConstantSource>
    <constant name='C' source='MyConstants'
       index='3'>A</constant>
  </WhatEverSampler>
</Samplers>
```

In the **AdaptiveMetropolis** input block, the user needs to specify the variables need to be sampled. As already mentioned, these variables are inputted within consecutive xml blocks called **<variable>**. In addition, the settings for this sampler need to be specified in the **<samplerInit>** XML block:

- **<samplerInit>**, ***required field***. In this xml-node, the following xml sub-nodes need to be specified:
  - **<limit>**, ***integer, required field***, number of Metropolis samples needs to be generated;
  - **<initialSeed>**, ***integer, optional field***, initial seeding of random number generator;
  - **<burnIn>**, ***integer, optional field***, specifies the number of initial samples that would be discarded.
    Default: 0

- **<tune>**, ***bool, optional field***, indicates whether to tune the scaling parameter of proposal distributions or not;
  Default: 'True'
- **<tuneInterval>**, ***integer, optional field***, the number of sample steps for each tuning of scaling parameter;
  Default: 100
- **<adaptiveInterval>**, ***integer, optional field***, the number of sample steps for each proposal parameters update;
  Default: 20

In addition to the **<variable>** nodes, the main XML node **<AdaptiveMetropolis>** needs to contain the following supplementary sub-nodes:

- **<likelihood>**, ***string, required node***, the output from the user provided likelihood function This node accept one attribute:
  - **log**, ***bool, optional field***, indicates whether the the log likelihood value is provided or not. When True, the code expects to receive the log likehood value.
    Default: 'False'
- **Assembler Objects** These objects are either required or optional depending on the functionality of the Metropolis Sampler. The objects must be listed with a rigorous syntax that, except for the XML node tag, is common among all the objects. Each of these nodes must contain 2 attributes that are used to identify them within the simulation framework:
  - **class**, ***required string attribute***, the main "class" of the listed object. For example, it can be **'Models'**, **'Functions'**, etc.
  - **type**, ***required string attribute***, the object identifier or sub-type. For example, it can be **'ROM'**, **'External'**, etc.

  The **Metropolis** approach requires or optionally accepts the following object types:

  - **<ConstantSource>**, ***string, optional field***, the body of this XML node must contain the name of an appropriate **DataObject** defined in the **<DataObjects>** block (see Section 12). It is used as a source from which constants can take values.
  - **<TargetEvaluation>**, ***string, required field***, represents the container where the system evaluations are stored. From a practical point of view, this XML node must contain the name of a data object defined in the **<DataObjects>** block (see Section 12). The object here specified must be input as **<Output>** in the Steps that employ this sampling strategy. The Adaptive Metropolis sampling accepts "DataObjects" of type "PointSet" only.

– **<Restart>**, ***string, optional field***, the body of this XML node must contain the name of an appropriate **DataObject** defined in the **<DataObjects>** block (see Section 12). It is used as a "restart" tool, where it accepts pre-existing solutions in the PointSet instead of recalculating solutions.

The following node is an additional option when a restart DataObject is provided:

– **<restartTolerance>**, ***float, optional field***, the body of this XML node must contain a valid floating point value. If a **<Restart>** node is supplied for this **<Sampler>**, this node offers a way to determine how strictly matching points are determined. Given a point in the input space, if that point is within a relative Euclidean distance (equal to the tolerance) of a restart point, the nearest restart point will be used.
Default: 1e-14

Example:

```
<Samplers>
  ...
  <AdaptiveMetropolis name="AdaptiveMetropolis">
    <samplerInit>
      <limit>1000</limit>
      <initialSeed>070419</initialSeed>
      <burnIn>500</burnIn>
    </samplerInit>
    <likelihood log="False">zout</likelihood>
    <variable name="xin">
      <distribution>normal</distribution>
      <initial>2</initial>
    </variable>
    <variable name="yin">
      <distribution>normal</distribution>
      <initial>2</initial>
    </variable>
    <TargetEvaluation class="DataObjects"
       type="PointSet">outSet</TargetEvaluation>
  </AdaptiveMetropolis>
  ...
</Samplers>
```

11 Optimizers

The optimizer is another important entity in the RAVEN framework. It performs the driving of a specific "goal function" or "objective function" over the model for value optimization. The Optimizer can be used almost anywhere a Sampler can be used, and is only distinguished from other AdaptiveSampler strategies for clarity.

11.1 GradientDescent

The **<GradientDescent>** optimizer represents an a la carte option for performing gradient-based optimization with a variety of gradient estimation techniques, stepping strategies, and acceptance criteria. Gradient descent optimization generally behaves as a ball rolling down a hill; the algorithm estimates the local gradient at a point, and attempts to move "downhill" in the opposite direction of the gradient (if minimizing; the opposite if maximizing). Once the lowest point along the iterative gradient search is discovered, the algorithm is considered converged. Note that gradient descent algorithms are particularly prone to being trapped in local minima; for this reason, depending on the model, multiple trajectories may be needed to obtain the global solution.

When used as part of a **<MultiRun>** step, this entity provides additional information through the **<SolutionExport>** DataObject. The following variables can be requested within the **<SolutionExport>**:

- `trajID`: integer identifier for different optimization starting locations and paths
- `iteration`: integer identifying which iteration (or step, or generation) a trajectory is on
- `accepted`: string acceptance status of the potential optimal point (algorithm dependent)
- `rejectReason`: discription of reject reason, 'noImprovement' means rejected the new optimization point for no improvement from last point, 'implicitConstraintsViolation' means rejected by implicit constraints violation, return None if the point is accepted
- {`VAR`}: any variable from the **<TargetEvaluation>** input or output; gives the value of that variable at the optimal candidate for this iteration.
- `stepSize`: the size of step taken in the normalized input space to arrive at each optimal point
- `conv_`{`CONV`}: status of each given convergence criteria
- `CG_task`: for ConjugateGradient, current task of line search. FD suggests continuing the search, and CONV indicates the line search converged and will pivot.

The **<GradientDescent>** node recognizes the following parameters:

- **verbosity**: ***[silent, quiet, all, debug], optional***, Desired verbosity of messages coming from this entity
- **name**: ***string, required***, User-defined name to designate this entity in the RAVEN input file.

The **<GradientDescent>** node recognizes the following subnodes:

- **<objective>**: ***string***, Name of the response variable (or "objective function") that should be optimized (minimized or maximized).
- **<variable>**: defines the input space variables to be sampled through various means. The **<variable>** node recognizes the following parameters:
  - **name**: ***string, optional***, user-defined name of this Sampler. **Note:** As for the other objects, this is the name that can be used to refer to this specific entity from other input blocks
  - **shape**: ***comma-separated integers, optional***, determines the number of samples and shape of samples to be taken. For example, **shape**="2,3" will provide a 2 by 3 matrix of values, while **shape**="10" will produce a vector of 10 values. Omitting this optional attribute will result in a single scalar value instead. Each of the values in the matrix or vector will be the same as the single sampled value. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

  The **<variable>** node recognizes the following subnodes:

  - **<distribution>**: ***string***, name of the distribution that is associated to this variable. Its name needs to be contained in the **<Distributions>** block explained in Section 9. In addition, if NDDistribution is used, the attribute **dim** is required. **Note:** Alternatively, this node must be omitted if the **<function>** node is supplied. The **<distribution>** node recognizes the following parameters:
    * **dim**: ***integer, optional***, for an NDDistribution, indicates the dimension within the NDDistribution that corresponds to this variable.
  - **<grid>**: ***string***, – no description yet – The **<grid>** node recognizes the following parameters:
    * **type**: ***string, optional***, – no description yet –
    * **construction**: ***string, optional***, – no description yet –
    * **steps**: ***integer, optional***, – no description yet –

- **<function>**: ***string***, name of the function that defines the calculation of this variable from other distributed variables. Its name needs to be contained in the **<Functions>** block explained in Section 16. This function must implement a method named "evaluate". **Note:** Each **<variable>** must contain only one **<Function>** or **<Distribution>**, but not both.
- **<initial>**: ***comma-separated floats***, indicates the initial values where independent trajectories for this optimization effort should begin. The number of entries should be the same for all variables, unless a variable is initialized with a sampler (see **<samplerInit>** below). Note these entries are ordered; that is, if the optimization variables are x and y, and the initial values for x are **'1, 2, 3, 4'** and initial values for y are **'5, 6, 7, 8'**, then there will be four starting trajectories beginning at the locations (1, 5), (2, 6), (3, 7), and (4, 8).

- **<TargetEvaluation>**: ***string***, name of the DataObject where the sampled outputs of the Model will be collected. This DataObject is the means by which the sampling entity obtains the results of requested samples, and so should require all the input and output variables needed for adaptive sampling. The **<TargetEvaluation>** node recognizes the following parameters:
  - **class**: ***string, required***, RAVEN class for this entity (e.g. Samplers, Models, DataObjects)
  - **type**: ***string, required***, RAVEN type for this entity; a subtype of the class (e.g. MonteCarlo, Code, PointSet)

- **<samplerInit>**: collection of nodes that describe the initialization of the optimization algorithm.

  The **<samplerInit>** node recognizes the following subnodes:
  - **<limit>**: ***integer***, limits the number of Model evaluations that may be performed as part of this optimization. For example, a limit of 100 means at most 100 total Model evaluations may be performed.
  - **<writeSteps>**: ***[final, every]***, delineates when the **<SolutionExport>** DataObject should be written to. In case of **'final'**, only the final optimal solution for each trajectory will be written. In case of **'every'**, the **<SolutionExport>** will be updated with each iteration of the Optimizer.
  - **<initialSeed>**: ***integer***, seed for random number generation. Note that by default RAVEN uses an internal seed, so this seed must be changed to observe changed behavior.
    Default: RAVEN-determined
  - **<type>**: ***[min, max]***, the type of optimization to perform. **'min'** will search for the lowest **<objective>** value, while **'max'** will search for the highest value.

- **<gradient>**: a required node containing the information about which gradient approximation algorithm to use, and its settings if applicable. Exactly one of the gradient approximation algorithms below may be selected for this Optimizer.

  The **<gradient>** node recognizes the following subnodes:

  - **<FiniteDifference>**: if node is present, indicates that gradient approximation should be performed using Finite Difference approximation. Finite difference makes use of orthogonal perturbations in each dimension of the input space to estimate the local gradient, requiring a total of N perturbations, where N is dimensionality of the input space. For example, if the input space $\mathbf{i} = (x, y, z)$ for objective function $f(\mathbf{i})$, then FiniteDifference chooses three perturbations (α, β, γ) and evaluates the following perturbation points:

    * $f(x + \alpha, y, z)$,
    * $f(x, y + \beta, z)$,
    * $f(x, y, z + \gamma)$

    and evaluates the gradient $\nabla f = (\nabla^{(x)} f, \nabla^{(y)} f, \nabla^{(z)} f)$ as

    $$\nabla^{(x)} f \approx \frac{f(x + \alpha, y, z) - f(x, y, z)}{\alpha},$$

    and so on for $\nabla^{(y)} f$ and $\nabla^{(z)} f$.

  - **<CentralDifference>**: if node is present, indicates that gradient approximation should be performed using Central Difference approximation. Central difference makes use of pairs of orthogonal perturbations in each dimension of the input space to estimate the local gradient, requiring a total of $2N$ perturbations, where N is dimensionality of the input space. For example, if the input space $\mathbf{i} = (x, y, z)$ for objective function $f(\mathbf{i})$, then CentralDifference chooses three perturbations (α, β, γ) and evaluates the following perturbation points:

    * $f(x \pm \alpha, y, z)$,
    * $f(x, y \pm \beta, z)$,
    * $f(x, y, z \pm \gamma)$

    and evaluates the gradient $\nabla f = (\nabla^{(x)} f, \nabla^{(y)} f, \nabla^{(z)} f)$ as

    $$\nabla^{(x)} f \approx \frac{f(x + \alpha, y, z) - f(x - \alpha, y, z)}{2\alpha},$$

    and so on for $\nabla^{(y)} f$ and $\nabla^{(z)} f$.

  - **<SPSA>**: if node is present, indicates that gradient approximation should be performed using the Simultaneous Perturbation Stochastic Approximation (SPSA). SPSA makes use of a single perturbation as a zeroth-order gradient approximation, requiring exactly 1 perturbation regardless of the dimensionality of the input space. For example, if the input space $\mathbf{i} = (x, y, z)$ for objective function $f(\mathbf{i})$, then SPSA chooses a single perturbation point $(\epsilon^{(x)}, \epsilon^{(y)}, \epsilon^{(z)})$ and evaluates the following perturbation point:

* $f(x+\epsilon^{(x)}, y+\epsilon^{(y)}, z+\epsilon^{(z)})$

and evaluates the gradient $\nabla f = (\nabla^{(x)} f, \nabla^{(y)} f, \nabla^{(z)} f)$ as

$$\nabla^{(x)} f \approx \frac{f(x+\epsilon^{(x)}, y+\epsilon^{(y)}, z+\epsilon^{(z)})) - f(x,y,z)}{\epsilon^{(x)}},$$

and so on for $\nabla^{(y)} f$ and $\nabla^{(z)} f$. This approximation is much less robust than FiniteDifference or CentralDifference, but has the benefit of being dimension agnostic.

- **`<stepSize>`**: a required node containing the information about which iterative stepping algorithm to use, and its settings if applicable. Exactly one of the stepping algorithms below may be selected for this Optimizer.

  The **`<stepSize>`** node recognizes the following subnodes:

  - **`<GradientHistory>`**: if this node is present, indicates that the iterative steps in the gradient descent algorithm should be determined by the sequential change in gradient. In particular, rather than using the magnitude of the gradient to determine step size, the directional change of the gradient versor determines whether to take larger or smaller steps. If the gradient in two successive steps changes direction, the step size shrinks. If the gradient instead continues in the same direction, the step size grows. The rate of shrink and growth are controlled by the **`<shrinkFactor>`** and **`<growthFactor>`**. Note these values have a large impact on the optimization path taken. Large growth factors converge slowly but explore more of the input space; large shrink factors converge quickly but might converge before arriving at a local minimum.

    The **`<GradientHistory>`** node recognizes the following subnodes:

    * **`<growthFactor>`**: ***float***, specifies the rate at which the step size should grow if the gradient continues in same direction through multiple iterative steps. For example, a growth factor of 2 means that if the gradient is identical twice, the step size is doubled.
      Default: 1.25
    * **`<shrinkFactor>`**: ***float***, specifies the rate at which the step size should shrink if the gradient changes direction through multiple iterative steps. For example, a shrink factor of 2 means that if the gradient completely flips direction, the step size is halved. Note that for stochastic surfaces or low-order gradient approximations such as SPSA, a small value for the shrink factor is recommended. If an optimization path appears to be converging early, increasing the shrink factor might improve the search.
      Default: 1.15

  - **`<ConjugateGradient>`**: Base class for Step Manipulation algorithms in the GradientDescent Optimizer.

- **`<acceptance>`**: a required node containing the information about the acceptability criterion for iterative optimization steps, i.e. when a potential new optimal point should be

rejected and when it can be accepted. Exactly one of the acceptance criteria below may be selected for this Optimizer.

The **<acceptance>** node recognizes the following subnodes:

- **<Strict>**: if this node is present, indicates that a Strict acceptance policy for potential new optimal points should be enforced; that is, for a potential optimal point to become the new point from which to take another iterative optimizer step, the new response value must be improved over the old response value. Otherwise, the potential opt point is rejected and the search continues with the previously-discovered optimal point.

- **<convergence>**: a node containing the desired convergence criteria for the optimization algorithm. Note that convergence is met when any one of the convergence criteria is met. If no convergence criteria are given, then nominal convergence on gradient value is used.

  The **<convergence>** node recognizes the following subnodes:

  - **<gradient>**: ***float***, provides the desired value for the local estimated of the gradient for convergence.
    Default: 1e-6, if no criteria specified
  - **<objective>**: ***float***, provides the maximum relative change in the objective function for convergence.
  - **<stepSize>**: ***float***, provides the maximum size in relative step size for convergence.
  - **<terminateFollowers>**: ***[True, Yes, 1, False, No, 0, t, y, 1, f, n, 0]***, indicates whether a trajectory should be terminated when it begins following the path of another trajectory. The **<terminateFollowers>** node recognizes the following parameters:
    - **proximity**: ***float, optional***, provides the normalized distance at which a trajectory's head should be proximal to another trajectory's path before terminating the following trajectory.
  - **<persistence>**: ***integer***, provides the number of consecutive times convergence should be reached before a trajectory is considered fully converged. This helps in preventing early false convergence.
  - **<constraintExplorationLimit>**: ***integer***, provides the number of consecutive times a functional constraint boundary can be explored for an acceptable sampling point before aborting search. Only apples if using a **<Constraint>**.
    Default: 500

- **<constant>**: ***comma-separated strings, integers, and floats***, allows variables that do not change value to be part of the input space. The **<constant>** node recognizes the following parameters:

- **name**: ***string, required***, variable name for this constant, which will be provided to the Model.
- **shape**: ***comma-separated integers, optional***, determines the shape of samples of the constant value. For example, **shape**="2,3" will shape the values into a 2 by 3 matrix, while **shape**="10" will shape into a vector of 10 values. Unlike the **<variable>**, the constant requires each value be entered; the number of required values is equal to the product of the **shape** values, e.g. 6 entries for shape "2,3"). **Note:** A model interface must be prepared to handle non- scalar inputs to use this option.
- **source**: ***string, optional***, the name of the DataObject containing the value to be used for this constant. Requires **<ConstantSource>** node with a **<DataObject>** identified for this Sampler/Optimizer.
- **index**: ***integer, optional***, the index of the realization in the **<ConstantSource>** **<DataObject>** containing the value for this constant. Requires **<ConstantSource>** node with a **<DataObject>** identified for this Sampler/Optimizer.

- **<ConstantSource>**: ***string***, identifies a **<DataObject>** to provide **<constant>** values to the input space of this entity while sampling. As an alternative to providing predefined values for constants, the **<ConstantSource>** provides a dynamic means of always providing the same value for a constant. This is often used as part of a larger multi-workflow calculation. The **<ConstantSource>** node recognizes the following parameters:
  - **class**: ***string, optional***, The RAVEN class for this source. Options include **'DataObject'**.
  - **type**: ***string, optional***, The RAVEN type for this source. Options include any valid **<DataObject>** type, such as HistorySet or PointSet.

- **<Constraint>**: ***string***, name of **<Function>** which contains explicit constraints for the sampling of the input space of the Model. From a practical point of view, this XML node must contain the name of a function defined in the **<Functions>** block (see Section 16). This external function must contain a method called "constrain", which returns True for inputs satisfying the explicit constraints and False otherwise. The **<Constraint>** node recognizes the following parameters:
  - **class**: ***string, required***, RAVEN class for this source. Options include **'Functions'**.
  - **type**: ***string, required***, RAVEN type for this source. Options include **<External>**.

- **<ImplicitConstraint>**: ***string***, name of **<Function>** which contains implicit constraints of the Model. From a practical point of view, this XML node must contain the name of a function defined in the **<Functions>** block (see Section 16). This external function must contain a method called "implicitConstrain", which returns True for outputs satisfying the implicit constraints and False otherwise. The **<ImplicitConstraint>** node recognizes the following parameters:
  - **class**: ***string, required***, RAVEN class for this source. Options include **'Functions'**.

- **type**: ***string, required***, RAVEN type for this source. Options include **<External>**.

- **<Sampler>**: ***string***, name of a Sampler that can be used to initialize the starting points for the trajectories of some of the variables. From a practical point of view, this XML node must contain the name of a Sampler defined in the **<Samplers>** block (see Section 10.1). The Sampler will be used to initialize the trajectories' initial points for some or all of the variables. For example, if the Sampler selected samples only 2 of the 5 optimization variables, the **<initial>** XML node is required only for the remaining 3 variables. The **<Sampler>** node recognizes the following parameters:

  - **class**: ***string, required***, RAVEN class for this entity (e.g. Samplers, Models, DataObjects)
  - **type**: ***string, required***, RAVEN type for this entity; a subtype of the class (e.g. MonteCarlo, Code, PointSet)

- **<Restart>**: ***string***, name of a DataObject. Used to leverage existing data when sampling a model. For example, if a Model has already been sampled, but some samples were not collected, the successful samples can be stored and used instead of rerunning the model for those specific samples. This RAVEN entity definition must be a DataObject with contents including the input and output spaces of the Model being sampled. The **<Restart>** node recognizes the following parameters:

  - **class**: ***string, optional***, The RAVEN class for this source. Options include **'DataObject'**.
  - **type**: ***string, optional***, The RAVEN type for this source. Options include any valid **<DataObject>** type, such as HistorySet or PointSet.

- **<restartTolerance>**: ***float***, specifies how strictly a matching point from a **<Restart>** DataObject must match the desired sample point in order to be used. If a potential restart point is within a relative Euclidean distance (as specified by the value in this node) of a desired sample point, the restart point will be used instead of sampling the Model.
  Default: 1e-15

- **<variablesTransformation>**: Allows transformation of variables via translation matrices. This defines two spaces, a "latent" transformed space sampled by RAVEN and a "manifest" original space understood by the Model. The **<variablesTransformation>** node recognizes the following parameters:

  - **distribution**: ***string, optional***, the name for the distribution defined in the XML node **<Distributions>**. This attribute indicates the values of **<manifestVariables>** are drawn from **distribution**.

  The **<variablesTransformation>** node recognizes the following subnodes:

  - **<latentVariables>**: ***comma-separated strings***, user-defined latent variables that are used for the variables transformation. All the variables listed under this node should be also mentioned in **<variable>**.

- **<manifestVariables>**: ***comma-separated strings***, user-defined manifest variables that can be used by the **<Model>**.
- **<manifestVariablesIndex>**: ***comma-separated strings***, user-defined manifest variables indices paired with **<manifestVariables>**. These indices indicate the position of manifest variables associated with multivariate normal distribution defined in the XML node **<Distributions>**. The indices should be postive integer. If not provided, the code will use the positions of manifest variables listed in **<manifestVariables>** as the indices.
- **<method>**: ***string***, the method that is used for the variables transformation. The currently available method is **'pca'**.

Gradient Descent Example:

```
<Optimizers>
  ...
  <GradientDescent name="opter">
    <objective>ans</objective>
    <variable name="x">
      <distribution>x_dist</distribution>
      <initial>-2</initial>
    </variable>
    <variable name="y">
      <distribution>y_dist</distribution>
      <initial>2</initial>
    </variable>
    <samplerInit>
      <limit>100</limit>
    </samplerInit>
    <gradient>
      <FiniteDifference/>
    </gradient>
    <stepSize>
      <GradientHistory/>
    </stepSize>
    <acceptance>
      <Strict/>
    </acceptance>
    <TargetEvaluation class="DataObjects"
       type="PointSet">optOut</TargetEvaluation>
  </GradientDescent>
  ...
</Optimizers>
```

11.2 SimulatedAnnealing

The **<SimulatedAnnealing>** optimizer is a metaheuristic approach to perform a global search in large design spaces. The methodology rose from statistical physics and was inspitred by metallurgy where it was found that fast cooling might lead to smaller and defected crystals, and that reheating and slowly controling cooling will lead to better states. This allows climbing to avoid being stuck in local minima and hence facilitates finding the global minima for non-convex probloems. More information can be found in: Kirkpatrick, S.; Gelatt Jr, C. D.; Vecchi, M. P. (1983). "Optimization by Simulated Annealing". Science. 220 (4598): 671–680.

When used as part of a **<MultiRun>** step, this entity provides additional information through the **<SolutionExport>** DataObject. The following variables can be requested within the **<SolutionExport>**:

- `trajID`: integer identifier for different optimization starting locations and paths
- `iteration`: integer identifying which iteration (or step, or generation) a trajectory is on
- `accepted`: string acceptance status of the potential optimal point (algorithm dependent)
- `rejectReason`: discription of reject reason, 'noImprovement' means rejected the new optimization point for no improvement from last point, 'implicitConstraintsViolation' means rejected by implicit constraints violation, return None if the point is accepted
- {`VAR`}: any variable from the **<TargetEvaluation>** input or output; gives the value of that variable at the optimal candidate for this iteration.
- `conv_`{`CONV`}: status of each given convergence criteria
- `amp_`{`VAR`}: amplitued associated to each variable used to compute step size based on cooling method and the corresponding next neighbour
- `delta_`{`VAR`}: step size associated to each variable
- `Temp`: temperature at current state
- `fraction`: current fraction of the max iteration limit

The **<SimulatedAnnealing>** node recognizes the following parameters:

- **verbosity**: ***[silent, quiet, all, debug], optional***, Desired verbosity of messages coming from this entity
- **name**: ***string, required***, User-defined name to designate this entity in the RAVEN input file.

The **<SimulatedAnnealing>** node recognizes the following subnodes:

- **<objective>**: ***string***, Name of the response variable (or "objective function") that should be optimized (minimized or maximized).
- **<variable>**: defines the input space variables to be sampled through various means. The **<variable>** node recognizes the following parameters:
  - **name**: ***string, optional***, user-defined name of this Sampler. **Note:** As for the other objects, this is the name that can be used to refer to this specific entity from other input blocks
  - **shape**: ***comma-separated integers, optional***, determines the number of samples and shape of samples to be taken. For example, **shape**="2,3" will provide a 2 by 3 matrix of values, while **shape**="10" will produce a vector of 10 values. Omitting this optional attribute will result in a single scalar value instead. Each of the values in the matrix or vector will be the same as the single sampled value. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

  The **<variable>** node recognizes the following subnodes:

  - **<distribution>**: ***string***, name of the distribution that is associated to this variable. Its name needs to be contained in the **<Distributions>** block explained in Section 9. In addition, if NDDistribution is used, the attribute **dim** is required. **Note:** Alternatively, this node must be omitted if the **<function>** node is supplied. The **<distribution>** node recognizes the following parameters:
    - **dim**: ***integer, optional***, for an NDDistribution, indicates the dimension within the NDDistribution that corresponds to this variable.
  - **<grid>**: ***string***, – no description yet – The **<grid>** node recognizes the following parameters:
    - **type**: ***string, optional***, – no description yet –
    - **construction**: ***string, optional***, – no description yet –
    - **steps**: ***integer, optional***, – no description yet –
  - **<function>**: ***string***, name of the function that defines the calculation of this variable from other distributed variables. Its name needs to be contained in the **<Functions>** block explained in Section 16. This function must implement a method named "evaluate". **Note:** Each **<variable>** must contain only one **<Function>** or **<Distribution>**, but not both.
  - **<initial>**: ***comma-separated floats***, indicates the initial values where independent trajectories for this optimization effort should begin. The number of entries should be the same for all variables, unless a variable is initialized with a sampler (see **<samplerInit>** below). Note these entries are ordered; that is, if the optimization variables are x and y,

and the initial values for x are **'1, 2, 3, 4'** and initial values for y are **'5, 6, 7, 8'**, then there will be four starting trajectories beginning at the locations (1, 5), (2, 6), (3, 7), and (4, 8).

- **<TargetEvaluation>**: ***string***, name of the DataObject where the sampled outputs of the Model will be collected. This DataObject is the means by which the sampling entity obtains the results of requested samples, and so should require all the input and output variables needed for adaptive sampling. The **<TargetEvaluation>** node recognizes the following parameters:
  - **class**: ***string, required***, RAVEN class for this entity (e.g. Samplers, Models, DataObjects)
  - **type**: ***string, required***, RAVEN type for this entity; a subtype of the class (e.g. MonteCarlo, Code, PointSet)
- **<samplerInit>**: collection of nodes that describe the initialization of the optimization algorithm.

  The **<samplerInit>** node recognizes the following subnodes:
  - **<limit>**: ***integer***, limits the number of Model evaluations that may be performed as part of this optimization. For example, a limit of 100 means at most 100 total Model evaluations may be performed.
  - **<writeSteps>**: ***[final, every]***, delineates when the **<SolutionExport>** DataObject should be written to. In case of **'final'**, only the final optimal solution for each trajectory will be written. In case of **'every'**, the **<SolutionExport>** will be updated with each iteration of the Optimizer.
  - **<initialSeed>**: ***integer***, seed for random number generation. Note that by default RAVEN uses an internal seed, so this seed must be changed to observe changed behavior.
    Default: RAVEN-determined
  - **<type>**: ***[min, max]***, the type of optimization to perform. **'min'** will search for the lowest **<objective>** value, while **'max'** will search for the highest value.
- **<convergence>**: a node containing the desired convergence criteria for the optimization algorithm. Note that convergence is met when any one of the convergence criteria is met. If no convergence criteria are given, then the defaults are used.

  The **<convergence>** node recognizes the following subnodes:
  - **<objective>**: ***float***, provides the desired value for the convergence criterion of the objective function (ϵ^{obj}), i.e., convergence is reached when:

    $$|newObjevtive - oldObjective| \leq \epsilon^{obj}$$

    .
    Default: 1e-6, if no criteria specified

- **<temperature>**: ***float***, provides the desired value for the convergence creiteron of the system temperature, (ϵ^{temp}), i.e., convergence is reached when:

$$T \leq \epsilon^{temp}$$

.
Default: 1e-10, if no criteria specified

- **<persistence>**: ***integer***, provides the number of consecutive times convergence should be reached before a trajectory is considered fully converged. This helps in preventing early false convergence.

- **<coolingSchedule>**: The function governing the cooling process. Currently, user can select between,**'exponential'**, **'cauchy'**, **'boltzmann'**,or **'veryfast'**.

  In case of **'exponential'** is provided, The cooling process will be governed by:

$$T^k = T^0 * \alpha^k$$

  In case of **'boltzmann'** is provided, The cooling process will be governed by:

$$T^k = \frac{T^0}{log(k+d)}$$

  In case of **'cauchy'** is provided, The cooling process will be governed by:

$$T^k = \frac{T^0}{k+d}$$

  In case of **'veryfast'** is provided, The cooling process will be governed by:

$$T^k = T^0 * \exp(-ck^{1/D}),$$

  where D is the dimentionality of the problem (i.e., number of optimized variables), k is the number of the current iteration $T^0 = \max(0.01, 1 - \frac{k}{\texttt{<limit>}})$ is the initial temperature, and T^k is the current temperature according to the specified cooling schedule.
  Default: exponential.

  The **<coolingSchedule>** node recognizes the following subnodes:

  - **<exponential>**: ***string***, exponential cooling schedule
    The **<exponential>** node recognizes the following subnodes:
    * **<alpha>**: ***float***, slowing down constant, should be between 0,1 and preferable very close to 1.
      Default: 0.94
  - **<veryfast>**: ***string***, veryfast cooling schedule
    The **<veryfast>** node recognizes the following subnodes:

    * **<c>**: ***float***, decay constant,
      Default: 1.0

  - **<cauchy>**: ***string***, cauchy cooling schedule

    The **<cauchy>** node recognizes the following subnodes:

    * **<d>**: ***float***, bias,
      Default: 1.0

  - **<boltzmann>**: ***string***, boltzmann cooling schedule

    The **<boltzmann>** node recognizes the following subnodes:

    * **<d>**: ***float***, bias,
      Default: 1.0

- **<constant>**: ***comma-separated strings, integers, and floats***, allows variables that do not change value to be part of the input space. The **<constant>** node recognizes the following parameters:

  - **name**: ***string, required***, variable name for this constant, which will be provided to the Model.
  - **shape**: ***comma-separated integers, optional***, determines the shape of samples of the constant value. For example, **shape**="2,3" will shape the values into a 2 by 3 matrix, while **shape**="10" will shape into a vector of 10 values. Unlike the **<variable>**, the constant requires each value be entered; the number of required values is equal to the product of the **shape** values, e.g. 6 entries for shape "2,3"). **Note:** A model interface must be prepared to handle non- scalar inputs to use this option.
  - **source**: ***string, optional***, the name of the DataObject containing the value to be used for this constant. Requires **<ConstantSource>** node with a **<DataObject>** identified for this Sampler/Optimizer.
  - **index**: ***integer, optional***, the index of the realization in the **<ConstantSource>** **<DataObject>** containing the value for this constant. Requires **<ConstantSource>** node with a **<DataObject>** identified for this Sampler/Optimizer.

- **<ConstantSource>**: ***string***, identifies a **<DataObject>** to provide **<constant>** values to the input space of this entity while sampling. As an alternative to providing predefined values for constants, the **<ConstantSource>** provides a dynamic means of always providing the same value for a constant. This is often used as part of a larger multi-workflow calculation. The **<ConstantSource>** node recognizes the following parameters:

  - **class**: ***string, optional***, The RAVEN class for this source. Options include **'DataObject'**.
  - **type**: ***string, optional***, The RAVEN type for this source. Options include any valid **<DataObject>** type, such as HistorySet or PointSet.

- **<Constraint>**: ***string***, name of **<Function>** which contains explicit constraints for the sampling of the input space of the Model. From a practical point of view, this XML node must contain the name of a function defined in the **<Functions>** block (see Section 16). This external function must contain a method called "constrain", which returns True for inputs satisfying the explicit constraints and False otherwise. The **<Constraint>** node recognizes the following parameters:
  - **class**: ***string, required***, RAVEN class for this source. Options include **'Functions'**.
  - **type**: ***string, required***, RAVEN type for this source. Options include **<External>**.
- **<ImplicitConstraint>**: ***string***, name of **<Function>** which contains implicit constraints of the Model. From a practical point of view, this XML node must contain the name of a function defined in the **<Functions>** block (see Section 16). This external function must contain a method called "implicitConstrain", which returns True for outputs satisfying the implicit constraints and False otherwise. The **<ImplicitConstraint>** node recognizes the following parameters:
  - **class**: ***string, required***, RAVEN class for this source. Options include **'Functions'**.
  - **type**: ***string, required***, RAVEN type for this source. Options include **<External>**.
- **<Sampler>**: ***string***, name of a Sampler that can be used to initialize the starting points for the trajectories of some of the variables. From a practical point of view, this XML node must contain the name of a Sampler defined in the **<Samplers>** block (see Section 10.1). The Sampler will be used to initialize the trajectories' initial points for some or all of the variables. For example, if the Sampler selected samples only 2 of the 5 optimization variables, the **<initial>** XML node is required only for the remaining 3 variables. The **<Sampler>** node recognizes the following parameters:
  - **class**: ***string, required***, RAVEN class for this entity (e.g. Samplers, Models, DataObjects)
  - **type**: ***string, required***, RAVEN type for this entity; a subtype of the class (e.g. MonteCarlo, Code, PointSet)
- **<Restart>**: ***string***, name of a DataObject. Used to leverage existing data when sampling a model. For example, if a Model has already been sampled, but some samples were not collected, the successful samples can be stored and used instead of rerunning the model for those specific samples. This RAVEN entity definition must be a DataObject with contents including the input and output spaces of the Model being sampled. The **<Restart>** node recognizes the following parameters:
  - **class**: ***string, optional***, The RAVEN class for this source. Options include **'DataObject'**.
  - **type**: ***string, optional***, The RAVEN type for this source. Options include any valid **<DataObject>** type, such as HistorySet or PointSet.

- **`<restartTolerance>`**: ***float***, specifies how strictly a matching point from a **`<Restart>`** DataObject must match the desired sample point in order to be used. If a potential restart point is within a relative Euclidean distance (as specified by the value in this node) of a desired sample point, the restart point will be used instead of sampling the Model.
  Default: 1e-15

- **`<variablesTransformation>`**: Allows transformation of variables via translation matrices. This defines two spaces, a "latent" transformed space sampled by RAVEN and a "manifest" original space understood by the Model. The **`<variablesTransformation>`** node recognizes the following parameters:

  - **`distribution`**: ***string, optional***, the name for the distribution defined in the XML node **`<Distributions>`**. This attribute indicates the values of **`<manifestVariables>`** are drawn from **`distribution`**.

  The **`<variablesTransformation>`** node recognizes the following subnodes:

  - **`<latentVariables>`**: ***comma-separated strings***, user-defined latent variables that are used for the variables transformation. All the variables listed under this node should be also mentioned in **`<variable>`**.
  - **`<manifestVariables>`**: ***comma-separated strings***, user-defined manifest variables that can be used by the **`<Model>`**.
  - **`<manifestVariablesIndex>`**: ***comma-separated strings***, user-defined manifest variables indices paired with **`<manifestVariables>`**. These indices indicate the position of manifest variables associated with multivariate normal distribution defined in the XML node **`<Distributions>`**. The indices should be postive integer. If not provided, the code will use the positions of manifest variables listed in **`<manifestVariables>`** as the indices.
  - **`<method>`**: ***string***, the method that is used for the variables transformation. The currently available method is **`'pca'`**.

Simulated Annealing Example:

```
<Optimizers>
  ...
  <SimulatedAnnealing name="simOpt">
    <samplerInit>
      <limit>2000</limit>
      <initialSeed>42</initialSeed>
      <writeSteps>every</writeSteps>
      <type>min</type>
    </samplerInit>
    <convergence>
```

```
        <objective>1e-6</objective>
        <temperature>1e-20</temperature>
        <persistence>1</persistence>
      </convergence>
      <coolingSchedule>
        <exponential>
          <alpha>0.94</alpha>
        </exponential>
      </coolingSchedule>
      <variable name="x">
        <distribution>beale_dist</distribution>
        <initial>-2.5</initial>
      </variable>
      <variable name="y">
        <distribution>beale_dist</distribution>
        <initial>3.5</initial>
      </variable>
      <objective>ans</objective>
      <TargetEvaluation class="DataObjects"
         type="PointSet">optOut</TargetEvaluation>
    </SimulatedAnnealing>
    ...
  </Optimizers>
```

11.3 GeneticAlgorithm

The **<GeneticAlgorithm>** optimizer is a metaheuristic approach to perform a global search in large design spaces. The methodology rose from the process of natural selection, and like others in the large class of the evolutionary algorithms, it utilizes genetic operations such as selection, crossover, and mutations to avoid being stuck in local minima and hence facilitates finding the global minima. More information can be found in: Holland, John H. ”Genetic algorithms.” Scientific american 267.1 (1992): 66-73.

When used as part of a **<MultiRun>** step, this entity provides additional information through the **<SolutionExport>** DataObject. The following variables can be requested within the **<SolutionExport>**:

- `trajID`: integer identifier for different optimization starting locations and paths
- `iteration`: integer identifying which iteration (or step, or generation) a trajectory is on
- `accepted`: string acceptance status of the potential optimal point (algorithm dependent)

- `rejectReason`: discription of reject reason, 'noImprovement' means rejected the new optimization point for no improvement from last point, 'implicitConstraintsViolation' means rejected by implicit constraints violation, return None if the point is accepted
- {`VAR`}: any variable from the **`<TargetEvaluation>`** input or output; gives the value of that variable at the optimal candidate for this iteration.
- `conv_`{`CONV`}: status of each given convergence criteria
- `fitness`: fitness of the current chromosome
- `age`: age of current chromosome
- `batchId`: Id of the batch to whom the chromosome belongs

The **`<GeneticAlgorithm>`** node recognizes the following parameters:

- **`verbosity`**: ***[silent, quiet, all, debug], optional***, Desired verbosity of messages coming from this entity
- **`name`**: ***string, required***, User-defined name to designate this entity in the RAVEN input file.

The **`<GeneticAlgorithm>`** node recognizes the following subnodes:

- **`<objective>`**: ***string***, Name of the response variable (or "objective function") that should be optimized (minimized or maximized).
- **`<variable>`**: defines the input space variables to be sampled through various means. The **`<variable>`** node recognizes the following parameters:
  - **`name`**: ***string, optional***, user-defined name of this Sampler. **Note:** As for the other objects, this is the name that can be used to refer to this specific entity from other input blocks
  - **`shape`**: ***comma-separated integers, optional***, determines the number of samples and shape of samples to be taken. For example, **`shape`**="2,3" will provide a 2 by 3 matrix of values, while **`shape`**="10" will produce a vector of 10 values. Omitting this optional attribute will result in a single scalar value instead. Each of the values in the matrix or vector will be the same as the single sampled value. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

  The **`<variable>`** node recognizes the following subnodes:

- **<distribution>**: ***string***, name of the distribution that is associated to this variable. Its name needs to be contained in the **<Distributions>** block explained in Section 9. In addition, if NDDistribution is used, the attribute **dim** is required. **Note:** Alternatively, this node must be omitted if the **<function>** node is supplied. The **<distribution>** node recognizes the following parameters:
  - **dim**: ***integer, optional***, for an NDDistribution, indicates the dimension within the NDDistribution that corresponds to this variable.
- **<grid>**: ***string***, – no description yet – The **<grid>** node recognizes the following parameters:
  - **type**: ***string, optional***, – no description yet –
  - **construction**: ***string, optional***, – no description yet –
  - **steps**: ***integer, optional***, – no description yet –
- **<function>**: ***string***, name of the function that defines the calculation of this variable from other distributed variables. Its name needs to be contained in the **<Functions>** block explained in Section 16. This function must implement a method named "evaluate". **Note:** Each **<variable>** must contain only one **<Function>** or **<Distribution>**, but not both.
- **<initial>**: ***comma-separated floats***, indicates the initial values where independent trajectories for this optimization effort should begin. The number of entries should be the same for all variables, unless a variable is initialized with a sampler (see **<samplerInit>** below). Note these entries are ordered; that is, if the optimization variables are x and y, and the initial values for x are **'1, 2, 3, 4'** and initial values for y are **'5, 6, 7, 8'**, then there will be four starting trajectories beginning at the locations (1, 5), (2, 6), (3, 7), and (4, 8).

- **<TargetEvaluation>**: ***string***, name of the DataObject where the sampled outputs of the Model will be collected. This DataObject is the means by which the sampling entity obtains the results of requested samples, and so should require all the input and output variables needed for adaptive sampling. The **<TargetEvaluation>** node recognizes the following parameters:
  - **class**: ***string, required***, RAVEN class for this entity (e.g. Samplers, Models, DataObjects)
  - **type**: ***string, required***, RAVEN type for this entity; a subtype of the class (e.g. MonteCarlo, Code, PointSet)

- **<samplerInit>**: collection of nodes that describe the initialization of the optimization algorithm.

  The **<samplerInit>** node recognizes the following subnodes:
  - **<limit>**: ***integer***, limits the number of Model evaluations that may be performed as part of this optimization. For example, a limit of 100 means at most 100 total Model evaluations may be performed.

- **`<writeSteps>`**: ***[final, every]***, delineates when the **`<SolutionExport>`** DataObject should be written to. In case of **`'final'`**, only the final optimal solution for each trajectory will be written. In case of **`'every'`**, the **`<SolutionExport>`** will be updated with each iteration of the Optimizer.
- **`<initialSeed>`**: ***integer***, seed for random number generation. Note that by default RAVEN uses an internal seed, so this seed must be changed to observe changed behavior.
  Default: RAVEN-determined
- **`<type>`**: ***[min, max]***, the type of optimization to perform. **`'min'`** will search for the lowest **`<objective>`** value, while **`'max'`** will search for the highest value.

- **`<GAparams>`**: Genetic Algorithm Parameters:
  - populationSize.
  - parentSelectors:
    - rouletteWheel.
    - tournamentSelection.
    - rankSelection.
  - Reproduction:
    - crossover:
      - onePointCrossover.
      - twoPointsCrossover.
      - uniformCrossover
    - mutators:
      - swapMutator.
      - scrambleMutator.
      - inversionMutator.
      - bitFlipMutator.
  - survivorSelectors:
    - ageBased.
    - fitnessBased.

  The **`<GAparams>`** node recognizes the following subnodes:

  - **`<populationSize>`**: ***integer***, The number of chromosomes in each population.
  - **`<parentSelection>`**: ***string***, A node containing the criterion based on which the parents are selected. This can be a fitness proportionate selection such as: a. ***rouletteWheel***, b. ***stochasticUniversalSampling***, c. ***Tournament***, d. ***Rank***, or e. ***randomSelection***

- **<reproduction>**: a node containing the reproduction methods. This accepts subnodes that specifies the types of crossover and mutation. The **<reproduction>** node recognizes the following parameters:
  - **nParents**: ***integer, required***, – no description yet –

  The **<reproduction>** node recognizes the following subnodes:
  - **<crossover>**: ***string***, a subnode containing the implemented crossover mechanisms. This includes: a. One Point Crossover, b. MultiPoint Crossover, c. Uniform Crossover, d. Whole Arithmetic Recombination, or e. Davis' Order Crossover. The **<crossover>** node recognizes the following parameters:
    - **type**: ***string, required***, – no description yet –

    The **<crossover>** node recognizes the following subnodes:
    - **<points>**: ***comma-separated integers***, point/gene(s) at which crossover will occur.
    - **<crossoverProb>**: ***float***, The probability governing the crossover occurance, i.e., the probability that if exceeded crossover will ocur.
  - **<mutation>**: ***string***, a subnode containing the implemented mutation mechanisms. This includes: a. Bit Flip, b. Random Resetting, c. Swap, d. Scramble, or e. Inversion. The **<mutation>** node recognizes the following parameters:
    - **type**: ***string, required***, – no description yet –

    The **<mutation>** node recognizes the following subnodes:
    - **<locs>**: ***comma-separated integers***, locations at which mutation will occur.
    - **<mutationProb>**: ***float***, The probability governing the mutation occurance, i.e., the probability that if exceeded mutation will ocur.
- **<survivorSelection>**: ***string***, a subnode containing the implemented servivor selection mechanisms. This includes: a. ageBased, or b. fitnessBased.
- **<fitness>**: ***string***, a subnode containing the implemented fitness functions. This includes: a. invLinear: $fitness = \frac{1}{a \times obj + b \times penalty}$. b. logistic: $fitness = \frac{1}{1+e^{a \times (obj - b)}}$ The **<fitness>** node recognizes the following parameters:
  - **type**: ***string, required***, [invLin, logistic]

  The **<fitness>** node recognizes the following subnodes:
  - **<a>**: ***float***, a: coefficient of objective function.
  - **<b>**: ***float***, b: coefficient of constraint penalty.

- **<convergence>**: a node containing the desired convergence criteria for the optimization algorithm. Note that convergence is met when any one of the convergence criteria is met. If no convergence criteria are given, then the defaults are used.

  The **<convergence>** node recognizes the following subnodes:

- **<objective>**: ***float***, provides the desired value for the convergence criterion of the objective function (ϵ^{obj}). In essence this is solving the inverse problem of finding the design variable at a given objective value, i.e., convergence is reached when:

$$Objevtive = \epsilon^{obj}$$

.
Default: 1e-6, if no criteria specified

- **<persistence>**: ***integer***, provides the number of consecutive times convergence should be reached before a trajectory is considered fully converged. This helps in preventing early false convergence.

- **<constant>**: ***comma-separated strings, integers, and floats***, allows variables that do not change value to be part of the input space. The **<constant>** node recognizes the following parameters:

  - **name**: ***string, required***, variable name for this constant, which will be provided to the Model.
  - **shape**: ***comma-separated integers, optional***, determines the shape of samples of the constant value. For example, **shape**="2,3" will shape the values into a 2 by 3 matrix, while **shape**="10" will shape into a vector of 10 values. Unlike the **<variable>**, the constant requires each value be entered; the number of required values is equal to the product of the **shape** values, e.g. 6 entries for shape "2,3"). **Note:** A model interface must be prepared to handle non- scalar inputs to use this option.
  - **source**: ***string, optional***, the name of the DataObject containing the value to be used for this constant. Requires **<ConstantSource>** node with a **<DataObject>** identified for this Sampler/Optimizer.
  - **index**: ***integer, optional***, the index of the realization in the **<ConstantSource>** **<DataObject>** containing the value for this constant. Requires **<ConstantSource>** node with a **<DataObject>** identified for this Sampler/Optimizer.

- **<ConstantSource>**: ***string***, identifies a **<DataObject>** to provide **<constant>** values to the input space of this entity while sampling. As an alternative to providing predefined values for constants, the **<ConstantSource>** provides a dynamic means of always providing the same value for a constant. This is often used as part of a larger multi-workflow calculation. The **<ConstantSource>** node recognizes the following parameters:

  - **class**: ***string, optional***, The RAVEN class for this source. Options include **'DataObject'**.
  - **type**: ***string, optional***, The RAVEN type for this source. Options include any valid **<DataObject>** type, such as HistorySet or PointSet.

- **<Constraint>**: ***string***, name of **<Function>** which contains explicit constraints for the sampling of the input space of the Model. From a practical point of view, this XML node

must contain the name of a function defined in the **<Functions>** block (see Section 16). This external function must contain a method called "constrain", which returns True for inputs satisfying the explicit constraints and False otherwise. The **<Constraint>** node recognizes the following parameters:

  - **class**: ***string, required***, RAVEN class for this source. Options include **'Functions'**.
  - **type**: ***string, required***, RAVEN type for this source. Options include **<External>**.

- **<ImplicitConstraint>**: ***string***, name of **<Function>** which contains implicit constraints of the Model. From a practical point of view, this XML node must contain the name of a function defined in the **<Functions>** block (see Section 16). This external function must contain a method called "implicitConstrain", which returns True for outputs satisfying the implicit constraints and False otherwise. The **<ImplicitConstraint>** node recognizes the following parameters:

  - **class**: ***string, required***, RAVEN class for this source. Options include **'Functions'**.
  - **type**: ***string, required***, RAVEN type for this source. Options include **<External>**.

- **<Sampler>**: ***string***, name of a Sampler that can be used to initialize the starting points for the trajectories of some of the variables. From a practical point of view, this XML node must contain the name of a Sampler defined in the **<Samplers>** block (see Section 10.1). The Sampler will be used to initialize the trajectories' initial points for some or all of the variables. For example, if the Sampler selected samples only 2 of the 5 optimization variables, the **<initial>** XML node is required only for the remaining 3 variables. The **<Sampler>** node recognizes the following parameters:

  - **class**: ***string, required***, RAVEN class for this entity (e.g. Samplers, Models, DataObjects)
  - **type**: ***string, required***, RAVEN type for this entity; a subtype of the class (e.g. MonteCarlo, Code, PointSet)

- **<Restart>**: ***string***, name of a DataObject. Used to leverage existing data when sampling a model. For example, if a Model has already been sampled, but some samples were not collected, the successful samples can be stored and used instead of rerunning the model for those specific samples. This RAVEN entity definition must be a DataObject with contents including the input and output spaces of the Model being sampled. The **<Restart>** node recognizes the following parameters:

  - **class**: ***string, optional***, The RAVEN class for this source. Options include **'DataObject'**.
  - **type**: ***string, optional***, The RAVEN type for this source. Options include any valid **<DataObject>** type, such as HistorySet or PointSet.

- **<restartTolerance>**: ***float***, specifies how strictly a matching point from a **<Restart>** DataObject must match the desired sample point in order to be used. If a potential restart

point is within a relative Euclidean distance (as specified by the value in this node) of a desired sample point, the restart point will be used instead of sampling the Model.
Default: 1e-15

- **`<variablesTransformation>`**: Allows transformation of variables via translation matrices. This defines two spaces, a "latent" transformed space sampled by RAVEN and a "manifest" original space understood by the Model. The **`<variablesTransformation>`** node recognizes the following parameters:

  - **`distribution`**: ***string, optional***, the name for the distribution defined in the XML node **`<Distributions>`**. This attribute indicates the values of **`<manifestVariables>`** are drawn from **`distribution`**.

  The **`<variablesTransformation>`** node recognizes the following subnodes:

  - **`<latentVariables>`**: ***comma-separated strings***, user-defined latent variables that are used for the variables transformation. All the variables listed under this node should be also mentioned in **`<variable>`**.
  - **`<manifestVariables>`**: ***comma-separated strings***, user-defined manifest variables that can be used by the **`<Model>`**.
  - **`<manifestVariablesIndex>`**: ***comma-separated strings***, user-defined manifest variables indices paired with **`<manifestVariables>`**. These indices indicate the position of manifest variables associated with multivariate normal distribution defined in the XML node **`<Distributions>`**. The indices should be postive integer. If not provided, the code will use the positions of manifest variables listed in **`<manifestVariables>`** as the indices.
  - **`<method>`**: ***string***, the method that is used for the variables transformation. The currently available method is **`'pca'`**.

Genetic Algorithm Example:

```
<Optimizers>
  ...
  <GeneticAlgorithm name="GAopt">
    <samplerInit>
      <limit>50</limit>
      <initialSeed>42</initialSeed>
      <writeSteps>every</writeSteps>
    </samplerInit>

    <GAparams>
      <populationSize>20</populationSize>
      <parentSelection>rouletteWheel</parentSelection>
```

```
    <reproduction nParents="4">
      <crossover type="onePointCrossover">
        <points>3</points>
        <crossoverProb>0.8</crossoverProb>
      </crossover>
      <mutation type="swapMutator">
        <locs>2,5</locs>
        <mutationProb>0.9</mutationProb>
      </mutation>
    </reproduction>
    <fitness type="invLinear">
      <a>2.0</a>
      <b>1.0</b>
    </fitness>
    <survivorSelection>fitnessBased</survivorSelection>
  </GAparams>

  <convergence>
    <objective>56</objective>
  </convergence>

  <variable name="x1">
    <distribution>uniform_dist_woRepl_1</distribution>
    <initial>1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20</initia
  </variable>

  <variable name="x2">
    <distribution>uniform_dist_woRepl_1</distribution>
    <initial>2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,1</initia
  </variable>

  <variable name="x3">
    <distribution>uniform_dist_woRepl_1</distribution>
    <initial>3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,1,2</initia
  </variable>

  <variable name="x4">
    <distribution>uniform_dist_woRepl_1</distribution>
    <initial>4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,1,2,3</initia
  </variable>

  <variable name="x5">
```

```
        <distribution>uniform_dist_woRepl_1</distribution>
        <initial>5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,1,2,3,4</initial
      </variable>

      <variable name="x6">
        <distribution>uniform_dist_woRepl_1</distribution>
        <initial>6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,1,2,3,4,5</initial
      </variable>

      <objective>ans</objective>
      <TargetEvaluation class="DataObjects"
         type="PointSet">optOut</TargetEvaluation>
    </GeneticAlgorithm>
    ...
  </Optimizers>
```

12 DataObjects

As seen in the previous chapters, different entities in the RAVEN code interact with each other in order to create, ideally, an infinite number of different calculation flows. These interactions are made possible through a data handling system that each entity understands. This system is called the "DataObjects" framework.

The **<DataObjects>** tag is a container of data objects of various types that can be constructed during the execution of a particular calculation flow. These data objects can be used as input or output for a particular **Model** (see Roles' meaning in section 15), etc. Currently, RAVEN supports 3 different data types, each with a particular conceptual meaning. These data types are instantiated as sub-nodes in the **<DataObjects>** block of an input file:

- **<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 generalization of the previously described DataObject, 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 (both dependent and/or independent). **Note: The <DataSet> is currently usable in the <EnsembleModel> only (see 15.6)**

In summary, the DataObjects accept the following data in their input/output spaces:

Table 2: DataObjects' accepted data formats.

DataObject	Input Space	Output Space
PointSet	scalars	scalars
HistorySet	scalars	vectors
DataSet	any	any

As noted above, each data object represents a mapping between a set of parameters and the resulting outcomes. The data objects are defined within the main XML block called **<DataObjects>**:

```
<Simulation>
   ...
  <DataObjects>
    <PointSet name='***'>...</PointSet>
    <HistorySet name='***'>...</HistorySet>
    <DataSet name='***'>...</DataSet>
  </DataObjects>
   ...
</Simulation>
```

Independently on the type of data, the respective XML node has the following available attributes:

- `name`, ***required string attribute***, is a user-defined identifier for this data object. **Note:** As with other objects, this name can be used to refer to this specific entity from other input blocks in the XML.
- `hierarchical`, ***optional boolean attribute***, This flag is going to "control" the printing/-plotting of the DataObject in case a hierarchical structure is determined (e.g. data coming from Dynamic Event Tree-like approaches):
  - if **True** all the branches of the tree are going to be printed/plotted independently (i.e. all the branches are going to be exposed independently)
  - if **False** all the branches are going to be walked back and reconstructed in order to create independent histories

  Default: False

In each XML node (e.g. `<PointSet>`, `<HistorySet>` or `<DataSet>`), the user specifies the following sub-nodes:

- `<Input>`, ***comma separated string, required field***, lists the input parameters to which this data is connected.
- `<Output>`, ***comma separated string, required field***, lists the output parameters to which this data is connected.
- `<Index>`, ***comma separated string, required for `<DataSet>`***, lists the dependent variables that depend on this index (specified through the attribute `var`). This XML node requires the following attribute:
  - `var`, ***required string attribute***, the dimension name of this index (e.g. time)

- **`<options>`**, ***optional node***, contains additional option nodes for data objects. This node contains the following subnodes:
  - **`<pivotParameter>`**, ***optional, string***, specifies the *pivotParameter* for a **`<HistorySet>`**. The pivot parameter is the shared index of the output variables in the data object.
    Default: time
  - **`<inputRow>`**, ***integer, optional field***, used to specify the row (in a CSV file or HDF5 table) from which the input space needs to be retrieved (e.g. the time-step);
  - **`<outputRow>`**, ***integer, optional field***, used to specify the row (in the CSV file or HDF5 table) from which the output space needs to be retrieved (e.g. the time-step). If this node is inputted, the nodes **`<operator>`** and **`<outputPivotValue>`** can not be inputted (mutually exclusive).
    Note: This XML node is available for DataObjects of type **`<PointSet>`** only;
  - **`<operator>`**, ***string, optional field***, is aimed to perform simple operations on the data to be stored. The 3 options currently available are:
    * **`'max'`**
    * **`'min'`**
    * **`'average'`**

    If this node is inputted, the nodes **`<outputRow>`** and **`<outputPivotValue>`** can not be inputted (mutually exclusive).
    Note: This XML node is available for DataObjects of type **`<PointSet>`** only;

The **`<PointSet>`** and **`<HistorySet>`** objects are a specialization of the **`<DataSet>`**. In the **`<PointSet>`**, the input and output space are all exclusively scalar values. These values might be extracted from a vector of values for each entry using the **`<options>`** node, but the end result is a single scalar per input or output variable.

For the **`<HistorySet>`**, all inputs must be scalar, and all outputs must share an index (the *pivotParameter*. There cannot be scalars in any of the outputs. The pivotParameter can be changed through the corresponding node in the **`<options>`** node.

Note that if the optional nodes in the block **`<options>`** are not inputted, the following default are applied:

- the Input space (scalars) is retrieved from the first row in the CSVs files or HDF5 tables (if the parameters specified are not among the variables sampled by RAVEN); In case of the **`<DataSet>`**, if any of the input space variables depend on an **`<Index>`**, they are going to be linked to the **`<Index>`** variable
- the output space defaults are as follows:

- if **<PointSet>**, the output space is retrieved from the last row in the CSVs files or HDF5 tables;
- if **<HistorySet>**, the output space is represented by all the rows found in the CSVs or HDF5 tables.
- if **<DataSet>**, the output space of the variables that do not depends on any index is retrieved from the last row in the CSVs files or HDF5 tables; on the contrary, the output space of the variables that depends on indexes is represented by all the rows found in the CSVs or HDF5 tables (if they match with the indexes' dimension)

```
<DataObjects>
  <PointSet name='outTPS1'>
    <options>
     <inputRow>1</inputRow>
     <outputRow>-1</outputRow>
    </options>
    <Input>pipe_Area,pipe_Dh,Dummy1</Input>
    <Output>pipe_Hw,pipe_Tw,time</Output>
  </PointSet>
  <HistorySet name='stories1'>
      <options>
          <pivotParameter>TIME</pivotParameter>
          <inputRow>1</inputRow>
          <outputRow>-1</outputRow>
      </options>
    <Input>pipe_Area,pipe_Dh</Input>
    <Output>pipe_Hw,pipe_Tw,time</Output>
  </HistorySet>
  <DataSet name='aDataSet'>
    <Input>pipe_Area,pipe_Dh</Input>
    <Output>pipe_Hw,pipe_Tw</Output>
    <Index var="time">pipe_Hw,pipe_Tw</Index>
  </DataSet>
</DataObjects>
```

13 Databases

The RAVEN framework provides the capability to store and retrieve data to/from an external database. Currently RAVEN has support for **netCDF4** and **HDF5** formats. NetCDF shares native format with RAVEN's DataObjects, but HDF5 is also included for convenience. This database, depending on the data format it is receiving, will organize itself in a "parallel" or "hierarchical" fashion. The user can create as many database objects as needed. The Database objects are defined within the main XML block called **<Databases>**:

```
<Simulation>
  ...
  <Databases>
    ...
    <NetCDF name="aDatabaseName1" readMode="overwrite"/>
    <HDF5 name="aDatabaseName2" readMode="overwrite"/>
    ...
  </Databases>
  ...
</Simulation>
```

The specifications for these two formats are listed below.

13.1 NetCDF

The specifications of each Database of type NetCDF needs to be defined within the XML block **<NetCDF>**, that recognizes the following attributes:

- **name**, ***required string attribute***, a user-defined identifier of this object. **Note:** As with other objects, this is name can be used to reference this specific entity from other input blocks in the XML.
- **readMode**, ***required string attribute***, defines whether an existing database should be read when loaded (**'read'**) or overwritten (**'overwrite'**). **Note:** if in **'read'** mode and the database is not found, RAVEN will read in the data as empty and raise a warning, NOT an error.
- **directory**, ***optional string attribute***, this attribute can be used to specify a particular directory path where the database will be created or read from. If an absolute path is given, RAVEN will respect it; otherwise, the path will be assumed to be relative to the **<WorkingDir>** from the **<RunInfo>** block. RAVEN recognizes path expansion tools such as tildes (*user dir*), single dots (*current dir*), and double dots (*parent dir*).
  Default: workingDir/DatabaseStorage. The **<workingDir>** is the one defined within the **<RunInfo>** XML block (see Section 6).

- **filename**, ***optional string attribute***, specifies the filename of the database that will be created in the **directory**. **Note:** When this attribute is not specified, the newer database filename will be named `name.nc`, where *name* corresponds to the **name** attribute of this object.
  Default: None

Example:

```
<Databases>
  <NetCDF name="name1" directory=''path_to_a_dir''
     readMode='overwrite'/>
  <HDF5 name="name2" filename=''Name2.nc'' readMode='read'/>
</Databases>
```

13.2 HDF5

The specifications of each Database of type HDF5 needs to be defined within the XML block **<HDF5>**, that recognizes the following attributes:

- **name**, ***required string attribute***, a user-defined identifier of this object. **Note:** As with other objects, this is name can be used to reference this specific entity from other input blocks in the XML.
- **readMode**, ***required string attribute***, defines whether an existing database should be read when loaded (**'read'**) or overwritten (**'overwrite'**). **Note:** if in **'read'** mode and the database is not found, RAVEN will read in the data as empty and raise a warning, NOT an error.
- **directory**, ***optional string attribute***, this attribute can be used to specify a particular directory path where the database will be created or read from. If an absolute path is given, RAVEN will respect it; otherwise, the path will be assumed to be relative to the **<WorkingDir>** from the **<RunInfo>** block. RAVEN recognizes path expansion tools such as tildes (*user dir*), single dots (*current dir*), and double dots (*parent dir*).
  Default: workingDir/DatabaseStorage. The **<workingDir>** is the one defined within the **<RunInfo>** XML block (see Section 6).
- **filename**, ***optional string attribute***, specifies the filename of the HDF5 that will be created in the **directory**. **Note:** When this attribute is not specified, the newer database filename will be named `name`.h5, where *name* corresponds to the **name** attribute of this object.
  Default: None
- **compression**, ***optional string attribute***, compression algorithm to be used. Available are:
  - **'gzip'**, best where portability is required. Good compression, moderate speed.

- `'lzf'`, Low to moderate compression, very fast.

Default: None

In addition, the `<HDF5>` recognizes the following subnodes:

- `<variables>`, ***optional, comma-separated string***, allows only a pre-specified set of variables to be included in the HDF5 when it is written to. If this node is not included, by default the HDF5 will include ALL of the input/output variables as a result of the step it is part of. If included, only the comma-separated variable names will be included if found.

  Note: RAVEN will not error if one of the requested variables is not found; instead, it will silently pass. It is recommended that a small trial run is performed, loading the HDF5 back into a data object, to check that the correct variables are saved to the HDF5 before performing large-scale calculations.

Example:

```
<Databases>
  <HDF5 name="aDatabaseName1" directory=''path_to_a_dir''
     compression=''lzf'' readMode='overwrite'/>
  <HDF5 name="aDatabaseName2" filename=''aDatabaseName2.h5''
     readMode='read'/>
</Databases>
```

14 OutStream system

The RAVEN framework provides the capabilities to visualize and print out the data generated, imported, and post-processed during RAVEN workflows. These capabilities are contained in the "OutStream" system. OutStream capabilities can be broadly classified into specific categories:

- **`<Print>`**, which allows data in memory to be saved to disk;
- **`<Plot>`**, which allows plotting data according to a variety of strategies.

Default implementations exist for **`<Print>`** and **`<Plot>`**, described in Sections 14.2 and 14.3. Other plotting strategies are described in section "Specific Plots" [14.4] below.

14.1 Defaults

Actually, two different default OutStream types are available:

- **Print**, module that lets the user dump the data contained in the internal objects;
- **Plot**, module, based on MatPlotLib [3], aimed to provide advanced plotting capabilities.

Both the types listed above accept as "input" a *DataObjects* object type. This choice is due to the "*DataObjects*" system (see section 12) having the main advantage of ensuring a standardized approach for exchanging the data/meta-data among the different framework entities. Every module can project its outcomes into a *DataObjects* object. This provides the user with the capability to visualize/dump all the modules' results. Additionally, the **Print** system can accept a ROM and inquire some of its specialized methods. As already mentioned, the RAVEN framework input is based on the e**X**tensible **M**arkup **L**anguage (**XML**) format. Thus, in order to activate the "*OutStream*" system, the input needs to contain a block identified by the **`<OutStreams>`** tag (as shown below).

```
<OutStreams>
  <!-- "OutStream" objects that need to be created-->
</OutStreams>
```

In the "OutStreams" block an unlimited number of "Plot" and "Print" sub-blocks can be specified. The input specifications and the main capabilities for both types are reported in the following sections.

14.2 Default Printing system

The Printing system has been created in order to let the user dump the data, contained in the internal data objects (see Section 12), out at anytime during the calculation. Additionally, the user

can inquire special methods of a **ROM** after training it, through a printing step. Currently, the only available output is a **C**omma **S**eparated **V**alue (**CSV**) file for **DataObjects**, and **XML** for **ROM** objects. This will facilitate the exchanging of results and provide the possibility to dump the solution of an analysis and "restart" another one constructing a data object from scratch, as well as access advanced features of particular reduced order models.

14.2.1 DataObjects Printing

The XML code, that is reported below, shows different ways to request a *Print* OutStream for **DataObjects**. The user needs to provide a name for each sub-block (XML attribute). These names are then used in the *Step* blocks to activate the Printing keywords at any time. The XML node has the following available attributes:

- **`name`**, ***required string attribute***, is a user-defined identifier for this data object. **Note:** As with other objects, this name can be used to refer to this specific entity from other input blocks in the XML.
- **`dir`**, ***optional string attribute***, is a user-defined directory in which the data are going to be streamed (i.e. printed). The directory can be either inputted with an relative (with respect the **`<workingDir>`** specified in the **`<RunInfo>`** XML node) or absolute path
  Default: ***`<workingDir>`***

As shown in the examples below, every **`<Print>`** block must contain, at least, the two required tags:

- **`<type>`**, the output file type (csv or xml). **Note:** Only **csv** is currently available for **`<DataObjects>`**
- **`<source>`**, the *Data* name (one of the *Data* items defined in the **`<DataObjects>`** block.

An optional tag **`<filename>`** can be used to specify the filename for the output. If this is not defined, then the default name will be the **`name`** identifier of the tag.

If only these two tags are provided (as in the "first-example" below), the output file will be filled with the whole content of the "d-name" *Data* object.

```
<OutStreams>
  <Print name='first-example'>
    <type>csv</type>
    <source>d-name</source>
  </Print>
  <Print name='second-example'>
    <type>csv</type>
    <source>d-name</source>
```

```
    <what>Output</what>
  </Print>
  <Print name='third-example'>
    <type>csv</type>
    <source>d-name</source>
    <what>Input</what>
  </Print>
  <Print name='fourth-example'>
    <type>csv</type>
    <source>d-name</source>
    <what>Input|var-name-in,Output|var-name-out</what>
  </Print>
  <Print name='fifth-example'>
    <type>csv</type>
    <source>d-name</source>
    <filename>example5</filename>
  </Print>
</OutStreams>
```

If just part of the **<source>** is important for a particular analysis, the additional XML tag **<what>** can be provided. In this block, the variables that need to be dumped must be specified, in comma separated format. The available options, for the **<what>** sub-block, are listed below:

- **Output**, the output space will be dumped out (see "second-example")
- **Input**, the input space will be dumped out (see "third-example")
- **Input—var-name-in/Output—var-name-out**, only the particular variables "var-name-in" and "var-name-out" will be reported in the output file (see "fourth-example")

Note all of the XML tags are case-sensitive but not their content.

14.2.2 ROM Printing

While all **ROM**s in RAVEN are designed to be used as surrogate models, some **ROM**s additionally offer information about the original model that isn't accessible through another means. For instance, **HDMRRom** objects can calculate sensitivity coefficients for subsets of the input domain. The XML code shown below demonstrates the methods to request these features from a **ROM**. The user needs to provide a **<name>** for each sub-block (XML attribute). These names are then used in the *Step* blocks to activate the Printing keywords at any time. As shown in the examples below, every **<Print>** block for ROMs must contain, at least, the three required tags

- **<type>**, the output file type (csv or xml). **Note:** Only **xml** is currently available for ROMs
- **<source>**, the *ROM* name (one of the **<ROM>** items defined in the **<Models>** block.

- **<what>**, the comma-separated list of desired metrics. The list of metrics available in each ROM is listed under that ROM type in Section 15.3. Alternatively, the keyword **'all'** can be provided to request all available metrics, if any.

Addtionally, when printing ROMs one optional node is available,

- **<target>**, the ROM target for which to inquire data

If the ROM is time-dependent, the printed properties will be collected by time step. ROM printing uses the same naming conventions as DataObjects printing. Examples:

```
<OutStreams>
  <Print name='first-ROM-example'>
    <type>xml</type>
    <source>mySobolRom</source>
    <what>all</what>
  </Print>
  <Print name='second-ROM-example'>
    <type>xml</type>
    <source>myGaussPolyRom</source>
    <what>mean,variance</what>
  </Print>
</OutStreams>
```

14.3 Default Plotting system

The Plotting system provides all the capabilities to visualize the analysis outcomes, in real-time or as a post-processing stage. The system is based on the Python library MatPlotLib [3]. MatPlotLib is a 2D/3D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. This external tool has been wrapped in the RAVEN framework, and is available to the user. Since it was unfeasible to support, in the source code, all the interfaces for all the available plot types, the RAVEN Plotting system directly provide a formatted input structure for 11 different plot types (2D/3D). The user may request a plot not present among the supported ones, since the RAVEN Plotting system has the capability to construct on the fly the interface for a Plot, based on XML instructions.

14.3.1 Plot input structure

In order to create a plot, the user needs to add, within the **<OutStreams>** block, a **<Plot>** sub-block. Similar to the **<Print>** OutStream, the user needs to specify a **name** as an attribute of the plot. This name will then be used to request the plot in the **<Steps>** block. In addition, the Plot block accepts the following attributes:

- **interactive**, ***optional bool attribute***, specifies if the Plot needs to be interactively created (real-time screen visualization).
  Default: False
- **overwrite**, ***optional bool attribute***, used when the plot needs to be dumped into picture file/s. This attribute determines whether the code needs to overwrite the image files every time a new plot (with the same name) is requested.
  Default: False
- **dir**, ***optional string attribute***, is a user-defined directory in which the data are going to be streamed (i.e. printed). The directory can be either inputted with an relative (with respect the **<workingDir>** specified in the **<RunInfo>** XML node) or absolute path
  Default: ***<workingDir>***

An optional tag **<filename>** can be used to specify the filename for the output. If this is not defined, then the default base name will be the **name** identifier of the tag prepended and appended with extra information that identifies the plot further.

As shown, in the XML input example below, the body of the Plot XML input contains two main sub-nodes:

- **<actions>**, where general control options for the figure layout are defined (see Section 14.3.1.1).
- **<plotSettings>**, where the actual plot options are provided.

These two main sub-block are discussed in the following paragraphs.

14.3.1.1 "Actions" input block

The input in the **<actions>** sub-node is common to all the Plot types, since, in it, the user specifies all the controls that need to be applied to the figure style. This block must be unique in the definition of the **<Plot>** main block. In the following list, all the predefined "actions" are reported:

- **<how>**, comma separated list of output types:
  - `screen`, show the figure on the screen in interactive mode
  - `pdf`, save the figure as a Portable Document Format file (PDF). **Note:** The pdf format does not support multiple layers that lay on the same pixel. If the user gets an error about this, he/she should move to another format.
  - `png`, save the figure as a Portable Network Graphics file (PNG)
  - `eps`, save the figure as an Encapsulated Postscript file (EPS)
  - `pgf`, save the figure as a LaTeX PGF Figure file (PGF)
  - `ps`, save the figure as a Postscript file (PS)

- `gif`, save the figure as a Graphics Interchange Format (GIF)
- `svg`, save the figure as a Scalable Vector Graphics file (SVG)
- `jpeg`, save the figure as a jpeg file (JPEG)
- `raw`, save the figure as a Raw RGBA bitmap file (RAW)
- `bmp`, save the figure as a Windows bitmap file (BMP)
- `tiff`, save the figure as a Tagged Image Format file (TIFF)
- `svgz`, save the figure as a Scalable Vector Graphics file (SVGZ)

- **`<title>`**, as the name suggests, within this block the user can specify the title of the figure. In the body of this node, a few other tags are available:

  - **`<text>`**, ***string type***, title of the figure
  - **`<kwargs>`**, within this block the user can specify optional parameters with the following format:

    ```
    <kwargs>
     <param1>value1</param1>
     <param2>value2</param2>
    </kwargs>
    ```

    The kwargs block is able to convert whatever string into a python type (for example **`<param1>`**`{'1stKey':45}`**`</param1>`** will be converted into a dictionary, **`<param2>`**`[56,67]`**`</param2>`** into a list, etc.). For reference regarding the available kwargs, see "matplotlib.pyplot.title" method in [3].

- **`<labelFormat>`**, within this block the default scale formatting can be modified. In the body, a few other tags are available:

  - **`<axis>`**, ***string***, the axis where to apply the defined format, 'x,' 'y,' or 'both'.
    Default: 'both' **Note:** If this action will be used in a 3-D plot, the user can input 'z' as well and 'both' will apply this format to all three axis.
  - **`<style>`**, ***string***, the style of the number notation, 'sci' or 'scientific' for scientific, 'plain' for plain notation.
    Default: scientific
  - **`<scilimits>`**, ***tuple, (m, n), pair of integers***, if style is 'sci', scientific notation will be used for numbers outside the range 10^m to 10^n. Use (0,0) to include all numbers. **Note:** The value for this keyword, needs to be specified between brackets [for example, (5,6)].
    Default: (0,0)
  - **`<useOffset>`**, ***bool or double***, if True, the offset will be calculated as needed; if False, no offset will be used; if a numeric offset is specified, it will be used.
    Default: False

- **<figureProperties>**, within this block the user specifies how to customize the figure style/quality. Thus, through this "action" the user has got full control on the quality of the figure, its dimensions, etc. This control is performed by the following keywords:
  - **<figsize>**, ***tuple (optional)***, (width, hight), in inches.
  - **<dpi>**, ***integer***, dots per inch.
  - **<facecolor>**, ***string***, set the figure background color (please refer to "matplotlib.figure.Figure" in [3] for a list of all the colors available).
  - **<edgecolor>**, ***string***, the figure edge background color (please refer to "matplotlib.figure.Figure" in [3] for a list of all the colors available).
  - **<linewidth>**, ***float***, the width of lines drawn on the plot.
  - **<frameon>**, ***bool***, if False, suppress drawing the figure frame.
- **<range>**, the range "action" specifies the ranges of all the axis. All the keywords in the body of this block are optional:
  - **<ymin>**, ***double (optional)***, lower boundary for the y axis.
  - **<ymax>**, ***double (optional)***, upper boundary for the y axis.
  - **<xmin>**, ***double (optional)***, lower boundary for the x axis.
  - **<xmax>**, ***double (optional)***, upper boundary for the x axis.
  - **<zmin>**, ***double (optional)***, lower boundary for the z axis. **Note:** This keyword is effective in 3-D plots only.
  - **<zmax>**, ***double (optional)***, upper boundary for the z axis. **Note:** This keyword is effective in 3-D plots only.
- **<camera>**, the camera item is available in 3-D plots only. Through this "action," it is possible to orientate the plot as one wishes. The controls are:
  - **<elevation>**, ***double (optional)***, stores the elevation angle in the z plane.
  - **<azimuth>**, ***double (optional)***, stores the azimuth angle in the x,y plane.
- **<scale>**, the scale block allows the specification of the axis scales:
  - **<xscale>**, ***string (optional)***, scale of the x axis. Three options are available: "linear,""log," or "symlog."
    Default: linear
  - **<yscale>**, ***string (optional)***, scale of the y axis. Three options are available: "linear," "log," or "symlog."
    Default: linear
  - **<zscale>**, ***string (optional)***, scale of the z axis. Three options are available: "linear," "log," or "symlog."
    Default: linear **Note:** This keyword is effective in 3-D plots only.

- **<addText>**, same as title.
- **<autoscale>**, is a convenience method for simple axis view autoscaling. It turns autoscaling on or off, and then, if autoscaling for either axis is on, it performs the autoscaling on the specified axis or axes. The following sub-nodes may be specified:
  - **<enable>**, ***bool (optional)***, True turns autoscaling on, False turns it off. None leaves the autoscaling state unchanged.
    Default: True
  - **<axis>**, ***string (optional)***, determines which axis to apply the defined format, 'x,' 'y,' or 'both.'
    Default: 'both' **Note:** If this action is used in a 3-D plot, the user can input 'z' as well and 'both' will apply this format to all three axis.
  - **<tight>**, ***bool (optional)***, if True, sets the view limits to the data limits; if False, let the locator and margins expand the view limits; if None, use tight scaling if the only output is an image file, otherwise treat tight as False.
- **<horizontalLine>**, this "action" provides the ability to draw a horizontal line in the current figure. This capability might be useful, for example, if the user wants to highlight a trigger function of a variable. The following sub-nodes may be specified:
  - **<y>**, ***double (optional)***, sets the y-value for the line.
    Default: 0
  - **<xmin>**, ***double (optional)***, is the starting coordinate on the x axis.
    Default: 0
  - **<xmax>**, ***double (optional)***, is the ending coordinate on the x axis.
    Default: 1
  - **<kwargs>**, within this block the user can specify optional parameters with the following format:

    ```
    <kwargs>
     <param1>value1</param1>
     <param2>value2</param2>
    </kwargs>
    ```

    The kwargs block is able to convert whatever string into a python type (for example **<param1>**`{'1stKey':45}`**</param1>** will be converted into a dictionary, **<param2>**`[56,67]`**</param2>** into a list, etc.). For reference regarding the available kwargs, see "matplotlib.pyplot.axhline" method in [3].

  Note: This capability is not available for 3-D plots.
- **<verticalLine>**, similar to the "horizontalLine" action, this block provides the ability to draw a vertical line in the current figure. This capability might be useful, for example, if

the user wants to highlight a trigger function of a variable. The following sub-nodes may be specified:

- **<x>**, ***double (optional)***, sets the x coordinate of the line.
  Default: 0
- **<ymin>**, ***double (optional)***, starting coordinate on the y axis.
  Default: 0
- **<ymax>**, ***double (optional)***, ending coordinate on the y axis.
  Default: 1
- **<kwargs>**, within this block the user can specify optional parameters with the following format:

```
<kwargs>
 <param1>value1</param1>
 <param2>value2</param2>
</kwargs>
```

  The kwargs block is able to convert whatever string into a python type (for example **<param1>**`{'1stKey':45}`**</param1>** will be converted into a dictionary, **<param2>**`[56,67]`**</param2>** into a list, etc.). For reference regarding the available kwargs, see "matplotlib.pyplot.axvline" method in [3].

Note: This capability is not available for 3-D plots.

- **<horizontalRectangle>**, this "action" provides the ability to draw, in the current figure, a horizontally orientated rectangle. This capability might be useful, for example, if the user wants to highlight a zone in the plot. The following sub-nodes may be specified:

  - **<ymin>**, ***double (required)***, starting coordinate on the y axis.
  - **<ymax>**, ***double (required)***, ending coordinate on the y axis.
  - **<xmin>**, ***double (optional)***, starting coordinate on the x axis.
    Default: 0
  - **<xmax>**, ***double (optional)***, ending coordinate on the x axis. Default = 1
  - **<kwargs>**, within this block the user can specify optional parameters with the following format:

```
<kwargs>
 <param1>value1</param1>
 <param2>value2</param2>
</kwargs>
```

  The kwargs block is able to convert whatever string into a python type (for example **<param1>**`{'1stKey':45}`**</param1>** will be converted into a dictionary, **<param2>**`[56,67]`**</param2>** into a list, etc.). For reference regarding the available kwargs, see "matplotlib.pyplot.axhspan" method in [3].

Note: This capability is not available for 3D plots.

- **<verticalRectangle>**, this "action" provides the possibility to draw, in the current figure, a vertically orientated rectangle. This capability might be useful, for example, if the user wants to highlight a zone in the plot. The following sub-nodes may be specified:
  - **<xmin>**, ***double (required)***, starting coordinate on the x axis.
  - **<xmax>**, ***double (required)***, ending coordinate on the x axis.
  - **<ymin>**, ***double (optional)***, starting coordinate on the y axis.
    Default: 0
  - **<ymax>**, ***double (optional)***, ending coordinate on the y axis.
    Default: 1
  - **<kwargs>**, within this block the user can specify optional parameters with the following format:

```
<kwargs>
 <param1>value1</param1>
 <param2>value2</param2>
</kwargs>
```

    The kwargs block is able to convert whatever string into a python type (for example **<param1>**`{'1stKey':45}`**</param1>** will be converted into a dictionary, **<param2>**`[56,67]`**</param2>** into a list, etc.). For reference regarding the available kwargs, see "matplotlib.pyplot.axvspan" method in [3].

Note: This capability is not available for 3D plots.

- **<axesBox>**, this keyword controls the axes' box. Its value can be 'on' or 'off'.
- **<axisProperties>**, this block is used to set axis properties. There are no fixed keywords. If only a single property needs to be set, it can be specified as the body of this block, otherwise a dictionary-like string needs to be provided. For reference regarding the available keys, refer to "matplotlib.pyplot.axis" method in [3].
- **<grid>**, this block is used to define a grid that needs to be added in the plot. The following keywords can be inputted:
  - **<b>**, ***boolean (required)***, toggles the grid lines on or off.
  - **<which>**, ***double (required)***, ending coordinate on the x axis.
  - **<axis>**, ***double (optional)***, starting coordinate on the y axis.
    Default: 0
  - **<kwargs>**, within this block the user can specify optional parameters with the following format:

```
<kwargs>
 <param1>value1</param1>
 <param2>value2</param2>
</kwargs>
```

The kwargs block is able to convert whatever string into a python type (for example **<param1>**{'1stKey':45}**</param1>** will be converted into a dictionary, **<param2>**[56,67]**</param2>** into a list, etc.).

14.3.1.2 "plotSettings" input block

The sub-block identified by the keyword **<plotSettings>** is used to define the plot characteristics. Within this sub-section at least a **<plot>** block must be present. the **<plot>** sub-section may not be unique within the **<plotSettings>** definition; the number of **<plot>** sub-blocks is equal to the number of plots that need to be placed in the same figure.

If sub-plots are to be defined then **<gridSpace>** needs to be present. **<gridSpace>** specifies the geometry of the grid that a subplot will be placed. The number of rows and number of columns of the grid need to be set.

For example, in the following XML cut, a "line" and a "scatter" type are combined in the same figure.

```
<OutStreams>
  <Plot name='example2PlotsCombined'>
    <actions>
      <!-- Actions -->
    </actions>
    <plotSettings>
      <gridSpace>2 2</gridSpace>
      <plot>
        <type>line</type>
        <x>d-type|Output|x1</x>
        <y>d-type|Output|y1</y>
        <xlabel>label X</xlabel>
        <ylabel>label Y</ylabel>
        <gridLocation>
          <x>0 2</x>
          <y>0</y>
        </gridLocation>
      </plot>
      <plot>
        <type>scatter</type>
```

```
        <x>d-type|Output|x2</x>
        <y>d-type|Output|y2</y>
        <xlabel>label X</xlabel>
        <ylabel>label Y</ylabel>
        <gridLocation>
          <x>0 2</x>
          <y>1</y>
        </gridLocation>
      </plot>
    </plotSettings>
  </Plot>
</OutStreams>
```

The axis labels are conditionally optional nodes that can be defined under the **<plotSetting>**. If the plot does not contain any sub-plots, i.e. **<gridSpace>** is not defined then the axis labels are global parameters for the figure which are defined under **<plotSettings>**, otherwise the axis labels can be defined under **<plot>** for each sub-plot seperately.

- **<xlabel>**, ***string, optional parameter***, the x axis label.
- **<ylabel>**, ***string, optional parameter***, the y axis label.
- **<zlabel>**, ***string, optional parameter (3D plots only)***, the z axis label.

One may also specify a **<legend>** tag that will place a legend on the plot. The legend accepts the following sub-nodes:

- **<loc>**, ***string, optional parameter***, the location where the legend will be placed on the plot. Valid values are:
  - **'best'**
  - **'upper right'**
  - **'upper left'**
  - **'lower left'**
  - **'lower right'**
  - **'right'**
  - **'center left'**
  - **'center right'**
  - **'lower center'**

- **'upper center'**
- **'center'**

Default: ***'best'***

- **<ncol>**, ***integer, optional parameter***, the number of columsn to include in the legend. *Default: 1*
- **<fontsize>**, ***string, optional parameter***, the font size of the legend. Valid values are:
  - **'xx-small'**
  - x-small
  - **'small'**
  - **'medium'**
  - **'large'**
  - **'x-large'**
  - **'xx-large'**
- **<title>**, ***string, optional parameter***, the title of the legend.

Note: The text associated to each **<plot>** tag in the legend is defined in the **<kwargs>** of that plot by specifying a **<label>** within the kwargs. An example usage is given below:

```
<Plot ...>
  ...
  <plotSettings>
    <plot>
      <type>scatter</type>
        <x>...</x>
        <y>...</y>
        <kwargs>
          <label>dots</label>
        </kwargs>
     </plot>
    <plot>
      <type>line</type>
        <x>...</x>
        <y>...</y>
        <kwargs>
          <label>line</label>
        </kwargs>
```

```
      </plot>
     <legend>
       <loc>best</loc>
       <ncol>2</ncol>
     </legend>
   </plotSettings>
</Plot>
```

This will create a plot with both scattered points and a line. The plot will also have a legend specifying the labels "dots" and "line" in two columns with the best location selected by matplotlib.

As already mentioned, within the **<plotSettings>** block, at least a **<plot>** sub-block needs to be specified. Independent of the plot type, some keywords are mandatory:

- **<type>**, ***string, required parameter***, the plot type (for example, line, scatter, wireframe, etc.).
- **<x>**, ***string, required parameter***, specifies the DataObject parameter to be plotted as the x coordinate. This parameter must be described in a specific manner, see Section 14.3.1.2.1 below for details.
- **<y>**, ***string, required parameter***, specifies the DataObject parameter to be plotted as the y coordinate. This parameter must be described in a specific manner, see Section 14.3.1.2.1 below for details.
- **<z>**, ***string, required parameter for plots with three dimensions***, specifies the DataObject parameter to be plotted as the z coordinate. This parameter must be described in a specific manner, see Section 14.3.1.2.1 below for details.

In addition, other plot-dependent keywords, reported in Section 14.3.1.3, can be provided.

Under the **<plot>** sub-block other optional keywords can be specified, such as:

- **<xlabel>**, ***string, optional parameter***, the x axis label.
- **<ylabel>**, ***string, optional parameter***, the y axis label.
- **<zlabel>**, ***string, optional parameter (3D plots only)***, the z axis label.
- **<gridLocation>**, ***xmlNode, optional xmlNode (depending on the grid geometry)***
  - **<x>**, ***integer, required parameter***, the position of the subPlot in the grid Space. if this node has a single value then the subplot occupies a single node at the specified location, otherwise the second integer represents the number of nodes that this subplot occupies,

i.e. in the example above the first subplot occupies 2 nodes starting from the zero node in x direction.

  - **<y>**, ***integer, required parameter***, the position of the subPlot in the grid Space. if this node has a single value then the subplot occupies a single node at the specified location, otherwise the second integer represents the number of nodes that this subplot occupies, i.e. in the example above the first subplot occupies a single node at the zero node in y direction.

- **<colorMap>**, ***string, optional parameter***, specifies a DataObject parameter whose value will be used to vary the color of plotted points. This parameter must be described in a specific manner, see Section 14.3.1.2.1 below for details.

14.3.1.2.1 Specifying What Values to Plot As already mentioned, the Plot system accepts as input for the visual parameters (i.e., x, y, z, colorMap), data only from a **DataObjects** object. Considering the structure of "DataObjects", the parameters are specified as three values separated by the vertical bar character ('|') as follows:

```
DataObject Name|Parameter Type|Parameter Name
```

Where:

Value	Description
`DataObject Name`	Name of the DataObject that contains the parameter
`Parameter Type`	Either `Input` or `Output` depending on whether the parameter is defined in the **<Input>** or **<Output>** part of the DataObject
`Parameter Name`	The name of the parameter in the DataObject to plot

Note: If the `Parameter Name` part of the variable specification itself contains the vertical bar character ('|') used to separate the three values, it must be enclosed in parenthesis to be interpreted properly. For example:

```
DataObject Name|Parameter Type|(parameter|name)
```

14.3.1.3 Predefined Plotting System: 2D/3D

As already mentioned above, the Plotting system provides a specialized input structure for several different kind of plots specified in the **<type>** node:

- *2 Dimensional plots*:

- `scatter` creates a scatter plot of x vs y, where x and y are sequences of numbers of the same length.
- `line` creates a line plot of x vs y, where x and y are sequences of numbers of the same length.
- `histogram` computes and draws the histogram of x. **Note:** This plot accepts only the XML node **`<x>`** even if it is considered as a 2D plot type.
- `stem` plots vertical lines at each x location from the baseline to y, and places a marker there.
- `step` creates a 2 dimensional step plot.
- `pseudocolor` creates a pseudocolor plot of a two dimensional array. The two dimensional array is built creating a mesh from **`<x>`** and **`<y>`** data, in conjunction with the data specified in the **`<colorMap>`** node.
- `contour` builds a contour plot creating a plot from **`<x>`** and **`<y>`** data, in conjunction with the data specified in the **`<colorMap>`** node.
- `filledContour` creates a filled contour plot from **`<x>`** and **`<y>`** data, in conjunction with the data specified in the **`<colorMap>`** node.

• *3 Dimensional plots*:

- `scatter` creates a scatter plot of (x,y) vs z, where x, y, z are sequences of numbers of the same length.
- `line` creates a line plot of (x,y) vs z, where x, y, z are sequences of numbers of the same length.
- `stem` creates a 3 Dimensional stem plot of (x,y) vs z.
- `surface` creates a surface plot of (x,y) vs z. By default it will be colored in shades of a solid color, but it also supports color mapping.
- `wireframe` creates a 3D wire-frame plot. No color mapping is supported.
- `tri-surface` creates a 3D tri-surface plot. It is a surface plot with automatic triangulation.
- `contour3D` builds a 3D contour plot creating the plot from **`<x>`**, **`<y>`** and **`<z>`** data, in conjunction with the data specified in **`<colorMap>`**.
- `filledContour3D` builds a filled 3D contour plot creating the plot from **`<x>`**, **`<y>`** and **`<z>`** data, in conjunction with the data specified in **`<colorMap>`**.
- `histogram` computes and draws the histogram of x and y. **Note:** This plot accepts only the XML nodes **`<x>`** and **`<y>`** even if it is considered as 3D plot type since the frequency is mapped to the third dimension.

As already mentioned, the settings for each plot type are specified within the XML block called **<plot>**. The sub-nodes that are available depends on the plot type as each plot type has its own set of parameters that can be specified.

In the following sub-sections all the options for the plot types listed above are reported.

14.3.2 2D & 3D Scatter plot

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

- **<s>**, ***integer, optional field***, represents the size in points$\hat{2}$. The "points" have the same meaning of the font size (e.g. Times New Roman, pts 10). In here the user specifies the area of the marker size.
  Default: 20

- **<c>**, ***string, optional field***, specifies the color or sequence of color to use. **<c>** can be a single color format string, a sequence of color specifications of length N, or a sequence of N numbers to be mapped to colors using the cmap and norm specified via **<kwargs>**. **Note: <c>** should not be a single numeric RGB or RGBA sequence because that is indistinguishable from an array of values to be colormapped. **<c>** can be a 2D array in which the rows are RGB or RGBA. **Note: <colorMap>** will overwrite **<c>**. If **<colorMap>** is defined then the color set used can be defined by **<cmap>**. If no **<cmap>** is given then the default color set of "matplotlib.pyplot.scatter" method in [3] is used. If **<colorMap>** is not defined then the plot is in solid color (default *blue*) as defined with **<color>** in **<kwargs>**.

- **<marker>**, ***string, optional field***, specifies the type of marker to use.
  Default: o

- **<alpha>**, ***string, optional field***, sets the alpha blending value, between 0 (transparent) and 1 (opaque).
  Default: None

- **<linewidths>**, ***string, optional field***, widths of lines used in the plot. Note that this is a tuple, and if you set the linewidths argument you must set it as a sequence of floats.
  Default: None;

- **<kwargs>**, within this block the user can specify optional parameters with the following format:

```
<kwargs>
 <param1>value1</param1>
 <param2>value2</param2>
```

```
</kwargs>
```

The kwargs block is able to convert whatever string into a python type (for example **<param1>** `{'1stKey':45}` **</param1>** will be converted into a dictionary, **<param2>** `[56,67]` **</param2>** into a list, etc.). For reference regarding the available kwargs, see "matplotlib.pyplot.scatter" method in [3].

14.3.3 2D & 3D Line plot

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

- **<interpolationType>**, ***string, optional field***, is the type of interpolation algorithm to use for the data. Available options are "nearest," "linear," "cubic," "multiquadric," "inverse," "gaussian," "Rbflinear," "Rbfcubic," "quintic," and "thin_plate."
  Default: linear
- **<interpPointsX>**, ***integer, optional field***, sets the number of points need to be used for interpolation of the x axis.
- **<interpPointsY>**, ***integer, optional field***, sets the number of points need to be used for interpolation of the y axis. (only 3D line plot). **Note:** If **<colorMap>** is used then a **scatter plot** will be plotted.

14.3.4 2D & 3D Histogram plot

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

- **<bins>**, ***integer or array_like, optional field***, sets the number of bins if an integer is used or a sequence of edges if a python list is used.
  Default: 10
- **<normed>**, ***boolean, optional field***, if True then the the histogram will be normalized to 1.
  Default: False
- **<weights>**, ***sequence, optional field***, represents an array of weights, of the same shape as x. Each value in x only contributes its associated weight towards the bin count (instead of 1). If normed is True, the weights are normalized, so that the integral of the density over the range remains 1.
  Default: None

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

- **<histtype>**, ***string, optional field***, The type of histogram to draw:
  - **bar** is a traditional bar-type histogram. If multiple data sets are given the bars are arranged side by side.
  - **barstacked** is a bar-type histogram where multiple data sets are stacked on top of each other.
  - **step** generates a line plot that is by default unfilled.
  - **stepfilled** generates a line plot that is by default filled.

  Default: bar

- **<align>**, ***string, optional field***, controls how the histogram is plotted.
  - **left** bars are centered on the left bin edge.
  - **mid** bars are centered between the bin edges.
  - **right** bars are centered on the right bin edges.

  Default: mid

- **<orientation>**, ***string, optional field***, specifies the orientation of the histogram:
  - **horizontal**
  - **vertical**

  Default: vertical

- **<rwidth>**, ***float, optional field***, sets the relative width of the bars as a fraction of the bin width.
  Default: None

- **<log>**, ***boolean, optional field***, sets a log scale.
  Default: False

- **<color>**, ***string, optional field***, specifies the color of the histogram.
  Default: blue;

- **<stacked>**, ***boolean, optional field***, if True, multiple data elements are stacked on top of each other. If False, multiple data sets are aranged side by side if histtype is 'bar' or on top of each other if histtype is 'step.'
  Default: False

- **<kwargs>**, within this block the user can specify optional parameters with the following format:

```
<kwargs>
 <param1>value1</param1>
 <param2>value2</param2>
</kwargs>
```

  The kwargs block is able to convert whatever string into a python type (for example **<param1>** `{'1stKey':45}` **</param1>** will be converted into a dictionary, **<param2>** `[56,67]` **</param2>** into a list, etc.). For reference regarding the available kwargs, see "matplotlib.pyplot.hist" method in [3].

14.3.5 2D & 3D Stem plot

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

- **<linefmt>**, ***string, optional field***, sets the line style used in the plot.
  Default: b-

- **<markerfmt>**, ***string, optional field***, sets the type of marker format to use in the plot.
  Default: bo

- **<basefmt>**, ***string, optional field***, sets the base format.
  Default: r-

- **<kwargs>**, within this block the user can specify optional parameters with the following format:

```
<kwargs>
 <param1>value1</param1>
 <param2>value2</param2>
</kwargs>
```

The kwargs block is able to convert whatever string into a python type (for example **<param1>** {'1stKey':45}**</param1>** will be converted into a dictionary, **<param2>** [56,67] **</param2>** into a list, etc.).

For reference regarding the available kwargs, see "matplotlib.pyplot.stem" method in [3].

14.3.6 2D Step plot

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

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

  Default: mid

- **<kwargs>**, within this block the user can specify optional parameters with the following format:

```
<kwargs>
 <param1>value1</param1>
 <param2>value2</param2>
</kwargs>
```

  The kwargs block is able to convert whatever string into a python type (for example **<param1>** {'1stKey':45}**</param1>** will be converted into a dictionary, **<param2>** [56,67] **</param2>** into a list, etc.). For reference regarding the available kwargs, see "matplotlib.pyplot.step" method in [3].

14.3.7 2D Pseudocolor plot

In order to create a 2D "pseudocolor" plot, the user needs to write in the **<type>** body the keyword "pseudocolor." In order to customize the plot, the user can define the following XML sub nodes:

- **<interpolationType>**, ***string, optional field***, is the type of interpolation algorithm to use for the data. Available options are "nearest," "linear," "cubic," "multiquadric," "inverse," "gaussian," "Rbflinear," "Rbfcubic," "quintic," and "thin_plate."
  Default: [linear]

- **<interpPointsX>**, ***integer, optional field***, sets the number of points need to be used for interpolation of the x axis.

- **<kwargs>**, within this block the user can specify optional parameters with the following format:

```
<kwargs>
 <param1>value1</param1>
 <param2>value2</param2>
</kwargs>
```

  The kwargs block is able to convert whatever string into a python type (for example **<param1>** `{'1stKey':45}` **</param1>** will be converted into a dictionary, **<param2>** `[56,67]` **</param2>** into a list, etc.). For reference regarding the available kwargs, see "matplotlib.pyplot.pcolor" method in [3].

14.3.8 2D Contour or filledContour plots

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

- **<numberBins>**, ***integer, optional field***, sets the number of bins.
  Default: 5

- **<interpolationType>**, ***string, optional field***, is the type of interpolation algorithm to use for the data. Available options are "nearest," "linear," "cubic," "multiquadric," "inverse," "gaussian," "Rbflinear," "Rbfcubic," "quintic," and "thin_plate."
  Default: linear

- **<interpPointsX>**, ***integer, optional field***, sets the number of points need to be used for interpolation of the x axis.

- **<colorMap>** vector is the array to visualize. If **<colorMap>** is defined then the color set used can be defined by **<cmap>**. If no **<cmap>** is given then the plot is in solid color (default *blue*) as defined with **<color>** in **<kwargs>**.

- **<cmap>**, ***string, optional field***, defines the color map to use for this plot.
  Default: None

- **<kwargs>**, within this block the user can specify optional parameters with the following format:

```
<kwargs>
 <param1>value1</param1>
 <param2>value2</param2>
</kwargs>
```

The kwargs block is able to convert whatever string into a python type (for example **<param1>** `{'1stKey':45}` **</param1>** will be converted into a dictionary, **<param2>** `[56,67]` **</param2>** into a list, etc.). For reference regarding the available kwargs, see "matplotlib.pyplot.contour" method in [3].

14.3.9 3D Surface Plot

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

- **<rstride>**, ***integer, optional field***, specifies the array row stride (step size).
  Default: 1
- **<cstride>**, ***integer, optional field***, specifies the array column stride (step size).
  Default: 1
- **<cmap>**, ***string, optional field***, defines the color map to use for this plot.
  Default: None **Note:** If **<colorMap>** is defined then the plot will always use a color set even if no **<cmap>** is given. In such a case, if no **<cmap>** is given, then the default color set of "matplotlib.pyplot.surface" method in [3] is used. If **<colorMap>** and **<cmap>** are both not defined then the plot is in solid color (*default blue*) as defined with **<color>** in **<kwargs>**.
- **<antialiased>**, ***boolean, optional field***, determines whether or not the rendering should be antialiased.
  Default: False
- **<linewidth>**, ***integer, optional field***, defines the widths of lines rendered on the plot.
  Default: 0
- **<interpolationType>**, ***string, optional field***, is the type of interpolation algorithm to use for the data. Available options are "nearest," "linear," "cubic," "multiquadric," "inverse," "gaussian," "Rbflinear," "Rbfcubic," "quintic," and "thin_plate."
  Default: linear
- **<interpPointsX>**, ***integer, optional field***, sets the number of points need to be used for interpolation of the x axis.

- **<interpPointsY>**, ***integer, optional field***, sets the number of points need to be used for interpolation of the y axis.

- **<kwargs>**, within this block the user can specify optional parameters with the following format:

```
<kwargs>
 <param1>value1</param1>
 <param2>value2</param2>
</kwargs>
```

  The kwargs block is able to convert whatever string into a python type (for example **<param1>** `{'1stKey':45}` **</param1>** will be converted into a dictionary, **<param2>** `[56,67]` **</param2>** into a list, etc.). For reference regarding the available kwargs, see "matplotlib.pyplot.surface" method in [3].

14.3.10 3D Wireframe Plot

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

- **<rstride>**, ***integer, optional field***, sets the array row stride (step size).
  Default: 1

- **<cstride>**, ***integer, optional field***, sets the array column stride (step size).
  Default: 1

- **<cmap>**, ***string, optional field***, defines the color map to use for this plot.
  Default: None **Note: <cmap>** is not applicable in the current version of MatPlotLib for wireframe plots. However, if the colorMap option is set then a surface plot is plotted with a transparency of 0.4 on top of wireframe to give a visual colormap. **Note:** If **<colorMap>** is defined then the plot will always use a color set even if no **<cmap>** is given. In such a case, if no **<cmap>** is given, then the default color set of "matplotlib.pyplot.surface" method in [3] is used. If **<colorMap>** and **<cmap>** are both not defined then the plot is in solid color (*default blue*) as defined with **<color>** in **<kwargs>**.

- **<interpolationType>**, ***string, optional field***, is the type of interpolation algorithm to use for the data. Available options are "nearest," "linear," "cubic," "multiquadric," "inverse," "gaussian," "Rbflinear," "Rbfcubic," "quintic," and "thin_plate."
  Default: linear

- **<interpPointsX>**, ***integer, optional field***, sets the number of points need to be used for interpolation of the x axis.

- **<interpPointsY>**, ***integer, optional field***, sets the number of points need to be used for interpolation of the y axis.

- **<kwargs>**, within this block the user can specify optional parameters with the following format:

```
<kwargs>
 <param1>value1</param1>
 <param2>value2</param2>
</kwargs>
```

  The kwargs block is able to convert whatever string into a python type (for example **<param1>** `{'1stKey':45}` **</param1>** will be converted into a dictionary, **<param2>** `[56,67]` **</param2>** into a list, etc.). For reference regarding the available kwargs, see "matplotlib.pyplot.wireframe" method in [3].

14.3.11 3D Tri-surface Plot

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

- **<color>**, ***string, optional field***, sets the color of the surface patches.
  Default: b

- **<shade>**, ***boolean, optional field***, determines whether to apply shading or not.
  Default: False

- **<cmap>**, ***string, optional field***, defines the color map to use for this plot.
  Default: None **Note:** If **<colorMap>** is defined then the plot will always use a color set even if no **<cmap>** is given. In such a case, if no **<cmap>** is given, then the default color set of "matplotlib.pyplot.trisurface" method in [3] is used. If **<colorMap>** and **<cmap>** are both not defined then the plot is in solid color (*default blue*) as defined with **<color>** in **<kwargs>**.

- **<kwargs>**, within this block the user can specify optional parameters with the following format:

```
<kwargs>
 <param1>value1</param1>
 <param2>value2</param2>
</kwargs>
```

  The kwargs block is able to convert whatever string into a python type (for example **<param1>** `{'1stKey':45}` **</param1>** will be converted into a dictionary, **<param2>** `[56,67]`

</param2> into a list, etc.). For reference regarding the available kwargs, see "matplotlib.pyplot.trisurface" method in [3].

14.3.12 3D Contour or filledContour plots

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

- **<numberBins>**, ***integer, optional field***, sets the number of bins to use.
  Default: 5
- **<interpolationType>**, ***string, optional field***, is the type of interpolation algorithm to use for the data. Available options are "nearest," "linear," "cubic," "multiquadric," "inverse," "gaussian," "Rbflinear," "Rbfcubic," "quintic," and "thin_plate."
  Default: linear
- **<interpPointsX>**, ***integer, optional field***, sets the number of points need to be used for interpolation of the x axis.
- **<interpPointsY>**, ***integer, optional field***, sets the number of points need to be used for interpolation of the y axis.
- **<colorMap>** vector is the array to visualize. If **<colorMap>** is defined then the color set used can be defined by **<cmap>**. If no **<cmap>** is given then the plot is in solid color (default *blue*) as defined with **<color>** in **<kwargs>**.
- **<kwargs>**, within this block the user can specify optional parameters with the following format:

```
<kwargs>
 <param1>value1</param1>
 <param2>value2</param2>
</kwargs>
```

  The kwargs block is able to convert whatever string into a python type (for example **<param1>** `{'1stKey':45}`**</param1>** will be converted into a dictionary, **<param2>** `[56,67]` **</param2>** into a list, etc.). For reference regarding the available kwargs, see "matplotlib.pyplot.contour3d" method in [3].

14.3.13 DataMining plots

In order to create a "DataMining" plot, the user needs to write in the **<type>** body the keyword "dataMining". "DataMining" plots are based on 2D or 3D Scattering plots, depending on the method/algorithm used in the "DataMining" postprocessor [see 15.5.9]. These plots are created to ease the color labeling the clusters, etc parameters in the data. The following are the optional or required input parameters that can be used in these plots additional to the coordinate inputs **<x>**, **<y>**, or **<z>** depending on the dimension:

- **<type>**, ***string, required field***, this block should read "dataMining" in order to create a data mining plot.
- **<SKLtype>**, ***string, required field***, name of the algorithm used in the "dataMining" postprocessor. It is one of:
  - cluster: for clustering algorithms, such as KMeans clustering.
  - bicluster (**Note:** not implemented yet!)
  - mixture: for Gaussian mixture algorithms, such as GMM classifier
  - manifold: for Manifold Learning algorithms, such as Spectral Embedding
  - decomposition: for decomposing signals in components algorithms, such as Principal Component Analysis (PCA)
- **<clusterLabels>**, ***string, optional field***, defines the place where the labels of the clusters are located. As in the visual parameters (i.e., x,y,z and colorMap) this is also from a **DataObjects** object. Considering the structure of "DataObjects", the labels inputted as follows: `DataObjectName|Output|DataMiningPPNameLabels`.
  Default: None
- **<noClusters>**, ***integer, optional field***, defines the number of clusters used in the "dataMining" postprocessor
  Default: 1
- **<kwargs>**, within this block the user can specify optional parameters with the following format:

```
<kwargs>
 <param1>value1</param1>
 <param2>value2</param2>
</kwargs>
```

The kwargs block is able to convert whatever string into a python type (for example **<param1>** `{'1stKey':45}` **</param1>** will be converted into a dictionary, **<param2>** `[56,67]` **</param2>** into a list, etc.).

For reference regarding the other available kwargs, see “matplotlib.pyplot.scatter” method in [3].

14.3.14 Example XML input

```
<OutStreams>
  <Plot name='2DHistoryPlot' interactive='False'
     overwrite='False'>
    <actions>
      <how>pdf,png,eps</how>
      <title>
        <text>***</text>
      </title>
    </actions>
    <plotSettings>
       <plot>
        <type>line</type>
        <x>stories|Output|time</x>
        <y>stories|Output|pipe1_Hw</y>
        <kwargs>
         <color>green</color>
         <label>pipe1-Hw</label>
        </kwargs>
      </plot>
       <plot>
        <type>line</type>
        <x>stories|Output|time</x>
        <y>stories|Output|pipe1_aw</y>
        <kwargs>
         <color>blue</color>
         <label>pipe1-aw</label>
        </kwargs>
      </plot>
      <xlabel>time [s]</xlabel>
      <ylabel>evolution</ylabel>
    </plotSettings>
  </Plot>
</OutStreams>
```

14.4 Specific Plots

For convenience, RAVEN offers tailored plotting strategies that apply in specific circumstances. They are not as flexible as the default plotting system, but may offer a quick method to view data with minimal input required. These are described in the following subsection.

Specific plotting strategies are requested using the **<Plot>** node attribute **subType**. For example,

```
<Simulation>
  ...
  <OutStreams>
    <Plot name="mySamples" subType="PlottingStrategyName">
      ...
    </Plot>
  </OutStreams>
</Simulations>
```

14.4.1 SamplePlot

The **'SamplePlot'** **subType** is a very simple plotting tool meant for quickly viewing the results of sampling, or as a basis for adding new plotting strategies. This plotter constructs a series of verticle plots whose x-axis is the sample ID and y-axis is a particular variables' values.

The **'SamplePlot'** **subType** requires the following nodes:

- **<source>**, ***required, string***, selects which DataObject should be used to take data for plotting. This DataObject must be listed in the Step in which this OutStream is used, and is usually the result of as **<MultiRun>** step.
- **<vars>**, ***required, comma-separated strings***, identifies which variables should be plotted as a function of sample number. These variables must be included in the **<source>** DataObject.

For example,

```
<Simulation>
  ...
  <OutStreams>
    <Plot name="mySamples" subType="SamplePlot">
      <source>dataObjectFromStep</source>
      <vars>a, b, x, y</vars>
```

```
    </Plot>
  </OutStreams>
</Simulations>
```

15 Models

In RAVEN, **Models** are important entities. A model is an object that employs a mathematical representation of a phenomenon, either of a physical or other nature (e.g. statistical operators, etc.). From a practical point of view, it can be seen, as a "black box" that, given an input, returns an output.

RAVEN has a strict classification of the different types of models. Each "class" of models is represented by the definition reported above, but it can be further classified based on its particular functionalities:

- **`<Code>`** represents an external system code that employs a high fidelity physical model.
- **`<Dummy>`** acts as "transfer" tool. The only action it performs is transferring the the information in the input space (inputs) into the output space (outputs). For example, it can be used to check the effect of a sampling strategy, since its outputs are the sampled parameters' values (input space) and a counter that keeps track of the number of times an evaluation has been requested.
- **`<ROM>`**, or reduced order model, is a 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 phenomenon. The "training" process is performed by sampling the response of a physical model with respect to variation of its parameters subject to probabilistic behavior. The results (outcomes of the physical model) of the sampling are fed into the algorithm representing the ROM that tunes itself to replicate those results.
- **`<ExternalModel>`**, as its name suggests, is an entity existing outside the RAVEN framework that is embedded in the RAVEN code at run time. This object allows the user to create a Python module that will be treated as a predefined internal model object.
- **`<EnsembleModel>`** is model that is able to combine **Code**, **ExternalModel** and **ROM** models. It is aimed 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.
- **`<PostProcessor>`** is a container of all the actions that can manipulate and process a data object in order to extract key information, such as statistical quantities, clustering, etc.

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

Example:

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

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

15.1 Code

The **Code** model represents an external system software employing a high fidelity physical model. The link between RAVEN and the driven code is performed at run-time, through coded interfaces that are the responsible for transferring information from the code to RAVEN and vice versa. In Section 19, all of the available interfaces are reported and, for advanced users, Section 20 explains how to couple a new code.

The specifications of this model must be defined within a `<Code>` XML block. This XML node accepts the following attributes:

- `name`, ***required string attribute***, user-defined identifier of this model. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.
- `subType`, ***required string attribute***, specifies the code that needs to be associated to this Model. **Note:** See Section 19 for a list of currently supported codes.

This model can be initialized with the following children:

- `<executable>` ***string, required field*** specifies the path of the executable to be used.
- `<walltime>` ***string, optional field*** specifies the maximum allowed run time of the code; if the code running time is greater than the specified walltime then the code run is stopped. The stopped run is then considered as if it crashed. **Note:** Both absolute and relative path can be used. In addition, the relative path to the working directory can also be used.
- `<preexec>` ***string, optional field*** specifies the path of pre-executable to be used. **Note:** Both absolute and relative path can be used. In addition, the relative path to the working directory can also be used.

- **<alias>** ***string, optional field*** specifies alias for any variable of interest in the input or output space for the Code. These aliases can be used anywhere in the RAVEN input to refer to the Code variables. 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.
  Default: None

- **<clargs>** ***string, optional field*** allows addition of command-line arguments to the execution command. If the code interface specified in **<Code> subType** does not specify how to determine the input file(s), this node must be used to specify them. There are several types of **<clargs>**, based on the **type**:

  - **type** ***string, required field*** specifies the type of command-line argument to add. Options include **'input'**, **'output'**, **'prepend'**, **'postpend'**, **'text'**, and **'python'**.
  - **arg** ***string, optional field*** specifies the flag to be used before the entry. For example, **arg='-i'** would place a `-i` before the entry in the execution command. Required for the **'output' type**.
  - **extension** ***string, optional field*** specifies the type of file extension to use (for example, `-i` or `-o`). This links the **<Input>** file in the **<Step>** to this location in the execution command. Required for **'input' type**.
  - **delimiter** ***string, optional field*** specifies the delimiter that is used between the **arg** and the provided input file with the extension given by **extension**. **Note:** This is currently only used to link the **arg** and input file. i.e. the **type** should be **'input'** in order to use this feature.

  The execution command is combined in the order **'prepend'**, **<python> <executable>**, **'input'**, **'output'**, **'text'**, **'postpend'**. The **'python'** is a special type that puts the name of the python command.

- **<fileargs>** ***string, optional field*** like **<clargs>**, but allows editing of input files to specify the output filename and/or auxiliary file names. The location in the input files to edit using these arguments are identified in the input file using the prefix-postfix notation, which defaults to `$RAVEN-var$` for variable keyword *var*. The variable keyword is then listed in the **<fileargs>** node in the attribute **arg** to couple it in Raven. If the code interface specified in **<Code> subType** does not specify how to name the output file, that must be specified either through **<clargs>** or **<filargs>**, with **type 'output'**. The attributes required for **<fileargs>** are as follows:

  - **type** ***string, required field*** specifies the type of entry to replace in the file. Possible values for **<fileargs> type** are **'input'** and **'output'**.

- `arg` ***string, required field*** specifies the Raven variable with which to replace the file of interest. This should match the entry in the template input file; that is, if `$RAVEN-auxinp$` is in the input file, the arg for the corresponding input file should be **`'auxinp'`**.
- `extension` ***string, optional field*** specifies the extension of the input file that should replace the Raven variable in the input file. This attribute is required for the **`'input'`** `type` and ignored for the **`'output'`** `type`. **Note:** Currently, there can only be a one-to-one pairing between input files and extensions; that is, multiple Raven-editable input files cannot have the same extension.

Example:

```
<Simulation>
  ...
  <Models>
    ...
    <Code name='aUserDefinedName' subType='RAVEN_Driven_code'>
      <executable>path_to_executable</executable>
      <alias variable='internal_RAVEN_input_variable_name1'
         type="input">
         External_Code_input_Variable_Name_1
      </alias>
      <alias variable='internal_RAVEN_input_variable_name2'
         type='input'>
         External_Code_input_Variable_Name_2
      </alias>
      <alias variable='internal_RAVEN__output_variable_name'
         type='output'>
         External_Code_output_Variable_Name_2
      </alias>
      <clargs type='prepend' arg='python'/>
      <clargs type='input' arg='-i' extension='.i'/>
      <fileargs type='input' arg='aux' extension='.two'
      <fileargs type='output' arg='out' />
    </Code>
    ...
  </Models>
  ...
</Simulation>
```

15.2 Dummy

The **Dummy** model is an object that acts as a pass-through tool. The only action it performs is transferring the information in the input space (inputs) to the output space (outputs). For example, it can be used to check the effect of a particular sampling strategy, since its outputs are the sampled parameters' values (input space) and a counter that keeps track of the number of times an evaluation has been requested.

The specifications of this model must be defined within a `<Dummy>` XML block. . This XML node accepts the following attributes:

- `name`, ***required string attribute***, user-defined identifier of this model. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.
- `subType`, ***required string attribute***, this attribute must be kept empty.

This model can be initialized with the following children:

- `<alias>` ***string, optional field*** specifies alias for any variable of interest in the input or output space for the Dummy. These aliases can be used anywhere in the RAVEN input to refer to the Dummy variables. 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.
  Default: None
  Since the **Dummy** model represents a transfer function only, the usage of the alias is relatively meaningless.

Given a particular *Step* using this model, if this model is linked to a *Data* with the role of **Output**, it expects one of the output parameters will be identified by the keyword "OutputPlaceHolder" (see Section 18).

Example:

```
<Simulation>
  ...
  <Models>
    ...
    <Dummy name='aUserDefinedName1' subType=''/>

    <Dummy name='aUserDefinedName2' subType=''>
      <alias variable="a_RAVEN_input_variable" type="input">
```

```
      another_name_for_this_variable_in_the_model
      </alias>
    </Dummy>
    ...
  </Models>
  ...
</Simulation>
```

15.3 ROM

A Reduced Order Model (ROM) is a mathematical model consisting of a fast solution trained to predict a response of interest of a physical system. The "training" process is performed by sampling the response of a physical model with respect to variations of its parameters subject, for example, to probabilistic behavior. The results (outcomes of the physical model) of the sampling are fed into the algorithm representing the ROM that tunes itself to replicate those results. RAVEN supports several different types of ROMs, both internally developed and imported through an external library called "scikit-learn" [4].

Currently in RAVEN, the ROMs are classified into several sub-types that, once chosen, provide access to several different algorithms. These sub-types are specified in the **subType** attribute and should be one of the following:

- **'GaussPolynomialRom'**, for both static and time-dependent regression
- **'HDMRRom'**, for both static and time-dependent regression
- **'NDinvDistWeight'**, for both static and time-dependent regression
- **'NDSpline'**, for both static and time-dependent regression
- **'SciKitLearn'**, for both static and time-dependent regression and classification
- **'MSR'**, for both static and time-dependent regression
- **'ARMA'**, for time-dependent stochastic regression (time series generator)
- **'PolyExponential'**, for time-dependent regression
- **'DMD'**, for time-dependent regression
- **'KerasMLPClassifier'**, for deep neuron network

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

- `name`, ***required string attribute***, user-defined identifier of this model. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.
- `subType`, ***required string attribute***, defines which of the sub-types should be used, choosing among the previously reported types. This choice conditions the subsequent the required and/or optional `<ROM>` sub-nodes.

In the `<ROM>` input block, the following XML sub-nodes are required, 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 (see Section 18.4);
- `<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 (see Section 18.4).

If a time-dependent ROM is requested, a **HistorySet** should be provided. The temporal vairable specified in the **HistorySet** should be also listed as sub-nodes inside `<ROM>`

- `<pivotParameter>`, ***string, optional parameter***, specifies the pivot variable (e.g. time, etc) used in the input HistorySet.
  Default: time

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

- `<alias>` ***string, optional field*** specifies alias for any variable of interest in the input or output space for the ROM. These aliases can be used anywhere in the RAVEN input to refer to the ROM variables. 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.
  Default: None

The types and meaning of the remaining sub-nodes depend on the sub-type specified in the attribute `subType`.

Note that 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.

It is important to NOTE that RAVEN uses a Z-score normalization of the training data before constructing most of the Reduced Order Models (e.g. most of the SciKitLearn-based ROMs):

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

In the following sections the specifications of each ROM type are reported, highlighting when a **Z-score normalization** is performed by RAVEN before constructing the ROM or when it is not performed.

15.3.1 NDspline

The NDspline sub-type contains a single ROM type, based on an N-dimensional spline interpolation/extrapolation scheme. In spline interpolation, the regressor is a special type of piecewise polynomial called tensor spline. The interpolation error can be made small even when using low degree polynomials for the spline. Spline interpolation avoids the problem of Runge's phenomenon, in which oscillation can occur between points when interpolating using higher degree polynomials.

In order to use this ROM, the **<ROM>** attribute **subType** needs to be **'NDspline'** (see the example below). No further XML sub-nodes are required. **Note:** This ROM type must be trained from a regular Cartesian grid. Thus, it can only be trained from the outcomes of a grid sampling strategy.

It is important to NOTE that RAVEN uses a Z-score normalization of the training data before constructing the *NDspline* ROM:

$$\mathbf{X}' = \frac{(\mathbf{X} - \mu)}{\sigma} \tag{4}$$

Example:

```
<Simulation>
  ...
  <Models>
    ...
    <ROM name='aUserDefinedName' subType='NDspline'>
       <Features>var1,var2,var3</Features>
```

```
      <Target>result1,result2</Target>
    </ROM>
    ...
  </Models>
  ...
</Simulation>
```

15.3.2 pickledROM

It is not uncommon for a reduced-order model (ROM) to be created and trained in one RAVEN run, then serialized to file (*pickled*), then loaded into another RAVEN run to be used as a model. When this is the case, 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.

Note that 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 know 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 pickledROM is not usable. It cannot 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.

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>
```

15.3.3 GaussPolynomialRom

The GaussPolynomialRom sub-type contains a single ROM type, based on a characteristic Gaussian polynomial fitting scheme: generalized polynomial chaos expansion (gPC). In gPC, sets of polynomials orthogonal with respect to the distribution of uncertainty are used to represent the original model. The method converges moments of the original model faster than Monte Carlo for small-dimension uncertainty spaces ($N < 15$). In order to use this ROM, the **<ROM>** attribute **subType** needs to be **'GaussPolynomialRom'** (see the example below). The GaussPolynomialRom is dependent on specific sampling; thus, this ROM cannot be trained unless a SparseGridCollocation or similar Sampler specifies this ROM in its input and is sampled in a MultiRun step. In addition to the common **<Target>** and **<Features>**, this ROM requires two more nodes and can accept multiple entries of a third optional node.

- **<IndexSet>**, ***string, required field***, specifies the rules by which to construct multidimensional polynomials. The options are **'TensorProduct'**, **'TotalDegree'**, **'HyperbolicCross'**, and **'Custom'**. Total degree is efficient for uncertain inputs with a large degree of regularity, while hyperbolic cross is more efficient for low-regularity input spaces. If **'Custom'** is chosen, the **<IndexPoints>** is required.

- **<PolynomialOrder>**, ***integer, required field***, indicates the maximum polynomial order in any one dimension to use in the polynomial chaos expansion. **Note:** If non-equal importance weights are supplied in the optional **<Interpolation>** node, the actual polynomial order in dimensions with high importance might exceed this value; however, this value is still used to limit the relative overall order.

- **<SparseGrid>**,***string, optional field***, allows specification of the multidimensional quadrature construction strategy. Options are **'smolyak'** and **'tensor'**. Default is **'smolyak'**.

- **<IndexPoints>**, ***list of tuples, required field***, used to specify the index set points in a **'Custom'** index set. The tuples are entered as comma-seprated values between parenthesis, with each tuple separated by a comma. Any amount of whitespace is acceptable. For example, **<IndexPoints>**`(0,1),(0,2),(1,1),(4,0)`**</IndexPoints>** **Note:** Using custom index sets does not guarantee accurate convergence.

- **<Interpolation>**, ***string, optional field***, offers the option to specify quadrature, polynomials, and importance weights for the given variable name. The ROM accepts any number

of **<Interpolation>** nodes up to the dimensionality of the input space. This node accepts several attributes, all of which are optional and default to the code-defined optimal choices based on the input dimension uncertainty distribution:

- **quad**, ***string, optional field***, specifies the quadrature type to use for collocation in this dimension. The default options depend on the uncertainty distribution of the input dimension, as shown in Table 3. Additionally, Clenshaw Curtis quadrature can be used for any distribution that doesn't include an infinite bound.
  Default: see Table 3. **Note:** For an uncertain distribution aside from the four listed on Table 3, this ROM makes use of the uniform-like range of the distribution's CDF to apply quadrature that is suited uniform uncertainty (Legendre). It converges more slowly than the four listed, but are viable choices. Choosing polynomial type Legendre for any non-uniform distribution will enable this formulation automatically.
- **poly**, ***string,optional field***, specifies the interpolating polynomial family to use for the polynomial expansion in this dimension. The default options depend on the quadrature type chosen, as shown in Table 3. Currently, no polynomials are available outside the default.
  Default: see Table 3.
- **weight**, ***float, optional field***, delineates the importance weighting of this dimension. A larger importance weight will result in increased resolution for this dimension at the cost of resolution in lower-weighted dimensions. The algorithm normalizes weights at run-time.
  Default: 1.

Unc. Distribution	Default Quadrature	Default Polynomials
Uniform	Legendre	Legendre
Normal	Hermite	Hermite
Gamma	Laguerre	Laguerre
Beta	Jacobi	Jacobi
Other	Legendre*	Legendre*

Table 3: GaussPolynomialRom defaults

Note: This ROM type must be trained from a collocation quadrature set. Thus, it can only be trained from the outcomes of a SparseGridCollocation sampler. Also, this ROM must be referenced in the SparseGridCollocation sampler in order to accurately produce the necessary sparse grid points to train this ROM.

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *GaussPolynomialRom* ROM.

Example:

```
<Simulation>
  ...
  <Samplers>
    ...
    <SparseGridCollocation name="mySG" parallel="0">
      <variable name="x1">
        <distribution>myDist1</distribution>
      </variable>
      <variable name="x2">
        <distribution>myDist2</distribution>
      </variable>
      <ROM class = 'Models' type = 'ROM' >myROM</ROM>
    </SparseGridCollocation>
    ...
  </Samplers>
  ...
  <Models>
    ...
    <ROM name='myRom' subType='GaussPolynomialRom'>
      <Target>ans</Target>
      <Features>x1,x2</Features>
      <IndexSet>TotalDegree</IndexSet>
      <PolynomialOrder>4</PolynomialOrder>
      <Interpolation quad='Legendre' poly='Legendre'
         weight='1'>x1</Interpolation>
      <Interpolation quad='ClenshawCurtis' poly='Jacobi'
         weight='2'>x2</Interpolation>
    </ROM>
    ...
  </Models>
  ...
</Simulation>
```

When Printing this ROM via a Print OutStream (see 14.2), the available metrics are:

- **'mean'**, the mean value of the ROM output within the input space it was trained,
- **'variance'**, the variance of the ROM output within the input space it was trained,
- **'samples'**, the number of distinct model runs required to construct the ROM,
- **'indices'**, the Sobol sensitivity indices (in percent), Sobol total indices, and partial variances,
- **'polyCoeffs'**, the polynomial expansion coefficients (PCE moments) of the ROM. These are listed by each polynomial combination, with the polynomial order tags listed in the order of the variables shown in the XML print.

15.3.4 HDMRRom

The HDMRRom sub-type contains a single ROM type, based on a Sobol decomposition scheme. In Sobol decomposition, also known as high-density model reduction (HDMR, specifically Cut-HDMR), a model is approximated as as the sum of increasing-complexity interactions. At its lowest level (order 1), it treats the function as a sum of the reference case plus a functional of each input dimesion separately. At order 2, it adds functionals to consider the pairing of each dimension with each other dimension. The benefit to this approach is considering several functions of small input cardinality instead of a single function with large input cardinality. This allows reduced order models like generalized polynomial chaos (see 15.3.3) to approximate the functionals accurately with few computations runs. In order to use this ROM, the **`<ROM>`** attribute **`subType`** needs to be **`'HDMRRom'`** (see the example below). The HDMRRom is dependent on specific sampling; thus, this ROM cannot be trained unless a Sobol or similar Sampler specifies this ROM in its input and is sampled in a MultiRun step. In addition to the common **`<Target>`** and **`<Features>`**, this ROM requires the same nodes as the GaussPolynomialRom (see 15.3.3. Additionally, this ROM requires the **`<SobolOrder>`** node.

- **`<SobolOrder>`**, ***integer, required field***, indicates the maximum cardinality of the input space used in the subset functionals. For example, order 1 includes only functionals of each independent dimension separately, while order 2 considers pair-wise interactions.

Note: This ROM type must be trained from a Sobol decomposition training set. Thus, it can only be trained from the outcomes of a Sobol sampler. Also, this ROM must be referenced in the Sobol sampler in order to accurately produce the necessary sparse grid points to train this ROM. Experience has shown order 2 Sobol decompositions to include the great majority of uncertainty in most models.

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *HDMRRom* ROM.

Example:

```
<Samplers>
  ...
  <Sobol name="mySobol" parallel="0">
    <variable name="x1">
      <distribution>myDist1</distribution>
    </variable>
    <variable name="x2">
      <distribution>myDist2</distribution>
    </variable>
    <ROM class = 'Models' type = 'ROM' >myHDMR</ROM>
  </Sobol>
  ...
```

```
</Samplers>
...
<Models>
  ...
  <ROM name='myHDMR' subType='HDMRRom'>
    <Target>ans</Target>
    <Features>x1,x2</Features>
    <SobolOrder>2</SobolOrder>
    <IndexSet>TotalDegree</IndexSet>
    <PolynomialOrder>4</PolynomialOrder>
    <Interpolation quad='Legendre' poly='Legendre'
       weight='1'>x1</Interpolation>
    <Interpolation quad='ClenshawCurtis' poly='Jacobi'
       weight='2'>x2</Interpolation>
  </ROM>
  ...
</Models>
```

When Printing this ROM via an OutStream (see 14.2), the available metrics are:

- **'mean'**, the mean value of the ROM output within the input space it was trained,
- **'variance'**, the ANOVA-calculated variance of the ROM output within the input space it was trained.
- **'samples'**, the number of distinct model runs required to construct the ROM,
- **'indices'**, the Sobol sensitivity indices (in percent), Sobol total indices, and partial variances.

15.3.5 MSR

The MSR sub-type contains a class of ROMs that perform a topological decomposition of the data into approximately monotonic regions and fits weighted linear patches to the identified monotonic regions of the input space. Query points have estimated probabilities that they belong to each cluster. These probabilities can eitehr be used to give a smooth, weighted prediction based on the associated linear models, or a hard classification to a particular local linear model which is then used for prediction. Currently, the probability prediction can be done using kernel density estimation (KDE) or through a one-versus-one support vector machine (SVM).

In order to use this ROM, the **<ROM>** attribute **subType** needs to be **'MSR'** (see the associated example). This model can be initialized with the following children:

- **<persistence>**, ***string, optional field***, specifies how to define the hierarchical simplification by assigning a value to each local minimum and maximum according to the one of

the strategy options below:

- `difference` - The function value difference between the extremum and its closest-valued neighboring saddle.
- `probability` - The probability integral computed as the sum of the probability of each point in a cluster divided by the count of the cluster.
- `count` - The count of points that flow to or from the extremum.

Default: `difference`

- **`<gradient>`**, ***string, optional field***, specifies the method used for estimating the gradient, available options are:
  - `steepest`

  Default: `steepest`

- **`<simplification>`**, ***float, optional field***, specifies the amount of noise reduction to apply before returning labels.
  Default: 0

- **`<graph>`** , ***string, optional field***, specifies the type of neighborhood graph used in the algorithm, available options are:
  - `beta skeleton`
  - `relaxed beta skeleton`
  - `approximate knn`

  Default: `beta skeleton`

- **`<beta>`**, ***float in the range: (0,2], optional field***, is only used when the **`<graph>`** is set to `beta skeleton` or `relaxed beta skeleton`.
  Default: 1.0

- **`<knn>`**, ***integer, optional field***, is the number of neighbors when using the `approximate knn` for the **`<graph>`** sub-node and used to speed up the computation of other graphs by using the approximate knn graph as a starting point for pruning. -1 means use a fully connected graph.
  Default: -1

- **`<partitionPredictor>`**, ***string, optional***, a flag that specifies how the predictions for query point classification should be performed. Available options are:

- `kde`
- `svm`

Default: kde

- **`<smooth>`**, if this node is present, the ROM will blend the estimates of all of the local linear models weighted by the probability the query point is classified as belonging to that partition of the input space.
- **`<kernel>`**, ***string, optional field***, this option is only used when the **`<partitionPredictor>`** is set to `kde` and specifies the type of kernel to use in the kernel density estimation. Available options are:
  - `uniform`
  - `triangular`
  - `gaussian`
  - `epanechnikov`
  - `biweight` or `quartic`
  - `triweight`
  - `tricube`
  - `cosine`
  - `logistic`
  - `silverman`
  - `exponential`

  Default: gaussian
- **`<bandwidth>`**, ***float or string, optional field***, this option is only used when the **`<partitionPredicto`** is set to `kde` and specifies the scale of the fall-off. A higher bandwidth implies a smooother blending. If set to `variable`, then the bandwidth will be set to the distance of the k-nearest neighbor of the query point where k is set by the **`<knn>`** parameter.
  Default: 1.

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *MSR* ROM.

Example:

```
<Simulation>
  ...
  <Models>
    ...
    </ROM>
    <ROM name='aUserDefinedName' subType='MSR'>
       <Features>var1,var2,var3</Features>
       <Target>result1,result2</Target>
       <!-- <weighted>true</weighted> -->
       <simplification>0.0</simplification>
       <persistence>difference</persistence>
       <gradient>steepest</gradient>
       <graph>beta skeleton</graph>
       <beta>1</beta>
       <knn>8</knn>
       <partitionPredictor>kde</partitionPredictor>
       <kernel>gaussian</kernel>
       <smooth/>
       <bandwidth>0.2</bandwidth>
     </ROM>
    ...
  </Models>
  ...
</Simulation>
```

15.3.6 NDinvDistWeight

The NDinvDistWeight sub-type contains a single ROM type, based on an N-dimensional inverse distance weighting formulation. Inverse distance weighting (IDW) is a type of deterministic method for multivariate interpolation with a known scattered set of points. The assigned values to unknown points are calculated via a weighted average of the values available at the known points.

In order to use this Reduced Order Model, the **<ROM>** attribute **subType** needs to be xmlStringNDinvDistWeight (see the example below). This model can be initialized with the following child:

- **<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).

It is important to NOTE that RAVEN uses a Z-score normalization of the training data before constructing the *NDinvDistWeight* ROM:

$$\mathbf{X}' = \frac{(\mathbf{X} - \mu)}{\sigma} \tag{5}$$

Example:

```
<Simulation>
  ...
  <Models>
    ...
    <ROM name='aUserDefinedName' subType='NDinvDistWeight'>
      <Features>var1,var2,var3</Features>
      <Target>result1,result2</Target>
      <p>3</p>
     </ROM>
    ...
  </Models>
  ...
</Simulation>
```

15.3.7 SciKitLearn

The SciKitLearn sub-type represents the container of several ROMs available in RAVEN through the external library scikit-learn [4].

In order to use this Reduced Order Model, the **`<ROM>`** attribute **`subType`** needs to be **`'SciKitLearn'`** (i.e. **`subType='SciKitLearn'`**). The specifications of a **`'SciKitLearn'`** ROM depend on the value assumed by the following sub-node within the main **`<ROM>`** XML node:

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

Based on the **<SKLtype>** several different algorithms are available. **Note:** for HistorySet's **'SciKitLearn'** performs the task given in **<SKLtype>** for each time step, and only synchronized HistorySet can be used as input to this ROM. For unsynchronized HistorySet, use **'HistorySetSync'** method in **'Interfaced'** post-processor to synchronize the input data before using **'SciKitLearn'**.

In the following paragraphs a brief explanation and the input requirements are reported for each of them.

15.3.7.1 Linear Models

The LinearModels' algorithms implement generalized linear models. They include Ridge regression, Bayesian regression, lasso, and elastic net estimators computed with least angle regression and coordinate descent. This class also implements stochastic gradient descent related algorithms. In the following, all of the linear models available in RAVEN are reported.

15.3.7.1.1 Linear Model: Automatic Relevance Determination Regression

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

<SKLtype>`linear_model|ARDRegression`**</SKLtype>**.

In addition to this XML node, several others are available: .

- **<n_iter>**, ***integer, optional field***, is the maximum number of iterations.
  Default: 300
- **<tol>**, ***float, optional field***, stop the algorithm if the convergence error felt below the tolerance specified here.
  Default: 1.e-3
- **<alpha_1>**, ***float, optional field***, is a shape hyperparameter for the Gamma distribution prior over the α parameter.
  Default: 1.e-6

- **<alpha_2>**, ***float, optional field***, inverse scale hyperparameter (rate parameter) for the Gamma distribution prior over the α parameter.
  Default: 1.e-6

- **<lambda_1>**, ***float, optional field***, shape hyperparameter for the Gamma distribution prior over the λ parameter.
  Default: 1.e-6

- **<lambda_2>**, ***float, optional field***, inverse scale hyperparameter (rate parameter) for the Gamma distribution prior over the λ parameter.
  Default: 1.e-6

- **<compute_score>**, ***boolean, optional field***, if True, compute the objective function at each step of the model.
  Default: False

- **<threshold_lambda>**, ***float, optional field***, specifies the threshold for removing (pruning) weights with high precision from the computation.
  Default: 1.e+4

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

- **<normalize>**, ***boolean, optional field***, if True, the regressors X will be normalized before regression.
  Default: False

- **<verbose>**, ***boolean, optional field***, use verbose mode when fitting the model.
  Default: False

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *ARDRegression* ROM.

15.3.7.1.2 Linear Model: Bayesian ridge regression

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

$$p(w|\lambda) = \mathcal{N}(w|0, \lambda^{-1}\mathbf{I_p}) \tag{6}$$

The priors over α and λ are chosen to be gamma distributions, the conjugate prior for the precision of the Gaussian. The resulting model is called Bayesian ridge regression, and is similar to the

classical ridge regression. The parameters w, α, and λ are estimated jointly during the fit of the model. The remaining hyperparameters are the parameters of the gamma priors over α and λ. These are usually chosen to be non-informative. The parameters are estimated by maximizing the marginal log likelihood. In order to use the *Bayesian ridge regressor*, the user needs to set the sub-node:

<SKLtype>`linear_model|BayesianRidge`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<n_iter>**, ***integer, optional field***, is the maximum number of iterations.
  Default: 300
- **<tol>**, ***float, optional field***, stop the algorithm if the convergence error felt below the tolerance specified here.
  Default: 1.e-3
- **<alpha_1>**, ***float, optional field***, is a shape hyperparameter for the Gamma distribution prior over the α parameter.
  Default: 1.e-6
- **<alpha_2>**, ***float, optional field***, inverse scale hyperparameter (rate parameter) for the Gamma distribution prior over the α parameter.
  Default: 1.e-6
- **<lambda_1>**, ***float, optional field***, shape hyperparameter for the Gamma distribution prior over the λ parameter.
  Default: 1.e-6
- **<lambda_2>**, ***float, optional field***, inverse scale hyperparameter (rate parameter) for the Gamma distribution prior over the λ parameter.
  Default: 1.e-6
- **<compute_score>**, ***boolean, optional field***, if True, compute the objective function at each step of the model.
  Default: False
- **<fit_intercept>**, ***boolean, optional field***, determines whether to calculate the intercept for this model. If set to False, no intercept will be used in the calculations (e.g. data is expected to be already centered).
  Default: True
- **<normalize>**, ***boolean, optional field***, if True, the regressors X will be normalized before regression.
  Default: False

- **<verbose>**, ***boolean, optional field***, use verbose mode when fitting the model.
  Default: False

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *BayesianRidge* ROM.

15.3.7.1.3 Linear Model: Elastic Net

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

$$1/(2*n_{samples})*||y-Xw||_2^2+alpha*l1_ratio*||w||_1+0.5*alpha*(1-l1_ratio)*||w||_2^2 \quad (7)$$

In order to use the *Elastic Net regressor*, the user needs to set the sub-node:

<SKLtype>`linear_model|ElasticNet`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<alpha>**,***float, optional field***, specifies a constant that multiplies the penalty terms. $alpha = 0$ is equivalent to an ordinary least square, solved by the **LinearRegression** object.
  Default: 1.0
- **<l1_ratio>**, ***float, optional field***, specifies the ElasticNet mixing parameter, with $0 <= l1_ratio <= 1$. For $l1_ratio = 0$ the penalty is an L2 penalty. For $l1_ratio = 1$ it is an L1 penalty. For $0 < l1_ratio < 1$, the penalty is a combination of L1 and L2.
  Default: 0.5
- **<fit_intercept>**, ***boolean, optional field***, determines whether to calculate the intercept for this model. If set to False, no intercept will be used in the calculations (e.g. data is expected to be already centered).
  Default: True
- **<normalize>**, ***boolean, optional field***, if True, the regressors X will be normalized before regression.
  Default: False
- **<max_iter>**, ***integer, optional field***, specifies the maximum number of iterations.
  Default: 1000
- **<tol>**, ***float, optional field***, specifies the tolerance for the optimization: if the updates are smaller than tol, the optimization code checks the dual gap for optimality and continues until it is smaller than tol.
  Default: 1.e-4

- **<warm_start>**, ***boolean, optional field***, when set to True, the model will reuse the solution of the previous call to fit as initialization, otherwise, it will just erase the previous solution.
  Default: False

- **<positive>**, ***boolean, optional field***, when set to True, this forces the coefficients to be positive.
  Default: False

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *ElasticNet* ROM.

15.3.7.1.4 Linear Model: Elastic Net CV

The *Elastic Net CV* is a linear regression similar to the Elastic Net model but with an iterative fitting along a regularization path. The best model is selected by cross-validation.

In order to use the *Elastic Net CV regressor*, the user needs to set the sub-node:

<SKLtype>`linear_model|ElasticNetCV`**</SKLtype>**.

In addition to this XML node, several others are available:

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

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

- **<n_alphas>**, ***integer, optional field***, is the number of alphas along the regularization path used for each $l1_ratio$.
  Default: 100

- **<precompute>**, ***boolean or string, optional field***, determines whether to use a precomputed Gram matrix to speed up calculations. If set to 'auto,' RAVEN will decide. The Gram matrix can also be passed as an argument. Available options are [True — False — 'auto' — array-like].
  Default: 'auto'

- **<max_iter>**, ***integer, optional field***, specifies the maximum number of iterations.
  Default: 1000

- **<tol>**, ***float, optional field***, specifies the tolerance for the optimization: if the updates are smaller than tol, the optimization code checks the dual gap for optimality and continues until it is smaller than tol.
  Default: 1.e-4

- **<positive>**, ***boolean, optional field***, when set to True, this forces the coefficients to be positive.
  Default: False

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *ElasticNetCV* ROM.

15.3.7.1.5 Linear Model: Least Angle Regression model

The *Least Angle Regression model* (LARS) is a regression algorithm for high-dimensional data. The LARS algorithm provides a means of producing an estimate of which variables to include, as well as their coefficients, when a response variable is determined by a linear combination of a subset of potential covariates.

In order to use the *Least Angle Regression model*, the user needs to set the sub-node:

<SKLtype>`linear_model|Lars`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<n_nonzero_coefs>**, ***integer, optional field***, represents the target number of non-zero coefficients.
  Default: 500

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

- **<verbose>**, ***boolean, optional field***, use verbose mode when fitting the model.
  Default: False

- **<precompute>**, ***boolean or string, optional field***, determines whether to use a precomputed Gram matrix to speed up calculations. If set to 'auto,' RAVEN will decide. The Gram matrix can also be passed as an argument. Available options are [True — False — 'auto' —

array-like].
Default: 'auto'

- **<normalize>**, ***boolean, optional field***, if True, the regressors X will be normalized before regression.
Default: True

- **<eps>**, ***float, optional field***, represents the machine precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems. Unlike the **<tol>** parameter in some iterative optimization-based algorithms, this parameter does not control the tolerance of the optimization.
Default: 2.2204460492503131e-16

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

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *Lars* ROM.

15.3.7.1.6 Linear Model: Cross-validated Least Angle Regression model

The *Cross-validated Least Angle Regression model* is a regression algorithm for high-dimensional data. It is similar to the LARS method, but the best model is selected by cross-validation. In order to use the *Cross-validated Least Angle Regression model*, the user needs to set the sub-node:

<SKLtype>`linear_model|LarsCV`**</SKLtype>**.

In addition to this XML node, several others are available:

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

- **<verbose>**, ***boolean, optional field***, use verbose mode when fitting the model.
Default: False

- **<normalize>**, ***boolean, optional field***, if True, the regressors X will be normalized before regression.
Default: True

- **<precompute>**, ***boolean or string, optional field***, determines whether to use a precomputed Gram matrix to speed up calculations. If set to 'auto,' RAVEN will decide. The Gram matrix can also be passed as an argument. Available options are [True — False — 'auto' — array-like].
  Default: 'auto'

- **<max_iter>**, ***integer, optional field***, specifies the maximum number of iterations.
  Default: 500

- **<max_n_alphas>**, ***integer, optional field***, specifies the maximum number of points on the path used to compute the residuals in the cross-validation.
  Default: 1000

- **<eps>**, ***float, optional field***, represents the machine-precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems. Unlike the *tol* parameter in some iterative optimization-based algorithms, this parameter does not control the tolerance of the optimization.
  Default: 2.2204460492503131e-16

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

The *Linear Model trained with L1 prior as regularizer (Lasso)* is a shrinkage and selection method for linear regression. It minimizes the usual sum of squared errors, with a bound on the sum of the absolute values of the coefficients. In order to use the *Linear Model trained with L1 prior as regularizer (Lasso)*, the user needs to set the sub-node:

<SKLtype>`linear_model|Lasso`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<alpha>**, ***float, optional field***, sets a constant multiplier for the L1 term. alpha = 0 is equivalent to an ordinary least square, solved by the LinearRegression object. For numerical reasons, using alpha = 0 with the Lasso object is not advised and you should instead use the LinearRegression object.
  Default: 1.0

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

- **<normalize>**, ***boolean, optional field***, if True, the regressors X will be normalized before regression.
  Default: False

- **<precompute>**, ***boolean or string, optional field***, determines whether to use a precomputed Gram matrix to speed up calculations. If set to 'auto,' RAVEN will decide. The Gram matrix can also be passed as an argument. Available options are [True — False — 'auto' — array-like].
  Default: False **Note:** For sparse input this option is always True to preserve sparsity.

- **<max_iter>**, ***integer, optional field***, specifies the maximum number of iterations.
  Default: 1000

- **<tol>**, ***float, optional field***, specifies the tolerance for the optimization: if the updates are smaller than tol, the optimization code checks the dual gap for optimality and continues until it is smaller than tol.
  Default: 1.e-4

- **<warm_start>**, ***boolean, optional field***, when set to True, the model will reuse the solution of the previous call to fit as initialization, otherwise, it will just erase the previous solution.
  Default: False

- **<positive>**, ***boolean, optional field***, when set to True, this forces the coefficients to be positive.
  Default: False

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *LarsCV* ROM.

15.3.7.1.8 Lasso linear model with iterative fitting along a regularization path

The *Lasso linear model with iterative fitting along a regularization path* is an algorithm of the Lasso family, that computes the linear regressor weights, identifying the regularization path in an iterative fitting (see http://www.jstatsoft.org/v33/i01/paper)

In order to use the *Lasso linear model with iterative fitting along a regularization path regressor*, the user needs to set the sub-node:

<SKLtype>`linear_model|LassoCV`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<eps>**, ***float, optional field***, represents the length of the path. eps=1e-3 means that alpha_min / alpha_max = 1e-3.
  Default: 1.0e-3

- **<n_alphas>**, ***int, optional field***, sets the number of alphas along the regularization path.
  Default: 100

- **<alphas>**, ***numpy array, optional field***, lists the locations of the alphas used to compute the models.
  Default: None If None, alphas are set automatically.

- **<precompute>**, ***boolean or string, optional field***, determines whether to use a precomputed Gram matrix to speed up calculations. If set to 'auto,' RAVEN will decide. The Gram matrix can also be passed as an argument. Available options are [True — False — 'auto' — array-like].
  Default: 'auto'

- **<max_iter>**, ***integer, optional field***, specifies the maximum number of iterations.
  Default: 1000

- **<tol>**, ***float, optional field***, specifies the tolerance for the optimization: if the updates are smaller than tol, the optimization code checks the dual gap for optimality and continues until it is smaller than tol.
  Default: 1.e-4

- **<verbose>**, ***boolean or integer, optional field***, use verbose mode when fitting the model.
  Default: False

- **<positive>**, ***boolean, optional field***, when set to True, this forces the coefficients to be positive.
  Default: False

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *LassoCV* ROM.

15.3.7.1.9 Lasso model fit with Least Angle Regression

Lasso model fit with Least Angle Regression (aka Lars) It is a Linear Model trained with an L1 prior as regularizer. In order to use the *Least Angle Regression model regressor*, the user needs to set the sub-node In order to use the *Least Angle Regression model regressor*, the user needs to set the sub-node:

<SKLtype>`linear_model|LassoLars`**</SKLtype>**.

In addition to this XML node, several others are available:

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

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

- **<verbose>**, ***boolean or integer, optional field***, use verbose mode when fitting the model.
  Default: False

- **<normalize>**, ***boolean, optional field***, if True, the regressors X will be normalized before regression.
  Default: True

- **<precompute>**, ***boolean or string, optional field***, determines whether to use a precomputed Gram matrix to speed up calculations. If set to 'auto,' RAVEN will decide. The Gram matrix can also be passed as an argument. Available options are [True — False — 'auto' — array-like].
  Default: 'auto'

- **<max_iter>**, ***integer, optional field***, specifies the maximum number of iterations.
  Default: 500

- **<eps>**, ***float, optional field***, sets the machine precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems.
  Default: 2.2204460492503131e-16

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *LassoLars* ROM.

15.3.7.1.10 Cross-validated Lasso, using the LARS algorithm

The *Cross-validated Lasso, using the LARS algorithm* is a cross-validated Lasso, using the LARS algorithm.

In order to use the *Cross-validated Lasso, using the LARS algorithm regressor*, the user needs to set the sub-node:

<SKLtype>`linear_model|LassoLarsCV`**</SKLtype>**.

In addition to this XML node, several others are available:

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

- **<verbose>**, ***boolean or integer, optional field***, use verbose mode when fitting the model.
  Default: False

- **<normalize>**, ***boolean, optional field***, if True, the regressors X will be normalized before regression.
  Default: True

- **<precompute>**, ***boolean or string, optional field***, determines whether to use a precomputed Gram matrix to speed up calculations. If set to 'auto,' RAVEN will decide. The Gram matrix can also be passed as an argument. Available options are [True — False — 'auto' — array-like].
  Default: 'auto'

- **<max_iter>**, ***integer, optional field***, specifies the maximum number of iterations.
  Default: 500

- **<max_n_alphas>**, ***integer, optional field***, specifies the maximum number of points on the path used to compute the residuals in the cross-validation.
  Default: 1000

- **<eps>**, ***float, optional field***, specifies the machine precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems.
  Default: 2.2204460492503131e-16

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *LassoLarsCV* ROM.

15.3.7.1.11 Lasso model fit with Lars using BIC or AIC for model selection

The *Lasso model fit with Lars using BIC or AIC for model selection* is a Lasso model fit with Lars using BIC or AIC for model selection. The optimization objective for Lasso is: $(1/(2 * n_samples)) * ||y - Xw||_2^2 + alpha * ||w||_1$ AIC is the Akaike information criterion and BIC is the Bayes information criterion. Such criteria are useful in selecting the value of the regularization parameter by making a trade-off between the goodness of fit and the complexity of the model. A good model explains the data well while maintaining simplicity. In order to use the *Lasso model fit with Lars using BIC or AIC for model selection regressor*, the user needs to set the sub-node:

<SKLtype>`linear_model|LassoLarsIC`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<criterion>**, ***'bic' — 'aic'*** , specifies the type of criterion to use.
  Default: 'aic'
- **<fit_intercept>**, ***boolean, optional field***, determines whether to calculate the intercept for this model. If set to False, no intercept will be used in the calculations (e.g. data is expected to be already centered).
  Default: True
- **<verbose>**, ***boolean or integer, optional field***, use verbose mode when fitting the model.
  Default: False
- **<normalize>**, ***boolean, optional field***, if True, the regressors X will be normalized before regression.
  Default: True
- **<precompute>**, ***boolean or string, optional field***, determines whether to use a precomputed Gram matrix to speed up calculations. If set to 'auto,' RAVEN will decide. The Gram matrix can also be passed as an argument. Available options are [True — False — 'auto' — array-like].
  Default: 'auto'
- **<max_iter>**, ***integer, optional field***, specifies the maximum number of iterations.
  Default: 500
- **<eps>**,***float, optional field***, represents the machine precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems. Unlike the tol parameter in some iterative optimization-based algorithms, this parameter does not control the tolerance of the optimization.
  Default: 2.2204460492503131e-16

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *LassoLarsIC* ROM.

15.3.7.1.12 Ordinary least squares Linear Regression

The *Ordinary least squares Linear Regression* is a method for estimating the unknown parameters in a linear regression model, with the goal of minimizing the differences between the observed responses in some arbitrary dataset and the responses predicted by the linear approximation of the data. In order to use the *Ordinary least squares Linear Regressor*, the user needs to set the sub-node:

<SKLtype>`linear_model|LinearRegression`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<fit_intercept>**, ***boolean, optional field***, determines whether to calculate the intercept for this model. If set to False, no intercept will be used in the calculations (e.g. data is expected to be already centered).
  Default: True
- **<normalize>**, ***boolean, optional field***, if True, the regressors X will be normalized before regression.
  Default: False

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *LinearRegression* ROM.

15.3.7.1.13 Logistic Regression

The *Logistic Regression* implements L1 and L2 regularized logistic regression using the liblinear library. It can handle both dense and sparse input. This regressor uses C-ordered arrays or CSR matrices containing 64-bit floats for optimal performance; any other input format will be converted (and copied). In order to use the *Logistic Regressor*, the user needs to set the sub-node:

<SKLtype>`linear_model|LogisticRegression`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<penalty>**, ***string, 'l1' or 'l2'***, specifies the norm used in the penalization.
  Default: 'l2'
- **<dual>**, ***boolean***, specifies the dual or primal formulation. Dual formulation is only implemented for the l2 penalty. Prefer dual=False when n_samples > n_features.
  Default: False
- **<C>**, ***float, optional field***, is the inverse of the regularization strength; must be a positive float. Like in support vector machines, smaller values specify stronger regularization.
  Default: 1.0
- **<fit_intercept>**, ***boolean***, specifies if a constant (a.k.a. bias or intercept) should be added to the decision function.
  Default: True

- **<intercept_scaling>**, ***float, optional field***, when self.fit_intercept is True, instance vector x becomes [x, self.intercept_scaling], i.e. a "synthetic" feature with constant value equal to intercept_scaling is appended to the instance vector. The intercept becomes intercept_scaling * synthetic feature weight. **Note:** The synthetic feature weight is subject to l1/l2 regularization as are all other features. To lessen the effect of regularization on synthetic feature weight (and therefore on the intercept) intercept_scaling has to be increased.
  Default: 1.0

- **<class_weight>**, ***dict, or 'balanced', optional*** Weights associated with classes in the form {class_label: weight}. If not given, all classes are supposed to have weight one. The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as n_samples / (n_classes * np.bincount(y)) Note that these weights will be multiplied with sample_weight (passed through the fit method) if sample_weight is specified. New in version 0.17: class_weight='balanced' instead of deprecated class_weight='auto'.
  Default: None

- **<random_state>**, ***int seed, RandomState instance, or None***, sets the seed of the pseudo random number generator to use when shuffling the data.
  Default: None

- **<tol>**, ***float, optional field***, specifies the tolerance for stopping criteria.
  Default: 0.0001

It is important to NOTE that RAVEN uses a Z-score normalization of the training data before constructing the *LogisticRegression* ROM:

$$\mathbf{X}' = \frac{(\mathbf{X} - \mu)}{\sigma} \tag{8}$$

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

The *Multi-task Lasso model trained with L1/L2 mixed-norm as regularizer* is a regressor where the optimization objective for Lasso is: $(1/(2*n_samples))*||Y-XW||^2_{Fro}+alpha*||W||_{21}$ Where: $||W||_{21} = \sum_i \sqrt{\sum_j w_{ij}^2}$ i.e. the sum of norm of each row. In order to use the *Multi-task Lasso model trained with L1/L2 mixed-norm as regularizer regressor*, the user needs to set the sub-node:

<SKLtype>`linear_model|MultiTaskLasso`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<alpha>**, ***float, optional field***, sets the constant multiplier for the L1/L2 term.
  Default: 1.0

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

- **<normalize>**, ***boolean, optional field***, if True, the regressors X will be normalized before regression.
  Default: False

- **<max_iter>**, ***integer, optional field***, specifies the maximum number of iterations.
  Default: 1000

- **<tol>**, ***float, optional field***, specifies the tolerance for the optimization: if the updates are smaller than tol, the optimization code checks the dual gap for optimality and continues until it is smaller than tol.
  Default: 1.e-4

- **<warm_start>**, ***boolean, optional field***, when set to True, the model will reuse the solution of the previous call to fit as initialization, otherwise, it will just erase the previous solution.
  Default: False

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *MultiTaskLasso* ROM.

15.3.7.1.15 Multi-task Elastic Net model trained with L1/L2 mixed-norm as regularizer

The *Multi-task Elastic Net model trained with L1/L2 mixed-norm as regularizer* is a regressor where the optimization objective for MultiTaskElasticNet is: $(1/(2 * n_samples)) * ||Y - XW||_2^{Fro} + alpha * l1_ratio * ||W||_{21} + 0.5 * alpha * (1 - l1_ratio) * ||W||_{Fro}^2$ Where: $||W||_{21} = \sum_i \sqrt{\sum_j w_{ij}^2}$ i.e. the sum of norm of each row. In order to use the *Multi-task ElasticNet model trained with L1/L2 mixed-norm as regularizer regressor*, the user needs to set the sub-node:

<SKLtype>`linear_model|MultiTaskElasticNet`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<alpha>**, ***float, optional field***, represents a constant multiplier for the L1/L2 term.
  Default: 1.0

- **<l1_ratio>**, ***float***, represents the Elastic Net mixing parameter, with $0 < l1_ratio \leq 1$. For $l1_ratio = 0$ the penalty is an L1/L2 penalty. For $l1_ratio = 1$ it is an L1 penalty. For $0 < l1_ratio < 1$, the penalty is a combination of L1/L2 and L2.
  Default: 0.5

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

- **<normalize>**, ***boolean, optional field***, if True, the regressors X will be normalized before regression.
  Default: False

- **<max_iter>**, ***integer, optional field***, specifies the maximum number of iterations.

- **<tol>**, ***float, optional field***, specifies the tolerance for the optimization: if the updates are smaller than tol, the optimization code checks the dual gap for optimality and continues until it is smaller than tol.
  Default: 1.e-4

- **<warm_start>**, ***boolean, optional field***, when set to True, the model will reuse the solution of the previous call to fit as initialization, otherwise, it will just erase the previous solution.
  Default: False

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *MultiTaskElasticNet* ROM.

15.3.7.1.16 Orthogonal Mathching Pursuit model (OMP)

The *Orthogonal Mathching Pursuit model (OMP)* is a type of sparse approximation which involves finding the "best matching" projections of multidimensional data onto an over-complete dictionary, D. In order to use the *Orthogonal Mathching Pursuit model (OMP) regressor*, the user needs to set the sub-node:

<SKLtype>`linear_model|OrthogonalMatchingPursuit`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<n_nonzero_coefs>**, ***int, optional field***, represents the desired number of non-zero entries in the solution. If None, this value is set to 10% of n_features.
  Default: None

- **<tol>**, ***float, optional field***, specifies the maximum norm of the residual. If not None, overrides n_nonzero_coefs.
  Default: None

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

- **<normalize>**, ***boolean, optional field***, if True, the regressors X will be normalized before regression.
  Default: True

- **<precompute>**, {***True, False, 'auto'***}, specifies whether to use a precomputed Gram and Xy matrix to speed up calculations. Improves performance when n_targets or n_samples is very large. **Note:** If you already have such matrices, you can pass them directly to the fit method.
  Default: 'auto'

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *OrthogonalMatchingPursuit* ROM.

15.3.7.1.17 Cross-validated Orthogonal Mathching Pursuit model (OMP)

The *Cross-validated Orthogonal Mathching Pursuit model (OMP)* is a regressor similar to OMP which has good performance in sparse recovery. In order to use the *Cross-validated Orthogonal Mathching Pursuit model (OMP) regressor*, the user needs to set the sub-node:

<SKLtype>`linear_model|OrthogonalMatchingPursuitCV`**</SKLtype>**.

In addition to this XML node, several others are available:

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

- **<normalize>**, ***boolean, optional field***, if True, the regressors X will be normalized before regression.
  Default: True

- **<max_iter>**, ***integer, optional field***, specifies the maximum number of iterations.
  Default: None Maximum number of iterations to perform, therefore maximum features to include 10% of n_features but at least 5 if available.

- **<cv>**, ***cross-validation generator, optional***, see sklearn.cross_validation.
  Default: None

- **<verbose>**, ***boolean or integer, optional field***, use verbose mode when fitting the model.
  Default: False

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *OrthogonalMatchingPursuitCV* ROM.

15.3.7.1.18 Passive Aggressive Classifier

The *Passive Aggressive Classifier* is a principled approach to linear classification that advocates minimal weight updates i.e., the least required to correctly classify the current training instance. In order to use the *Passive Aggressive Classifier*, the user needs to set the sub-node:

<SKLtype>`linear_model|PassiveAggressiveClassifier`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<C>**, ***float***, specifies the maximum step size (regularization).
  Default: 1.0

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

- **<shuffle>**, ***boolean, optional field***, specifies whether or not the training data should be shuffled after each epoch.
  Default: True

- **<n_iter_no_change>**, ***integer, optional field***, number of iterations with no improvement to wait before early stopping.
  Default: 5

- **<random_state>**, ***int seed, RandomState instance, or None***, sets the seed of the pseudo random number generator to use when shuffling the data.
  Default: None

- **<verbose>**, ***boolean or integer, optional field***, use verbose mode when fitting the model.
  Default: 0

- **<loss>**, ***string, optional field***, the loss function to be used:

- hinge: equivalent to PA-I (http://jmlr.csail.mit.edu/papers/volume7/crammer06a/crammer06a.pdf)
- squared_hinge: equivalent to PA-II (http://jmlr.csail.mit.edu/papers/volume7/crammer06a/crammer06a

Default: 'hinge'

- **<warm_start>**, ***boolean, optional field***, when set to True, the model will reuse the solution of the previous call to fit as initialization, otherwise, it will just erase the previous solution.
  Default: False

It is important to NOTE that RAVEN uses a Z-score normalization of the training data before constructing the *PassiveAggressiveClassifier* ROM:

$$\mathbf{X}' = \frac{(\mathbf{X} - \mu)}{\sigma} \tag{9}$$

15.3.7.1.19 Passive Aggressive Regressor

The *Passive Aggressive Regressor* is similar to the Perceptron in that it does not require a learning rate. However, contrary to the Perceptron, this regressor includes a regularization parameter, C.

In order to use the *Passive Aggressive Regressor*, the user needs to set the sub-node:

<SKLtype>`linear_model|PassiveAggressiveRegressor`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<C>**, ***float***, sets the maximum step size (regularization).
  Default: 1.0
- **<epsilon>**, ***float***, if the difference between the current prediction and the correct label is below this threshold, the model is not updated.
  Default: 0.1
- **<fit_intercept>**, ***boolean, optional field***, determines whether to calculate the intercept for this model. If set to False, no intercept will be used in the calculations (e.g. data is expected to be already centered).
  Default: True
- **<n_iter>**, ***int, optional field***, specifies the number of passes over the training data (aka epochs).
  Default: 5

- **<shuffle>**, ***boolean, optional field***, specifies whether or not the training data should be shuffled after each epoch.
  Default: True

- **<random_state>**, ***int seed, RandomState instance, or None***, sets the seed of the pseudo random number generator to use when shuffling the data.
  Default: None

- **<verbose>**, ***boolean or integer, optional field***, use verbose mode when fitting the model.
  Default: 0

- **<loss>**, ***string, optional field***, specifies the loss function to be used:

  – epsilon_insensitive: equivalent to PA-I in the reference paper (http://jmlr.csail.mit.edu/papers/volume7/crammer06a/crammer06a.pdf).

  – squared_epsilon_insensitive: equivalent to PA-II in the reference paper (http://jmlr.csail.mit.edu/papers/volume7/crammer06a/crammer06a.pdf).

  Default: 'epsilon_insensitive'

- **<warm_start>**, ***boolean, optional field***, when set to True, the model will reuse the solution of the previous call to fit as initialization, otherwise, it will just erase the previous solution.
  Default: False

It is important to NOTE that RAVEN uses a Z-score normalization of the training data before constructing the *PassiveAggressiveRegressor* ROM:

$$\mathbf{X}' = \frac{(\mathbf{X} - \mu)}{\sigma} \tag{10}$$

15.3.7.1.20 Perceptron

The *Perceptron* method is an algorithm for supervised classification of an input into one of several possible non-binary outputs. It is a type of linear classifier, i.e. a classification algorithm that makes its predictions based on a linear predictor function combining a set of weights with the feature vector. The algorithm allows for online learning, in that it processes elements in the training set one at a time. In order to use the *Perceptron classifier*, the user needs to set the sub-node:

<SKLtype>`linear_model|Perceptron`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<penalty>**, ***None, 'l2' or 'l1' or 'elasticnet'***, defines the penalty (aka regularization term) to be used.
  Default: None

- **<alpha>**, ***float***, sets the constant multiplier for the regularization term if regularization is used.
  Default: 0.0001

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

- **<n_iter_no_change>**, ***integer, optional field***, number of iterations with no improvement to wait before early stopping.
  Default: 5

- **<shuffle>**, ***boolean, optional field***, specifies whether or not the training data should be shuffled after each epoch.
  Default: True

- **<random_state>**, ***int seed, RandomState instance, or None***, sets the seed of the pseudo random number generator to use when shuffling the data.
  Default: 0

- **<verbose>**, ***boolean or integer, optional field***, use verbose mode when fitting the model.
  Default: 0

- **<eta0>**, ***double, optional field***, defines the constant multiplier for the updates.
  Default: 1.0

- **<class_weight>**, ***dict, {class_label: weight} or "balanced" or None, optional*** Preset for the class_weight fit parameter. Weights associated with classes. If not given, all classes are supposed to have weight one. The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as n_samples / (n_classes * np.bincount(y))

- **<warm_start>**, ***boolean, optional field***, when set to True, the model will reuse the solution of the previous call to fit as initialization, otherwise, it will just erase the previous solution.
  Default: False

It is important to NOTE that RAVEN uses a Z-score normalization of the training data before constructing the *PassiveAggressiveRegressor* ROM:

$$\mathbf{X}' = \frac{(\mathbf{X} - \mu)}{\sigma} \tag{11}$$

15.3.7.1.21 Linear least squares with l2 regularization

The *Linear least squares with l2 regularization* solves a regression model where the loss function is the linear least squares function and the regularization is given by the l2-norm. Also known as Ridge Regression or Tikhonov regularization. This estimator has built-in support for multivariate regression (i.e., when y is a 2d-array of shape [n_samples, n_targets]). In order to use the *Linear least squares with l2 regularization*, the user needs to set the sub-node:

<SKLtype>`linear_model|Ridge`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<alpha>**, ***float, array-like***, shape = [n_targets] Small positive values of alpha improve the conditioning of the problem and reduce the variance of the estimates. Alpha corresponds to $(2 * C)^{-}1$ in other linear models such as LogisticRegression or LinearSVC. If an array is passed, penalties are assumed to be specific to the targets. Hence they must correspond in number.
  Default: 1.0
- **<fit_intercept>**, ***boolean, optional field***, determines whether to calculate the intercept for this model. If set to False, no intercept will be used in the calculations (e.g. data is expected to be already centered).
  Default: True
- **<n_iter_no_change>**, ***integer, optional field***, number of iterations with no improvement to wait before early stopping.
  Default: 5
- **<max_iter>**, ***integer, optional field***, specifies the maximum number of iterations.
  Default: determined by scipy.sparse.linalg.
- **<normalize>**, ***boolean, optional field***, if True, the regressors X will be normalized before regression.
  Default: False
- **<solver>**, {***'auto', 'svd', 'cholesky', 'lsqr', 'sparse_cg'***}, specifies the solver to use in the computational routines:
  - 'auto' chooses the solver automatically based on the type of data.
  - 'svd' uses a singular value decomposition of X to compute the ridge coefficients. More stable for singular matrices than 'cholesky.'
  - 'cholesky' uses the standard scipy.linalg.solve function to obtain a closed-form solution.

- 'sparse_cg' uses the conjugate gradient solver as found in scipy.sparse.linalg.cg. As an iterative algorithm, this solver is more appropriate than 'cholesky' for large-scale data (possibility to set tol and max_iter).
- 'lsqr' uses the dedicated regularized least-squares routine scipy.sparse.linalg.lsqr. It is the fatest but may not be available in old scipy versions. It also uses an iterative procedure.

All three solvers support both dense and sparse data.
Default: 'auto'

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *Ridge* ROM.

15.3.7.1.22 Classifier using Ridge regression

The *Classifier using Ridge regression* is a classifier based on linear least squares with l2 regularization. In order to use the *Classifier using Ridge regression*, the user needs to set the sub-node:

<SKLtype>`linear_model|RidgeClassifier`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<alpha>**, ***float***, small positive values of alpha improve the conditioning of the problem and reduce the variance of the estimates. Alpha corresponds to $(2 * C)^{-}1$ in other linear models such as LogisticRegression or LinearSVC.
  Default: 1.0
- **<class_weight>**, ***dict, optional field***, specifies weights associated with classes in the form class_label: weight. If not given, all classes are assumed to have weight one.
  Default: None
- **<fit_intercept>**, ***boolean, optional field***, determines whether to calculate the intercept for this model. If set to False, no intercept will be used in the calculations (e.g. data is expected to be already centered).
  Default: True
- **<max_iter>**, ***integer, optional field***, specifies the maximum number of iterations.
  Default: determined by scipy.sparse.linalg.
- **<normalize>**, ***boolean, optional field***, if True, the regressors X will be normalized before regression.
  Default: False

- **<solver>**, {***'auto', 'svd', 'cholesky', 'lsqr', 'sparse_cg'***}, specifies the solver to use in the computational routines:
  - 'auto' chooses the solver automatically based on the type of data.
  - 'svd' uses a singular value decomposition of X to compute the ridge coefficients. More stable for singular matrices than 'cholesky.'
  - 'cholesky' uses the standard scipy.linalg.solve function to obtain a closed-form solution.
  - 'sparse_cg' uses the conjugate gradient solver as found in scipy.sparse.linalg.cg. As an iterative algorithm, this solver is more appropriate than 'cholesky' for large-scale data (possibility to set tol and max_iter).
  - 'lsqr' uses the dedicated regularized least-squares routine scipy.sparse.linalg.lsqr. It is the fatest but may not be available in old scipy versions. It also uses an iterative procedure.

  All three solvers support both dense and sparse data.
  Default: 'auto'

- **<tol>**, ***float***, defines the required precision of the solution.
  Default: 0.001

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *RidgeClassifier* ROM.

15.3.7.1.23 Ridge classifier with built-in cross-validation

The *Ridge classifier with built-in cross-validation* performs Generalized Cross-Validation, which is a form of efficient leave-one-out cross-validation. Currently, only the n_features $>$ n_samples case is handled efficiently. In order to use the *Ridge classifier with built-in cross-validation classifier*, the user needs to set the sub-node:

<SKLtype>`linear_model|RidgeClassifierCV`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<alphas>**, ***numpy array of shape [n_alphas]***, is an array of alpha values to try. Small positive values of alpha improve the conditioning of the problem and reduce the variance of the estimates. Alpha corresponds to $(2*C)^{-1}$ in other linear models such as LogisticRegression or LinearSVC.
  Default: (0.1, 1.0, 10.0)

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

- **<normalize>**, ***boolean, optional field***, if True, the regressors X will be normalized before regression.
  Default: False

- **<scoring>**, ***string, callable or None, optional***, is a string (see model evaluation documentation) or a scorer callable object / function with signature scorer(estimator, X, y).
  Default: None

- **<cv>**, ***cross-validation generator, optional***, If None, Generalized Cross-Validation (efficient leave-one-out) will be used.
  Default: None

- **<class_weight>**, ***dic, optional field***, specifies weights associated with classes in the form class_label:weight. If not given, all classes are supposed to have weight one.
  Default: None

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *RidgeClassifierCV* ROM.

15.3.7.1.24 Ridge regression with built-in cross-validation

The *Ridge regression with built-in cross-validation* performs Generalized Cross-Validation, which is a form of efficient leave-one-out cross-validation. In order to use the *Ridge regression with built-in cross-validation regressor*, the user needs to set the sub-node:

<SKLtype>`linear_model|RidgeCV`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<alphas>**, ***numpy array of shape [n_alphas]***, specifies an array of alpha values to try. Small positive values of alpha improve the conditioning of the problem and reduce the variance of the estimates. Alpha corresponds to $(2 * C)^{-1}$ in other linear models such as LogisticRegression or LinearSVC.
  Default: (0.1, 1.0, 10.0)

- **<fit_intercept>**, ***boolean, optional field***, determines whether to calculate the intercept for this model. If set to False, no intercept will be used in the calculations (e.g. data is

expected to be already centered).
Default: True

- **<normalize>**, ***boolean, optional field***, if True, the regressors X will be normalized before regression.
Default: False

- **<scoring>**, ***string, callable or None, optional***, is a string (see model evaluation documentation) or a scorer callable object / function with signature scorer(estimator, X, y).
Default: None

- **<cv>**, ***cross-validation generator, optional field***, if None, Generalized Cross-Validation (efficient leave-one-out) will be used.
Default: None

- **<gcv_mode>**, {***None, 'auto,' 'svd,' 'eigen'***}, ***optional field***, is a flag indicating which strategy to use when performing Generalized Cross-Validation. Options are:

  – 'auto:' use svd if n_samples ¿ n_features or when X is a sparse matrix, otherwise use eigen

  – 'svd:' force computation via singular value decomposition of X (does not work for sparse matrices)

  – 'eigen:' force computation via eigendecomposition of $X^T X$

  The 'auto' mode is the default and is intended to pick the cheaper option of the two depending upon the shape and format of the training data.
Default: None

- **<store_cv_values>**, ***boolean***, is a flag indicating if the cross-validation values corresponding to each alpha should be stored in the cv_values_attribute (see below). This flag is only compatible with cv=None (i.e. using Generalized Cross-Validation).
Default: False

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *RidgeCV* ROM.

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

The *Linear classifiers (SVM, logistic regression, a.o.) with SGD training* implements regularized linear models with stochastic gradient descent (SGD) learning: the gradient of the loss is estimated for each sample at a time and the model is updated along the way with a decreasing strength schedule (aka learning rate). SGD allows minibatch (online/out-of-core) learning, see the

partial_fit method. This implementation works with data represented as dense or sparse arrays of floating point values for the features. The model it fits can be controlled with the loss parameter; by default, it fits a linear support vector machine (SVM). The regularizer is a penalty added to the loss function that shrinks model parameters towards the zero vector using either the squared Euclidean norm L2 or the absolute norm L1 or a combination of both (Elastic Net). If the parameter update crosses the 0.0 value because of the regularizer, the update is truncated to 0.0 to allow for learning sparse models and achieves online feature selection. In order to use the *Linear classifiers (SVM, logistic regression, a.o.) with SGD training*, the user needs to set the sub-node:

<SKLtype>`linear_model|SGDClassifier`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<loss>**, ***str, 'hinge,' 'log,' 'modified_huber,' 'squared_hinge,' 'perceptron,' or a regression loss: 'squared_loss,' 'huber,' 'epsilon_insensitive,' or 'squared_epsilon_insensitive'***, dictates the loss function to be used. The available options are:
  - 'hinge' gives a linear SVM.
  - 'log' loss gives logistic regression, a probabilistic classifier.
  - 'modified_huber' is another smooth loss that brings tolerance to outliers as well as probability estimates.
  - 'squared_hinge' is like hinge but is quadratically penalized.
  - 'perceptron' is the linear loss used by the perceptron algorithm.

  The other losses are designed for regression but can be useful in classification as well; see SGDRegressor for a description.
  Default: 'hinge'

- **<penalty>**, ***str, 'l2' or 'l1' or 'elasticnet'***, defines the penalty (aka regularization term) to be used. 'l2' is the standard regularizer for linear SVM models. 'l1' and 'elasticnet' might bring sparsity to the model (feature selection) not achievable with 'l2.'
  Default: 'l2'

- **<alpha>**, ***float***, is the constant multiplier for the regularization term.
  Default: 0.0001

- **<l1_ratio>**, ***float***, is the Elastic Net mixing parameter, with 0 ¡= l1_ratio ¡= 1. l1_ratio=0 corresponds to L2 penalty, l1_ratio=1 to L1.
  Default: 0.15

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

- **<n_iter_no_change>**, ***integer, optional field***, number of iterations with no improvement to wait before early stopping.
  Default: 5

- **<shuffle>**, ***boolean, optional field***, specifies whether or not the training data should be shuffled after each epoch.
  Default: True

- **<random_state>**, ***int seed, RandomState instance, or None***, represents the seed of the pseudo random number generator to use when shuffling the data for probability estimation.
  Default: None

- **<verbose>**, ***boolean or integer, optional field***, use verbose mode when fitting the model.
  Default: 0

- **<epsilon>**, ***float, optional field***, varies meaning depending on the value of **<loss>**. If loss is 'huber', 'epsilon_insensitive' or 'squared_epsilon_insensitive' then this is the epsilon in the epsilon-insensitive loss functions. For 'huber', determines the threshold at which it becomes less important to get the prediction exactly right. For 'epsilon_insensitive, any differences between the current prediction and the correct label are ignored if they are less than this threshold.
  Default: 0.1

- **<learning_rate>**, ***string, optional field***, specifies the learning rate:

  - 'constant:' eta = eta0
  - 'optimal:' eta = 1.0 / (t + t0)
  - 'invscaling:' eta = eta0 / pow(t, power_t)

  Default: 'optimal'

- **<eta0>**, ***double***, specifies the initial learning rate for the 'constant' or 'invscaling' schedules. The default value is 0.0 as eta0 is not used by the default schedule 'optimal.'
  Default: 0.0

- **<power_t>**, ***double***, represents the exponent for the inverse scaling learning rate.
  Default: 0.5

- **<class_weight>**, ***dict, class_label***, is the preset for the class_weight fit parameter. Weights associated with classes. If not given, all classes are assumed to have weight one. The "auto" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies.
  Default: None

- **<warm_start>**, ***boolean, optional field***, when set to True, the model will reuse the solution of the previous call to fit as initialization, otherwise, it will just erase the previous solution.
  Default: False

It is important to NOTE that RAVEN uses a Z-score normalization of the training data before constructing the *SGDClassifier* ROM:

$$\mathbf{X}' = \frac{(\mathbf{X} - \mu)}{\sigma} \tag{12}$$

15.3.7.1.26 Linear model fitted by minimizing a regularized empirical loss with SGD

The *Linear model fitted by minimizing a regularized empirical loss with SGD* is a model where SGD stands for Stochastic Gradient Descent: the gradient of the loss is estimated each sample at a time and the model is updated along the way with a decreasing strength schedule (aka learning rate). The regularizer is a penalty added to the loss function that shrinks model parameters towards the zero vector using either the squared euclidean norm L2 or the absolute norm L1 or a combination of both (Elastic Net). If the parameter update crosses the 0.0 value because of the regularizer, the update is truncated to 0.0 to allow for learning sparse models and achieving online feature selection. This implementation works with data represented as dense numpy arrays of floating point values for the features. In order to use the *Linear model fitted by minimizing a regularized empirical loss with SGD*, the user needs to set the sub-node:

<SKLtype>`linear_model|SGDRegressor`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<loss>**, ***str, 'squared_loss,' 'huber,' 'epsilon_insensitive,' or 'squared_epsilon_insensitive'***, specifies the loss function to be used. Defaults to 'squared_loss' which refers to the ordinary least squares fit. 'huber' modifies 'squared_loss' to focus less on getting outliers correct by switching from squared to linear loss past a distance of epsilon. 'epsilon_insensitive' ignores errors less than epsilon and is linear past that; this is the loss function used in SVR. 'squared_epsilon_insensitive' is the same but becomes squared loss past a tolerance of epsilon.
  Default: 'squared_loss'

- **<penalty>**, ***str, 'l2' or 'l1' or 'elasticnet'***, sets the penalty (aka regularization term) to be used. Defaults to 'l2' which is the standard regularizer for linear SVM models. 'l1' and 'elasticnet' might bring sparsity to the model (feature selection) not achievable with 'l2'.
  Default: 'l2'

- **<alpha>**, ***float***, Constant that multiplies the regularization term. Defaults to 0.0001

- **<l1_ratio>**, ***float***, is the Elastic Net mixing parameter, with $0 \leq l1_ratio \leq 1$. l1_ratio=0 corresponds to L2 penalty, l1_ratio=1 to L1.
  Default: 0.15

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

- **<n_iter_no_change>**, ***integer, optional field***, number of iterations with no improvement to wait before early stopping.
  Default: 5

- **<shuffle>**, ***boolean, optional field***, specifies whether or not the training data should be shuffled after each epoch.
  Default: True

- **<random_state>**, ***int seed, RandomState instance, or None***, sets the seed of the pseudo random number generator to use when shuffling the data.
  Default: None

- **<verbose>**, ***boolean or integer, optional field***, use verbose mode when fitting the model.
  Default: 0

- **<epsilon>**, ***float***, sets the epsilon in the epsilon-insensitive loss functions; only if loss is 'huber,' 'epsilon_insensitive,' or 'squared_epsilon_insensitive.' For 'huber', determines the threshold at which it becomes less important to get the prediction exactly right. For epsilon-insensitive, any differences between the current prediction and the correct label are ignored if they are less than this threshold.
  Default: 0.1

- **<learning_rate>**, ***string, optional field***, Learning rate:

  - constant: eta = eta0
  - optimal: eta = 1.0/(t+t0)
  - invscaling: eta= eta0 / pow(t, power_t)

  Default: invscaling

- **<eta0>**, ***double***, specifies the initial learning rate.
  Default: 0.01

- **<power_t>**, ***double, optional field***, specifies the exponent for inverse scaling learning rate.

  Default: 0.25

- **<warm_start>**, ***boolean, optional field***, when set to True, the model will reuse the solution of the previous call to fit as initialization, otherwise, it will just erase the previous solution. *Default: False*

It is important to NOTE that RAVEN uses a Z-score normalization of the training data before constructing the *SGDRegressor* ROM:

$$\mathbf{X}' = \frac{(\mathbf{X} - \mu)}{\sigma} \tag{13}$$

15.3.7.2 Support Vector Machines

In machine learning, **Support Vector Machines** (SVMs, also support vector networks) are supervised learning models with associated learning algorithms that analyze data and recognize patterns, used for classification and regression analysis. Given a set of training examples, each marked as belonging to one of two categories, an SVM training algorithm builds a model that assigns new examples into one category or the other, making it a non-probabilistic binary linear classifier. An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall on. In addition to performing linear classification, SVMs can efficiently perform a non-linear classification using what is called the kernel trick, implicitly mapping their inputs into high-dimensional feature spaces.
It is important to NOTE that RAVEN uses a Z-score normalization of the training data before constructing the *SVM-based* ROM:

$$\mathbf{X}' = \frac{(\mathbf{X} - \mu)}{\sigma} \tag{14}$$

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

15.3.7.2.1 Linear Support Vector Classifier

The *Linear Support Vector Classifier* is similar to SVC with parameter kernel='linear', but implemented in terms of liblinear rather than libsvm, so it has more flexibility in the choice of penalties and loss functions and should scale better (to large numbers of samples). This class supports both dense and sparse input and the multiclass support is handled according to a one-vs-the-rest scheme. In order to use the *Linear Support Vector Classifier*, the user needs to set the sub-node:

<SKLtype>`svm|LinearSVC`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<C>**, ***float, optional field***, sets the penalty parameter C of the error term.
  Default: 1.0

- **<loss>**, ***string, 'hinge' or 'squared_hinge'***, specifies the loss function. 'hinge' is the hinge loss (standard SVM) while 'squared_hinge' is the squared hinge loss.
  Default: 'squared_hinge'

- **<penalty>**, ***string, 'l1' or 'l2'***, specifies the norm used in the penalization. The 'l2' penalty is the standard used in SVC. The 'l1' leads to coef_vectors that are sparse.
  Default: 'l2'

- **<dual>**, ***boolean***, selects the algorithm to either solve the dual or primal optimization problem. Prefer dual=False when n_samples > n_features.
  Default: True

- **<tol>**, ***float, optional field***, specifies the tolerance for stopping criteria.
  Default: 1e-4

- **<multi_class>**, ***string, 'ovr' or 'crammer_singer'***, Determines the multi-class strategy if y contains more than two classes. ovr trains n_classes one-vs-rest classifiers, while crammer_singer optimizes a joint objective over all classes. While crammer_singer is interesting from a theoretical perspective as it is consistent, it is seldom used in practice and rarely leads to better accuracy and is more expensive to compute. If crammer_singer is chosen, the options loss, penalty and dual will be ignored.
  Default: 'ovr'

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

- **<intercept_scaling>**, ***float, optional field***, when True, the instance vector x becomes [x,self.intercept_scaling], i.e. a "synthetic" feature with constant value equals to intercept_scaling is appended to the instance vector. The intercept becomes intercept_scaling * synthetic feature weight. **Note:** The synthetic feature weight is subject to l1/l2 regularization as are all other features. To lessen the effect of regularization on the synthetic feature weight (and therefore on the intercept) intercept_scaling has to be increased.
  Default: 1

- **<class_weight>**, ***dict, 'auto', optional***, sets the parameter C of class i to class_weight[i]*C for SVC. If not given, all classes are assumed to have weight one. The 'auto' mode uses the values of y to automatically adjust weights inversely proportional to class frequencies.
  Default: None

- **<verbose>**, ***boolean or integer, optional field***, use verbose mode when fitting the model. *Default: 0* **Note:** This setting takes advantage of a per-process runtime setting in liblinear that, if enabled, may not work properly in a multithreaded context.

- **<random_state>**, ***int seed, RandomState instance, or None***, represents the seed of the pseudo random number generator to use when shuffling the data for probability estimation. *Default: None*

15.3.7.2.2 C-Support Vector Classification

The *C-Support Vector Classification* is a based on libsvm. The fit time complexity is more than quadratic with the number of samples which makes it hard to scale to datasets with more than a couple of 10000 samples. The multiclass support is handled according to a one-vs-one scheme. In order to use the *C-Support Vector Classifier*, the user needs to set the sub-node:

<SKLtype>`svm|SVC`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<C>**, ***float, optional field***, sets the penalty parameter C of the error term. *Default: 1.0*

- **<kernel>**, ***string, optional***, specifies the kernel type to be used in the algorithm. It must be one of:

  - 'linear'
  - 'poly'
  - 'rbf'
  - 'sigmoid'
  - 'precomputed'
  - a callable object

  If a callable is given it is used to pre-compute the kernel matrix. *Default: 'rbf'*

- **<degree>**, ***int, optional field***, determines the degree of the polynomial kernel function ('poly'). Ignored by all other kernels. *Default: 3.0*

- **<gamma>**, ***float, optional field***, sets the kernel coefficient for the kernels 'rbf,' 'poly,' and 'sigmoid.' If gamma is 'auto' then 1/n_features will be used instead. *Default: 'auto'*

- **<coef0>**, ***float, optional field***, is an independent term in kernel function. It is only significant in 'poly' and 'sigmoid.'
  Default: 0.0

- **<probability>**, ***boolean, optional field***, determines whether or not to enable probability estimates. This must be enabled prior to calling fit, and will slow down that method.
  Default: False

- **<shrinking>**, ***boolean, optional field***, determines whether or not to use the shrinking heuristic.
  Default: True

- **<tol>**, ***float, optional field***, specifies the tolerance for stopping criteria.
  Default: 1e-3

- **<cache_size>**, ***float, optional field***, specifies the size of the kernel cache (in MB).

- **<class_weight>**, ***dict, 'auto', optional***, sets the parameter C of class i to class_weight[i]*C for SVC. If not given, all classes are assumed to have weight one. The 'auto' mode uses the values of y to automatically adjust weights inversely proportional to class frequencies.
  Default: None

- **<verbose>**, ***boolean, optional field***, use verbose mode when fitting the model.
  Default: False **Note:** This setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.

- **<max_iter>**, ***integer, optional field***, specifies the maximum number of iterations.
  Default: -1

- **<random_state>**, ***int seed, RandomState instance, or None***, represents the seed of the pseudo random number generator to use when shuffling the data for probability estimation.
  Default: None

15.3.7.2.3 Nu-Support Vector Classification

The *Nu-Support Vector Classification* is similar to SVC but uses a parameter to control the number of support vectors. The implementation is based on libsvm. In order to use the *Nu-Support Vector Classifier*, the user needs to set the sub-node:

<SKLtype>`svm|NuSVC`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<nu>**, ***float, optional field***, is an upper bound on the fraction of training errors and a lower bound of the fraction of support vectors. Should be in the interval (0, 1].
  Default: 0.5

- **`<kernel>`**, ***string, optional***, specifies the kernel type to be used in the algorithm. It must be one of:
  - 'linear'
  - 'poly'
  - 'rbf'
  - 'sigmoid'
  - 'precomputed'
  - a callable object

  If a callable is given it is used to pre-compute the kernel matrix.
  Default: 'rbf'

- **`<degree>`**, ***int, optional field***, determines the degree of the polynomial kernel function ('poly'). Ignored by all other kernels.
  Default: 3

- **`<gamma>`**, ***float, optional field***, sets the kernel coefficient for the kernels 'rbf,' 'poly,' and 'sigmoid.' If gamma is 'auto' then 1/n_features will be used instead.
  Default: 'auto'

- **`<coef0>`**, ***float, optional field***, is an independent term in kernel function. It is only significant in 'poly' and 'sigmoid.'
  Default: 0.0

- **`<probability>`**, ***boolean, optional field***, determines whether or not to enable probability estimates. This must be enabled prior to calling fit, and will slow down that method.
  Default: False

- **`<shrinking>`**, ***boolean, optional field***, determines whether or not to use the shrinking heuristic.
  Default: True

- **`<tol>`**, ***float, optional field***, specifies the tolerance for stopping criteria.
  Default: 1e-3

- **`<cache_size>`**, ***float, optional field***, specifies the size of the kernel cache (in MB).

- **`<verbose>`**, ***boolean, optional field***, use verbose mode when fitting the model.
  Default: False **Note:** This setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.

- **`<max_iter>`**, ***integer, optional field***, specifies the maximum number of iterations.
  Default: -1

- **<random_state>**, ***int seed, RandomState instance, or None***, represents the seed of the pseudo random number generator to use when shuffling the data for probability estimation.
  Default: None

15.3.7.2.4 Support Vector Regression

The *Support Vector Regression* is an epsilon-Support Vector Regression. The free parameters in this model are C and epsilon. The implementations is a based on libsvm. In order to use the *Support Vector Regressor*, the user needs to set the sub-node:

<SKLtype>`svm|SVR`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<C>**, ***float, optional field***, sets the penalty parameter C of the error term.
  Default: 1.0
- **<epsilon>**, ***float, optional field***, specifies the epsilon-tube within which no penalty is associated in the training loss function with points predicted within a distance epsilon from the actual value.
  Default: 0.1
- **<kernel>**, ***string, optional***, specifies the kernel type to be used in the algorithm. It must be one of:
  - 'linear'
  - 'poly'
  - 'rbf'
  - 'sigmoid'
  - 'precomputed'
  - a callable object

  If a callable is given it is used to pre-compute the kernel matrix.
  Default: 'rbf'
- **<degree>**, ***int, optional field***, determines the degree of the polynomial kernel function ('poly'). Ignored by all other kernels.
  Default: 3.0
- **<gamma>**, ***float, optional field***, sets the kernel coefficient for the kernels 'rbf,' 'poly,' and 'sigmoid.' If gamma is 'auto' then 1/n_features will be used instead.
  Default: 'auto'

- **<coef0>**, ***float, optional field***, is an independent term in kernel function. It is only significant in 'poly' and 'sigmoid.'
  Default: 0.0
- **<shrinking>**, ***boolean, optional field***, determines whether or not to use the shrinking heuristic.
  Default: True
- **<tol>**, ***float, optional field***, specifies the tolerance for stopping criteria.
  Default: 1e-3
- **<cache_size>**, ***float, optional field***, specifies the size of the kernel cache (in MB).
- **<verbose>**, ***boolean, optional field***, use verbose mode when fitting the model.
  Default: False **Note:** This setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.
- **<max_iter>**, ***integer, optional field***, specifies the maximum number of iterations.
  Default: -1

15.3.7.3 Multi Class

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

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *multi-class-based* ROM.

In the following, all the multi-class models available in RAVEN are reported.

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

The *One-vs-the-rest (OvR) multiclass/multilabel strategy*, also known as one-vs-all, consists in fitting one classifier per class. For each classifier, the class is fitted against all the other classes. In addition to its computational efficiency (only n_classes classifiers are needed), one advantage of this approach is its interpretability. Since each class is represented by one and one classifier only, it is possible to gain knowledge about the class by inspecting its corresponding classifier. This is the most commonly used strategy and is a fair default choice.

In order to use the *One-vs-the-rest (OvR) multiclass/multilabel classifier*, the user needs to set the sub-node:

<SKLtype>`multiClass|OneVsRestClassifier`**</SKLtype>**.

In addition to this XML node, several others are available:

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

15.3.7.3.2 One-vs-one multiclass strategy

The *One-vs-one multiclass strategy* consists in fitting one classifier per class pair. At prediction time, the class which received the most votes is selected. Since it requires to fit n_classes * (n_classes - 1) / 2 classifiers, this method is usually slower than one-vs-the-rest, due to its O(n_classes^2) complexity. However, this method may be advantageous for algorithms such as kernel algorithms which do not scale well with n_samples. This is because each individual learning problem only involves a small subset of the data whereas, with one-vs-the-rest, the complete dataset is used n_classes times.

In order to use the *One-vs-one multiclass classifier*, the user needs to set the sub-node:

<SKLtype>`multiClass|OneVsOneClassifier`**</SKLtype>**.

In addition to this XML node, several others are available:

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

15.3.7.3.3 Error-Correcting Output-Code multiclass strategy

The *Error-Correcting Output-Code multiclass strategy* consists in representing each class with a binary code (an array of 0s and 1s). At fitting time, one binary classifier per bit in the code book is fitted. At prediction time, the classifiers are used to project new points in the class space and the class closest to the points is chosen. The main advantage of these strategies is that the number of classifiers used can be controlled by the user, either for compressing the model ($0 < code_size < 1$) or for making the model more robust to errors ($code_size > 1$).

In order to use the *Error-Correcting Output-Code multiclass classifier*, the user needs to set the sub-node:

<SKLtype>`multiClass|OutputCodeClassifier`**</SKLtype>**.

In addition to this XML node, several others are available:

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

15.3.7.4 Naive Bayes

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

$$P(y \mid x_1, \ldots, x_n) = \frac{P(y)P(x_1, \ldots x_n \mid y)}{P(x_1, \ldots, x_n)} \tag{15}$$

Using the naive independence assumption that

$$P(x_i|y, x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) = P(x_i|y), \tag{16}$$

for all i, this relationship is simplified to

$$P(y \mid x_1, \ldots, x_n) = \frac{P(y) \prod_{i=1}^{n} P(x_i \mid y)}{P(x_1, \ldots, x_n)} \tag{17}$$

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

$$P(y \mid x_1, \ldots, x_n) \propto P(y) \prod_{i=1}^{n} P(x_i \mid y) \Downarrow \tag{18}$$

$$\hat{y} = \arg\max_{y} P(y) \prod_{i=1}^{n} P(x_i \mid y), \tag{19}$$

and we can use Maximum A Posteriori (MAP) estimation to estimate $P(y)$ and $P(x_i \mid y)$; the former is then the relative frequency of class y in the training set. The different naive Bayes classifiers differ mainly by the assumptions they make regarding the distribution of $P(x_i \mid y)$.

In spite of their apparently over-simplified assumptions, naive Bayes classifiers have worked quite well in many real-world situations, famously document classification and spam filtering. They require a small amount of training data to estimate the necessary parameters. (For theoretical reasons why naive Bayes works well, and on which types of data it does, see the references below.) Naive Bayes learners and classifiers can be extremely fast compared to more sophisticated methods. The decoupling of the class conditional feature distributions means that each distribution can be independently estimated as a one dimensional distribution. This in turn helps to alleviate problems stemming from the curse of dimensionality.

On the flip side, although naive Bayes is known as a decent classifier, it is known to be a bad estimator, so the probability outputs from predict_proba are not to be taken too seriously. In the following, all the Naive Bayes available in RAVEN are reported.

15.3.7.4.1 Gaussian Naive Bayes

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

$$P(x_i \mid y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right) \tag{20}$$

The parameters σ_y and μ_y are estimated using maximum likelihood.

In order to use the *Gaussian Naive Bayes strategy*, the user needs to set the sub-node:

<SKLtype>`naiveBayes|GaussianNB`**</SKLtype>**.

There are no additional sub-nodes available for this method.

It is important to NOTE that RAVEN uses a Z-score normalization of the training data before constructing the *GaussianNB* ROM:

$$\mathbf{X}' = \frac{(\mathbf{X} - \mu)}{\sigma} \tag{21}$$

15.3.7.4.2 Multinomial Naive Bayes

The *Multinomial Naive Bayes* implements the naive Bayes algorithm for multinomially distributed data, and is one of the two classic naive Bayes variants used in text classification (where the data is typically represented as word vector counts, although tf-idf vectors are also known to work well in

practice). The distribution is parametrized by vectors $\theta_y = (\theta_{y1}, \ldots, \theta_{yn})$ for each class y, where n is the number of features (in text classification, the size of the vocabulary) and θ_{yi} is the probability $P(x_i \mid y)$ of feature i appearing in a sample belonging to class y. The parameters θ_y are estimated by a smoothed version of maximum likelihood, i.e. relative frequency counting:

$$\hat{\theta}_{yi} = \frac{N_{yi} + \alpha}{N_y + \alpha n} \tag{22}$$

where $N_{yi} = \sum_{x \in T} x_i$ is the number of times feature i appears in a sample of class y in the training set T, and $N_y = \sum_{i=1}^{|T|} N_{yi}$ is the total count of all features for class y. The smoothing priors $\alpha \geq 0$ account for features not present in the learning samples and prevents zero probabilities in further computations. Setting $\alpha = 1$ is called Laplace smoothing, while $\alpha < 1$ is called Lidstone smoothing. In order to use the *Multinomial Naive Bayes strategy*, the user needs to set the sub-node:

<SKLtype>`naiveBayes|MultinomialNB`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<alpha>**, ***float, optional field***, specifies an additive (Laplace/Lidstone) smoothing parameter (0 for no smoothing).
  Default: 1.0
- **<fit_prior>**, ***boolean, required field***, determines whether to learn class prior probabilities or not. If false, a uniform prior will be used.
  Default: True
- **<class_prior>**, ***array-like float (n_classes), optional field***, specifies prior probabilities of the classes. If specified, the priors are not adjusted according to the data.
  Default: None

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *MultinomialNB* ROM.

15.3.7.4.3 Bernoulli Naive Bayes

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

$$P(x_i \mid y) = P(i \mid y)x_i + (1 - P(i \mid y))(1 - x_i) \tag{23}$$

which differs from multinomial NB's rule in that it explicitly penalizes the non-occurrence of a feature i that is an indicator for class y, where the multinomial variant would simply ignore a non-occurring feature. In the case of text classification, word occurrence vectors (rather than word count vectors) may be used to train and use this classifier. *Bernoulli Naive Bayes* might perform better on some datasets, especially those with shorter documents. It is advisable to evaluate both models, if time permits. In order to use the *Bernoulli Naive Bayes strategy*, the user needs to set the sub-node:

<SKLtype>`naiveBayes|BernoulliNB`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<alpha>**, ***float, optional field***, specifies an additive (Laplace/Lidstone) smoothing parameter (0 for no smoothing).
  Default: 1.0
- **<binarize>**, ***float, optional field***, Threshold for binarizing (mapping to booleans) of sample features. If None, input is presumed to already consist of binary vectors.
  Default: 0.0
- **<fit_prior>**, ***boolean, required field***, determines whether to learn class prior probabilities or not. If false, a uniform prior will be used.
  Default: True
- **<class_prior>**, ***array-like float (n_classes), optional field***, specifies prior probabilities of the classes. If specified, the priors are not adjusted according to the data.
  Default: None

It is important to NOTE that RAVEN uses a Z-score normalization of the training data before constructing the *BernoulliNB* ROM:

$$\mathbf{X}' = \frac{(\mathbf{X} - \mu)}{\sigma} \tag{24}$$

15.3.7.5 Neighbors

The *Neighbors* class provides functionality for unsupervised and supervised neighbor-based learning methods. The unsupervised nearest neighbors method is the foundation of many other learning methods, notably manifold learning and spectral clustering. Supervised neighbors-based learning comes in two flavors: classification for data with discrete labels, and regression for data with continuous labels.

The principle behind nearest neighbor methods is to find a predefined number of training samples closest in distance to the new point, and predict the label from these. The number of samples can be a user-defined constant (k-nearest neighbor learning), or vary based on the local density of points (radius-based neighbor learning). The distance can, in general, be any metric measure: standard Euclidean distance is the most common choice. Neighbor-based methods are known as non-generalizing machine learning methods, since they simply "remember" all of its training data (possibly transformed into a fast indexing structure such as a Ball Tree or KD Tree.).

It is important to NOTE that RAVEN uses a Z-score normalization of the training data before constructing the *Neighbors-based* ROM:

$$\mathbf{X}' = \frac{(\mathbf{X} - \mu)}{\sigma} \tag{25}$$

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

15.3.7.5.1 K Neighbors Classifier

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

In order to use the *K Neighbors Classifier*, the user needs to set the sub-node:

<SKLtype>`neighbors|KNeighborsClassifier`**</SKLtype>**.

In addition to this XML node, several others are available:

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

Default: uniform

- **<algorithm>**, ***string, optional field***, specifies the algorithm used to compute the nearest neighbors:
  - *ball_tree* will use BallTree.
  - *kd_tree* will use KDtree.
  - *brute* will use a brute-force search.
  - *auto* will attempt to decide the most appropriate algorithm based on the values passed to fit method.

  Note: Fitting on sparse input will override the setting of this parameter, using brute force.
  Default: auto

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

- **<metric>**, ***string, optional field***, sets the distance metric to use for the tree. The Minkowski metric with p=2 is equivalent to the standard Euclidean metric.
  Default: minkowski

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

15.3.7.5.2 Radius Neighbors Classifier

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

In order to use the *Radius Neighbors Classifier*, the user needs to set the sub-node:

<SKLtype>`neighbors|RadiusNeighbors`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<radius>**, ***float, optional field***, specifies the range of parameter space to use by default for 'radius_neighbors' queries.
  Default: 1.0

- **<weights>**, ***string, optional field***, specifies the weight function used in prediction. Possible values:

  - *uniform* : uniform weights. All points in each neighborhood are weighted equally;
  - *distance* : weight points by the inverse of their distance. In this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.

  Default: uniform

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

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

  Note: Fitting on sparse input will override the setting of this parameter, using brute force.
  Default: auto

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

- **<metric>**, ***string, optional field***, sets the distance metric to use for the tree. The Minkowski metric with p=2 is equivalent to the standard Euclidean metric.
  Default: minkowski

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

- **<outlier_label>**, ***integer, optional field***, is a label, which is given for outlier samples (samples with no neighbors on a given radius). If set to None, ValueError is raised, when an outlier is detected.
  Default: None

15.3.7.5.3 K Neighbors Regressor

The *K Neighbors Regressor* can be used in cases where the data labels are continuous rather than discrete variables. The label assigned to a query point is computed based on the mean of the labels of its nearest neighbors. It implements learning based on the k nearest neighbors of each query point, where k is an integer value specified by the user.

In order to use the *K Neighbors Regressor*, the user needs to set the sub-node:

<SKLtype>`neighbors|KNeighborsRegressor`**</SKLtype>**.

In addition to this XML node, several others are available:

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

  Default: uniform
- **<algorithm>**, ***string, optional field***, specifies the algorithm used to compute the nearest neighbors:
  - *ball_tree* will use BallTree.
  - *kd_tree* will use KDtree.
  - *brute* will use a brute-force search.
  - *auto* will attempt to decide the most appropriate algorithm based on the values passed to fit method.

  Note: Fitting on sparse input will override the setting of this parameter, using brute force.
  Default: auto
- **<leaf_size>**, ***integer, optional field***, sets the leaf size passed to the BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.
  Default: 30

- **<metric>**, ***string, optional field***, sets the distance metric to use for the tree. The Minkowski metric with p=2 is equivalent to the standard Euclidean metric.
  Default: minkowski

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

15.3.7.5.4 Radius Neighbors Regressor

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

In order to use the *Radius Neighbors Regressor*, the user needs to set the sub-node:

<SKLtype>`neighbors|RadiusNeighborsRegressor`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<radius>**, ***float, optional field***, specifies the range of parameter space to use by default for 'radius_neighbors' queries.
  Default: 1.0

- **<weights>**, ***string, optional field***, specifies the weight function used in prediction. Possible values:
  - *uniform* : uniform weights. All points in each neighborhood are weighted equally;
  - *distance* : weight points by the inverse of their distance. In this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.

  Default: uniform

- **<algorithm>**, ***string, optional field***, specifies the algorithm used to compute the nearest neighbors:
  - *ball_tree* will use BallTree.
  - *kd_tree* will use KDtree.
  - *brute* will use a brute-force search.
  - *auto* will attempt to decide the most appropriate algorithm based on the values passed to fit method.

Note: Fitting on sparse input will override the setting of this parameter, using brute force.
Default: auto

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

- **<metric>**, ***string, optional field***, sets the distance metric to use for the tree. The Minkowski metric with p=2 is equivalent to the standard Euclidean metric.
  Default: minkowski

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

15.3.7.5.5 Nearest Centroid Classifier

The *Nearest Centroid classifier* is a simple algorithm that represents each class by the centroid of its members. It also has no parameters to choose, making it a good baseline classifier. It does, however, suffer on non-convex classes, as well as when classes have drastically different variances, as equal variance in all dimensions is assumed.

In order to use the *Nearest Centroid Classifier*, the user needs to set the sub-node:

<SKLtype>`neighbors|NearestCentroid`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<shrink_threshold>**, ***float, optional field***, defines the threshold for shrinking centroids to remove features.
  Default: None

The *Quadratic Discriminant Analysis* is a classifier with a quadratic decision boundary, generated by fitting class conditional densities to the data and using Bayes' rule. The model fits a Gaussian density to each class.

In order to use the *Quadratic Discriminant Analysis Classifier*, the user needs to set the sub-node:

<SKLtype>`qda|QDA`**</SKLtype>**.

In addition to this XML node, several others are available:

- **`<priors>`**, ***array-like (n_classes), optional field***, specifies the priors on the classes.
  Default: None
- **`<reg_param>`**, ***float, optional field***, regularizes the covariance estimate as (1-reg_param)*Sigma + reg_param*Identity(n_features).
  Default: 0.0

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *QDA* ROM.

15.3.7.6 Tree

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

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

The disadvantages of decision trees include:

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

It is important to NOTE that RAVEN uses a Z-score normalization of the training data before constructing the *tree-based* ROM:

$$\mathbf{X}' = \frac{(\mathbf{X} - \mu)}{\sigma} \tag{26}$$

In the following, all the tree-based algorithms available in RAVEN are reported.

15.3.7.6.1 Decision Tree Classifier

The *Decision Tree Classifier* is a classifier that is based on the decision tree logic.

In order to use the *Decision Tree Classifier*, the user needs to set the sub-node:

<SKLtype>`tree|DecisionTreeClassifier`**</SKLtype>**.

In addition to this XML node, several others are available:

- **`<criterion>`**, ***string, optional field***, specifies the function used to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain.
  Default: gini

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

- **<max_features>**, ***int, float or string, optional field***, sets the number of features to consider when looking for the best split:

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

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

- **<max_depth>**, ***integer, optional field***, determines the maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples. Ignored if max_samples_leaf is not None.
  Default: None

- **<min_samples_split>**, ***integer, optional field***, sets the minimum number of samples required to split an internal node.
  Default: 2

- **<min_samples_leaf>**, ***integer, optional field***, sets the minimum number of samples required to be at a leaf node.
  Default: 1

- **<max_leaf_nodes>**, ***integer, optional field***, grow a tree with max_leaf_nodes in best-first fashion. Best nodes are defined by relative reduction in impurity. If None then unlimited number of leaf nodes. If not None then max_depth will be ignored.
  Default: None

15.3.7.6.2 Decision Tree Regressor

The *Decision Tree Regressor* is a Regressor that is based on the decision tree logic. In order to use the *Decision Tree Regressor*, the user needs to set the sub-node:

<SKLtype>`tree|DecisionTreeRegressor`**</SKLtype>**.

In addition to this XML node, several others are available:

- **`<criterion>`**, ***string, optional field***, specifies the function used to measure the quality of a split. The only supported criterion is "mse" for mean squared error.
  Default: mse

- **`<splitter>`**, ***string, optional field***, specifies the strategy used to choose the split at each node. Supported strategies are "best" to choose the best split and "random" to choose the best random split.
  Default: best

- **`<max_features>`**, ***int, float or string, optional field***, sets the number of features to consider when looking for the best split:

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

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

- **`<max_depth>`**, ***integer, optional field***, determines the maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples. Ignored if max_samples_leaf is not None.
  Default: None

- **`<min_samples_split>`**, ***integer, optional field***, sets the minimum number of samples required to split an internal node.
  Default: 2

- **`<min_samples_leaf>`**, ***integer, optional field***, sets the minimum number of samples required to be at a leaf node.
  Default: 1

- **`<max_leaf_nodes>`**, ***integer, optional field***, grow a tree with max_leaf_nodes in best-first fashion. Best nodes are defined by relative reduction in impurity. If None then unlimited number of leaf nodes. If not None then max_depth will be ignored.
  Default: None

15.3.7.6.3 Extra Tree Classifier

The *Extra Tree Classifier* is an extremely randomized tree classifier. Extra-trees differ from classic decision trees in the way they are built. When looking for the best split to separate the samples of a node into two groups, random splits are drawn for each of the max_features randomly selected features and the best split among those is chosen. When max_features is set 1, this amounts to building a totally random decision tree.

In order to use the *Extra Tree Classifier*, the user needs to set the sub-node:

<SKLtype>`tree|ExtraTreeClassifier`**</SKLtype>**.

In addition to this XML node, several others are available:

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

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

- **<max_features>**, ***int, float or string, optional field***, sets the number of features to consider when looking for the best split:

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

  Note: The search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features.
  Default: auto

- **<max_depth>**, ***integer, optional field***, determines the maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples. Ignored if max_samples_leaf is not None.
  Default: None

- **<min_samples_split>**, ***integer, optional field***, sets the minimum number of samples required to split an internal node.
  Default: 2

- **<min_samples_leaf>**, ***integer, optional field***, sets the minimum number of samples required to be at a leaf node.
  Default: 1

- **<max_leaf_nodes>**, ***integer, optional field***, grow a tree with max_leaf_nodes in best-first fashion. Best nodes are defined by relative reduction in impurity. If None then unlimited number of leaf nodes. If not None then max_depth will be ignored.
  Default: None

15.3.7.6.4 Extra Tree Regressor

The *Extra Tree Regressor* is an extremely randomized tree regressor. Extra-trees differ from classic decision trees in the way they are built. When looking for the best split to separate the samples of a node into two groups, random splits are drawn for each of the max_features randomly selected features and the best split among those is chosen. When max_features is set 1, this amounts to building a totally random decision tree.

In order to use the *Extra Tree Regressor*, the user needs to set the sub-node:

<SKLtype>`tree|ExtraTreeRegressor`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<criterion>**, ***string, optional field***, specifies the function used to measure the quality of a split. The only supported criterion is "mse" for mean squared error.
  Default: mse

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

- **<max_features>**, ***int, float or string, optional field***, sets the number of features to consider when looking for the best split:

  – If int, then consider max_features features at each split.

  – If float, then max_features is a percentage and int(max_features * n_features) features are considered at each split.

  – If "auto," then max_features=sqrt(n_features).

- If "sqrt," then max_features=sqrt(n_features).
- If "log2," then max_features=log2(n_features).
- If None, then max_features=n_features.

Note: The search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features.
Default: auto

- **`<max_depth>`**, ***integer, optional field***, determines the maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples. Ignored if max_samples_leaf is not None.
Default: None

- **`<min_samples_split>`**, ***integer, optional field***, sets the minimum number of samples required to split an internal node.
Default: 2

- **`<min_samples_leaf>`**, ***integer, optional field***, sets the minimum number of samples required to be at a leaf node.
Default: 1

- **`<max_leaf_nodes>`**, ***integer, optional field***, grow a tree with max_leaf_nodes in best-first fashion. Best nodes are defined by relative reduction in impurity. If None then unlimited number of leaf nodes. If not None then max_depth will be ignored.
Default: None

15.3.7.7 Gaussian Process

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

- The prediction interpolates the observations (at least for regular correlation models).
- The prediction is probabilistic (Gaussian) so that one can compute empirical confidence intervals and exceedance probabilities that might be used to refit (online fitting, adaptive fitting) the prediction in some region of interest.
- Versatile: different linear regression models and correlation models can be specified. Common models are provided, but it is also possible to specify custom models provided they are stationary.

The disadvantages of Gaussian Processes for Machine Learning include:

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

In order to use the *Gaussian Process Regressor*, the user needs to set the sub-node:

<SKLtype>`GaussianProcess|GaussianProcess`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<normalize_y>**, ***boolean, optional field***, if True, the observations y are centered and reduced w.r.t. means and standard deviations estimated from the n_samples observations provided.
  Default: True
- **<optimizer>**, ***string, optional field***, specifies the optimization algorithm to be used. Available optimizers are: 'fmin_cobyla', 'Welch'.
  Default: fmin_cobyla
- **<random_state>**, ***integer, optional field***, is the seed of the internal random number generator.
  Default: None
- **<n_restarts_optimizer >**, ***integer, optional field***, The number of restarts of the optimizer for finding the kernel's parameters which maximize the log-marginal likelihood. The first run of the optimizer is performed from th e kernel's initial parameters, the remaining ones (if any) from thetas sampled log-uniform randomly from the space of allowed theta-values. If greater than 0, all bounds must be finite. Note that n_restarts_optimizer $== 0$ implies that one run is performed.
  Default: 0
- **<alpha>**, ***float, optional field***, Value added to the diagonal of the kernel matrix during fitting. Larger values correspond to increased noise level in the observations. This can also prevent a potential numerical issue during fitting, by ensuring that the calculated values form a positive definite matrix. If an array is passed, it must have the same number of entries as the data used for fitting and is used as datapoint-dependent noise level. Note that this is equivalent to adding a WhiteKernel with c=alpha. Allowing to specify the noise level directly as a parameter is mainly for convenience and for consistency with Ridge.
  Default: 1e-10

It is important to NOTE that RAVEN does not pre-normalize the training data before constructing the *GaussianProcessRegressor* ROM.

Example:

```
<Simulation>
  ...
  <Models>
    ...
   <ROM name='aUserDefinedName' subType='SciKitLearn'>
     <Features>var1,var2,var3</Features>
     <Target>result1</Target>
     <SKLtype>linear_model|LinearRegression</SKLtype>
     <fit_intercept>True</fit_intercept>
     <normalize>False</normalize>
   </ROM>
    ...
  </Models>
  ...
</Simulation>
```

15.3.7.8 Neural Network Models

It has been more than 70 years since Warren McCulloch and Water Pitts modeled the first artificial neural network (ANN) that mimicked the way brains work. These days, deep learning based on ANN is showing outstanding results for solving a wide variety of robotic tasks in the areas of perception, planning, localization, and control. **Multi-layer Perceptron (MLP)** is a supervised learning algorithm that can learn a non-linear function approximator for either classifcation or regression. It is different from logistic regression, in that between the input and output layer, there can be one or more non-linear layers, called hidden layers. The advantages of Multi-layer Perceptron are:

- Capability to learn non-linear models
- Capability to learn models in real-time (online learning)

The disadvantages of Multi-layer Perceptron include:

- MLP with hidden layers have a non-convex loss function where there exists more than one local minimum. Therefore different random weight initializations can lead to different validation accuracy.

- MLP requires tuning a number of hyperparameters such as the number of hidden neurons, layers and iterations.
- MLP is sensitive to feature scaling

It is important to NOTE that RAVEN uses a Z-score normalization of the training data before constructing the *Multi-layer Perceptron* ROM:

$$\mathbf{X}' = \frac{(\mathbf{X} - \mu)}{\sigma} \tag{27}$$

In the following, Multi-layer perceptron classification and regression algorithms available in RAVEN are reported.

15.3.7.8.1 MLPClassifier

The *MLPClassifier* implements a multi-layer perceptron algorithm that trains using **Backpropagation** More precisely, it trains using some form of gradient descent and the gradients are calculated using Backpropagation. For classification, it minimizes the Cross-Entropy loss function, and it supports multi-class classification.

In order to use the *MLPClassifier*, the user needs to set the sub-node:

`<SKLtype>``neural_network|MLPClassifier`**`</SKLtype>`**.

In addition to this XML node, several others are available:

- **`<hidden_layer_sizes>`**, ***tuple, length = n_layers - 2, optional field***, the ith element represents the number of neurons in the ith hidden layer.
  Default: (100,)
- **`<activation>`**, ***string, optional field***, activation function for the hidden layer.
  - *identity*, no-op activation, useful to implement linear bottleneck, returns f(x) = x
  - *logistic*, the logistic sigmoid function, returns f(x) = 1/(1+exp(-x))
  - *tanh*, the hyperbolic tan function, returns f(x) = tanh(x)
  - *relu*, the rectified linear function, returns f(x) = max(0, x)

  Default: 'relu'
- **`<solver>`**, ***string, optional field***, The solver for weight optimization.

- *lbfgs*, is an optimizer in the family of quasi-Newton methods
- *sgd*, refers to stochastic gradient descent
- *adam*, refers to a stochastic gradient-based optimizer proposed by Kingma, Diederik, and Jimmy Ba.

Note: The default solver *adam* works pretty well on relatively large datasets (with thousands of training samples or more) in terms of both training time and validation score. For small datasets, however, *lbfgs* can converge faster and perform better.
Default: 'adam'

- **<alpha>**, ***float, optional field***, L2 penalty parameter
Default: 0.0001

- **<batch_size>**, ***int, optional field***, Size of minibatches for stochastic optimizers. If the solver is *lbfgs*, the classifier will not use minibatch.
Default: min(200, n_samples)
Default: 'auto'

- **<learning_rate>**, ***string, optional field***, Learning rate schedule for weight updates

  - *constant*, is a constant learning rate given by *learning_rate_init*
  - *invscaling*, gradually decreases the learning rate at each time step 't' using an inverse scaling exponent of 'power_t'. effective_learning_rate = learning_rate_init / pow(t, power_t)
  - *adaptive*, keeps the learning rate constant to 'learning_rate_init' as long as training loss keeps decreasing. Each time two consecutive epochs fail to decrease training loss by at least tol, or fail to increase validation score by at least tol if 'early_stopping' is on, the current learning rate is divided by 5. **Note:** Only used when *solver = 'sgd'*

Default: 'constant'

- **<learning_rate_init>**, ***float, optional field***, The initial learning rate used. It controls the step-size in updating the weights. Only used when solver = 'sgd' or 'adam'
Default: 0.001

- **<power_t>**, ***float, optional field***, the exponent for inverse scaling learning rate. It is used in updating effective learning rate when the learning_rate is set to 'invscaling'. only used when solver = 'sgd'
Default: 0.5

- **<max_iter>**, ***int, optional field***, maximum number of iterations. The solver iterates until convergence (determined by 'tol') or this number of iterations. For stochastic solvers ('sgd', 'adam'), note that this determines the number of epochs (how many times each data point

will be used), not the number of gradient steps
Default: 200

- **<shuffle>**, ***bool, optional field***, whether to shuffle samples in each iteration. Only used when solver = 'sgd' or 'adam'
Default: True

- **<random_state>**, ***int, RandomState instance or None, optional field*** if int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random.
Default: None

- **<tol>**, ***float, optional field***, tolerance for optimization. When the loss or score is not improving by at least tol for two consecutive iterations, unless learning_rate is set to 'adaptive', convergence is considered to be reached and training stops.
Default: 1e-4

- **<verbose>**, ***bool, optional field***, whether to print progress messages or stdout
Default: False

- **<warm_start>**, ***bool, optional field***, when set to True, reuse the solution of previous call to fit as initialization, otherise, just erase the previous solution.
Default: False

- **<momentum>**, ***float, optional field***, momentum for gradient descent update. Should be between 0 and 1. Only used when solver = 'sgd'.
Default: 0.9

- **<nesterovs_momentum>**, ***bool, optional field***, whether to use Nesterov's momentum. Only used when solver='sgd' and momentum ¿ 0.
Default: True

- **<early_stopping>**, ***bool, optional field***, whether to use early stopping to terminate training when validation score is not improving. If set to true, it will automatically set aside 10% of training data as validation and terminate training when validation score is not improving by at least tol for two consecutive epochs. Only effective when solver='sgd' or 'adam'.
Default: False

- **<validation_fraction>**, ***float, optional field***, the proportion of training data to set aside as validation set for early stopping. Must be between 0 and 1. Only used if early_stopping is True.
Default: 0.1

- **<beta_1>**, ***float, optional field***, exponential decay rate for estimates of first moment vector in adam. should be in [0, 1). only used when solver = 'adam'
Default: 0.9

- **<beta_2>**, ***float, optional field***, exponentail decay rate for estimates of second moment vector in adam. should be in [0, 1). only used when solver = 'adam'
  Default: 0.999

- **<epsilon>**, ***float, optional field***, value for numerical stability in adam. Only used when solver = 'adam'
  Default: 1e-8

15.3.7.8.2 MLPRegressor

The *MLPRegressor* implements a multi-layer perceptron algorithm that trains using **Backpropagation** with no activation function in the output layer, which can also be seen as using the identity function as activation function. Therefore, it uses the square error as the loss function, and the output is a set of continuous values. *MLPRegressor* also supports multi-output regression, in which a sample can have more than one target.

In order to use the *MLPRegressor*, the user needs to set the sub-node:

<SKLtype>`neural_network|MLPRegressor`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<hidden_layer_sizes>**, ***tuple, length = n_layers - 2, optional field***, the ith element represents the number of neurons in the ith hidden layer.
  Default: (100,)

- **<activation>**, ***string, optional field***, activation function for the hidden layer.
  - *identity*, no-op activation, useful to implement linear bottleneck, returns f(x) = x
  - *logistic*, the logistic sigmoid function, returns f(x) = 1/(1+exp(-x))
  - *tanh*, the hyperbolic tan function, returns f(x) = tanh(x)
  - *relu*, the rectified linear function, returns f(x) = max(0, x)

  Default: 'relu'

- **<solver>**, ***string, optional field***, The solver for weight optimization.
  - *lbfgs*, is an optimizer in the family of quasi-Newton methods
  - *sgd*, refers to stochastic gradient descent
  - *adam*, refers to a stochastic gradient-based optimizer proposed by Kingma, Diederik, and Jimmy Ba.

Note: The default solver *adam* works pretty well on relatively large datasets (with thousands of training samples or more) in terms of both training time and validation score. For small datasets, however, *lbfgs* can converge faster and perform better.
Default: 'adam'

- **<alpha>**, ***float, optional field***, L2 penalty parameter
  Default: 0.0001

- **<batch_size>**, ***int, optional field***, Size of minibatches for stochastic optimizers. If the solver is *lbfgs*, the classifier will not use minibatch.
  Default: min(200, n_samples)
  Default: 'auto'

- **<learning_rate>**, ***string, optional field***, Learning rate schedule for weight updates

  – *constant*, is a constant learning rate given by *learning_rate_init*

  – *invscaling*, gradually decreases the learning rate at each time step 't' using an inverse scaling exponent of 'power_t'. effective_learning_rate = learning_rate_init / pow(t, power_t)

  – *adaptive*, keeps the learning rate constant to 'learning_rate_init' as long as training loss keeps decreasing. Each time two consecutive epochs fail to decrease training loss by at least tol, or fail to increase validation score by at least tol if 'early_stopping' is on, the current learning rate is divided by 5. **Note:** Only used when *solver = 'sgd'*

  Default: 'constant'

- **<learning_rate_init>**, ***float, optional field***, The initial learning rate used. It controls the step-size in updating the weights. Only used when solver = 'sgd' or 'adam'
  Default: 0.001

- **<power_t>**, ***float, optional field***, the exponent for inverse scaling learning rate. It is used in updating effective learning rate when the learning_rate is set to 'invscaling'. only used when solver = 'sgd'
  Default: 0.5

- **<max_iter>**, ***int, optional field***, maximum number of iterations. The solver iterates until convergence (determined by 'tol') or this number of iterations. For stochastic solvers ('sgd', 'adam'), note that this determines the number of epochs (how many times each data point will be used), not the number of gradient steps
  Default: 200

- **<shuffle>**, ***bool, optional field***, whether to shuffle samples in each iteration. Only used when solver = 'sgd' or 'adam'
  Default: True

- **<random_state>**, ***int, RandomState instance or None, optional field*** if int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random.
  Default: None

- **<tol>**, ***float, optional field***, tolerance for optimization. When the loss or score is not improving by at least tol for two consecutive iterations, unless learning_rate is set to 'adaptive', convergence is considered to be reached and training stops.
  Default: 1e-4

- **<verbose>**, ***bool, optional field***, whether to print progress messages or stdout
  Default: False

- **<warm_start>**, ***bool, optional field***, when set to True, reuse the solution of previous call to fit as initialization, otherise, just erase the previous solution.
  Default: False

- **<momentum>**, ***float, optional field***, momentum for gradient descent update. Should be between 0 and 1. Only used when solver = 'sgd'.
  Default: 0.9

- **<nesterovs_momentum>**, ***bool, optional field***, whether to use Nesterov's momentum. Only used when solver='sgd' and momentum ¿ 0.
  Default: True

- **<early_stopping>**, ***bool, optional field***, whether to use early stopping to terminate training when validation score is not improving. If set to true, it will automatically set aside 10% of training data as validation and terminate training when validation score is not improving by at least tol for two consecutive epochs. Only effective when solver='sgd' or 'adam'.
  Default: False

- **<validation_fraction>**, ***float, optional field***, the proportion of training data to set aside as validation set for early stopping. Must be between 0 and 1. Only used if early_stopping is True.
  Default: 0.1

- **<beta_1>**, ***float, optional field***, exponential decay rate for estimates of first moment vector in adam. should be in [0, 1). only used when solver = 'adam'
  Default: 0.9

- **<beta_2>**, ***float, optional field***, exponentail decay rate for estimates of second moment vector in adam. should be in [0, 1). only used when solver = 'adam'
  Default: 0.999

- **`<epsilon>`**, ***float, optional field***, value for numerical stability in adam. Only used when solver = 'adam'
  Default: 1e-8

15.3.8 SyntheticHistory

The SyntheticHistory sub-type uses various Time Series Analysis (TSA) algorithms to characterize and reproduce synthetic histories. It is a more general implementation of the ARMA ROM. The available algorithms are discussed in more detail below. The SyntheticHistory ROM uses the TSA algorithms to characterize then reproduce time series in sequence; for example, if using Fourier then ARMA, the SyntheticHistory ROM will characterize the Fourier properties using the Fourier TSA algorithm on a training signal, then send the residual to the ARMA TSA algorithm for characterization. Generating new signals works in reverse, first generating a signal using the ARMA TSA algorithm then superimposing the Fourier TSA algorithm.

In order to use this Reduced Order Model, the **`<ROM>`** attribute **`subType`** needs to be **`'SyntheticHistory`** (see the example below). This model can be initialized with the following child:

- **`<Features>`**, ***comma separated string, required field***, defines the features (e.g., scaling). Note that only one feature is allowed for **`'SyntheticHistory'`** and in current implementation this is used for evaluation only.
- **`<Target>`**, ***comma separated string, required field***, defines the variable(s) of the time series. Should include the pivot parameter (or Index).
- **`<pivotParameter>`**, ***string, required field***, defines the pivot variable (e.g., time) that is non-decreasing in the input HistorySet.

In addition, any number of the following TSA algorithms may be included as subnodes of this ROM:

- **`<Wavelet>`** performs a discrete wavelet transform on time-dependent data. Note: This TSA module requires pywavelets to be installed within your python environment.
  - **`target`**, ***comma sperated string, require field***, indicates which target sigals should be trained as part of this ROM using this TSA algorithm.
  - **`<family>`**, ***string, required field***, indicates which family of wavelets to use.
    There are several possible families to choose from, and most families contain more than one variation. For more information regarding the wavelet families, refer to the Pywavelets documentation.
    Possible values are:

    * **haar family**: haar
    * **db family**: db1, db2, db3, db4, db5, db6, db7, db8, db9, db10, db11, db12, db13, db14, db15, db16, db17, db18, db19, db20, db21, db22, db23, db24, db25, db26, db27, db28, db29, db30, db31, db32, db33, db34, db35, db36, db37, db38
    * **sym family**: sym2, sym3, sym4, sym5, sym6, sym7, sym8, sym9, sym10, sym11, sym12, sym13, sym14, sym15, sym16, sym17, sym18, sym19, sym20
    * **coif family**: coif1, coif2, coif3, coif4, coif5, coif6, coif7, coif8, coif9, coif10, coif11, coif12, coif13, coif14, coif15, coif16, coif17
    * **bior family**: bior1.1, bior1.3, bior1.5, bior2.2, bior2.4, bior2.6, bior2.8, bior3.1, bior3.3, bior3.5, bior3.7, bior3.9, bior4.4, bior5.5, bior6.8
    * **rbio family**: rbio1.1, rbio1.3, rbio1.5, rbio2.2, rbio2.4, rbio2.6, rbio2.8, rbio3.1, rbio3.3, rbio3.5, rbio3.7, rbio3.9, rbio4.4, rbio5.5, rbio6.8
    * **dmey family**: dmey
    * **gaus family**: gaus1, gaus2, gaus3, gaus4, gaus5, gaus6, gaus7, gaus8
    * **mexh family**: mexh
    * **morl family**: morl
    * **cgau family**: cgau1, cgau2, cgau3, cgau4, cgau5, cgau6, cgau7, cgau8
    * **shan family**: shan
    * **fbsp family**: fbsp
    * **cmor family**: cmor

- **`<PolynomialRegression>`** fits time-series data using a polynomial function of degree one or greater.

  `<PolynomialRegression>` has the following attributes:

  - `target`, ***comma seperated string, required field***, indicates which target signals should be trained as part of this ROM using this TSA algorithm.

  `<PolynomialRegression>` has the following subnodes:

  - **`<degree>`**, ***integer, required field***, indicates which degree of polynomial to fit to the presented data.

- **`<Fourier>`** uses regression to fit requested Fourier bases by their amplitudes to deterministically match the training signal. Fourier signals are defined with the following form:

$$F_m(t) = C_m \sin\left(\frac{2\pi}{k_m}t + \phi_m\right)$$

  where m indexes a particular base period k_m, C_m is the amplitude of this Fourier base in the training signal, and ϕ_m is the phase shift of this Fourier base in the training signal.

  `<Fourier>` has the following parameters:

- **target**, ***comma seperated string, required field***, indicates which target signals should be trained as part of this ROM using this TSA algorithm.
- **seed**, ***integer, optional***, provides a static seed to be used for random number generation in this ROM. Unused for the Fourier TSA algorithm.

<Fourier> further has the following subnodes:

- **<periods>**, ***comma seperated floats, required field***, indicates which base periods whose Fourier reperesentations should be fit to the training signal. For example, in a signal with hourly measurements, selecting the period **'12, 24'** would fit the daily (24-hour) and half-daily (12-hour) periodic trends.

- **<ARMA>** characterizes the signal using Auto-Regressive and Moving Average coefficients to stochastically fit the training signal. The ARMA representation has the following form:

$$A_t = \sum_{i=1}^{P} \phi_i A_{t-i} + \epsilon_t + \sum_{j=1}^{Q} \theta_j \epsilon_{t-j},$$

where t indicates a discrete time step, ϕ are the signal lag (or auto-regressive) coefficients, P is the number of signal lag terms to consider, ϵ is a random noise term, θ are the noise lag (or moving average) coefficients, and Q is the number of noise lag terms to consider. The ARMA algorithms are developed in RAVEN using the `statsmodels` Python library.

<ARMA> has the following parameters:

- **target**, ***comma seperated string, required field***, indicates which target signals should be trained as part of this ROM using this TSA algorithm.
- **seed**, ***integer, optional***, provides a static seed to be used for random number generation in this ROM. This applies both to the training and sampling of this ROM.
- **reduce_memory**, ***boolean, optional field***, activates a lower memory usage ARMA training. This does tend to result in a slightly slower training time, at the benefit of lower memory usage. For example, in one 1000-length history test, low memory reduced memory usage by 2.3 MiB (80%), but increased training time by 0.4 seconds (20%). No change in results has been observed switching between modes. Note that the ARMA must be retrained to change this property; it cannot be applied to serialized ARMAs.
  Default: False

<ARMA> further has the following subnodes:

- **<SignalLag>**, ***integer, required field***, number of signal lag terms to include in the autoregression term.
- **<NoiseLag>**, ***integer, required field***, number of noise lag terms to include in the moving average term.

SyntheticHistory Example:

```
<Simulation>
  ...
  <Models>
    ...
    <ROM name="synth" subType="SyntheticHistory">
      <Target>signal1, signal2, hour</Target>
      <Features>scaling</Features>
      <pivotParameter>hour</pivotParameter>
      <fourier target="signal1,␣signal2">
        <periods>12, 24</periods>
      </fourier>
      <arma target="signal1,␣signal2" seed='42'>
        <SignalLag>2</SignalLag>
        <NoiseLag>3</NoiseLag>
      </arma>
    </ROM>
    ...
  </Models>
  ...
</Simulation>
```

15.3.9 ARMA

The ARMA sub-type contains a single ROM type, based on an autoregressive moving average time series model with Fourier signal processing, sometimes referred to as a FARMA. ARMA is a type of time dependent model that characterizes the autocorrelation between time series data. The mathematic description of ARMA is given as

$$x_t = \sum_{i=1}^{p} \phi_i x_{t-i} + \alpha_t + \sum_{j=1}^{q} \theta_j \alpha_{t-j},$$

where x is a vector of dimension n, and ϕ_i and θ_j are both n by n matrices. When $q = 0$, the above is autoregressive (AR); when $p = 0$, the above is moving average (MA). When training an ARMA, the input needs to be a synchronized HistorySet. For unsynchronized data, use PostProcessor methods to synchronize the data before training an ARMA.

The ARMA model implemented allows an option to use Fourier series to detrend the time series before fitting to ARMA model to train. The Fourier trend will be stored in the trained

ARMA model for data generation. The following equation describes the detrending process.

$$
\begin{aligned}
x_t &= y_t - \sum_m \{a_m \sin(2\pi f_m t) + b_m \cos(2\pi f_m t)\} \\
&= y_t - \sum_m c_m \sin(2\pi f_m t + \phi_m)
\end{aligned}
$$

where $1/f_m$ is defined by the user parameter **<Fourier>**. **Note:** a_m and b_m will be calculated then transformed to c_m and ϕ. The c_m will be stored as **'amplitude'**, and ϕ will be stored as **'phase'**.

By default, each target in the training will be considered independent and have an unique ARMA for each target. Correlated targets can be specified through the **<correlate>** node, at which point the correlated targets will be trained together using a vector ARMA (or VARMA). Due to limitations in the VARMA, in order to seed samples the VARMA must be trained with the node **<seed>**, which acts independently from the global random seed used by other RAVEN entities.

Both the ARMA and VARMA make use of the `statsmodels` python package.

In order to use this Reduced Order Model, the **<ROM>** attribute **subType** needs to be **'ARMA'** (see the example below). This model can be initialized with the following child:

- **<pivotParameter>**, ***string, required field***, defines the pivot variable (e.g., time) that is non-decreasing in the input HistorySet.
- **<Features>**, ***comma separated string, required field***, defines the features (e.g., scaling). Note that only one feature is allowed for **'ARMA'** and in current implementation this is used for evaluation only.
- **<Target>**, ***comma separated string, required field***, defines the variable(s) of the time series. Should include the pivot parameter (or Index).
- **<correlate>**, ***comma separated string, optional field***, indicates the listed variables should be considered as influencing each other, and trained together instead of independently. This node can only be listed once, so all variables that are desired for correlation should be included. **Note:** The correlated VARMA takes notably longer to train than the independent ARMAs for the same number of targets.
- **<seed>**, ***integer, optional field***, provides seed for VARMA and ARMA sampling. Must be provided before training. If no seed is assigned, then a random number will be used.
- **<reseedCopies>**, ***boolean, optional field***, if **'True'** then whenever the ARMA is loaded from file, a random reseeding will be performed to ensure different histories. **Note:** If reproducible histories are desired for an ARMA loaded from file, **<reseedCopies>** should

be set to **'False'**, and in the **<RunInfo>** block **<batchSize>** needs to be 1 and **<internalParallel>** should be **'False'** for RAVEN runs sampling the trained ARMA model.

If **<InternalParallel>** is **'True'** and the ARMA has **<reseedCopies>** as **'False'**, an identical ARMA history will always be provided regardless of how many samples are taken.

If **<InternalParallel>** is **'False'** and **<batchSize>** is more than 1, it is not possible to guarantee the order of RNG usage by the separate processes, so it is not possible to guarantee reproducible histories are generated.
Default: True

- **<P>**, ***integer, optional field***, defines the value of p.
  Default: 3

- **<Q>**, ***integer, optional field***, defines the value of q.
  Default: 3

- **<Fourier>**, ***comma-separated integers, optional field***, must be positive integers. This defines the based period that will be used for Fourier detrending, i.e., this field defines $1/f_m$ in the above equation. When this filed is not specified, the ARMA considers no Fourier detrend.

- **<outTruncation>**, ***comma-separated string, optional***, defines whether and how output time series are limited in domain. This node has one attribute, **domain**, whose value can be **'positive'** or **'negative'**. The value of this node contains the list of targets to whom this domain limitation should be applied. In the event a negative value is discovered in a target whose domain is strictly positive, the absolute value of the original negative value will be used instead, and similarly for the negative domain.
  Default: None

- **<Peaks>**, ***node, optional***, designed to estimate the peaks in signals that repeat with some frequency, often in periodic data. The **<Peaks>** node has the following attributes:

  - **target**, ***required string attribute***, defines the name of one target (besides the pivot parameter) expected to have periodic peaks.
  - **threshold**, ***required float attribute***, user-defined minimum required height of peaks (absolute value).
  - **period**, ***required float attribute***, user-defined expected period for target variable.

  For example,

```
<Peaks target='Speed' threshold='0' period='86400'>
```

  This means the **'Speed'** signal is a daily signal, and the minimal height required to define a peak is 0. Additionally, the **<Peaks>** requires the sub-node:

- **`<window>`**, ***comma-separated floats, required field***, lists the window of time within each period in which a peak should be discovered. The text of this node is the upper and lower boundary of this window *relative to* the start of the period, separated by a comma. User can define the lower bound to be a negative number if the window passes through one side of one period. For example, if the period is 24 hours, the window can be -2,2 which is equivalent to 22, 2. This node has one attribute:
  - **`width`**, ***required float attribute*** which is the user defined width of peaks in that window. The width is in the unit of the signal as well.

  The number of **`<window>`** is unlimited.

- **`<preserveInputCDF>`**, ***boolean, optional field***, enables a final transform on sampled data coercing it to have the same distribution as the original data. If **`'True'`**, then every sample generated by this ARMA after training will have a distribution of values that conforms within numerical accuracy to the original data. This is especially useful when variance is desired not to stretch the most extreme events (high or low signal values), but instead the sequence of events throughout this history. For example, this transform can preserve the load duration curve for a load signal.
  Default: False

- **`<ZeroFilter>`**, ***comma-separated string, optional field***, turns on *zero filtering* for the listed targets. Zero filtering is a very specific algorithm, and should not be used without understanding its application. When zero filtering is enabled, the ARMA will remove all the values from the training data equal to zero for the target, then train on the remaining data (including Fourier detrending if applicable). If the target is set as correlated to another target, the second target will be treated as two distinct series: one containing times in which the original target is zero, and one in the remaining times. The results from separated ARMAs are recombined after sampling. This can be a methodology for treating histories with long zero-value segments punctuated periodically by peaks.

- **`<SpecificFourier>`**, ***node, optional***, provides a means to specify different Fourier decomposition for different target variables. Values given in the subnodes of this node will supercede the defaults set by the **`<Fourier>`** and **`<FourierOrder>`** nodes. This node requires the following attribute:
  - **`variables`**, ***comma-separated list, required field***, lists the variables to whom the **`<SpecificFourier>`** parameters will apply.

  Additionally, the **`<SpecificFourier>`** node takes the following subnodes:
  - **`<periods>`**, ***comma-separated integers, required field***, lists the (fundamental) periodic wavelength of the Fourier decomposition for these variables, as in the **`<Fourier>`** general node.

- **`<Multicycle>`**, ***node, optional***, indicates that each sample of the ARMA should yield multiple sequential samples. For example, if an ARMA model is trained to produce a year's

worth of data, enabling **<Multicycle>** causes it to produce several successive years of data. Multicycle sampling is independent of ROM training, and only changes how samples of the ARMA are created.

Note: The output of a multicycle ARMA must be stored in a **<DataSet>**, as the targets will depend on both the **<pivotParameter>** as well as the cycle, **'Cycle'**. The cycle is a second **<Index>** that all targets should depend on, with variable name **'Cycle'**.

The **<Multicycle>** node accepts the following subnodes:

- **<cycles>**, ***integer, required field***, the number of cycles the ARMA should produce each time it yields a sample.
- **<growth>**, ***float, optional field***, if provided then the histories produced by the ARMA will be increased by the growth factor for successive cycles. This node can be added multiple times with different settings for different targets. The text of this node is the growth factor in percentage. Some examples are in Table 4, where *Growth factor* is the value used in the RAVEN input and *Scaling factor* is the value by which the history will be multiplied.

Growth factor	Scaling factor	Description
50	1.5	growing by 50% each cycle
-50	0.5	shrinking by 50% each cycle
150	2.5	growing by 150% each cycle

Table 4: ARMA Growth Factor Examples

  The **<growth>** node takes the following attributes as settings:

  * **targets**, ***comma-seperated list, required field***, lists the targets in this ARMA that this growth factor should apply to.
  * **mode**, ***string, required field***, either **'linear'** or **'exponential'**, determines the manner in which the growth factor is applied.
    If **'linear'**, then the scaling factor is $(1 + y \cdot g/100)$;
    if **'exponential'**, then the scaling factor is $(1 + g/100)^y$;
    where y is the cycle after the first and g is the provided scaling factor.

- ****, ***node, optional***, provides an alternative way to build the ROM. When this mode is enabled, the subspace of the ROM (e.g. "time") will be divided into segments as requested, then a distinct ROM will be trained on each of the segments. This is especially helpful if during the subspace the ROM representation of the signal changes significantly. For example, if the signal is different during summer and winter, then a signal can be divided and a distinct ROM trained on the segments. By default, no segmentation occurs.

  To futher enable clustering of the segments, the **** has the following attributes:

  - **grouping**, ***string, optional field*** enables the use of ROM subspace clustering in addition to segmenting if set to **'cluster'**. If set to **'segment'**, then performs

segmentation without clustering. If clustering, then an additional node needs to be included in the **** node, as described below.
Default: segment

This node takes the following subnodes:

- **<subspace>**, ***string, required field*** designates the subspace to divide. This should be the pivot parameter (often "time") for the ROM. This node also requires an attribute to determine how the subspace is divided, as well as other attributes, described below:
  - **pivotLength**, ***float, optional field***, provides the value in the subspace that each segment should attempt to represent, independently of how the data is stored. For example, if the subspace has hourly resolution, is measured in seconds, and the desired segmentation is daily, the **pivotLength** would be 86400. Either this option or **divisions** must be provided.
  - **divisions**, ***integer, optional field***, as an alternative to **pivotLength**, this attribute can be used to specify how many data points to include in each subdivision, rather than use the pivot values. The algorithm will attempt to split the data points as equally as possible. Either this option or **pivotLength** must be provided.
  - **shift**, ***string, optional field***, governs the way in which the subspace is treated in each segment. By default, the subspace retains its actual values for each segment; for example, if each segment is 4 hours long, the first segment starts at time 0, the second at 4 hours, the third at 8 hours, and so forth. Options to change this behavior are **'zero'** and **'first'**. In the case of **'zero'**, each segment restarts the pivot with the subspace value as 0, shifting all other values similarly. In the example above, the first segment would start at 0, the second at 0, and the third at 0, with each ending at 4 hours. Note that the pivot values are restored when the ROM is evaluated. Using **'first'**, each segment subspace restarts at the value of the first segment. This is useful in the event subspace 0 is not a desirable value.
- **<Classifier>**, ***string, optional field*** associates a **<PostProcessor>** defined in the **<Models>** block to this segmentation. If clustering is enabled (see **grouping** above), then this associated Classifier will be used to cluster the segmented ROM subspaces. The attributes **class='Models'** and **type='PostProcessor'** must be set, and the text of this node is the **name** of the requested Classifier. Note this Classifier must be a valid Classifier; not all PostProcessors are suitable. For example, see the DataMining PostProcessor subtype Clustering.
- **<clusterFeatures>**, ***string, optional field***, if clustering then delineates the fundamental ROM features that should be considered while clustering. The available features are ROM-dependent, and an exception is raised if an unrecognized request is given. See individual ROMs for options.
  *Default: A*ll ROM-specific options.
- **<evalMode>**, ***string, optional field***, one of **'truncated'**, **'full'**, or **'clustered'**, determines how the evaluations are represented, as follows:

    * **'full'**, reproduce the full signal using representative cluster segments,
    * **'truncated'**, reproduce a history containing exactly segment from each cluster placed back-to-back, with the **<pivotParameter>** spanning the clustered dimension. Note this will almost surely not be the same length as the original signal; information about indexing can be found in the ROM's XML metadata.
    * **'clustered'**, reproduce a N-dimensional object with the variable `_ROM_cluster` as one of the indexes for the ROM's sampled variables. Note that in order to use the option, the receiving **<DataObject>** should be of type **<DataSet>** with one of the indices being `_ROM_cluster`.
  - **<evaluationClusterChoice>**, ***string, optional field***, one of **'first'** or **'random'**, determines, if **grouping**= $cluster$, which strategy needs to be followed for the evaluation stage. If "first", the first ROM (representative segmented ROM),in each cluster, is considered to be representative of the full space in the cluster (i.e. the evaluation is always performed interrogating the first ROM in each cluster); If "random", a random ROM, in each cluster, is choosen when an evaluation is requested. **Note:** if "first" is used, there is *substantial* memory savings when compared to using "random".
    Default: first

Note that when loading the ARMA model from a serialized file as a **<pickledROM>**, several nodes can be used to modify the evaluation behavior:

- **<reseed>**, ***integer, optional***,
- **<Multicycle>**, ***node, optional***, as **<Multicycle>** above, allows resetting a growth factor and number of cycles to sample.
- **<clusterEvalMode>**, ***string, optional***, one of **'truncated'**, **'full'**, or **'clustered'**, changes the structure of the samples for Clustered Segmented ROMs. These are identical to the options for **<evalMode>** node under **<Segmented>** as described above.

the **<seed>** as well as the **<Multicycle>** nodes can be added to change the behavior of the ARMA ROM.

When using ROM segmentation/clustering, the ARMA provides the following classes of features that can be used for clustering:

- **'global'**, the segment-long mean values of the signal;
- **'fourier'**, the fundamental Fourier frequency amplitudes discovered during training;
- **'arma'**, the standard deviation, p, and q coefficients obtained during training;

- **'peak'**, the peak probability, mean, standard deviation, and most probable index discovered during training.

General ARMA Example:

```
<Simulation>
  ...
  <Models>
    ...
    <ROM name='aUserDefinedName' subType='ARMA'>
      <pivotParameter>Time</pivotParameter>
      <Features>scaling</Features>
      <Target>Speed1,Speed2</Target>
      <P>5</P>
      <Q>4</Q>
      <Segment>
        <subspace pivotLength="1296000"
           shift="first">Time</subspace>
      </Segment>
      <preserveInputCDF>True</preserveInputCDF>
      <Fourier>604800,86400</Fourier>
      <FourierOrder>2, 4</FourierOrder>
      <Peaks target='Speed1' threshold='0.1' period='86400'>
        <window width='14400' >-7200,10800</window>
        <window width='18000' >64800,75600</window>
      </Peaks>
     </ROM>
    ...
  </Models>
  ...
</Simulation>
```

15.3.10 PolyExponential

The PolyExponential sub-type contains a single ROM type, aimed to construct a time-dependent (or any other monotonic variable) surrogate model based on polynomial sum of exponential term. This surrogate have the form:

$$SM(X, z) = \sum_{i=1}^{N} P_i(X) \times \exp(-Q_i(X) \times z) \tag{28}$$

where:

- $\mathbf{z}$ is the independent monotonic variable (e.g. time)
- $\mathbf{X}$ is the vector of the other independent (parametric) variables (Features)
- $\mathbf{P_i}(X)$ is a polynomial of rank M function of the parametric space X
- $\mathbf{Q_i}(X)$ is a polynomial of rank M function of the parametric space X
- $\mathbf{N}$ is the number of requested exponential terms.

It is crucial to notice that this model is quite suitable for FOMs whose drivers are characterized by an exponential-like behavior. In addition, it is important to notice that the exponential terms' coefficients are computed running a genetic-algorithm optimization problem, which is quite slow in case of increasing number of "numberExpTerms". In order to use this Reduced Order Model, the **<ROM>** attribute **subType** needs to be set equal to **'PolyExponential'** (see the example below). This model can be initialized with the following child:

- **<pivotParameter>**, ***string, optional field***, defines the pivot variable (e.g., time) that represents the independent monotonic variable
  Default: time
- **<Features>**, ***comma separated string, required field***, defines the features (i.e. input parameters) of this model
- **<Target>**, ***comma separated string, required field***, defines output FOMs that are going to be predicted
- **<numberExpTerms>**, ***integer, optional field***, the number of exponential terms to be used (N above)
  Default: 3
- **<coeffRegressor>**, ***string, optional field***, defines which regressor to use for interpolating the exponential coefficient. Available are "spline","poly" and "nearest".
  Default: spline
- **<polyOrder>**, ***integer, optional field***, the polynomial order to be used for interpolating the exponential coefficients. Only valid in case of **<coeffRegressor>** set to "poly".
  Default: 2
- **<tol>**, ***float, optional field***, relative tolerance of the optimization problem (differential evolution optimizer)
  Default: 1e-3
- **<maxNumberIter>**, ***integer, optional field***, maximum number of iterations (generations) for the optimization problem (differential evolution optimizer)
  Default: 5000

Example:

```
<Simulation>
  ...
  <Models>
    ...
   <ROM name='PolyExp' subType='PolyExponential'>
     <Target>time,decay_heat, xe135_dens</Target>
     <Features>enrichment,bu</Features>
     <pivotParameter>time</pivotParameter>
     <numberExpTerms>5</numberExpTerms>
     <max_iter>1000000</max_iter>
     <tol>0.000001</tol>
  </ROM>
    ...
  </Models>
  ...
</Simulation>
```

Once the ROM is trained (**Step** **`<RomTrainer>`**), its coefficients can be exported into an XML file via an **`<OutStream>`** of type **`Print`**. The following variable/parameters can be exported (i.e. **`<what>`** node in **`<OutStream>`** of type **`Print`**):

- **`<expTerms>`**, see XML input specifications above, inquired pre-pending the keyword "output—" (e.g. output— expTerms)
- **`<coeffRegressor>`**, see XML input specifications above
- **`<polyOrder>`**, see XML input specifications above
- **`<features>`**, see XML input specifications above
- **`<timeScale>`**, XML node containing the array of the training time steps values
- **`<coefficients>`**, XML node containing the exponential terms' coefficients for each realization

See the following example:

```
<Simulation>
  ...
  <OutStreams>
    ...
    <Print name = 'dumpAllCoefficients'>
```

```
      <type>xml</type>
      <source>PolyExp</source>
      <!--
        here the <what> node is omitted. All the available
           params/coefficients
        are going to be printed out
      -->
    </Print>
    <Print name = 'dumpSomeCoefficients'>
      <type>xml</type>
      <source>PolyExp</source>
      <what>coefficients,timeScale</what>
    </Print>
    ...
  </OutStreams>
  ...
</Simulation>
```

15.3.11 DMD

The DMD sub-type contains a single ROM type, aimed to construct a time-dependent (or any other monotonic variable) surrogate model based on Dynamic Mode Decomposition (ref. [5] and [6]). This surrogate is aimed to perform a "dimensionality reduction regression", where, given time series (or any monotonic-dependent variable) of data, a set of modes each of which is associated with a fixed oscillation frequency and decay/growth rate is computed in order to represent the data-set. In order to use this Reduced Order Model, the **`<ROM>`** attribute **`subType`** needs to be set equal to **`'DMD'`** (see the example below). This model can be initialized with the following child:

- **`<dmdType>`**, ***string, optional field***, the type of Dynamic Mode Decomposition to apply. Available are:
  - *dmd*, for classical DMD
  - *hodmd*, for high order DMD.

  Default: dmd

- **`<pivotParameter>`**, ***string, optional field***, defines the pivot variable (e.g., time) that represents the independent monotonic variable
  Default: time

- **`<Features>`**, ***comma separated string, required field***, defines the features (i.e. input parameters) of this model
- **`<Target>`**, ***comma separated string, required field***, defines output FOMs that are going to be predicted
- **`<rankSVD>`**, ***integer, optional field***, defines the truncation rank to be used for the SVD. Available options are:
  - *-1*, no truncation is performed
  - *0*, optimal rank is internally computed
  - *¿1*, this rank is going to be used for the truncation

  Default: -1
- **`<energyRankSVD>`**, ***float, optional field***, energy level ($0.0 < float < 1.0$) used to compute the rank such as computed rank is the number of the biggest singular values needed to reach the energy identified by **`<energyRankSVD>`**. This node has always priority over **`<rankSVD>`**
  Default: None
- **`<rankTLSQ>`**, ***integer, optional field***, $int > 0$ that defines the truncation rank to be used for the total least square problem. If not inputted, no truncation is applied
  Default: None
- **`<exactModes>`**, ***bool, optional field***, True if the exact modes need to be computed (eigenvalues and eigenvectors), otherwise the projected ones (using the left-singular matrix after SVD).
  Default: True
- **`<optimized>`**, ***float, optional field***, True if the amplitudes need to be computed minimizing the error between the modes and all the time-steps or False, if only the 1st timestep only needs to be considered
  Default: True

Example:

```
<Simulation>
  ...
  <Models>
    ...
   <ROM name='DMD' subType='DMD'>
      <Target>time,totals_watts, xe135_dens</Target>
      <Features>enrichment,bu</Features>
```

```
      <dmdType>dmd</dmdType>
      <pivotParameter>time</pivotParameter>
      <rankSVD>0</rankSVD>
      <rankTLSQ>5</rankTLSQ>
      <exactModes>False</exactModes>
      <optimized>True</optimized>
    </ROM
    ...
  </Models>
  ...
</Simulation>
```

Once the ROM is trained (**Step** **<RomTrainer>**), its parameters/coefficients can be exported into an XML file via an **<OutStream>** of type **Print**. The following variable/parameters can be exported (i.e. **<what>** node in **<OutStream>** of type **Print**):

- **<rankSVD>**, see XML input specifications above
- **<energyRankSVD>**, see XML input specifications above
- **<rankTLSQ>**, see XML input specifications above
- **<exactModes>**, see XML input specifications above
- **<optimized>**, see XML input specifications above
- **<features>**, see XML input specifications above
- **<timeScale>**, XML node containing the array of the training time steps values
- **<dmdTimeScale>**, XML node containing the array of time scale in the DMD space (can be used as mapping between the **<timeScale>** and **<dmdTimeScale>**)
- **<eigs>**, XML node containing the eigenvalues (imaginary and real part)
- **<amplitudes>**, XML node containing the amplitudes (imaginary and real part)
- **<modes>**, XML node containing the dynamic modes (imaginary and real part)

See the following example:

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

```
    <Print name = 'dumpAllCoefficients'>
      <type>xml</type>
      <source>DMD</source>
      <!--
        here the <what> node is omitted. All the available
           params/coefficients
        are going to be printed out
      -->
    </Print>
    <Print name = 'dumpSomeCoefficients'>
      <type>xml</type>
      <source>PolyExp</source>
      <what>eigs,amplitudes,modes</what>
    </Print>
    ...
  </OutStreams>
  ...
</Simulation>
```

15.3.12 TensorFlow-Keras Deep Neural Networks

It is important to NOTE that Python3 is required in order to use these deep neural networks. If python2 is installed, these ROMs will not be imported by RAVEN, and an error will be raised if the user tries to use these capabilities.

TensorFlow is an open source software library for high performance numerical computation. Its flexible architecture allows easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices. Originally developed by researchers and engineers from the Google Brain team within Google's AI organization, it comes with strong support for machine learning and deep learning and the flexible numerical computation core is used across many other scientific domains.

Keras is a high-level API to build and train deep learning models. It's used for fast prototyping, advanced research, and production, with three key advantages:

- *User friendly*: Keras has a simple, consistent interface optimized for common use cases. It provides clear and actionable feedback for user errors.
- *Modular and composable*: Keras models are made by connecting configurable building blocks together, with few restrictions.

- *Easy to extend*: Write custom building blocks to express new ideas for research. Create new layers, loss functions, and develop state-of-the-art models.

tf.keras is TensorFlow's implementation of the Keras API specification. This is a high-level API to build and train models that include first-class support for TensorFlow-specific functionality, such as eager *execution*, *tf.data* pipelines, and *Estimators*. **tf.keras** makes TensorFlow easier to use without sacrificing flexibility and performance. RAVEN will utilize this high-level API to build and train deep neural networks (DNNs) as ROMs, and these ROMs can be employed by other RAVEN entities to perform uncertainty quantification, model opimization and data analysis.

Before analyzing each classifier in detail, it is important to mention that each type has a similar syntax. In the example below, the subnodes that can be included in the main XML node **<ROM>** are reported: **Example:**

```
<Simulation>
  ...
  <Models>
    ...
    <ROM name='aUserDefinedName' subType='whatever'>
      <Features>X,Y</Features>
      <Target>Z</Target>
      <loss>mean_squared_error</loss>
      <metrics>accuracy</metrics>
      <batch_size>4</batch_size>
      <epochs>4</epochs>
      <num_classes>2</num_classes>
      <validation_split>0.25</validation_split>
      <optimizerSetting>
        <optimizer>Adam</optimizer>
        ...
      </optimizerSetting>
      <WhateverLayer1 name="layerName1">
        ...
      </WhateverLayer1>
      ...
      <WhateverLayerN name="layerNameN">
        ...
      </WhateverLayerN>
      <layer_layout>layerName1, ..., layerNameN</layer_layout>
    </ROM>
    ...
  </Models>
  ...
```

```
</Simulation>
```

As shown in above example, in addition to the common subnodes **<Target>** and **<Features>**, the **<ROM>** of DNNs can be initialized with the following children:

- **<loss>**, ***string or comma separated string, optional field***, if the model has multiple outputs, you can use a different loss metric on each output by passing a list of loss metrics. The value that will be minimized by the model will then be the sum of all individual value from each loss metric. Available loss functions include *mean_squared_error*, *mean_absolute_error*, *mean_absolute_percentage_error*, *mean_squared_logarithmic_error*, *squared_hinge*, *hinge*, *categorical_hinge*, *logcosh*, *categorical_crossentropy*, *sparse_categorical_crossentropy*, *binary_crossentropy*, *kullback_leibler_divergence*, *poisson*, *cosine_proximity*.
  Default: mean_squared_error

- **<metrics>**, ***string or comma separated string, optional field***, list of metrics to be evaluated by the model during training and testing. available metrics include *binary_accuracy*, *categorical_accuracy*, *sparse_categorical_accuracy*, *top_k_categorical_accuracy*, *sparse_top_k_categorical_*

  Default: accuracy

- **<batch_size>**, ***integer, optional field***, number of samples per gradient update.
  Default: 20

- **<epochs>**, ***integer, optional field***, number of epochs to train the model. An epoch is an iteration over the entire training data.
  Default: 20

- **<num_classes>**, ***positive integer, optional field***, dimensionality of the output space of given classifier.
  Default: 1

- **<validation_split>**,***float between 0 and 1, optional field***, fraction of the training data to be used as validation data.
  Default: 0.25

- **<plot_model>**, ***boolean, optional field***, if true the DNN model constructed by RAVEN will be plotted and stored in the working directory. The file name will be *"ROM name"* + *"_"* + *"model.png"*. **Note:** This capability requires the following libraries, i.e. pydot-ng and graphviz to be installed.
  Default: False

- **<optimizerSetting>**, ***optional field***, including several subnode depending on the type of optimizers.

  - **<optimizer>**, ***string, optional field***, name of optimizer.

Default: Adam **Note:** The users can also choose different optimizers to train the ROM. The default algorithm is *Adam*. Other available optimizers include: *SGD*, *RMSprop*, *Adagrad*, *Adadelta*, *Adamx*, *Nadam*. For the detailed information, i.e. the parameters for each optimization, the user can refer to `https://keras.io/optimizers/`. In raven, the user can use **`<optimizerSetting>`** to set the parameters of the above optimizer as follows:

- **Adam**, adam optimizer
  - **`<beta_1>`**, ***float, optional field***, $0 < beta < 1$. Generally close to 1.
    Default: 0.9
  - **`<beta_2>`**, ***float, optional field***, $0 < beta < 1$. Generally close to 1.
    Default: 0.999
  - **`<epsilon>`**, ***float, optional field***, fuzz factor.
    Default: None
  - **`<decay>`**, ***float, optional field***, learning rate decay over each update.
    Default: 0.0
  - **`<lr>`**, ***float, optional field***, learning rate.
    Default: 0.001
- **SGD**, stochastic gradient descent optimizer.
  - **`<momentum>`**, ***float, optional field***, > 0. Parameter that accelerates SGD in the relevant direction and dampens oscillations.
    Default: 0.0
  - **`<nesterov>`**, ***boolean, optional field***, whether to apply Nesterov momentum
    Default: False
  - **`<decay>`**, ***float, optional field***, learning rate decay over each update.
    Default: 0.0
  - **`<lr>`**, ***float, optional field***, learning rate.
    Default: 0.001
- **RMSprop**, RMSProp optimizer.
  - **`<rho>`**, ***float, optional field***, > 0.
    Default: 0.9
  - **`<decay>`**, ***float, optional field***, learning rate decay over each update.
    Default: 0.0
  - **`<lr>`**, ***float, optional field***, learning rate.
    Default: 0.001
  - **`<epsilon>`**, ***float, optional field***, fuzz factor.
    Default: None
- **Adagrad**, Adagrad optimizer.
  - **`<decay>`**, ***float, optional field***, learning rate decay over each update.
    Default: 0.0

    * **<lr>**, ***float, optional field***, learning rate.
      Default: 0.01
    * **<epsilon>**, ***float, optional field***, fuzz factor.
      Default: None
  - **Adadelta**, Adadelta optimizer.
    * **<decay>**, ***float, optional field***, learning rate decay over each update.
      Default: 0.0
    * **<lr>**, ***float, optional field***, learning rate.
      Default: 1.0
    * **<epsilon>**, ***float, optional field***, fuzz factor.
      Default: None
    * **<rho>**, ***float, optional field***, > 0.
      Default: 0.95
  - **Adamax**, Adamax optimizer
    * **<beta_1>**, ***float, optional field***, $0 < beta < 1$. Generally close to 1.
      Default: 0.9
    * **<beta_2>**, ***float, optional field***, $0 < beta < 1$. Generally close to 1.
      Default: 0.999
    * **<epsilon>**, ***float, optional field***, fuzz factor.
      Default: None
    * **<decay>**, ***float, optional field***, learning rate decay over each update.
      Default: 0.0
    * **<lr>**, ***float, optional field***, learning rate.
      Default: 0.002
  - **Nadam**,
    * **<beta_1>**, ***float, optional field***, $0 < beta < 1$. Generally close to 1.
      Default: 0.9
    * **<beta_2>**, ***float, optional field***, $0 < beta < 1$. Generally close to 1.
      Default: 0.999
    * **<epsilon>**, ***float, optional field***, fuzz factor.
      Default: None
    * **<lr>**, ***float, optional field***, learning rate.
      Default: 0.002

- **<layer_layout>**, ***comma seperated string, required***, the layout/order of layers in the deep neural networks. The values in the subnode should be the name of layers defined in layer node, such as **<Dense>**, **<Dropout>**, and **<Conv1D>**.

Note: The descriptions regarding the **<WhateverLayer>** node will be introduced in following subsections. Basically, different classifiers will require different layers. In addition, most core layers will accept the **<activation>** subnode (see 15.3.12.1).

15.3.12.1 Activation Functions

Activations can either be used through an **<Activation>** layer, or through the **<activation>** argument supported by all forward layers. Available activations include:

- *relu*, the rectified linear unit function, returns $f(x) = max(0, x)$.
- *tanh*, the hyperbolic tan function, returns $f(x) = tanh(x)$.
- *elu*, exponential linear units try to make the mean activations closer to zero which speeds up learning. $f(x) = x$ if $x \geq 0$, otherwise $(exp(x) - 1.)$.
- *selu*, scaled exponential linear unit, i.e. $scale * elu(x, alpha)$, where $scale, alpha$ are predefined constants.
- *softplus*, a smooth approximation to the rectifier linear unit function, return $f(x) = log(1. + exp(x))$.
- *softsign*, return $f(x) = \frac{x}{1.+|x|}$.
- *sigmoid*,return $f(x) = \frac{1.}{1.+exp(-x)}$.
- *hard_sigmoid*, hard sigmoid activation function.
- *linear*, i.e. identity.
- *softmax*, softmax activation function, return $f(x) = \frac{exp(x_i)}{\sum_i exp(x_i)}$

15.3.12.2 Initializer Functions

Initializations define the way to set the initial random weights of TensorFlow-Keras layers. The keyword arguments used to passing initializers to layers will depend on the layer. Usually it is simply **<kernel_initializer>** and **<bias_initializer>**. Available initializers include:

- *Zeros*, generates tensors initialized to 0.
- *Ones*, generates tensors initialized to 1.
- *Constant*, generates tensors initialized to a constant value.
- *RandomNormal*, generates tensors with a normal distribution.
- *RandomUniform*, generates tensors with a uniform distribution.
- *TruncatedNormal*, generates a truncated normal distribution.

- *VarianceScaling*, initializer capable of adapting its scale to the shape of weights.
- *Orthogonal*, generates a random orthogonal matrix.
- *Identity*, generates the identity matrix.
- *lecun_uniform*, LeCun uniform initializer. It draws samples from a uniform distribution within $[-limit, limit]$ where *limit* is $sqrt(3/fanIn)$ where *fanIn* is the number of input dimensions in the weight tensor.
- *glorot_normal*, Glorot normal initializer. It draws samples from a truncated normal distribution centered on 0 with $stddev = sqrt(2/(fanIn + fanOut))$ where *fanIn* is the number of input dimensions in the weight tensor and *fanOut* is the number of output dimensions in the weight tensors.
- *glorot_uniform*, Glorot uniform initializer. It draws samples from a uniform distribution within $[-limit, limit]$ where *limit* is $sqrt(6/(fanIn + fanOut))$.
- *he_normal*, He normal initializer. It draws samples from a truncated normal distribution centered on 0 with $stddev = sqrt(2/fanIn)$.
- *lecun_normal*, LeCun normal initializer. It draws samples from a truncated normal distribution centered on 0 with $stddev = sqrt(1/fanIn)$.
- *he_uniform*, He uniform variance scaling initializer. It draws samples from a uniform distribution within $[-limit, limit]$ where *limit* is $sqrt(6/fanIn)$ where *fanIn* is the number of input dimensions in the weight tensor.

15.3.12.3 Regularizer Functions

Regularizers allow to apply penalities on layer parameters or layer activity during optimization. These penalties are incorporated in the loss function that the network optimizes. The exact API will depend on the layer, but the layers **`<Dense, Conv1D, Conv2D, and Conv3D>`** have a unified API. Available regularizers include:

- *l1*, l1 regularization
- *l2*, l2 regularization
- *l1_l2*, l1 and l2 regularization

15.3.12.4 Constraint Functions

Functions from the *constraint* module allow setting constraints on network parameters during optimization. Available constraints include:

- *MaxNorm*, constrains the weights incident to each hidden unit to have a norm less than or equal to a desired value.
- *NonNeg*, constrains the weights to be non-negative
- *UnitNorm*, constrains the weights incident to each hidden unit to have unit norm.
- *MinMaxNorm*, constrains the weights incident to each hidden unit to have the norm between a lower bound and an upper bound.

15.3.12.5 KerasMLPClassifier

Multi-Layer Perceptron (MLP) (or Artificial Neural Network - ANN), a class of feedforward ANN, can be viewed as a logistic regression classifier where input is first transformed using a non-linear transformation. This transformation probjects the input data into a space where it becomes linearly separable. This intermediate layer is referred to as a **hidden layer**. An MLP consists of at least three layers of nodes. Except for the input nodes, each node is a neuron that uses a nonlinear **activation function**. MLP utilizes a suppervised learning technique called **Backpropagation** for training. Generally, a single hidden layer is sufficient to make MLPs a universal approximator. However, many hidden layers, i.e. deep learning, can be used to model more complex nonlinear relationships. The extra layers enable composition of features from lower layers, potentially modeling complex data with fewer units than a similarly performing shallow network.

It is important to NOTE that RAVEN uses a Z-score normalization of the training data before constructing the *KerasMLPClassifier* ROM:

$$\mathbf{X}' = \frac{(\mathbf{X} - \mu)}{\sigma} \tag{29}$$

In order to use this ROM, the **`<ROM>`** attribute **`subType`** needs to be **`'KerasMLPClassifier'`** (see the example below). This model can be initialized with the following layers:

- **`<Dense>`**, ***required field***, regular densely-connected neural network layer. This node require the following attribute:

- **name**, ***string, required field***, name of this layer. The value will be used in **<layer_layout>** to construct the fully connected neural network.

In addition, this node also accepts the following subnodes

- **<activation>**, ***string, optional field***, including 'relu', 'tanh', 'elu', 'selu', 'softplus', 'softsign', 'sigmoid', 'hard_sigmoid', 'linear', 'softmax'. (see 15.3.12.1)
  Default: linear
- **<dim_out>**, ***positive integer, required except if this layer is used as the last output layer***, dimensionality of the output space of this layer
- **<use_bias>**, ***boolean, optional field***, whether the layer uses a bias vector.
  Default: True
- **<kernel_initializer>**, ***string, optional field***, initializer for the kernel weights matrix (see 15.3.12.2).
  Default: glorot_uniform
- **<bias_initializer>**, ***string, optional field***, intializer for the bias vector (see 15.3.12.2).
  Default: zeros
- **<kernel_regularizer>**, ***string, optional field***, regularizer function applied to the kernel weights matrix (see 15.3.12.3).
  Default: None
- **<bias_regularizer>**, ***string, optional field***, regularizer function applied to the bias vector (see 15.3.12.3).
  Default: None
- **<activity_regularizer>**, ***string, optional field***, regularizer function applied to the output of the layer (its "activation"). (see 15.3.12.3)
  Default: None
- **<kernel_constraint>**, ***string, optional field***, constraint function applied to the kernel weights matrix (see 15.3.12.4).
  Default: None
- **<bias_constraint>**, ***string, optional field***, constraint function applied to the bias vector (see 15.3.12.4)
  Default: None

• **<Dropout>**, ***optional field***, applies Dropout to the input. Dropout consists in randomly setting a fraction **<rate>** of input units to 0 at each update during training time, which helps prevent overfitting. This node require the following attribute:

- **name**, ***string, required field***, name of this layer. The value will be used in **<layer_layout>** to construct the fully connected neural network.

In addition, this node also accepts the following subnode

- **<rate>**, ***float between 0 and 1, optional field***, fraction of the input units to drop.
  Default: 0
- **<noise_shape>**, ***list of integers, optional field***, 1D integer tensor representing the shape of the binary dropout mask that will be multiplied with the input.
  Default: None
- **<seed>**, ***integer, optional field***, a integer to use as random seed.
  Default: None

Example:

```
<Simulation>
  ...
  <Models>
    ...
    <ROM name='aUserDefinedName' subType='KerasMLPClassifier'>
      <Features>X,Y</Features>
      <Target>Z</Target>
      <loss>mean_squared_error</loss>
      <metrics>accuracy</metrics>
      <batch_size>4</batch_size>
      <epochs>4</epochs>
      <optimizerSetting>
        <beta_1>0.9</beta_1>
        <optimizer>Adam</optimizer>
        <beta_2>0.999</beta_2>
        <epsilon>1e-8</epsilon>
        <decay>0.0</decay>
        <lr>0.001</lr>
      </optimizerSetting>
      <Dense name="layer1">
          <activation>relu</activation>
          <dim_out>15</dim_out>
      </Dense>
      <Dropout name="dropout1">
          <rate>0.2</rate>
      </Dropout>
      <Dense name="layer2">
          <activation>tanh</activation>
          <dim_out>8</dim_out>
      </Dense>
      <Dropout name="dropout2">
          <rate>0.2</rate>
```

```
      </Dropout>
      <Dense name="outLayer">
          <activation>sigmoid</activation>
      </Dense>
      <layer_layout>layer1, dropout1, layer2, dropout2,
         outLayer</layer_layout>
    </ROM>
    ...
  </Models>
  ...
</Simulation>
```

15.3.12.6 KerasConvNetClassifier

Convolutional Neural Network (CNN) is a deep learning algorithm which can take in an input image, assign importance to various objects in the image and be able to differentiate one from the other. The architecture of a CNN is analogous to that of the connectivity pattern of Neurons in the Human Brain and was inspired by the organization of the Visual Cortex. Individual neurons respond to stimuli only in a restricted region of the visual field known as the Receptive Field. A collection of such fields overlap to cover the entire visual area. CNN is able to successfully capture the spatial and temporal dependencies in an image through the applicaiton of relevant filters. The architecture performs a better fitting to the image dataset due to the reduction in the number of parameters involved and reusability of weights. In other words, the network can be trained to understand the sophistication of the image better.

It is important to NOTE that RAVEN uses a Z-score normalization of the training data before constructing the *KerasConvNetClassifier* ROM:

$$\mathbf{X}' = \frac{(\mathbf{X} - \mu)}{\sigma} \tag{30}$$

In order to use this ROM, the **<ROM>** attribute **subType** needs to be **'KerasConvNetClassifier'** (see the example below). This model can be initialized with the following layers:

- **<Dense>**, ***required field***, regular densely-connected neural network layer. This node require the following attribute:
  - **name**, ***string, required field***, name of this layer. The value will be used in **<layer_layout>** to construct the fully connected neural network.

In addition, this node also accepts the following subnodes

- **<activation>**, ***string, optional field***, including 'relu', 'tanh', 'elu', 'selu', 'softplus', 'softsign', 'sigmoid', 'hard_sigmoid', 'linear', 'softmax'. (see 15.3.12.1)
  Default: linear
- **<dim_out>**, ***positive integer, required except if this layer is used as the last output layer***, dimensionality of the output space of this layer
- **<use_bias>**, ***boolean, optional field***, whether the layer uses a bias vector.
  Default: True
- **<kernel_initializer>**, ***string, optional field***, initializer for the kernel weights matrix (see 15.3.12.2).
  Default: glorot_uniform
- **<bias_initializer>**, ***string, optional field***, intializer for the bias vector (see 15.3.12.2).
  Default: zeros
- **<kernel_regularizer>**, ***string, optional field***, regularizer function applied to the kernel weights matrix (see 15.3.12.3).
  Default: None
- **<bias_regularizer>**, ***string, optional field***, regularizer function applied to the bias vector (see 15.3.12.3).
  Default: None
- **<activity_regularizer>**, ***string, optional field***, regularizer function applied to the output of the layer (its "activation"). (see 15.3.12.3)
  Default: None
- **<kernel_constraint>**, ***string, optional field***, constraint function applied to the kernel weights matrix (see 15.3.12.4).
  Default: None
- **<bias_constraint>**, ***string, optional field***, constraint function applied to the bias vector (see 15.3.12.4)
  Default: None

• **<Dropout>**, ***optional field***, applies Dropout to the input. Dropout consists in randomly setting a fraction **<rate>** of input units to 0 at each update during training time, which helps prevent overfitting. This node require the following attribute:

- **name**, ***string, required field***, name of this layer. The value will be used in **<layer_layout>** to construct the fully connected neural network.

In addition, this node also accepts the following subnode

- **<rate>**, ***float between 0 and 1, optional field***, fraction of the input units to drop.
  Default: 0

- **<noise_shape>**, ***list of integers, optional field***, 1D integer tensor representing the shape of the binary dropout mask that will be multiplied with the input.
  Default: None
- **<seed>**, ***integer, optional field***, a integer to use as random seed.
  Default: None

- **<Conv1D>**, ***optional field***, This node require the following attribute:
  - **name**, ***string, required field***, name of this layer. The value will be used in **<layer_layout>** to construct the fully connected neural network.

  In addition, this node also accepts the following subnodes
  - **<activation>**, ***string, optional field***, including 'relu', 'tanh', 'elu', 'selu', 'softplus', 'softsign', 'sigmoid', 'hard_sigmoid', 'linear', 'softmax'. (see 15.3.12.1)
    Default: linear
  - **<dim_out>**, ***positive integer, required except if this layer is used as the last output layer***, dimensionality of the output space of this layer
  - **<use_bias>**, ***boolean, optional field***, whether the layer uses a bias vector.
    Default: True
  - **<kernel_size>**, ***integer or list of integers, required field***, specifying the length of the 1D convolution window.
  - **<strides>**, ***integer or list of integers, optional field***, pecifying the stride length of the convolution. Specifying any stride value not equal 1 is incompatible with specifying any dilation_rate value not equal 1.
    Default: 1
  - **<padding>**, ***string, optional field***, one of "valid", "causal" or "same" (case-insensitive). "valid" means "no padding". "same" results in padding the input such that the output has the same length as the original input. "causal" results in causal (dilated) convolutions, e.g. output[t] does not depend on input[t + 1:]. A zero padding is used such that the output has the same length as the original input. Useful when modeling temporal data where the model should not violate the temporal order.
    Default: valid
  - **<data_format>**, ***string, optional field***, A string, one of "channels_last" (default) or "channels_first". The ordering of the dimensions in the inputs. "channels_last" corresponds to inputs with shape (batch, steps, channels) (default format for temporal data in Keras) while "channels_first" corresponds to inputs with shape (batch, channels, steps).

    Default: channels_last
  - **<dilation_rate>**, ***integer or list of integers, optional field***, specifying the dilation rate to use for dilated convolution. Currently, specifying any dilation_rate value not

equal 1 is incompatible with specifying any strides value not equal 1.
Default: 1

- **<kernel_initializer>**, ***string, optional field***, initializer for the kernel weights matrix (see 15.3.12.2).
Default: glorot_uniform
- **<bias_initializer>**, ***string, optional field***, intializer for the bias vector (see 15.3.12.2).
Default: zeros
- **<kernel_regularizer>**, ***string, optional field***, regularizer function applied to the kernel weights matrix (see 15.3.12.3).
Default: None
- **<bias_regularizer>**, ***string, optional field***, regularizer function applied to the bias vector (see 15.3.12.3).
Default: None
- **<activity_regularizer>**, ***string, optional field***, regularizer function applied to the output of the layer (its "activation"). (see 15.3.12.3)
Default: None
- **<kernel_constraint>**, ***string, optional field***, constraint function applied to the kernel weights matrix (see 15.3.12.4).
Default: None
- **<bias_constraint>**, ***string, optional field***, constraint function applied to the bias vector (see 15.3.12.4)
Default: None

• **<Conv2D>**, ***optional field***, In addition, this node also accepts the following subnodes

- **<activation>**, ***string, optional field***, including 'relu', 'tanh', 'elu', 'selu', 'softplus', 'softsign', 'sigmoid', 'hard_sigmoid', 'linear', 'softmax'. (see 15.3.12.1)
Default: linear
- **<dim_out>**, ***positive integer, required except if this layer is used as the last output layer***, dimensionality of the output space of this layer
- **<use_bias>**, ***boolean, optional field***, whether the layer uses a bias vector.
Default: True
- **<kernel_size>**, ***integer or list of integers, required field***, specifying the length of the 1D convolution window.
- **<strides>**, ***integer or list of integers, optional field***, pecifying the stride length of the convolution. Specifying any stride value not equal 1 is incompatible with specifying any dilation_rate value not equal 1.
Default: 1

- **<padding>**, ***string, optional field***, one of "valid", "causal" or "same" (case-insensitive). "valid" means "no padding". "same" results in padding the input such that the output has the same length as the original input. "causal" results in causal (dilated) convolutions, e.g. output[t] does not depend on input[t + 1:]. A zero padding is used such that the output has the same length as the original input. Useful when modeling temporal data where the model should not violate the temporal order.
  Default: valid
- **<data_format>**, ***string, optional field***, A string, one of "channels_last" (default) or "channels_first". The ordering of the dimensions in the inputs. "channels_last" corresponds to inputs with shape (batch, steps, channels) (default format for temporal data in Keras) while "channels_first" corresponds to inputs with shape (batch, channels, steps).

  Default: channels_last
- **<dilation_rate>**, ***integer or list of integers, optional field***, specifying the dilation rate to use for dilated convolution. Currently, specifying any dilation_rate value not equal 1 is incompatible with specifying any strides value not equal 1.
  Default: 1
- **<kernel_initializer>**, ***string, optional field***, initializer for the kernel weights matrix (see 15.3.12.2).
  Default: glorot_uniform
- **<bias_initializer>**, ***string, optional field***, intializer for the bias vector (see 15.3.12.2).
  Default: zeros
- **<kernel_regularizer>**, ***string, optional field***, regularizer function applied to the kernel weights matrix (see 15.3.12.3).
  Default: None
- **<bias_regularizer>**, ***string, optional field***, regularizer function applied to the bias vector (see 15.3.12.3).
  Default: None
- **<activity_regularizer>**, ***string, optional field***, regularizer function applied to the output of the layer (its "activation"). (see 15.3.12.3)
  Default: None
- **<kernel_constraint>**, ***string, optional field***, constraint function applied to the kernel weights matrix (see 15.3.12.4).
  Default: None
- **<bias_constraint>**, ***string, optional field***, constraint function applied to the bias vector (see 15.3.12.4)
  Default: None

• **<Conv3D>**, ***optional field***, In addition, this node also accepts the following subnodes

- **<activation>**, ***string, optional field***, including 'relu', 'tanh', 'elu', 'selu', 'softplus', 'softsign', 'sigmoid', 'hard_sigmoid', 'linear', 'softmax'. (see 15.3.12.1)
  Default: linear
- **<dim_out>**, ***positive integer, required except if this layer is used as the last output layer***, dimensionality of the output space of this layer
- **<use_bias>**, ***boolean, optional field***, whether the layer uses a bias vector.
  Default: True
- **<kernel_size>**, ***integer or list of integers, required field***, specifying the length of the 1D convolution window.
- **<strides>**, ***integer or list of integers, optional field***, pecifying the stride length of the convolution. Specifying any stride value not equal 1 is incompatible with specifying any dilation_rate value not equal 1.
  Default: 1
- **<padding>**, ***string, optional field***, one of "valid", "causal" or "same" (case-insensitive). "valid" means "no padding". "same" results in padding the input such that the output has the same length as the original input. "causal" results in causal (dilated) convolutions, e.g. output[t] does not depend on input[t + 1:]. A zero padding is used such that the output has the same length as the original input. Useful when modeling temporal data where the model should not violate the temporal order.
  Default: valid
- **<data_format>**, ***string, optional field***, A string, one of "channels_last" (default) or "channels_first". The ordering of the dimensions in the inputs. "channels_last" corresponds to inputs with shape (batch, steps, channels) (default format for temporal data in Keras) while "channels_first" corresponds to inputs with shape (batch, channels, steps).

  Default: channels_last
- **<dilation_rate>**, ***integer or list of integers, optional field***, specifying the dilation rate to use for dilated convolution. Currently, specifying any dilation_rate value not equal 1 is incompatible with specifying any strides value not equal 1.
  Default: 1
- **<kernel_initializer>**, ***string, optional field***, initializer for the kernel weights matrix (see 15.3.12.2).
  Default: glorot_uniform
- **<bias_initializer>**, ***string, optional field***, intializer for the bias vector (see 15.3.12.2).
  Default: zeros
- **<kernel_regularizer>**, ***string, optional field***, regularizer function applied to the kernel weights matrix (see 15.3.12.3).
  Default: None

- **<bias_regularizer>**, ***string, optional field***, regularizer function applied to the bias vector (see 15.3.12.3).
  Default: None
- **<activity_regularizer>**, ***string, optional field***, regularizer function applied to the output of the layer (its "activation"). (see 15.3.12.3)
  Default: None
- **<kernel_constraint>**, ***string, optional field***, constraint function applied to the kernel weights matrix (see 15.3.12.4).
  Default: None
- **<bias_constraint>**, ***string, optional field***, constraint function applied to the bias vector (see 15.3.12.4)
  Default: None

• **<Flatten>**, ***optional field***, In addition, this node also accepts the following subnodes

- **<data_format>**, ***string, optional field***, A string, one of "channels_last" (default) or "channels_first". The ordering of the dimensions in the inputs. "channels_last" corresponds to inputs with shape (batch, steps, channels) (default format for temporal data in Keras) while "channels_first" corresponds to inputs with shape (batch, channels, steps).

  Default: channels_last

• **<MaxPooling1D>**, ***optional field***, In addition, this node also accepts the following subnodes

- **<pool_size>**, ***integer, required field***, size of the max pooling windows.
  Default: 2
- **<strides>**, ***integer or list of integers, optional field***, pecifying the stride length of the convolution. Specifying any stride value not equal 1 is incompatible with specifying any dilation_rate value not equal 1.
  Default: 1
- **<padding>**, ***string, optional field***, one of "valid", "causal" or "same" (case-insensitive). "valid" means "no padding". "same" results in padding the input such that the output has the same length as the original input. "causal" results in causal (dilated) convolutions, e.g. output[t] does not depend on input[t + 1:]. A zero padding is used such that the output has the same length as the original input. Useful when modeling temporal data where the model should not violate the temporal order.
  Default: valid
- **<data_format>**, ***string, optional field***, A string, one of "channels_last" (default) or "channels_first". The ordering of the dimensions in the inputs. "channels_last" corresponds to inputs with shape (batch, steps, channels) (default format for temporal data in Keras) while "channels_first" corresponds to inputs with shape (batch, channels, steps).

Default: channels_last

- **<MaxPooling2D>**, ***optional field***, In addition, this node also accepts the following subnodes
  - **<pool_size>**, ***integer, required field***, size of the max pooling windows.
    Default: 2
  - **<strides>**, ***integer or list of integers, optional field***, pecifying the stride length of the convolution. Specifying any stride value not equal 1 is incompatible with specifying any dilation_rate value not equal 1.
    Default: 1
  - **<padding>**, ***string, optional field***, one of "valid", "causal" or "same" (case-insensitive). "valid" means "no padding". "same" results in padding the input such that the output has the same length as the original input. "causal" results in causal (dilated) convolutions, e.g. output[t] does not depend on input[t + 1:]. A zero padding is used such that the output has the same length as the original input. Useful when modeling temporal data where the model should not violate the temporal order.
    Default: valid
  - **<data_format>**, ***string, optional field***, A string, one of "channels_last" (default) or "channels_first". The ordering of the dimensions in the inputs. "channels_last" corresponds to inputs with shape (batch, steps, channels) (default format for temporal data in Keras) while "channels_first" corresponds to inputs with shape (batch, channels, steps).

    Default: channels_last

- **<MaxPooling3D>**, ***optional field***, In addition, this node also accepts the following subnodes
  - **<pool_size>**, ***integer, required field***, size of the max pooling windows.
    Default: 2
  - **<strides>**, ***integer or list of integers, optional field***, pecifying the stride length of the convolution. Specifying any stride value not equal 1 is incompatible with specifying any dilation_rate value not equal 1.
    Default: 1
  - **<padding>**, ***string, optional field***, one of "valid", "causal" or "same" (case-insensitive). "valid" means "no padding". "same" results in padding the input such that the output has the same length as the original input. "causal" results in causal (dilated) convolutions, e.g. output[t] does not depend on input[t + 1:]. A zero padding is used such that the output has the same length as the original input. Useful when modeling temporal data where the model should not violate the temporal order.
    Default: valid

- **<data_format>**, ***string, optional field***, A string, one of "channels_last" (default) or "channels_first". The ordering of the dimensions in the inputs. "channels_last" corresponds to inputs with shape (batch, steps, channels) (default format for temporal data in Keras) while "channels_first" corresponds to inputs with shape (batch, channels, steps).

  Default: channels_last

- **<AveragePooling1D>**, ***optional field***, In addition, this node also accepts the following subnodes

  - **<pool_size>**, ***integer, required field***, size of the max pooling windows.
    Default: 2
  - **<strides>**, ***integer or list of integers, optional field***, pecifying the stride length of the convolution. Specifying any stride value not equal 1 is incompatible with specifying any dilation_rate value not equal 1.
    Default: 1
  - **<padding>**, ***string, optional field***, one of "valid", "causal" or "same" (case-insensitive). "valid" means "no padding". "same" results in padding the input such that the output has the same length as the original input. "causal" results in causal (dilated) convolutions, e.g. output[t] does not depend on input[t + 1:]. A zero padding is used such that the output has the same length as the original input. Useful when modeling temporal data where the model should not violate the temporal order.
    Default: valid
  - **<data_format>**, ***string, optional field***, A string, one of "channels_last" (default) or "channels_first". The ordering of the dimensions in the inputs. "channels_last" corresponds to inputs with shape (batch, steps, channels) (default format for temporal data in Keras) while "channels_first" corresponds to inputs with shape (batch, channels, steps).

    Default: channels_last

- **<AveragePooling2D>**, ***optional field***, In addition, this node also accepts the following subnodes

  - **<pool_size>**, ***integer, required field***, size of the max pooling windows.
    Default: 2
  - **<strides>**, ***integer or list of integers, optional field***, pecifying the stride length of the convolution. Specifying any stride value not equal 1 is incompatible with specifying any dilation_rate value not equal 1.
    Default: 1
  - **<padding>**, ***string, optional field***, one of "valid", "causal" or "same" (case-insensitive). "valid" means "no padding". "same" results in padding the input such that the output has the same length as the original input. "causal" results in causal (dilated) convolutions, e.g. output[t] does not depend on input[t + 1:]. A zero padding is used such that

the output has the same length as the original input. Useful when modeling temporal data where the model should not violate the temporal order.
Default: valid

- **<data_format>**, ***string, optional field***, A string, one of "channels_last" (default) or "channels_first". The ordering of the dimensions in the inputs. "channels_last" corresponds to inputs with shape (batch, steps, channels) (default format for temporal data in Keras) while "channels_first" corresponds to inputs with shape (batch, channels, steps).

  Default: channels_last

- **<AveragePooling3D>**, ***optional field***, In addition, this node also accepts the following subnodes

  - **<pool_size>**, ***integer, required field***, size of the max pooling windows.
    Default: 2
  - **<strides>**, ***integer or list of integers, optional field***, pecifying the stride length of the convolution. Specifying any stride value not equal 1 is incompatible with specifying any dilation_rate value not equal 1.
    Default: 1
  - **<padding>**, ***string, optional field***, one of "valid", "causal" or "same" (case-insensitive). "valid" means "no padding". "same" results in padding the input such that the output has the same length as the original input. "causal" results in causal (dilated) convolutions, e.g. output[t] does not depend on input[t + 1:]. A zero padding is used such that the output has the same length as the original input. Useful when modeling temporal data where the model should not violate the temporal order.
    Default: valid
  - **<data_format>**, ***string, optional field***, A string, one of "channels_last" (default) or "channels_first". The ordering of the dimensions in the inputs. "channels_last" corresponds to inputs with shape (batch, steps, channels) (default format for temporal data in Keras) while "channels_first" corresponds to inputs with shape (batch, channels, steps).

    Default: channels_last

- **<GlobalMaxPooling1D>**, ***optional field***, In addition, this node also accepts the following subnodes

  - **<data_format>**, ***string, optional field***, A string, one of "channels_last" (default) or "channels_first". The ordering of the dimensions in the inputs. "channels_last" corresponds to inputs with shape (batch, steps, channels) (default format for temporal data in Keras) while "channels_first" corresponds to inputs with shape (batch, channels, steps).

    Default: channels_last

- **<GlobalMaxPooling2D>**, ***optional field***, In addition, this node also accepts the following subnodes
  - **<data_format>**, ***string, optional field***, A string, one of "channels_last" (default) or "channels_first". The ordering of the dimensions in the inputs. "channels_last" corresponds to inputs with shape (batch, steps, channels) (default format for temporal data in Keras) while "channels_first" corresponds to inputs with shape (batch, channels, steps).

    Default: channels_last
- **<GlobalMaxPooling3D>**, ***optional field***, In addition, this node also accepts the following subnodes
  - **<data_format>**, ***string, optional field***, A string, one of "channels_last" (default) or "channels_first". The ordering of the dimensions in the inputs. "channels_last" corresponds to inputs with shape (batch, steps, channels) (default format for temporal data in Keras) while "channels_first" corresponds to inputs with shape (batch, channels, steps).

    Default: channels_last
- **<GlobalAveragePooling1D>**, ***optional field***, In addition, this node also accepts the following subnodes
  - **<data_format>**, ***string, optional field***, A string, one of "channels_last" (default) or "channels_first". The ordering of the dimensions in the inputs. "channels_last" corresponds to inputs with shape (batch, steps, channels) (default format for temporal data in Keras) while "channels_first" corresponds to inputs with shape (batch, channels, steps).

    Default: channels_last
- **<GlobalAveragePooling2D>**, ***optional field***, In addition, this node also accepts the following subnodes
  - **<data_format>**, ***string, optional field***, A string, one of "channels_last" (default) or "channels_first". The ordering of the dimensions in the inputs. "channels_last" corresponds to inputs with shape (batch, steps, channels) (default format for temporal data in Keras) while "channels_first" corresponds to inputs with shape (batch, channels, steps).

    Default: channels_last
- **<GlobalAveragePooling3D>**, ***optional field***, In addition, this node also accepts the following subnodes
  - **<data_format>**, ***string, optional field***, A string, one of "channels_last" (default) or "channels_first". The ordering of the dimensions in the inputs. "channels_last" corresponds to inputs with shape (batch, steps, channels) (default format for temporal data in

Keras) while ”channels_first” corresponds to inputs with shape (batch, channels, steps).

Default: channels_last

Example:

```
<Simulation>
  ...
  <Models>
    ...
    <ROM name='aUserDefinedName'
       subType='KerasConvNetClassifier'>
      <Features>x1,x2</Features>
      <Target>labels</Target>
      <loss>mean_squared_error</loss>
      <metrics>accuracy</metrics>
      <batch_size>1</batch_size>
      <epochs>2</epochs>
      <plot_model>True</plot_model>
      <validation_split>0.25</validation_split>
      <num_classes>1</num_classes>
      <optimizerSetting>
        <beta_1>0.9</beta_1>
        <optimizer>Adam</optimizer>
        <beta_2>0.999</beta_2>
        <epsilon>1e-8</epsilon>
        <decay>0.0</decay>
        <lr>0.001</lr>
      </optimizerSetting>
      <Conv1D name="firstConv1D">
          <activation>relu</activation>
          <strides>1</strides>
          <kernel_size>2</kernel_size>
          <padding>valid</padding>
          <dim_out>32</dim_out>
      </Conv1D>
      <MaxPooling1D name="pooling1">
          <strides>2</strides>
          <pool_size>2</pool_size>
      </MaxPooling1D>
      <Conv1D name="SecondConv1D">
          <activation>relu</activation>
          <strides>1</strides>
```

```
            <kernel_size>2</kernel_size>
            <padding>valid</padding>
            <dim_out>32</dim_out>
        </Conv1D>
        <MaxPooling1D name="pooling2">
            <strides>2</strides>
            <pool_size>2</pool_size>
        </MaxPooling1D>
        <Flatten name="flatten">
        </Flatten>
        <Dense name="dense1">
            <activation>relu</activation>
            <dim_out>10</dim_out>
        </Dense>
        <Dropout name="dropout1">
            <rate>0.25</rate>
        </Dropout>
        <Dropout name="dropout2">
            <rate>0.25</rate>
        </Dropout>
        <Dense name="dense2">
            <activation>softmax</activation>
        </Dense>
        <layer_layout>firstConv1D, pooling1, dropout1,
           SecondConv1D, pooling2, dropout2, flatten, dense1,
           dense2</layer_layout>
    </ROM>
    ...
  </Models>
  ...
</Simulation>
```

15.3.12.7 KerasLSTMClassifier

Long Short Term Memory networks (LSTM) are a special kind of recurrent neural network, capable of learning long-term dependencies. They work tremendously well on a large variety of problems, and are now widely used. LSTMs are explicity designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something that they struggle to learn.

It is important to NOTE that RAVEN uses a Z-score normalization of the training data before constructing the *KerasLSTMClassifier* ROM:

$$\mathbf{X}' = \frac{(\mathbf{X} - \mu)}{\sigma} \tag{31}$$

In order to use this ROM, the **<ROM>** attribute **subType** needs to be **'KerasLSTMClassifier'** (see the example below). This model can be initialized with the following layers:

- **<Dense>**, ***required field***, regular densely-connected neural network layer. This node require the following attribute:
  - **name**, ***string, required field***, name of this layer. The value will be used in **<layer_layout>** to construct the fully connected neural network.

  In addition, this node also accepts the following subnodes
  - **<activation>**, ***string, optional field***, including 'relu', 'tanh', 'elu', 'selu', 'softplus', 'softsign', 'sigmoid', 'hard_sigmoid', 'linear', 'softmax'. (see 15.3.12.1)
    Default: linear
  - **<dim_out>**, ***positive integer, required except if this layer is used as the last output layer***, dimensionality of the output space of this layer
  - **<use_bias>**, ***boolean, optional field***, whether the layer uses a bias vector.
    Default: True
  - **<kernel_initializer>**, ***string, optional field***, initializer for the kernel weights matrix (see 15.3.12.2).
    Default: glorot_uniform
  - **<bias_initializer>**, ***string, optional field***, intializer for the bias vector (see 15.3.12.2).
    Default: zeros
  - **<kernel_regularizer>**, ***string, optional field***, regularizer function applied to the kernel weights matrix (see 15.3.12.3).
    Default: None
  - **<bias_regularizer>**, ***string, optional field***, regularizer function applied to the bias vector (see 15.3.12.3).
    Default: None
  - **<activity_regularizer>**, ***string, optional field***, regularizer function applied to the output of the layer (its "activation"). (see 15.3.12.3)
    Default: None

- **<kernel_constraint>**, ***string, optional field***, constraint function applied to the kernel weights matrix (see 15.3.12.4).
  Default: None
- **<bias_constraint>**, ***string, optional field***, constraint function applied to the bias vector (see 15.3.12.4)
  Default: None

- **<Dropout>**, ***optional field***, applies Dropout to the input. Dropout consists in randomly setting a fraction **<rate>** of input units to 0 at each update during training time, which helps prevent overfitting. This node require the following attribute:

  - **name**, ***string, required field***, name of this layer. The value will be used in **<layer_layout>** to construct the fully connected neural network.

  In addition, this node also accepts the following subnode

  - **<rate>**, ***float between 0 and 1, optional field***, fraction of the input units to drop.
    Default: 0
  - **<noise_shape>**, ***list of integers, optional field***, 1D integer tensor representing the shape of the binary dropout mask that will be multiplied with the input.
    Default: None
  - **<seed>**, ***integer, optional field***, a integer to use as random seed.
    Default: None

- **<LSTM>**, ***required field***, long short-term memory layer. This node require the following attribute:

  - **name**, ***string, required field***, name of this layer. The value will be used in **<layer_layout>** to construct the fully connected neural network.

  In addition, this node also accepts the following subnodes

  - **<activation>**, ***string, optional field***, including 'relu', 'tanh', 'elu', 'selu', 'softplus', 'softsign', 'sigmoid', 'hard_sigmoid', 'linear', 'softmax'. (see 15.3.12.1)
    Default: linear
  - **<dim_out>**, ***positive integer, required except if this layer is used as the last output layer***, dimensionality of the output space of this layer
  - **<recurrent_activation>**, ***string, optional field***, activation function to use for the recurrent step, including 'relu', 'tanh', 'elu', 'selu', 'softplus', 'softsign', 'sigmoid', 'hard_sigmoid', 'linear', 'softmax'.
    Default: hard_sigmoid
  - **<dropout>**, ***float between 0 and 1, optional field***, fraction of the units to drop for the linear transformation of the inputs
    Default: 0

- **<recurrent_dropout>**, ***float between 0 and 1, optional field***, fraction of the units to drop for the linear transformation of the recurrent state.
  Default: 0
- **<return_sequence>**, ***boolean, optional field***, whether to return the last output in the output sequence, or full sequence.
  Default: False
- **<use_bias>**, ***boolean, optional field***, whether the layer uses a bias vector.
  Default: True
- **<kernel_initializer>**, ***string, optional field***, initializer for the kernel weights matrix (see 15.3.12.2).
  Default: glorot_uniform
- **<recurrent_initializer>**, ***string, optional field***, used for the linear transformation of the recurrent state (see 15.3.12.2).
  Default: orthogonal
- **<bias_initializer>**, ***string, optional field***, intializer for the bias vector (see 15.3.12.2).
  Default: zeros
- **<unit_forget_bias>**, ***boolean, optional field***, add 1 to the bias of the forget gate at initialization if True.
  Default: True
- **<kernel_regularizer>**, ***string, optional field***, regularizer function applied to the kernel weights matrix (see 15.3.12.3).
  Default: None
- **<recurrent_regularizer>**, ***string, optional field***, regularizer function applied to the *recurrent_kernel* weights matrix (see 15.3.12.3).
  Default: None
- **<bias_regularizer>**, ***string, optional field***, regularizer function applied to the bias vector (see 15.3.12.3).
  Default: None
- **<activity_regularizer>**, ***string, optional field***, regularizer function applied to the output of the layer (its "activation"). (see 15.3.12.3)
  Default: None
- **<kernel_constraint>**, ***string, optional field***, constraint function applied to the kernel weights matrix (see 15.3.12.4).
  Default: None
- **<recurrent_constraint>**, ***string, optional field***, constraint function applied to the *recurrent_kernel* weights matrix (see **??**).
  Default: None

- **<bias_constraint>**, ***string, optional field***, constraint function applied to the bias vector (see 15.3.12.4)
  Default: None
- **<implementation>**, ***integer, optional field***, implementation mode, either 1 or 2. Mode 1 will structure its operations as a larger number of smaller dot products and additions, whereas mode 2 will batch them into fewer, larger operations. These modes will have different performance profiles on different hardware and for different applications.
  Default: 1
- **<return_state>**, ***boolean, optional field***, whether to return the last output in the output sequence, or the full sequence.
  Default: False
- **<go_backwards>**, ***boolean, optional field***, if True, process the input sequence backwards and return the reversed sequence.
  Default: False
- **<stateful>**, ***boolean, optional field***, if True, the last state for each sample at index i in a batch will be used as initial state for the sample of index i in the following batch.
  Default: False
- **<unroll>**, ***boolean, optional field***, if True, the network will be unrolled, else a symbolic loop will be used. Unrolling can speed-up a RNN, although it tends to be more memory-intensive. Unrolling is only suitable for short sequences.
  Default: False

Example:

```
<Simulation>
  ...
  <Models>
    ...
    <ROM name='aUserDefinedName' subType='KerasLSTMClassifier'>
      <Features>x</Features>
      <Target>y</Target>
      <loss>categorical_crossentropy</loss>
      <metrics>accuracy</metrics>
      <batch_size>1</batch_size>
      <epochs>10</epochs>
      <validation_split>0.25</validation_split>
      <num_classes>26</num_classes>
      <optimizerSetting>
        <beta_1>0.9</beta_1>
        <optimizer>Adam</optimizer>
```

```
        <beta_2>0.999</beta_2>
        <epsilon>1e-8</epsilon>
        <decay>0.0</decay>
        <lr>0.001</lr>
      </optimizerSetting>
      <LSTM name="lstm1">
          <activation>tanh</activation>
          <dim_out>32</dim_out>
      </LSTM>
      <LSTM name="lstm2">
          <activation>tanh</activation>
          <dim_out>16</dim_out>
      </LSTM>
      <Dropout name="dropout">
          <rate>0.25</rate>
      </Dropout>
      <Dense name="dense">
          <activation>softmax</activation>
      </Dense>
      <layer_layout>lstm1,lstm2,dropout,dense</layer_layout>
    </ROM>
    ...
  </Models>
  ...
</Simulation>
```

15.4 External Model

As the name suggests, an external model is an entity that is embedded in the RAVEN code at run time. This object allows the user to create a python module that is going to be treated as a predefined internal model object. In other words, the **External Model** is going to be treated by RAVEN as a normal external Code (e.g. it is going to be called in order to compute an arbitrary quantity based on arbitrary input).

The specifications of an External Model must be defined within the XML block **`<ExternalModel>`**. This XML node needs to contain the attributes:

- **`name`**, ***required string attribute***, user-defined name of this External Model. **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***, external model type. It must be kept empty, except in the following cases:
  - ***pluginType***, in this case the `subType` is the spefic plugin to use (e.g. **TEAL.CashFlow**)
  - ***pickledModel***, a pickled external model (serialized model). See 15.4.2 for details.
- `ModuleToLoad`, ***required string attribute***, file name with its absolute or relative path. This attribute is specified just for Generic ExternalModel (i.e. empty `subType`) **Note:** If a relative path is specified, the code first checks relative to the working directory, then it checks with respect to where the user runs the code. Using the relative path with respect to where the code is run is not recommended.

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

- `<variables>`, ***string, optional 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 (`<variables>`) is deprecated and will be removed in RAVEN 3.0 in favor of the two following nodes (`<inputs>`, `<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 inputted:

- `<alias>` ***string, optional field*** specifies alias for any variable of interest in the input or output space for the ExternalModel. These aliases can be used anywhere in the RAVEN input to refer to the ExternalModel variables. 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.
  Default: None

When the external function 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 module (each method implemented) as a python "self."

15.4.1 Generic External Model

As mentioned before, the generic external model is a python module that is going to be treated as a predefined internal model object.

In the External Python module (if not a plugin or ***pickledModel***), the user can implement all the methods that are needed for the functionality of the model, but only the following methods, if present, are called by the framework:

- **`def _readMoreXML`**, ***OPTIONAL METHOD***, can be implemented by the user if the XML input that belongs to this External Model needs to be extended to contain other information. The information read needs to be stored in "self" in order to be available to all the other methods (e.g. if the user needs to add a couple of newer XML nodes with information needed by the algorithm implemented in the "run" method).
- **`def initialize`**, ***OPTIONAL METHOD***, can implement all the actions need to be performed at the initialization stage.
- **`def createNewInput`**, ***OPTIONAL METHOD***, creates a new input with the information coming from the RAVEN framework. In this function the user can retrieve the information coming from the RAVEN framework, during the employment of a calculation flow, and use them to construct a new input that is going to be transferred to the "run" method.
- **`def run`**, ***REQUIRED METHOD***, is the actual location where the user needs to implement the model action (e.g. resolution of a set of equations, etc.). This function is going to receive the Input (or Inputs) generated either by the External Model "createNewInput" method or the internal RAVEN one.

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

15.4.1.1 Method: `def _readMoreXML`

As already mentioned, the **readMoreXML** method can be implemented by the user if the XML input that belongs to this External Model needs to be extended to contain other information. The information read needs to be stored in "self" in order to be available to all the other methods (e.g. if the user needs to add a couple of newer XML nodes with information needed by the algorithm implemented in the "run" method). If this method is implemented in the **External Model**, RAVEN is going to call it when the node **`<ExternalModel>`** is found parsing 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:

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

Corresponding Python function:

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

15.4.1.2 `def initialize`

The **initialize** method can be implemented in the **External Model** in order to initialize some variables needed by it. For example, it can be used to compute a quantity needed by the "run" method before performing the actual calculation). If this method is implemented in the **External Model**, RAVEN is going to call it at the initialization stage of each "Step" (see section 18. RAVEN will

communicate, thorough a set of method attributes, all the information that are generally needed to perform a initialization:

- runInfo, a dictionary containing information regarding how the calculation is set up (e.g. number of processors, etc.). It contains the following attributes:
  - `DefaultInputFile` – default input file to use
  - `SimulationFiles` – the xml input file
  - `ScriptDir` – the location of the pbs script interfaces
  - `FrameworkDir` – the directory where the framework is located
  - `WorkingDir` – the directory where the framework should be running
  - `TempWorkingDir` – the temporary directory where a simulation step is run
  - `NumMPI` – the number of mpi process by run
  - `NumThreads` – number of threads by run
  - `numProcByRun` – total number of core used by one run (number of threads by number of mpi)
  - `batchSize` – number of contemporaneous runs
  - `ParallelCommand` – the command that should be used to submit jobs in parallel (mpi)
  - `numNode` – number of nodes
  - `procByNode` – number of processors by node
  - `totalNumCoresUsed` – total number of cores used by driver
  - `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.

In the following an example is reported:

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

15.4.1.3 Method: `def createNewInput`

The **createNewInput** method can be implemented by the user to create a new input with the information coming from the RAVEN framework. In this function, the user can retrieve the information coming from the RAVEN framework, during the employment of a calculation flow, and use them to construct a new input that is going to be transferred to the "run" method. The new input created needs to be returned to RAVEN (i.e. "return NewInput").
This method expects that the new input is returned in a Python "dictionary". RAVEN communicates, thorough 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 12), the internal "createNewInput" method is going to convert it in a dictionary of values.

Here we present an example:

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

```
    avariable = inputs['something']/inputs['something2']
  return avariable*Kwargs['SampledVars']['aSampledVar']
```

15.4.1.4 Method: `def run`

As stated previously, the only method that *must* be present in an External Module is the **run** function. In this function, the user needs to implement the algorithm that RAVEN will execute. The **run** method is generally called after having inquired the "createNewInput" method (either the internal or the user-implemented one). The only attribute this method is going to receive is a Python list of inputs (the inputs coming from the `createNewInput` method). If the user wants RAVEN to collect the results of this method, the outcomes of interest need to be stored in "self." **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,Input):
  # in here the actual run of the
  # model is implemented
  input = Input[0]
  self.outcome = self.sigma*self.rho*input[``whatEver'']
```

15.4.2 pickledModel

It is not uncommon for a Model to be created and encapsulate in one RAVEN run, then serialized to file (pickled), then loaded into another RAVEN run to be used as a model. When this is the case, a **<ExternalModel>** with subtype 'pickledModel' is used to hold the place of the ExternalModel that will be loaded from file. The notation for this ExternalModel is much less than a typical ExternalModel; it only requires a name and its subtype. Note that when loading ExternalModels from file, RAVEN will not perform any checks on the expected inputs or outputs of an ExternalModel; it is expected that a user know at least the I/O of a ExternalModel before trying to use it as a model. Initially, a pickledModel is not usable. It cannot be sampled; attempting to do so will raise an error. An **<IOStep>** is used to load the ExternalModel from file, at which point the ExternalModel will have all the same characteristics as when it was pickled in a previous RAVEN run. Example: For this example the ExternalModel has already been created in another RAVEN run, then serialized 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="pickle.pk" type="">pickle.pk</Input>
  </Files>
  ...
  <Models>
    ...
    <ExternalModel name="myModel" subType="pickledModel"/>
    ...
  </Models>
  ...
  <Steps>
    ...
    <IOStep name="loadModel">
      <Input class="Files" type="">pickle.pk</Input>
      <Output class="Models" type="pickledModel">myModel</Output>
    </IOStep>
    ...
  </Steps>
  ...
</Simulation>
```

15.5 PostProcessor

A Post-Processor (PP) can be considered as an action performed on a set of data or other type of objects. Most of the post-processors contained in RAVEN, employ a mathematical operation on the data given as "input". RAVEN supports several different types of PPs.

Currently, the following types are available in RAVEN:

- **BasicStatistics**
- **ComparisonStatistics**
- **ImportanceRank**
- **SafestPoint**
- **LimitSurface**
- **LimitSurfaceIntegral**
- **External**
- **TopologicalDecomposition**
- **DataMining**
- **RiskMeasureDiscrete**
- **Metric**
- **CrossValidation**

- **DataClassifier**
- **ValueDuration**
- **FastFourierTransform**
- **SampleSelector**
- **ParetoFrontier**
- **MCSImporter**
- **EconomicRatio**

The specifications of these types must be defined within the XML block `<PostProcessor>`. This XML node needs to contain the attributes:

- `name`, ***required string attribute***, user-defined identifier of this post-processor. **Note:** As with other objects, this is the name that can be used to refer to this specific entity from other input XML blocks.
- `subType`, ***required string attribute***, defines which of the post-processors needs to be used, choosing among the previously reported types. This choice conditions the subsequent required and/or optional `<PostProcessor>` sub nodes.

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

15.5.1 BasicStatistics

The **BasicStatistics** post-processor is the container of the algorithms to compute many of the most important statistical quantities. It is important to notice that this post-processor can accept as input both ***PointSet*** and ***HistorySet*** data objects, depending on the type of statistics the user wants to compute:

- ***PointSet***: Static Statistics;
- ***HistorySet***: Dynamic Statistics. Depending on a "pivot parameter" (e.g. time) the post-processor is going to compute the statistics for each value of it (e.g. for each time step). In case an **HistorySet** is provided as Input, the Histories needs to be synchronized (use ***Interfaced*** post-processor of type **HistorySetSync**).

In order to use the *BasicStatistics post-processor* PP, the user needs to set the `subType` of a `<PostProcessor>` node:

```
<PostProcessor name='ppName' subType='BasicStatistics'/>.
```

Several sub-nodes are available:

- **<"metric">**, ***comma separated string or node list, required field***, specifications for the metric to be calculated. The name of each node is the requested metric. There are two forms for specifying the requested parameters of the metric. For scalar values such as **<expectedValue>** and **<variance>**, the text of the node is a comma-separated list of the parameters for which the metric should be calculated. For matrix values such as **<sensitivty>** and **<covariance>**, the matrix node requires two sub-nodes, **<targets>** and **<features>**, each of which is a comma-separated list of the targets for which the metric should be calculated, and the features for which the metric should be calculated for that target. See the example below.

  Note: When defining the metrics to use, it is possible to have multiple nodes with the same name. For example, if a problem has inputs W, X, Y, and Z, and the responses are A, B, and C, it is possible that the desired metrics are the **<sensitivity>** of A and B to X and Y, as well as the **<sensitivity>** of C to W and Z, but not the sensitivity of A to W. In this event, two copies of the **<sensitivity>** node are added to the input. The first has targets A, B and features X, Y, while the second node has target C and features W, Z. This could reduce some computation effort in problems with many responses or inputs. An example of this is shown below.
  Currently the scalar quantities available for request are:

  - **expectedValue**: expected value or mean
  - **minimum**: The minimum value of the samples.
  - **maximum**: The maximum value of the samples.
  - **median**: The weighted median of the samples (50% weighted percentile). If probablitity weights are not assigned, uniform distribution will be assigned. The median x_k satisfying:

$$\sum_{i=1}^{k-1} w_i \leq 1/2 and \sum_{i=k+1}^{n} w_i \leq 1/2 \tag{32}$$

  - **variance**: variance
  - **sigma**: standard deviation
  - **percentile**: the percentile. If this quantity is inputted as *percentile* the 5% and 95% percentile(s) are going to be computed. Otherwise the user can specify this quantity with a parameter *percent='X'*, where the *X* represents the requested percentile (a floating point value between 0.0 and 100.0)
  - **variationCoefficient**: coefficient of variation, i.e. **sigma/expectedValue**. **Note:** If the **expectedValue** is zero, the **variationCoefficient** will be **INF**.
  - **skewness**: skewness
  - **kurtosis**: excess kurtosis (also known as Fisher's kurtosis)
  - **samples**: the number of samples in the data set used to determine the statistics.

The matrix quantities available for request are:

- **sensitivity**: matrix of sensitivity coefficients, computed via linear regression method. (**Note:** The condition number is computed every time this quantity is requsted. If it results to be greater then 30, a multicollinearity problem exists and the sensitivity coefficients might be incorrect and a Warning is spooned by the code)
- **covariance**: covariance matrix
- **pearson**: matrix of correlation coefficients
- **NormalizedSensitivity**: matrix of normalized sensitivity coefficients. **Note:** It is the matrix of normalized VarianceDependentSensitivity
- **VarianceDependentSensitivity**: matrix of sensitivity coefficients dependent on the variance of the variables

This XML node needs to contain the attribute:

- **`prefix`**, ***required string attribute***, user-defined prefix for the given **metric**. For scalar quantifies, RAVEN will define a variable with name defined as: "prefix" + "_" + "parameter name". For example, if we define "mean" as the prefix for **expectedValue**, and parameter "x", then variable "mean_x" will be defined by RAVEN. For matrix quantities, RAVEN will define a variable with name defined as: "prefix" + "_" + "target parameter name" + "_" + "feature parameter name". For example, if we define "sen" as the prefix for **sensitivity**, target "y" and feature "x", then variable "sen_y_x" will be defined by RAVEN. **Note:** These variable will be used by RAVEN for the internal calculations. It is also accessible by the user through **DataObjects** and **OutStreams**.

Note: If the weights are present in the system then weighted quantities are calculated automatically. In addition, if a matrix quantity is requested (e.g. Covariance matrix, etc.), only the weights in the output space are going to be used for both input and output space (the computation of the joint probability between input and output spaces is not implemented yet).
Note: Certain ROMs provide their own statistical information (e.g., those using the sparse grid collocation sampler such as: **`'GaussPolynomialRom'`** and **`'HDMRRom'`**) which can be obtained by printing the ROM to file (xml). For these ROMs, computing the basic statistics on data generated from one of these sampler/ROM combinations may not provide the information that the user expects.
In addition, RAVEN will automatically calculate the standard errors on the following scalar quantities:

- **expectedValue**
- **median**
- **variance**
- **sigma**

- **skewness**
- **kurtosis**

RAVEN will define a variable with name defined as: "prefix for given **metric**" + "_ste_" + "parameter name" to store standard error of given **metric** with respect to given parameter. This information will be stored in the DataObjects, i.e. **PointSet** and **HistorySet**, and by default will be printed out in the "CSV" output files by the **OutStreams**. Option node **`<what>`** can be used in the **OutStreams** to select the information that the users want to print. In the case when the users want to store all the calculations results in general **DataSets**, RAVEN will employ a variable with name defined as: "**metric**" + "_ste" to store standard error with respect to all target parameters. An additional index "target" will added in the **DataSets** with respect to these variables. All these quantities will be automatically computed and stored in the given **DataSet**, and the users do not need to specify these quantities in their RAVEN input files.

- **`<pivotParameter>`**, ***string, optional field***, name of the parameter that needs to be used for the computation of the Dynamic BasicStatistics (e.g. time). This node needs to be inputted just in case an **HistorySet** is used as Input. It represents the reference monotonic variable based on which the statistics is going to be computed (e.g. time-dependent statistical moments).
  Default: None

- **`<biased>`**, ***string (boolean), optional field***, if *True* biased quantities are going to be calculated, if *False* unbiased.
  Default: False

- **`<dataset>`**, ***boolean, optional field***, if *True* **`'DataSet'`** will be used to store the calculation results, if *False* **`'PointSet'`** or **`'HistorySet'`** will be used to store the calculation results. **Note:** The optional **`'DataSet'`** is added only to this PostProcessor, one can still use the **`'OutStreams'`** to print the variables available in the *DataSet*. The **`'"metric"'`** names are used as the variable names, i.e. variable names listed in **`<Input>`** or **`<Output>`** in the defined **`'DataSet'`**. In addition, the extra node **`<Index>`** is required, and the value for **`var`** can be found in the following:

  - scalar metrics, such as **`<expectedValue>`** and **`<variance>`**, are requested, the index variable **`'targets'`** will be required.
  - vector metrics, such as **`<covariance>`** and **`<sensitivity>`**, are requested, the index variables **`'targets'`** and **`'features'`** will be required.
  - If **`<percentile>`** is requested, an additional index variable **`'percent'`** should be added.
  - when dynamic BasicStatistics (e.g. time) is requested, the index variable **`'time'`** will be required.

Default: False

- **<multipleFeatures>**, (boolean, optional field), if **False**, this node can be used when the users want to compute sensitivities based on one target variable with respect to one feature variable, i.e. the sensitivity calculations are directly computed using the **Linear Regression** or **Best Linear Predictor** method with single feature. This method can be useful when the input features depend on each other. The default value is **True**, which means the sensitivity calculations are performed using **Linear Regression** or **Best Linear Predictor** method with multiple features. If the input features are not fully correlated, the default value for **<multipleFeatures>** is always recommanded. **Note:** this node only affects the calculations of metrics such as **<sensitivity>**, **<VarianceDependentSensitivity>** and **<NormalizedSensitivity>**.
  Default: True

Example (Static Statistics): This example demonstrates how to request the expected value of **'x01'** and **'x02'**, along with the sensitivity of both **'x01'** and **'x02'** to **'a'** and **'b'**.

```
<Simulation>
  ...
  <Models>
    ...
    <PostProcessor name='aUserDefinedName'
       subType='BasicStatistics' verbosity='debug'>
      <expectedValue prefix='mean'>x01,x02</expectedValue>
      <sensitivity prefix='sen'>
        <targets>x01,x02</targets>
        <features>a,b</features>
      </sensitivity>
    </PostProcessor>
    ...
  </Models>
  ...
</Simulation>
```

In this case, the RAVEN variables "mean_x01, mean_x02, sen_x01_a, sen_x02_a, sen_x01_b, sen_x02_b" will be created and accessible for the RAVEN entities **DataObjects** and **OutStreams**.

Example (Static, multiple matrix nodes): This example shows how multiple nodes can specify particular metrics multiple times to include different target/feature combinations. This postprocessor calculates the expected value of A, B, and C, as well as the sensitivity of both A and B to X and Y as well as the sensitivity of C to W and Z.

```
<Simulation>
```

```
  ...
  <Models>
    ...
    <PostProcessor name='aUserDefinedName'
       subType='BasicStatistics' verbosity='debug'>
      <expectedValue prefix='mean'>A,B,C</expectedValue>
      <sensitivity prefix='sen1'>
        <targets>A,B</targets>
        <features>x,y</features>
      </sensitivity>
      <sensitivity prefix='sen2'>
        <targets>C</targets>
        <features>w,z</features>
      </sensitivity>
    </PostProcessor>
    ...
  </Models>
  ...
</Simulation>
```

Example (Dynamic Statistics):

```
<Simulation>
  ...
  <Models>
    ...
    <PostProcessor name='aUserDefinedNameForDynamicPP'
       subType='BasicStatistics' verbosity='debug'>
      <expectedValue prefix='mean'>x01,x02</expectedValue>
      <sensitivity prefix='sen'>
        <targets>x01,x02</targets>
        <features>a,b</features>
      </sensitivity>
      <pivotParameter>time</pivotParameter>
    </PostProcessor>
    ...
  </Models>
  ...
  <HistorySet name='basicStatHistorySet'>
    <Output>
      mean_x01,mean_x02,
      sen_x01_a, sen_x01_b,
      sen_x02_a, sen_x02_b
```

```
    </Output>
    <options>
      <pivotParameter>time</pivotParameter>
    </options>
  </HistorySet>
</Simulation>
```

Example (Dumping the results into DataSet):

```
<Simulation>
  ...
  <Models>
    ...
    <PostProcessor name='aUserDefinedNameForDynamicPP'
       subType='BasicStatistics' verbosity='debug'>
      <dataset>True</dataset>
      <expectedValue prefix='mean'>x01,x02</expectedValue>
      <sensitivity prefix='sen'>
        <targets>x01,x02</targets>
        <features>a,b</features>
      </sensitivity>
      <pivotParameter>time</pivotParameter>
    </PostProcessor>
    ...
  </Models>
  ...
  <DataObjects>
    <DataSet name='basicStatDataSet'>
      <Output>expectedValue,sensitivity</Output>
      <Index var='time'>expectedValue,sensitivity</Index>
      <Index var='targets'>expectedValue,sensitivity</Index>
      <Index var='features'>sensitivity</Index>
    </DataSet>
  </DataObjects>
</Simulation>
```

15.5.2 ComparisonStatistics

The **ComparisonStatistics** post-processor computes statistics for comparing two different dataObjects. This is an experimental post-processor, and it will definitely change as it is further developed.

There are four nodes that are used in the post-processor.

- **<kind>**: specifies information to use for comparing the data that is provided. This takes either uniformBins which makes the bin width uniform or equalProbability which makes the number of counts in each bin equal. It can take the following attributes:
  - **numBins** which takes a number that directly specifies the number of bins
  - **binMethod** which takes a string that specifies the method used to calculate the number of bins. This can be either square-root or sturges.
- **<compare>**: specifies the data to use for comparison. This can either be a normal distribution or a dataObjects:
  - **<data>**: This will specify the data that is used. The different parts are separated by |'s.
  - **<reference>**: This specifies a reference distribution to be used. It takes distribution to use that is defined in the distributions block. A name parameter is used to tell which distribution is used.
- **<fz>**: If the text is true, then extra comparison statistics for using the f_z function are generated. These take extra time, so are not on by default.
- **<interpolation>**: This switches the interpolation used for the cdf and the pdf functions between the default of quadratic or linear.

The **ComparisonStatistics** post-processor generates a variety of data. First for each data provided, it calculates bin boundaries, and counts the numbers of data points in each bin. From the numbers in each bin, it creates a cdf function numerically, and from the cdf takes the derivative to generate a pdf. It also calculates statistics of the data such as mean and standard deviation. The post-processor can generate a CSV file only.

The post-processor uses the generated pdf and cdf function to calculate various statistics. The first is the cdf area difference which is:

$$cdf_area_difference = \int_{-\infty}^{\infty} \|CDF_a(x) - CDF_b(x)\| dx \tag{33}$$

This given an idea about how far apart the two pieces of data are, and it will have units of x.

The common area between the two pdfs is calculated. If there is perfect overlap, this will be 1.0, if there is no overlap, this will be 0.0. The formula used is:

$$pdf_common_area = \int_{-\infty}^{\infty} \min(PDF_a(x), PDF_b(x)) dx \tag{34}$$

The difference pdf between the two pdfs is calculated. This is calculated as:

$$f_Z(z) = \int_{-\infty}^{\infty} f_X(x) f_Y(x - z) dx \tag{35}$$

This produces a pdf that contains information about the difference between the two pdfs. The mean can be calculated as (and will be calculated only if fz is true):

$$\bar{z} = \int_{-\infty}^{\infty} z f_Z(z) dz \tag{36}$$

The mean can be used to get an signed difference between the pdfs, which shows how their means compare.

The variance of the difference pdf can be calculated as (and will be calculated only if fz is true):

$$var = \int_{-\infty}^{\infty} (z - \bar{z})^2 f_Z(z) dz \tag{37}$$

The sum of the difference function is calculated if fz is true, and is:

$$sum = \int_{-\infty}^{\infty} f_z(z) dz \tag{38}$$

This should be 1.0, and if it is different that points to approximations in the calculation.

Example:

```
<Simulation>
   ...
   <Models>
      ...
      <PostProcessor name="stat_stuff"
         subType="ComparisonStatistics">
      <kind binMethod='sturges'>uniformBins</kind>
      <compare>
        <data>OriData|Output|tsin_TEMPERATURE</data>
        <reference name='normal_410_2' />
      </compare>
      <compare>
        <data>OriData|Output|tsin_TEMPERATURE</data>
        <data>OriData|Output|tsout_TEMPERATURE</data>
      </compare>
      </PostProcessor>
      <PostProcessor name="stat_stuff2"
         subType="ComparisonStatistics">
```

```
        <kind numBins="6">equalProbability</kind>
        <compare>
          <data>OriData|Output|tsin_TEMPERATURE</data>
        </compare>
        <Distribution class='Distributions'
           type='Normal'>normal_410_2</Distribution>
      </PostProcessor>
      ...
   </Models>
   ...
   <Distributions>
      <Normal name='normal_410_2'>
         <mean>410.0</mean>
         <sigma>2.0</sigma>
      </Normal>
   </Distributions>
</Simulation>
```

15.5.3 ImportanceRank

The **ImportanceRank** post-processor is specifically used to compute sensitivity indices and importance indices with respect to input parameters associated with multivariate normal distributions. In addition, the user can also request the transformation matrix and the inverse transformation matrix when the PCA reduction is used. In order to use the *ImportanceRank* PP, the user needs to set the `subType` of a `<PostProcessor>` node:

```
<PostProcessor name='ppName' subType='ImportanceRank'/>.
```

Several sub-nodes are available:

- `<what>`, ***comma separated string, required field***, List of quantities to be computed. Currently the quantities available are:
  - `'SensitivityIndex'`: used to measure the impact of sensitivities on the model.
  - `'ImportanceIndex'`: used to measure the impact of sensitivities and input uncertainties on the model.
  - `'PCAIndex'`: the indices of principal component directions, used to measure the impact of principal component directions on input covariance matrix. **Note:** `'PCAIndex'` can be only requested when subnode `<latent>` is defined in `<features>`.
  - `'transformation'`: the transformation matrix used to map the latent variables to the manifest variables in the original input space.

- **'InverseTransformation'**: the inverse transformation matrix used to map the manifest variables to the latent variables in the transformed space.
- **'ManifestSensitivity'**: the sensitivity coefficients of **<target>** with respect to **<manifest>** variables defined in **<features>**.

  Note: In order to request **'transformation'** matrix or **'InverseTransformation'** matrix or **'ManifestSensitivity'**, the subnodes **<latent>** and **<manifest>** under **<features>** are required (more details can be found in the following).

Note: For each computed quantity, RAVEN will define a unique variable name so that the data can be accessible by the users through RAVEN entities **DataObjects** and **OutStreams**. These variable names are defined as follows:

- **'SensitivityIndex'**: 'sensitivityIndex' + '_' + 'targetVariableName' + '_' + 'latentFeatureVariableName'
- **'ImportanceIndex'**: 'importanceIndex' + '_' + 'targetVariableName' + '_' + 'latentFeatureVariableName'
- **'PCAIndex'**: 'pcaIndex' + '_' + 'latentFeatureVariableName'
- **'transformation'**: 'transformation' + '_' + 'manifestFeatureVariableName' + '_' + 'latentFeatureVariableName'
- **'InverseTransformation'**: 'inverseTransformation' + '_' + 'latentFeatureVariableName' + '_' + 'manifestFeatureVariableName'
- **'ManifestSensitivity'**: 'manifestSensitivity' + '_' + 'targetVariableName' + '_' + 'manifestFeatureVariableName'

If all the quantities need to be computed, the user can input in the body of **<what>** the string **'all'**. **Note:** **'all'** equivalent to **'SensitivityIndex, ImportanceIndex, PCAIndex'**.

Since the transformation and InverseTransformation matrix can be very large, they are not printed with option **'all'**. In order to request the transformation matrix (or inverse transformation matrix) from this post processor, the user need to specify **'transformation'** or **'InverseTransformation'** in **<what>**. In addition, both **<manifest>** and **<latent>** subnodes are required and should be defined in node **<features>**. For example, let $\mathbf{L}, \mathbf{P}$ represent the transformation and inverse transformation matrices, respectively. We will define vectors $\mathbf{x}$ as manifest variables and vectors $\mathbf{y}$ as latent variables. If a absolute covariance matrix is used in given distribution, the following equation will be used:

$\delta\mathbf{x} = \mathbf{L} * \mathbf{y}$

$\mathbf{y} = \mathbf{P} * \delta\mathbf{x}$

If a relative covariance matrix is used in given distribution, the following equation will be used:

$\frac{\delta\mathbf{x}}{\mu} = \mathbf{L} * \mathbf{y}$

$\mathbf{y} = \mathbf{P} * \frac{\delta \mathbf{x}}{\mu}$

where $\delta\mathbf{x}$ denotes the changes in the input vector $\mathbf{x}$, and μ denotes the mean values of the input vector $\mathbf{x}$.

- **`<features>`**, ***XML node, required parameter***, used to specify the information for the input variables. In this xml-node, the following xml sub-nodes need to be specified:
  - **`<manifest>`**,***XML node, optional parameter***, used to indicate the input variables belongs to the original input space. It can accept the following child node:
    - **`<variables>`**,***comma separated string, required field***, lists manifest variables.
    - **`<dimensions>`**, ***comma separated integer, optional field***, lists the dimensions corresponding to the manifest variables. If not provided, the dimensions are determined by the order indices of given manifest variables.
  - **`<latent>`**,***XML node, optional parameter***, used to indicate the input variables belongs to the transformed space. It can accept the following child node:
    - **`<variables>`**,***comma separated string, required field***, lists latent variables.
    - **`<dimensions>`**, ***comma separated integer, optional field***, lists the dimensions corresponding to the latent variables. If not provided, the dimensions are determined by the order indices of given latent variables.

    Note: At least one of the subnodes, i.e. **`<manifest>`** and **`<latent>`** needs to be specified.
- **`<targets>`**, ***comma separated string, required field***, lists output responses.
- **`<mvnDistribution>`**, ***string, required field***, specifies the multivariate normal distribution name. The **`<MultivariateNormal>`** node must be present. It requires two attributes:
  - **`class`**, ***required string attribute***, is the main "class" the listed object is from, the only acceptable class for this post-processor is **`'Distributions'`**;
  - **`type`**, ***required string attribute***, is the type of distributions, the only acceptable type is **`'MultivariateNormal'`**

Here is an example to show the user how to request the transformation matrix, the inverse transformation matrix, the manifest sensitivities and other quantities.

Example:

```
<Simulation>
  ...
  <Models>
    ...
```

```
    <PostProcessor name='aUserDefinedName'
       subType='ImportanceRank'>
      <what>SensitivityIndex,ImportanceIndex,Transformation,
         InverseTransformation,ManifestSensitivity</what>
      <features>
        <manifest>
          <variables>x1,x2</variables>
          <dimensions>1,2</dimensions>
        </manifest>
        <latent>
          <variables>latent1</variables>
          <dimensions>1</dimensions>
        </latent>
      </features>
      <targets>y</targets>
      <mvnDistribution>MVN</mvnDistribution>
    </PostProcessor>
    ...
  </Models>
  ...
</Simulation>
```

The calculation results can be accessible via variables "sensitivityIndex_y_latent1, importanceIndex_y_latent1, manifestSensitivity_y_x1, manifestSensitivity_y_x2, transformation_x1_latent1, transformation_x2_latent1, inverseTransformation_latnet1_x1, inverseTransformation_laent1_x2" through RAVEN entities **DataObjects** and **OutStreams**.

15.5.4 SafestPoint

The **SafestPoint** post-processor provides the coordinates of the farthest point from the limit surface that is given as an input. The safest point coordinates are expected values of the coordinates of the farthest points from the limit surface in the space of the "controllable" variables based on the probability distributions of the "non-controllable" variables.

The term "controllable" identifies those variables that are under control during the system operation, while the "non-controllable" variables are stochastic parameters affecting the system behaviour randomly.

The "SafestPoint" post-processor requires the set of points belonging to the limit surface, which must be given as an input. The probability distributions as "Assembler Objects" are required in the "Distribution" section for both "controllable" and "non-controllable" variables.

The sampling method used by the "SafestPoint" is a "value" or "CDF" grid. At present only the "equal" grid type is available.

In order to use the *Safest Point* PP, the user needs to set the `subType` of a `<PostProcessor>` node:

`<PostProcessor name='ppName' subType='SafestPoint'/>`.

Several sub-nodes are available:

- `<Distribution>`, ***Required***, represents the probability distributions of the "controllable" and "non-controllable" variables. These are **Assembler Objects**, each of these nodes must contain 2 attributes that are used to identify those within the simulation framework:
  - `class`, ***required string attribute***, is the main "class" the listed object is from.
  - `type`, ***required string attribute***, is the object identifier or sub-type.
- `<outputName>`, ***string, required field***, specifies the name of the output variable where the probability is going to be stored. **Note:** This variable name must be listed in the `<Output>` field of the Output DataObject
- `<controllable>`, ***XML node, required field***, lists the controllable variables. Each variable is associated with its name and the two items below:
  - `<distribution>` names the probability distribution associated with the controllable variable.
  - `<grid>` specifies the `type`, `steps`, and tolerance of the sampling grid.
- `<non-controllable>`, ***XML node, required field***, lists the non-controllable variables. Each variable is associated with its name and the two items below:
  - `<distribution>` names the probability distribution associated with the non-controllable variable.
  - `<grid>` specifies the `type`, `steps`, and tolerance of the sampling grid.

Example:

```
<Simulation>
  ...
    <Models>
    ...
    <PostProcessor name='SP' subType='SafestPoint'>
      <Distribution  class='Distributions'
         type='Normal'>x1_dst</Distribution>
```

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

15.5.5 LimitSurface

The **LimitSurface** post-processor is aimed to identify the transition zones that determine a change in the status of the system (Limit Surface).

In order to use the *LimitSurface* PP, the user needs to set the `subType` of a `<PostProcessor>` node:

`<PostProcessor name='ppName' subType='LimitSurface'/>`.

Several sub-nodes are available:

- `<parameters>`, ***comma separated string, required field***, lists the parameters that define the uncertain domain and from which the LS needs to be computed.

- **<tolerance>**, ***float, optional field***, sets the absolute value (in CDF) of the convergence tolerance. This value defines the coarseness of the evaluation grid.
  Default: 1.0e-4

- **<side>**, ***string, optional field***, in this node the user can specify which side of the limit surface needs to be computed. Three options are available:
  negative, Limit Surface corresponding to the goal function value of "-1";
  positive, Limit Surface corresponding to the goal function value of "1";
  both, either positive and negative Limit Surface is going to be computed.
  Default: negative

- **Assembler Objects** These objects are either required or optional depending on the functionality of the Adaptive Sampler. The objects must be listed with a rigorous syntax that, except for the xml node tag, is common among all the objects. Each of these nodes must contain 2 attributes that are used to map those within the simulation framework:

  – **class**, ***required string attribute***, is the main "class" of the listed object. For example, it can be "Models," "Functions," etc.

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

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

  – **<ROM>**, ***string, optional field***, body of this xml node must contain the name of a ROM defined in the **<Models>** block (see section 15.3).

  – **<Function>**, ***string, required field***, the body of this xml block needs to contain the name of an External Function defined within the **<Functions>** main block (see section 16). This object represents the boolean function that defines the transition boundaries. This function must implement a method called *__residuumSign(self)*, that returns either -1 or 1, depending on the system conditions (see section 16).

Example:

```
<Simulation>
 ...
 <Models>
  ...
    <PostProcessor name="computeLimitSurface"
       subType='LimitSurface' verbosity='debug'>
      <parameters>x0,y0</parameters>
      <ROM class='Models' type='ROM'>Acc</ROM>
      <!-- Here, you can add a ROM defined in Models block.
           If it is not Present, a nearest neighbor algorithm
           will be used.
```

```
      -->
      <Function class='Functions' type='External'>
        goalFunctionForLimitSurface
      </Function>
    </PostProcessor>
    ...
  </Models>
  ...
</Simulation>
```

15.5.6 LimitSurfaceIntegral

The **LimitSurfaceIntegral** post-processor is aimed to compute the likelihood (probability) of the event, whose boundaries are represented by the Limit Surface (either from the LimitSurface post-processor or Adaptive sampling strategies). The inputted Limit Surface needs to be, in the **Post-Process** step, of type **PointSet** and needs to contain both boundary sides (-1.0, +1.0).
The **LimitSurfaceIntegral** post-processor accepts as output **PointSet**s only.

In order to use the *LimitSurfaceIntegral* PP, the user needs to set the `subType` of a `<PostProcessor>` node:

`<PostProcessor name='ppName' subType='LimitSurfaceIntegral'/>`.

Several sub-nodes are available:

- `<variable>`, ***XML node, required parameter*** can specify the following attribute:
  - `name`, ***required string attribute***, user-defined name of this variable.
  - `shape`, ***comma-separated integers, optional field***, determines the number of samples and shape of samples to be taken. For example, `shape`="2,3" will provide a 2 by 3 matrix of values, while `shape`="10" will produce a vector of 10 values. Omitting this optional attribute will result in a single scalar value instead. Each of the values in the matrix or vector will be the same as the single sampled value. **Note:** A model interface must be prepared to handle non-scalar inputs to use this option.

  This `<variable>` recognizes the following child node:

  - `<outputName>`, ***string, required field***, specifies the name of the output variable where the probability is going to be stored. **Note:** This variable name must be listed in the `<Output>` field of the Output DataObject
  - `<distribution>`, ***string, optional field***, name of the distribution that is associated to this variable. Its name needs to be contained in the `<Distributions>`

block explained in Section 9. If this node is not present, the **<lowerBound>** and **<upperBound>** XML nodes must be inputted. It requires the following two attributes:

  * **class**, ***required string attribute***, is the main "class" the listed object is from, the only acceptable class for this post-processor is **'Distributions'**;
  * **type**, ***required string attribute***, is the type of distributions, i.e. Normal, Uniform.

- **<lowerBound>**, ***float, optional field***, lower limit of integration domain for this dimension (variable). If this node is not present, the **<distribution>** XML node must be inputted.
- **<upperBound>**, ***float, optional field***, upper limit of integration domain for this dimension (variable). If this node is not present, the **<distribution>** XML node must be inputted.

- **<tolerance>**, ***float, optional field***, specifies the tolerance for numerical integration confidence.
  Default: 1.0e-4

- **<integralType>**, ***string, optional field***, specifies the type of integrations that need to be used. Currently only MonteCarlo integration is available
  Default: MonteCarlo

- **<computeBounds>**, ***bool, optional field***, activates the computation of the bounding error of the limit surface integral (maximum error in the identification of the limit surface location). If True, the bounding error is stored in a variable named as **<outputName>** appending the suffix "_err". For example, if **<outputName>** is "EventProbability", the bounding error will be stored as "EventProbability_err" (this variable name must be listed as variable in the output DataObject).
  Default: False

- **<seed>**, ***integer, optional field***, specifies the random number generator seed.
  Default: 20021986

- **<target>**, ***string, optional field***, specifies the target name that represents the $f(\bar{x})$ that needs to be integrated.
  Default: last output found in the inputted PointSet

Example:

```
<Simulation>
 ...
 <Models>
  ...
    <PostProcessor name="LimitSurfaceIntegralDistributions"
       subType='LimitSurfaceIntegral'>
```

```
        <tolerance>0.0001</tolerance>
        <integralType>MonteCarlo</integralType>
        <seed>20021986</seed>
        <target>goalFunctionOutput</target>
        <outputName>EventProbability</outputName>
        <variable name='x0'>
          <distribution>x0_distrib</distribution>
        </variable>
        <variable name='y0'>
          <distribution>y0_distrib</distribution>
        </variable>
    </PostProcessor>
    <PostProcessor name="LimitSurfaceIntegralLowerUpperBounds"
       subType='LimitSurfaceIntegral'>
        <tolerance>0.0001</tolerance>
        <integralType>MonteCarlo</integralType>
        <seed>20021986</seed>
        <target>goalFunctionOutput</target>
        <outputName>EventProbability</outputName>
        <variable name='x0'>
          <lowerBound>-2.0</lowerBound>
          <upperBound>12.0</upperBound>
        </variable>
        <variable name='y0'>
            <lowerBound>-1.0</lowerBound>
            <upperBound>11.0</upperBound>
        </variable>
    </PostProcessor>
    ...
  </Models>
  ...
</Simulation>
```

15.5.7 External

The **External** post-processor will execute an arbitrary python function defined externally using the *Functions* interface (see Section 16 for more details).

In order to use the *External* PP, the user needs to set the `subType` of a `<PostProcessor>` node:

<PostProcessor name='ppName' subType='External'/>.

Several sub-nodes are available:

- **<method>**, ***comma separated string, required field***, lists the method names of an external Function that will be computed (each returning a post-processing value). **Note:** New variable names will be defined as: "Function Name in this post-processor" + "_" + "variable name in XML node **<method>**". These new varialbes will be used to store the computed values from the list of methods, and can be accessed by the users through RAVEN entities **DataObjects** and **OutStreams**.
- **<Function>**, ***xml node, required string field***, specifies the name of a Function where the *methods* listed above are defined. **Note:** This name should match one of the Functions defined in the **<Functions>** block of the input file. The objects must be listed with a rigorous syntax that, except for the XML node tag, is common among all the objects. Each of these sub-nodes must contain 2 attributes that are used to map them within the simulation framework:
  - **class**, ***required string attribute***, is the main "class" the listed object is from, the only acceptable class for this post-processor is **'Functions'**;
  - **type**, ***required string attribute***, is the object identifier or sub-type, the only acceptable type for this post-processor is **'External'**.

This Post-Processor accepts as Input/Output both **'PointSet'** and **'HistorySet'**:

- If a **'PointSet'** is used as Input, the parameters are passed in the external **'Function'** as numpy arrays. The methods' return type must be either a new array or a scalar. In the following it is reported an example with two methods, one that returns a scalar and the other one that returns an array:

```
import numpy as np
def sum(self):
  return np.sum(self.aParameterInPointSet)

def sumTwoArraysAndReturnAnotherone(self):
  return self.aParamInPointSet1+self.aParamInPointSet2
```

- If a **'HistorySet'** is used as Input, the parameters are passed in the external **'Function'** as a list of numpy arrays. The methods' return type must be either a new list of arrays (if the Output is another **'HistorySet'**), a scalar or a single array (if the Output is **'PointSet'** . In the following it is reported an example with two methods, one that returns a new list of arrays (Output = HistorySet) and the other one that returns an array (Output = PointSet):

```
import numpy as np
def newHistorySetParameter(self):
  x = []*len(self.time)
  for history in range(len(self.time)):
    for ts in range(len(self.time[history])):
      if self.time[history][ts] >= 0.001: break
    x[history] = self.x[history][ts:]
  return x

def aNewPointSetParameter(self):
  x = []*len(self.time)
  for history in range(len(self.time)):
    x[history] = self.x[history][-1]
  return x
```

Example:

```
<Simulation>
  ...
  <Models>
    ...
    <PostProcessor name="externalPP" subType='External'
       verbosity='debug'>
      <method>Delta,Sum</method>
      <Function class='Functions'
         type='External'>operators</Function>
        <!-- Here, you can add a Function defined in the
             Functions block. This should be present or
             else RAVEN will not know where to find the
             defined methods. -->
    </PostProcessor>
    ...
  </Models>
  ...
</Simulation>
```

Note: The calculation results from this post-processor are stored in the internal variables. These variables are accessible by the users through RAVEN entities **DataObjects** and **OutStreams**. The names of these variables are defined as: "Function Name in this post-processor" + "_" + "variable name in XML node **<method>**". For example, in previous case, variables "operators_Delta" and "operators_Sum" are defined by RAVEN to store the outputs of this post-processor.

15.5.8 TopologicalDecomposition

The **TopologicalDecomposition** post-processor will compute an approximated hierarchical Morse-Smale complex which will add two columns to a dataset, namely `minLabel` and `maxLabel` that can be used to decompose a dataset.

The topological post-processor can also be run in 'interactive' mode, that is by passing the keyword `interactive` to the command line of RAVEN's driver. In this way, RAVEN will initiate an interactive UI that allows one to explore the topological hierarchy in real-time and adjust the simplification setting before adjusting a dataset. Use in interactive mode will replace the parameter **`<simplification>`** described below with whatever setting is set in the UI upon exiting it.

In order to use the **TopologicalDecomposition** post-processor, the user needs to set the attribute `subType`: **`<PostProcessor subType='TopologicalDecomposition'>`**. The following is a list of acceptable sub-nodes:

- **`<graph>`** ***, string, optional field***, specifies the type of neighborhood graph used in the algorithm, available options are:
  - `beta skeleton`
  - `relaxed beta skeleton`
  - `approximate knn`

  Default: `beta skeleton`
- **`<gradient>`**, ***string, optional field***, specifies the method used for estimating the gradient, available options are:
  - `steepest`

  Default: `steepest`
- **`<beta>`**, ***float in the range: (0,2], optional field***, is only used when the **`<graph>`** is set to `beta skeleton` or `relaxed beta skeleton`.
  Default: 1.0
- **`<knn>`**, ***integer, optional field***, is the number of neighbors when using the **`'approximate knn'`** for the **`<graph>`** sub-node and used to speed up the computation of other graphs by using the approximate knn graph as a starting point for pruning. -1 means use a fully connected graph.
  Default: -1

- **<weighted>**, ***boolean, optional***, a flag that specifies whether the regression models should be probability weighted.
  Default: False

- **<interactive>**, if this node is present *and* the user has specified the keyword `interactive` at the command line, then this will initiate a graphical interface for exploring the different simplification levels of the topological hierarchy. Upon exit of the graphical interface, the specified simplification level will be updated to use the last value of the graphical interface before writing any "output" results.

- **<persistence>**, ***string, optional field***, specifies how to define the hierarchical simplification by assigning a value to each local minimum and maximum according to the one of the strategy options below:

  – `difference` - The function value difference between the extremum and its closest-valued neighboring saddle.

  – `probability` - The probability integral computed as the sum of the probability of each point in a cluster divided by the count of the cluster.

  – `count` - The count of points that flow to or from the extremum.

  Default: `difference`

- **<simplification>**, ***float, optional field***, specifies the amount of noise reduction to apply before returning labels.
  Default: 0

- **<parameters>**, ***comma separated string, required field***, lists the parameters defining the input space.

- **<response>**, ***string, required field***, is a single variable name defining the scalar output space.

Example:

```
<Simulation>
  ...
  <Models>
    ...
    <PostProcessor name="***" subType='TopologicalDecomposition'>
      <graph>beta skeleton</graph>
      <gradient>steepest</gradient>
      <beta>1</beta>
      <knn>8</knn>
      <normalization>None</normalization>
```

```
      <parameters>X,Y</parameters>
      <response>Z</response>
      <weighted>true</weighted>
      <simplification>0.3</simplification>
      <persistence>difference</persistence>
    </PostProcessor>
    ...
  <Models>
  ...
<Simulation>
```

15.5.9 DataMining

Knowledge discovery in databases (KDD) is the process of discovering useful knowledge from a collection of data. This widely used data mining technique is a process that includes data preparation and selection, data cleansing, incorporating prior knowledge on data sets and interpreting accurate solutions from the observed results. Major KDD application areas include marketing, fraud detection, telecommunication and manufacturing.

DataMining is the analysis step of the KDD process. The overall of the data mining process is to extract information from a data set and transform it into an understandable structure for further use. The actual data mining task is the automatic or semi-automatic analysis of large quantities of data to extract previously unknown, interesting patterns such as groups of data records (cluster analysis), unusual records (anomaly detection), and dependencies (association rule mining).
In order to use the **DataMining** post-processor, the user needs to set the attribute `subType`:

`<PostProcessor subType= 'DataMining'>`.

The following is a list of acceptable sub-nodes:

- `<KDD>` ***string,required field***, the subnodes specifies the necessary information for the algorithm to be used in the postprocessor. The `<KDD>` has the required attribute: `lib`, the name of the library the algorithm belongs to. Current algorithms applied in the KDD model is based on SciKit-Learn library. Thus currently there is only one library:

  - `'SciKitLearn'`

  The `<KDD>` has the optional attribute: `labelFeature`, the name associated to labels or dimensions generated by the **DataMining** post-processor. The default name depends on the type of algorithm employed. For clustering and mixture models it is the name of the Post-Processor followed by "Labels" (e.g., if the name of a clustering PostProcessor is "kMeans"

then the default name associated to the labels is "kMeansLabels" if not specified in the attribute **labelFeature**). For decomposition and manifold models, the default names are the name of the PostProcessor followed by "Dimension" and an integer identifier beginning with 1. (e.g., if the name of a dimensionality reduction PostProcessor is "dr" and the user specifies 3 components, then the output dataObject will have three new outputs named "drDimension1," "drDimension2," and "drDimension3."). **Note:** The "Labels" are automatically added to the output **DataObjects**. It is also accessible by the users using the variable name defined above.

15.5.9.1 SciKitLearn

'SciKitLearn' is based on algorithms in SciKit-Learn library, and it performs data mining over PointSet and HistorySet. Note that for HistorySet's **'SciKitLearn'** performs the task given in **<SKLType>** (see below) for each time step, and so only synchronized HistorySet can be used as input to this model. For unsynchronized HistorySet, use **'HistorySetSync'** method in **'Interfaced'** post-processor to synchronize the input data before using **'SciKitLearn'**. The rest of this subsection and following subsection is dedicated to the **'SciKitLearn'** library.

The temporal variable for a HistorySet **'SciKitLearn'** is specified in the **<pivotParameter>** node:

- **<pivotParameter>**, ***string, optional parameter*** specifies the pivot variable (e.g., time, etc) in the input HistorySet.
  Default: None.

The algorithm for the dataMining is chosen by the subnode **<SKLType>** under the parent node **<KDD>**. The format is same as in 15.3.7. However, for the completeness sake, it is repeated here.

The data that are used in the training of the **DataMining** postprocessor are suplied with subnode **<Features>** in the parent node **<KDD>**.

- **<SKLtype>**, ***vertical bar (|) separated string, required field***, contains a string that represents the data mining algorithm to be used. As mentioned, its format is:
  <SKLtype>`mainSKLclass|algorithm`**</SKLtype>** where the first word (before the "|" symbol) represents the main class of algorithms, and the second word (after the "|" symbol) represents the specific algorithm.

- **<Features>**, ***string, required field***, defines the data to be used for training the data mining algorithm. It can be:

  – the name of the variable in the defined dataObject entity

- the location (i.e. input or output). In this case the data mining is applied to all the variables in the defined space.

The **<KDD>** node can have either optional or required subnodes depending on the dataMining algorithm used. The possible subnodes will be described separately for each algorithm below. The time dependent clustering data mining algorithms have a **<reOrderStep>** option that will try and keep the same labels on the clusters. The higher the number, the longer the history that the clustering algorithm will look through to maintain the same labeling between time steps.

All the available algorithms are described in the following sections.

15.5.9.2 Gaussian mixture models

A Gaussian mixture model is a probabilistic model that assumes all the data points are generated from a mixture of a finite number of Gaussian distributions with unknown parameters.
Scikit-learn implements different classes to estimate Gaussian mixture models, that correspond to different estimation strategies, detailed below.

15.5.9.2.1 GMM classifier

The GMM object implements the expectation-maximization (EM) algorithm for fitting mixture-of-Gaussian models. The GMM comes with different options to constrain the covariance of the difference classes estimated: spherical, diagonal, tied or full covariance.

In order to use the *Gaussian Mixture Model*, the user needs to set the sub-node:

<SKLtype>`mixture|GMM`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<n_components>**, ***integer, optional field*** Number of mixture components.
  Default: 1
- **<covariance_type>**, ***string, optional field***, describes the type of covariance parameters to use. Must be one of 'spherical', 'tied', 'diag', 'full'.
  Default: diag
- **<random_state>**, ***integer seed or random number generator instance, optional field***, A random number generator instance
  Default: 0 or None

- **<min_covar>**, ***float, optional field***, Floor on the diagonal of the covariance matrix to prevent overfitting.
  Default: 1e-3.

- **<thresh>**, ***float, optional field***, convergence threshold.
  Default: 0.01

- **<n_iter>**, ***integer, optional field***, Number of EM iterations to perform.
  Default: 100

- **<n_init>**, ***integer, optional***, Number of initializations to perform. the best results is kept.
  Default: 1

- **<init_params>**, ***string, optional field***, The method used to initialize the weights, the means and the precisions. Must be one of "kmeans" (responsibilities are initialized using kmeans) or "random" (responsibilities are initialized randomly)
  Default: kmeans

Example:

```
<Simulation>
  ...
  <Models>
    ...
      <PostProcessor name='PostProcessorName'
         subType='DataMining'>
          <KDD lib='SciKitLearn'>
              <Features>variableName</Features>
              <SKLtype>mixture|GMM</SKLtype>
              <n_components>2</n_components>
              <covariance_type>spherical</covariance_type>
          </KDD>
      </PostProcessor>
    ...
  <Models>
  ...
<Simulation>
```

15.5.9.2.2 Variational GMM Classifier (VBGMM)

The VBGMM object implements a variant of the Gaussian mixture model with variational inference algorithms. The API is identical to GMM.

In order to use the *Variational Gaussian Mixture Model*, the user needs to set the sub-node:

<SKLtype>`mixture|VBGMM`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<n_components>**, ***integer, optional field*** Number of mixture components.
  Default: 1
- **<covariance_type>**, ***string, optional field***, describes the type of covariance parameters to use. Must be one of 'spherical', 'tied', 'diag', 'full'.
  Default: diag
- **<alpha>**, ***float, optional field***, represents the concentration parameter of the dirichlet process. Intuitively, the Dirichlet Process is as likely to start a new cluster for a point as it is to add that point to a cluster with alpha elements. A higher alpha means more clusters, as the expected number of clusters is $\alpha * log(N)$.
  Default: 1.

15.5.9.3 Clustering

Clustering of unlabeled data can be performed with this subType of the DataMining PostProcessor.

An overwiev of the different clustering algorithms is given in Table5.

Table 5: Overview of Clustering Methods

Method name	**Parameters**	**Scalability**	**Usecase**	**Geometry (metric used)**
K-Means	number of clusters	Very large n_samples, medium n_clusters with MiniBatch code	General-purpose, even cluster size, flat geometry, not too many clusters	Distances between points
Affinity propagation	damping, sample preference	Not scalable with n_samples	Many clusters, uneven cluster size, non-flat geometry	Graph distance (e.g. nearest-neighbor graph)
Mean-shift	bandwidth	Not scalable with n_samples	Many clusters, uneven cluster size, non-flat geometry	Distances between points
Spectral clustering	number of clusters	Medium n_samples, small n_clusters	Few clusters, even cluster size, non-flat geometry	Graph distance (e.g. nearest-neighbor graph)
Ward hierarchical clustering	number of clusters	Large n_samples and n_clusters	Many clusters, possibly connectivity constraints	Distances between points
Agglomera-tive clustering	number of clusters, linkage type, distance	Large n_samples and n_clusters	Many clusters, possibly connectivity constraints, non Euclidean distances	Any pairwise distance
DBSCAN	neighborhood size	Very large n_samples, medium n_clusters	Non-flat geometry, uneven cluster sizes	Distances between nearest points
Gaussian mixtures	many	Not scalable	Flat geometry, good for density estimation	Mahalanobis distances to centers

15.5.9.3.1 K-Means Clustering

The KMeans algorithm clusters data by trying to separate samples in n groups of equal variance, minimizing a criterion known as the inertia or within-cluster sum-of-squares. This algorithm requires the number of clusters to be specified. It scales well to large number of samples and has been used across a large range of application areas in many different fields

In order to use the *K-Means Clustering*, the user needs to set the sub-node:

<SKLtype>`cluster|KMeans`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<n_clusters>**, ***integer, optional field*** The number of clusters to form as well as the number of centroids to generate.
  Default: 8
- **<max_iter>**, ***integer, optional field***, Maximum number of iterations of the k-means algorithm for a single run.
  Default: 300
- **<n_init>**, ***integer, optional field***, Number of time the k-means algorithm will be run with different centroid seeds. The final results will be the best output of n_init consecutive runs in terms of inertia.
  Default: 3
- **<init>**, ***string, optional***, Method for initialization, k-means++', 'random' or an ndarray:
  - 'k-means++' : selects initial cluster centers for k-mean clustering in a smart way to speed up convergence.
  - 'random': choose k observations (rows) at random from data for the initial centroids.
  - If an ndarray is passed, it should be of shape (n_clusters, n_features) and gives the initial centers.
- **<precompute_distances>**, ***boolean, optional field***, Precompute distances (if true faster but takes more memory).
  Default: true
- **<tol>**, ***float, optional field***, Relative tolerance with regards to inertia to declare convergence.
  Default: 1e-4
- **<n_jobs>**, ***integer, optional field***, The number of jobs to use for the computation. This works by breaking down the pairwise matrix into n jobs even slices and computing them in parallel. If -1 all CPUs are used. If 1 is given, no parallel computing code is used at all,

which is useful for debugging. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Thus for n_jobs = -2, all CPUs but one are used.
Default: 1

- **<random_state>**, ***integer or numpy.RandomState, optional field*** The generator used to initialize the centers. If an integer is given, it fixes the seed.
Default: the global numpy random number generator.

Example:

```
<Simulation>
  ...
  <Models>
    ...
      <PostProcessor name='PostProcessorName'
         subType='DataMining'>
          <KDD lib='SciKitLearn'>
              <Features>variableName</Features>
              <SKLtype>cluster|KMeans</SKLtype>
              <n_clusters>2</n_clusters>
              <tol>0.0001</tol>
              <init>random</init>
          </KDD>
      </PostProcessor>
    ...
  <Models>
  ...
<Simulation>
```

15.5.9.3.2 Mini Batch K-Means

The MiniBatchKMeans is a variant of the KMeans algorithm which uses mini-batches to reduce the computation time, while still attempting to optimise the same objective function. Mini-batches are subsets of the input data, randomly sampled in each training iteration.

MiniBatchKMeans converges faster than KMeans, but the quality of the results is reduced. In practice this difference in quality can be quite small.

In order to use the *Mini Batch K-Means Clustering*, the user needs to set the sub-node:

<SKLtype>`cluster|MiniBatchKMeans`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<n_clusters>**, ***integer, optional field*** The number of clusters to form as well as the number of centroids to generate.
  Default: 8

- **<max_iter>**, ***integer, optional field***, Maximum number of iterations of the k-means algorithm for a single run.
  Default: 100

- **<max_no_improvement>**, ***integer, optional firld***, Control early stopping based on the consecutive number of mini batches that does not yield an improvement on the smoothed inertia. To disable convergence detection based on inertia, set max_no_improvement to None.

  Default: 10

- **<tol>**, ***float, optional field***, Control early stopping based on the relative center changes as measured by a smoothed, variance-normalized of the mean center squared position changes. This early stopping heuristics is closer to the one used for the batch variant of the algorithms but induces a slight computational and memory overhead over the inertia heuristic. To disable convergence detection based on normalized center change, set tol to 0.0 (default).
  Default: 0.0

- **<batch_size>**, ***integer, optional field***, Size of the mini batches.
  Default: 100

- init_size, ***integer, optional field***, Number of samples to randomly sample for speeding up the initialization (sometimes at the expense of accuracy): the only algorithm is initialized by running a batch KMeans on a random subset of the data. *This needs to be larger than k.*,
  *Default: 3 * **<batch_size>***

- **<init>**, ***string, optional***, Method for initialization, k-means++', 'random' or an ndarray:

  - 'k-means++' : selects initial cluster centers for k-mean clustering in a smart way to speed up convergence.
  - 'random': choose k observations (rows) at random from data for the initial centroids.
  - If an ndarray is passed, it should be of shape (n_clusters, n_features) and gives the initial centers.

- **<precompute_distances>**, ***boolean, optional field***, Precompute distances (if true faster but takes more memory).
  Default: true

- **<n_init>**, ***integer, optional field***, Number of time the k-means algorithm will be run with different centroid seeds. The final results will be the best output of n_init consecutive runs in terms of inertia.
  Default: 3

- **<compute_labels>**, ***boolean, optional field***, Compute label assignment and inertia for the complete dataset once the minibatch optimization has converged in fit.
  Default: True

- **<random_state>**, ***integer or numpy.RandomState, optional field*** The generator used to initialize the centers. If an integer is given, it fixes the seed.
  Default: the global numpy random number generator.

- reassignment_ratio, **<float, optional field>**, Control the fraction of the maximum number of counts for a center to be reassigned. A higher value means that low count centers are more easily reassigned, which means that the model will take longer to converge, but should converge in a better clustering.
  Default: 0.01

15.5.9.3.3 Affinity Propagation

AffinityPropagation creates clusters by sending messages between pairs of samples until convergence. A dataset is then described using a small number of exemplars, which are identified as those most representative of other samples. The messages sent between pairs represent the suitability for one sample to be the exemplar of the other, which is updated in response to the values from other pairs. This updating happens iteratively until convergence, at which point the final exemplars are chosen, and hence the final clustering is given.

In order to use the *AffinityPropogation Clustering*, the user needs to set the sub-node:

<SKLtype>`cluster|AffinityPropogation`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<damping>**, ***float, optional field***, Damping factor between 0.5 and 1.
  Default: 0.5

- **<convergence_iter>**, ***integer, optional field***, Number of iterations with no change in the number of estimated clusters that stops the convergence.
  Default: 15

- **<max_iter>**, ***integer, optional field***, Maximum number of iterations.
  Default: 200

- **<copy>**, ***boolean, optional field***, Make a copy of input data or not.
  Default: True

- **<preference>**, ***array-like, shape (n_samples,) or float, optional field***, Preferences for each point - points with larger values of preferences are more likely to be chosen as exem-

plars. The number of exemplars, ie of clusters, is influenced by the input preferences value.
Default: If the preferences are not passed as arguments, they will be set to the median of the input similarities.

- **<affinity>**, ***string, optional field***,Which affinity to use. At the moment precomputed and euclidean are supported. euclidean uses the negative squared euclidean distance between points.
Default: "euclidean"

- **<verbose>**, ***boolean, optional field***, Whether to be verbose.
Default: False

15.5.9.3.4 Mean Shift

MeanShift clustering aims to discover blobs in a smooth density of samples. It is a centroid based algorithm, which works by updating candidates for centroids to be the mean of the points within a given region. These candidates are then filtered in a post-processing stage to eliminate near-duplicates to form the final set of centroids.

In order to use the *Mean Shift Clustering*, the user needs to set the sub-node:

<SKLtype>`cluster|MeanShift`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<bandwidth>**, ***float, optional field***, Bandwidth used in the RBF kernel. If not given, the bandwidth is estimated using *sklearn.cluster.estimate_bandwidth*; see the documentation for that function for hints on scalability.

- **<seeds>**, ***array, shape=[n_samples, n_features], optional field***, Seeds used to initialize kernels. If not set, the seeds are calculated by *clustering.get_bin_seeds* with bandwidth as the grid size and default values for other parameters.

- **<bin_seeding>**, ***boolean, optional field***, If true, initial kernel locations are not locations of all points, but rather the location of the discretized version of points, where points are binned onto a grid whose coarseness corresponds to the bandwidth. Setting this option to True will speed up the algorithm because fewer seeds will be initialized.
Default: False Ignored if seeds argument is not None.

- **<min_bin_freq>**, ***integer, optional field***, To speed up the algorithm, accept only those bins with at least min_bin_freq points as seeds.
Default: 1.

- **<cluster_all>**, ***boolean, optional field***, If true, then all points are clustered, even those orphans that are not within any kernel. Orphans are assigned to the nearest kernel. If false,

then orphans are given cluster label -1.
Default: True

15.5.9.3.5 Spectral clustering

SpectralClustering does a low-dimension embedding of the affinity matrix between samples, followed by a *KMeans* in the low dimensional space. It is especially efficient if the affinity matrix is sparse and the pyamg module is installed.

In order to use the *Spectral Clustering*, the user needs to set the sub-node:

<SKLtype>`cluster|Spectral`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<n_clusters>**, ***integer, optional field***, The dimension of the projection subspace.
  Default: 8
- **<affinity>**, ***string, array-like or callable, optional field***, If a string, this may be one of:
  - 'nearest_neighbors',
  - 'precomputed',
  - 'rbf' or
  - one of the kernels supported by *sklearn.metrics.pairwise_kernels*.

  Only kernels that produce similarity scores (non-negative values that increase with similarity) should be used. This property is not checked by the clustering algorithm.
  Default: 'rbf'
- **<gamma>**, ***float, optional field***, Scaling factor of RBF, polynomial, exponential chi^2 and sigmoid affinity kernel. Ignored for $affinity =' nearest_neighbors'$.
  Default: 1.0
- **<degree>**,***float, optional field***, Degree of the polynomial kernel. Ignored by other kernels.

  Default: 3
- **<coef0>**, ***float, optional field***, Zero coefficient for polynomial and sigmoid kernels. Ignored by other kernels.
  Default: 1
- **<n_neighbors>**, ***integer, optional field***, Number of neighbors to use when constructing the affinity matrix using the nearest neighbors method. Ignored for affinity='rbf'.
  Default: 10

- **<eigen_solver>** ***string, optional field***, The eigenvalue decomposition strategy to use:
  - None,
  - 'arpack',
  - 'lobpcg', or
  - 'amg'

  Note: AMG requires pyamg to be installed. It can be faster on very large, sparse problems, but may also lead to instabilities

- **<random_state>**, ***integer seed, RandomState instance, or None, optional field***, A pseudo random number generator used for the initialization of the lobpcg eigen vectors decomposition when $eigen_solver == 'amg'$ and by the K-Means initialization.
  Default: None

- **<n_init>**, ***integer, optional field***, Number of time the k-means algorithm will be run with different centroid seeds. The final results will be the best output of n_init consecutive runs in terms of inertia.
  Default: 10

- **<eigen_tol>**,***float, optional field***, Stopping criterion for eigendecomposition of the Laplacian matrix when using arpack eigen_solver.
  Default: 0.0

- **<assign_labels>**, ***string, optional field***, The strategy to use to assign labels in the embedding space. There are two ways to assign labels after the laplacian embedding:
  - 'kmeans',
  - 'discretize'

  k-means can be applied and is a popular choice. But it can also be sensitive to initialization. Discretization is another approach which is less sensitive to random initialization.
  Default: 'kmeans'

- **<kernel_params>**, ***dictionary of string to any, optional field***, Parameters (keyword arguments) and values for kernel passed as callable object. Ignored by other kernels.
  Default: None

Notes

If you have an affinity matrix, such as a distance matrix, for which 0 means identical elements, and high values means very dissimilar elements, it can be transformed in a similarity matrix that is well suited for the algorithm by applying the Gaussian (RBF, heat) kernel:

$$np.exp(-X**2/(2.*delta**2)) \tag{39}$$

Another alternative is to take a symmetric version of the k nearest neighbors connectivity matrix of the points. If the *pyamg* package is installed, it is used: this greatly speeds up computation.

15.5.9.3.6 DBSCAN Clustering

The Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm views clusters as areas of high density separated by areas of low density. Due to this rather generic view, clusters found by DBSCAN can be any shape, as opposed to k-means which assumes that clusters are convex shaped.

In order to use the *DBSCAN Clustering*, the user needs to set the sub-node:

<SKLtype>`cluster|DBSCAN`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<eps>**, ***float, optional field***, The maximum distance between two samples for them to be considered as in the same neighborhood.
  Default: 0.5
- **<min_samples>**, ***integer, optional field***, The number of samples in a neighborhood for a point to be considered as a core point.
  Default: 5
- **<metric>**, ***string, or callable, optional field*** The metric to use when calculating distance between instances in a feature array. If metric is a string or callable, it must be one of the options allowed by *metrics.pairwise.calculate_distance* for its metric parameter. If metric is "precomputed", X is assumed to be a distance matrix and must be square.
  Default: 'euclidean'
- **<random_state>**, ***numpy.RandomState, optional field***, The generator used to initialize the centers.
  Default: numpy.random.

15.5.9.3.7 Agglomerative Clustering

Hierarchical clustering is a general family of clustering algorithms that build nested clusters by merging or splitting them successively. This hierarchy of clusters is represented as a tree (or dendrogram). The root of the tree is the unique cluster that gathers all of the samples, the leaves being the clusters with only one sample. The AgglomerativeClustering object performs a hierarchical clustering using a bottom up approach: each observation starts in its own cluster, and clusters are successively merged together. The linkage criteria determines the metric used for the merge strategy:

- Ward: it minimizes the sum of squared differences within all clusters. It is a variance-minimizing approach and in this sense is similar to the k-means objective function but tackled with an agglomerative hierarchical approach.

- Maximum or complete linkage: it minimizes the maximum distance between observations of pairs of clusters.
- Average linkage: it minimizes the average of the distances between all observations of pairs of clusters.

AgglomerativeClustering can also scale to large number of samples when it is used jointly with a connectivity matrix, but is computationally expensive when no connectivity constraints are added between samples: it considers at each step all of the possible merges.

In order to use the *Agglomerative Clustering*, the user needs to set the sub-node:

<SKLtype>`cluster|Agglomerative`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<n_clusters>**, ***int, optional field***, The number of clusters to find.
  Default: 2
- **<connectivity>**, ***array like or callable, optional field***, Connectivity matrix. Defines for each sample the neighboring samples following a given structure of the data. This can be a connectivity matrix itself or a callable that transforms the data into a connectivity matrix, such as derived from kneighbors graph. Default is None, i.e, the hierarchical clustering algorithm is unstructured.
  Default: None
- **<affinity>**, ***string or callable, optional field***, Metric used to compute the linkage. Can be "euclidean", "$l1$", "$l2$", "manhattan","cosine", or "precomputed". If linkage is "ward", only "euclidean" is accepted.
  Default: euclidean
- **<n_components>**, ***int, optional field***, Number of connected components. If None the number of connected components is estimated from the connectivity matrix. NOTE: This parameter is now directly determined from the connectivity matrix and will be removed in 0.18.
- **<linkage>**, ***ward,complete,average, optional field***, Which linkage criterion to use. The linkage criterion determines which distance to use between sets of observation. The algorithm will merge the pairs of cluster that minimize this criterion. Ward minimizes the variance of the clusters being merged. Average uses the average of the distances of each observation of the two sets. Complete or maximum linkage uses the maximum distances between all observations of the two sets..
  Default: ward

15.5.9.3.8 Clustering performance evaluation

Evaluating the performance of a clustering algorithm is not as trivial as counting the number of errors or the precision and recall of a supervised classification algorithm. In particular any evaluation metric should not take the absolute values of the cluster labels into account but rather if this clustering define separations of the data similar to some ground truth set of classes or satisfying some assumption such that members belong to the same class are more similar that members of different classes according to some similarity metric.

If the ground truth labels are not known, evaluation must be performed using the model itself. The **Silhouette Coefficient** is an example of such an evaluation, where a higher Silhouette Coefficient score relates to a model with better defined clusters. The Silhouette Coefficient is defined for each sample and is composed of two scores:

1. The mean distance between a sample and all other points in the same class.
2. The mean distance between a sample and all other points in the next nearest cluster.

The Silhoeutte Coefficient s for a single sample is then given as:

$$s = \frac{b - a}{max(a, b)} \tag{40}$$

The Silhouette Coefficient for a set of samples is given as the mean of the Silhouette Coefficient for each sample. In normal usage, the Silhouette Coefficient is applied to the results of a cluster analysis.

Advantages

- The score is bounded between -1 for incorrect clustering and +1 for highly dense clustering. Scores around zero indicate overlapping clusters.
- The score is higher when clusters are dense and well separated, which relates to a standard concept of a cluster.

Drawbacks

The Silhouette Coefficient is generally higher for convex clusters than other concepts of clusters, such as density based clusters like those obtained through DBSCAN.

15.5.9.4 Decomposing signals in components (matrix factorization problems)

15.5.9.4.1 Principal component analysis (PCA)

- **Exact PCA and probabilistic interpretation**
  Linear Dimensionality reduction using Singular Value Decomposition of the data and keeping only the most significant singular vectors to project the data to a lower dimensional space. In order to use the *Exact PCA*, the user needs to set the sub-node:

  <SKLtype>`decomposition|PCA`**</SKLtype>**.

  In addition to this XML node, several others are available:

  - **<n_components>**, ***integer, None or String, optional field***, Number of components to keep. if
  - **<n_components>** is not set all components are kept,
    Default: all components
  - **<copy>**, ***boolean, optional field***, If False, data passed to fit are overwritten and running fit(X).transform(X) will not yield the expected results, use fit_transform(X) instead.
    Default: True
  - **<whiten>**, ***boolean, optional field***, When True the components_ vectors are divided by n_samples times singular values to ensure uncorrelated outputs with unit component-wise variances. Whitening will remove some information from the transformed signal (the relative variance scales of the components) but can sometime improve the predictive accuracy of the downstream estimators by making there data respect some hard-wired assumptions.
    Default: False

  Example:

```
<Simulation>
  ...
  <Models>
    ...
      <PostProcessor name='PostProcessorName'
         subType='DataMining'>
          <KDD lib='SciKitLearn'>
              <Features>variable1,variable2,variable3,
                 variable4,variable5</Features>
              <SKLtype>decomposition|PCA</SKLtype>
              <n_components>2</n_components>
          </KDD>
      </PostProcessor>
    ...
  <Models>
  ...
<Simulation>
```

- **Randomized (Approximate) PCA**
  Linear Dimensionality reduction using Singular Value Decomposition of the data and keeping only the most significant singular vectors to project the data to a lower dimensional space. In order to use the *Randomized PCA*, the user needs to set the sub-node:

  <SKLtype>`decomposition|RandomizedPCA`**</SKLtype>**.

  In addition to this XML node, several others are available:

  - **<n_components>**, ***interger, None or String, optional field***, Number of components to keep. if n_components is not set all components are kept.
    Default: all components
  - **<copy>**, ***boolean, optional field***, If False, data passed to fit are overwritten and running fit(X).transform(X) will not yield the expected results, use fit_transform(X) instead.
    Default: True
  - **<iterated_power>**, ***integer, optional field***, Number of iterations for the power method.
    Default: 3
  - **<whiten>**, ***boolean, optional field***, When True the components_ vectors are divided by n_samples times singular values to ensure uncorrelated outputs with unit component-wise variances. Whitening will remove some information from the transformed signal (the relative variance scales of the components) but can sometime improve the predictive accuracy of the downstream estimators by making there data respect some hard-wired assumptions.
    Default: False
  - **<random_state>**, ***int, or Random State instance or None, optional field***, Pseudo Random Number generator seed control. If None, use the numpy.random singleton.
    Default: None

- **Kernel PCA**
  Non-linear dimensionality reduction through the use of kernels. In order to use the *Kernel PCA*, the user needs to set the sub-node:

  <SKLtype>`decomposition|KernelPCA`**</SKLtype>**.

  In addition to this XML node, several others are available:

  - **<n_components>**, ***interger, None or String, optional field***, Number of components to keep. if n_components is not set all components are kept.
    Default: all components
  - **<kernel>**, ***string, optional field***, name of the kernel to be used, options are:
    * linear
    * poly

* rbf
* sigmoid
* cosine
* precomputed

Default: linear **<degree>**, ***integer, optional field***, Degree for poly kernels, ignored by other kernels.
Default: 3 **<gamma>**, ***float, optional field***, Kernel coefficient for rbf and poly kernels, ignored by other kernels.
Default: 1/n_features

- **<coef0>**, ***float, optional field***, independent term in poly and sigmoig kernels, ignored by other kernels.
- **<kernel_params>**, ***mapping of string to any, optional field***, Parameters (keyword arguments) and values for kernel passed as callable object. Ignored by other kernels.
  Default: 3
- alpha, ***int, optional field***, Hyperparameter of the ridge regression that learns the inverse transform (when fit_inverse_transform=True).
  Default: 1.0
- **<fit_inverse_transform>**, ***bool, optional field***, Learn the inverse transform for non-precomputed kernels. (i.e. learn to find the pre-image of a point)
  Default: False
- **<eigen_solver>**, ***string, optional field***, Select eigensolver to use. If n_components is much less than the number of training samples, arpack may be more efficient than the dense eigensolver. Options are:
  * auto
  * dense
  * arpack

  Default: False
- tol, ***float, optional field***, convergence tolerance for arpack.
  Default: 0 (optimal value will be chosen by arpack)
- max_iter, ***int, optional field***, maximum number of iterations for arpack.
  Default: None (optimal value will be chosen by arpack)
- **<remove_zero_eig>**, ***boolean, optional field***, If True, then all components with zero eigenvalues are removed, so that the number of components in the output may be ¡ n_components (and sometimes even zero due to numerical instability). When n_components is None, this parameter is ignored and components with zero eigenvalues are removed regardless.
  Default: True

- **Sparse PCA**
  Finds the set of sparse components that can optimally reconstruct the data. The amount of sparseness is controllable by the coefficient of the L1 penalty, given by the parameter alpha. In order to use the *Sparse PCA*, the user needs to set the sub-node:

  <SKLtype>`decomposition|SparsePCA`**</SKLtype>**.

  In addition to this XML node, several others are available:

  - **<n_components>**, ***integer, optional field***, Number of sparse atoms to extract.
    Default: None
  - **<alpha>**, ***float, optional field***, Sparsity controlling parameter. Higher values lead to sparser components.
    Default: 1.0
  - **<ridge_alpha>**, ***float, optional field***, Amount of ridge shrinkage to apply in order to improve conditioning when calling the transform method.
    Default: 0.01
  - **<max_iter>**, ***float, optional field***, maximum number of iterations to perform.
    Default: 1000
  - **<tol>**, ***float, optional field***, convergence tolerance.
    Default: 1E-08
  - **<method>**, ***string, optional field***, method to use, options are:
    * lars: uses the least angle regression method to solve the lasso problem (linear_model.lars_path)
    * cd: uses the coordinate descent method to compute the Lasso solution (linear_model.Lasso)

    Lars will be faster if the estimated components are sparse.
    Default: lars
  - **<n_jobs>**, ***int, optional field***, number of parallel runs to run.
    Default: 1
  - **<U_init>**, ***array of shape (n_samples, n_components) , optional field***, Initial values for the loadings for warm restart scenarios
    Default: None
  - **<V_init>**, ***array of shape (n_components, n_features), optional field***, Initial values for the components for warm restart scenarios
    Default: None
  - verbose, ***boolean, optional field***, Degree of verbosity of the printed output.
    Default: False
  - random_state, ***int or Random State, optional field***, Pseudo number generator state used for random sampling.
    Default: None

- **Mini Batch Sparse PCA**
  Finds the set of sparse components that can optimally reconstruct the data. The amount of sparseness is controllable by the coefficient of the L1 penalty, given by the parameter alpha. In order to use the *Mini Batch Sparse PCA*, the user needs to set the sub-node:

  <SKLtype>`decomposition|MiniBatchSparsePCA`**</SKLtype>**.

  In addition to this XML node, several others are available:

  - **<n_components>**, ***integer, optional field***, Number of sparse atoms to extract.
    Default: None
  - **<alpha>**, ***float, optional field***, Sparsity controlling parameter. Higher values lead to sparser components.
    Default: 1.0
  - **<ridge_alpha>**, ***float, optional field***, Amount of ridge shrinkage to apply in order to improve conditioning when calling the transform method.
    Default: 0.01
  - **<n_iter>**, ***float, optional field***, number of iterations to perform per mini batch.
    Default: 100
  - **<callback>**, ***callable, optional field***, callable that gets invoked every five iterations.

    Default: None
  - **<batch_size>**, ***int, optional field***, the number of features to take in each mini batch.

    Default: 3
  - **<verbose>**, ***boolean, optional field***, Degree of verbosity of the printed output.
    Default: False
  - **<shuffle>**, ***boolean, optional field***, whether to shuffle the data before splitting it in batches.
    Default: True
  - **<n_jobs>**, ***integer, optional field***, Parameters (keyword arguments) and values for kernel passed as callable object. Ignored by other kernels.
    Default: 3
  - **<metho>**, ***string, optional field***, method to use, options are:
    * lars: uses the least angle regression method to solve the lasso problem (linear_model.lars_path),
    * cd: uses the coordinate descent method to compute the Lasso solution (linear_model.Lasso)

    Lars will be faster if the estimated components are sparse.
    Default: lars
  - **<random_state>**, ***integer or Random State, optional field***, Pseudo number generator state used for random sampling.
    Default: None

15.5.9.4.2 Truncated singular value decomposition

Dimensionality reduction using truncated SVD (aka LSA). In order to use the *Truncated SVD*, the user needs to set the sub-node:

<SKLtype>`decomposition|TruncatedSVD`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<n_components>**, ***integer, optional field***, Desired dimensionality of output data. Must be strictly less than the number of features. The default value is useful for visualisation. For LSA, a value of 100 is recommended.
  Default: 2
- **<algorithm>**, ***string, optional field***, SVD solver to use:
  - Randomized: randomized algorithm
  - Arpack: ARPACK wrapper in.

  Default: Randomized
- **<n_iter>**, ***float, optional field***, number of iterations andomized SVD solver. Not used by ARPACK.
  Default: 5
- **<random_state>**, ***int or Random State, optional field***, Pseudo number generator state used for random sampling. If not given, the numpy.random singleton is used.
  Default: None
- **<tol>**, ***float, optional field***, Tolerance for ARPACK. 0 means machine precision. Ignored by randomized SVD solver.
  Default: 0.0

15.5.9.4.3 Fast ICA

A fast algorithm for Independent Component Analysis. In order to use the *Fast ICA*, the user needs to set the sub-node:

<SKLtype>`decomposition|FastICA`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<n_components>**, ***integer, optional field***, Number of components to use. If none is passed, all are used.
  Default: None

- **<algorithm>**, ***string, optional field***, algorithm used in FastICA:
  - parallel,
  - deflation.

  Default: parallel

- **<fun>**, ***string or function, optional field***, The functional form of the G function used in the approximation to neg-entropy. Could be either:
  - logcosh,
  - exp, or
  - cube.

  One can also provide own function. It should return a tuple containing the value of the function, and of its derivative, in the point.
  Default: logcosh

- **<fun_args>**, ***dictionary, optional field***, Arguments to send to the functional form. If empty and if fun='logcosh', fun_args will take value 'alpha' : 1.0.
  Default: None

- **<max_iter>**, ***float, optional field***, maximum number of iterations during fit.
  Default: 200

- **<tol>**, ***float, optional field***, Tolerance on update at each iteration.
  Default: 0.0001

- **<w_init>**, ***None or an (n_components, n_components) ndarray, optional field***, The mixing matrix to be used to initialize the algorithm.
  Default: None

- **<randome_state>**, ***int or Random State, optional field***, Pseudo number generator state used for random sampling.
  Default: None

15.5.9.5 Manifold learning

A manifold is a topological space that resembles a Euclidean space locally at each point. Manifold learning is an approach to non-linear dimensionality reduction. It assumes that the data of interest lie on an embedded non-linear manifold within the higher-dimensional space. If this manifold is of low dimension, data can be visualized in the low-dimensional space. Algorithms for this task are based on the idea that the dimensionality of many data sets is only artificially high.

15.5.9.5.1 Isomap

Non-linear dimensionality reduction through Isometric Mapping (Isomap). In order to use the *Isometric Mapping*, the user needs to set the sub-node:

<SKLtype>`manifold|Isomap`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<n_neighbors>**, ***integer, optional field***, Number of neighbors to consider for each point.
  Default: 5
- **<n_components>**, ***integer, optional field***, Number of coordinates to manifold.
  Default: 2
- **<eigen_solver>**, ***string, optional field***, eigen solver to use:
  - auto: Attempt to choose the most efficient solver for the given problem,
  - arpack: Use Arnoldi decomposition to find the eigenvalues and eigenvectors
  - dense: Use a direct solver (i.e. LAPACK) for the eigenvalue decomposition

  Default: auto
- **<tol>**, ***float, optional field***, Convergence tolerance passed to arpack or lobpcg. not used if eigen_solver is 'dense'.
  Default: 0.0
- **<max_iter>**, ***float, optional field***, Maximum number of iterations for the arpack solver. not used if eigen_solver == 'dense'.
  Default: None
- **<path_method>**, ***string, optional field***, Method to use in finding shortest path. Could be either:
  - Auto: attempt to choose the best algorithm
  - FW: Floyd-Warshall algorithm
  - D: Dijkstra algorithm with Fibonacci Heaps

  Default: auto
- **<neighbors_algorithm>**, ***string, optional field***, Algorithm to use for nearest neighbors search, passed to neighbors.NearestNeighbors instance.
  - auto,

- brute
- kd_tree
- ball_tree

Default: auto

Example:

```
<Simulation>
  ...
  <Models>
    ...
      <PostProcessor name='PostProcessorName'
         subType='DataMining'>
          <KDD lib='SciKitLearn'>
              <Features>input</Features>
              <SKLtype>manifold|Isomap</SKLtype>
              <n_neighbors>5</n_neighbors>
              <n_components>3</n_components>
              <eigen_solver>arpack</eigen_solver>
              <neighbors_algorithm>kd_tree</neighbors_algorithm>
            </KDD>
      </PostProcessor>
    ...
  <Models>
  ...
<Simulation>
```

15.5.9.5.2 Locally Linear Embedding

In order to use the *Locally Linear Embedding*, the user needs to set the sub-node:

`<SKLtype>manifold|LocallyLinearEmbedding</SKLtype>`.

In addition to this XML node, several others are available:

- **`<n_neighbors>`**, ***integer, optional field***, Number of neighbors to consider for each point. *Default: 5*
- **`<n_components>`**, ***integer, optional field***, Number of coordinates to manifold. *Default: 2*

- **<reg>**, ***float, optional field***, regularization constant, multiplies the trace of the local covariance matrix of the distances.
  Default: 0.01

- **<eigen_solver>**, ***string, optional field***, eigen solver to use:

  - auto: Attempt to choose the most efficient solver for the given problem,
  - arpack: use arnoldi iteration in shift-invert mode.
  - dense: use standard dense matrix operations for the eigenvalue

  Default: auto

- **<tol>**, ***float, optional field***, Convergence tolerance passed to arpack. not used if eigen_solver is 'dense'.
  Default: 1E-06

- **<max_iter>**, ***int, optional field***, Maximum number of iterations for the arpack solver. not used if eigen_solver == 'dense'.
  Default: 100

- **<method>**, ***string, optional field***, Method to use. Could be either:

  - Standard: use the standard locally linear embedding algorithm
  - hessian: use the Hessian eigenmap method
  - itsa: use local tangent space alignment algorithm

  Default: standard

- **<hessian_tol>**, ***float, optional field***, Tolerance for Hessian eigenmapping method. Only used if method == 'hessian'
  Default: 0.0001

- **<modified_tol>**, ***float, optional field***, Tolerance for modified LLE method. Only used if method == 'modified'
  Default: 0.0001

- **<neighbors_algorithm>**, ***string, optional field***, Algorithm to use for nearest neighbors search, passed to neighbors.NearestNeighbors instance.

  - auto,
  - brute
  - kd_tree

- ball_tree

Default: auto

- **<random_state>**, ***int or numpy random state, optional field***, the generator or seed used to determine the starting vector for arpack iterations.
  Default: None

15.5.9.5.3 Spectral Embedding

Spectral embedding for non-linear dimensionality reduction, it forms an affinity matrix given by the specified function and applies spectral decomposition to the corresponding graph laplacian. The resulting transformation is given by the value of the eigenvectors for each data point In order to use the *Spectral Embedding*, the user needs to set the sub-node:

<SKLtype>`manifold|SpectralEmbedding`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<n_components>**, ***integer, optional field***, the dimension of projected sub-space.
  Default: 2
- **<eigen_solver>**, ***string, optional field***, the eigen value decomposition strategy to use:
  - none,
  - arpack.
  - lobpcg,
  - amg

  Default: none
- **<random_state>**, ***integer or numpy random state, optional field***, A pseudo random number generator used for the initialization of the lobpcg eigen vectors decomposition when eigen_solver == 'amg.
  Default: None
- **<affinity>**, ***string or callable, optional field***, How to construct the affinity matrix:
  - *nearest_neighbors* : construct affinity matrix by knn graph
  - *rbf* : construct affinity matrix by rbf kernel
  - *precomputed* : interpret X as precomputed affinity matrix

- *callable* : use passed in function as affinity the function takes in data matrix (n_samples, n_features) and return affinity matrix (n_samples, n_samples).

Default: nearest_neighbor

- **<gamma>**, ***float, optional field***, Kernel coefficient for rbf kernel.
  Default: None

- **<n_neighbors>**, ***int, optional field***, Number of nearest neighbors for nearest_neighbors graph building.
  Default: None

15.5.9.5.4 Multi-dimensional Scaling (MDS)

In order to use the *Multi Dimensional Scaling*, the user needs to set the sub-node:

<SKLtype>`manifold|MDS`**</SKLtype>**.

In addition to this XML node, several others are available:

- **<metric>**, ***boolean, optional field***, compute metric or nonmetric SMACOF (Scaling by Majorizing a Complicated Function) algorithm
  Default: True

- **<n_components>**, ***integer, optional field***, number of dimension in which to immerse the similarities overridden if initial array is provided.
  Default: 2

- **<n_init>**, ***integer, optional field***, Number of time the smacof algorithm will be run with different initialisation. The final results will be the best output of the n_init consecutive runs in terms of stress.
  Default: 4

- **<max_iter>**, ***integer, optional field***, Maximum number of iterations of the SMACOF algorithm for a single run
  Default: 300

- **<verbose>**, ***integer, optional field***, level of verbosity
  Default: 0

- **<eps>**, ***float, optional field***, relative tolerance with respect to stress to declare converge
  Default: 1E-06

- **<n_jobs>**, ***integer, optional field***, The number of jobs to use for the computation. This works by breaking down the pairwise matrix into n_jobs even slices and computing them in parallel. If -1 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Thus for n_jobs = -2, all CPUs but one are used.
  Default: 1

- **<random_state>**, **<integer or numpy random state, optional field>**, The generator used to initialize the centers. If an integer is given, it fixes the seed. Defaults to the global numpy random number generator.
  Default: None

- **<dissimilarity>**, ***string, optional field***, Which dissimilarity measure to use. Supported are 'euclidean' and 'precomputed'.
  Default: euclidean

15.5.9.6 Scipy

'Scipy' provides a Hierarchical clustering that performs clustering over PointSet and HistorySet. This algorithm also automatically generates a dendrogram in .pdf format (i.e., dendrogram.pdf).

- **<SCIPYtype>**, ***string, required field***, SCIPY algorithm to be employed.

- **<Features>**, ***string, required field***, defines the data to be used for training the data mining algorithm. It can be:

  - the name of the variable in the defined dataObject entity
  - the location (i.e. input or output). In this case the data mining is applied to all the variables in the defined space.

- **<method>**, ***string, required field***, The linkage algorithm to be used
  Default: single, complete, weighted, centroids, median, ward.

- **<metric>**, ***string, required field***, The distance metric to be used
  Default: 'braycurtis', 'canberra', 'chebyshev', 'cityblock', 'correlation', 'cosine', 'dice', 'euclidean', 'hamming', 'jaccard', 'kulsinski', 'mahalanobis', 'matching', 'minkowski', 'rogerstanimoto', 'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean', 'yule'.

- **<level>**, ***float, required field***, Clustering distance level where actual clusters are formed.

- **<criterion>**, ***string, required field***, The criterion to use in forming flat clusters. This can be any of the following values:

- "inconsistent" : If a cluster node and all its descendants have an inconsistent value less than or equal to 't' then all its leaf descendants belong to the same flat cluster. When no non-singleton cluster meets this criterion, every node is assigned to its own cluster. (Default)
- "distance" : Forms flat clusters so that the original observations in each flat cluster have no greater a cophenetic distance than t.
- "maxclust" : Finds a minimum threshold "r" so that the cophenetic distance between any two original observations in the same flat cluster is no more than "r" and no more than t flat clusters are formed.
- "monocrit" : Forms a flat cluster from a cluster node c with index i when $monocrit[j] <= t$.
- "maxclust_monocrit" : Forms a flat cluster from a non-singleton cluster node "c" when $monocrit[i] <= r$ for all cluster indices "i" below and including "c". "r" is minimized such that no more than "t" flat clusters are formed. monocrit must be monotonic.

- **`<dendrogram>`**, ***boolean, required field***, If True the dendrogram is actually created.
- **`<truncationMode>`**, ***string, required field***, The dendrogram can be hard to read when the original observation matrix from which the linkage is derived is large. Truncation is used to condense the dendrogram. There are several modes:
  - "None": No truncation is performed (Default).
  - "lastp": The last p non-singleton formed in the linkage are the only non-leaf nodes in the linkage; they correspond to rows $Z[n-p-2:end]$ in Z. All other non-singleton clusters are contracted into leaf nodes.
  - "level"/"mtica": No more than p levels of the dendrogram tree are displayed. This corresponds to Mathematica behavior.
- **`<p>`**, ***int, required field***, The p parameter for truncationMode.
- **`<leafCounts>`**, ***boolean, required field***, When True the cardinality non singleton nodes contracted into a leaf node is indiacted in parenthesis.
- **`<showContracted>`**, ***boolean, required field***, When True the heights of non singleton nodes contracted into a leaf node are plotted as crosses along the link connecting that leaf node.
- **`<annotatedAbove>`**, ***float, required field***, Clustering level above which the branching level is annotated.

Example:

```
<Simulation>
  ...
  <Models>
    ...
    <PostProcessor name="hierarchical" subType="DataMining"
       verbosity="quiet">
      <KDD lib="Scipy" labelFeature='labels'>
        <SCIPYtype>cluster|Hierarchical</SCIPYtype>
        <Features>output</Features>
        <method>single</method>
        <metric>euclidean</metric>
        <level>75</level>
        <criterion>distance</criterion>
        <dendrogram>true</dendrogram>
        <truncationMode>lastp</truncationMode>
        <p>20</p>
        <leafCounts>True</leafCounts>
        <showContracted>True</showContracted>
        <annotatedAbove>10</annotatedAbove>
      </KDD>
    </PostProcessor>
    ...
  <Models>
  ...
<Simulation>
```

15.5.10 Discrete Risk Measures

This Post-Processor calculates a series of risk importance measures from a PointSet. This calculation is performed for a set of input parameters given an output target.

The user is required to provide the following information:

- the set of input variables. For each variable the following need to be specified:
  - the set of values that imply a reliability value equal to 1 for the input variable
  - the set of values that imply a reliability value equal to 0 for the input variable
- the output target variable. For this variable it is needed to specify the values of the output target variable that defines the desired outcome.

The following variables are first determined for each input variable i:

- R_0 Probability of the outcome of the output target variable (nominal value)
- R_i^+ Probability of the outcome of the output target variable if reliability of the input variable is equal to 0
- R_i^- Probability of the outcome of the output target variable if reliability of the input variable is equal to 1

Available measures are:

- Risk Achievement Worth (RAW): $RAW = R_i^+/R_0$
- Risk Achievement Worth (RRW): $RRW = R_0/R_i^-$
- Fussell-Vesely (FV): $FV = (R_0 - R_i^-)/R_0$
- Birnbaum (B): $B = R_i^+ - R_i^-$

In order to use the *RiskMeasureDiscrete* PP, the user needs to set the **subType** of a **<PostProcessor>** node:

<PostProcessor name='ppName' subType='RiskMeasureDiscrete'/>.

Several sub-nodes are available:

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

- **<measures>**, ***string, required field***, desired risk importance measures that have to be computed (RRW, RAW, FV, B)
- **<variable>**, ***string, required field***, ID of the input variable. This node is provided for each input variable. This nodes needs to contain also these attributes:
  - **R0values**, ***float, required field***, interval of values (comma separated values) that implies a reliability value equal to 0 for the input variable
  - **R1values**, ***float, required field***, interval of values (comma separated values) that implies a reliability value equal to 1 for the input variable
- **<target>**, ***string, required field***, ID of the output variable. This nodes needs to contain also the attribute **values**, ***string, required field***, interval of values of the output target variable that defines the desired outcome

Example: This example shows an example where it is desired to calculate all available risk importance measures for two input variables (i.e., pumpTime and valveTime) given an output target variable (i.e., Tmax). A value of the input variable pumpTime in the interval $[0, 240]$ implies a reliability value of the input variable pumpTime equal to 0. A value of the input variable valveTime in the interval $[0, 60]$ implies a reliability value of the input variable valveTime equal to 0. A value of the input variables valveTime and pumpTime in the interval $[1441, 2880]$ implies a reliability value of the input variables equal to 1. The desired outcome of the output variable Tmax occurs in the interval $[2200, 2500]$.

```
<Simulation>
  ...
  <Models>
    ...
    <PostProcessor name="riskMeasuresDiscrete"
       subType="RiskMeasuresDiscrete">
      <measures>B,FV,RAW,RRW</measures>
      <variable R0values='0,240'
         R1values='1441,2880'>pumpTime</variable>
      <variable R0values='0,60'
         R1values='1441,2880'>valveTime</variable>
      <target   values='2200,2500'
         >Tmax</target>
    </PostProcessor>
    ...
  </Models>
  ...
</Simulation>
```

This Post-Processor allows the user to consider also multiple datasets (a data set for each initiating event) and calculate the global risk importance measures. This can be performed by:

- Including all datasets in the step

```
<Simulation>
  ...
  </Steps>
    ...
    <PostProcess name="PP">
      <Input    class="DataObjects"  type="PointSet"
         >outRun1</Input>
      <Input    class="DataObjects"  type="PointSet"
         >outRun2</Input>
      <Model    class="Models"       type="PostProcessor"
         >riskMeasuresDiscrete</Model>
```

```
      <Output  class="DataObjects"  type="PointSet"
         >outPPS</Output>
      <Output  class="OutStreams"   type="Print"
         >PrintPPS_dump</Output>
    </PostProcess>
  </Steps>
  ...
</Simulation>
```

- Adding in the Post-processor the frequency of the initiating event associated to each dataset

```
<Simulation>
  ...
  <Models>
    ...
    <PostProcessor name="riskMeasuresDiscrete"
       subType="RiskMeasuresDiscrete">
      <measures>FV,RAW</measures>
      <variable R1values='-0.1,0.1'
         R0values='0.9,1.1'>Astatus</variable>
      <variable R1values='-0.1,0.1'
         R0values='0.9,1.1'>Bstatus</variable>
      <variable R1values='-0.1,0.1'
         R0values='0.9,1.1'>Cstatus</variable>
      <variable R1values='-0.1,0.1'
         R0values='0.9,1.1'>Dstatus</variable>
      <target   values='0.9,1.1'>outcome</target>
      <data     freq='0.01'>outRun1</data>
      <data     freq='0.02'>outRun2</data>
    </PostProcessor>
    ...
  </Models>
  ...
</Simulation>
```

This post-processor can be made time dependent if a single HistorySet is provided among the other data objects. The HistorySet contains the temporal profiles of a subset of the input variables. This temporal profile can be only boolean, i.e., 0 (component offline) or 1 (component online). Note that the provided history set must contains a single History; multiple Histories are not allowed. When this post-processor is in a dynamic configuration (i.e., time-dependent), the user is required to specify an xml node **<temporalID>** that indicates the ID of the temporal variable. For each time instant, this post-processor determines the temporal profiles of the desired

risk importance measures. Thus, in this case, an HistorySet must be chosen as an output data object. An example is shown below:

```
<Simulation>
  ...
  <Models>
    ...
    <PostProcessor name="riskMeasuresDiscrete"
       subType="RiskMeasuresDiscrete">
      <measures>B,FV,RAW,RRW,R0</measures>
      <variable R1values='-0.1,0.1'
         R0values='0.9,1.1'>Astatus</variable>
      <variable R1values='-0.1,0.1'
         R0values='0.9,1.1'>Bstatus</variable>
      <variable R1values='-0.1,0.1'
         R0values='0.9,1.1'>Cstatus</variable>
      <target   values='0.9,1.1'>outcome</target>
      <data     freq='1.0'>outRun1</data>
      <temporalID>time</temporalID>
    </PostProcessor>
    ...
  </Models>
  ...
  <Steps>
    ...
    <PostProcess name="PP">
      <Input     class="DataObjects"  type="PointSet"
         >outRun1</Input>
      <Input     class="DataObjects"  type="HistorySet"
         >timeDepProfiles</Input>
      <Model     class="Models"       type="PostProcessor"
         >riskMeasuresDiscrete</Model>
      <Output    class="DataObjects"  type="HistorySet"
         >outHS</Output>
      <Output    class="OutStreams"   type="Print"
         >PrintHS</Output>
    </PostProcess>
    ...
  </Steps>
  ...
</Simulation>
```

15.5.11 HistorySetDelay

This Post-Processor allows history sets to add delayed or lagged variables. It copies a variable, but with a delay. For example, if there a variable price that is set hourly, than new variable called price_prev_hour could be set by using a delay of -1 as seen in the listing below. This can be useful for training a ROM or other data analysis.

In order to use the *HistorySetDelay* PP, the user needs to set the `subType` of a `<PostProcessor>` node:

`<PostProcessor name='ppName' subType='HistorySetDelay'/>`.

Several sub-nodes are available:

In the `<PostProcessor>` input block, one or more of the following XML sub-nodes are required:

- `<delay>`, ***empty***, a delay node with the following required parameters:
  - `original`, ***string, required field***, the variable to start with
  - `new`, ***string, required field***, the new variable to create
  - `steps`, ***integer, required field***, the delay (if negative) or steps into the future (if positive) to use for the new variable (so -1 gives the previous, 1 gives the next)
  - `default`, ***float, required field***, the value to use for cases where there is no previous or next value (such as the beginning when a negative delay is used, or the end when the delay is positive).

```
<Simulation>
  ...
  <Models>
    ...
    <PostProcessor name="delayPP" subType="HistorySetDelay">
      <delay original="price" new="price_prev_hour" steps="-1"
         default="0.0"/>
      <delay original="price" new="price_prev_day" steps="-24"
         default="0.0"/>
      <delay original="price" new="price_prev_week" steps="-168"
         default="-1.0"/>
    </PostProcessor>
  </Models>
  ...
  <Steps>
```

```
    ...
    <PostProcess name="delay">
      <Input class="DataObjects"
         type="HistorySet">samples</Input>
      <Model class="Models" type="PostProcessor">delayPP</Model>
      <Output class="DataObjects"
         type="HistorySet">delayed_samples</Output>
    </PostProcess>
    ...
  </Steps>
  ...
  <DataObjects>
    <HistorySet name="samples">
      <Input>demand</Input>
      <Output>price</Output>
      <options>
        <pivotParameter>hour</pivotParameter>
      </options>
    </HistorySet>
    <HistorySet name="delayed_samples">
      <Input>demand</Input>
      <Output>price,price_prev_hour,price_prev_day,price_prev_week</Output>
      <options>
        <pivotParameter>hour</pivotParameter>
      </options>
    </HistorySet>
    ...
  </DataObjects>
</Simulation>
```

15.5.12 Interfaced

The **Interfaced** post-processor is a Post-Processor that allows the user to create its own Post-Processor. While the External Post-Processor (see Section 15.5.7 allows the user to create case-dependent Post-Processors, with this new class the user can create new general purpose Post-Processors.

In order to use the *Interfaced* PP, the user needs to set the `subType` of a `<PostProcessor>` node:

```
<PostProcessor name='ppName' subType='Interfaced'/>.
```

Several sub-nodes are available:

- **<method>**, ***comma separated string, required field***, lists the method names of a method that will be computed (each returning a post-processing value). All available methods need to be included in the "/raven/framework/PostProcessorFunctions/" folder

Example:

```
<Simulation>
  ...
  <Models>
    ...
    <PostProcessor name="example"
       subType='InterfacedPostProcessor'verbosity='debug'>
       <method>testInterfacedPP</method>
       <!--Here, the xml nodes required by the chosen method
          have to be
       included.
        -->
    </PostProcessor>
    ...
  </Models>
  ...
</Simulation>
```

All the **Interfaced** post-processors need to be contained in the "/raven/framework/PostProcessorFunctions/" folder. In fact, once the **Interfaced** post-processor is defined in the RAVEN input file, RAVEN search that the method of the post-processor is located in such folder.

The class specified in the **Interfaced** post-processor has to inherit the PostProcessorInterfaceBase class and the user must specify this set of methods:

- initialize: in this method, the internal parameters of the post-processor are initialized. Mandatory variables that needs to be specified are the following:
  - self.inputFormat: type of dataObject expected in input
  - self.outputFormat: type of dataObject generated in output
- readMoreXML: this method is in charge of reading the PostProcessor xml node, parse it and fill the PostProcessor internal variables.
- run: this method performs the desired computation of the dataObject.

```
from PostProcessorInterfaceBaseClass import PostProcessorInterfaceBase
class testInterfacedPP(PostProcessorInterfaceBase):
  def initialize(self)
  def readMoreXML(self,xmlNode)
  def run(self,inputDic)
```

15.5.12.1 Data Format

The user is not allowed to modify directly the DataObjects, however the content of the DataObjects is available in the form of a python dictionary. Both the dictionary give in input and the one generated in the output of the PostProcessor are structured as follows:

```
inputDict = {'data':{}, 'metadata':{}}
```

where:

```
inputDict['data'] = {'input':{}, 'output':{}}
```

In the input dictonary, each input variable is listed as a dictionary that contains a numpy array with its own values as shown below for a simplified example

```
inputDict['data']['input'] = {'inputVar1': array([ 1.,2.,3.]),
                              'inputVar2': array([4.,5.,6.])}
```

Similarly, if the dataObject is a PointSet then the output dictionary is structured as follows:

```
inputDict['data']['output'] = {'outputVar1': array([ .1,.2,.3]),
                               'outputVar2':array([.4,.5,.6])}
```

Howevers, if the dataObject is a HistorySet then the output dictionary is structured as follows:

```
inputDict['data']['output'] = {'hist1': {}, 'hist2':{}}
```

where

```
inputDict['output']['data'][hist1] = {'time': array([ .1,.2,.3]),
                              'outputVar1':array([ .4,.5,.6])}
inputDict['output']['data'][hist2] = {'time': array([ .1,.2,.3]),
                              'outputVar1':array([ .14,.15,.16])}
```

15.5.12.2 Method: HStoPSOperator

This Post-Processor performs the conversion from HistorySet to PointSet performing a projection of the output space.

In the **`<PostProcessor>`** input block, the following XML sub-nodes are available:

- **`<pivotParameter>`**, ***string, optional field***, ID of the temporal variable. Default is "time". **Note:** Used just in case the **`<pivotValue>`**-based operation is requested
- **`<operator>`**, ***string, optional field***, the operation to perform on the output space:
  - **min**, compute the minimum of each variable along each single history
  - **max**, compute the maximum of each variable along each single history
  - **average**, compute the average of each variable along each single history
  - **all**, join together all of the each variable in the history, and make the pivotParameter a regular parameter. Unlike the min and max operators, this keeps all the data, just organized differently. This operator does this by propogating the other input parameters for each item of the pivotParameter. Table 6 shows an example HistorySet with input parameter x, pivot parameter t, and output parameter b and then Table 7 shows the resulting PointSet with input parameters x and t, and output parameter b. Note that which parameters are input and which are output in the resulting PointSet depends on the DataObject specification.

  Note: This node can be inputted only if **`<pivotValue>`** and **`<row>`** are not present
- **`<pivotValue>`**, ***float, optional field***, the value of the pivotParameter with respect to the other outputs need to be extracted. **Note:** This node can be inputted only if **`<operator>`** and **`<row>`** are not present
- **`<pivotStrategy>`**, ***string, optional field***, The strategy to use for the pivotValue:
  - **nearest**, find the value that is the nearest with respect the **`<pivotValue>`**
  - **floor**, find the value that is the nearest with respect to the **`<pivotValue>`** but less then the **`<pivotValue>`**
  - **celing**, find the value that is the nearest with respect to the **`<pivotValue>`** but greater then the **`<pivotValue>`**
  - **interpolate**, if the exact **`<pivotValue>`** can not be found, interpolate using a linear approach

  Note: Valid just in case **`<pivotValue>`** is present

- **<row>**, ***int, optional field***, the row index at which the outputs need to be extracted. **Note:** This node can be inputted only if **<operator>** and **<pivotValue>** are not present

This example will show how the XML input block would look like:

```
<Simulation>
  ...
  <Models>
    ...
    <PostProcessor name="HStoPSperatorRows"
       subType="InterfacedPostProcessor">
      <method>HStoPSOperator</method>
      <row>-1</row>
    </PostProcessor>
    <PostProcessor name="HStoPSoperatorPivotValues"
       subType="InterfacedPostProcessor">
        <method>HStoPSOperator</method>
        <pivotParameter>time</pivotParameter>
        <pivotValue>0.3</pivotValue>
    </PostProcessor>
    <PostProcessor name="HStoPSoperatorOperatorMax"
       subType="InterfacedPostProcessor">
        <method>HStoPSOperator</method>
        <pivotParameter>time</pivotParameter>
        <operator>max</operator>
    </PostProcessor>
    <PostProcessor name="HStoPSoperatorOperatorMin"
       subType="InterfacedPostProcessor">
        <method>HStoPSOperator</method>
        <pivotParameter>time</pivotParameter>
        <operator>min</operator>
    </PostProcessor>
    <PostProcessor name="HStoPSoperatorOperatorAverage"
       subType="InterfacedPostProcessor">
        <method>HStoPSOperator</method>
        <pivotParameter>time</pivotParameter>
        <operator>average</operator>
    </PostProcessor>
    ...
  </Models>
  ...
</Simulation>
```

Table 6: Starting HistorySet for operator all

x	t	b
5.0		
	1.0	6.0
	2.0	7.0

Table 7: Resulting PointSet after operator all

x	t	b
5.0	1.0	6.0
5.0	2.0	7.0

15.5.12.3 Method: HistorySetSampling

This Post-Processor performs the conversion from HistorySet to HistorySet The conversion is made so that each history H is re-sampled accordingly to a specific sampling strategy. It can be used to reduce the amount of space required by the HistorySet.

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

- **`<samplingType>`**, ***string, required field***, specifies the type of sampling method to be used:
  - uniform: the set of **`<numberOfSamples>`** samples are uniformly distributed along the time axis
  - firstDerivative: the set of **`<numberOfSamples>`** samples are distributed along the time axis in regions with higher first order derivative
  - secondDerivative: the set of **`<numberOfSamples>`** samples are distributed along the time axis in regions with higher second order derivative
  - filteredFirstDerivative: samples are located where the first derivative is greater than the specified **`<tolerance>`** value (hence, the number of samples can vary from history to history)
  - filteredSecondDerivative: samples are located where the second derivative is greater than the specified **`<tolerance>`** value (hence, the number of samples can vary from history to history)
- **`<numberOfSamples>`**, ***integer, optional field***, number of samples (required only for the following sampling types: uniform, firstDerivative secondDerivative)
- **`<pivotParameter>`**, ***string, required field***, ID of the temporal variable

- **`<interpolation>`**, ***string, optional field***, type of interpolation to be employed for the history reconstruction (required only for the following sampling types: uniform, firstDerivative secondDerivative). Valid types of interpolation to specified: linear, nearest, zero, slinear, quadratic, cubic, intervalAverage

- **`<tolerance>`**, ***string, optional field***, tolerance level (required only for the following sampling types: filteredFirstDerivative or filteredSecondDerivative)

15.5.12.4 Method: HistorySetSync

This Post-Processor performs the conversion from HistorySet to HistorySet The conversion is made so that all histories are synchronized in time. It can be used to allow the histories to be sampled at the same time instant.

There are two possible synchronization methods, specified through the **`<syncMethod>`** node. If the **`<syncMethod>`** is **`'grid'`**, a **`<numberOfSamples>`** node is specified, which yields an equally-spaced grid of time points. The output values for these points will be linearly derived using nearest sampled time points, and the new HistorySet will contain only the new grid points.

The other methods are used by specifying **`<syncMethod>`** as **`'all'`**, **`'min'`**, or **`'max'`**. For **`'all'`**, the postprocessor will iterate through the existing histories, collect all the time points used in any of them, and use these as the new grid on which to establish histories, retaining all the exact original values and interpolating linearly where necessary. In the event of **`'min'`** or **`'max'`**, the postprocessor will find the smallest or largest time history, respectively, and use those time values as nodes to interpolate between.

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

- **`<pivotParameter>`**, ***string, required field***, ID of the temporal variable

- **`<extension>`**, ***string, required field***, type of extension when the sync process goes outside the boundaries of the history (zeroed or extended)

- **`<syncMethod>`**, ***string, required field***, synchronization strategy to employ (see description above). Options are **`'grid'`**, **`'all'`**, **`'max'`**, **`'min'`**.

- **`<numberOfSamples>`**, ***integer, optional field***, required if **`<syncMethod>`** is **`'grid'`**, number of new time samples

15.5.12.5 Method: HistorySetSnapShot

This Post-Processor performs a conversion from HistorySet to PointSet. The conversion is made so that each history H is converted to a single point P. There are several methods that can be employed to choose the single point from the history:

- min: Take a time slice when the **`<pivotVar>`** is at its smallest value,
- max: Take a time slice when the **`<pivotVar>`** is at its largest value,
- average: Take a time slice when the **`<pivotVar>`** is at its time-weighted average value,
- value: Take a time slice when the **`<pivotVar>`** *first passes* its specified value,
- timeSlice: Take a time slice index from the sampled time instance space.

To demonstrate the timeSlice, assume that each history H is a dict of n output variables $x_1 = [...], x_n = [...]$, then the resulting point P is at time instant index t: $P = [x_1[t], ..., x_n[t]]$.

Choosing one the these methods for the **`<type>`** node will take a time slice for all the variables in the output space based on the provided parameters. Alternatively, a **`'mixed'`** type can be used, in which each output variable can use a different time slice parameter. In other words, you can take the max of one variable while taking the minimum of another, etc.

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

- **`<type>`**, ***string, required field***, type of operation: **`'min'`**, **`'max'`**, **`'average'`**, **`'value'`**, **`'timeSlice'`**, or **`'mixed'`**
- **`<extension>`**, ***string, required field***, type of extension when the sync process goes outside the boundaries of the history (zeroed or extended)
- **`<pivotParameter>`**, ***string, optional field***, name of the temporal variable. Required for the **`'average'`** and **`'timeSlice'`** methods.

If a **`'timeSlice'`** type is in use, the following nodes also are required:

- **`<timeInstant>`**, ***integer, required field***, required and only used in the **`'timeSlice'`** type. Location of the time slice (integer index)
- **`<numberOfSamples>`**, ***integer, required field***, number of samples

If instead a **'min'**, **'max'**, **'average'**, or **'value'** is used, the following nodes are also required:

- **<pivotVar>**, ***string, required field***, Name of the chosen indexing variable (the variable whose min, max, average, or value is used to determine the time slice)
- **<pivotVal>**, ***float, optional field***, required for **'value'** type, the value for the chosen variable

Lastly, if a **'mixed'** approach is used, the following nodes apply:

- **<max>**, ***string, optional field***, the names of variables whose output should be their own maximum value within the history.
- **<min>**, ***string, optional field***, the names of variables whose output should be their own minimum value within the history.
- **<average>**, ***string, optional field***, the names of variables whose output should be their own average value within the history. Note that a **<pivotParameter>** node is required to perform averages.
- **<value>**, ***string, optional field***, the names of variables whose output should be taken at a time slice determined by another variable. As with the non-mixed **'value'** type, the first time the **pivotVar** crosses the specified **pivotVal** will be the time slice taken. This node requires two attributes, if used:
  - **pivotVar**, ***string, required field***, the name of the variable on which the time slice will be performed. That is, if we want the value of y when $t = 0.245$, this attribute would be **'t'**.
  - **pivotVal**, ***float, required field***, the value of the **pivotVar** on which the time slice will be performed. That is, if we want the value of y when $t = 0.245$, this attribute would be **'0.245'**.

  Note that all the outputs of the **<DataObject>** output of this postprocessor must be listed under one of the **'mixed'** node types in order for values to be returned.

Example (mixed): This example will output the average value of x for x, the value of y at time$= 0.245$ for y, and the value of z at $x = 4.0$ for z.

```
<Simulation>
  ...
  <Models>
    ...
```

```
    <PostProcessor name="mampp2"
       subType="InterfacedPostProcessor">
      <method>HistorySetSnapShot</method>
      <type>mixed</type>
      <average>x</average>
      <value pivotVar="time" pivotVal="0.245">y</value>
      <value pivotVar="x" pivotVal="4.0">z</value>
      <pivotParameter>time</pivotParameter>
      <extension>zeroed</extension>
    </PostProcessor>
    ...
  </Models>
  ...
</Simulation>
```

15.5.12.6 Method: HS2PS

This Post-Processor performs a conversion from HistorySet to PointSet. The conversion is made so that each history H is converted to a single point P. Assume that each history H is a dict of n output variables $x_1 = [...], x_n = [...]$, then the resulting point P is $P = concat(x_1, ..., x_n)$. Note: it is here assumed that all histories have been sync so that they have the same length, start point and end point. If you are not sure, do a pre-processing the the original history set.

In the **`<PostProcessor>`** input block, the following XML sub-nodes are required, independent of the **`subType`** specified (min, max, avg and value case):

- **`<pivotParameter>`**, ***string, optional field***, ID of the temporal variable (only for avg)

15.5.12.7 Method: TypicalHistoryFromHistorySet

This Post-Processor performs a simplified procedure of [7] to form a "typical" time series from multiple time series. The input should be a HistorySet, with each history in the HistorySet synchronized. For HistorySet that is not synchronized, use Post-Processor method **HistorySetSync** to synchronize the data before running this method.

Each history in input HistorySet is first converted to multiple histories each has maximum time specified in **`<outputLen>`** (see below). Each converted history H_i is divided into a set of subsequences $\{H_i^j\}$, and the division is guided by the **`<subseqLen>`** node specified in the input XML. The value of **`<subseqLen>`** should be a list of positive numbers that specify the length

of each subsequence. If the number of subsequence for each history is more than the number of values given in **<subseqLen>**, the values in **<subseqLen>** would be reused.

For each variable x, the method first computes the empirical CDF (cumulative density function) by using all the data values of x in the HistorySet. This CDF is termed as long-term CDF for x. Then for each subsequence H_i^j, the method computes the empirical CDF by using all the data values of x in H_i^j. This CDF is termed as subsequential CDF. For the first interval window (i.e., $j = 1$), the method computes the Finkelstein-Schafer (FS) statistics [8] between the long term CDF and the subsequential CDF of H_i^1 for each i. The FS statistics is defined as following.

$$FS = \sum_x FS_x$$

$$FS_x = \frac{1}{N} \sum_{n=1}^{N} \delta_n$$

where N is the number of value reading in the empirical CDF and δ_n is the absolute difference between the long term CDF and the subsequential CDF at value x_n. The subsequence H_i^1 with minimal FS statistics will be selected as the typical subsequence for the interval window $j = 1$. Such process repeats for $j = 2, 3, \ldots$ until all subsequences have been processed. Then all the typical subsequences will be concatenated to form a complete history.

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

- **<pivotParameter>**, ***string, optional field***, ID of the temporal variable
  Default: Time
- **<subseqLen>**, ***integers, required field***, length of the divided subsequence (see above)
- **<outputLen>**, ***integer, optional field***, maximum value of the temporal variable for the generated typical history
  Default: Maximum value of the variable with name of ***<pivotParameter>***

For example, consider history of data collected over three years in one-second increments, where the user wants a single *typical year* extracted from the data. The user wants this data constructed by combining twelve equal *typical month* segments. In this case, the parameter **<outputLen>** should be `31536000` (the number of seconds in a year), while the parameter **<subseqLen>** should be `2592000` (the number of seconds in a month). Using a value for **<subseqLen>** that is either much, much smaller than **<outputLen>** or of equal size to **<outputLen>** might have unexpected results. In general, we recommend using a **<subseqLen>** that is roughly an order of magnitude smaller than **<outputLen>**.

15.5.12.8 Method: dataObjectLabelFilter

This Post-Processor allows to filter the portion of a dataObject, either PointSet or HistorySet, with a given clustering label. A clustering algorithm associates a unique cluster label to each element of the dataObject (PointSet or HistorySet). This cluster label is a natural number ranging from 0 (or 1 depending on the algorithm) to N where N is the number of obtained clusters. Recall that some clustering algorithms (e.g., K-Means) receive N as input while others (e.g., Mean-Shift) determine N after clustering has been performed. Thus, this Post-Processor is naturally employed after a data-mining clustering techniques has been performed on a dataObject so that each clusters can be analyzed separately.

In the **`<PostProcessor>`** input block, the following XML sub-nodes are required, independently of the **`subType`** specified:

- **`<label>`**, ***string, required field***, name of the clustering label
- **`<clusterIDs>`**, ***integers, required field***, ID of the selected clusters. Note that more than one ID can be provided as input

15.5.13 ParetoFrontier

The **ParetoFrontier** post-processor is designed to identify the points lying on the Pareto Frontier in a multi-dimensional trade-space. This post-processor receives as input a **DataObject** (a PointSet only) which contains all data points in the trade-space space and it returns the subset of points lying in the Pareto Frontier as a PointSet.

It is here assumed that each data point of the input PointSet is a realization of the system under consideration for a specific configuration to which corresponds several objective variables (e.g., cost and value).

In order to use the *ParetoFrontier* PP, the user needs to set the **`subType`** of a **`<PostProcessor>`** node:

```
<PostProcessor name='ppName' subType='ParetoFrontier'/>.
```

Several sub-nodes are available:

- **`<objective>`**,***string, required parameter***, ID of the objective variable that represents a dimension of the trade-space space. The **`<costID>`** requires one identifying attribute:
  - **`goal`**, ***string, required field***, Goal of the objective variable characteristic: minimzation (min) or maximization (max)

- `upperLimit`, ***string, optional field***, Desired upper limit of the objective variable for the points in the Pareto frontier
- `lowerLimit`, ***string, optional field***, Desired lower limit of the objective variable for the points in the Pareto frontier

The following is an example where a set of realizations (the "candidates" PointSet) has been generated by changing two parameters (var1 and var2) which produced two output variables: cost (which it is desired to be minimized) and value (which it is desired to be maximized). The **ParetoFrontier** post-processor takes the "candidates" PointSet and populates a Point similar in structure (the "paretoPoints" PointSet).

Example:

Listing 1: ParetoFrontier input example (no expand).

```
<Models>
  <PostProcessor name="paretoPP" subType="ParetoFrontier">
    <objective goal='min' upperLimit='0.5'>cost</objective>
    <objective goal='max' lowerLimit='0.5'>value</objective>
  </PostProcessor>
</Models>

<Steps>
  <PostProcess name="PP">
    <Input     class="DataObjects"  type="PointSet"
       >candidates</Input>
    <Model     class="Models"       type="PostProcessor"
       >paretoPP</Model>
    <Output    class="DataObjects"  type="PointSet"
       >paretoPoints</Output>
  </PostProcess>
</Steps>

<DataObjects>
  <PointSet name="candidates">
    <Input>var1,var2</Input>
    <Output>cost,value</Output>
  </PointSet>
  <PointSet name="paretoPoints">
    <Input>var1,var2</Input>
    <Output>cost,value</Output>
  </PointSet>
</DataObjects>
```

Note: it is possible to specify both upper and lower limits for each objective variable. When one or both of these limits are specified, then the pareto frontier is filtered such that all pareto frontier points that satisfy those limits are preserved.

15.5.14 MCSImporter

The **MCSImporter** post-processor has been designed to import Minimal Cut Sets (MCSs) into RAVEN. This post-processor reads a csv file which contain the list of MCSs and it save this list as a DataObject (i.e., a PointSet). The csv file is composed by three columns; the first contains the ID number of the MCS, the second one contains the MCS probability value, the third one lists all the Basic Events contained in the MCS. An example of csv file is shown in Table 8.

Table 8: Example of csv file which contains four MCSs.

ID,	Prob,	MCS,
1.,	1.8E-2,	D
2.,	4.0E-3,	B
3.,	3.0E-4,	A,C
4.,	2.1E-5,	E,C

The PointSet is structured to include all Basic Event, the MCS ID, the MCS probability, and the outcome of such MCS (always set to 1). MCS ID and MCS probability are copied directly from the csv file. For each MCS, the Basic Events can have two possible values:

- 0: Basic Event is not included in the MCS
- 1: Basic Event is included in the MCS

The PointSet generated from the csv file of Table 8 is shown in Table 9.

Table 9: PointSet generated by RAVEN for the list of MCSs shown in Table 8.

A	B	C	D	E	MCS_ID	probability	out
0	0	0	1	0	1	1.8E-2	1
0	1	0	0	0	2	4.0E-3	1
1	0	1	0	0	3	3.0E-4	1
0	0	1	0	1	4	4.0E-3	1

In order to use the *MCSImporter* PP, the user needs to set the `subType` of a `<PostProcessor>` node:

```
<PostProcessor name='ppName' subType='MCSImporter'/>.
```

Several sub-nodes are available:

- **<expand>**,***bool, required parameter***, expand the set of Basic Events by including all PRA Basic Events and not only the once listed in the MCSs
- **<BElistColumn>**,***string, optional parameter***, if expand is set to True, then this node contains the column of the csv file which contains all the PRA Basic Events

Example:

Listing 2: MCS Importer input example (no expand).

```
<Files>
  <Input name="MCSlistFile" type="MCSlist">MCSlist.csv</Input>
</Files>

<Models>
  <PostProcessor name="MCSImporter" subType="MCSImporter">
    <expand>False</expand>
  </PostProcessor>
</Models>

<Steps>
  <PostProcess name="import">
    <Input    class="Files"        type="MCSlist"
       >MCSlistFile</Input>
    <Model    class="Models"       type="PostProcessor"
       >MCSImporter</Model>
    <Output   class="DataObjects"  type="PointSet"
       >MCS_PS</Output>
  </PostProcess>
</Steps>

<DataObjects>
  <PointSet name="MCS_PS">
    <Input>A,B,C,D,E</Input>
    <Output>MCS_ID,probability,out</Output>
  </PointSet>
</DataObjects>
```

Example:

Listing 3: MCS Importer input example (expanded).

```
<Files>
  <Input name="MCSlistFile" type="MCSlist">MCSlist.csv</Input>
  <Input name="BElistFile"  type="BElist" >BElist.csv</Input>
</Files>

<Models>
  <PostProcessor name="MCSImporter" subType="MCSImporter">
    <expand>False</expand>
  </PostProcessor>
</Models>

<Steps>
  <PostProcess name="import">
    <Input   class="Files"        type="MCSlist"
       >MCSlistFile</Input>
    <Input   class="Files"        type="BElist"
       >BElistFile</Input>
    <Model   class="Models"       type="PostProcessor"
       >MCSimporter</Model>
    <Output  class="DataObjects"  type="PointSet"
       >MCS_PS</Output>
  </PostProcess>
</Steps>

<DataObjects>
  <PointSet name="MCS_PS">
    <Input>A,B,C,D,E,F,G</Input>
    <Output>MCS_ID,probability,out</Output>
  </PointSet>
</DataObjects>
```

15.5.15 ETImporter

The **ETImporter** post-processor has been designed to import Event-Tree (ET) object into RAVEN. Since several ET file formats are available, as of now only the OpenPSA format (see https://open-psa.github.io/joomla1.5/index.php.html) is supported. As an example, the OpenPSA format ET is shown below:

Listing 4: ET in OpenPSA format.

```
<define-event-tree name="eventTree">
    <define-functional-event name="ACC"/>
```

```
    <define-functional-event name="LPI"/>
    <define-functional-event name="LPR"/>
    <define-sequence name="1"/>
    <define-sequence name="2"/>
    <define-sequence name="3"/>
    <define-sequence name="4"/>
    <initial-state>
        <fork functional-event="ACC">
            <path state="0">
                <fork functional-event="LPI">
                    <path state="0">
                        <fork functional-event="LPR">
                            <path state="0">
                                <sequence name="1"/>
                            </path>
                            <path state="+1">
                                <sequence name="2"/>
                            </path>
                        </fork>
                    </path>
                    <path state="+1">
                        <sequence name="3"/>
                    </path>
                </fork>
            </path>
            <path state="+1">
                <sequence name="4"/>
            </path>
        </fork>
    </initial-state>
</define-event-tree>
```

This is performed by saving the structure of the ET (from file) as a **PointSet** (only **PointSet** are allowed), since an ET is a static Boolean logic structure. Each realization in the **PointSet** represents a unique accident sequence of the ET, and the **PointSet** is structured as follows:

- Input variables of the **PointSet** are the branching conditions of the ET. The value of each input variable can be:
  - 0: event did occur (typically upper branch)
  - 1: event did not occur (typically lower branch)
  - -1: event is not queried (no branching occured)

- Output variables of the **PointSet** are the ID of each branch of the ET (i.e., positive integers greater than 0)

Note: that the 0 or 1 values are specified in the `<path state="0">` or `<path state="1">` nodes in the ET OpenPSA file.

Provided this definition, the ET described in Listing 4 will be converted to **PointSet** that is characterized by the following variables:

- Input variables: statusACC, statusLPI, statusLPR
- Output variable: sequence

and the corresponding **PointSet** if the `<expand>` node is set to False is shown in Table 10. If `<expand>` set to True, the corresponding **PointSet** is shown in Table 11.

Table 10: PointSet generated by RAVEN by employing the ET Importer Post-Processor with `<expand>` set to False for the ET of Listing 4.

ACC	LPI	LPR	sequence
0.	0.	0.	1.
0.	0.	1.	2.
0.	1.	-1.	3.
1.	-1.	-1.	4.

Table 11: PointSet generated by RAVEN by employing the ET Importer Post-Processor with `<expand>` set to True for the ET of Listing 4.

ACC	LPI	LPR	sequence
0.	0.	0.	1.
0.	0.	1.	2.
0.	1.	0.	3.
0.	1.	1.	3.
1.	0.	0.	4.
1.	0.	1.	4.
1.	1.	0.	4.
1.	1.	1.	4.

The ETImporter PP supports also:

- links to sub-trees **Note:** If the ET is split in two or more ETs (and thus one file for each ET), then it is only required to list all files in the Step. RAVEN automatically detect links among ETs and merge all of them into a single PointSet.

- by-pass branches
- symbolic definition of outcomes: typically outcomes are defined as either 0 (upper branch) or 1 (lower branch). If instead the ET uses the **success/failure** labels, then they are converted into 0/1 labels **Note:** If the branching condition is not binary or **success/failure**, then the ET Importer Post-Processor just follows the numerical value of the **<state>** attribute of the **<<path>>** node in the ET OpenPSA file.
- symbolic/numerical definition of sequences: if the ET contains a symbolic sequence then a .xml file is generated. This file contains the mapping between the sequences defined in the ET and the numerical IDs created by RAVEN. The file name is the concatenation of the ET name and "_mapping". As an example the file "eventTree_mapping.xml" generated by RAVEN:

```
<map Tree="eventTree">
  <sequence ID="0">seq_1</sequence>
  <sequence ID="1">seq_2</sequence>
  <sequence ID="2">seq_3</sequence>
  <sequence ID="3">seq_4</sequence>
</map>
```

  contains the mapping of four sequences defined in the ET (seq_1,seq_2,seq_3,seq_4) with the IDs generated by RAVEN (0,1,2,3). Note that if the sequences defined in the ET are both numerical and symbolic then they are all mapped.
- The ET can contain a branch that is defined as a separate block in the **<define-branch>** node and it is replicated in the ET; in such case RAVEN automatically replicate such branch when generating the PointSet.

The **<collect-formula>** are not considered since this node is used to connect the Boolean formulae generated by the Fault-Trees to the branch (i.e., fork) point.

In order to use the *ETImporter* PP, the user needs to set the **subType** of a **<PostProcessor>** node:

```
<PostProcessor name='ppName' subType='ETImporter'/>.
```

Several sub-nodes are available:

- **<fileFormat>**, ***string, required field***, specifies the format of the file that contains the ET structure (supported format: OpenPSA).
- **<expand>**,***bool, required parameter***, expand the ET branching conditions for all branches even if they are not queried

Example:

Listing 5: ET Importer input example.

```
<Files>
  <Input name="eventTreeTest" type="">eventTree.xml</Input>
</Files>

<Models>
  ...
  <PostProcessor name="ETimporter" subType="ETImporter">
    <fileFormat>OpenPSA</fileFormat>
    <expand>False</expand>
  </PostProcessor>
  ...
</Models>

<Steps>
  ...
  <PostProcess name="import">
    <Input    class="Files"        type=""
       >eventTreeTest</Input>
    <Model    class="Models"       type="PostProcessor"
       >ETimporter</Model>
    <Output   class="DataObjects"  type="PointSet"
       >ET_PS</Output>
  </PostProcess>
  ...
</Steps>

<DataObjects>
  ...
  <PointSet name="ET_PS">
    <Input>ACC,LPI,LPR</Input>
    <Output>sequence</Output>
  </PointSet>
  ...
</DataObjects>
```

15.5.16 FTImporter

The **FTImporter** post-processor has been designed to import Fault-Tree (FT) object into RAVEN. Since several FT file formats are available, as of now only the OpenPSA format (see https://open-psa.github.io/joomla1.5/index.php.html) is supported. As an example, the FT in OpenPSA format is shown in Listing 6.

Listing 6: FT in OpenPSA format.

```
<opsa-mef>
    <define-fault-tree name="FT">
        <define-gate name="TOP">
            <or>
                <gate name="G1"/>
                <gate name="G2"/>
                <gate name="G3"/>
            </or>
        </define-gate>
        <define-component name="A">
            <define-gate name="G1">
                <and>
                    <basic-event name="BE1"/>
                    <basic-event name="BE2"/>
                </and>
            </define-gate>
            <define-gate name="G2">
                <and>
                    <basic-event name="BE1"/>
                    <basic-event name="BE3"/>
                </and>
            </define-gate>
            <define-basic-event name="BE1">
                <float value="1.2e-3"/>
            </define-basic-event>
            <define-component name="B">
                <define-basic-event name="BE2">
                    <float value="2.4e-3"/>
                </define-basic-event>
                <define-basic-event name="BE3">
                    <float value="5.2e-3"/>
                </define-basic-event>
            </define-component>
        </define-component>
        <define-component name="C">
```

```
            <define-gate name="G3">
                <and>
                    <basic-event name="BE1"/>
                    <basic-event name="BE4"/>
                </and>
            </define-gate>
            <define-basic-event name="BE4">
                <float value="1.6e-3"/>
            </define-basic-event>
        </define-component>
    </define-fault-tree>
</opsa-mef>
```

This is performed by saving the structure of the FT (from file) as a **PointSet** (only **PointSet** are allowed).

Each Point in the PointSet represents a unique combination of the basic events. The PointSet is structured as follows: input variables are the basic events, output variable is the top event of the FT. The value for each input and output variable can have the following values:

- 0: False
- 1: True

Provided this definition, the FT model of Listing 6 can be converted to **PointSet** that is characterized by these variables:

- Input variables: BE1, BE2, BE3, BE4
- Output variable: out

and it is structured is shown in Table 12.

In order to use the *FTImporter* PP, the user needs to set the `subType` of a `<PostProcessor>` node:

`<PostProcessor name='ppName' subType='FTImporter'/>`.

Several sub-nodes are available:

- `<fileFormat>`, ***string, required field***, specifies the format of the file that contains the FT structure (supported format: OpenPSA).

Table 12: PointSet generated by RAVEN by employing the FT Importer Post-Processor for the FT of Listing 6.

BE1	BE2	BE3	BE4	TOP
0.	0.	0.	0.	0.
0.	0.	0.	1.	0.
0.	0.	1.	0.	0.
0.	0.	1.	1.	0.
0.	1.	0.	0.	0.
0.	1.	0.	1.	0.
0.	1.	1.	0.	0.
0.	1.	1.	1.	0.
1.	0.	0.	0.	0.
1.	0.	0.	1.	1.
1.	0.	1.	0.	1.
1.	0.	1.	1.	1.
1.	1.	0.	0.	1.
1.	1.	0.	1.	1.
1.	1.	1.	0.	1.
1.	1.	1.	1.	1.

- **`<topEventID>`**,***string, required parameter***, the name of the top event of the FT

The example of FTImporter PostProcessor is shown in Listing 7

Listing 7: FT Importer input example.

```
<Files>
  <Input name="faultTreeTest"
     type="">FTimporter_not.xml</Input>
</Files>

<Models>
  ...
  <PostProcessor name="FTimporter" subType="FTImporter">
    <fileFormat>OpenPSA</fileFormat>
    <topEventID>TOP</topEventID>
  </PostProcessor>
  ...
</Models>

<Steps>
  ...
  <PostProcess name="import">
```

```
    <Input    class="Files"         type=""
      >faultTreeTest</Input>
    <Model    class="Models"        type="PostProcessor"
      >FTimporter</Model>
    <Output   class="DataObjects"   type="PointSet"
      >FT_PS</Output>
  </PostProcess>
  ...
</Steps>

<DataObjects>
  ...
  <PointSet name="FT_PS">
    <Input>BE1,BE2,BE3,BE4</Input>
    <Output>TOP</Output>
  </PointSet>
  ...
</DataObjects>
```

Important notes and capabilities:

- If the FT is split in two or more FTs (and thus one file for each FT), then it is only required to list all files in the Step. RAVEN automatically detect links among FTs and merge all of them into a single PointSet.
- Allowed gates: AND, OR, NOT, ATLEAST, CARDINALITY, IFF, imply, NAND, NOR, XOR
- If an house-event is defined in the FT:

Listing 8: FT Importer input example: house-event.

```
<opsa-mef>
    <define-fault-tree name="FT">
        <define-gate name="TOP">
            <or>
                <basic-event name="BE1"/>
                <basic-event name="BE2"/>
                <house-event name="HE1"/>
            </or>
        </define-gate>
        <define-house-event name="HE1">
                <constant value="true"/>
        </define-house-event>
```

```
    </define-fault-tree>
</opsa-mef>
```

then the HE1 is not part of the PointSet (value is fixed)

15.5.17 Metric

The **Metric** post-processor is specifically used to calculate the distance values among points from PointSets and histories from HistorySets, while the **Metrics** block (See Chapter 17) allows the user to specify the similarity/dissimilarity metrics to be used in this post-processor. Both **PointSet** and **HistorySet** can be accepted by this post-processor. If the name of given variable is unique, it can be used directly, otherwise the variable can be specified with $DataObjectName|InputOrOutput|VariableName$ like other places in RAVEN. Some of the Metrics also accept distributions to calculate the distance against. These are specified by using the name of the distribution. In order to use the *Metric* PP, the user needs to set the `subType` of a `<PostProcessor>` node:

`<PostProcessor name='ppName' subType='Metric'/>`.

Several sub-nodes are available:

- `<Features>`, ***comma separated string, required field***, specifies the names of the features. This xml-node accepts the following attribute:
  - `type`, ***required string attribute***, the type of provided features. Currently only accept 'variable'.
- `<Targets>`, ***comma separated string, required field***, contains a comma separated list of the targets. **Note:** Each target is paired with a feature listed in xml node `<Features>`. In this case, the number of targets should be equal to the number of features. This xml-node accepts the following attribute:
  - `type`, ***required string attribute***, the type of provided features. Currently only accept 'variable'.
- `<multiOutput>`, ***optional string attribute***, only used when **HistorySet** is used as input. Defines aggregating of time-dependent metrics' calculations. Available options include: **mean, max, min, raw_values** over the time. For example, when 'mean' is used, the metrics' calculations will be averaged over the time. When 'raw_values' is used, the full set of metrics' calculations will be dumped.
  Default: raw_values
- `<weight>`, ***comma separated floats, optional field***, when 'mean' is provided for `<multiOutput>`, the user can specify the weights that can be used for the average calculation of all outputs.

- **<pivotParameter>**, ***optional string attribute***, only used when **HistorySet** is used as input. The pivotParameter for given metrics' calculations.
  Default: time

- **<Metric>**, ***string, required field***, specifies the **Metric** name that is defined via **Metrics** entity. In this xml-node, the following xml attributes need to be specified:

  – **class**, ***required string attribute***, the class of this metric (e.g. Metrics)

  – **type**, ***required string attribute***, the sub-type of this Metric (e.g. SKL, Minkowski)

Example:

```
<Simulation>
 ...
  <Models>
    ...
    <PostProcessor name="pp1" subType="Metric">
      <Features type="variable">ans</Features>
      <Targets type="variable">ans2</Targets>
      <Metric class="Metrics" type="SKL">euclidean</Metric>
      <Metric class="Metrics" type="SKL">cosine</Metric>
      <Metric class="Metrics" type="SKL">manhattan</Metric>
      <Metric class="Metrics"
         type="ScipyMetric">braycurtis</Metric>
      <Metric class="Metrics"
         type="ScipyMetric">canberra</Metric>
      <Metric class="Metrics"
         type="ScipyMetric">correlation</Metric>
      <Metric class="Metrics"
         type="ScipyMetric">minkowski</Metric>
    </PostProcessor>
    ...
  </Models>
 ...
</Simulation>
```

In order to access the results from this post-processor, RAVEN will define the variables as "MetricName" + "_" + "TargetVariableName" + "_" + "FeatureVariableName" to store the calculation results, and these variables are also accessible by the users through RAVEN entities **DataObjects** and **OutStreams**. **Note:** We will replace "—" in "TargetVariableName" and "FeatureVariableName" with "_". In previous example, variables such as *euclidean_ans2_ans, cosine_ans2_ans, poly_ans2_ans* are accessible by the users.

15.5.18 CrossValidation

The **CrossValidation** post-processor is specifically used to evaluate estimator (i.e. ROMs) performance. Cross-validation is a statistical method of evaluating and comparing learning algorithms by dividing data into two portions: one used to 'train' a surrogate model and the other used to validate the model, based on specific scoring metrics. In typical cross-validation, the training and validation sets must crossover in successive rounds such that each data point has a chance of being validated against the various sets. The basic form of cross-validation is k-fold cross-validation. Other forms of cross-validation are special cases of k-fold or involve repeated rounds of k-fold cross-validation. **Note:** It is important to notice that this post-processor currently can only accept **PointSet** data object. In order to use the *CrossValidation* PP, the user needs to set the **subType** of a **<PostProcessor>** node:

<PostProcessor name='ppName' subType='CrossValidation'/>.

Several sub-nodes are available:

- **<SciKitLearn>**, ***string, required field***, the subnodes specifies the necessary information for the algorithm to be used in the post-processor. 'SciKitLearn' is based on algorithms in SciKit-Learn library, and currently it performs cross-validation over **PointSet** only.
- **<Metric>**, ***string, required field***, specifies the **Metric** name that is defined via **Metrics** entity. In this xml-node, the following xml attributes need to be specified:
  - **class**, ***required string attribute***, the class of this metric (e.g. Metrics)
  - **type**, ***required string attribute***, the sub-type of this Metric (e.g. SKL, Minkowski)

  Note: Currently, cross-validation post-processor only accepts **<SKL>** metrics with **<metricType>** **'mean_absolute_error'**, **'explained_variance_score'**, **'r2_score'**, **'mean_squared_e** and **'median_absolute_error'**.

Example:

```
<Simulation>
 ...
  <Files>
    <Input name="output_cv" type="">output_cv.xml</Input>
    <Input name="output_cv.csv" type="">output_cv.csv</Input>
  </Files>
  <Models>
    ...
    <ROM name="surrogate" subType="SciKitLearn">
      <SKLtype>linear_model|LinearRegression</SKLtype>
```

```
      <Features>x1,x2</Features>
      <Target>ans</Target>
      <fit_intercept>True</fit_intercept>
      <normalize>True</normalize>
    </ROM>
    <PostProcessor name="pp1" subType="CrossValidation">
        <SciKitLearn>
            <SKLtype>KFold</SKLtype>
            <n_splits>3</n_splits>
            <shuffle>False</shuffle>
        </SciKitLearn>
        <Metric class="Metrics" type="SKL">m1</Metric>
    </PostProcessor>
    ...
  </Models>
  <Metrics>
    <SKL name="m1">
      <metricType>mean_absolute_error</metricType>
    </SKL>
  </Metrics>
  <Steps>
    <PostProcess name="PP1">
        <Input class="DataObjects"
           type="PointSet">outputDataMC</Input>
        <Input class="Models" type="ROM">surrogate</Input>
        <Model class="Models" type="PostProcessor">pp1</Model>
        <Output class="Files" type="">output_cv</Output>
        <Output class="Files" type="">output_cv.csv</Output>
    </PostProcess>
  </Steps>
 ...
</Simulation>
```

In order to access the results from this post-processor, RAVEN will define the variables as "cv" + "_" + "MetricName" + "_" + "ROMTargetVariable" to store the calculation results, and these variables are also accessible by the users through RAVEN entities **DataObjects** and **OutStreams**. In previous example, variable *cv_m1_ans* are accessible by the users.

15.5.18.1 SciKitLearn

The algorithm for cross-validation is chosen by the subnode **<SKLtype>** under the parent node **<SciKitLearn>**. In addition, a special subnode **<average>** can be used to obtain the average cross validation results.

- **<SKLtype>**, ***string, required field***, contains a string that represents the cross-validation algorithm to be used. As mentioned, its format is:

  <SKLtype>algorithm**</SKLtype>**.

- **<average>**, ***boolean, optional field***, if 'True', dump the average cross validation results into the output files.

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

15.5.18.2 K-fold

KFold divides all the samples in k groups of samples, called folds (if $k = n$, this is equivalent to the **Leave One Out** strategy), of equal sizes (if possible). The prediction function is learned using $k - 1$ folds, and fold left out is used for test. In order to use this algorithm, the user needs to set the subnode: **<SKLtype>**KFold**</SKLtype>**. In addition to this XML node, several others are available:

- **<n_splits>**, ***integer, optional field***, number of folds, must be at least 2.
  Default: 3

- **<shuffle>**, ***boolean, optional field***, whether to shuffle the data before splitting into batches.

- **<random_state>**, ***integer, optional field***, when shuffle=True, pseudo-random number generator state used for shuffling. If not present, use default numpy RNG for shuffling.

15.5.18.3 Stratified k-fold

StratifiedKFold is a variation of *k-fold* which returns stratified folds: each set contains approximately the same percentage of samples of each target class as the complete set. In order to use this algorithm, the user needs to set the subnode:

<SKLtype>StratifiedKFold**</SKLtype>**.

In addition to this XML node, several others are available:

- **<labels>**, ***list of integers, (n_samples), required field***, contains a label for each sample.
- **<n_splits>**, ***integer, optional field***, number of folds, must be at least 2.
  Default: 3
- **<shuffle>**, ***boolean, optional field***, whether to shuffle the data before splitting into batches.
- **<random_state>**, ***integer, optional field***, when shuffle=True, pseudo-random number generator state used for shuffling. If not present, use default numpy RNG for shuffling.

15.5.18.4 Label k-fold

LabelKFold is a variation of *k-fold* which ensures that the same label is not in both testing and training sets. This is necessary for example if you obtained data from different subjects and you want to avoid over-fitting (i.e., learning person specific features) by testing and training on different subjects. In order to use this algorithm, the user needs to set the subnode:

<SKLtype>LabelKFold**</SKLtype>**.

In addition to this XML node, several others are available:

- **<labels>**, ***list of integers with length (n_samples,), required field***, contains a label for each sample. The folds are built so that the same label does not appear in two different folds.
- **<n_splits>**, ***integer, optional field***, number of folds, must be at least 2.
  Default: 3

15.5.18.5 Leave-One-Out - LOO

LeaveOneOut (or LOO) is a simple cross-validation. Each learning set is created by taking all the samples except one, the test set being the sample left out. Thus, for n samples, we have n different training sets and n different tests set. This is cross-validation procedure does not waste much data as only one sample is removed from the training set. In order to use this algorithm, the user needs to set the subnode:

<SKLtype>LeaveOneOut**</SKLtype>**.

15.5.18.6 Leave-P-Out - LPO

LeavePOut is very similar to **LeaveOneOut** as it creates all the possible training/test sets by removing p samples from the complete set. For n samples, this produces $\binom{n}{p}$ train-test pairs. Unlike

LeaveOneOut and **KFold**, the test sets will overlap for $p > 1$. In order to use this algorithm, the user needs to set the subnode:

<SKLtype>LeavePOut**</SKLtype>**.

In addition to this XML node, several others are available:

- **<p>**, ***integer, required field***, size of the test sets

15.5.18.7 Leave-One-Label-Out - LOLO

LeaveOneLabelOut (LOLO) is a cross-validation scheme which holds out the samples according to a third-party provided array of integer labels. This label information can be used to encode arbitrary domain specific pre-defined cross-validation folds. Each training set is thus constituted by all samples except the ones related to a specific label. In order to use this algorithm, the user needs to set the subnode:

<SKLtype>LeaveOneLabelOut**</SKLtype>**.

In addition to this XML node, several others are available:

- **<labels>**, ***list of integers, (n_samples,), required field***, arbitrary domain-specific stratificatioin of the data to be used to draw the splits.

15.5.18.8 Leave-P-Label-Out

LeavePLabelOut is imilar as *Leave-One-Label-Out*, but removes samples related to P labels for each training/test set. In order to use this algorithm, the user needs to set the subnode:

<SKLtype>LeavePLabelOut**</SKLtype>**.

In addition to this XML node, several others are available:

- **<labels>**, ***list of integers, (n_samples,), required field***, arbitrary domain-specific stratificatioin of the data to be used to draw the splits.
- **<n_groups>**, ***integer, optional field***, number of samples to leave out in the test split.

15.5.18.9 ShuffleSplit

ShuffleSplit iterator will generate a user defined number of independent train/test dataset splits. Samples are first shuffled and then split into a pair of train and test sets. it is possible to control the randomness for reproducibility of the results by explicitly seeding the **<random_state>** pseudo random number generator. In order to use this algorithm, the user needs to set the subnode:

<SKLtype>ShuffleSplit**</SKLtype>**.

In addition to this XML node, several others are available:

- **<n_splits>**, ***integer, optional field***, number of re-shuffling and splitting iterations
  Default: 10.
- **<test_size>**, ***float or integer, optional field***, if float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the test split.
  Default: 0.1 If integer, represents the absolute number of test samples. If not present, the value is automatically set to the complement of the train size.
- **<train_size>**, ***float or integer, optional field***, if float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the train split. If integer, represents the absolute number of train samples. If not present, the value is automatically set to the complement of the test size.
- **<random_state>**, ***integer, optional field***, when shuffle=True, pseudo-random number generator state used for shuffling. If not present, use default numpy RNG for shuffling.

15.5.18.10 Label-Shuffle-Split

LabelShuffleSplit iterator behaves as a combination of **ShuffleSplit** and **LeavePLabelOut**, and generates a sequence of randomized partitions in which a subset of labels are held out for each split. In order to use this algorithm, the user needs to set the subnode:

<SKLtype>LabelShuffleSplit**</SKLtype>**.

In addition to this XML node, several others are available:

- **<labels>**, ***list of integers, (n_samples)***, labels of samples.
- **<n_splits>**, ***integer, optional field***, number of re-shuffling and splitting iterations
  Default: 10.

- **<test_size>**, ***float or integer, optional field***, if float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the test split.
  Default: 0.1 If integer, represents the absolute number of test samples. If not present, the value is automatically set to the complement of the train size.

- **<train_size>**, ***float or integer, optional field***, if float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the train split. If integer, represents the absolute number of train samples. If not present, the value is automatically set to the complement of the test size.

- **<random_state>**, ***integer, optional field***, when shuffle=True, pseudo-random number generator state used for shuffling. If not present, use default numpy RNG for shuffling.

15.5.19 DataClassifier

The **DataClassifier** post-processor is specifically used to classify the data stored in the DataObjects. It accepts two DataObjects, one is used as the classifier which must be a **PointSet**, the other one, i.e. **PointSet** or **HistorySet**, is used as the input DataObject to be classified. In order to use the *DataClassifier* PP, the user needs to set the **subType** of a **<PostProcessor>** node:

```
<PostProcessor name='ppName' subType='DataClassifier'/>.
```

Several sub-nodes are available:

- **<label>**, ***string, required field***, the name of the label that are used for the classifier. This label must exist in the DataObject that is used as the classifer. This name will also be used as the label name for the DataObject that is classified.

- **<variable>**, ***required, xml node***. In this node, the following attribute should be specified:

  - **name**, ***required, string attribute***, the variable name, which should be exist in the DataObject that is used as classifier.

  and the following sub-node should also be specified:

  - **<Function>**, ***string, required field***, this function creates the mapping from input DataObject to the Classifier.

    * **class**, ***string, required field***, the class of this function (e.g. Functions)
    * **type**, ***string, required field***, the type of this function (e.g. external)

In order to use this post-processor, the users need to specify two different DataObjects, i.e.

```
  <DataObjects>
    <PointSet name="ET_PS">
      <Input>ACC, LPI</Input>
      <Output>sequence</Output>
    </PointSet>
    <PointSet name="sim_PS">
      <Input>ACC_status, LPI_status</Input>
      <Output>out</Output>
    </PointSet>
  </DataObjects>
```

The first data object "ET_PS" contains the event tree with input variables "ACC, LPI" and output label "sequence". This data object will be used to classify the data in the second data object "sim_PS". The results will be stored in the output data object with the same label "sequence". Since these two data objects contain different inputs, **<Functions>** will be used to create the maps between the inputs:

```
  <Functions>
    <External file="func_ACC.py" name="func_ACC">
      <variable>ACC_status</variable>
    </External>
    <External file="func_LPI.py" name="func_LPI">
      <variable>LPI_status</variable>
    </External>
  </Functions>
```

The inputs to these functions are the inputs of the data object that will be classified, and the outputs of these functions are the inputs of data object that is used as the classifier.

Example Python Function for "func_ACC.py"

```
def evaluate(self):
  return self.ACC_status
```

Example Python Function for "func_LPI.py"

```
def evaluate(self):
  return self.LPI_status
```

Note: All the functions that are used to create the maps should be include the "evaluate" method.

The **DataClassifier** post-processor is specifically used to classify the data stored in the DataOb-

jects. It accepts two DataObjects, one is used as the classifier which should be always **PointSet**, the other one, i.e. either **PointSet** or **HistorySet** is used as the input DataObject to be classified.

The **DataClassifier** is provided below:

```
    <PostProcessor name="ET_Classifier" subType="DataClassifier">
      <label>sequence</label>
      <variable name='ACC'>
        <Function class="Functions"
           type="External">func_ACC</Function>
      </variable>
      <variable name='LPI'>
        <Function class="Functions"
           type="External">func_LPI</Function>
      </variable>
    </PostProcessor>
```

The definitions for the XML nodes can be found in the RAVEN user manual. The label "sequence" and the variables "ACC, LPI" should be exist in the data object that is used as the classifier, while the functions "func_ACC, func_LPI" are used to map relationships between the input data objects.

The classification can be achieved via the **<Steps>** as shown below:

```
<Simulation>
 ...
  <Steps>
    <PostProcess name="classify">
      <Input    class="DataObjects"  type="PointSet"
         >ET_PS</Input>
      <Input    class="DataObjects"  type="PointSet"
         >sim_PS</Input>
      <Model    class="Models"       type="PostProcessor"
         >ET_Classifier</Model>
      <Output   class="DataObjects"  type="PointSet"
         >sim_PS</Output>
    </PostProcess>
  </Steps>
 ...
</Simulation>
```

15.5.20 ValueDuration

The **<ValueDuration>** postprocessor is a tool to construct a particular kind of histogram, where the independent variable is the number of times a variable exceeds a particular value, and the dependent variable is the values themselves. An example of this is the Load Duration Curve in energy modeling. This approach is similar to that used in Lebesgue integration. Note that for each realization in the input **<HistorySet>**, a seperate load duration curve will be created for each target.

The **<ValueDuration>** postprocessor can only act on **<HistorySet>** data objects, and generates a **<HistorySet>** in return. Two output variables are created for each **target**: **'counts_x'** and **'bins_x'**, where **'x'** is replaced by the name of the target. These must be specified in the output data object in order to be collected.

To plot a traditional Load Duration Curve, the x-axis should be the bins variable, and the y-axis should be the counts variable.

In order to use the *ValueDuration* PP, the user needs to set the **subType** of a **<PostProcessor>** node:

<PostProcessor name='ppName' subType='ValueDuration'/>.

Several sub-nodes are available:

- **<target>**, ***comma separated strings, required field***, specifies the names of the target(s) for which Value Duration histograms should be generated.
- **<bins>**, ***integer, required field***, specifies the number of bins that the values of the targets should be counted into.

Example:

```
<Simulation>
 ...
  <Models>
    ...
    <PostProcessor name="pp" subType="ValueDuration">
      <target>x, y</target>
      <bins>100</bins>
    </PostProcessor>
    ...
  </Models>
 ...
</Simulation>
```

15.5.21 FastFourierTransform

The **<FastFourierTransform>** postprocessor provides access to the Numpy fast fourier transform function `numpy.fft.fft` and provides the frequencies, periods, and amplitudes from performing the transform. The periods are simply the inverse of the frequencies, and the frequency units are the deltas between pivot values in the provided input. For example, if data is collected every 3600 seconds, the units of frequency are per-hour. This postrpocessor expects uniformly-spaced pivot values. Note that for each realization in the input data object, a separate fft will be created for each target.

The **<FastFourierTransform>** postprocessor can act on any target in a DataObject that depends on a single index, and generates three histories per sample per target: an independent variable **'target_fft_frequency'**, and two dependent values **'target_fft_period'** and **'target_fft_amplitude'**, which both depend on the frequency by default. In all three outputs, *target* is replaced by the name of the target for which the fft was requested.

In order to use the *FastFourierTransform* PP, the user needs to set the **subType** of a **<PostProcessor>** node:

<PostProcessor name='ppName' subType='FastFourierTransform'/>.

Several sub-nodes are available:

- **<target>**, ***comma separated strings, required field***, specifies the names of the target(s) for which the fast Fourier transform should be calculated.

Example:

```
<Simulation>
 ...
  <Models>
    ...
    <PostProcessor name="pp" subType="FastFourierTransform">
      <target>x, y</target>
    </PostProcessor>
    ...
  </Models>
 ...
</Simulation>
```

15.5.22 SampleSelector

The **<SampleSelector>** postprocessor is a tool to select a row from a dataset, depending on different criteria. The different criteria that can be used are listed below.

The **<SampleSelector>** postprocessor can act on any **<DataObjects>**, and generates a **<DataObject>** with a single realization in return.

In order to use the *SampleSelector* PP, the user needs to set the **subType** of a **<PostProcessor>** node:

<PostProcessor name='ppName' subType='SampleSelector'/>.

Several sub-nodes are available:

- **<criterion>**, ***string, required field***, specifies the criterion to select the realization from the input DataObject. Options are as follows:
  - **'min'**, choose the realization that has the lowest value of the **<target>** variable. The target must be scalar.
  - **'max'**, choose the realization that has the highest value of the **<target>** variable. The target must be scalar.
  - **'index'**, choose the realization that has the provided index. The index must be an integer and is zero-based, meaning the first entry is at index 0, the second entry is at index 1, etc. The realization order is taken from the order in which they were entered originally into the input DataObject. If this option is used, the **<criterion>** node must have an **value** attribute that gives the index.
- **<target>**, ***string, optional field***, required if the criterion targets a particular variable (such as the minimum and maximum criteria). Specifies the name of the target for which the criterion should be evaluated.

Example:

```
<Simulation>
 ...
  <Models>
    ...
    <PostProcessor name="select_min" subType="SampleSelector">
      <target>x</target>
      <criterion>min</criterion>
    </PostProcessor>
    ...
```

```
    <PostProcessor name="select_index" subType="SampleSelector">
      <criterion value='3'>index</criterion>
    </PostProcessor>
    ...
  </Models>
 ...
</Simulation>
```

15.5.23 EconomicRatio

The **<EconomicRatio>** post-processor provides the economic metrics from the percent change period return of the asset or strategy that is given as an input. These metrics measure the risk-adjusted returns. In order to use the *EconomicRatio* PP, the user needs to set the **subType** of a **<PostProcessor>** node:

<PostProcessor name='ppName' subType='EconomicRatio'/>.

Several sub-nodes are available:

- **<"metric">**, ***comma separated string or node list, required field***, specifications for the metric to be calculated. The name of each node is the requested metric. The text of the node is a comma-separated list of the parameters for which the metric should be calculated.

  Currently the scalar quantities available for request are:

  - **sharpeRatio**: the Sharpe Ratio, measures the performance of an investment. It is defined as the historical returns of the investment, divided by the standard deviation of the investment(Volatility).
  - **sortinoRatio**: the Sortino ratio, measures the risk-adjusted return of an investment asset. Discounts the excess return of a portfolio above a target threshold by the volatility of downside returns. If this quantity is inputted as *sortinoRatio* the threshold for separate upside and downside value will assign as 0. Otherwise the user can specify this quantity with a parameter **threshold='X'**, where the **X** represents the requested threshold **median** or **zero**.
  - **gainLossRatio**: the gain-loss ratio, discounts the first-order higher partial moment of a portfolio's returns, by the first-order lower partial moment of a portfolio's returns. If this quantity is inputted as *gainLossRatio* the threshold for separate upside and downside value will assign as 0. Otherwise the user can specify this quantity with a parameter **threshold='X'**, where the **X** represents the requested threshold **median** or **zero**.

– **expectedShortfall**: the expected shortfall (Es) or conditional value at risk (CVaR), the expected return on the portfolio in the worst q of cases. If this quantity is inputted as *ExpectedShortfall* the q value will assign as 5%. Otherwise the user can specify this quantity with a parameter `threshold='X'`, where the `X` represents the requested q value (a floating point value between 0.0 and 1.0)

$$ES_\alpha = -\frac{1}{\alpha}\int_0^\alpha \mathrm{VaR}_\gamma(X)\,d\gamma \tag{41}$$

– **valueAtRisk**: the value at risk for investments. Estimates the maximum possible loss after exclude worse outcomes whose combined probability is at most α. If this quantity is inputted as *ValueAtRisk* the α value will assign as 5%. Otherwise the user can specify this quantity with a parameter `threshold='X'`, where the `X` represents the requested α value (a floating point value between 0.0 and 1.0)

$$\mathrm{VaR}_\alpha(X) = -\inf\left\{x \in \mathbb{R} : F_X(x) > \alpha\right\} = F_Y^{-1}(1-\alpha). \tag{42}$$

This XML node needs to contain the attribute:

– `prefix`, ***required string attribute***, user-defined prefix for the given **metric**. For scalar quantifies, RAVEN will define a variable with name defined as: "prefix" + "_" + "parameter name". For example, if we define "mean" as the prefix for **expectedValue**, and parameter "x", then variable "mean_x" will be defined by RAVEN. For matrix quantities, RAVEN will define a variable with name defined as: "prefix" + "_" + "target parameter name" + "_" + "feature parameter name". For example, if we define "sen" as the prefix for **sensitivity**, target "y" and feature "x", then variable "sen_y_x" will be defined by RAVEN. **Note:** These variable will be used by RAVEN for the internal calculations. It is also accessible by the user through **DataObjects** and **OutStreams**.

Example:

```
<Simulation>
  ...
    <Models>
    ...
    <PostProcessor name="EconomicRatio" subType="EconomicRatio"
       verbosity="debug">
      <sharpeRatio prefix="SR">x0,y0,z0,x,y,z</sharpeRatio>
      <sortinoRatio threshold='zero'
         prefix="stR">x01,y01,x,z</sortinoRatio>
      <sortinoRatio threshold='median'
         prefix="stR2">z01,x0,x01</sortinoRatio>
      <valueAtRisk threshold='0.07'
         prefix="VaR">z01,x0,x01</valueAtRisk>
```

```
      <expectedShortfall threshold='0.99'
         prefix="CVaR">z01,x0,x01</expectedShortfall>
      <gainLossRatio
         prefix="glR">x01,y01,z0,x,y,z</gainLossRatio>
    </PostProcessor>
    ...
  </Models>
  ...
</Simulation>
```

15.6 EnsembleModel

As already mentioned, the **EnsembleModel** is able to combine **Code**(see 15.1), **ExternalModel**(see 15.4) and **ROM**(see 15.4) Models.
It is aimed 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.
Currently this model is able to share information (i.e. data) using **PointSet**, **HistorySet** and **DataSet**

The specifications of a EnsembleModel must be defined within the XML block **`<EnsembleModel>`**. This XML node needs to contain the attributes:

- **`name`**, ***required string attribute***, user-defined name of this EnsembleModel. **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 kept empty.

Within the **`<EnsembleModel>`** XML node, the multiple Models that constitute this EnsembleModel needs to be inputted. Each Model is specified within a **`<Model>`** block (**Note:** each model here specified need to be inputted in the**`<Models>`** main XML block) :

- **`<Model>`**, ***XML node, required parameter***. The text portion of this node needs to contain the name of the Model
  This XML node needs to contain the attributes:
  - **`class`**, ***required string attribute***, the class of this sub-model (e.g. Models)
  - **`type`**, ***required string attribute***, the sub-type of this Model (e.g. ExternalModel, ROM, Code)

  In addition the following XML sub-nodes need to be inputted (or optionally inputted):

- **`<TargetEvaluation>`**, ***string, required field***, represents the container where the output of this Model are stored. From a practical point of view, this XML node must contain the name of a data object defined in the **`<DataObjects>`** block (see Section 12). Currently, the **`<EnsembleModel>`** accept all ***DataObjects'*** types: **PointSet**, **HistorySet** and **DataSet** **Note:** The **`<TargetEvaluation>`** is primary used for input-output identification. If the linked DataObject is not placed as additional output of the Step where the EnsembleModel is used, it will not be filled with the data coming from the calculation and it will be kept empty.
- **`<Input>`**, ***string, required field***, represents the input entities that need to be passed to this sub-model The user can specify as many **`<Input>`** as required by the sub-model. **Note:** All the inputs here specified need to be listed in the Steps where the EnsembleModel is used.
- **`<Output>`**, ***string, optional field***, represents the output entities that need to be linked to this sub-model. **Note:** The **`<Output>`**s here specified are not part of the determination of the EnsembleModel execution but represent an additional storage of results from the sub-models. For example, if the **`<TargetEvaluation>`** is of type PointSet (since just scalar data needs to be transferred to other models) and the sub-model is able to also output history-type data, this Output can be of type HistorySet. Note that the structure of each Output dataObject must include only variables (either input or output) that are defined among the model. As an example, the Output dataObjects cannot contained variables that are defined at the Ensemble model level. The user can specify as many **`<Output>`** (s) as needed. The optional **`<Output>`**s can be of both classes "DataObjects" and "Databases" (e.g. *PointSet*, *HistorySet*, *DataSet*, *HDF5*) **Note: The `<Output>` (s) here specified MUST be listed in the Step in which the EnsembleModel is used.**

It is important to notice that when the EnsembleModel detects a chain of models that evolve in a non-linear system, a Picard's Iteration scheme is activated. In this case, an additional XML sub-node within the main **`<EnsembleModel>`** XML node needs to be specified:

- **`<settings>`**, ***XML node, required parameter (if Picard's activated)***. The body of this sub-node contains the following XML sub-nodes:
  - **`<maxIterations>`**, ***integer, optional field***, maximum number of Picard's iteration to be performed (in case the iteration scheme does not previously converge).
    Default: 30;
  - **`<tolerance>`**, ***float, optional field***, convergence criterion. It represents the L2 norm residue below which the Picard's iterative scheme is considered converged.
    Default: 0.001;
  - **`<initialConditions>`**, ***XML node, required parameter (if Picard's activated)***, Within this sub-node, the initial conditions for the input variables (that are part of a

loop) need to be specified in sub-nodes named with the variable name (e.g. **<varName>**). The body of the **<varName>** contains the value of the initial conditions (scalar or arrays, depending of the type of variable). If an array needs to be inputted, the user can specify the attribute **repeat** and the code is going to repeat for **repeat**-times the value inputted in the body.

– **<initialStartModels>**, ***XML node, only required parameter when Picard's iteration is activated***, specifies the list of models that will be initially executed. **Note:** Do not input this node for non-Picard calculations, otherwise an error will be raised.

Note: It is crucial to understand that the choice of the <DataObject> used as <TargetEvaluation> determines how the data are going to be transferred from a model to the other. If for example the chain of models is $A \rightarrow B$:

- **If model B expects as input scalars and outputs time-series, the <TargetEvaluation> of the model B will be a *HistorySet* and the <TargetEvaluation> of the model A will be either a *PointSet* or a *DataSet* (where the output variables that need to be transferred to the model A are scalars)**
- **If model B expects as input scalars and time-series and outputs time-series or scalars or both, the <TargetEvaluation> of the model B will be a *DataSet* and the <TargetEvaluation> of the model A will be either a *HistorySet* or a *DataSet***
- **If both model A and B expect as input scalars and output scalars, the <TargetEvaluation> of the both models A and B will be *PointSets***

Example (Linear System):

```
<Simulation>
  ...
  <Models>
    ...
    <EnsembleModel name="heatTransferEnsembleModel" subType="">
      <Model class="Models" type="ExternalModel">
        thermalConductivityComputation
        <TargetEvaluation class="DataObjects" type="PointSet">
          thermalConductivityComputationContainer
        </TargetEvaluation>
        <Input class="DataObjects" type="PointSet">
          inputHolder
        </Input>
      </Model>
      <Model class="Models" type="ExternalModel" >
          heatTransfer
```

```
          <TargetEvaluation class="DataObjects" type="PointSet">
            heatTransferContainer
          </TargetEvaluation>
        <Input class="DataObjects" type="PointSet">
          inputHolder
        </Input>
        <Output class="DataObjects" type="HistorySet">
          thisModelLinkedOutput
        </Output>
        <Output class="Databases" type="HDF5">
          thisModelLinkedHDF5
        </Output>
      </Model>
    </EnsembleModel>
    ...
  </Models>
  ...
</Simulation>
```

Example (Non-Linear System):

```
<Simulation>
  ...
  <Models>
    ...
    <EnsembleModel name="heatTransferEnsembleModel" subType="">
      <settings>
        <maxIterations>8</maxIterations>
         <tolerance>0.01</tolerance>
         <initialConditions>
           <!-- the value 0.7 is going to be repeated 10 times
              in order to create  an array for var1 -->
           <var1 repeat="10">0.7</var1>
           <!-- an array for var2 has been inputted -->
           <var2> 0.5 0.3 0.4</var2>
           <!-- a scalar for var3 has been inputted -->
           <var3> 45.0</var3>
         </initialConditions>
      </settings>

      <Model class="Models" type="ExternalModel">
        thermalConductivityComputation
        <TargetEvaluation class="DataObjects" type="PointSet">
```

```
          thermalConductivityComputationContainer
        </TargetEvaluation>
        <Input class="DataObjects" type="PointSet">
          inputHolder
        </Input>
      </Model>
      <Model class="Models" type="ExternalModel" >
          heatTransfer
          <TargetEvaluation class="DataObjects" type="PointSet">
            heatTransferContainer
          </TargetEvaluation>
        <Input class="DataObjects" type="PointSet">
          inputHolder
        </Input>
      </Model>
    </EnsembleModel>
    ...
  </Models>
  ...
</Simulation>
```

15.7 HybridModel

The **HybridModel** is a new *Model* entity. This new Model is able to combine reduced order model (ROMs) and any other high-fidelity Model (i.e. Code, ExternalModel). The ROMs will be trained based on the results from the high-fidelity model. The accuracy of the ROMs will be evaluated based on the cross validation scores, and the validity of the ROMs will be determined via some local validation metrics (**Note:** currently only one metric is available, i.e. CrowdingDistance). After these ROMs are trained, the **HybridModel** can decide which of the Model (i.e the ROMs or high-fidelity model) to be executed based on the accuracy and validity of the ROMs.

Currently this model is only able to share information (i.e. data) using **PointSet**.

The specifications of a HybridModel must be defined within the XML block **`<HybridModel>`**. This XML node needs to contain the attributes:

- **`name`**, ***required string attribute***, user-defined name of this HybridModel. **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 kept empty.

Within the **`<HybridModel>`** XML node, the multiple entities that constitute this HybridModel needs to be inputted.

- **`<Model>`**, ***XML node, required parameter***. The text portion of this node needs to contain the name of the Model This XML node needs to contain the attributes:
  - **`class`**, ***required string attribute***, the main "class" of the Model.
  - **`type`**, ***required string attribute***, the sub-type of the Model.
- **`<ROM>`**, ***XML node, required parameter***. The text portion of this node needs to contain the name of the ROM The user can specify as many **`<ROM>`** as required by the **`<Model>`**. **Note:** The outputs of each ROM should be different, and the total set of ROMs' outputs should be the same as the set of *Model's* outputs. This XML node needs to contain the attributes:
  - **`class`**, ***required string attribute***, the main "class" of the Model.
  - **`type`**, ***required string attribute***, the sub-type of the Model.
- **`<CV>`**, ***XML node, required parameter***. The text portion of this node needs to contain the name of the **`<PostProcessor>`** with **`subType`** "CrossValidation". This XML node needs to contain the attributes:
  - **`class`**, ***required string attribute***, the main "class" of the Model.
  - **`type`**, ***required string attribute***, the sub-type of the Model.
- **`<TargetEvaluation>`**, ***XML node, required parameter***. The text portion of this node needs to contain the name of a data object defined in the **`<DataObjects>`** block. **Note:** currently only accept data object with type "PointSet". The **`<TargetEvaluation>`** is primary used for training ROMs. **Note:** The linked DataObject should be placed as additional output of the Step where the **HybridModel** is used. This XML node needs to contain the attributes:
  - **`class`**, ***required string attribute***, the main "class" of the DataObjects.
  - **`type`**, ***required string attribute***, the sub-type of the DataObjects.

An additional XML sub-node within the main **`<HybridModel>`** XML node needs to be specified:

- **`<settings>`**, ***XML node, optional parameter***. The body of this sub-node contains the following XML sub-nodes:
  - **`<minInitialTrainSize>`**, ***integer, optional field***, the minimum initial number of high-fidelity model runs before starting train the ROMs.
    Default: 10;

- **<tolerance>**, ***float, optional field***, ROMs convergence criterion indicates the displacement from the optimum results of cross validation. In other words, small tolerance indicates tight convergence criterion of the ROMs, while large tolerance indicates loose convergence criterion of the ROMs. **Note:** Currently, this tolerance can be only used for cross validations with SKL Metrics: *explained_variance_score*, *r2_score*, *median_absolute_error*, *mean_squared_error* and *mean_absolute_error*.
  Default: 0.01;
- **<maxTrainSize>**, ***XML node, optional field***, the maximum size of training set of ROMs.
  Default: 1.0E6

- **<validationMethod>**, ***XML node, optional parameter***. The validity methods that are used to determine which model to run (i.e. ROMs or high-fidelity Model). This XML node needs to contain the attributes:

  - **name**, ***required string attribute***, user-defined name of this **<validationMethod>**. **Note:** Currently, only one method is available, ie. "CrowdingDistance".

  The body of this sub-node contains the following XML sub-nodes:

  - **<threshold>**, ***XML node, required field***, the threshold that is used for "CrowdingDistance" method.

Example (ExternalModel):

```
<Simulation>
  ...
  <Metrics>
    <SKL name="m1">
      <metricType>mean_absolute_error</metricType>
    </SKL>
  </Metrics>

  <Models>
    <ExternalModel ModuleToLoad="EM2linear"
       name="thermalConductivityComputation" subType="">
      <variables>leftTemperature,rightTemperature,k,averageTemperature</var
    </ExternalModel>
    <ROM name="knr" subType="SciKitLearn">
      <SKLtype>neighbors|KNeighborsRegressor</SKLtype>
      <Features>leftTemperature, rightTemperature</Features>
      <Target>k</Target>
      <n_neighbors>5</n_neighbors>
      <weights>uniform</weights>
```

```
      <algorithm>auto</algorithm>
      <leaf_size>30</leaf_size>
      <metric>minkowski</metric>
      <p>2</p>
    </ROM>
    <PostProcessor name="pp1" subType="CrossValidation">
        <SciKitLearn>
            <SKLtype>KFold</SKLtype>
            <n_splits>10</n_splits>
            <shuffle>False</shuffle>
        </SciKitLearn>
        <Metric class="Metrics" type="SKL">m1</Metric>
    </PostProcessor>
    <HybridModel name="hybrid" subType="">
        <Model class="Models"
           type="ExternalModel">thermalConductivityComputation</Model>
        <ROM class="Models" type="ROM">knr</ROM>
        <TargetEvaluation class="DataObjects"
           type="PointSet">thermalConductivityComputationContainer</TargetEv
        <CV class="Models" type="PostProcessor">pp1</CV>
        <settings>
            <tolerance>0.01</tolerance>
            <trainStep>1</trainStep>
            <maxTrainSize>1000</maxTrainSize>
            <initialTrainSize>10</initialTrainSize>
        </settings>
        <validationMethod name="CrowdingDistance">
            <threshold>0.2</threshold>
        </validationMethod>
    </HybridModel>
  </Models>
  ...
</Simulation>
```

Example (Code):

```
<Simulation>
  ...
  <Metrics>
    <SKL name="m1">
      <metricType>mean_absolute_error</metricType>
    </SKL>
  </Metrics>
```

```
<Models>
  <Code name="poly" subType="GenericCode">
    <executable>runCode/poly_inp_io.py</executable>
    <clargs arg="python" type="prepend"/>
    <clargs arg="-i" extension=".one" type="input"/>
    <fileargs arg="aux" extension=".two" type="input"/>
    <fileargs arg="output" type="output"/>
    <prepend>python</prepend>
  </Code>
  <ROM name="knr" subType="SciKitLearn">
    <SKLtype>neighbors|KNeighborsRegressor</SKLtype>
    <Features>x, y</Features>
    <Target>poly</Target>
    <n_neighbors>5</n_neighbors>
    <weights>uniform</weights>
    <algorithm>auto</algorithm>
    <leaf_size>30</leaf_size>
    <metric>minkowski</metric>
    <p>2</p>
  </ROM>
  <PostProcessor name="pp1" subType="CrossValidation">
      <SciKitLearn>
          <SKLtype>KFold</SKLtype>
          <n_splits>10</n_splits>
          <shuffle>False</shuffle>
      </SciKitLearn>
      <Metric class="Metrics" type="SKL">m1</Metric>
  </PostProcessor>
  <HybridModel name="hybrid" subType="">
      <Model class="Models" type="Code">poly</Model>
      <ROM class="Models" type="ROM">knr</ROM>
      <TargetEvaluation class="DataObjects"
         type="PointSet">samples</TargetEvaluation>
      <CV class="Models" type="PostProcessor">pp1</CV>
      <settings>
          <tolerance>0.1</tolerance>
          <trainStep>1</trainStep>
          <maxTrainSize>1000</maxTrainSize>
          <initialTrainSize>10</initialTrainSize>
      </settings>
      <validationMethod name="CrowdingDistance">
```

```
            <threshold>0.2</threshold>
        </validationMethod>
    </HybridModel>
  </Models>
  ...
  <Steps>
    <MultiRun name="hybridModelCode">
      <Input class="Files" type="">gen.one</Input>
      <Input class="Files" type="">gen.two</Input>
      <Input class="DataObjects"
         type="PointSet">inputHolder</Input>
      <Model class="Models" type="HybridModel">hybrid</Model>
      <Sampler class="Samplers" type="Stratified">LHS</Sampler>
      <Output class="DataObjects"
         type="PointSet">samples</Output>
      <Output class="OutStreams" type="Print">samples</Output>
    </MultiRun>
  </Steps>
  ...
</Simulation>
```

Note: For this example, the user needs to provide all the inputs for the **HybridModel**, i.e. Files for the **Code** and DataObject for the **ROM** defined in the **HybridModel**.

15.8 LogicalModel

The **LogicalModel** is a model aimed to execute ROMs, Codes and ExternalModels via a user provided control function. Basically, the control function utilizes the inputs generated by RAVEN and the control logic provided by the user to determine which model to execute. **Note:** For this type of model, we currently require all models listed under **LogicalModel** should have the same inputs and outputs from RAVEN point of view.

The specifications of a LogicalModel must be defined within the XML block `<LogicalModel>`. This XML node needs to contain the attributes:

- `name`, ***required string attribute***, user-defined name of this LogicalModel. **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 kept empty.

Within the `<LogicalModel>` XML node, the multiple entities that constitute this LogicalModel needs to be inputted.

- **<Model>**, ***XML node, required parameter***. The text portion of this node needs to contain the name of the Model This XML node needs to contain the attributes:
  - **class**, ***required string attribute***, the main "class" of the Model.
  - **type**, ***required string attribute***, the sub-type of the Model.

  Note: The user can provided various **<Model>** entities, including **'Code'**, **'ROM'** and **'ExternalModel'**.

- **<ControlFunction>**, ***XML node, required parameter***. The text portion of this node needs to contain the name of the function. This XML node needs to contain the attributes:
  - **class**, ***required string attribute***, the main "class" of the ControlFunction.
  - **type**, ***required string attribute***, the sub-type of the ControlFunction.

  Note: In order to work properly, this function must have a method named "evaluate" that returns a single python str object representing the model that would be executed.

Example (LogicalModel using external models):

```
<Simulation>
  ...
  <Models>
    <ExternalModel ModuleToLoad="sum" name="sum" subType="">
      <variables>x, y, z</variables>
    </ExternalModel>

    <ExternalModel ModuleToLoad="minus" name="minus" subType="">
      <variables>x, y, z</variables>
    </ExternalModel>

    <ExternalModel ModuleToLoad="multiply" name="multiply"
       subType="">
      <variables>x, y, z</variables>
    </ExternalModel>

    <LogicalModel name="logical" subType="">
      <Model class="Models" type="ExternalModel">sum</Model>
      <Model class="Models" type="ExternalModel">minus</Model>
      <Model class="Models" type="ExternalModel">multiply</Model>
      <ControlFunction class="Functions"
         type="External">control</ControlFunction>
    </LogicalModel>
  </Models>
```

```
  ...
  <Steps>
    <MultiRun name="mc">
      <Input class="DataObjects"
         type="PointSet">inputHolder</Input>
      <Model class="Models" type="LogicalModel">logical</Model>
      <Sampler class="Samplers"
         type="MonteCarlo">MonteCarlo</Sampler>
      <Output class="DataObjects" type="PointSet">outSet</Output>
      <Output class="DataObjects"
         type="PointSet">tagetSet</Output>
      <Output class="OutStreams" type="Print">dumpOut</Output>
    </MultiRun>
  </Steps>
  ...
</Simulation>
```

Corresponding Python function for **<ControlFunction>**:

```
def evaluate(self):
  """
    Method required by RAVEN to run this as an external model.
    @ In, self, object, object to store members on
    @ Out, model, str, the name of external model that
      will be executed by hybrid model
  """
  model = None
  if self.x > 0 and self.y >1:
    model = 'sum'
  elif self.x > 0 and self.y <= 1:
    model = 'multiply'
  else:
    model = 'minus'
  return model
```

Example (LogicalModel using codes):

```
<Simulation>
  ...
  <Models>
    <Code name="poly" subType="GenericCode">
      <executable>logicalCode/poly_code.py</executable>
      <clargs arg="python" type="prepend"/>
```

```
      <clargs arg="-i" extension=".one" type="input"/>
      <fileargs arg="aux" extension=".two" type="input"/>
      <fileargs arg="output" type="output"/>
    </Code>
    <Code name="exp" subType="GenericCode">
      <executable>logicalCode/exp_code.py</executable>
      <clargs arg="python" type="prepend"/>
      <clargs arg="-i" extension=".one" type="input"/>
      <fileargs arg="aux" extension=".two" type="input"/>
      <fileargs arg="output" type="output"/>
    </Code>
    <LogicalModel name="logical" subType="">
      <Model class="Models" type="Code">poly</Model>
      <Model class="Models" type="Code">exp</Model>
      <ControlFunction class="Functions"
         type="External">control</ControlFunction>
    </LogicalModel>
  </Models>
  ...
  <Steps>
    <MultiRun name="logicalModelCode">
      <Input class="Files" type="">gen.one</Input>
      <Input class="Files" type="">gen.two</Input>
      <Model class="Models" type="LogicalModel">logical</Model>
      <Sampler class="Samplers" type="Stratified">LHS</Sampler>
      <Output class="DataObjects"
         type="PointSet">samples</Output>
      <Output class="OutStreams" type="Print">samples</Output>
    </MultiRun>
  </Steps>
  ...
</Simulation>
```

Corresponding Python function for **<ControlFunction>**:

```
def evaluate(self):
  """
    Method required by RAVEN to run this as an external model.
    @ In, self, object, object to store members on
    @ Out, model, str, the name of external model that
      will be executed by hybrid model
  """
  model = None
```

```
  if self.x > 0.5 and self.y > 1.5:
    model = 'poly'
  else:
    model = 'exp'

  return model
```

Note: For these examples, the user needs to provide all the inputs for the models listed under **LogicalModel**, i.e. Files for the **Code** and DataObject for the **ExternalModel** defined in the **LogicalModel**.

16 Functions

The RAVEN code provides support for the usage of user-defined external functions. These functions are python modules, with a format that is automatically interpretable by the RAVEN framework. For example, users can define their own method to perform a particular post-processing activity and the code will be embedded and use the function as though it were an active part of the code itself. In this section, the XML input syntax and the format of the accepted functions are fully specified.

The specifications of an external function must be defined within the XML block **`<External>`**. This XML node requires the following attributes:

- **`name`**, ***required string attribute***, user-defined name of this function. **Note:** As with other objects, this name can be used to refer to this specific entity from other input blocks in the XML.
- **`file`**, ***required string attribute***, absolute or relative path specifying the code associated to this function. **Note:** If a relative path is specified, it must be relative with respect to where the user is running the instance of RAVEN.

In order to make the RAVEN code aware of the variables the user is going to manipulate/use in her/his own python function, the variables need to be specified in the **`<External>`** input block. The user needs to input, within this block, only the variables directly used by the external function and not the local variables that the user does not want, for example, those stored in a RAVEN internal object. These variables are inputted in a single **`<variables>`** XML node:

- **`<variables>`**, ***comma separated list, required parameter***, in the body of this XML node, the user needs to specify the name of the variables (separated by commas). This variables need to match variables used/defined in the external python function.

When the external function variables are defined, at runtime, RAVEN initializes them and keeps track of their values during the simulation. Each variable defined in the **`<External>`** block is available in the function as a python `self.` member. In the following, an example of a user-defined external function is reported (a python module and its related XML input specifications).

Example Python Function:

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

Example XML Input:

```
  ...
  <Functions>
    ...
    <External name='whatever' file='path_to_python_file'>
     ...
     <variables>var1,var2</variables>
     ...
    </External>
    ...
  </Functions>
   ...
</Simulation>
```

17 Metrics

The Metrics block allows the user to specify the similarity/dissimilarity metrics to be used for other RAVEN entities, such as **PostProcessors**, and **HybridModel**.

In the RAVEN input file these metrics are defined as follows:

```
<Simulation>
  ...
  <Metrics>
    ...
    <MetricID name='metricName'>
      ...
     <param1>value</param1>
      ...
    </MetricID>
    ...
  </Metrics>
  ...
</Simulation>
```

The metrics, that are available in RAVEN, can be categorized into several main classes:

- **Paired Distance Metric**, distance metrics between two variables u and v, such as **'euclidean'**, **'manhattan'**, **'minkowski'** and so on.
- **Regression Metric**, measure the regression performance, such as **'mean_squared_error'**, **'r2_score'**, **'explained_variance_score'** and **'mean_absolute_error'**.
- **Boolean Metric**, distance metrics between two boolean variables u and v, such as **'dice'**, **'hamming'**, **'yule'** and so on.
- **Pairwise Metric**, compute the distance or kernel between each pair of the two collections of input or observations in n-dimensional space. **Note:** These metrics can be only used in the clustering post-processor of data mining.
- **Other metric**, such as **'DTW'**.

The valid **MetricID**s are: **<SKL>**, **<ScipyMetric>**, **<DTW>**, **<CDFAreaDifference>**, and **<PDFCommonArea>**. This XML node requires the following attributes:

- **name**, ***required string attribute***, user-defined name of this metric. **Note:** As with other objects, this name can be used to refer to this specific entity from other input blocks in the XML.

Note: If you are using **<ScipyMetric>**, please pay more attention on the weight associated with the metric calculations. Scipy does not normalize the weight during the calculation, and the results can be significant difference from the normalized weight.

In RAVEN, lots of metrics are just interfaces directly with metrics available within **Scipy** and **SciKit-Learn**. In this case, the algorithm for the metrics is choosen by the subnode **<metricType>** under the parent node **<SKL>** (metric from SciKit-Learn) or **<ScipyMetric>** (metric from Scipy). For example, **<metricType>'paired_distance|euclidean'</metricType>**.

In the following sub-sections, the input requirements for all of the metrics are presented in the following sections.

17.1 Paired Distance Metric

17.1.1 Euclidean

This metric compute the paired euclidean distances between u and v, i.e.

$$||u - v||_2 \tag{43}$$

This metric interface directly with the metric available within *SciKit-Learn*. The specifications of this metric must be defined within the XML block **<SKL>**. This XML node needs to contain the following subnode:

- **<metricType>**`paired_distance|euclidean`**</metricType>**, ***vertical bar (|) separated string, required field***.

17.1.2 Cosine

This metric computes the paired cosine distances between u and v, i.e.

$$1 - \frac{u \cdot v}{||u||_2 ||v||_2} \tag{44}$$

where $u \cdot v$ is the dot product of u and v

This metric interface directly with the metric available within *SciKit-Learn*. The specifications of this metric must be defined within the XML block **<SKL>**. This XML node needs to contain the following subnode:

- **<metricType>**`paired_distance|cosine`**</metricType>**, ***vertical bar (|) separated string, required field***.

17.1.3 Manhattan

This metric computes the L1 distances between u and v, i.e.

$$\sum_i |u_i - v_i| \tag{45}$$

This metric interface directly with the metric available within *SciKit-Learn*. The specifications of this metric must be defined within the XML block **<SKL>**. This XML node needs to contain the following subnode:

- **<metricType>**`paired_distance|manhattan`**</metricType>**, ***vertical bar (|) separated string, required field***.

17.1.4 Braycurtis

This metric computes the Bray-Curtis distances between u and v, i.e.

$$\sum |u_i - v_i| / \sum |u_i + v_i| \tag{46}$$

The Bray-Curtis distance is in the range $[0, 1]$. This metric interface directly with the metric available within *Scipy*. The specifications of this metric must be defined within the XML block **<ScipyMetric>**. This XML node needs to contain the following subnode:

- **<metricType>**`paired_distance|braycurtis`**</metricType>**, ***vertical bar (|) separated string, required field***.

In addition to this XML subnode, the users can also specify the weights for given metric:

- **<w>**, ***comma separated floats, optional parameter***, the weights for each value in *u* and *v*. Default is None, which gives each value a weight of 1.0.

17.1.5 Canberra

This metric computes the Canberra distance between u and v, i.e.

$$d(u, v) = \sum_i \frac{|u_i - v_i|}{|u_i| + |v_i|} \tag{47}$$

This metric interface directly with the metric available within *Scipy*. The specifications of this metric must be defined within the XML block **<ScipyMetric>**. This XML node needs to contain the following subnode:

- **<metricType>**`paired_distance|canberra`**</metricType>**, ***vertical bar (|) separated string, required field***.

In addition to this XML subnode, the users can also specify the weights for given metric:

- **<w>**, ***comma separated floats, optional parameter***, the weights for each value in u and v. Default is None, which gives each value a weight of 1.0.

17.1.6 Correlation

This metric computes the correlation distance between u and v, i.e.

$$1 - \frac{(u - \bar{u}) \cdot (v - \bar{v})}{||(u - \bar{u})||_2 ||(v - \bar{v})||_2} \tag{48}$$

where $\bar{u}$ is the mean of the elements of u

This metric interface directly with the metric available within *Scipy*. The specifications of this metric must be defined within the XML block **<ScipyMetric>**. This XML node needs to contain the following subnode:

- **<metricType>**`paired_distance|correlation`**</metricType>**, ***vertical bar (|) separated string, required field***.

In addition to this XML subnode, the users can also specify the weights for given metric:

- **<w>**, ***comma separated floats, optional parameter***, the weights for each value in u and v. Default is None, which gives each value a weight of 1.0.

17.1.7 Minkowski

This metric computes the Minkowski distance between u and v, i.e.

$$||u - v||_p = (\sum |u_i - v_i|^p)^{1/p} \tag{49}$$

This metric interface directly with the metric available within *Scipy*. The specifications of this metric must be defined within the XML block **<ScipyMetric>**. This XML node needs to contain the following subnode:

- **<metricType>**paired_distance|minkowski**</metricType>**, ***vertical bar (|) separated string, required field***.

In addition to this XML subnode, the users can also specify the weights for given metric:

- **<w>**, ***comma separated floats, optional parameter***, the weights for each value in u and v. Default is None, which gives each value a weight of 1.0.

- **<p>**, ***float, required field***, value for the parameter p

In the RAVEN input file, these metrics are defined as follows:

```
<Simulation>
  ...
  <Metrics>
    <SKL name="euclidean">
        <metricType>paired_distance|euclidean</metricType>
    </SKL>
    <SKL name="cosine">
        <metricType>paired_distance|cosine</metricType>
    </SKL>
    <SKL name="manhattan">
        <metricType>paired_distance|manhattan</metricType>
    </SKL>
    <ScipyMetric name="braycurtis">
      <metricType>paired_distance|braycurtis</metricType>
    </ScipyMetric>
    <ScipyMetric name="canberra">
        <metricType>paired_distance|canberra</metricType>
    </ScipyMetric>
    <ScipyMetric name="correlation">
        <metricType>paired_distance|correlation</metricType>
    </ScipyMetric>
    <ScipyMetric name="minkowski">
        <metricType>paired_distance|minkowski</metricType>
        <p>5</p>
        <w>0.1, 0.1, 0.1, 0.1, 0.1</w>
```

```
    </ScipyMetric>
  </Metrics>
  ...
</Simulation>
```

17.2 Regression Metric

17.2.1 Explained variance score

This metric computes the explained variance regression score, i.e.

$$1.0 - \frac{Var[u - v]}{Var[u]} \tag{50}$$

The best possible score is 1.0, lower values are worse.

This metric interface directly with the metric available within *Scikit-Learn*. The specifications of this metric must be defined within the XML block **<SKL>**. This XML node needs to contain the following subnode:

- **<metricType>**`regression|explained_variance_score`**</metricType>**, ***vertical bar (|) separated string, required field***.

In addition to this XML subnode, the users can also specify the weights for given metric:

- **<sample_weight>**, ***comma separated floats, optional parameter***, the weights for each value in *u* and *v*. Default is None, which gives each value a weight of 1.0.

17.2.2 Mean absolute error

This metric computes mean absolute error, a risk metric corresponding to the expected value of the absolute error loss or *l1*-norm loss.

$$\frac{1}{n_{samples}} \sum_{i=0}^{n_{samples}-1} |u_i - v_i| \tag{51}$$

This metric interface directly with the metric available within *Scikit-Learn*. The specifications of this metric must be defined within the XML block **<SKL>**. This XML node needs to contain the following subnode:

- **<metricType>**regression|mean_absolute_error**</metricType>**, ***vertical bar (|) separated string, required field***.

In addition to this XML subnode, the users can also specify the weights for given metric:

- **<sample_weight>**, ***comma separated floats, optional parameter***, the weights for each value in *u* and *v*. Default is None, which gives each value a weight of 1.0.

17.2.3 Mean squared error

This metric computes mean square error, a risk metric corresponding to the expected value of the squared error or loss.

$$\frac{1}{n_{samples}} \sum_{i=0}^{n_{samples}-1} (u_i - v_i)^2 \tag{52}$$

This metric interface directly with the metric available within *Scikit-Learn*. The specifications of this metric must be defined within the XML block **<SKL>**. This XML node needs to contain the following subnode:

- **<metricType>**regression|mean_squared_error**</metricType>**, ***vertical bar (|) separated string, required field***.

In addition to this XML subnode, the users can also specify the weights for given metric:

- **<sample_weight>**, ***comma separated floats, optional parameter***, the weights for each value in *u* and *v*. Default is None, which gives each value a weight of 1.0.

17.2.4 R2 score

This metric computes the coefficient of determination, i.e.

$$1.0 - \frac{\sum_{i=0}^{n_{samples}-1} (u_i - v_i)^2}{\sum_{i=0}^{n_{samples}-1} (u_i - mean[u])^2} \tag{53}$$

It provides a measure of how well future samples are likely to be predicted by the model. Best possible score is 1.0 and it can be negative.

This metric interface directly with the metric available within *Scikit-Learn*. The specifications of this metric must be defined within the XML block **<SKL>**. This XML node needs to contain the following subnode:

- **<metricType>**regression|r2_score**</metricType>**, ***vertical bar (|) separated string, required field***.

In addition to this XML subnode, the users can also specify the weights for given metric:

- **<sample_weight>**, ***comma separated floats, optional parameter***, the weights for each value in u and v. Default is None, which gives each value a weight of 1.0.

In the RAVEN input file, these metrics are defined as follows:

```
<Simulation>
  ...
  <Metrics>
    <SKL name="explained_variance_score">
        <metricType>regression|explained_variance_score</metricType>
        <sample_weight>0.1,0.1,0.1,0.05,0.05</sample_weight>
    </SKL>
    <SKL name="mean_absolute_error">
        <metricType>regression|mean_absolute_error</metricType>
        <sample_weight>0.1,0.1,0.1,0.05,0.05</sample_weight>
    </SKL>
    <SKL name="r2_score">
        <metricType>regression|r2_score</metricType>
        <sample_weight>0.1,0.1,0.1,0.05,0.05</sample_weight>
    </SKL>
    <SKL name="mean_squared_error">
        <metricType>regression|mean_squared_error</metricType>
        <sample_weight>0.1,0.1,0.1,0.05,0.05</sample_weight>
    </SKL>
  </Metrics>
  ...
</Simulation>
```

17.3 Boolean Metric

17.3.1 Dice

This metric computes the Dice dissimilarity between two boolean variables u and v

$$\frac{c_{TF} + c_{FT}}{2c_{TT} + c_{FT} + c_{TF}} \tag{54}$$

where c_{ij} is the number of occurrences of $\mathtt{u[k]} = i$ and $\mathtt{v[k]} = j$ for $k < n$

This metric interface directly with the metric available within *Scipy*. The specifications of this metric must be defined within the XML block **`<ScipyMetric>`**. This XML node needs to contain the following subnode:

- **`<metricType>`**`boolean|dice`**`</metricType>`**, ***vertical bar (|) separated string, required field***.

In addition to this XML subnode, the users can also specify the weights for given metric:

- **`<w>`**, ***comma separated floats, optional parameter***, the weights for each value in *u* and *v*. Default is None, which gives each value a weight of 1.0.

17.3.2 Hamming

This metric computes the Hamming distance between two boolean variables u and v, i.e.

$$\frac{c_{01} + c_{10}}{n} \tag{55}$$

where c_{ij} is the number of occurrences of $\mathtt{u[k]} = i$ and $\mathtt{v[k]} = j$ for $k < n$

This metric interface directly with the metric available within *Scipy*. The specifications of this metric must be defined within the XML block **`<ScipyMetric>`**. This XML node needs to contain the following subnode:

- **`<metricType>`**`boolean|hamming`**`</metricType>`**, ***vertical bar (|) separated string, required field***.

In addition to this XML subnode, the users can also specify the weights for given metric:

- **`<w>`**, ***comma separated floats, optional parameter***, the weights for each value in *u* and *v*. Default is None, which gives each value a weight of 1.0.

17.3.3 Jaccard

This metric computes the Jaccard-Needham dissimilarity distance between two boolean variables u and v, i.e.

$$\frac{c_{TF} + c_{FT}}{c_{TT} + c_{FT} + c_{TF}} \tag{56}$$

where c_{ij} is the number of occurrences of $\mathtt{u[k]} = i$ and $\mathtt{v[k]} = j$ for $k < n$

This metric interface directly with the metric available within *Scipy*. The specifications of this metric must be defined within the XML block **`<ScipyMetric>`**. This XML node needs to contain the following subnode:

- **`<metricType>`**`boolean|jaccard`**`</metricType>`**, ***vertical bar (|) separated string, required field***.

In addition to this XML subnode, the users can also specify the weights for given metric:

- **`<w>`**, ***comma separated floats, optional parameter***, the weights for each value in *u* and *v*. Default is None, which gives each value a weight of 1.0.

17.3.4 Kulsinski

This metric computes the Kulsinski dissimilarity distance between two boolean variables u and v, i.e.

$$\frac{c_{TF} + c_{FT} - c_{TT} + n}{c_{FT} + c_{TF} + n} \tag{57}$$

where c_{ij} is the number of occurrences of $\mathtt{u[k]} = i$ and $\mathtt{v[k]} = j$ for $k < n$

This metric interface directly with the metric available within *Scipy*. The specifications of this metric must be defined within the XML block **`<ScipyMetric>`**. This XML node needs to contain the following subnode:

- **`<metricType>`**`boolean|kulsinski`**`</metricType>`**, ***vertical bar (|) separated string, required field***.

In addition to this XML subnode, the users can also specify the weights for given metric:

- **`<w>`**, ***comma separated floats, optional parameter***, the weights for each value in *u* and *v*. Default is None, which gives each value a weight of 1.0.

17.3.5 Rogerstanimoto

This metric computes the Rogers-Tanimoto dissimilarity distance between two boolean variables u and v, i.e.

$$\frac{R}{c_{TT} + c_{FF} + R} \tag{58}$$

where c_{ij} is the number of occurrences of $\mathtt{u[k]} = i$ and $\mathtt{v[k]} = j$ for $k < n$ and $R = 2(c_{TF} + c_{FT})$

This metric interface directly with the metric available within *Scipy*. The specifications of this metric must be defined within the XML block **`<ScipyMetric>`**. This XML node needs to contain the following subnode:

- **`<metricType>`**`boolean|rogerstanimoto`**`</metricType>`**, ***vertical bar (|) separated string, required field***.

In addition to this XML subnode, the users can also specify the weights for given metric:

- **`<w>`**, ***comma separated floats, optional parameter***, the weights for each value in *u* and *v*. Default is None, which gives each value a weight of 1.0.

17.3.6 Russellrao

This metric computes the Russell-Rao dissimilarity distance between two boolean variables u and v, i.e.

$$\frac{n - c_{TT}}{n} \tag{59}$$

where c_{ij} is the number of occurrences of $\mathtt{u[k]} = i$ and $\mathtt{v[k]} = j$ for $k < n$

This metric interface directly with the metric available within *Scipy*. The specifications of this metric must be defined within the XML block **`<ScipyMetric>`**. This XML node needs to contain the following subnode:

- **`<metricType>`**`boolean|russellrao`**`</metricType>`**, ***vertical bar (|) separated string, required field***.

In addition to this XML subnode, the users can also specify the weights for given metric:

- **`<w>`**, ***comma separated floats, optional parameter***, the weights for each value in *u* and *v*. Default is None, which gives each value a weight of 1.0.

17.3.7 Sokalmichener

This metric computes the Sokal-Michener dissimilarity distance between two boolean variables u and v, i.e.

$$\frac{R}{S + R} \tag{60}$$

where c_{ij} is the number of occurrences of $\mathtt{u[k]} = i$ and $\mathtt{v[k]} = j$ for $k < n$, $R = 2 * (c_{TF} + c_{FT})$ and $S = c_{FF} + c_{TT}$

This metric interface directly with the metric available within *Scipy*. The specifications of this metric must be defined within the XML block **`<ScipyMetric>`**. This XML node needs to contain the following subnode:

- **`<metricType>`**`boolean|sokalmichener`**`</metricType>`**, ***vertical bar (|) separated string, required field***.

In addition to this XML subnode, the users can also specify the weights for given metric:

- **`<w>`**, ***comma separated floats, optional parameter***, the weights for each value in *u* and *v*. Default is None, which gives each value a weight of 1.0.

17.3.8 Sokalsneath

This metric computes the Sokal-Sneath dissimilarity distance between two boolean variables u and v, i.e.

$$\frac{R}{c_{TT} + R} \tag{61}$$

where c_{ij} is the number of occurrences of $\mathtt{u[k]} = i$ and $\mathtt{v[k]} = j$ for $k < n$ and $R = 2(c_{TF} + c_{FT})$

This metric interface directly with the metric available within *Scipy*. The specifications of this metric must be defined within the XML block **`<ScipyMetric>`**. This XML node needs to contain the following subnode:

- **`<metricType>`**`boolean|sokalsneath`**`</metricType>`**, ***vertical bar (|) separated string, required field***.

In addition to this XML subnode, the users can also specify the weights for given metric:

- **`<w>`**, ***comma separated floats, optional parameter***, the weights for each value in *u* and *v*. Default is None, which gives each value a weight of 1.0.

17.3.9 Yule

This metric computes the Yule dissimilarity distance between two boolean variables u and v, i.e.

$$\frac{R}{c_{TT} * c_{FF} + \frac{R}{2}} \tag{62}$$

where c_{ij} is the number of occurrences of $\mathtt{u}[\mathtt{k}] = i$ and $\mathtt{v}[\mathtt{k}] = j$ for $k < n$ and $R = 2.0 * c_{TF} * c_{FT}$

This metric interface directly with the metric available within *Scipy*. The specifications of this metric must be defined within the XML block **<ScipyMetric>**. This XML node needs to contain the following subnode:

- **<metricType>**boolean|yule**</metricType>**, ***vertical bar (|) separated string, required field***.

In addition to this XML subnode, the users can also specify the weights for given metric:

- **<w>**, ***comma separated floats, optional parameter***, the weights for each value in *u* and *v*. Default is None, which gives each value a weight of 1.0.

An example of Boolean metric defined in RAVEN is provided below:

```
<Simulation>
  ...
  <Metrics>
    ...
    <ScipyMetric name="rogerstanimoto">
        <metricType>boolean|rogerstanimoto</metricType>
    </ScipyMetric>
    <ScipyMetric name="dice">
        <metricType>boolean|dice</metricType>
    </ScipyMetric>
    <ScipyMetric name="hamming">
        <metricType>boolean|hamming</metricType>
    </ScipyMetric>
    <ScipyMetric name="jaccard">
        <metricType>boolean|jaccard</metricType>
    </ScipyMetric>
    <ScipyMetric name="kulsinski">
        <metricType>boolean|kulsinski</metricType>
    </ScipyMetric>
    <ScipyMetric name="russellrao">
        <metricType>boolean|russellrao</metricType>
    </ScipyMetric>
    <ScipyMetric name="sokalmichener">
        <metricType>boolean|sokalmichener</metricType>
    </ScipyMetric>
    <ScipyMetric name="sokalsneath">
```

```
        <metricType>boolean|sokalsneath</metricType>
    </ScipyMetric>
    <ScipyMetric name="yule">
        <metricType>boolean|yule</metricType>
    </ScipyMetric>
    ...
  </Metrics>
  ...
</Simulation>
```

17.4 Dynamic Time Warping

The Dynamic Time Warping (DTW) is a distance metrice that is used to measure similarity between two sequences, i.e. temporal sequences.

The specifications of a DTW distance must be defined within the XML block. **<DTW>**.

This XML node needs to contain the attributes:

- **<order>**, ***int, required field***, order of the DTW calculation: 0 specifices a classical DTW caluclation and 1 specifies a derivative DTW calculation
- **<localDistance>**, ***string, required field***, the ID of the distance function to be employed to determine the local distance evaluation of two time series. Available options are provided by the Scipy pairwise distances (cityblock, cosine, euclidean, $l1$, $l2$, manhattan, braycurtis, canberra, chebyshev, correlation, dice, hamming, jaccard, kulsinski, mahalanobis, matching, minkowski, rogerstanimoto, russellrao, seuclidean, sokalmichener, sokalsneath, sqeuclidean, yule)

An example of Minkowski distance defined in RAVEN is provided below:

```
<Simulation>
  ...
  <Metrics>
    ...
    <DTW name="example">
      <order>0</order>
      <localDistance>euclidean</localDistance>
    </DTW>
    ...
  </Metrics>
```

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

17.5 CDFAreaDifference

This calculates the difference in area between the two CDFs. This metric supports using distributions as input. Other inputs are converted to a CDF.

$$\text{CDF area difference} = \int_{-\infty}^{\infty} \|CDF_a(x) - CDF_b(x)\| dx \tag{63}$$

This metric has the same units as x. The closer the number is to zero, the closer the match. A perfect match would be 0.0.

An example is provided below:

```
<Simulation>
  ...
  <Metrics>
    ...
    <CDFAreaDifference name="cdf_diff" />
    ...
  </Metrics>
  ...
</Simulation>
```

17.6 PDFCommonArea

This calculates the common area between the two PDFs. The higher the value the closer the PDFs are. This metric supports distributions as inputs. Other inputs are converted to a PDF.

$$\text{PDF common area} = \int_{-\infty}^{\infty} \min(PDF_a(x), PDF_b(x)) dx \tag{64}$$

A perfect match would be 1.0.

An example is provided below:

```
<Simulation>
  ...
  <Metrics>
    ...
    <PDFCommonArea name="pdf_area" />
    ...
  </Metrics>
  ...
</Simulation>
```

17.7 Pairwise Metric

This calculates the pairwise distance or kernel between each row of the two collections of inputs. This metric can be only used in the **DataMining** post-processor.

17.7.1 Polynomial

Compute the polynomial kernel between X and Y:

$$K(X,Y) = (gamma < X, Y > +coef0)^{degree} \tag{65}$$

This metric interface directly with the metric available within *Scikit-Learn*. The specifications of this metric must be defined within the XML block **`<PairwiseMetric>`**. This XML node needs to contain the following subnode:

- **`<metricType>`**`kernel|Polynomial`**`</metricType>`**, ***vertical bar (|) separated string, required field***.

In addition to this XML subnode, the users can also specify the following subnodes:

- **`<degree>`**, ***integer, optional parameter***, default '3'
- **`<gamma>`**, ***float, optional parameter***, default $1.0/numberColumnsInX$
- **`<coef0>`**, ***integer, optional parameter***, default '1'

17.7.2 additive_chi2

Computes the additive chi-squared kernel between observations in X and Y The chi-squared kernel is computed between each pair of rows in X and Y. X and Y have to be non-negative. This kernel is most commonly applied to histograms. The chi-squared kernel is given by:

$$K(x, y) = -Sum[(x - y)^2/(x + y)] \tag{66}$$

This metric interface directly with the metric available within *Scikit-Learn*. The specifications of this metric must be defined within the XML block **`<PairwiseMetric>`**. This XML node needs to contain the following subnode:

- **`<metricType>`**`kernel|additive_chi2`**`</metricType>`**, ***vertical bar (|) separated string, required field***.

17.7.3 chi2

Computes the exponential chi-squared kernel between observations in X and Y The chi-squared kernel is computed between each pair of rows in X and Y. X and Y have to be non-negative. This kernel is most commonly applied to histograms. The chi-squared kernel is given by:

$$K(x, y) = exp(-gamma * Sum[(x - y)^2/(x + y)]) \tag{67}$$

This metric interface directly with the metric available within *Scikit-Learn*. The specifications of this metric must be defined within the XML block **`<PairwiseMetric>`**. This XML node needs to contain the following subnode:

- **`<metricType>`**`kernel|chi2`**`</metricType>`**, ***vertical bar (|) separated string, required field***.

In addition to this XML subnode, the users can also specify the following subnodes:

- **`<gamma>`**, ***float, optional parameter***, default '1'

17.7.4 cosine_similarity

Compute the cosine similarity between X and Y:

$$K(X, Y) =< X, Y > /(||X|| * ||Y||) \tag{68}$$

This metric interface directly with the metric available within *Scikit-Learn*. The specifications of this metric must be defined within the XML block **<PairwiseMetric>**. This XML node needs to contain the following subnode:

- **<metricType>**`kernel|cosine_similarity`**</metricType>**, ***vertical bar (|) separated string, required field***.

17.7.5 laplacian

Computes the laplacian kernel between observations in X and Y The laplacian kernel is given by:

$$K(x, y) = exp(-gamma * ||x - y||_1) \tag{69}$$

for each pair of rows x in X and y in Y. This metric interface directly with the metric available within *Scikit-Learn*. The specifications of this metric must be defined within the XML block **<PairwiseMetric>**. This XML node needs to contain the following subnode:

- **<metricType>**`kernel|laplacian`**</metricType>**, ***vertical bar (|) separated string, required field***.

In addition to this XML subnode, the users can also specify the following subnodes:

- **<gamma>**, ***float, optional parameter***, default $1.0/numberColumnsInX$

17.7.6 linear

computes the linear kernel between X and Y

$$K(X, Y) = X^T * Y \tag{70}$$

This metric interface directly with the metric available within *Scikit-Learn*. The specifications of this metric must be defined within the XML block **<PairwiseMetric>**. This XML node needs to contain the following subnode:

- **<metricType>**`kernel|linear`**</metricType>**, ***vertical bar (|) separated string, required field***.

17.7.7 rbf

Computes the laplacian kernel between observations in X and Y The laplacian kernel is given by:

$$K(x, y) = exp(-gamma * ||x - y||^2) \tag{71}$$

for each pair of rows x in X and y in Y. This metric interface directly with the metric available within *Scikit-Learn*. The specifications of this metric must be defined within the XML block **`<PairwiseMetric>`**. This XML node needs to contain the following subnode:

- **`<metricType>`**`kernel|rbf`**`</metricType>`**, ***vertical bar (|) separated string, required field***.

- **`<gamma>`**, ***float, optional parameter***, default $1.0/numberColumnsInX$

17.7.8 sigmoid

Compute the sigmoid kernel between X and Y:

$$K(X, Y) = tanh(gamma < X, Y > +coef0) \tag{72}$$

This metric interface directly with the metric available within *Scikit-Learn*. The specifications of this metric must be defined within the XML block **`<PairwiseMetric>`**. This XML node needs to contain the following subnode:

- **`<metricType>`**`kernel|sigmoid`**`</metricType>`**, ***vertical bar (|) separated string, required field***.

- **`<gamma>`**, ***float, optional parameter***, default $1.0/numberColumnsInX$
- **`<coef0>`**, ***integer, optional parameter***, default '1'

17.7.9 Distance Based Metric

This metric interface directly with the metric available within *Scipy or Scikit-Learn*. The specifications of this metric must be defined within the XML block **`<PairwiseMetric>`**. This XML node needs to contain the following subnode:

- **`<metricType>`**`pairwise|'metric'`**`</metricType>`**, ***vertical bar (|) separated string, required field***.

Note: ‘metric’, the distance metric to use, this can be ‘braycurtis’, ‘canberra’, ‘chebyshev’, ‘correlation’, ‘cosine’, ‘dice’, ‘euclidean’, ‘hamming’, ‘jaccard’, ‘kulsinski’, ‘matching’, ‘minkowski’, ‘rogerstanimoto’, ‘russellrao’, ‘sokalmichener’, ‘sokalsneath’, ‘yule’, ‘manhatten’. The definition for each metric can be found in previous sections.

In addition to this XML subnode, the users can also specify the corresponding parameters for each ‘metric’ according to previous sections.

18 Steps

The core of the RAVEN calculation flow is the **Step** system. The **Step** is in charge of assembling different entities in RAVEN (e.g. Samplers, Models, Databases, etc.) in order to perform a task defined by the kind of step being used. A sequence of different **Steps** represents the calculation flow.

Before analyzing each **Step** type, it is worth to 1) explain how a general **Step** entity is organized, and 2) introduce the concept of step "role" . In the following example, a general example of a **Step** is shown below:

```
<Simulation>
  ...
  <Steps>
    ...
    <WhatEverStepType name='aName'>
        <Role1 class='aMainClassType'
           type='aSubType'>userDefinedName1</Role1>
        <Role2 class='aMainClassType'
           type='aSubType'>userDefinedName2</Role2>
        <Role3 class='aMainClassType'
           type='aSubType'>userDefinedName3</Role3>
        <Role4 class='aMainClassType'
           type='aSubType'>userDefinedName4</Role4>
    </WhatEverStepType>
    ...
  </Steps>
  ...
</Simulation>
```

As shown above each **Step** consists of a list of entities organized into "Roles." Each role represents a behavior the entity (object) will assume during the evaluation of the **Step**. In RAVEN, several different roles are available:

- **Input** represents the input of the **Step**. The allowable input objects depend on the type of **Model** in the **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**.
- **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.

- **Sampler** defines the sampling strategy to be used to probe the model.
  It is worth to mention that, 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 important 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. For the moment, there are two uses: 1) A **Step** is employing the search of the Limit Surface (LS), through the class of Adaptive **Samplers**); in this case, it contains the coordinates of the LS in the input space; and 2) Some of the post-processors employ clustering algorithms and the cluster centers will be stored in this file with the input being the cluster labels.

Depending on the **Step** type, different combinations of these roles can be used. For this reason, it is important to analyze each **Step** type in details.

The available steps are the following

- SingleRun (see Section 18.1)
- MultiRun(see Section 18.2)
- IOStep(see Section 18.3)
- RomTrainer(see Section 18.4)
- PostProcess(see Section 18.5)

18.1 SingleRun

The **SingleRun** is the simplest step the user can use to assemble a calculation flow: perform a single action of a **Model**. For example, it can be used to run a single job (Code Model) and collect the outcome(s) in a "**DataObjects**" object of type **Point** or **History** (see Section 12 for more details on available data representations).

The specifications of this Step must be defined within a `<SingleRun>` XML block. This XML node has the following definable attributes:

- `name`, ***required string attribute***, user-defined name of this **Step**. **Note:** This name is used to reference this specific entity in the `<RunInfo>` block, under the `<Sequence>` node. If

the name of this **Step** is not listed in the **<Sequence>** block, its action is not going to be performed.

- **repeatFailureRuns**, ***optional integer attribute***, this optional attribute could be used to set a certain number of repetitions that need to be performed when a realization (i.e. run) fails (e.g. **repeatFailureRuns** = "3", 3 tries).
- **pauseAtEnd**, ***optional boolean/string attribute (case insensitive)***, if True (True values = True, yes, y, t), the code will pause at the end of the step, waiting for a user signal to continue. This is used in case one or more of the **Outputs** are of type **OutStreams**. For example, it can be used when an **OutStreams** of type **Plot** is output to the screen. Thus, allowing the user to interact with the **Plot** (e.g. rotate the figure, change the scale, etc.).
  Default: False.
- **clearRunDir**, ***optional boolean attribute***, indicates whether the run directory should be cleared (removed) before beginning the Step calculation. The run directory has the same **name** as the **<Step>** and is located within the **<WorkingDir>**. Note this directory is only used for a **<Step>** with certain **<Model>** types, such as **<Code>**.

In the **<SingleRun>** input block, the user needs to specify the objects needed for the different allowable roles. This step accepts the following roles:

- **<Input>**, ***string, required parameter***, names an entity (defined elsewhere in the RAVEN input) that will be used as input for the model specified in this step. This XML node accepts the following attributes:
  - **class**, ***required string attribute***, main object class type. This string corresponds to the tag of the main object's type used in the input. For example, **'Files'**, **'DataObjects'**, **'Databases'**, etc.
  - **type**, ***required string attribute***, the actual entity type. This attribute needs to specify the object type within the main object class. For example, if the **class** attribute is **'DataObjects'**, the **type** attribute might be **'PointSet'**. **Note:** The class **'Files'** has no type (i.e. **type=''**).

  Note: The **class** and, consequently, the **type** usable for this role depends on the particular **<Model>** being used. In addition, the user can specify as many **<Input>** nodes as needed.

- **<Model>**, ***string, required parameter***, names an entity defined elsewhere in the input file to be used as a model for this step. This XML node accepts the following attributes:
  - **class**, ***required string attribute***, main object class type. For this role, only **'Models'** can be used.
  - **type**, ***required string attribute***, the actual entity type. This attribute needs to specify the object type within the `Models` object class. For example, the **type** attribute might be **'Code'**, **'ROM'**, etc.

- **<Output>**, ***string, required parameter*** names an entity defined elsewhere in the input to use as the output for the **Model**. This XML node recognizes the following attributes:
  - **class**, ***required string attribute***, main object class type. For this role, only **'DataObjects'**, **'Databases'**, and **'OutStreams'** can be used.
  - **type**, ***required string attribute***, the actual entity type. This attribute needs to specify the object type within the main object class. For example, if the **class** attribute is **'DataObjects'**, the **type** attribute might be **'PointSet'**.

  Note: The number of **<Output>** nodes is unlimited.

Example:

```
<Steps>
  ...
  <SingleRun name='StepName' pauseAtEnd='false'>
        <Input    class='Files'
           type=''>anInputFile.i</Input>
        <Input    class='Files'          type=''>aFile</Input>
        <Model   class='Models'      type='Code'>aCode</Model>
        <Output  class='Databases' type='HDF5'>aDatabase</Output>
        <Output  class='DataObjects'
           type='History'>aData</Output>
  </SingleRun>
  ...
</Steps>
```

18.2 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.

The specifications of this type of step must be defined within a **<MultiRun>** XML block. This XML node recognizes the following list of attributes:

- **name**, ***required string attribute***, user-defined name of this Step. **Note:** As with other objects, this name is used to reference this specific entity in the **<RunInfo>** block, under the **<Sequence>** node. If the name of this **Step** is not listed in the **<Sequence>** block, its action is not going to be performed.
- **re-seeding**, ***optional integer/string attribute***, this optional attribute could be used to control the seeding of the random number generator (RNG). If inputted, the RNG can be

reseeded. The value of this attribute can be: either 1) an integer value with the seed to be used (e.g. `re-seeding` = "20021986"), or 2) string value named "continue" where the RNG is not re-initialized

- `repeatFailureRuns`, ***optional integer attribute***, this optional attribute could be used to set a certain number of repetitions that need to be performed when a realization (i.e. run) fails (e.g. `repeatFailureRuns` = "3", 3 tries).
- `pauseAtEnd`, ***optional boolean/string attribute***, if True (True values = True, yes, y, t), the code will pause at the end of the step, waiting for a user signal to continue. This is used in case one or more of the **Outputs** are of type **OutStreams**. For example, it can be used when an **OutStreams** of type **Plot** is output to the screen. Thus, allowing the user to interact with the **Plot** (e.g. rotate the figure, change the scale, etc.).
- `sleepTime`, ***optional float attribute***, in this attribute the user can specify the waiting time (seconds) between two subsequent inquiries of the status of the submitted job (i.e. check if a run has finished).
  Default: 0.05.

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>`, ***string, required parameter***, names an entity to be used as input for the model specified in this step. This XML node accepts the following attributes:
  - `class`, ***required string attribute***, main object class type. This string corresponds to the tag of the main object's type used in the input. For example, `'Files'`, `'DataObjects'`, `'Databases'`, etc.
  - `type`, ***required string attribute***, the actual entity type. This attribute specifies the object type within the main object class. For example, if the `class` attribute is `'DataObjects'`, the `type` attribute might be `'PointSet'`. **Note:** The class `'Files'` has no type (i.e. `type=''`).

  Note: The `class` and, consequently, the `type` usable for this role depend on the particular `<Model>` being used. The user can specify as many `<Input>` nodes as needed.

- `<Model>`, ***string, required parameter*** names an entity defined elsewhere in the input that will be used as the model for this step. This XML node recognizes the following attributes:
  - `class`, ***required string attribute***, main object class type. For this role, only `'Models'` can be used.
  - `type`, ***required string attribute***, the actual entity type. This attribute needs to specify the object type within the `Models` object class. For example, the `type` attribute might be `'Code'`, `'ROM'`, etc.

- `<Sampler>`, ***string, optional parameter*** names an entity defined elsewhere in the input file to be used as a sampler. As mentioned in Section 10, the **Sampler** is in charge of defining the strategy to characterize the input space. This XML node recognizes the following attributes:

- **class**, ***required string attribute***, main object class type. This string corresponds to the tag of the main object's type used. Only **'Samplers'** can be used for this role.
- **type**, ***required string attribute***, the actual entity type. This attribute needs to specify the object type within the `Samplers` object class. For example, the **type** attribute might be **'MonteCarlo'**, **'Adaptive'**, **'AdaptiveDET'**, etc. See Section 10 for all the different types currently supported.

- **<Optimizer>**, ***string, optional parameter*** names an entity defined elsewhere in the input file to be used as an optimizer. As mentioned in Section 11, the **Optimizer** is in charge of defining the strategy to optimize an user-specified variable. This XML node recognizes the following attributes:
  - **class**, ***required string attribute***, main object class type. This string corresponds to the tag of the main object's type used. Only **'Optimizers'** can be used for this role.
  - **type**, ***required string attribute***, the actual entity type. This attribute needs to specify the object type within the `Optimizers` object class. For example, the **type** attribute might be **'SPSA'**, etc. See Section 11 for all the different types currently supported.

  Note: For Multi-Run, either one **<Sampler>** or one **<Optimizer>** is required.

- **<SolutionExport>**, ***string, optional parameter*** identifies an entity to be used for exporting key information coming from the **Sampler** or **Optimizer** object during the simulation. This XML node accepts the following attributes:
  - **class**, ***required string attribute***, main object class type. This string corresponds to the tag of the main object's type used in the input. For this role, only **'DataObjects'** can be used.
  - **type**, ***required string attribute***, the actual entity type. This attribute needs to specify the object type within the `DataObjects` object class. For example, the **type** attribute might be **'PointSet'**, **'HistorySet'**, etc.
    Note: Whether or not it is possible to export the **Sampler** solution depends on the **type**. Currently, only the Samplers in the **'Adaptive'** category and all Optimizers can export their solution into a **<SolutionExport>** entity. For Samplers, the **<Outputs>** node in the `DataObjects` needs to contain the goal **<Function>** name. For example, if **<Sampler>** is of type **'Adaptive'**, the **<SolutionExport>** needs to be of type **'PointSet'** and it will contain the coordinates, in the input space, that belong to the "Limit Surface". For Optimizers, the **<SolutionExport>** needs to be of type **'HistorySet'** and it will contains all the optimization trajectories, each as a history, that record how the variables are updated along each optimization trajectory.

- **<Output>**, ***string, required parameter*** identifies an entity to be used as output for this step. This XML node recognizes the following attributes:

- **class**, ***required string attribute***, main object class type. This string corresponds to the tag of the main object's type used in the input. For this role, only **'DataObjects'**, **'Databases'**, and **'OutStreams'** may be used.
- **type**, ***required string attribute***, the actual entity type. This attribute specifies the object type within the main object class. For example, if the **class** attribute is **'DataObjects'**, the **type** attribute might be **'PointSet'**.

Note: The number of **<Output>** nodes is unlimited.

Example:

```
<Steps>
  ...
  <MultiRun name='StepName1' pauseAtEnd='False' sleepTime='0.01'>
    <Input class='Files' type=''>anInputFile.i</Input>
    <Input class='Files' type=''>aFile</Input>
    <Sampler class='Samplers' type = 'Grid'>aGridName</Sampler>
    <Model class='Models' type='Code'>aCode</Model>
    <Output class='Databases' type='HDF5'>aDatabase</Output>
    <Output class='DataObjects' type='History'>aData</Output>
  </MultiRun >
  <MultiRun name='StepName2' pauseAtEnd='True' sleepTime='0.02'>
    <Input   class='Files' type=''>anInputFile.i</Input>
    <Input   class='Files' type=''>aFile</Input>
    <Sampler class='Samplers' type='Adaptive'>anAS</Sampler>
    <Model   class='Models' type='Code'>aCode</Model>
    <Output  class='Databases' type='HDF5'>aDatabase</Output>
    <Output  class='DataObjects' type='History'>aData</Output>
    <SolutionExport class='DataObjects' type='PointSet'>
      aTPS
    </SolutionExport>
  </MultiRun>
  ...
</Steps>
```

18.3 IOStep

As the name suggests, the **IOStep** is the step where the user can perform input/output operations among the different I/O entities available in RAVEN. This step type is used to:

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

- construct/update a *DataObject* from a *CSV* file contained in a directory;
- 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 (see section 14);
- store/retrieve a *ROM* or *ExternalModel* to/from an external *File* using Pickle module of Python. This function can be used to create and store ExternalModel or mathematical model (ROM) of fast solution trained to predict a response of interest of a physical system. These models 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.
- export a *ROM* or *ExternalModel* to an external FMI/FMU *File* using the RAVEN native FMI/FMU exporting capability

The specifications of this type of step must be defined within an **`<IOStep>`** XML block. This XML node can accept the following attributes:

- **`name`**, ***required string attribute***, user-defined name of this Step. **Note:** As for the other objects, this is the name that can be used to refer to this specific entity in the **`<RunInfo>`** block, under the **`<Sequence>`** node.
- **`pauseAtEnd`**, ***optional boolean/string attribute (case insensitive)***, if True (True values = True, yes, y, t), the code will pause at the end of the step, waiting for a user signal to continue. This is used in case one or more of the **Outputs** are of type **OutStreams**. For example, it can be used when an **OutStreams** of type **Plot** is output to the screen. Thus, allowing the user to interact with the **Plot** (e.g. rotate the figure, change the scale, etc.).
  Default: False.
- **`fromDirectory`**, ***optional string attribute***, The directory where the input files can be found when loading data from a file or series of files directly into a DataObject.

In the **`<IOStep>`** input block, the user specifies the objects that need to be used for the different allowable roles. This step accepts the following roles:

- **`<Input>`**, ***string, required parameter***, names an entity that is going to be used as a source (input) from which the information needs to be extracted. This XML node recognizes the following attributes:
  - **`class`**, ***required string attribute***, main object class type. This string corresponds to the tag of the main object's type used in the input. As already mentioned, the allowable main classes are **`'DataObjects'`**, **`'Databases'`**, **`'Models'`** and **`'Files'`**.
  - **`type`**, ***required string attribute***, the actual entity type. This attribute needs to specify the object type within the main object class. For example, if the **`class`** attribute

is **'DataObjects'**, the **type** attribute might be **'PointSet'**. If the **class** attribute is **'Models'**, the **type** attribute must be **'ROM'** or **'ExternalModel'** and if the **class** attribute is **'Files'**, the **type** attribute must be **' '**

- **<Output>**, ***string, required parameter*** names an entity to be used as the target (output) where the information extracted in the input will be stored. This XML node needs to contain the following attributes:
  - **class**, ***required string attribute***, main object class type. This string corresponds to the tag of the main object's type used in the input. The allowable main classes are **'DataObjects'**, **'Databases'**, **'OutStreams'**, **'Models'** and **'Files'**.
  - **type**, ***required string attribute***, the actual entity type. This attribute specifies the object type within the main object class. For example, if the **class** attribute is **'OutStreams'**, the **type** attribute might be **'Plot'**.

This step acts as a "transfer network" among the different RAVEN storing (or streaming) objects. The number of **<Input>** and **<Output>** nodes is unlimited, but should match. This step assumes a 1-to-1 mapping (e.g. first **<Input>** is going to be used for the first **<Output>**, etc.).
Note: This 1-to-1 mapping is not present when **<Output>** nodes are of **class** **'OutStreams'**, since **OutStreams** objects are already linked to a Data object in the relative RAVEN input block. In this case, the user needs to provide all of the "DataObjects" objects linked to the OutStreams objects (see the example below) in the **<Input>** nodes.

```
<Steps>
  ...
  <IOStep name='OutStreamStep'>
    <Input class='DataObjects'
       type='HistorySet'>aHistorySet</Input>
    <Input class='DataObjects' type='PointSet'>aTPS</Input>
    <Output class='OutStreams' type='Plot'>plot_hist
    </Output>
    <Output class='OutStreams' type='Print'>print_hist
    </Output>
    <Output class='OutStreams' type='Print'>print_tps
    </Output>
    <Output class='OutStreams' type='Print'>print_tp
    </Output>
  </IOStep>
  ...
  <IOStep name='PushDataObjectsIntoDatabase'>
    <Input  class='DataObjects'
       type='HistorySet'>aHistorySet</Input>
    <Input  class='DataObjects'     type='PointSet'>aTPS</Input>
```

```
    <Output class='Databases' type='NetCDF'>aDatabase</Output>
    <Output class='Databases' type='HDF5'>aDatabase</Output>
  </IOStep>
  ...
</Steps>
```

A summary of the objects that can go from/to other objects is shown in Table 13:

<Input>	**<Output>**	Resulting behavior
DataObject	Database	Store to On-Disk Database
	OutStream	Print or Plot Data
Database	DataObject	Load from On-Disk Database
File	DataObject	Load from On-Disk CSV
	ExternalModel	Load On-Disk Serialized ExternalModel
	ROM	Load On-Disk Serialized ROM
ROM	DataObject	Print ROM Metadata to CSV, XML
	File	Serialize ROM to Disk
ExternalModel	File	Serialize ExternalModel to Disk
	File	If ext is fmu ($myOut.fmu$) Serialize ExternalModel to FMU

Table 13: Object options for **<IOStep>** operations

As already mentioned, the **<IOStep>** can be used to export (serialize) a ROM or ExternalModel in a binary file. To use the exported ROM or ExternalModel in an external Python (or Python-compatible) code, the RAVEN framework must be present in end-user machine. The main reason for this is that the *Pickle* module uses the class definitions to template the reconstruction of the serialized object in memory.
In order to faciliate the usage of the serialized ROM or ExternalModel in an external Python code, the RAVEN team provided a utility class contained in :

```
./raven/scripts/externalROMloader.py
```

An example of how to use this utility class to load and use a serialized ROM (already trained) or ExternalModel is reported below: Example Python Function:

```
from externalROMloader import ravenROMexternal
import numpy as np
rom = ravenROMexternal("path_to_pickled_rom/ROM.pk",
                                        "path_to_RAVEN_framework")
request = {"x1":np.atleast_1d(Value1),"x2":np.atleast_1d(Value2)}
eval  = rom.evaluate(request)
print str(eval)
```

The module above can also be used to evaluate a ROM or ExternalModel from input file:

```
python ./raven/scripts/externalROMloader.py input_file.xml
```

The input file has the following format:

```
<?xml version="1.0" ?>
<external_rom>
  <RAVENdir>path_to_RAVEN_framework</RAVENdir>
  <ROMfile>ath_to_pickled_rom/ROM.pk</ROMfile>
  <evaluate>
    <x1>0. 1. 0.5</x1>
    <x2>0. 0.4 2.1</x2>
  </evaluate>
  <inspect>true</inspect>
  <outputFile>output_file_name</outputFile>
</external_rom>
```

The output of the above command would look like as follows:

```
<?xml version="1.0" ?>
<UROM>
  <settings>
    <Target>ans</Target>
    <name>UROM</name>
    <IndexSet>TensorProduct</IndexSet>
    <Features>[u'x1' u'x2']</Features>
    <PolynomialOrder>2</PolynomialOrder>
  </settings>
  <evaluations>
    <evaluation realization="1">
      <x2>0.0</x2>
      <x1>0.0</x1>
      <ans>-3.1696867353e-14</ans>
    </evaluation>
    <evaluation realization="2">
      <x2>0.4</x2>
      <x1>1.0</x1>
      <ans>1.4</ans>
    </evaluation>
    <evaluation realization="3">
      <x2>2.1</x2>
      <x1>0.5</x1>
      <ans>2.6</ans>
```

```
    </evaluation>
  </evaluations>
</UROM>
```

18.4 RomTrainer

The **RomTrainer** step type performs the training of a Reduced Order Model (aka Surrogate Mode). The specifications of this step must be defined within a **`<RomTrainer>`** block. This XML node accepts the attributes:

- **`name`**, ***required string attribute***, user-defined name of this step. **Note:** As for the other objects, this is the name that can be used to refer to this specific entity in the **`<RunInfo>`** block under **`<Sequence>`**.

In the **`<RomTrainer>`** input block, the user will specify the objects needed for the different allowable roles. This step accepts the following roles:

- **`<Input>`**, ***string, required parameter*** names an entity to be used as a source (input) from which the information needs to be extracted. This XML node accepts the following attributes:
  - **`class`**, ***required string attribute***, main object class type. This string corresponds to the tag of the main object's type used in the input. The only allowable main class is **`'DataObjects'`**.
  - **`type`**, ***required string attribute***, the actual entity type. This attribute specifies the object type within the main object class. For example, the **`type`** attribute might be **`'PointSet'`**. **Note:** Depending on which type of **`'DataObjects'`** is used, the ROM will be a Static or Dynamic (i.e. time-dependent) model. This implies that both **`'PointSet'`** and **`'HistorySet'`** are allowed (but not **`'DataSet'`** yet).
- **`<Output>`**, ***string, required parameter***, names a ROM entity that is going to be trained. This XML node recognizes the following attributes:
  - **`class`**, ***required string attribute***, main object class type. This string corresponds to the tag of the main objects type used in the input. The only allowable main class is **`'Models'`**.
  - **`type`**, ***required string attribute***, the actual entity type. This attribute needs to specify the object type within the main object class. The only type accepted here is, currently, **`'ROM'`**.

Example:

```
<Steps>
  ...
  <RomTrainer name='aStepNameStaticROM'>
    <Input  class='DataObjects'  type='PointSet'>aPS</Input>
    <Output class='Models'         type='ROM'     >aROM</Output>
  </RomTrainer>
    <RomTrainer name='aStepNameTimeDependentROM'>
    <Input    class='DataObjects'  type='HistorySet'>aHS</Input>
    <Output class='Models'           type='ROM'
       >aTimeDepROM</Output>
  </RomTrainer>
  ...
</Steps>
```

18.5 PostProcess

The **PostProcess** step is used to post-process data or manipulate RAVEN entities. It is aimed at performing a single action that is employed by a **Model** of type **PostProcessor**.

The specifications of this type of step is defined within a **`<PostProcess>`** XML block. This XML node specifies the following attributes:

- **`name`**, ***required string attribute***, user-defined name of this Step. **Note:** As for the other objects, this is the name that is used to refer to this specific entity in the **`<RunInfo>`** block under the **`<Sequence>`** node.
- **`pauseAtEnd`**, ***optional boolean/string attribute (case insensitive)***, if True (True values = True, yes, y, t), the code will pause at the end of the step, waiting for a user signal to continue. This is used in case one or more of the **Outputs** are of type **OutStreams**. For example, it can be used when an **OutStreams** of type **Plot** is output to the screen. Thus, allowing the user to interact with the **Plot** (e.g. rotate the figure, change the scale, etc.).
  Default: False.

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

- **`<Input>`**, ***string, required parameter***, names an entity to be used as input for the model specified in this step. This XML node accepts the following attributes:
  - **`class`**, ***required string attribute***, main object class type. This string corresponds to the tag of the main object's type used in the input. For example, **`'Files'`**, **`'DataObjects'`**, **`'Databases'`**, etc.

- `type`, ***required string attribute***, the actual entity type. This attribute specifies the object type within the main object class. For example, if the `class` attribute is `'DataObjects'`, the `type` attribute might be `'PointSet'`. **Note:** The class `'Files'` has no type (i.e. `type=''`).

Note: The `class` and, consequently, the `type` usable for this role depends on the particular type of **PostProcessor** being used. In addition, the user can specify as many `<Input>` nodes as needed by the model.

- `<Model>`, ***string, required parameter***, names an entity to be used as a model for this step. This XML node recognizes the following attributes:
  - `class`, ***required string attribute***, main object class type. This string corresponds to the tag of the main object's type used in the input. For this role, only `'Models'` can be used.
  - `type`, ***required string attribute***, the actual entity type. This attribute needs to specify the object type within the `'Models'` object class. The only type accepted here is `'PostProcessor'`.
- `<Output>`, ***string, required/optional parameter***, names an entity to be used as output for the PostProcessor. The necessity of this XML block and the types of entities that can be used as output depend on the type of **PostProcessor** that has been used as a **Model** (see section 15.5). This XML node specifies the following attributes:
  - `class`, ***required string attribute***, main object class type. This string corresponds to the tag of the main object's type used in the input.
  - `type`, ***required string attribute***, the actual entity type. This attribute specifies the object type within the main object class. For example, if the `class` attribute is `'DataObjects'`, the `type` attribute might be `'PointSet'`.

Note: The number of `<Output>` nodes is unlimited.

Example:

```
<Steps>
  ...
  <PostProcess name='PP1'>
      <Input  class='DataObjects'  type='PointSet' >aData</Input>
      <Model  class='Models' type='PostProcessor'>aPP</Model>
      <Output class='Files'  type=''>anOutputFile</Output>
  </PostProcess>
  ...
</Steps>
```

19 Existing Interfaces

19.1 Generic Interface

The GenericCode interface is meant to handle a wide variety of generic codes that take straightforward input files and produce output CSV files. There are some limitations for this interface. If a code:

- accepts a keyword-based input file with no cross-dependent inputs,
- has no more than one filetype extension per command line flag,
- and returns a CSV with the input parameters and output parameters,

the GenericCode interface should cover the code for RAVEN.

The GenericCode interface leverages a wildcard-based approach to editing input files. Using the special wildcard format `$RAVEN-$`, RAVEN parses text-based inputs and replaces the wildcards with sampled values. For example, consider RAVEN sampling variables named `initial_velocity` and `initial_angle`. Assume we're using a projectile tracking model with keyword based entry input files; for example,

```
initial_height = 0      # starting height, m
initial_angle = 35      # starting angle, degrees
initial_velocity = 40   # starting velocity, m/s
gravity = 9.8           # accel due to grav, m/s/s
auxfile = gen.two       # additional properties file
case = myOut            # output name (adds .csv)
```

Since we want to sample `initial_velocity` and `initial_angle`, we create a new template input and replace the values where samples should go with the wildcard and the variable name:

```
initial_height = 0      # starting height, m
initial_angle = $RAVEN-initial_angle$ # starting angle, degrees
initial_velocity = $RAVEN-initial_velocity$ # starting velocity, m/s
gravity = 9.8           # accel due to grav, m/s/s
auxfile = gen.two       # additional properties file
case = myOut            # output name (adds .csv)
```

See more discussion of replacing the output case and auxiliary file names below. When RAVEN samples values for the initial height and velocity, it will generate a new input file with those values in place, for example,

```
initial_height = 0      # starting height, m
initial_angle = 22.7589 # starting angle, degrees
```

```
initial_velocity = 47.2076 # starting velocity, m/s
gravity = 9.8            # accel due to grav, m/s/s
auxfile = gen.two        # additional properties file
case = myOut             # output name (adds .csv)
```

If a code contains cross-dependent data, the generic interface is not able to edit the correct values. For example, if a geometry-building script specifies inner_radius, outer_radius, and thickness, the generic interface cannot calculate the thickness given the outer and inner radius, or vice versa. In this case, the *function* method explained in the Samplers (see 10) and Optimizers (see 11) sections can be used.

An example of the code interface is shown here. The input parameters are read from the input files `gen.one` and `gen.two` respectively. The code is run using `python`, so that is part of the **`<clargs>`** node with the **`type`** equal **`'prepend'`**. The command line entry to normally run the code is

```
python poly_inp.py -i gen.one -a gen.two -o myOut
```

and produces the output `myOut.csv`.

Example:

```
<Code name="poly" subType="GenericCode">
  <executable>GenericInterface/poly_inp.py</executable>
  <inputExtentions>.one,.two</inputExtentions>
  <clargs type='prepend' arg='python'/>
  <clargs type='input'   arg='-i' extension='.one'/>
  <clargs type='input'   arg='-a' extension='.two'/>
  <clargs type='output'  arg='-o'/>
</Code>
```

If a code doesn't accept necessary Raven-editable auxiliary input files or output filenames through the command line, the GenericCode interface can also edit the input files and insert the filenames there. For example, in the previous example, say instead of `-a gen.two` and `-o myOut` in the command line, `gen.one` has the following lines:

```
...
auxfile = gen.two
case = myOut
...
```

Then, our example XML for the code would be

Example:

```
    <Code name="poly" subType="GenericCode">
      <executable>GenericInterface/poly_inp.py</executable>
      <inputExtentions>.one,.two</inputExtentions>
      <clargs   type='prepend' arg='python'/>
      <clargs   type='input'   arg='-i'  extension='.one'/>
      <fileargs type='input'   arg='two' extension='.two'/>
      <fileargs type='output'  arg='out'/>
    </Code>
```

and the corresponding template input file lines would be changed to read

```
...
auxfile = $RAVEN-two$
case = $RAVEN-out$
...
```

If a code has hard-coded output file names that are not changeable, the GenericCode interface can be invoked using the **<outputFile>** node in which the output file name (CSV only) must be specified. For example, in the previous example, say instead of `-a gen.two` and `-o myOut` in the command line, the code always produce a CSV file named "fixed_output.csv";

Then, our example XML for the code would be

Example:

```
    <Code name="poly" subType="GenericCode">
      <executable>GenericInterface/poly_inp.py</executable>
      <inputExtentions>.one,.two</inputExtentions>
      <clargs   type='prepend' arg='python'/>
      <clargs   type='input'   arg='-i'  extension='.one'/>
      <fileargs type='input'   arg='two' extension='.two'/>
      <outputFile>fixed_output.csv</outputFile>
    </Code>
```

In addition, the "wild-cards" above can contain two special and optional symbols:

- `:`, that defines an eventual default value;
- `|`, that defines the format of the value. The Generic Interface currently supports the following formatting options (* in the examples means blank space):
  - **plain integer**, in this case the value that is going to be replaced by the Generic Interface, will be left-justified with a string length equal to the integer value specified here (e.g. "`|6`", the value is left-justified with a string length of 6);

- **d**, signed integer decimal, the value is going to be formatted as an integer (e.g. if the value is 9 and the format "|10d", the replaced value will be formatted as follows: "*********9");
- **e**, floating point exponential format (lowercase), the value is going to be formatted as a float in scientific notation (e.g. if the value is 9.1234 and the format "|10.3e", the replaced value will be formatted as follows: "*9.123e+00");
- **E**, floating point exponential format (uppercase), the value is going to be formatted as a float in scientific notation (e.g. if the value is 9.1234 and the format "|10.3E", the replaced value will be formatted as follows: "*9.123E+00");
- **f or F**, floating point decimal format, the value is going to be formatted as a float in decimal notation (e.g. if the value is 9.1234 and the format "|10.3f", the replaced value will be formatted as follows: "*****9.123");
- **g**, floating point format. Uses lowercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise (e.g. if the value is 9.1234 and the format "|10.3g", the replaced value will be formatted as follows: "******9.12");
- **G**, floating point format. Uses uppercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise (e.g. if the value is 0.000009 and the format "|10.3G", the replaced value will be formatted as follows: "*****9E-06").

—

For example:

```
...
auxfile = $RAVEN-two:3$
case = $RAVEN-out:5|10$
...
```

Where,

- **:**, in case the variable "two" is not defined in the RAVEN XML input file, the Parser, will replace it with the value "3".;
- **|**, the value that is going to be replaced by the Generic Interface, will be left- justified with a string length of "10";

19.2 RAVEN Interface

The RAVEN interface is meant to provide the possibility to execute a RAVEN input file driving a set of SLAVE RAVEN calculations. For example, if the user wants to optimize the parameters of a

surrogate model (e.g. minimizing the distance between the surrogate predictions and the real data), he can achieve this task by setting up a RAVEN input file (master) that performs an optimization on the feature space characterized by the surrogate model parameters, whose training and validation assessment is performed in the SLAVE RAVEN runs.
There are some limitations for this interface:

- only one sub-level of RAVEN can be executed (i.e. if the SLAVE RAVEN input file contains the run of another RAVEN SLAVE, the MASTER RAVEN will error out)
- only data from Outstreams of type Print can be collected by the MASTER RAVEN
- only a maximum of two Outstreams can be collected (1 PointSet and 1 HistorySet)

Like for every other interface, most of the RAVEN workflow stays the same independently of which type of Model (i.e. Code) is used.
Similarly to any other code interface, the user provides paths to executables and aliases for sampled variables within the **`<Models>`** block. The **`<Code>`** block will contain attributes **`name`** and **`subType`**. **`name`** identifies that particular **`<Code>`** model within RAVEN, and **`subType`** specifies which code interface the model will use (In this case **`subType`**="RAVEN"). The **`<executable>`** block should contain the absolute or relative (with respect to the current working directory) path to the RAVEN framework script (**raven_framework**).
In addition to the attributes and xml nodes reported above, the RAVEN accepts the following XML nodes (required and optional):

- **`<outputDatabase>`**, ***string, required parameter*** will specify the **`<Database>`** that will be loaded as outputs of the INNER RAVEN. If this node is not specifed, **`<outputExportOutStreams>`** may be used instead.
- **`<outputExportOutStreams>`**, ***comma separated list, required parameter*** will specify the **`<OutStreams>`** that will be loaded as outputs of the SLAVE RAVEN. Maximum two **`<OutStreams>`** can be listed here (1 for PointSet and/or 1 for HistorySet).
- **`<conversion>`**, ***Node,optional parameter*** will specify details of conversion scripts to be used in creating the inner RAVEN input file. This node contains the following nodes:
  - **`<module>`**, ***Node, optional parameter*** a module for directly manipulating the xml structure of perturbed input files. This can be used to modify the template input file in arbitrary ways; however, it should be used with caution, and is considered an advanced method. This node has the following attribute:
    - **`source`**, ***string, required*** provides the path to the manipulation module including the module file itself. The following method should be defined in order to perform the input manipulation:

· ***modifyInput***, manipulates the input in arbitrary ways. This method takes two arguments. The first is the root **`<Simulation>`** node of the template input file that has already been modified with the perturbed samples (the object is a Python `xml.etree.ElementTree.Element` object). The second input is a dictionary with all the modification information used to previously modify the template xml. The method should return the modified root. Example:

```
import xml.etree.ElementTree as ET
def modifyInput(root, modDict):
  """
    Manipulate the inner RAVEN xml input.
    @ In, root, ET.Element, perturbed RAVEN input
    @ In, modDict, dictionary, modifications made to the input
    @ Out, root, ET.Element, modified RAVEN input
  """
  # adds the file <Input name='aux_inp'>auxfile.txt</Input> to
  filesNode = root.find('Files')
  newNode = ET.Element('Input')
  newNode.text = 'auxfile.txt'
  newNode.attrib['name'] = 'aux_inp'
  filesNode.append(newNode)
  return root
```

– **`<module>`**, ***Node, optional parameter*** contains the information about a specific conversion module (python file). This node can be repeated multiple times. This node has the following attribute:

  * **`source`**, ***string, required*** provides the path to the conversion module including the module file itself. There are two methods that can be placed in the conversion module:

    · ***manipulateScalarSampledVariables***, a method that is aimed to manipulate sampled variables and to create more in case needed. Example:

```
def manipulateScalarSampledVariables(sampledVariables):
  """
  This method is aimed to manipulate scalar variables.
  The user can create new variables based on the
  variables sampled by RAVEN
   @ In, sampledVariables, dict, dictionary of
       sampled variables ({"var1":value1,"var2":value2})
   @ Out, None, the new variables should be
                 added in the "sampledVariables" dictionary
  """
  newVariableValue =
    sampledVariables['Distributions|Uniform@name:a_dist|lowerB
```

```
    + 1.0
  sampledVariables['Distributions|Uniform@name:a_dist|upperBou
    newVariableValue
  return
```

· ***convertNotScalarSampledVariables***, a method that is aimed to convert not scalar variables (e.g. 1D arrays) into multiple scalar variables (e.g. **<constant>**(s) in a sampling strategy). This method is going to be required in case not scalar variables are detected by the interface. Example:

```
def convertNotScalarSampledVariables(noScalarVariables):
  """
  This method is aimed to convert not scalar
  variables into multiple scalar variables. The user MUST
   create new variables based on the not Scalar Variables
    sampled (and passed in) by RAVEN
  @ In, noScalarVariables, dict, dictionary of sampled
      variables that are not scalar ({"var1":1Darray1,"var2"
  @ Out, newVars, dict,  the new variables that have
      been created based on the not scalar variables
      contained in "noScalarVariables" dictionary
  """
  oneDimensionalArray =
      noScalarVariables['temperatureHistory']
  newVars = {}
  for cnt, value in enumerate(oneDimensionalArray):
    newVars['Samplers|MonteCarlo@name:myMC|constant'+
                '@name=temperatureHistory'+str(cnt)] =
                oneDimensionalArray[cnt]
  return newVars
```

The **<module>** node also takes the following node:

* **<variables>**, ***comma-separated list, required*** provides a comma-separated list of the variables from the MASTER RAVEN that need to be accessed by the conversion script module. The variables listed here use the pipe naming system (unaliased names).

Code input example:

```
<Code name="RAVENrunningRAVEN" subType="RAVEN">
  <executable>../../../raven_framework</executable>
  <outputExportOutStreams>
     HistorySetOutStream,PointSetOutStream
```

```
  </outputExportOutStreams>
  <conversion>
    <module source=/Users/username/whateverConversionModule.py>
      <variables>a,b,x,y</variables>
    </module>
  </conversion>
</Code>
```

Like for every other interface, the syntax of the variable names is important to make the parser understand how to perturb an input file.
For the RAVEN interface, a syntax inspired by the XPath nomenclature is used.

```
<Samplers>
  <MonteCarlo name="MC_external">
      ...
    <variable name="Models|ROM@subType:SciKitLearn@name:ROM1|C">
      <distribution>C_distrib</distribution>
    </variable>
    <variable
       name="Models|ROM@subType:SciKitLearn@name:ROM1|tol">
      <distribution>toll_distrib</distribution>
    </variable>
    <variable name="Samplers|Grid@name:'+
␣␣␣␣␣␣␣␣␣␣
   'GridName|variable@name:var1|grid@construction:equal@type:value@steps">
      <distribution>categorical_step_distrib</distribution>
    </variable>
    ...
  </MonteCarlo>
</Samplers>
```

In the above example, it can be inferred that each XML node (subnode) needs to be separated by a "—" separator. In addition, every time an XML node has attributes, the user can specify them using the "@" separator to specify a value for them. The first variable above will be pointing to the following XML sub-node (**<C>**):

```
<Models>
  <ROM name="ROM1" subType="SciKitLearn">
      ...
      <C>10.0</C>
    ...
 </ROM>
</Models>
```

The second variable above will be pointing to the following XML sub-node (**<tol>**):

```
<Models>
  <ROM name="ROM1" subType="SciKitLearn">
      ...
      <tol>0.0001</tol>
     ...
 </ROM>
</Models>
```

The third variable above will be pointing to the following XML attribute (**steps**):

```
<Samplers>
  <Grid name="GridName">
      ...
    <variable name="var1">
       ...
      <grid construction="equal" type="value" steps="1">0
         1</grid>
       ...
    </variable>

    ...
  </MonteCarlo>
</Samplers>
```

The above nomenclature must be used for all the variables to be sampled and for the variables generated by the two methods contained, in case, in the module that gets specified by the **<conversionModule>** in the **<Code>** section.
Finally the SLAVE RAVEN input file (s) must be "tagged" with the attribute **type="raven"** in the Files section. For example,

```
<Files>
    <Input name="slaveRavenInputFile" type="raven" >
      test_rom_trainer.xml
    </Input>
</Files>
```

19.2.1 ExternalXML and RAVEN interface

Care must be taken if the SLAVE RAVEN uses **<ExternalXML>** nodes. In this case, each file containing external XML nodes must be added in the **<Step>** as an **<Input>** class **Files** to

make sure it gets copied to the individual run directory. The type for these files can be anything, with the exception of type **'raven'**.

19.3 RELAP5 Interface

19.3.1 Sequence

In the **<Sequence>** section, the names of the steps declared in the **<Steps>** block should be specified. As an example, if we called the first multirun "Grid_Sampler" and the second multirun "MC_Sampler" in the sequence section we should see this:

```
<Sequence>Grid_Sampler,MC_Sampler</Sequence>
```

19.3.2 batchSize and mode

For the **<batchSize>** and **<mode>** sections please refer to the **<RunInfo>** block in the previous chapters.

19.3.3 RunInfo

After all of these blocks are filled out, a standard example RunInfo block may look like the example below:

```
<RunInfo>
  <WorkingDir>~/workingDir</WorkingDir>
  <Sequence>Grid_Sampler,MC_Sampler</Sequence>
  <batchSize>1</batchSize>
  <mode>mpi</mode>
  <expectedTime>1:00:00</expectedTime>
  <ParallelProcNumb>1</ParallelProcNumb>
</RunInfo>
```

19.3.4 Files

In the **<Files>** section, as specified before, all of the files needed for the code to run should be specified. In the case of RELAP5, the files typically needed are:

- RELAP5 Input file
- Table file or files that RELAP needs to run

Example:

```
<Files>
  <Input name='tpfh2o' type=''>tpfh2o</Input>
  <Input name='inputrelap.i' type=''>X10.i</Input>
</Files>
```

It is a good practice to put inside the working directory all of these files and also:

- the RAVEN input file
- the license for the executable of RELAP5

It is important to notice that the interface output collection relies on the MINOR EDITS. The user must specify the MINOR EDITS block and those variables are the only one the INTERFACE will read and make available to RAVEN. In addition, it is important to notice that:

- **the simulation time is stored in a variable called *"time"*;**
- **all the variables specified in the MINOR EDIT block are going to be converted using underscores (e.g. an edit such as** $301 \quad p \quad 345010000$ **will be named in the converted CSVs as** $p_345010000$**).In addition, if a variable contains spaces, the trailing spaces are going to be removed and internal spaces are replaced with underscores (e.g.** $HTTEMP113100812$ **will become** $HTTEMP_1131008_12$.

Remember also that a RELAP5 simulation run is considered successful (i.e., the simulation did not crash) if it terminates with the following message: Transient terminated by end of time step cards or Transient terminated by trip

If the a RELAP5 simulation run stops with messages other than this one (e.g., " Transient terminated by failure.") than the simulation is considered as crashed, i.e., it will not be saved. Hence, it is strongly recommended to set up the RELAP5 input file so that the simulation exiting conditions are set through control logic trip variables (e.g., simulation mission time and clad temperature equal to clad failure temperature).

19.3.5 Models

For the **<Models>** block here is a standard example of how it would look when using RELAP5 as the external model:

```
<Models>
  <Code name='MyRELAP' subType='Relap5'>
    <executable>~/path_to_the_executable</executable>
  </Code>
</Models>
```

In case the **multi-deck** approach is used in RELAP5, the interface is going to load all the outputs in one CSV RAVEN is going to read. This means that all the decks' outputs are going to be loaded in one of the Output of RAVEN. In case the user wants to select the outputs coming from only one deck, the following XML node needs to be specified:

- **<outputDeckNumber>**, ***integer, optional parameter***, the deck number from which the results needs to be retrieved.
  Default: all.

In addition, if some command line parameters need to be passed to RELAP5 (e.g. "-r $restartFileWithCustomName.r$"), the user might use (optionally) the **<clargs>** XML nodes.

```
<Models>
  <Code name='MyRELAP' subType='Relap5'>
    <executable>~/path_to_the_executable</executable>
    <outputDeckNumber>1</outputDeckNumber>
    <clargs type="text" arg="-r␣restartFileWithCustomName.r"/>
  </Code>
</Models>
```

An additional feature of the RELAP5 Code interface is the possibility to specify operation based on the value of user-inputted cards. For example, let's assume the values in cards 1180801:2 and 1180802:2 must come from a calculation based on sampled variables (e.g. 20100154:2 and 20100155:2), the user can specify the following XML node:

- **<operator>**, ***XML node, optional parameter***, The operator block. This XML node must contain the following attribute:
  - *variables*, ***comma separated list, required parameter***, The list of variables (coming from a Sampler) that will be used in the **<expression>** XML node.

Within the **<operator>** the following XML sub-nodes must be specified:

- **<expression>**, ***string, required parameter***, The string representing the expression to be performed. The "card" (if needed to be used) must be identified with the token %card% and it will be replaced with the values of the cards (specified in the XML node **<cards>**) from the original input file. In this expression, all the functions available in the Python $math$ module can be used (e.g. $sqrt$, exp, sin, etc.).
- **<cards>**, ***comma separated list, required parameter***, The list of cards in the original input file whose values need to be replaced by the value resulting from the expression contained in **<expression>**.

Note: the user can specify as many **<operator>** nodes as needed.

An example is reported below:

```
<Models>
  <Code name='MyRELAP' subType='Relap5'>
    <executable>~/path_to_the_executable</executable>
    ...
    <operator variables="20100154:2,20100155:2">
      <expression> %card%*20100155:2*2./20100155:2</expression>
      <cards>1180801:2,1180802:2,1180901:3</cards>
    </operator>
    ...
  </Code>
</Models>
```

19.3.6 Distributions

The **<Distribution>** block defines the distributions that are going to be used for the sampling of the variables defined in the **<Samplers>** block. For all the possibile distributions and all their possible inputs please see the chapter about Distributions (see 9). Here we give a general example of three different distributions:

```
<Distributions verbosity='debug'>
  <Triangular name='BPfailtime'>
    <apex>5.0</apex>
    <min>4.0</min>
    <max>6.0</max>
  </Triangular>
  <LogNormal name='BPrepairtime'>
    <mean>0.75</mean>
```

```
    <sigma>0.25</sigma>
  </LogNormal>
  <Uniform name='ScalFactPower'>
    <lowerBound>1.0</lowerBound>
    <upperBound>1.2</upperBound>
  </Uniform>
 </Distributions>
```

It is good practice to name the distribution something similar to what kind of variable is going to be sampled, since there might be many variables with the same kind of distributions but different input parameters.

19.3.7 Samplers

In the **<Samplers>** block we want to define the variables that are going to be sampled. **Example**: We want to do the sampling of 3 variables:

- Battery Fail Time
- Battery Repair Time
- Scaling Factor Power Rate

We are going to sample these 3 variables using two different sampling methods: grid and MonteCarlo.

In RELAP5, the sampler reads the variable as, given the name, the first number is the card number and the second number is the word number. In this example we are sampling:

- For card 0000588 (trip) the word 6 (battery failure time)
- For card 0000575 (trip) the word 6 (battery repair time)
- For card 20210000 (reactor power) the word 4 (reactor scaling factor)

We proceed to do so for both the Grid sampling and the MonteCarlo sampling.

```
<Samplers verbosity='debug'>
  <Grid name='Grid_Sampler' >
    <variable name='0000588:6'>
      <distribution>BPfailtime</distribution>
```

```
      <grid type='value' construction='equal'  steps='10'>0.0
         28800</grid>
    </variable>
    <variable name='0000575:6'>
      <distribution>BPrepairtime</distribution>
      <grid type='value' construction='equal' steps='10'>0.0
         28800</grid>
    </variable>
    <variable name='20210000:4'>
      <distribution>ScalFactPower</distribution>
      <grid type='value' construction='equal' steps='10'>1.0
         1.2</grid>
    </variable>
  </Grid>
  <MonteCarlo name='MC_Sampler'>
     <samplerInit>
       <limit>1000</limit>
     </samplerInit>
    <variable name='0000588:6'>
      <distribution>BPfailtime</distribution>
    </variable>
    <variable name='0000575:6'>
      <distribution>BPrepairtime</distribution>
    </variable>
    <variable name='20210000:4'>
      <distribution>ScalFactPower</distribution>
    </variable>
  </MonteCarlo>
</Samplers>
```

In case the RELAP5 input file is a multi-deck, the user can specify the deck to which each sampled variable corresponds to. As an example, the following sampling strategy:

```
<MonteCarlo name='MC_Sampler'>
   <samplerInit>
     <limit>1000</limit>
   </samplerInit>
  <variable name='1|0000588:6'>
    <distribution>BPfailtime</distribution>
  </variable>
  <variable name='2|0000575:6'>
    <distribution>BPrepairtime</distribution>
  </variable>
```

```
</MonteCarlo>
</Samplers>
```

performs:

- the sampling of the distribution
  <BPfailtime> and it provides the sampled value to the 6th word of card 0000588 for the first deck
- the sampling of the distribution
  <BPrepairtime> and it provides the sampled value to the 6th word of card 0000575 for the second deck

It can be seen that each variable is connected with a proper distribution defined in the **<Distributions>** block (from the previous example). The following demonstrates how the input for the first variable is read.

We are sampling a a variable situated in word 6 of the card 0000588 using a Grid sampling method. The distribution that this variable is following is a Triangular distribution (see section above). We are sampling this variable beginning from 0.0 in 10 *equal* steps of 2880. In case of Dynamic Event Tree-based sampling, the input is very similar to the other sampling strategies with the only "limitation" that the sampled variables directly linked to the Dynamic Event Tree must be part of a RELAP5 trip. In case, other variables must be sampled, they are considered "epistemic" variables and should be sampled using the Hybrid Dynamic Event Tree approach.

For example, the Dynamic Event Tree sampling and the Hybrid Dynamic Event Tree sampling would look like the following:

```
<Samplers verbosity='debug'>
  <DynamicEventTree name='DET'>
    <variable name='414:6'>
      <distribution>endtimedist</distribution>
      <grid type='CDF' construction='custom'>0.1 0.3 0.99</grid>
    </variable>
    <variable name='454:6'>
      <distribution>endtime2dist</distribution>
      <grid type='CDF' construction='custom'>0.11 0.5 0.99</grid>
    </variable>
  </DynamicEventTree>

  <DynamicEventTree name='HDET'>
    <HybridSampler type="MonteCarlo">
      <!-- in here we specify the epistemic like variables -->
```

```
      <samplerInit>
        <limit>10</limit>
      </samplerInit>
      <variable name="200:1">
        <distribution>missionTimeDist</distribution>
      </variable>
    </HybridSampler>
    <variable name='414:6'>
      <distribution>endtimedist</distribution>
      <grid type='CDF' construction='custom'>0.1 0.3 0.99</grid>
    </variable>
    <variable name='454:6'>
      <distribution>endtime2dist</distribution>
      <grid type='CDF' construction='custom'>0.11 0.5 0.99</grid>
    </variable>
  </DynamicEventTree>
</Samplers>
```

19.3.8 Steps

For a RELAP interface, the **<MultiRun>** step type will most likely be used. First, the step needs to be named: this name will be one of the names used in the **<Sequence>** block. In our example, `Grid_Sampler` and `MC_Sampler`.

```
     <MultiRun name='Grid_Sampler' verbosity='debug'>
```

With this step, we need to import all the files needed for the simulation:

- RELAP input file
- element tables – tpfh2o

```
    <Input   class='Files' type=''>inputrelap.i</Input>
    <Input   class='Files' type=''>tpfh2o</Input>
```

We then need to define which model will be used:

```
    <Model  class='Models' type='Code'>MyRELAP</Model>
```

We then need to specify which Sampler is used, and this can be done as follows:

```
    <Sampler class='Samplers' type='Grid'>Grid_Sampler</Sampler>
```

And lastly, we need to specify what kind of output the user wants. For example the user might want to make a database (in RAVEN the database created is an HDF5 file). Here is a classical example:

```
    <Output  class='Databases' type='HDF5'>Grid_out</Output>
```

Following is the example of two MultiRun steps which use different sampling methods (grid and Monte Carlo), and creating two different databases for each one:

```
<Steps verbosity='debug'>
  <MultiRun name='Grid_Sampler' verbosity='debug'>
    <Input   class='Files' type=''>inputrelap.i</Input>
    <Input   class='Files'     type=''    >tpfh2o</Input>
    <Model   class='Models'    type='Code'>MyRELAP</Model>
    <Sampler class='Samplers'  type='Grid'>Grid_Sampler</Sampler>
    <Output  class='Databases' type='HDF5'>Grid_out</Output>
  </MultiRun>
  <MultiRun name='MC_Sampler' verbosity='debug'
     re-seeding='210491'>
    <Input   class='Files' type=''>inputrelap.i</Input>
    <Input   class='Files'     type=''          >tpfh2o</Input>
    <Model   class='Models'    type='Code'      >MyRELAP</Model>
    <Sampler class='Samplers'
       type='MonteCarlo'>MC_Sampler</Sampler>
    <Output  class='Databases' type='HDF5'      >MC_out</Output>
  </MultiRun>
</Steps>
```

19.3.9 Databases

As shown in the **<Steps>** block, the code is creating two database objects called `Grid_out` and `MC_out`. So the user needs to input the following:

```
<Databases>
  <HDF5 name="Grid_out" readMode="overwrite"/>
  <HDF5 name="MC_out" readMode="overwrite"/>
</Databases>
```

As listed before, this will create two databases. The files will have names corresponding to their **name** appended with the .h5 extension (i.e. `Grid_out.h5` and `MC_out.h5`).

19.3.10 Modified Version of the Institute of Nuclear Safety System Incorporated (Japan)

The Institute of Nuclear Safety System Incorporated (Japan) has modified the **RELAP5** source code in order to be able to control some additional parameters from an auxiliary input file (**modelPar.inp**). In order to use this interface, the user needs to input the $subType$ attribute**Relap5inssJp**:

```
<Models>
  <Code name='MyRELAP' subType='Relap5'>
    <executable>~/path_to_the_executable</executable>
    <!-- here is taking the output from the first deck only -->
    <outputDeckNumber>1</outputDeckNumber>
  </Code>
</Models>
```

For perturbing such input file, the approach presented in section 19.1 (Generic Interface) has been employed. For the standard **RELAP5** input, the same approach previously in this section is used. For example, in the following Sampler block, the card 9100101 is perturbed with the same approach used in standard **RELAP5**; in addition, the variable $modelParTest$ is going to be perturbed in the **modelPar.inp** input file.

```
    <MonteCarlo name="mc_loca">
      <samplerInit>
        <limit>1</limit>
      </samplerInit>
      <variable name="9100101:3">
        <distribution>break_size</distribution>
      </variable>
      <variable name="modelParTest">
          <distribution>break_size</distribution>
      </variable>
    </MonteCarlo>
```

19.4 RELAP7 Interface

This section covers the input specifications for running RELAP7 through RAVEN. It is important to notice that this short explanation assumes that the reader already knows how to use the control logic system in RELAP7. Since the presence of the control logic system in RELAP7, this code interface is different with respect to the others and uses some special keyword available in RAVEN (see the following).

19.4.1 Files

In the **<Files>** section, as specified before, all of the files needed for the code to run should be specified. In the case of RELAP7, the files typically needed are the following:

- RELAP7 Input file
- Control Logic file

Example:

```
<Files>
  <Input name='nat_circ.i' type=''>nat_circ.i</Input>
  <Input name='control_logic.py' type=''>control_logic.py</Input>
</Files>
```

The RAVEN/RELAP7 interface recognizes as RELAP7 inputs the files with the extensions "*.i", "*.inp" and "*.in".

19.4.2 Models

For the **<Models>** block RELAP7 uses the RAVEN executable, since through this executable the stochastic environment gets activated (possibility to sample parameters directly in the control logic system) Here is a standard example of what can be used to use RELAP7 as the model:

```
<Models>
    <Code name='MyRAVEN'
       subType='RAVEN'><executable>~path/to/RAVEN-opt</executable></Code>
</Models>
```

19.4.3 Distributions

As for all the other codes interfaces the **<Distributions>** block needs to be specified in order to employ as sampling strategy (e.g. MonteCarlo, Stratified, etc.). In this block, the user specifies the distributions that need to be used. Once the user defines the distributions in this block, RAVEN activates the Distribution environment in the RAVEN/RELAP7 control logic system. The sampling of the parameters is then performed directly in the control logic input file.

For example, let's consider the sampling of a normal distribution for the primary pressure in RELAP7:

```
<Distributions>
 <Normal name="Prim_Pres">
 <mean>1000000</mean>
 <sigma>100<sigma/>
 </Normal>
</Distributions>
```

In order to change a parameter (independently on the sampling strategy), the control logic input file should be modified as follows:

```
def initial_function(monitored, controlled, auxiliary)
    print("monitored",monitored,"controlled",
    controlled,"auxiliary",auxiliary)

    controlled.pressureInPressurizer =
     distributions.Prim_Pres.getDistributionRandom()
    return
```

19.4.4 Samplers

In the **<Samplers>** block, all the variables that needs to be sampled must be specified. In case some of these variables are directly sampled in the Control Logic system, the **<variable>** needs to be replaced with **<Distribution>**. In this way, RAVEN is able to understand which variables needs to be directly modified through input file (i.e. modifying the original input file *.i) and which variables are going to be "sampled" through the control logic system. For the example, we are performing Grid Sampling. The global initial pressure wasn't specified in the control logic so it is going to be specified using the node **<variable>**. The "pressureInPressurizer" variable is instead sampled in the control logic system; for this reason, it is going to be specified using the node **<Distribution>**. For example,

```
<Samplers>
 <Grid name="MC_samp">
   <samplerInit> <limit>500</limit> </samplerInit>
   <variable name="GlobalParams|global_init_P">
      <distribution>Prim_Pres</distribution>
      <grid construction="equal" steps="10" type="CDF">0.0
         1.0</grid>
   </variable>
   <Distribution name="pressureInPressurizer">
      <distribution>Prim_Pres</distribution>
```

```
      <grid construction="equal" steps="10" type="CDF">0.0
         1.0</grid>
   </Distribution>
 </Grid>
</Samplers>
```

19.5 MooseBasedApp Interface

19.5.1 Files

In the **<Files>** section, as specified before, all of the files needed for the code to run should be specified. In the case of any MooseBasedApp, the files typically needed are the following:

- MooseBasedApp GetPot input file
- Restart Files (if the calculation is instantiated from a restart point)
- Mesh Files (in case the mesh is externally specified)
- Any other generic input file (CSVs with Power histories, boundary conditions files, etc.)

Example:

```
<Files>
  <Input name='mooseBasedApp.i' type=''>mooseBasedApp.i</Input>
  <Input name='0020_mesh.cpr' type=''>0020_mesh.cpr</Input>
  <Input name='0020.xdr.0000' type="">0020.xdr.0000</Input>
  <Input name='0020.rd-0' type="">0020.rd-0</Input>
  <Input name='exodus_mesh.e' type="">exodus_mesh.e</Input>
  <Input name='a_generic_additional_input_file.csv'
     type="Generic">a_generic_additional_input_file.csv</Input>
</Files>
```

If any file is tagged with the type `Generic`, it will be perturbable with the approach (wildcards) explained in the generic code interface (see 19.1).

19.5.2 Models

In the **<Models>** block particular MooseBasedApp executable needs to be specified. Here is a standard example of what can be used to use with a typical MooseBasedApp (Bison) as the model:

```
<Models>
    <Code name='MyMooseBasedApp'
       subType='MooseBasedApp'><executable>~path/to/Bison-opt</executable></
</Models>
```

19.5.3 Distributions

The **<Distributions>** block defines the distributions that are going to be used for the sampling of the variables defined in the **<Samplers>** block. For all the possible distributions and all their possible inputs please see the chapter about Distributions (see 9). Here we give a general example of three different distributions:

```
<Distributions>
    <Normal name='ThermalConductivity1'>
        <mean>1</mean>
        <sigma>0.001</sigma>
        <lowerBound>0.5</lowerBound>
        <upperBound>1.5</upperBound>
    </Normal>
    <Normal name='SpecificHeat'>
        <mean>1</mean>
        <sigma>0.4</sigma>
        <lowerBound>0.5</lowerBound>
        <upperBound>1.5</upperBound>
    </Normal>
    <Triangular name='ThermalConductivity2'>
        <apex>1</apex>
        <min>0.1</min>
        <max>4</max>
    </Triangular>
</Distributions>
```

It is good practice to name the distribution something similar to what kind of variable is going to be sampled, since there might be many variables with the same kind of distributions but different input parameters.

19.5.4 Samplers

In the **<Samplers>** block we want to define the variables that are going to be sampled. **Example**: We want to do the sampling of 3 variables:

- Thermal Conductivity of the Fuel;
- Specific Heat Transfer Ratio of the Cladding;
- Thermal Conductivity of the Cladding.

We are going to sample these 3 variables using two different sampling methods: Grid and Monte-Carlo.

In order to perturb any MooseBasedApp, the user needs to specify the variables to be sampled indicating the path to the value separated with the symbol "|". For example, if the variable that we want to perturb is specified in the input as follows:

```
[Materials]
  ...
  [./heatStructure]
     ...
     thermal_conductivity = 1.0
     ...
  [../]
  ...
[]
```

the variable name in the Sampler input block needs to be named as follows:

```
...
<Samplers>
  <aSampler name='aUserDefinedName' >
    <variable
       name='Materials|heatStructure|thermal_conductivity'>
      ...
    </variable>
  </aSampler>
</Samplers>
...
```

In case some variables in external (`Generic` input files) need to be perturbed, the wildcard approach can be used (for those variables):

```
...
<Samplers>
  <aSampler name='aUserDefinedName' >
    <variable name='aWildCard1'>
      ...
    </variable>
        <variable name='aWildCard2'>
      ...
    </variable>
    <variable
       name='Materials|heatStructure|thermal_conductivity'>
      ...
    </variable>
  </aSampler>
</Samplers>
...
```

In this case the tagged file (`Generic`) will be parsed to find the variables `$RAVEN-aWildCard1$` and `$RAVEN-aWildCard1$` and to replace their values with the corresponding sampled variables (for more details, see 19.1)

In this example, we proceed to do so for both the Grid sampling and the Monte-Carlo sampling.

```
<Samplers verbosity='debug'>
    <Grid name='myGrid'>
      <variable
         name='Materials|heatStructure1|thermal_conductivity' >
        <distribution>ThermalConductivity1</distribution>
        <grid          type='value' construction='custom' >0.6
           0.7 0.8</grid>
      </variable>
      <variable name='Materials|heatStructure1|specific_heat' >
        <distribution >SpecificHeat</distribution>
        <grid          type='CDF'    construction='custom'>0.5
           1.0 0.0</grid>
      </variable>
      <variable
         name='Materials|heatStructure2|thermal_conductivity'>
        <distribution  >ThermalConductivity2</distribution>
        <grid type='value' upperBound='4' construction='equal'
           steps='1'>0.5</grid>
      </variable>
```

```
      <variable name='aWildCard1'>
        <distribution  >ThermalConductivity2</distribution>
        <grid type='value' upperBound='4' construction='equal'
           steps='1'>0.5</grid>
      </variable>
    </Grid>
  <MonteCarlo name='MC_Sampler' limit='1000'>
      <variable
         name='Materials|heatStructure1|thermal_conductivity' >
        <distribution>ThermalConductivity1</distribution>
      </variable>
      <variable name='Materials|heatStructure1|specific_heat' >
        <distribution >SpecificHeat</distribution>
      </variable>
      <variable
         name='Materials|heatStructure2|thermal_conductivity'>
        <distribution  >ThermalConductivity2</distribution>
      </variable>
      <variable name='aWildCard1'>
        <distribution  >ThermalConductivity2</distribution>
      </variable>
  </MonteCarlo>
</Samplers>
```

19.5.5 Steps

For a MooseBasedApp, the **<MultiRun>** step type will most likely be used, as first step. First, the step needs to be named: this name will be one of the names used in the **<Sequence>** block. In our example, `Grid_Sampler` and `MC_Sampler`.

```
     <MultiRun name='Grid_Sampler' >
```

With this step, we need to import all the files needed for the simulation:

- MooseBasedApp YAML input file;
- eventual restart files (optional);
- other auxiliary files (e.g., powerHistory tables, etc.).

```
    <Input   class='Files' type=''>mooseBasedApp.i</Input>
    <Input   class='Files' type=''>0020_mesh.cpr</Input>
    <Input   class='Files' type=''>0020.xdr.0000</Input>
    <Input   class='Files' type=''>0020.rd-0</Input>
```

We then need to define which model will be used:

```
    <Model  class='Models' type='Code'>MyMooseBasedApp</Model>
```

We then need to specify which Sampler is used, and this can be done as follows:

```
    <Sampler class='Samplers' type='Grid'>Grid_Sampler</Sampler>
```

And lastly, we need to specify what kind of output the user wants. For example the user might want to make a database (in RAVEN the database created is an HDF5 file) and a DataObject of type PointSet, to use in sub-sequential post-processing. Here is a classical example:

```
    <Output  class='Databases' type='HDF5'>MC_out</Output>
    <Output  class='DataObjects'
       type='PointSet'>MCOutData</Output>
```

Following is the example of two MultiRun steps which use different sampling methods (grid and Monte Carlo), and creating two different databases for each one:

```
<Steps verbosity='debug'>
  <MultiRun name='Grid_Sampler' verbosity='debug'>
    <Input  class='Files' type=''>mooseBasedApp.i</Input>
    <Input  class='Files' type=''>0020_mesh.cpr</Input>
    <Input  class='Files' type='' >0020.xdr.0000</Input>
    <Input  class='Files' type=''>0020.rd-0</Input>
    <Model  class='Models'    type='Code'>MyMooseBasedApp</Model>
    <Sampler class='Samplers'  type='Grid'>Grid_Sampler</Sampler>
    <Output  class='Databases' type='HDF5'>Grid_out</Output>
    <Output  class='DataObjects'
       type='PointSet'>gridOutData</Output>
  </MultiRun>
  <MultiRun name='MC_Sampler' verbosity='debug'
     re-seeding='210491'>
    <Input  class='Files' type=''>mooseBasedApp.i</Input>
    <Input  class='Files' type=''>0020_mesh.cpr</Input>
    <Input  class='Files' type='' >0020.xdr.0000</Input>
    <Input  class='Files' type=''>0020.rd-0</Input>
    <Model  class='Models'    type='Code'>MyMooseBasedApp</Model>
```

```
    <Sampler class='Samplers'  type='MonteCarlo'
       >MC_Sampler</Sampler>
    <Output  class='Databases' type='HDF5' >MC_out</Output>
    <Output  class='DataObjects'
       type='PointSet'>MCOutData</Output>
  </MultiRun>
</Steps>
```

19.5.6 Databases

As shown in the **<Steps>** block, the code is creating two database objects called `Grid_out` and `MC_out`. So the user needs to input the following:

```
<Databases>
  <HDF5 name="Grid_out" readMode="overwrite"/>
  <HDF5 name="MC_out" readMode="overwrite"/>
</Databases>
```

As listed before, this will create two databases. The files will have names corresponding to their **name** appended with the .h5 extension (i.e. `Grid_out.h5` and `MC_out.h5`).

19.5.7 DataObjects

As shown in the **<Steps>** block, the code is creating two DataObjects of type PointSet called `gridOutData` and `MCOutData`. So the user needs to input the following:

```
<DataObjects>
    <PointSet name='gridOutData'>
      <Input>
          Materials|heatStructure2|thermal_conductivity,
          Materials|heatStructure1|specific_heat,
          Materials|heatStructure2|thermal_conductivity
      </Input>
      <Output>aveTempLeft</Output>
    </PointSet>
    <PointSet name='MCOutData'>
      <Input>
          Materials|heatStructure2|thermal_conductivity,
          Materials|heatStructure1|specific_heat,
          Materials|heatStructure2|thermal_conductivity
```

```
      </Input>
      <Output>aveTempLeft</Output>
    </PointSet>
</DataObjects>
```

As listed before, this will create two DataObjects that can be used in sub-sequential post-processing.

19.5.8 OutStreams

As fully explained in section 14, if the user want to print out or plot the content of a **DataObjects**, he needs to create an **OutStream** in the **<OutStreams>** XML block.
As it shown in the example below, for MooseBasedApp (and any other Code interface that might use the symbol | for the Sampler's variable syntax), in the Plot **<x>** and **<y>** specification, the user needs to utilize curly brackets.

```
<OutStreams>
  <Print name='gridOutDataDumpCSV'>
    <type>csv</type>
    <source>gridOutData</source>
  </Print>
   <Plot verbosity='debug' name='test'   overwrite='False'>
    <plotSettings>
       <plot>
        <type>line</type>
        <x>MCOutData|Input|{Materials|heatStructure2|thermal_conductivity}</
        <y>MCOutData|Output|aveTempLeft</y>
        <kwargs><color>blue</color></kwargs>
      </plot>
    </plotSettings>
    <actions><how>screen,png</how></actions>
  </Plot>
</OutStreams>
```

19.6 MooseVPP Interface

The Moose Vector Post Processor is used mainly in the solid mechanics analysis. This interface loads the values of the vector ouput processor to a **<DataObjects>** object.

To use this interface the [DomainIntegral] needs to be present in the MooseBasedApp's input file and the subnode **<fileargs>** should be defined in the subnode **<Code>** in the **<Models>**

block of the RAVEN input file. The **<fileargs>** is required to have attributes with the below specified values:

- **type**, ***string, required field***, must be ”MooseVPP”
- **arg**, ***string, required field***, the string value attached to the vector post processor action while creating the output files.

This interface is actually identical to the MooseBasedApp interface, however there is few constraints on defining the output values of the post processor. The definition of these outputs in the **<DataObjects>** depends on the definition of the [DomainIntegral].

The location of the value outputted is defined as *ID#* and the value is as *value#*. The ”#” defines the number of the location. The example below contains 3 locations in the [DomainIntegral] where the values are outputted.

Example:

```
...
 <Models>
   <Code name="MOOSETestApp" subType="MooseBasedApp">
     <executable>%FRAMEWORK_DIR%/../../moose/
       modules/combined/modules-%METHOD%</executable>
     <fileargs type = "MooseVPP" arg = "_J_1_" />
     <alias variable = "poissonsRatio" >
       Materials|stiffStuff|poissons_ratio</alias>
     <alias variable = "youngModulus"  >
       Materials|stiffStuff|youngs_modulus</alias>
   </Code>
 </Models>
...
 <DataObjects>
   <PointSet name="collset">
     <Input>youngModulus,poissonsRatio</Input>
     <Output>ID1,ID2,ID3,value1,value2,value3</Output>
   </PointSet>
 </DataObjects>
...
```

19.7 OpenModelica Interface

OpenModelica (`http://www.openmodelica.org`) is an open souce implementation of the Modelica simulation language. Modelica is "a non-proprietary, object-oriented, equation based language to conveniently model complex physical systems containing, e.g., mechanical, electrical, electronic, hydraulic, thermal, control, electric power or process-oriented subcomponents."[1]. Modelica models are specified in text files with a file extension of .mo. A standard Modelica example called BouncingBall which simulates the trajectory of an object falling in one dimension from a height is shown as an example:

```
model BouncingBall
  parameter Real e=0.7 "coefficient␣of␣restitution";
  parameter Real g=9.81 "gravity␣acceleration";
  Real h(start=1) "height␣of␣ball";
  Real v "velocity␣of␣ball";
  Boolean flying(start=true) "true,␣if␣ball␣is␣flying";
  Boolean impact;
  Real v_new;
  Integer foo;

equation
  impact = h <= 0.0;
  foo = if impact then 1 else 2;
  der(v) = if flying then -g else 0;
  der(h) = v;

  when {h <= 0.0 and v <= 0.0,impact} then
    v_new = if edge(impact) then -e*pre(v) else 0;
    flying = v_new > 0;
    reinit(v, v_new);
  end when;

end BouncingBall;
```

19.7.1 Files

An OpenModelica installation specific to the operating system is used to create a stand-alone executable program that performs the model calculations. A separate XML file containing model parameters and initial conditions is also generated as part of the build process. The RAVEN OpenModelica interface modifies input parameters by changing copies of this file. Both the executable

[1] `http://www.modelica.org`

and XML parameter file names must be provided to RAVEN. In the case of the BouncingBall model previously mentioned on the Windows operating system, the <Files>specification would look like:

```
<Files>
  <Input name='BouncingBall_init.xml'
     type=''>BouncingBall_init.xml</Input>
  <Input name='BouncingBall.exe' type=''>BouncingBall.exe</Input>
</Files>
```

19.7.2 Models

OpenModelica models may provide simulation output in a number of formats. The particular format used is specified during the model generation process. RAVEN works best with Comma-Separated Value (CSV) files, which is one of the possible output format options. Models are generated using the OpenModelica Shell (OMS) command-line interface, which is part of the OpenModelica installation. To generate an executable that provides CSV-formatted output, use OMSl commands as follows:

1. Change to the directory containing the .mo file to generate an executable for:

```
>> cd("C:/MinGW/msys/1.0/home/bobk/projects/raven/framework/
   ↪ CodeInterfaces/OpenModelica")
"C:/MinGW/msys/1.0/home/bobk/projects/raven/framework/
   ↪ CodeInterfaces/OpenModelica"
```

2. Load the model file into memory:

```
>> loadFile("BouncingBall.mo")
true
```

3. Create the model executable, specifying CSV output format:

```
>> buildModel(BouncingBall, outputFormat="csv")
{"C:/MinGW/msys/1.0/home/bobk/projects/raven/framework/
   ↪ CodeInterfaces/OpenModelica/BouncingBall","
   ↪ BouncingBall_init.xml"}
Warning: The initial conditions are not fully specified. Use
   ↪ +d=initialization for more information.
```

   At this point the model executable and XML initialization file should have been created in the same directory as the original model file.

The model executable is specified to RAVEN using the <Models>section of the input file as follows:

```
<Simulation>
    ...
  <Models>
    <Code name="BouncingBall" subType = "OpenModelica">
      <executable>BouncingBall.exe</executable>
    </Code>
  </Models>
    ...
</Simulation>
```

19.7.3 CSV Output

The CSV files produced by OpenModelica model executables require adjustment before it may be read by RAVEN. The first few lines of original CSV output from the BouncingBall example is shown below:

```
"time","h","v","der(h)","der(v)","v_new","foo","flying","impact",
0,1,0,0,-9.810000000000001,0,2,1,0,
  ...
```

RAVEN will not properly read this file as-generated for two reasons:

- The variable names in the first line are each enclosed in double-quotes.
- Each line has a trailing comma.

The OpenModelica inteface will automatically remove the double-quotes and trailing commas through its implementation of the finalizeCodeOutput function.

19.8 Dymola Interface

Modelica is ”a non-proprietary, object-oriented, equation-based language to conveniently model complex physical systems containing, e.g., mechanical, electrical, electronic, hydraulic, thermal, control, electric power or process-oriented subcomponents.”[2]. Modelica models (with a file extension of .mo) are built, translated (compiled), and simulated in Dymola (http://www.modelon.com/p-

[2] http://www.modelica.org

roducts/dymola/), which is a commercial modeling and simulation environment based on the Modelica modeling language. A standard Modelica example called BouncingBall, which simulates the trajectory of an object falling in one dimension from a height, is shown as an example:

```
model BouncingBall
  parameter Real e=0.7 "coefficient␣of␣restitution";
  parameter Real g=9.81 "gravity␣acceleration";
  parameter Real hstart = 10 "height␣of␣ball␣at␣time␣zero";
  parameter Real vstart = 0 "velocity␣of␣ball␣at␣time␣zero";
  Real h(start=hstart,fixed=true) "height␣of␣ball";
  Real v(start=vstart,fixed=true) "velocity␣of␣ball";
  Boolean flying(start=true) "true,␣if␣ball␣is␣flying";
  Boolean impact;
  Real v_new;
  Integer foo;

equation
  impact = h <= 0.0;
  foo = if impact then 1 else 2;
  der(v) = if flying then -g else 0;
  der(h) = v;

  when {h <= 0.0 and v <= 0.0,impact} then
    v_new = if edge(impact) then -e*pre(v) else 0;
    flying = v_new > 0;
    reinit(v, v_new);
  end when;

  annotation (uses(Modelica(version="3.2.1")),
    experiment(StopTime=10, Interval=0.1),
    __Dymola_experimentSetupOutput);

end BouncingBall;
```

19.8.1 Files

When a modelica model, e.g., BouncingBall model, is implemented in Dymola, the platform dependent C-code from a Modelica model and the corresponding executable code (i.e., by default dymosim.exe on the Windows operating system) are generated for simulation. After the executable is generated, it may be run multiple times (with Dymola license). A separate TEXT file (by default dsin.txt) containing model parameters and initial conditions are also generated as part of the

build process. The RAVEN Dymola interface modifies input parameters by changing copies of this file. Both the executable and TEXT parameter file (or simulation initialization file) names must be provided to RAVEN. The TEXT parameter file must be of type 'DymolaInitialisation'. In the case of the BouncingBall model previously mentioned on the Windows operating system, the <Files>specification would look like:

```
<Files>
  <Input name='dsin.txt'
     ↪ type='DymolaInitialisation'>dsin.txt</Input>
</Files>
```

The Dymola interface can only pass scalar values into the TEXT parameter file. If the user wants to pass vector information to Dymola, he can do so by providing an optional TEXT vector file to Dymola. This file must have the type 'DymolaVectors'. This additional file can then be read by the Dymola model. If vecor data is passed from RAVEN to the Dymola interface and the TEXT vector file is not specified, the interface will display an error and stop the Dymola execution. If the TEXT vector file is specified (and vector data is passed to the interface), the interface will write the datd into the specified file, but also display a warning, saying that the Dymola interface found vector data to be passed and if this data is supposed to go into the simulation initialisation file of type 'DymolaInitialisation' the array must be split into scalars. The <Files>specification for the vector data look as follows:

```
<Files>
  <Input name='timeSeriesData.txt'
     ↪ type='DymolaVectors'>timeSeriesData.txt</Input>
</Files>
```

19.8.2 Models

An executable (dymosim.exe) and a simulation initialization file (dsin.txt) can be generated after either translating or simulating the Modelica model (BouncingBall.mo) using the Dymola Graphical User Interface (GUI) or Dymola Application Programming Interface (API)-routines. To generate an executable and a simulation initialization file, use the Dymola API-routines (or Dymola GUI) to translate the model as follows:

1. Change to the directory containing the .mo file to generate an executable. In Dymola GUI, this corresponds to File/Change Directory in menus:

```
>> cd("C:/msys64/home/KIMJ/projects/raven/framework/
   ↪ CodeInterfaces/Dymola");
C:/msys64/home/KIMJ/projects/raven/framework/CodeInterfaces/
   ↪ Dymola
```

```
 = true
```

2. Reads the specified file and displays its window. In Dymola GUI, this corresponds to File/Open in the menus:

```
>> openModel("BouncingBall.mo")
 = true
```

3. Compile the model (with current settings), and create the model executable and the corresponding simulation initialization file. In Dymola GUI, this corresponds to Translate Model in the menus:

```
>> translateModel("BouncingBall");
 = true
```

At this point the model executable and the simulation initialization file should have been created in the same directory as the original model file. Additionally, they could be created by simulating the model. The following command corresponds to Simulate in the menus in Dymola GUI:

```
>> simulateModel("BouncingBall", stopTime=10,
   ↪ numberOfIntervals=0, outputInterval=0.1, method="dassl"
   ↪ , resultFile="BouncingBall");
 = true
```

The file extension (.mat) is automatically added to a output file (resultFile), e.g., BouncingBall.mat. If the generated executable code is triggered directly from a command prompt, the output file is always named as ”dsres.mat”.

The model executable is specified to RAVEN using the <Models>section of the input file as follows:

```
<Simulation>
    ...
  <Models>
    <Code name="BouncingBall" subType = "Dymola">
      <executable>dymosim.exe</executable>
    </Code>
  </Models>
    ...
</Simulation>
```

RAVEN works best with Comma-Separated Value (CSV) files. Therefore, the default .mat output type needs to be converted to .csv output. The Dymola interface will automatically convert the

.mat output to human-readable forms, i.e., .csv output, through its implementation of the finalizeCodeOutput function.
In order to speed up the reading and conversion of the .mat file, the user can specify the list of variables (in addition to the Time variable) that need to be imported and converted into a csv file minimizing the IO memory usage as much as possible. Within the **<Code>** the following XML node (in addition ot the **<executable>** one) can be inputted:

- **<outputVariablesToLoad>**, ***space separated list, optional parameter***, a space separated list of variables that need be exported from the .mat file (in addition to the Time variable).
  Default: all the variables in the .mat file.

For example:

```
<Simulation>
    ...
  <Models>
    <Code name="BouncingBall" subType = "Dymola">
      <executable>dymosim.exe</executable>
      <outputVariablesToLoad>var1 var2
         ↪ var3</outputVariablesToLoad>
    </Code>
  </Models>
    ...
</Simulation>
```

19.9 Mesh Generation Coupled Interfaces

Some software requires a provided mesh that requires a separate code run to generate. In these cases, we use sampled geometric variables to generate a new mesh for each perturbation of the original problem, then run the input with the remainder of the perturbed parameters and the perturbed mesh. RAVEN currently provides two interfaces for this type of calculation, listed below.

19.9.1 MooseBasedApp and Cubit Interface

Many MOOSE-based applications use Cubit (`https://cubit.sandia.gov`) to generate Exodus II files as geometry and meshing for calculations. To use the developed interface, Cubit's bin directory must be added to the user's PYTHONPATH. Input parameters for Cubit can be listed in a journal (`.jou`) file. Parameter values are typically hardcoded into the Cubit command syntax,

but variables may be predefined in a journal file through Aprepro syntax. This is an example of a journal file that generates a rectangle of given height and width, meshes it, defines its volume and sidesets, lists its element type, and writes it as an Exodus file:

```
#{x = 3}
#{y = 3}
#{out_name = "'out_mesh.e'"}
create surface rectangle width {x} height {y} zplane
mesh surface 1
set duplicate block elements off
block 1 surface 1
Sideset 1 curve 3
Sideset 2 curve 4
Sideset 3 curve 1
Sideset 4 curve 2
Block all element type QUAD4
export genesis {out_name} overwrite
```

The first three lines are the Aprepro variable definitions that RAVEN requires to insert sampled variables. All variables that RAVEN samples need to be defined as Aprepro variables in the journal file. One essential caveat to running this interface is that an Aprepro variable MUST be defined with the name "out_name". In order to run this script without RAVEN inserting the correct syntax for the output file name and properly generate the Exodus file for a mesh, the output file name is REQUIRED to be in both single and double quotation marks with the file extension appended to the end of the file base name (e.g. '"output_file.e"').

19.9.1.1 Files

<Files> works the same as in other interfaces with name and type attributes for each node entry. The **name** attribute is a user-chosen internal name for the file contained in the node, and **type** identifies which base-level interface the file is used within. **<type>** should only be specified for inputs that RAVEN will perturb. For Moose input files, **<type>** should be **'MooseInput'** and for Cubit journal files, the **<type>** should be **'CubitInput'**. The node should contain the path to the file from the working directory. The following is an example of a typical **<Files>** block.

```
<Files>
  <Input name='moose_test'
     ↪ type='MooseInput'>simple_diffusion.i</Input>
  <Input name='mesh_in'
     ↪ type='CubitInput'>rectangle.jou</Input>
  <Input name='other_file' type=''
     ↪ >some_file_moose_input_needs.ext</Input>
```

```
</Files>
```

19.9.1.2 Models

A user provides paths to executables and aliases for sampled variables within the **<Models>** block. The **<Code>** block will contain attributes name and subType. Name identifies that particular **<Code>** model within RAVEN, and subType specifies which code interface the model will use. The **<executable>** block should contain the absolute or relative (with respect to the current working directory) path to the MooseBasedApp that RAVEN will use to run generated input files. The absolute or relative path to the Cubit executable is specified within **<preexec>**. If the **<preexec>** block is not needed, the MooseBasedApp interface is probably preferable to the Cubit-Moose interface.

Aliases are defined by specifying the variable attribute in an **<alias>** node with the internal RAVEN variable name chosen with the node containing the model variable name. The Cubit-Moose interface uses the same syntax as the MooseBasedApp to refer to model variables, with pipes separating terms starting with the highest YAML block going down to the individual parameter that RAVEN will change. To specify variables that are going to be used in the Cubit journal file, the syntax is "Cubit—aprepro_var". The Cubit-Moose interface will look for the Cubit tag in all variables passed to it and upon finding it, send it to the Cubit interface. If the model variable does not begin with **'Cubit'**, the variable MUST be specified in the MooseBasedApp input file. While the model variable names are not required to have aliases defined (the **<alias>** blocks are optional), it is highly suggested to do so not only to ensure brevity throughout the RAVEN input, but to easily identify where variables are being sent in the interface.

An example **<Models>** block follows.

```
<Models>
  <Code name="moose-modules" subType="CubitMoose">
    <executable>%FRAMEWORK_DIR%/../../moose/modules/combined/...
      modules-%METHOD%</executable>
    <preexec>/hpc-common/apps/local/cubit/13.2/bin/cubit</preexec>
    <alias variable="length">Cubit@y</alias>
    <alias variable="bot_BC">BCs|bottom|value</alias>
  </Code>
</Models>
```

19.9.1.3 Distributions

The **<Distributions>** block defines all distributions used to sample variables in the current RAVEN run.

For all the possible distributions and their possible inputs please refer to the Distributions chapter (see 9). It is good practice to name the distribution something similar to what kind of variable is going to be sampled, since there might be many variables with the same kind of distributions but different input parameters.

19.9.1.4 Samplers

The **<Samplers>** block defines the variables to be sampled.

After defining a sampling scheme, the variables to be sampled and their distributions are identified in the **<variable>** blocks. The name attribute in the **<variable>** block must either be the full MooseBasedApp model variable name or the alias name specifed in **<Models>**. If the sampled variable is a geometric property that will be used to generate a mesh with Cubit, remember the syntax for variables being passed to journal files (Cubit—aprepro_var).

For listings of available samplers refer to the Samplers chapter (see 10).

See the following for an example of a grid based sampler for length and the bottom boundary condition (both of which have aliases defined in **<Models>**).

```
<Samplers>
  <Grid name="Grid_sampling">
    <variable name="length" >
      <distribution>length_dist</distribution>
      <grid type="value" construction="custom">1.0 2.0</grid>
    </variable>
    <variable name="bot_BC">
      <distribution>bot_BC_dist</distribution>
      <grid type="value" construction="custom">3.0 6.0</grid>
    </variable>
  </Grid>
</Samplers>
```

19.9.1.5 Steps,OutStreams,DataObjects

This interface's **<Steps>**, **<OutStreams>**, and **<DataObjects>** blocks do not deviate significantly from other interfaces' respective nodes. Please refer to previous entries for these blocks if needed.

19.9.1.6 File Cleanup

The Cubit-Moose interface automatically removes files that are commonly unwanted after the RAVEN run reaches completion. Cubit has been described as "talkative" due to additional journal files with execution information being generated by the program after every completed journal file run. The quantity of these files can quickly become unwieldly if the working directory is not kept clean; thus these files are removed. In addition, some users may wish to remove Exodus files after the RAVEN run is complete as the typical size of each file is quite large and it is assumed that any output quantities of interest will be collected by appropriate postprocessors and the OutStreams. Exodus files are not automatically removed, but by using the **<deleteOutExtension>** node in **<RunInfo>**, one may specify the Exodus extension to save a fair amount of storage space after RAVEN completes a sequence. For example:

```
<RunInfo>
  ...
  <deleteOutExtension>e</deleteOutExtension>
  ...
</RunInfo>
```

19.9.2 MooseBasedApp and Bison Mesh Script Interface

For BISON users, a Python mesh generation script is included in the %BISON_DIR%/tools/UO2/ directory. This script generates 3D or 2D (RZ) meshes for nuclear fuel rods using Cubit with templated commands. The BISON Mesh Script (BMS) is capable of generating rods with discrete fuel pellets of various size in assorted configurations. To use this interface, Cubit's bin directory must be added to the user's PYTHONPATH.

19.9.2.1 Files

Similar to the Cubit-Moose interface, the BisonAndMesh interface requires users to specify all files required to run their input so that these file may be copied into the respective sequence's working directory. The user will give each file an internal RAVEN designation with the name attribute, and

the MooseBasedApp and BISON Mesh Script inputs must be assigned their respective types in another attribute of the **<Input>** node. An example follows.

```
<Files>
  <Input name='bison_test'
     ↪ type='MooseInput'>simple_bison_test.i</Input>
  <Input name='mesh_in'
     ↪ type='BisonMeshInput'>coarse_input.py</Input>
  <Input name='other_file'
     ↪ type=''>some_file_moose_input_needs.ext</Input>
</Files>
```

19.9.2.2 Models

A user provides paths to executables and aliases for sampled variables within the **<Models>** block. The **<Code>** block will contain attributes **name** and **subType**. **name** identifies that particular **<Code>** model within RAVEN, and **subType** specifies which code interface the model will use. The **<executable>** block should contain the absolute or relative (with respect to the current working directory) path to the MooseBasedApp that RAVEN will use to run generated input files. The absolute or relative path to the mesh script python file is specified within **<preexec>**. If the **<preexec>** block is not needed, use the MooseBasedApp interface.

Aliases are defined by specifying the variable attribute in an **<alias>** node with the internal RAVEN variable name chosen with the node containing the model variable name. The BisonAndMesh interface uses the same syntax as the MooseBasedApp to refer to model variables, with pipes separating terms starting with the highest YAML block going down to the individual parameter that RAVEN will change. To specify variables that are going to be used in the BISON Mesh Script python input, the syntax is "Cubit—dict_name—var_name". The interface will look for the Cubit tag in all variables passed to it and upon finding the tag, send it to the BISON Mesh Script interface. If the model variable does not begin with Cubit, the variable MUST be specified in the MooseBasedApp input file. While the model variable names are not required to have aliases defined (the **<alias>** blocks are optional), it is highly suggested to do so not only to ensure brevity throughout the RAVEN input, but to easily identify where variables are being sent in the interface.

An example **<Models>** block follows.

```
<Models>
  <Code name="Bison-opt" subType="BisonAndMesh">
    <executable>%FRAMEWORK_DIR%/../../bison/bison-%METHOD%</executable>
    <preexec>%FRAMEWORK_DIR%/../../bison/tools/UO2/mesh_script.py</preexec>
    <alias variable="pellet_radius"
       ↪ >Cubit@Pellet1|outer_radius</alias>
```

```
    <alias
       ↪ variable="clad_thickness">Cubit@clad|clad_thickness</alias>
    <alias variable="fuel_k"
       ↪ >Materials|fuel_thermal|thermal_conductivity</alias>
    <alias variable="clad_k"
       ↪ >Materials|clad_thermal|thermal_conductivity</alias>
  </Code>
</Models>
```

19.9.2.3 Distributions

The **<Distributions>** block defines all distributions used to sample variables in the current RAVEN run.

For all the possible distributions and their possible inputs please refer to the Distributions chapter (see 9). It is good practice to name the distribution something similar to what kind of variable is going to be sampled, since there might be many variables with the same kind of distributions but different input parameters.

19.9.2.4 Samplers

The **<Samplers>** block defines the variables to be sampled.

After defining a sampling scheme, the variables to be sampled and their distributions are identified in the **<variable>** blocks. The name attribute in the **<variable>** block must either be the full MooseBasedApp model variable name or the alias name specified in **<Models>**. If the sampled variable is a geometric property that will be used to generate a mesh with Cubit, remember the syntax for variables being passed to journal files (Cubit—aprepro_var).

For listings of available samplers refer to the Samplers chapter (see 10).

See the following for an example of a grid based sampler for length and the bottom boundary condition (both of which have aliases defined in **<Models>**).

```
<Samplers>
  <Grid name="Grid_sampling">
    <variable name="length" >
      <distribution>length_dist</distribution>
      <grid type="value" construction="custom">1.0 2.0</grid>
    </variable>
    <variable name="bot_BC">
```

```
      <distribution>bot_BC_dist</distribution>
      <grid type="value" construction="custom">3.0 6.0</grid>
    </variable>
  </Grid>
</Samplers>
```

19.9.2.5 Steps,OutStreams,DataObjects

This interface's **<Steps>**, **<OutStreams>**, and **<DataObjects>** blocks do not deviate significantly from other interfaces' respective nodes. Please refer to previous entries for these blocks if needed.

19.9.2.6 File Cleanup

The BisonAndMesh interface automatically removes files that are commonly unwanted after the RAVEN run reaches completion. Cubit has been described as "talkative" due to additional journal files with execution information being generated by the program after every completed journal file run. The BISON Mesh Script creates a journal file to run with cubit after reading input parameters; so Cubit will generate its "redundant" journal files, and .pyc files will litter the working directory as artifacts of the python mesh script reading from the .py input files. The quantity of these files can quickly become unwieldly if the working directory is not kept clean, thus these files are removed. Some users may wish to remove Exodus files after the RAVEN run is complete as the typical size of each file is quite large and it is assumed that any output quantities of interest will be collected by appropriate postprocessors and the OutStreams. Exodus files are not automatically removed, but by using the **<deleteOutExtension>** node in **<RunInfo>**, one may specify the Exodus extension (*.e) to save a fair amount of storage space after RAVEN completes a sequence. For example:

```
<RunInfo>
  ...
  <deleteOutExtension>e</deleteOutExtension>
  ...
</RunInfo>
```

19.10 Rattlesnake Interfaces

This section covers the input specification for running Rattlesnake through RAVEN. It is important to notice that this short explanation assumes that the reader already knows how to use Rattlesnake.

The interface can be used to perturb the Rattlesnake MOOSE-based input file as well as the Yak cross section libraries XML input files (e.g. multigroup cross section libraries) and Instant format cross section libraries.

19.10.1 Files

<Files> works the same as in other interfaces with name and type attributes for each node entry. The **name** attribute is a user-chosen internal name for the file contained in the node, and **type** identifies which base-level interface the file is used within. **type** should only be specified for inputs that RAVEN will perturb. Take Rattlesnake input files for example, **type** should be **'RattlesnakeInput'**.

19.10.1.1 Perturb Yak Multigroup Cross Section Libraries

If the user would like to perturb the Yak multigroup cross section libraries, the user need to use the **'YakXSInput'** for the **type** of the libaries. In addition, the **type** of the alias files that are used to perturb the Yak multigroup cross section libraries should be **'YakXSAliasInput'**. The following is an example of a typical **<Files>** block.

```
<Files>
  <Input name='rattlesnakeInput'
     ↪ type='RattlesnakeInput'>simple_diffusion.i</Input>
  <Input name='crossSection'    type='YakXSInput'>xs.xml</Input>
  <Input name='alias' type='YakXSAliasInput'>alias.xml</Input>
</Files>
```

The alias files are employed to define the variables that will be used to perturb Yak multigroup cross section libraries. The following is an example of a typical alias file:

```
<Multigroup_Cross_Section_Libraries Name="twigl" NGroup="2"
   ↪ Type="rel">
    <Multigroup_Cross_Section_Library ID="1">
        <Fission gridIndex="1" mat="pseudo-seed1"
           ↪ gIndex="1">f11</Fission>
        <Capture gridIndex="1" mat="pseudo-seed1"
           ↪ gIndex="1">c11</Capture>
        <TotalScattering gridIndex="1" mat="pseudo-seed1"
           ↪ gIndex="1">t11</TotalScattering>
        <Nu gridIndex="1" mat="pseudo-seed1" gIndex="1">n11</Nu>
        <Fission gridIndex="1" mat="pseudo-seed2"
           ↪ gIndex="2">f22</Fission>
```

```
        <Capture gridIndex="1" mat="pseudo-seed2"
           ↪ gIndex="2">c22</Capture>
        <TotalScattering gridIndex="1" mat="pseudo-seed2"
           ↪ gIndex="2">t22</TotalScattering>
        <Nu gridIndex="1" mat="pseudo-seed2" gIndex="2">n22</Nu>
        <Fission gridIndex="1" mat="pseudo-seed1-dup"
           ↪ gIndex="1">f11</Fission>
        <Capture gridIndex="1" mat="pseudo-seed1-dup"
           ↪ gIndex="1">c11</Capture>
        <TotalScattering gridIndex="1" mat="pseudo-seed1-dup"
           ↪ gIndex="1">t11</TotalScattering>
        <Nu gridIndex="1" mat="pseudo-seed1-dup"
           ↪ gIndex="1">n11</Nu>
        <Transport gridIndex="1" mat="pseudo-seed1-dup"
           ↪ gIndex="1">d11</Transport>
    </Multigroup_Cross_Section_Library>
</Multigroup_Cross_Section_Libraries>
```

In the above alias file, the **Name** of **<Multigroup_Cross_Section_Libraries>** are used to indicate which Yak multigroup cross section library input file will be perturbed. The **NGroup**, **ID**, and **<Multigroup_Cross_Section_Library>** should be consistent with the Yak multigroup cross section library input files. The **<Fission>**, **<Capture>**, **<TotalScattering>**, **<Nu>**, **gridIndex**, **mat**, and **gIndex** are used to find the corresponding cross sections in the Yak multigroup cross section library input files. For example:

```
<Fission gridIndex="1" mat="pseudo-seed1"
   ↪ gIndex="1">f11</Fission>
```

This node defines an alias with name **'f11'** used to represent the fission cross section at energy group **'1'** for material with name 'pseudo-seed1' at grid index **'1'** in the Yak multigroup cross section library input files.

Note: The attribute **Type="rel"** indicates that the cross sections will be perturbed relatively (i.e. perturbed by percents). In this case, the user also needs to specify a relative covariance matrix for **<covaraince type="rel">** in **<MultivariateNormal>** distribution, and the values for **<mu>** should be 'ones'. In the other case, if the user choose **Type="abs"**, the cross sections will be perturbed absolutely (i.e. perturbed by values), and the user needs to provide an absolute covariance matrix and specify 'zeros' for **<mu>** in **<MultivariateNormal>** distribution.

Note: Currently, only the following cross sections can be perturbed by the user: Fission, Capture, Nu, TotalScattering, and Transport.

19.10.1.2 Perturb Instant format Cross Section Libraries

If the user would like to perturb the Instant cross section libraries, the user need to use the **'InstantXSInput'** for the **type** of the libaries. In addition, the **type** of the alias files that are used to perturb the Instant format cross section libraries should be **'InstantXSAliasInput'**. The following is an example of a typical **<Files>** block.

```
<Files>
  <Input name='rattlesnakeInput'
     ↪ type='RattlesnakeInput'>iaea2d_ls_sn.i</Input>
  <Input name='crossSection'
     ↪ type='InstantXSInput'>iaea2d_materials.xml</Input>
  <Input name='alias'
     ↪ type='InstantXSAliasInput'>alias.xml</Input>
</Files>
```

The alias files are employed to define the variables that will be used to perturb Instant format cross section libraries. The following is an example of a typical alias file:

```
<Materials>
  <Macros NG="2" Type="rel">
    <material ID="1">
      <FissionXS gIndex="1">f11</FissionXS>
      <CaptureXS gIndex="1">c11</CaptureXS>
      <TotalScatteringXS gIndex="1">t11<TotalScatteringXS>
      <Nu gIndex="1">n11</Nu>
      <DiffusionCoefficient gIndex="1">d11</DiffusionCoefficient>
    </material>
  </Macros>
</Materials>
```

In the above alias file, the **NG** and **ID** should be consistent with the Instant format cross section library input files. The **<FissionXS>**, **<CaptureXS>**, **<TotalScatteringXS>**, **<Nu>**, **gIndex**, are used to find the corresponding cross sections in the Instant format cross section library input files. For example, the variable **'f11'** used to represent the fission cross section at energy group **'1'** for material with **'ID'** equal **'1'** in the given cross section library.

Note: The attribute **Type="rel"** indicates that the cross sections will be perturbed relatively (i.e. perturbed by percents). In this case, the user also needs to specify a relative covariance matrix for **<covaraince type="rel">** in **<MultivariateNormal>** distribution, and the values for **<mu>** should be 'ones'. In the other case, if the user choose **Type="abs"**, the cross sections will be perturbed absolutely (i.e. perturbed by values), and the user needs to provide an absolute covariance matrix and specify 'zeros' for **<mu>** in **<MultivariateNormal>** distribution.

Note: Currently, only the following cross sections can be perturbed by the user: FissionXS, CaptureXS, Nu, TotalScatteringXS, and DiffusionCoefficient.

19.10.2 Models

A user provides paths to executables and aliases for sampled variables within the **<Models>** block. The **<Code>** block will contain attributes **<name>** and **<subType>**. The **<name>** identifies that particular **<Code>** model within RAVEN, and **<subType>** specifies which code interface the model will use. The **<executable>** block should contain the absolute or relative (with respect to the current working directory) path to Rattlesnake that RAVEN will use to run generated input files.

An example **<Models>** block follows.

```
<Models>
  <Code name="Rattlesnake" subType="Rattlesnake">
    <executable>%FRAMEWORK_DIR%/../../rattlesnake/
     rattlesnake-%METHOD%</executable>
  </Code>
</Models>
```

19.10.3 Distributions

The **<Distributions>** block defines all distributions used to sample variables in the current RAVEN run.

For all the possible distributions and their possible inputs please refer to the Distributions chapter (see 9). It is good practice to name the distribution something similar to what kind of variable is going to be sampled, since there might be many variables with the same kind of distributions but different input parameters.

19.10.3.1 Samplers

The **<Samplers>** block defines the variables to be sampled. After defining a sampling scheme, the variables to be sampled and their distributions are identified in the **<variable>** blocks. The name attribute in the **<variable>** block must either be the full MooseBasedApp (Rattlesnake) model variable name, the alias name specifed in **<Models>**, or the variable name specified in the provided alias files.

For listings of available samplers, please refer to the Samplers chapter (see 10). See the following for an example of a grid based sampler for the first energy group fission and capture cross sections (both of which have defined in alias files provided in **<Files>**).

```
<Samplers>
  <Grid name="Grid_sampling">
    <variable name="fission_group_1" >
      <distribution>fission_dist</distribution>
      <grid type="value" construction="custom">1.0 2.0</grid>
    </variable>
    <variable name="capture_group_1">
      <distribution>capture_dist</distribution>
      <grid type="value" construction="custom">3.0 6.0</grid>
    </variable>
  </Grid>
</Samplers>
```

19.10.4 Steps

For a Rattlesnake interface, the **<MultiRun>** step type will most likely be used. First, the step needs to be named: this name will be one of the names used in the **<Sequence>** block. In our example, **'Grid_Rattlesnake'**.

```
<MultiRun name='Grid_Rattlesnake' verbosity='debug'>
    <Input   class='Files' type=''>RattlesnakeInput.i</Input>
    <Input   class='Files' type=''>xs.xml</Input>
    <Input   class='Files' type=''>alias.xml</Input>
    <Model   class='Models' type='Code'>Rattlesnake</Model>
    <Sampler class='Samplers'
      ↪ type='Grid'>Grid_Samplering</Sampler>
    <Output  class='DataObjects' type='PointSet'>solns</Ouput>
```

With this step, we need to import all the files needed for the simulation:

- Rattlesnake MOOSE-based input file;
- Yak multigroup cross section libraries input files (XML);
- Yak alias files used to define the perturbed variables (XML).

We then need to define **<Model>**, **<Sampler>** and **<Output>**. The **<Output>** can be **<DataObjects>** or **<OutStreams>**.

19.11 MAAP5 Interface

This section presents the main aspects of the interface coupling RAVEN with MAAP5, the consequent RAVEN input adjustments and the modifications of the MAAP5 files required to run the two coupled codes. The interface works both for forward sampling and the DET, however there are some differences depending on the selected sampling strategy.

19.11.1 RAVEN Input file

19.11.1.1 Files

MAAP5 requires more than one file to run a simulation. This means that, since the **<Files>** section has to contain all the files required by the external model (MAAP5) to be run, all these files need to be included within this node. This involves not only the input file (.inp) but also the include file, the parameter file, all the files defining the different "PLOTFILs", if any, and the other files which could result useful for the MAAP5 simulation run.

Example:

```
<Files>
  <Input name="test.inp" type="">test.inp</Input>
  <Input name="include" type="">include</Input>
  <Input name="plot.txt" type="">plot.txt</Input>
  <Input name="plant.par" type="">plant.par</Input>
</Files>
```

The files mentioned in this section need, then, to be put into the working directory specified by the **<workingDir>** node into the **<RunInfo>** block.

19.11.1.2 Models

The **<Models>** block contains the name of the executable file of MAAP5 (with the path, if necessary), and the name of the interface (e.g. MAAP5_GenericV7). The block has also some required nodes:

- **<boolMaapOutputVariables>**: containing the number of the MAAP5 IEVNT corresponding to the boolean events of interest;
- **<contMaapOutputVariables>**: containing the list of all the continuous variables we are interested at, and that we want to monitor;

- **<stopSimulation>**: this node is required only in case of DET sampling strategy. The user needs to specify if the MAAP5 simulation run stops due to the reached END TIME, specifying "mission_time", or due to the occurrence of a specific event by inserting the number of the corresponding MAAP5 IEVNT (e.g IEVNT(691) for core uncovery)
- **<includeForTimer>**: also this node is required only in case of DET sampling strategy and it contains the name of the MAAP5 include file where the TIMERS for the different variables are defined (see paragraph "MAAP5 include file below" for more information about timers).

A **<Models>** block is shown as an example below:

```
<Models>
  <Code name="MyMAAP" subType="MAAP5\_GenericV7">
    <executable>MAAP5.exe</executable>
    <clargs type='input' extension='.inp'/>
    <boolMaapOutputVariables>691</boolMaapOutputVariables>
    <contMaapOutputVariables>PPS,PSGGEN(1),ZWDC2SG(1)
    </contMaapOutputVariables>
    <stopSimulation>mission_time</stopSimulation>
    <includeForTimer>include</includeForTimer>
  </Code>
</Models>
```

19.11.1.3 Other blocks

All the other blocks (e.g. **<Distributions>**, **<Samplers>**, **<Steps>**, **<Databases>**, **<OutStream>**, etc.) do not require any particular arrangements than already provided by a RAVEN input. User can, therefore, refer to the corresponding sections of the User's Manual. This is valid for both forward sampling and DET.

19.11.2 MAAP5 Input files

The coupling of RAVEN and MAAP5 requires modifications to some MAAP5 files in order to work. This is particularly true when a DET analysis is performed. The MAAP5 input files that need to be modified are:

- MAAP5 include file
- MAAP5 input file (.inp)
- PLOTFIL blocks

19.11.2.1 MAAP5 include file

Usually MAAP5 simulation provides the presence of some include files, for example, containing the user-defined variables, timers, definition of the plotfil, etc. The adjustments explained in this section are required only in case of a DET analysis. The user needs to modify the include file containing the set of the timers used into the run, by adding the definition of the different timers, one for each variable that causes a branching. The include file to be modified should correspond to that one defined in the **<includeForTimer>** block of the RAVEN xml input.

User is supposed to check that the numbers used for the different timers definition are not already used in any of the other MAAP5 files. These timers should be preceeded by a line reporting ”C Branching + name of the variable sampled by RAVEN causing the branching”.

For example, we assume that DIESEL is the name of the variable corresponding to the failure time of the Diesel generators (user defined). User has to firstly ensure that, for example, ”TIMER 100” is not already used into the model, then the following lines need to be added into the selected include file for the set of the timer corresponding to the Diesel generators failure:

```
C Branching DIESEL
WHEN (TIM>DIESEL)
   SET TIMER 100
END
```

It is worth mentioning that at this step a TIMER should be defined also for the event IEVNT specified into the **<stopSimulation>**, if this is the stop condition for the MAAP5 run:

```
WHEN IEVNT(691) == 1.0
  SET TIMER 10
END
```

The interface will check that one timer is defined for each variable of the DET. If not, an error arises suggesting to user the name of the variable having no timer defined.

19.11.2.2 MAAP5 input file

In the ”parameter change” section of the MAAP5 input file, the user should declare the name of the variables sampled by RAVEN according to the following statement:

```
 variable = $ RAVEN-variable:default$
```

where the dafault value is optional.

For example:

```
DIESEL = $RAVEN-DIESEL:-1$
```

This is valid for both forward and DET sampled variables. In particular, in case of DET analysis, the variables causing the occurrence of the branch should be assigned within a block identified by the comment ”C DET Sampled variables”:

```
C DET Sampled Variables
DIESEL = $RAVEN-DIESEL:-1$
C End DET Sampled Variables
```

If the sampled variables are user-defined, then the user shall ensure that they are initialized (to the default value) and set within the user-defined variables section of one of the include file. As usual, a distribution and a sampling strategy should then correspond to each of these variables into the RAVEN xml input file.

Only for the DET analysis, then, the occurence of a branch will be identified by a comment before. This comment is ”C BRANCHING + name of the variable determining the branch” and acts as a sort of branching marker. Looking for these markers, indeed, the interface (in case of DET sampler) verifies that at least one branching exists, and furthermore, that one branching is defined for each of the variables contained into ”DET sampled variables”.

Within the block, the occurrence of the branching leads the value of a variable (user-defined) called ”TIM+number of the corresponding timer set into the include file” to switch to 1.0. The code, in fact, detects if a branch has occurred by monitoring the value of these kind of variables. SInce these variables are user-defined, they need to be initialized to a value (different from 1.0), into the ”user-defined variables” section of one of the include files.

Therefore following the previous example, if we want that, when the diesels failure occurs it leads to the event ”Loss of AC Power” (IEVNT(205) of MAAP5), we will have:

```
C Branching TIMELOCA
WHEN TIM > DIESEL
 TIM100=1.0
 IEVNT(205)=1.0
END
```

It is worth noticing that no comments should be contained within the line of assignment (i.e. IEVNT(205)=1.0 //LOSS OF AC POWER is not allowed).

Finally, only in case of DET analysis, a stop simulation condition (provided by the comment ”C Stop Simulation condition”) needs to be put into the input. The original input should have all the timers (linked with the branching) separated by an OR condition, even including that one of the event that stops the simulation (e.g. IEVNT(691)), if any.

```
C Stop Simulation condition
IF (TIMER 10 > 0) OR (TIMER 100 > 0) OR ... (TIMER N > 0)
 TILAST=TIM
END
```

This allows the simulation run to stop when a branch condition occurs, creating the restart file that will be used by the two following branches.

For each branch, then, the interface will automatically update the name of the RESTART FILE to be used and of the RESTART TIME that will be equal to the difference between the END TIME of the ”parent” simulation and the PRINT INTERVAL (which specifies the interval at which the restart output is written).

19.11.2.3 MAAP5 PLOTFIL blocks

This section refers to the ”PLOTFIL blocks” used to modify the plot file (.csv) defined into the parameter file. These blocks need to be modified in order to include some variables. It is important, indeed, that the MAAP5 csv PLOTFIL files contain the evolution of:

- RAVEN sampled variables (e.g. DIESEL) (both for Forward and DET sampling)
- the variables whose value is modified by the occurrence of one of the branches, either continuous or boolean (e.g. IEVNT(225))
- the variables of interest defined within **`<boolMaapOutputVariables>`** and **`<contMaapOutputVariables>`** blocks (both for Forward and DET sampling)

If one of these variables is not contained into one of csv files, RAVEN will give an error.

19.12 MAMMOTH Interface

This section covers the input specification for running MAMMOTH through RAVEN. It is important to notice that this short explanation assumes that the reader already knows how to use MAMMOTH. The interface can be used to perturb Bison, Rattlesnake, RELAP-7, and general MOOSE input files that utilize MOOSE’s standard YAML input structure as well as Yak multigroup cross section library XML input files.

19.12.1 Files

<Files> works the same as in other interfaces with name and type attributes for each node entry. The **name** attribute is a user-chosen internal name for the file contained in the node, and **type** identifies which base-level interface the file is used within. **type** should be specified for all inputs used in RAVEN's MultiRun for MAMMOTH (including files not perturbed by RAVEN). The MAMMOTH input file's **type** should have **'MAMMOTHInput'** prepended to the driver app's input specification (e.g. **'MAMMOTHInput|appNameInput'**). Any other app's input file needs a **type** with the app's name prepended to **'Input'** (e.g. **'BisonInput'**, **'Relap7Input'**, etc.). In addition, the **type** for any mesh input is the app in which that mesh is utilized prepended to **'|Mesh'**; so a Bison mesh would have a **type** of **'Bison|Mesh'** and similarly a mesh for Rattlesnake would have **'Rattlesnake|Mesh'** as its **type**. In cases where a file needs to be copied to each perturbed run directory (to be used as function input, control logic, etc.), one can use the **type** **'AncillaryInput'** to make it clear in the RAVEN input file that this is file is required for the simulation to run but contains no perturbed parameters. For Yak multigroup cross section libraries, the **type** should be **'YakXSInput'**, and for the Yak alias files that are used to perturb the Yak multigroup cross section libraries, the **type** should be **'YakXSAliasInput'**.

The node should contain the path to the file from the working directory. The following is an example of a typical **<Files>** block.

```
<Files>
  <Input name='mammothInput'
     ↪ type='MAMMOTHInput|RattlesnakeInput'>test_mammoth.i</Input>
  <Input name='crossSection'    type='YakXSInput'>xs.xml</Input>
  <Input name='alias' type='YakXSAliasInput'>alias.xml</Input>
  <Input name='bisonInput'
     ↪ type='BisonInput'>test_bison.xml</Input>
  <Input name='bisonMesh'
     ↪ type='Bison|Mesh'>bisonMesh.e</Input>
  <Input name='fuelCTEfunct'
     ↪ type='AncillaryInput'>uo2_CTE.csv</Input>
  <Input name='rattlesnakeMesh'
     ↪ type='Rattlesnake|Mesh'>rattlesnakeMesh.e</Input>
</Files>
```

The alias files are employed to define the variables that will be used to perturb Yak multigroup cross section libraries. Please see the section 19.10 for the example.

19.12.2 Models

A user provides paths to executables and aliases for sampled variables within the **<Models>** block. The **<Code>** block will contain **name** and **subType**. The attribute **name** identifies that particular **<Code>** model within RAVEN, and **subType** specifies which code interface the model will use. The **<executable>** block should contain the absolute or relative (with respect to the current working directory) path to MAMMOTH that RAVEN will use to run generated input files.

An example **<Models>** block follows.

```
<Models>
  <Code name="Mammoth" subType="MAMMOTH">
    <executable>\%FRAMEWORK_DIR\%/../../mammoth/
     mammoth-%METHOD%</executable>
  </Code>
</Models>
```

19.12.3 Distributions

The **<Distributions>** block defines all distributions used to sample variables in the current RAVEN run.

For all the possible distributions and their possible inputs please refer to the Distributions chapter (see 9). It is good practice to name the distribution something similar to what kind of variable is going to be sampled, since there might be many variables with the same kind of distributions but different input parameters.

19.12.3.1 Samplers

The **<Samplers>** block defines the variables to be sampled. After defining a sampling scheme, the variables to be sampled and their distributions are identified in the **<variable>** blocks. The **name** attribute in the **<variable>** block must either be the app's name prepended to the full MooseBasedApp model variable name, the alias name specifed in **<Models>**, or the variable name specified in the provided alias files.

For listings of available samplers, please refer to the Samplers chapter (see 10). See the following for an example of a grid based sampler used to generate the samples for the first energy group fission and capture cross sections (both of which have defined in alias files provided in **<Files>**), the initial condition temperature defined in Rattlesnake input file and the poissons ratio, clad thickness, and gap width defined in Bison input files with clad and gap parameters calculated using an external function with sampled clad inner and outer diameters as inputs.

```
<Samplers>
  <Grid name="Grid_sampling">
    <variable name="Rattlesnake@fission_group_1" >
      <distribution>fission_dist</distribution>
      <grid type="value" construction="custom">1.0 2.0</grid>
    </variable>
    <variable name="Rattlesnake@capture_group_1">
      <distribution>capture_dist</distribution>
      <grid type="value" construction="custom">3.0 6.0</grid>
    </variable>
    <variable
       ↪ name="Rattlesnake@AuxVariables|Temp|initial_condition">
      <distribution>uniform</distribution>
      <grid type="value" construction="custom">3.0 6.0</grid>
    </variable>
    <variable
       ↪ name="Bison@Materials|fuel_solid_mechanics_elastic|poissons_ratio"
      <distribution>normal</distribution>
      <grid type="value" construction="custom">3.0 6.0</grid>
    </variable>
    <variable name='clad_outer_diam'>
      <distribution>clad_outer_diam_dist</distribution>
      <grid construction='equal' steps='144' type='CDF'>0.02275
         ↪ 0.97725</grid>
    </variable>
    <variable name='clad_inner_diam'>
      <distribution>clad_inner_diam_dist</distribution>
      <grid construction='equal' steps='144' type='CDF'>0.02275
         ↪ 0.97725</grid>
    </variable>
    <variable name='Bison@Mesh|clad_thickness'>
      <function>clad_thickness_calc</function>
    </variable>
    <variable name='Bison@Mesh|clad_gap_width'>
      <function>clad_gap_width_calc</function>
    </variable>
  </Grid>
</Samplers>
```

In order to make the input variables of one application distinct from input variables of another, an app's name followed by the '@' symbol is prepended to the variable name (e.g. **'appName@varName'**).

Each variable to be used in an app's input file and sampled in the MAMMOTH interface is required to have a destination app specified. All variables utilizing Rattlesnake's executable (whether they are in the Rattlesnake input file or not) are listed as Rattlesnake variables as that application's interface will sort input file and cross section variables itself. Notice that the clad inner and outer diameter sampled parameters have no app name specified. These parameters are utilized to sample values used as inputs for the clad thickness and gap width variables in BISON, so by not specifying a destination app, these are passed through the interface having only been used in an external function to calculate parameters usable in an app's input.

19.12.4 Steps

For a MAMMOTH interface run, the **<MultiRun>** step type will most likely be used. First, the step needs to be named: this name will be one of the names used in the **<Sequence>** block. In our example, **'Grid_Mammoth'**.

```
<MultiRun name='Grid_Mammoth' verbosity='debug'>
    <Input   class='Files' type=''>mammothInput</Input>
    <Input   class='Files' type=''>crossSection</Input>
    <Input   class='Files' type=''>alias</Input>
    <Input   class='Files' type=''>bisonInput</Input>
    <Input   class='Files' type=''>bisonMesh</Input>
    <Input   class='Files' type=''>fuelCTEfunct</Input>
    <Input   class='Files' type=''>rattlesnakeMesh</Input>
    <Model   class='Models' type='Code'>Mammoth</Model>
    <Sampler class='Samplers'
       ↪ type='Grid'>Grid_Samplering</Sampler>
    <Output  class='DataObjects' type='PointSet'>solns</Output>
</MultiRun>
```

With this step, we need to import all the files needed for the simulation:

- MAMMOTH—Rattlesnake YAML input file;
- Yak multigroup cross section libraries input files (XML);
- Yak alias files used to define the perturbed variables (XML);
- Bison YAML input file;
- Bison mesh file;
- Bison function file for the fuel's coefficient of thermal expansion as a function of temperature;

- Rattlesnake mesh file.

As well as **<Model>**, **<Sampler>** and outputs, such as **<OutStreams>** and **<DataObjects>**.

19.13 MELCOR Interface

The current implementation of MELCOR interface is valid for MELCOR 2.1/2.2; its validity for MELCOR 1.8 is **not been tested**.

19.13.1 Sequence

In the **<Sequence>** section, the names of the steps declared in the **<Steps>** block should be specified. As an example, if we called the first multirun "Grid_Sampler" and the second multirun "MC_Sampler" in the sequence section we should see this:

```
<Sequence>Grid_Sampler,MC_Sampler</Sequence>
```

19.13.2 batchSize and mode

For the **<batchSize>** and **<mode>** sections please refer to the **<RunInfo>** block in the previous chapters.

19.13.3 RunInfo

After all of these blocks are filled out, a standard example RunInfo block may look like the example below:

```
<RunInfo>
  <WorkingDir>~/workingDir</WorkingDir>
  <Sequence>Grid_Sampler,MC_Sampler</Sequence>
  <batchSize>8</batchSize>
</RunInfo>
```

In this example, the **<batchSize>** is set to 8; this means that 8 simultaneous (parallel) instances of MELCOR are going to be executed when a sampling strategy is employed.

19.13.4 Files

In the **<Files>** section, as specified before, all of the files needed for the code to run should be specified. In the case of MELCOR, the files typically needed are:

- MELCOR Input file (file extension ".i" or ".inp")
- Restart file (if present)

Example:

```
<Files>
  <Input name='melcorInputFile' type=''>inputFileMelcor.i</Input>
  <Input name='aRestart' type=''>restartFile</Input>
</Files>
```

It is a good practice to put inside the working directory (**<WorkingDir>**) all of these files.

It is important to notice that the interface output collection (i.e. the parser of the MELCOR output) currently is able to extract *CONTROL VOLUME HYDRODYNAMICS EDIT AND CONTROL FUNCTION EDIT* data only. Only those variables are going to be exported and make available to RAVEN. In addition, it is important to notice that:

- **the simulation time is stored in a variable called *"time"*;**
- **all the variables specified in the *CONTROL VOLUME HYDRODYNAMICS EDIT* block are going to be converted using underscores. For example, the following EDITs:**

VOLUME	PRESSURE	TLIQ	TVAP	MASS
	PA	K	K	KG
1	1.00E+07	584.23	584.23	1.66E+03

will be converted in the following way (CSV):

$time$	$volume_1_PRESSURE$	$volume_1_TLIQ$	$volume_1_TVAP$	$volume_1_MASS$
1.0	1.00E+07	584.23	584.23	1.66E+03

CONTROL FUNCTION EDIT data will not be converted in this manner. All data will be labeled using a label identical to what was entered in the MELCOR input file, with no changes.

Remember also that a MELCOR simulation run is considered successful (i.e., the simulation did not crash) if it terminates with the following message:

Normal termination

If the a MELCOR simulation run stops with messages other than this one than the simulation is considered as crashed, i.e., it will not be saved. Hence, it is strongly recommended to set up the MELCOR input file so that the simulation exiting conditions are set through control logic trip variables.

19.13.5 Models

For the **<Models>** block here is a standard example of how it would look when using MELCOR 2.1/2.2 as the external code:

```
<Models>
  <Code name='MyMELCOR' subType='Melcor'>
    <executable>~/path_to_the_executable_of_melcor</executable>
    <preexec>~/path_to_the_executable_of_melgen</preexec>
  </Code>
</Models>
```

As it can be seen above, the **<preexec>** node must be specified, since MELCOR 2.1/2.2 must run the MELGEN utility code before executing. Once the **<preexec>** node is inputted, the execution of MELGEN is performed automatically by the Interface.
In addition, if some command line parameters need to be passed to MELCOR, the user might use (optionally) the **<clargs>** XML nodes.

```
<Models>
  <Code name='MyMELCOR' subType='Melcor'>
    <executable>~/path_to_the_executable_of_melcor</executable>
    <preexec>~/path_to_the_executable_of_melgen</preexec>
    <clargs type="text" arg="-r␣whatever␣command␣line␣
      ↪ instruction"/>
  </Code>
</Models>
```

19.13.6 Distributions

The **<Distribution>** block defines the distributions that are going to be used for the sampling of the variables defined in the **<Samplers>** block. For all the possible distributions and all their possible inputs please see the chapter about Distributions (see 9). Here we report an example of a Normal distribution:

```
<Distributions verbosity='debug'>
    <Normal name="temper">
      <mean>1.E+7</mean>
      <sigma>1.5</sigma>
      <upperBound>9.E+6</upperBound>
      <lowerBound>1.1E+7</lowerBound>
    </Normal>
 </Distributions>
```

It is good practice to name the distribution something similar to what kind of variable is going to be sampled, since there might be many variables with the same kind of distributions but different input parameters.

19.13.7 Samplers

In the **<Samplers>** block we want to define the variables that are going to be sampled. **Example**: We want to do the sampling of 1 single variable:

- The in pressure (P_in) of a control volume regulated by a Tabular Function TF_TAB

We are going to sample this variable using two different sampling methods: Grid and Monte-Carlo.

The interface of MELCOR uses the ***GenericCode*** (see section 19.1) interface for the input perturbation; this means that the original input file (listed in the **<Files>** XML block) needs to implement wild-cards. In this example we are sampling the variable:

- *PRE*, which acts on the Tabular Function TF_TAB whose TF_ID is P_in.

We proceed to do so for both the Grid sampling and the MonteCarlo sampling.

```
<Samplers verbosity='debug'>
  <Grid name='Grid_Sampler' >
    <variable name='PRE'>
      <distribution>temper</distribution>
      <grid type='CDF' construction='equal'  steps='10'>0.001
         ↪ 0.999</grid>
    </variable>
  </Grid>
  <MonteCarlo name='MC_Sampler'>
```

```
      <samplerInit>
        <limit>1000</limit>
      </samplerInit>
     <variable name='PRE'>
       <distribution>temper</distribution>
  </MonteCarlo>
</Samplers>
```

It can be seen that each variable is connected with a proper distribution defined in the **<Distributions>** block (from the previous example). The following demonstrates how the input for the variable is read.

We are sampling a variable whose wild-card in the original input file is named $\$RAVEN-PRE\$$ using a Grid sampling method. The distribution that this variable is following is a Normal distribution (see section above). We are sampling this variable beginning from 0.001 (CDF) in 10 *equal* steps of 0.0998 (CDF).

19.13.8 Steps

For a MELCOR interface, the **<MultiRun>** step type will most likely be used. First, the step needs to be named: this name will be one of the names used in the **<sequence>** block. In our example, `Grid_Sampler` and `MC_Sampler`.

```
      <MultiRun name='Grid_Sampler' verbosity='debug'>
```

With this step, we need to import all the files needed for the simulation:

- MELCOR input file
- any other file needed by the calculation (e.g. restart file)

```
    <Input    class='Files' type=''>inputFileMelcor.i</Input>
    <Input    class='Files' type=''>restartFile</Input>
```

We then need to define which model will be used:

```
    <Model  class='Models' type='Code'>MyMELCOR</Model>
```

We then need to specify which Sampler is used, and this can be done as follows:

```
    <Sampler class='Samplers' type='Grid'>Grid_Sampler</Sampler>
```

And lastly, we need to specify what kind of output the user wants. For example the user might want to make a database (in RAVEN the database created is an HDF5 file). Here is a classical example:

```
    <Output  class='Databases' type='HDF5'>Grid_out</Output>
```

Following is the example of two MultiRun steps which use different sampling methods (Grid and Monte Carlo), and creating two different databases for each one:

```
<Steps verbosity='debug'>
  <MultiRun name='Grid_Sampler' verbosity='debug'>
    <Input   class='Files' type=''>inputFileMelcor.i</Input>
    <Input   class='Files' type=''>restartFile</Input>
    <Model   class='Models'    type='Code'>MyMELCOR</Model>
    <Sampler class='Samplers'  type='Grid'>Grid_Sampler</Sampler>
    <Output  class='Databases' type='HDF5'>Grid_out</Output>
    <Output  class='DataObjects' type='PointSet'
       ↪ >GridMelcorPointSet</Output>
    <Output  class='DataObjects'
       ↪ type='HistorySet'>GridMelcorHistorySet</Output>
  </MultiRun>
  <MultiRun name='MC_Sampler' verbosity='debug'
     ↪ re-seeding='210491'>
    <Input   class='Files' type=''>inputFileMelcor.i</Input>
    <Input   class='Files' type=''>restartFile</Input>
    <Model   class='Models'    type='Code'>MyMELCOR</Model>
    <Sampler class='Samplers'
       ↪ type='MonteCarlo'>MC_Sampler</Sampler>
    <Output  class='Databases' type='HDF5'      >MC_out</Output>
    <Output  class='DataObjects' type='PointSet'
       ↪ >MonteCarloMelcorPointSet</Output>
    <Output  class='DataObjects'
       ↪ type='HistorySet'>MonteCarloMelcorHistorySet</Output>
  </MultiRun>
</Steps>
```

19.13.9 Databases

As shown in the **<Steps>** block, the code is creating two database objects called `Grid_out` and `MC_out`. So the user needs to input the following:

```
<Databases>
  <HDF5 name="Grid_out" readMode="overwrite"/>
```

```
  <HDF5 name="MC_out" readMode="overwrite"/>
</Databases>
```

As listed before, this will create two databases. The files will have names corresponding to their **name** appended with the .h5 extension (i.e. `Grid_out.h5` and `MC_out.h5`).

19.13.10 DataObjects

As shown in the **<Steps>** block, the code is creating 4 data objects (2 HistorySet and 2 PointSet) called `GridMelcorPointSet` `GridMelcorHistorySet` `MonteCarloMelcorPointSet` and `MonteCarloMelcorHistorySet`. So the user needs to input the following block as well, where the Input and Output variables are listed:

```
  <DataObjects>
    <PointSet name="GridMelcorPointSet">
      <Input>PRE</Input>
      <Output>
        time,volume_1_PRESSURE,volume_1_TLIQ,
        volume_1_TVAP,volume_1_MASS
      </Output>
    </PointSet>
    <HistorySet name="GridMelcorHistorySet">
      <Input>PRE</Input>
      <Output>
        time,volume_1_PRESSURE,volume_1_TLIQ,
        volume_1_TVAP,volume_1_MASS
      </Output>
    </HistorySet>
    <PointSet name="MonteCarloMelcorPointSet">
      <Input>PRE</Input>
      <Output>
        time,volume_1_PRESSURE,volume_1_TLIQ,
        volume_1_TVAP,volume_1_MASS
      </Output>
    </PointSet>
    <HistorySet name="MonteCarloMelcorHistorySet">
      <Input>PRE</Input>
      <Output>
        time,volume_1_PRESSURE,volume_1_TLIQ,
        volume_1_TVAP,volume_1_MASS
      </Output>
    </HistorySet>
```

```
</DataObjects>
```

As mentioned before, this will create 4 DataObjects.

19.14 SCALE Interface

This section presents the main aspects of the interface between RAVEN and SCALE system, the consequent RAVEN input adjustments and the modifications of the SCALE files required to run the two coupled codes.
Note: Considering the large amount of SCALE sequences, this interface is currently limited in driving the following SCALE calculation codes:

- ***ORIGEN***
- ***TRITON (using NEWT as transport solver)***

In the following sections a short explanation on how to use RAVEN coupled with SCALE is reported.

19.14.1 Models

As for any other Code, in order to activate the SCALE interface, a **`<Code>`** XML node needs to be inputted, within the main XML node **`<Models>`**.
The **`<Code>`** XML node contains the information needed to execute the specific External Code.

This XML node accepts the following attributes:

- **`name`**, ***required string attribute***, user-defined identifier of this model. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.
- **`subType`**, ***required string attribute***, specifies the code that needs to be associated to this Model. **Note:** See Section 19 for a list of currently supported codes.

This model can be initialized with the following children:

- **`<executable>`** ***string, required field*** specifies the path of the executable to be used. **Note:** Either an absolute or relative path can be used.
- **`<alias>`** ***string, optional field*** specifies alias for any variable of interest in the input or output space for the Code. These aliases can be used anywhere in the RAVEN input to refer

to the Code variables. 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.
Default: None

In addition (and specifc for the SCALE interface), the **<Code>** can contain the following optional nodes:

- **<sequence>**, optional, comma separated list. In this node the user can specify a list of sequences that need to be executed in sequence. For example, if a TRITON calculation needs to be followed by an ORIGEN decay heat calculation the user would input here the sequence "*triton,origen*".
  Default: triton.
  Note: Currently only the following entries are supported:
  - "*triton*"
  - "*origen*"
  - "*triton,origen*"
- **<timeUOM>**, optional, string. In this node the user can specify the *units* for the independent variable "time". If the outputs are exported by SCALE in a different unit (e.g days, years, etc.), the SCALE interface will convert all the different time scales into the unit here specified (in order to have a consistent (and unique) time scale). Available are:
  - "*s*", seconds
  - "*m*", minutes
  - "*h*", hours
  - "*d*", days
  - "*y*", years

  Default: s

An example is shown below:

```
<Models>
    <Code name="MyScale" subType="Scale">
      <executable>path/to/scalerte</executable>
```

```
      <sequence>triton,origen</sequence>
      <timeUOM>d</timeUOM>
    </Code>
</Models>
```

19.14.2 Files

The **<Files>** XML node has to contain all the files required by the particular sequence (s) of the external code (SCALE) to be run. This involves not only the input file(s) (.inp) but also the auxiliary files that might be needed (e.g. binary initial compositions, etc.). As mentioned, the current SCALE interface only supports TRITON and ORIGEN sequences. For this reason, depending on the type of sequence (see previous section) to be run, the relative input files need to be marked with the sequence they are associated with. This means that the type of the input file must be either "triton" or "origen". The auxiliary files that might be needed by a particular sequence (e.g. binary initial compositions, etc.) should not be marked with any specific type (i.e. *type*=""). Example:

```
<Files>
  <Input name="triton_input"
     ↪ type="triton">pwr_depletion.inp</Input>
  <Input name="origen_input" type="origen">decay_calc.inp</Input>
  <Input name="binary_comp" type="">pwr_depletion.f71</Input>
</Files>
```

The files mentioned in this section need, then, to be placed into the working directory specified by the **<workingDir>** node in the **<RunInfo>** XML node.

19.14.2.1 Output Files conversion

Since RAVEN expects to receive a CSV file containing the outputs of the simulation, the results in the SCALE output files need to be converted by the code interface.
As mentioned, the current interface is able to collect data from TRITON and ORIGEN sequences only.
The following information is collected from TRITON output:

- ***k-eff and k-inf time-dep information***

```
 Outer    Eigenvalue  Eigenvalue  Max Flux    Max Flux      Max Fuel    Max Fuel      Wall    Elapsed    Iteration  CPU    Inners
 Iter. #              Delta       Delta     Location(r,g)    Delta     Location(r,g)  Clock   CPU Time    CPU Time  Usage  Converged
 ---------------------------------------------------------------------------------------------------------------------------------
     1    1.00000    0.000E+00 6.480E+09 (    4,252)   1.000E+00 (  614,  0) 14:16:42   89.9 s     89.9 s  92.7%     F
     2    0.35701    1.801E+00 4.149E+01 (  319,  4)   2.673E+00 ( 7035,  0) 14:18:16  182.8 s     92.9 s  98.8%     F
k-eff =        0.94724509      Time=       0.00d Nominal conditions

  Four-Factor Estimate of k-infinity.  Fast/Thermal boundary:   0.6250 eV
```

```
Fiss. neutrons/thermal abs. (eta):          1.279827
Thermal utilization (f):                    0.960903
Resonance Escape probability (p):           0.706209
Fast-fission factor (epsilon):              1.091716
                                          ------------
Infinite neutron multiplication             0.948143
```

that will be converted in the following way (CSV):

Table 14: CSV transport info

time	keff	iter_number	keff_delta	max_flux_delta	kinf	kinf_epsilon	kinf_p	kinf_f	kinf_eta
0.00	0.94724509	2	1.801E+00	4.149e+01	0.948143	1.091716	0.706209	0.960903	1.279827

- ***material powers***

```
--- Material powers for depletion pass no.   1 (MW/MTIHM) ---
    Time =      0.00 days (    0.000 y), Burnup =     0.000     GWd/MTIHM, Transport k=  0.9473

                   Total      Fractional   Mixture      Mixture      Mixture
   Mixture         Power        Power      Power      Thermal Flux  Total Flux
   Number      (MW/MTIHM)      (---)     (MW/MTIHM)   n/(cm^2*sec)  n/(cm^2*sec)
     13          32.985      0.99054      32.985      5.3666e+13    1.2574e+14
      6           0.252      0.00757       N/A        2.7587e+13    9.1781e+13
   Total         33.300      1.00000
```

that will be converted in the following way (CSV):

Table 15: CSV material powers

time	bu	tot_power_mix_13	fract_power_mix_13	th_flux_mix_13	tot_flux_mix_13	tot_power_mix_6	fract_power_mix_6	th_flux_mix_6	tot_flux_mix_6
1.0E-06	0.0	32.985	0.99054	5.3666e+13	1.2574e+14	0.252	0.00757	2.7587e+13	9.1781e+13

- ***nuclide/element tables***

```
            | nuclide concentrations
            | time: days
     grams  |    0.00e+00d
------------+-------------------
      u235  |   2.9619e+04
      u238  |   9.6993e+05
  subtotal  |   1.0010e+06
     total  |   1.1858e+06
```

that will be converted in the following way (CSV):

Table 16: CSV Nuclide/element Tables

time	u235_conc	u238_conc
0.00	2.9619e+04	9.6993e+05

The following information is collected from ORIGEN output:

- ***history overview***

```
=================================================================================================================
=   History overview for case 'decay' (#1/1)                                                                    =
-----------------------------------------------------------------------------------------------------------------
  step        t0          t1          dt          t        flux      fluence      power      energy
  (-)       (sec)       (sec)       (s)         (s)    (n/cm2-s)    (n/cm2)      (MW)       (MWd)
    1   0.0000E+00  1.0000E-06  1.0000E-06  1.0000E-06  0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00
```

Table 17: CSV History Overview

time	t0	t1	dt	flux	fluence	power	energy
1.0E-06	0.0	1.0E-06	1.0E-06	0.0	0.0	0.0	0.0

that will be converted in the following way (CSV):

- ***concentration tables***

```
=================================================================================================================
=   Nuclide concentrations in watts, actinides for case 'decay' (#1/1)                                         =
-----------------------------------------------------------------------------------------------------------------
  (relative cutoff; integral of concentrations over time >   1.00E-04 % of integral of all concentrations over time)
.
                 0.0E+00sec   1.0E-06sec
  th231        8.6167E-08   8.6167E-08
  th234        7.7763E-09   7.7763E-09
____________
  totals        4.6831E+03   4.6831E+03
=================================================================================================================
.
.
=================================================================================================================
=   Nuclide concentrations in watts, fission products for case 'decay' (#1/1)                                  =
-----------------------------------------------------------------------------------------------------------------
  (relative cutoff; integral of concentrations over time >   1.00E-04 % of integral of all concentrations over time)
.
                 0.0E+00sec   1.0E-06sec
  ga74         2.4264E-01   2.4264E-01
  ga75         1.8106E+00   1.8106E+00
____________
  totals        1.2266E+06   1.2266E+06
```

that will be converted in the following way (CSV):

Table 18: CSV Concentration Tables

time	ga74_watts	ga75_watts	subtotals_fission_products	th231_watts	th234_watts	subtotals_actinides	totals_watts
0.0E+00	2.4264E-01	1.8106E+00	1.2266E+06	8.6167E-08	7.7763E-09	4.6831E+03	1.2313E+06
1.0E-06	2.4264E-01	1.8106E+00	1.2266E+06	8.6167E-08	7.7763E-09	4.6831E+03	1.2313E+06

Remember also that a SCALE simulation run is considered successful (i.e., the simulation did not crash) if it does not contain, in the last 20 lines, the following message:

terminated due to errors

If the a SCALE simulation terminates with this message, the simulation is considered "failed", i.e., it will not be saved.

19.14.3 Samplers or Optimizers

In the **`<Samplers>`** or **`<Optimizers>`** block we want to define the variables that are going to be sampled or optimized.
The perturbation or optimization of the input of any SCALE sequence is performed using the approach detailed in the *Generic Interface* section (see 19.1). Briefly, this approach uses "wild-cards" (placed in the original input files) for injecting the perturbed values. For example, if the original input file (that needs to be perturbed) is the following:

```
=origen
case(actual_mass){
  lib{ file="end7dec" }
  mat{ iso=[zr-95=1.0] units="moles" }
  time=[1.0] %1 day
}
end
```

and the initial moles of “zr-95” need to be perturbed, a RAVEN “wild-card” will be defined:

```
=origen
case(actual_mass){
  lib{ file="end7dec" }
  mat{ iso=[zr-95=$RAVEN-zrMoles$] units="moles" }
  time=[1.0] %1 day
}
end
```

Finally, the variable ***zrMoles*** needs to be specified in the specific Sampler or Optimizer that will be used:

```
...
<Samplers>
  <aSampler name='aUserDefinedName' >
    <variable name='zrMoles'>
      ...
    </variable>
  </aSampler>
</Samplers>
...
<Optimizers>
  <anOptimizer name='aUserDefinedName' >
    <variable name='zrMoles'>
      ...
    </variable>
  </anOptimizer>
</Samplers>
...
```

19.15 CTF Interface

This section presents the main aspects of the interface between RAVEN and CTF (COBRA-TF) system, the consequent RAVEN input adjustments and the modifications of the CTF files required to run the two coupled codes.

Note: This interface is currently working only with the specific type of CTF output file (.ctf.out or deck.out (if input file name is deck.inp))

In the following sections a short explanation on how to use RAVEN coupled with CTF is reported.

19.15.1 Sequence

In the **<Sequence>** section, the names of the steps declared in the **<Steps>** block should be specified. As an example, if we called the first MultiRun "Grid_Sampler" and the second MultiRun "MC_Sampler" in the sequence section we should see this:

```
<Sequence>Grid_Sampler, MC_Sampler</Sequence>
```

19.15.2 batchSize and mode

For the **<batchSize>** and **<mode>** sections please refer to the **<RunInfo>** block in the previous chapters.

19.15.3 RunInfo

After all of these blocks are filled out, a standard example RunInfo block may look like the example below:

```
<RunInfo>
  <WorkingDir>~/workingDir</WorkingDir>
  <Sequence>Grid_Sampler,MC_Sampler</Sequence>
  <batchSize>8</batchSize>
</RunInfo>
```

In this example, the **<batchSize>** is set to 8; this means that 8 simulatenous (parallel) instances of CTF are going to be executed when a sampling strategy is employed.

19.15.4 Models

As any other Code, in order to activate the CTF interface, a **<Code>** XML node needs to be inputted, within the main XML node **<Models>**.
The **<Code>** XML node contains the information needed to execute the specific External Code.

This XML node accepts the following attributes:

- **name**, ***required string attribute***, user-defined identifier of this model. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.
- **subType**, ***required string attribute***, specifies the code that needs to be associated to this Model. **Note:** See Section 19 for a list of currently supported codes.

This model can be initialized with the following children:

- **<executable>** ***string, required field*** specifies the path of the executable to be used. **Note:** Either an absolute or relative path can be used.
- **<alias>** ***string, optional field*** specifies alias for any variable of interest in the input or output space for the Code. These aliases can be used anywhere in the RAVEN input to refer to the Code variables. 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.
  Default: None

An example is shown below:

```
<Models>
    <Code name="MyCobraTF" subType="CTF">
      <executable>path/to/cobratf</executable>
    </Code>
</Models>
```

19.15.5 Files

The **<Files>** XML node has to contain all the files required to run the external code (CTF). For RAVEN coupled with CTF, there are three input files. CTF input file (.inp) is required by the code. This input file includes all the geometry, boundary and calculation definitions.

There are two additional files (*optional*) that can be used for model parameter perturbation(s) (vuq_param.txt, vuq_mult.txt). These two files can be used to change variables of models embedded in CTF. The "vuq_param.txt" file includes parameter values, and "vuq_mult.txt" file includes multipliers or additions to parameters. These files are not required by CTF unless a parameter exposure is desired. One, both or neither of them can be included in the simulation folder. The code first controls if these files exist and does modifications accordingly if needed.

The **<Files>** XML node contains the information needed to execute CTF.

This XML node accepts the following attributes:

- **name**, ***required string attribute***, user-defined identifier of this model. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.
- **type**, ***required string attribute***, specifies the input type used by CTF (ctf, vuq_mult, vuq_param). Accepted types are as follows.
  - CTF, ***required string attribute***, identifies the CTF input file (geometry, boundary, calculation options, etc.) and the code currently accept any name for input. One CTF input file is required.
  - vuq_mult, ***optional string attribute if closure modifiers are used***, identifies the closure term multiplier input file. If user needs to alter closure terms this file should be used and named vuq_mult.txt. No other file name is accepted.
  - vuq_param, ***optional string attribute if model parameter modifiers are used***, identifies the model parameter input file. If user needs to change model parameters this file this model should be used and named vuq_params.txt. No other file name is accepted.
  - "", Empty type is also accepted by RAVEN input to perturb. Currently, CTF does not use any other input file that is not mentioned above, but to sample auxiliary files, this option can be used.

Example:

```
<Files>
  <Input name="CTF_input" type="ctf">case1.inp</Input>
  <Input name="vuq_param_input"
     ↪ type="vuq_param">vuq_param.txt</Input>
  <Input name="vuq_mult_input"
     ↪ type="vuq_mult">vuq_mult.txt</Input>
  <Input name="auxiliary_input" type="">auxiliaryInput</Input>
</Files>
```

The files mentioned in this section need, then, to be placed into the working directory specified by the **<workingDir>** node in the **<RunInfo>** XML node.

19.15.5.1 Output Files Conversion

Since RAVEN expects to receive a CSV file containing the outputs of the simulation, the results in the CTF output files (.ctf.out or deck.out) need to be converted by the code interface.

It is important to note that the interface output collection (i.e., the parser of the CTF output) is currently able to extract major edit data (.ctf.out or deck.out) only. Only those variables printed in the "major edit" output files are exported and made available to RAVEN.

The following information is collected from CTF output file (.ctf.out or deck.out):

- ***average properties for channels***

```
*****************************************************************************************************************
          simulation time =      1.03030  seconds            aver. properties for channels
node   dist.  quality     void fraction              mass flow                  enthalpy incr.     enthalpy     heat added
 no.    (ft.)                                         (lbm/s)                       (btu/s)          (btu/s)       (btu/s)
                      liq.   vapor   entr.  liquid  vapor  entr.   integr.  liquid  vapor  integr.   mixture   liquid  vapor  integr.

 50    12.00   -.119   1.000  0.000  0.000   16.39   0.00   0.00   16.39     32.41   0.00   32.41   10683.98   32.37   0.00   32.37
```

that will be converted in the following way (CSV):

Table 19: CSV transport info (average properties for channels)

time	AVG_ch_ax50_quality	AVG_ch_ax50_voidFractionLiquid	AVG_ch_ax50_voidFractionVapor	AVG_ch_ax50_volumeEntrainFraction	...
1.03030	-.119	1.000	0.000	0.000	...

- ***fluid properties for each sub-channel***

```
                 simulation time =      0.00000   seconds            fluid properties for channel   19
node   dist. pressure   velocity                void fraction            flow rate          flow     heat added          gama
 no.   (ft.) (psi)      (ft/sec)                                          (lbm/s)           reg.      (btu/s)           (lbm/s)
                      liquid  vapor  entr.  liquid   vapor   entr.   liquid   vapor    entr.          liquid     vapor


155  0.00   1251.687   2.66    2.66   0.01   1.0000  0.0000  0.0000   0.12456   0.0000   0.00000   0    0.595E-01  0.000E+00    0.00
```

that will be converted in the following way (CSV):

Table 20: CSV transport info (fluid properties for channels)

time	ch19_ax155_pressure	ch19_ax155_velocityLiquid	ch19_ax155_velocityVapor	ch19_ax155_velocityEntrain	ch19_ax155_voidFractionLiquid	...
0.00	1251.687	2.66	2.66	0.01	1.00	...

- ***nuclear fuel rod***

```
          nuclear fuel rod no.  1                          simulation time =     0.00 seconds
             surface no.  1 of  1
          ------------------------                 conducts heat to channels  1  0  0  0  0  0        geometry type =  1
                                                   and azimuthally to surfaces   1 and   1            no. of radial nodes = 13

*********************************************************************************************

  rod     axial     fluid temperatures   surface      heat       -clad temperatures-      gap        -fuel temperatures-
  node   location      (deg-f)           heat flux   transfer        (deg-f)          conductance        (deg-f)
  no.    (in.)     liquid   vapor        (b/h-ft2)    mode      outside   inside      (b/h-ft2-f)    surface   center
  ----   --------  ------   -----       ---------   --------   -------   ------     -----------    -------   ------

   10    22.80     464.1    467.1       0.5929E+04    spl       466.08    592.98       1594.2        859.58    2946.22
```

that will be converted in the following way (CSV):

Table 21: CSV transport info (nuclear fuel rod)

time	fuelRod10_surface1_ax10_fluidTemperatureLiquid	fuelRod10_surface1_ax10_fluidTemperatureVapor	...
0.00	464.1	467.1	...

- ***cyclindrical tube***

  Warning: CTF reports results for cylindrical tubes based on the flow type. Not every result will be available depending on the internal or external flow type. Check output file and see if the flow around heat slab is internal or external. If the user requests values that are not in the output file reported values will be from different columns and wrong. For example there is no outside surface liquid temperature when flow is internal.

```
   cylindrical tube rod no.  5                          simulation time =    2.00 seconds
         surface no.  1 of  4
  ------------------------          conducts heat to channels 10  0  0  0  0  0              geometry type =  2
                                    and azimuthally to surfaces   4 and   2                   no. of radial nodes =  2


*****************************************************************************************************************

rod    axial    *------------- outside surface ----------------*  *---------------- inside surface ----------------*
node   location   heat flux   h.t.  **** temperatures (deg-f) ****   **** temperatures (deg-f) ****    h.t.   heat flux
no.    (in.)     (b/h-ft2)   mode    wall     vapor    liquid       liquid    vapor    wall            mode   (b/h-ftl)
----   --------  ---------   ----   -------  -------  -------      -------  -------  -------          ----   ---------

 51    144.00   -0.4424E+03  spl    629.71   653.31   629.77                         629.68                  0.0000E+00
```

that will be converted in the following way (CSV):

Table 22: CSV transport info (cylindrical tube)

time	cylRod10_surface1_ax51_outsideSurfaceHeatFlux	cylRod10_surface1_ax51_outsideSurfaceWallTemperature	...
0.00	-0.4424E+03	629.71	...

- ***heat slab (tube)***

  Warning: CTF reports results for heat slabs based on the flow type. Not every result will be available depending on the internal or external flow type. Check output file and see if the flow around heat slab is internal or external. If the user requests values that are not in the output file reported values will be from different columns and wrong. For example there is no outside surface liquid temperature when flow is internal.

```
          heat slab no.  1  (tube)                 simulation time =   20.00 seconds
                                                   fluid channel on inside surface =  1
                                                   fluid channel on outside surface =  0
                                                   geometry type =  1
                                                   no. of nodes =   2


*****************************************************************************************************************

  rod    axial    *------------- outside surface ----------------*  *---------------- inside surface ----------------*
  node   location   heat flux   h.t.  ** temperatures (deg-f) **  ** temperatures (deg-f) ****    h.t.   heat flux
  no.    (in.)     (b/h-ft2)   mode    wall     vapor    liquid   liquid    vapor    wall            mode   (b/h-ftl)
  ----   --------  ---------   ----   -------  -------  -------  -------  -------  -------          ----   ---------

   21     19.69    999666E+00         200.00                     212.00   212.00   247.85          tran   0.2641E+02
```

that will be converted in the following way (CSV):

Table 23: CSV transport info (heat slab (tube) tube)

time	heatSlab1_ax21_outsideSurfaceHeatFlux	heatSlab1_ax21_outsideSurfaceWallTemperature	...
0.00	999666E+00	200.00	...

- ***CTF's Output Variables and Corresponding Names in CSV file***

  In CSV file, the output results obtained from the CTF output file (.ctf.out) will be saved with the names as described in Table 24.

Table 24: Variables Name List in CSV File
NN: Axial Node Number; CN: Channel Number;
RN: Rod Number; SN: Surface Number: HN: Heat Slab Number

Output Variable	**Name in CSV file**
simulation time	time
channels' average height	AVG_ch_axNN_height
channels' average quality	AVG_ch_axNN_quality
channels' average void fraction (liquid)	AVG_ch_axNN_voidFractionLiquid
channels' average void fraction (vapor)	AVG_ch_axNN_voidFractionVapor
channels' average entrainment (volumetric) fraction	AVG_ch_axNN_volumeEntrainFraction
channels' average mass flow rate (liquid)	AVG_ch_axNN_massFlowRateLiquid
channels' average mass flow rate (vapor)	AVG_ch_axNN_massFlowRateVapor
channels' average entrainment rate (mass flow rate)	AVG_ch_axNN_massFlowRateEntrain
channels' average mass flow rate (integrated)	AVG_ch_axNN_massFlowRateIntegrated
channels' average enthalpy increase (liquid)	AVG_ch_axNN_enthalpyIncreaseLiquid
channels' average enthalpy increase (vapor)	AVG_ch_axNN_enthalpyIncreaseVapor
channels' average enthalpy increase (integrated)	AVG_ch_axNN_enthalpyIncreaseIntegrated
channels' average mixture enthalpy	AVG_ch_axNN_enthalpyMixture
channels' average heat added to liquid	AVG_ch_axNN_heatAddedToLiquid
channels' average heat added to vapor	AVG_ch_axNN_heatAddedToVapor
channels' average heat added (integrated)	AVG_ch_axNN_heatAddedIntegrated'
channel height	chCN_axNN_height
channel pressure	chCN_axNN_pressure
channel liquid velocity	chCN_axNN_velocityLiquid
channel vapor velocity	chCN_axNN_velocityVapor
channel entrainment rate (velocity)	chCN_axNN_velocityEntrain
channel void fraction (liquid)	chCN_axNN_voidFractionLiquid
channel void fraction (vapor)	chCN_axNN_voidFractionVapor
channel volume fraction of entrainment liquid	chCN_axNN_volumeEntrainFraction
channel mass flow rate (liquid)	chCN_axNN_massFlowRateLiquid
channel mass flow rate (vapor)	chCN_axNN_massFlowRateVapor
channel entrainment rate (mass flow rate)	chCN_axNN_massFlowRateEntrain
channel flow regime ID	chCN_axNN_flowRegimeID
channel heat added to liquid	chCN_axNN_heatAddedToLiquid
channel heat added to vapor	chCN_axNN_heatAddedToVapor
channel evaporation rate	chCN_axNN_evaporationRate
channel enthalpy of vapor	chCN_axNN_enthalpyVapor
channel enthalpy of saturated vapor	chCN_axNN_enthalpySaturatedVapor
channel enthalpy difference between vapor and saturated vapor	chCN_axNN_enthalpyVapor-SaturatedVapor
channel enthalpy of liquid	chCN_axNN_enthalpyLiquid
channel enthalpy of saturated liquid	chCN_axNN_enthalpySaturatedLiquid
channel enthalpy difference between liquid and saturated liquid	chCN_axNN_enthalpyLiquid-SaturatedLiquid
channel enthalpy of mixture	chCN_axNN_enthalpyMixture
channel density of liquid	chCN_axNN_densityLiquid
channel density of vapor	chCN_axNN_densityVapor
channel density of mixture	chCN_axNN_densityMixture

Output Variable	Name in CSV file
channel net entrainment rate (difference between entrainment rate and de-entrainment rate)	chCN_axNN_netEntrainRate
channel enthalpy of the mixture of non-condensable gases	chCN_axNN_enthalpyNonCondensableMixture
channel density of the mixture of non-condensable gases	chCN_axNN_densityNonCondensableMixture
channel steam volume fraction [0-100]	chCN_axNN_volumeFractionSteam
channel air volume fraction [0-100]	chCN_axNN_volumeFractionAir
channel total equivalent diameter of the liquid droplets (all droplets as a single big one) (diam-ld)	chCN_axNN_equiDiameterLiquidDroplet
channel averaged diameter of liquid droplets field (diam-sd)	chCN_axNN_avgDiameterLiquidDroplet
channel averaged flow rate of liquid droplets field (flow-sd)	chCN_axNN_avgFlowRateLiquidDroplet
channel averaged velocity of liquid droplets field (veloc-sd)	chCN_axNN_avgVelocityLiquidDroplet
channel evaporation rate of liquid droplets field (gamsd)	chCN_axNN_evaporationRateLiquidDroplet
fuel rod height	fuelRodRN_surfaceSN_axNN_height
fuel rod fluid temperatures (liquid)	fuelRodRN_surfaceSN_axNN_fluidTemperatureLiquid
fuel rod fluid temperatures (vapor)	fuelRodRN_surfaceSN_axNN_fluidTemperatureVapor
fuel rod surface heat flux	fuelRodRN_surfaceSN_axNN_surfaceHeatflux
clad outer surface temperature	fuelRodRN_surfaceSN_axNN_cladOutTemperature
clad iRNer surface temperature	fuelRodRN_surfaceSN_axNN_cladInTemperature
gap conductance	fuelRodRN_surfaceSN_axNN_gapConductance
fuel outer suface temperature	fuelRodRN_surfaceSN_axNN_fuelTemperatureSurface
fuel center temperature	fuelRodRN_surfaceSN_axNN_fuelTemperatureCenter
cylindrical tube height	cylRodRN_surfaceSN_axNN_height
cylindrical tube outside surface heat flux	cylRodRN_surfaceSN_axNN_outsideSurfaceHeatFlux
cylindrical tube outside surface wall temperature	cylRodRN_surfaceSN_axNN_outsideSurfaceWallTemperature
cylindrical tube outside surface vapor temperature	cylRodRN_surfaceSN_axNN_outsideSurfaceVaporTemperature
cylindrical tube outside surface liquid temperature	cylRodRN_surfaceSN_axNN_outsideSurfaceLiquidTemperature
cylindrical tube inside surface wall temperature	cylRodRN_surfaceSN_axNN_insideSurfaceWallTemperature
cylindrical tube inside surface vapor temperature	cylRodRN_surfaceSN_axNN_insideSurfaceVaporTemperature
cylindrical tube inside surface liquid temperature	cylRodRN_surfaceSN_axNN_insideSurfaceLiquidTemperature
cylindrical tube inside surface heat flux	cylRodRN_surfaceSN_axNN_insideSurfaceHeatFlux
heat slab (tube) height	heatSlabHN_axNN_height
heat slab (tube) outside surface heat flux	heatSlabHN_axNN_outsideSurfaceHeatFlux
heat slab (tube) outside surface wall temperature	heatSlabHN_axNN_outsideSurfaceWallTemperature
heat slab (tube) outside surface vapor temperature	heatSlabHN_axNN_outsideSurfaceVaporTemperature
heat slab (tube) outside surface liquid temperature	heatSlabHN_axNN_outsideSurfaceLiquidTemperature
heat slab (tube) inside surface wall temperature	heatSlabHN_axNN_insideSurfaceWallTemperature
heat slab (tube) inside surface vapor temperature	heatSlabHN_axNN_insideSurfaceVaporTemperature
heat slab (tube) inside surface liquid temperature	heatSlabHN_axNN_insideSurfaceLiquidTemperature
heat slab (tube) inside surface heat flux	heatSlabHN_axNN_insideSurfaceHeatFlux

***Note:* RAVEN, regonizes failed or crashed CTF runs and no data will be saved from those.**

19.15.6 Distributions

The **<Distribution>** block defines the distributions that are going to be used for the sampling of the variables defined in the **<Samplers>** block. For all the possibile distributions and all their possible inputs please see the chapter about Distributions (see 9). Here we report an example of a Normal distribution:

```
<Distributions verbosity='debug'>
    <Normal name="GridLossCoeff">
      <mean>0.7</mean>
      <sigma>0.1</sigma>
      <upperBound>0.9</upperBound>
      <lowerBound>0.6</lowerBound>
    </Normal>
```

```
    <Uniform name="DB1dist">
      <upperBound>0.026</upperBound>
      <lowerBound>0.020</lowerBound>
    </Uniform>
    <Uniform name="DB2dist">
      <upperBound>0.9</upperBound>
      <lowerBound>0.7</lowerBound>
    </Uniform>
    <Uniform name="DB3dist">
      <upperBound>0.5</upperBound>
      <lowerBound>0.3</lowerBound>
    </Uniform>
 </Distributions>
```

It is good practice to name the distribution something similar to what kind of variable is going to be sampled, since there might be many variables with the same kind of distributions but different input parameters.

19.15.7 Samplers

In the **<Samplers>** block we want to define the variables that are going to be sampled.
The perturbation or optimization of the input of any CTF sequence is performed using the approach detailed in the *Generic Interface* section (see 19.1). Briefly, this approach uses "wild-cards" (placed in the original input files) for injecting the perturbed values. For example, if the original input file (that needs to be perturbed) is the following:
Example: We want to do the sampling of 1 single variable:

- The Grid Loss Coefficient Data is used from sampled values.

We are going to sample this variable using two different sampling methods: Grid and MonteCarlo. The RAVEN input is then written as follows:

```
<Samplers verbosity='debug'>
  <Grid name='Grid_Sampler' >
    <variable name='GrdLss'>
      <distribution>GridLossCoeff</distribution>
      <grid type='CDF' construction='equal'  steps='10'>0.001
         ↪ 0.999</grid>
    </variable>
  </Grid>
  <MonteCarlo name='MC_Sampler'>
     <samplerInit>
       <limit>1000</limit>
     </samplerInit>
    <variable name='GrdLss'>
```

```
      <distribution>GridLossCoeff</distribution>
    </variable>
    <variable name='DB1'>
      <distribution>DB1dist</distribution>
    </variable>
    <variable name='DB2'>
      <distribution>DB2dist</distribution>
    </variable>
    <variable name='DB3'>
      <distribution>DB3dist</distribution>
    </variable>
  </MonteCarlo>
</Samplers>
```

CTF input file should be modified with wild-cards in the following way.

```
*****************************************************************************************
*GROUP 7 - Grid Loss Coefficient Data                                                   *
*****************************************************************************************
**NGR
    7
*Card 7.1
**  NCD NGT  IFGQF IFSDRP  IFESPV  IFTPE  IGTEMP  NFBS  NDM9 NDM10 NDM11 NDM12 NDM13 NDM14
     21   0      0      0       0      0       0     0     0     0     0     0     0     0
*Card 7.2
**        CDL      J   CD1   CD2   CD3   CD4   CD5   CD6   CD7   CD8   CD9  CD10  CD11  CD12
 $RAVEN-GrdLss$    1     1     2     3     4     5     6     7     8     9    10    11    12
      0.90700      1    13    14    15    16    17    18    19    20    21    22    23    24
      0.90700      1    25    26    27    28    29    30    31    32    33    34    35    36
```

It is also possible to modify input values in parameter exposure input files.

Example: We want to do the sampling of 3 correlation parameters (Dittus-Boelter parameter modification, DB1 $\times$ Re^{DB2} $\times$ Pr^{DB3}):

- DB1, DB2, DB3 values will be sampled and vuq_param.txt will be modified with sampled values.

vuq_param.txt and vuq_mult.txt files are modified similarly with defined variable names.

```
k_chen_1 = 0.24
k_chen_2 = 0.75
k_db_1 = $RAVEN-DB1$
k_db_2 = $RAVEN-DB2$
k_db_3 = $RAVEN-DB3$
k_db_4 = 7.86
k_wf_1 = 1.691
k_wf_2 = 0.43
```

It can be seen that each variable is connected with a proper distribution defined in the **<Distributions>** block (from the previous example).

19.15.8 Steps

For a CTF interface, the **<MultiRun>** step type will most likely be used. But **<SingleRun>** step can also be used for plotting and data extraction purposes. First, the step needs to be

named: this name will be one of the names used in the **<sequence>** block. In our example, `Grid_Sampler` and `MC_Sampler`.

```
    <MultiRun name='Grid_Sampler' verbosity='debug'>
```

With this step, we need to import all the files needed for the simulation:

- CTF input files

```
    <Input class="Files" type="ctf">CTFinput</Input>
    <Input class="Files" type="vuq_param">vuq_param</Input>
    <Input class="Files" type="vuq_mult" >vuq_mult</Input>
```

We then need to define which model will be used:

```
    <Model  class='Models' type='Code'>MyCobraTF</Model>
```

We then need to specify which Sampler is used, and this can be done as follows:

```
    <Sampler class='Samplers' type='Grid'>Grid_Sampler</Sampler>
```

And lastly, we need to specify what kind of output the user wants. For example the user might want to make a database (in RAVEN the database created is an HDF5 file). Here is a classical example:

```
    <Output  class='Databases' type='HDF5'>Grid_out</Output>
```

Following is the example of two MultiRun steps which use different sampling methods (Grid and Monte Carlo), and creating two different databases for each one:

```
<Steps verbosity='debug'>
  <MultiRun name='Grid_Sampler' verbosity='debug'>
    <Input   class='Files' type="ctf">CTFinput</Input>
    <Input   class="Files" type="vuq_param">vuq_param</Input>
    <Input   class="Files" type="vuq_mult" >vuq_mult</Input>
    <Model   class='Models'    type='Code'>MyCobraTF</Model>
    <Sampler class='Samplers'  type='Grid'>Grid_Sampler</Sampler>
    <Output  class='Databases' type='HDF5'>Grid_out</Output>
    <Output  class='DataObjects' type='PointSet'
       ↪ >GridCTFPointSet</Output>
    <Output  class='DataObjects'
       ↪ type='HistorySet'>GridCTFHistorySet</Output>
  </MultiRun>
  <MultiRun name='MC_Sampler' verbosity='debug'
     ↪ re-seeding='210491'>
    <Input   class='Files' type=''>CTFinput</Input>
    <Input   class="Files" type="">vuq_param</Input>
    <Input   class="Files" type="" >vuq_mult</Input>
    <Model   class='Models'    type='Code'>MyCobraTF</Model>
```

```
    <Sampler class='Samplers'
       ↪ type='MonteCarlo'>MC_Sampler</Sampler>
    <Output  class='Databases' type='HDF5'      >MC_out</Output>
    <Output  class='DataObjects' type='PointSet'
       ↪ >MonteCarloCobraPointSet</Output>
    <Output  class='DataObjects'
       ↪ type='HistorySet'>MonteCarloCobraHistorySet</Output>
  </MultiRun>
</Steps>
```

19.16 SAPHIRE Interface

This section covers the input specification for running SAPHIRE through RAVEN. It is important to notice that this short explanation assumes that the reader already knows how to use SAPHIRE.

19.16.1 Files

In the **<Files>** section, as specified before, all the files needed for the code to run should be specified. In the case of SAPHIRE, the files typically needed are the following:

- SAPHIRE compressed project inputs with file extension '.zip';
- SAPHIRE macro input file with file extension '.mac'.

Example:

```
  <Files>
    <Input name="macro" type="">changeSet.mac</Input>
    <Input name="saphireInput" type="">saphireInput.zip</Input>
  </Files>
```

19.16.2 Models

In the **<Models>** block SAPHIRE executable needs to be specified. Here is a standard example of what can be used:

```
  <Models>
    <Code name="saphire" subType="Saphire">
      <executable>"C:\Saphire␣8\tools\SAPHIRE.exe"</executable>
      <clargs arg="macro" extension=".mac" type="input"
         ↪ delimiter="="/>
      <clargs arg="project" extension=".zip" type="input"
         ↪ delimiter="="/>
      <outputFile>fixed_output.csv</outputFile>
```

```
    <codeOutput type="uncertainty">et_uq.csv</codeOutput>
    <codeOutput type="uncertainty">ft_uq.csv</codeOutput>
  </Code>
</Models>
```

The **<Code>** XML node contains the information needed to execute the specific External Code. This XML node accepts the following attributes:

- **name**, ***required string attribute***, user-defined identifier of this model. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.
- **subType**, ***required string attribute***, specifies the code that needs to be associated to this Model.

This model can be initialized with the following children:

- **<executable>**, ***string, required field***, specifies the path of the executable to be used; **Note:** Either an absolute or relative path can be used.
- **<clargs>**, ***string, required field***, allows addition of command-line arguments to the execution command. This node is used to specify the input files that are required by SAPHIRE. This node accepts the following attributes:
  - **type**, ***require string attribute***, specifies the type of command-line argument to add. The current option is **'input'**
  - **arg**, ***string, required field*** specifies the flag to be used before the entry.
  - **extension**, ***string, required field***, specifies the type of file extension to use. This links the **<Input>** file in the **<Steps>** to this location in the execution command. Currently only accepts '.zip' and '.mac'.
  - **delimiter**, ***string, required field***, uses to link the **arg** and the **<Input>** with the extension given by **extension**

  Note: As shown in previous example, the following command will be generated:

```
"C:\Saphire␣8\tools\SAPHIRE.exe" project=path/to/
  ↪ saphireInput.zip macro=path/to/changeSet.mac
```

- **<outputFile>**, ***string, optional field***, uses to specify the output file name (CSV only). In this case, the code interface always produce a CSV file named "fixed_output.csv".
- **<codeOutput>**, ***string, required field***, uses to specify output file generated by SAPHIRE that will be processed via the code interface. The following attributes can be specified:
  - **type**, ***required string attribute***, the actual type of the provided file. The only type accepted here is **'uncertainty'**

In this example, two output files “eq_uq.csv” amd “ft_uq.csv” will be processed by the SAPHIRE code interface, and the results will be saved in output file with name “fixed_output.csv”.

19.16.3 Distributions

The **<Distributions>** block defines the distributions that are going to be used for the sampling of the variables defined in the **<Samplers>** block. For all the possible distributions and all their possible inputs, please see the chapter about Distributions (see 9). Here we give a general example:

```
<Distributions>
  <Normal name="allEvents">
      <mean>0.1</mean>
      <sigma>0.025</sigma>
      <lowerBound>0.05</lowerBound>
      <upperBound>0.15</upperBound>
  </Normal>
  <Normal name="mov1Event">
      <mean>0.5</mean>
      <sigma>0.1</sigma>
      <lowerBound>0.3</lowerBound>
      <upperBound>0.8</upperBound>
  </Normal>
  <Normal name="single1">
      <mean>0.2</mean>
      <sigma>0.05</sigma>
      <lowerBound>0.1</lowerBound>
      <upperBound>0.3</upperBound>
  </Normal>
</Distributions>
```

It is good practice to name the distribution similar to what kind of variable is going to be sampled.

19.16.4 Samplers

In the **<Samplers>** block we want to define the variables that are going to be sampled. The perturbation of the input of SAPHIRE MACRO is performed using the approach detailed in the *Generic Interface* section (see 19.1). This approach uses the “wild-cards” (placed in the original input files) for injecting the perturbed values. For example, if one wants to perturb the event tree probabilities of the original input file, i.e.

```
<change set>
  <unmark></unmark>
  <delete>
    <name>ALL-EVENTS</name>
  </delete>
```

```
  <add>
    <name>ALL-EVENTS</name>
    <description>Class change all events Change Set</description>
    <class>
      <event name>*</event name>
      <suscept>1</suscept>
      <probability>1.E-2</probability>
    </class>
  </add>
  <mark name>ALL-EVENTS</mark name>
  <generate></generate>
</change set>
...
<change set>
  <unmark></unmark>
  <delete>
    <name>MOV-1-EVENTS</name>
  </delete>
  <add>
    <name>MOV-1-EVENTS</name>
    <description>Class change subset events Change
      ↪ Set</description>
    <class>
      <event name>?-MOV-CC-1</event name>
      <calc type>1</calc type>
      <probability>5E-3</probability>
    </class>
  </add>
    <mark name>MOV-1-EVENTS</mark name>
    <generate></generate>
</change set>
...

<change set>
  <unmark></unmark>
  <delete>
    <name>SINGLE-1</name>
  </delete>
  <add>
    <name>SINGLE-1</name>
    <description>Single Event Change Set</description>
    <single>
```

```
      <event name>E-MOV-CC-A</event name>
      <calc type>1</calc type>
      <probability>4E-1</probability>
    </single>
  </add>
  <mark name>SINGLE-1</mark name>
  <generate></generate>
</change set>
```

One need to use the RAVEN “wild-cards“ to inject the perturbed values, i.e.

```
<change set>
  <unmark></unmark>
  <delete>
    <name>ALL-EVENTS</name>
  </delete>
  <add>
    <name>ALL-EVENTS</name>
    <description>Class change all events Change Set</description>
    <class>
      <event name>*</event name>
      <suscept>1</suscept>
      <probability>$RAVEN-allEventsPb$</probability>
    </class>
  </add>
  <mark name>ALL-EVENTS</mark name>
  <generate></generate>
</change set>
...
<change set>
  <unmark></unmark>
  <delete>
    <name>MOV-1-EVENTS</name>
  </delete>
  <add>
    <name>MOV-1-EVENTS</name>
    <description>Class change subset events Change
      ↪ Set</description>
    <class>
      <event name>?-MOV-CC-1</event name>
      <calc type>1</calc type>
      <probability>$RAVEN-mov1EventPb$</probability>
    </class>
  </add>
```

```
    <mark name>MOV-1-EVENTS</mark name>
    <generate></generate>
</change set>
...

<change set>
  <unmark></unmark>
  <delete>
    <name>SINGLE-1</name>
  </delete>
  <add>
    <name>SINGLE-1</name>
    <description>Single Event Change Set</description>
    <single>
      <event name>E-MOV-CC-A</event name>
      <calc type>1</calc type>
      <probability>$RAVEN-single1Pb$</probability>
    </single>
  </add>
  <mark name>SINGLE-1</mark name>
  <generate></generate>
</change set>
```

The RAVEN **<Samplers>** input will be
Example:

```
  <Samplers>
    <MonteCarlo name="mcSaphire">
        <samplerInit>
            <limit>2</limit>
        </samplerInit>
        <variable name="allEventsPb">
            <distribution>allEvents</distribution>
        </variable>
        <variable name="mov1EventPb">
            <distribution>mov1Event</distribution>
        </variable>
        <variable name="single1Pb">
            <distribution>single1</distribution>
        </variable>
    </MonteCarlo>
```

19.16.5 Steps

In this section, the **<MultiRun>** will be used. As shown in the following, two SAPHIRE input files listed in **<Files>** are linked here using **<Input>**, the **<Model>** and **<Sampler>** defined in previous sections will be used in this **<MultiRun>**. The outputs will be saved in the **DataObject** "saphireDump", and will be printed via **OutStreams**.

```
<Steps>
  <MultiRun name="sample">
    <Input class="Files" type="">macro</Input>
    <Input class="Files" type="">saphireInput</Input>
    <Model class="Models" type="Code">saphire</Model>
    <Sampler class="Samplers"
      ↪ type="MonteCarlo">mcSaphire</Sampler>
    <Output class="DataObjects"
      ↪ type="PointSet">saphireDump</Output>
    <Output class="OutStreams"
      ↪ type="Print">saphirePrint</Output>
  </MultiRun>
</Steps>
```

19.17 PHISICS Interface

19.17.1 General Information

This section covers the input specification for running PHISICS through RAVEN. The interface can be used to perturb the PHISICS input files including the INSTANT and MRTAU input decks and the following libraries: cross sections, fission yield, decay, fission Q-values, decay Q-values and the XML material input. The interface also the cabablity to work in MRTAU standlone calculations, in INSTANT/MRTAU mode (PHISICS) and in PHISICS/RELAP5 coupled mode (see 19.18).

19.17.2 Files

<Files> includes two attributes **name** and **type** entries, identically as other interfaces. It also includes two optional attributes **perturbable** and **subDirectory**.
The **name** attribute is a user-defined internal name for the file contained in the node. Default: None (required entry).
The **type** attribute identifies which base-level parser an input file is used within. The **type** has to be specified as long as the file is parsed by the interface or an interface's parser. **type** is hardcoded for this speciific inputs, in order to assign each input to its corresponding RAVEN parser. Default: None (required entry if parsed).
The corresponding hardcoded flags accepted by RAVEN are given in Table 26. The type attributes are case-incensitive.

Table 26: Correspondance between the type attributes required and the PHISICS input files

Type Attribute	*Corresponding PHISICS input*	*Perturbable*
decay	MrTau decay library	Yes
inp	XML Instant input	No
path	XML MrTau path-to-libraries file	No
material	XML MrTau material input	Yes
depletion_input	XML MrTau depletion input	No
Xs-Library	XML MrTau library input	No
FissionYield	MrTau fission yield library	Yes
FissQValue	MrTau fission Q-values	Yes
AlphaDecay	MrTau α decay library	Yes
Beta+Decay	MrTau β^+ decay library	Yes
Beta+xDecay	MrTau β^{+*} decay library	Yes
BetaDecay	MrTau β decay library	Yes
BetaxDecay	MrTau β^* decay library	Yes
IntTraDecay	MrTau internal transition decay library	Yes
XS	XML XS scaling factors file	Yes
N,2N	MrTau n,2n library	No
N,ALPHA	MrTau n,α library	No
N,G	MrTau n,γ library	No
N,Gx	MrTau n,γ^* library	No
N,P	MrTau n,proton library	No
budep	MrTau burn-up history	No
CRAM_coeff_PF	MrTau CRAM coefficients	No
IsotopeList	MrTau isotope list input	No
mass	MrTau mass input	No
tabMap	XML tabulation mapping	No

The cross section libraries files can be defined with any **type** attribute. The tabulation mapping file is optional. If a **type='tabMap'** is found in an **<Input>** node, the cross section parser will be based on the tabulation points provided in the tabulation mapping file. If no tabulation mapping file is provided, the cross section parser will print the perturbed cross section in one single tabulation point. The **perturbable** attribute indicates whether the input file can be perturbed. It is an optional boolean attribute. Default: False
The **subDirectory** indicates the subdirectory to which RAVEN search an input file. It is an optional attribute. Default: . / (relative path of the working directory)
The **'Input File'** string is the user-defined input file name. The XML file specifying the library input paths corresponding to the decay, fission yields, the Q-values and the Mrtau standalone inputs will be automatically populated according to the user-input file names. The Instant-MrTau input files material, library, instant control, library path and depletion are also user-defined in the input section. The user does not have to use the default file names. Example:

```
<File>
  <Input name="path"  type="path"  perturbable="False"
    ↪                     >pathMrTau.xml</Input>
  <Input name="dec"   type="decay" perturbable="True"
    ↪ subDirectory="libF" >decayLibrary.dat</Input>
  <Input name="input" type="inp"   perturbable="False"
    ↪                     >inpInstant.xml</Input>
</File>
```

in the example, the path-to-MrTau-libraries is pointed by the **type**="path". The file name associated to the path-to-MrTau-libraries file is then user-defined as **'pathMrTau.xml'**. It is located in the working directory and it cannot be perturbed.
The decay library is pointed by the **type**="decay". The file name associated to the decay library is then user-defined as **'decayLibrary.dat'**. The decay library is located at the relative path "libF". This path and name will be populated in the **'pathMrTau.xml'** file automatically. The decay library can be perturbed.
The Instant XML input is pointed by the **type**="inp". The file name associated to the Instant input is then user-defined as **'inpInstant.xml'**. it is located in the working directory and it cannot be perturbed.

19.17.3 Models

The user provides paths to executables for sampled variables within the **<Models>** block.
The **<Code>** block will contain attributes **name** and **subType**. The **name** identifies that particular **<Code>** model within RAVEN, and **subType** specifies which code interface the model will use.
The **<executable>** block contains the absolute or relative path (with respect to the current working directory) to PHISICS that RAVEN will use to run generated input files.
The **<mrtauStandAlone>** node informs whether or not MrTau is ran in standalone mode. The **<mrtauStandAlone>** accepts only a boolean entry (**'true'**, **'t'**, **'false'** ,**'f'**). It is

case insensitive. Default: false. If **<mrtauStandAlone>** is false, a coupled INSTANT+MrTau calculation is ran, using the Phisics executable.
The **<printSpatialRR>** node indicates if the spatial reaction rates computed by PHISICS are to be included in the RAVEN csv output. The entry is case insensitive. . If False, the total reaction rates are printed instead. Default: False (spatial reaction rate not printed).
The **<printSpatialFlux>** node indicates if the spatial neutron fluxes computed by PHISICS are to be included in the RAVEN csv output. The entry is case insensitive. Default: False (spatial fluxes not printed).
An example of the **<Models>** block is given below:

```
<Models>
  <Code name="PHISICS" subType="Phisics">
     <executable>./path/to/instant/executable</executable>
     <mrtauStandAlone>F</mrtauStandAlone>
     <printSpatialRR>F</printSpatialRR>
     <printSpatialFlux>T</printSpatialFlux>
  </Code>
</Models>
```

In the example, note that because **<mrtauStandAlone>** is false. If **<mrtauStandAlone>** is changed to **'true'**, the path **'/path/to/mrtau/executable'** will be read to get the MrTau executable. In the example, RAVEN is used in PHISICS standalone mode. Spatial neutron fluxes are printed a and no spatial reaction rates are printed in the RAVEN output.

19.17.4 Distributions

The **<Distributions>** block defines all distributions used to sample variables in the current RAVEN run.
For all the possible distributions and their possible inputs please refer to the Distributions chapter (see 9). It is good practice to name a distribution and its corresponding sampled variable with identical root names, and appending sufix to the distribution name, since there might be many variables with the same kind of distributions but different input parameters.

19.17.5 Samplers

The **<Samplers>** block defines the variables to be sampled. After defining a sampling scheme, the variables to be sampled and their distributions are identified in the **<variable>** blocks. The **name** must be formatted according to library which the variable belongs to. The description of the **'variable'** template is detailed in the next sub-sections for the decay constants (19.17.5.1), the fission yields (19.17.5.2), the number densities (19.17.5.3), the fission Q-values (19.17.5.4), the α decay Q-values (19.17.5.5), the β^+ Q-values (19.17.5.6), β^{+*} Q-values (19.17.5.7), β Q-values (19.17.5.8), β^* Q-values (19.17.5.9), the internal transition decay Q-values (19.17.5.10) and the cross section scaling factors (19.17.5.11).

19.17.5.1 Decay constant variable

The **`'variable'`** template is: DECAY|TYPE_OF_DECAY|ISOTOPE. The type of decay (TYPE_OF_DECAY) is the decay mode relative to the isotope's decay constant perturbed. The type of decay depends on the isotope perturbed. If the isotope is an actinide, the available decay modes are:

- BETA;
- BETA+;
- ALPHA.

If the isotope is a fission product, the available decay modes are:

- BETA;
- BETA*;
- BETA+;
- BETA+*;
- ALPHA;
- INTER_TRAN.

The decay types are immediately parsed from the MRTAU decay library. Hence, If the user modifies the decay labels in the decay library, the user will have to modify the her/his decay type in the RAVEN input. The isotope defined in the variable has to originally exist in the decay library.

19.17.5.2 Fission yield variable

The **`'variable'`** template is FY|SPECTRUM|FISSION_ISOTOPE|FISSION_PRODUCT.
The types of spectrum (SPECTRUM) available are: FAST; THERMAL. The fission isotopes (FISSION_ISOTOPE) and fission products (FISSION_PRODUCT) in the variable have to originally exist in the fission yield library.

19.17.5.3 Number density variable

The **`'variable'`** template is DENSITY|MATERIAL_ID|ISOTOPE. The Material ID (MATERIAL_ID) has to originally exist in the material XML input. The isotope in the variable has to be originally defined within the material ID aforementioned.

19.17.5.4 Fission Q-values variables

The **`'variable'`** template is QVALUES| ISOTOPE.
The isotope (ISOTOPE) in the variable has to originally exist in the fission Q-values library.

19.17.5.5 α decay variable

The **'variable'** template is ALPHADECAY|ISOTOPE.
The isotope (ISOTOPE) in the variable has to originally exist in the α decay Q-values library.

19.17.5.6 β^+ decay variable

The **'variable'** template is BETA+DECAY| ISOTOPE.
The isotope (ISOTOPE) in the variable has to originally exist in the β^+ decay Q-values library.

19.17.5.7 β^{+*} decay variable

The **'variable'** template is BETA+XDECAY|ISOTOPE.
The isotope (ISOTOPE) in the variable has to originally exist in the β^{+*} decay Q-values library.

19.17.5.8 β decay variable

The **'variable'** template is BETADECAY|ISOTOPE.
The isotope (ISOTOPE) in the variable has to originally exist in the β decay Q-values library.

19.17.5.9 β^* decay variable

The **'variable'** template is BETAXDECAY|ISOTOPE.
The isotope (ISOTOPE) in the variable has to originally exist in the β^* decay Q-values library.

19.17.5.10 Internal transition decay variable

The **'variable'** template is INTTRADECAY|ISOTOPE.
The isotope (ISOTOPE) in the variable has originally to exist in the internal transition decay Q-value library.

19.17.5.11 Cross section scaling factors

The **'variable'** template is:
XS|TABULATION_POINT|MATERIAL_ID|ISOTOPE|OPERATOR|XS_TYPE|GROUP_NUMBER.

- The tabulation point (TABULATION_POINT) is the integer referrencing to the tabulation point. The tabulation numbering is given by the XML input file 'tabMapping.xml' (Section **??**). If there are no tabulations, the tabulation number has to be 1.
- The material ID (MATERIAL_ID) is the string referring to the material in which the isotope is defined. The Material ID has to originally exist in the material XML file.
- The isotope (ISOTOPE) is the ISOTOPE that the user desires to perturbed. The isotope perturbed has to originally exist in the material ID.

- The operator (OPERATOR) determines how the cross section is perturbed from its nominal value. The operators available are:
  - ADDITIVE; The additive operator adds the user-defined value to the nominal cross section value.
  - MULTIPLIER; the multiplier operator multiplies the user-defined factor to the nominal value.
  - ABSOLUTE; the absolute operator replaces the nominal value by the user-defined value.
- The types of cross sections (CROSS_SECTION_TYPE) available are:
  - FISSIONXS; Fission cross section.
  - NPXS; Neutron to proton capture cross section.
  - NGXS; Neutron to gamma capture cross section.
  - NUFISSIONXS; ν*fission cross section. The ν*fission is coordinated with the fission cross section so that only the coefficient ν is perturbed.
  - SCATTERINGXS. Total scattering cross section.
  - N2NXS. n,2n cross section.
  - NALPHAXS. Neutron to alpha capture cross section.
  - KAPPAXS. Kappa coefficient.

1. The group number (GROUP_NUMBER) is the group number of the cross section perturbed. The group number is an integer and has to be inferior or equal to the number of groups used in the cross section library.

An example is given for each type of variable in Table 27.
The following example is a Monte Carlo-based sampler with two variables. The β decay constant of uranium ^{235}U and the ^{135}Xe n,p cross section in group 12 at tabulation point 1 within the material ID "F1" are to be perturbed. The absolute operator is chosen for the n,p cross section, which means the nominal value will be replaced by the averaged value defined in the distribution block, with its corresponding distribution.

```
<Samplers>
  <MonteCarlo name="MC_samp">
    <samplerInit>
      <limit>100</limit>
    </samplerInit>
    <variable name="DECAY|BETA|U235">
      <distribution>DECAY|BETA|U235_dis</distribution>
    </variable>
    <variable name="XS|1|FUEL1|XE135|ABSOLUTE|NPXS|12">
```

Table 27: Examples of possible variable names

Input file perturbed	*Variable perturbed*	*Example*
Decay lib.	Decay constant	DECAY|BETAX|SE78
Fission yield lib.	Fission yield	FY|FAST|U235|NB93
XML Material	Number densitiy	DENSITY|FUEL1|PU239
Fission Q-values lib.	Fission Q-value	QVALUES|U235
α decay lib.	α decay Q-value	ALPHADECAY|U234
β^+ decay lib.	β^+ decay Q-value	BETA+DECAY|U235
β^{+*} decay lib.	β^{+*} decay Q-value	BETA+XDECAY|U236
β decay lib.	β decay Q-value	BETADECAY|CM242
β^* decay lib.	β^* decay Q-value	BETAXDECAY|PU240
Internal transition lib.	Internal transition Q-value	INTTRADECAY|U238
XS scaling factors	XS scaling factor	XS|1|FUEL1|U238|ABSOLUTE|N2NXS|4

```
        <distribution>XS|1|FUEL1|XE135|ABSOLUTE|NPXS|12_dis
          ↪                          </distribution>
      </variable>
    </MonteCarlo>
  </Samplers>
```

19.17.6 Steps

For a PHISICS interface, the **<MultiRun>** step type will most likely be used. First, the step needs to be named: this name will be one of the names used in the **<Sequence>** block. In the example, the step is called **'testDummyStep'**.

```
<Steps>
  <MultiRun name='testDummyStep' verbosity='debug'>
    <Input   class='Files' type='decay'>decay.dat</Input>
    <Input   class='Files' type='inp'>inp.xml</Input>
    <Input   class='Files' type='XS'>xs.xml</Input>
    <Model   class='Models' type='Code'>PHISICS</Model>
    <Sampler class='Samplers' type='MonteCarlo'>MC_samp</Sampler>
    <Output  class='Databases' type='HDF5'>DataB_REL5_1</Ouput>
  </MultiRun>
</Steps>
```

19.17.7 Additional Input

In addition to the usual PHISICS inputs required (INSTANT input, depletion input, material input, library input, path input) and the regular MrTau libraries, additional inputs may be required, depending on the user's needs.

- A file 'tabMapping.xml' is (optional) maps the cross section tabulation points for interpolation purposes. The tabulation mapping assigns an integer to a given tabulation in order to identify it in the RAVEN variable definition. The format of the tabulation mapping is the following:

```
<mapping>
  <tabulation set="1">
    <tab name="mod_temperature">559.0</tab>
    <tab name="BURN-UP">0.0</tab>
  </tabulation>
  <tabulation set="2">
    <tab name="mod_temperature">1000</tab>
    <tab name="BURN-UP">0.0</tab>
  </tabulation>
  <tabulation set="3">
    <tab name="mod_temperature">1000</tab>
    <tab name="BURN-UP">100</tab>
  </tabulation>
</mapping>
```

  In **<tabulation>**, the **set** refers to the user-defined number assigned to a given tabulation point. This **set** number corresponds to the second argument of the cross section scaling factor variable. The user enters the tabulation parameters and corresponding tabulation values with the **<tabulation>**, using the sub-node **<tab>**.

- If the user perturbs cross sections, an XML file 'scaled_xs.xml' will be generated at each perturbation in the output folder. The file 'scaled_xs.xml' is automatically created from the cross section variables defined in the RAVEN **<Sampler>** block (see 19.17.5.11). The cross section file 'scaled_xs.xml' has the following format:

```
<scaling_library>
  <tabulation>
    <tab name="mod_temperature">559.0</tab>
    <tab name="BURN-UP">0.0</tab>
    <library lib_name="fuel1" >
      <isotope id="xe135" type="absolute">
        <npxs g="8,3,12">8.889E+02,3.333E+02,1.212E+05</npxs>
      </isotope>
      <isotope id="u235" type="multiplier">
        <fissionxs g="1">2.019E-02</fissionxs>
      </isotope>
    </library>
  </tabulation>
<scaling_library>
```

The tabulation points **<tab>** are optional have to agree with the tabulation points defined in the "tabMapping.xml" if they are provided. The **<library>** has one required attribute **lib_name** corresponding to one of the libraries listed in the PHISICS library input. The **<isotope>** provides the information related to an isotope included in the library aforementioned. The **id** gives the isotope ID (no dash allowed). The **type** specifies the type of operator used (**'additive'**, **'multiplier'** or **'absolute'**). The sub-node **<XS>** (where 'XS' is the perturb-able type of cross sections listed in section 19.17.5.11) provides the cross section information. The **g** attribute refers to the group numbers to be perturbed, separated by commas. The **'XS'** provides the scaling factors or the new cross section values.

19.17.8 Output Files Conversion

The PHISICS output available for RAVEN post-processing are described in this section. The PHISICS outputs are by convention separated by '|' if they are contained in a matrix form such as group-wise or region-wise values. In the PHISICS mode (i.e. **<mrtauStandalone>** is **'False'**) The variables available for RAVEN post-processing are:

- the MrTau time;
- the multiplication factor;
- the multiplication factor error;
- the spatial reaction rates (only if **<printSpatialRR>** is **'True'**);
- the spatial power (only if **<printSpatialRR>** is **'True'**);
- the spatial fluxes by region (only if **<printSpatialRR>** is **'True'**);
- the neutron fluxes by cell (only if **<printSpatialFlux>** is **'True'**);
- the neutron fluxes by material (only if **<printSpatialFlux>** is **'True'**);
- the total reaction rates (only if **<printSpatialRR>** is **'False'**);
- the decay heat (only if decay heat flag is on in the PHISICS input);
- the burnup;
- the cross section values (only for perturbed cross sections);
- the PHISICS cpu time.

In the MrTau standalone mode (i.e **<mrtauStandalone>** is **'True'**) The variables available for RAVEN post-processing are:

- the MRTAU time;

Table 28: template of the RAVEN output variables

Variable	*Variable template*	*Comment*
MrTau Time	timeMrTau	
Multiplication Factor	keff	
Multiplication Factor Error	errorKeff	
n2n Reaction Rate	n2n|gr7|reg4	only if **<printSpatialRR>** is **'True'**)
Power	power|gr7|reg4	only if **<printSpatialRR>** is **'True'**)
Absorption Reaction Rate	absorption|gr7|reg4	only if **<printSpatialRR>** is **'True'**)
Fission Reaction Rate	fission|gr7|reg4	only if **<printSpatialRR>** is **'True'**)
Neutron Flux	flux|gr7|reg4	only if **<printSpatialRR>** is **'True'**)
ν Fission Reaction Rate	neutron|gr7|reg4	only if **<printSpatialRR>** is **'True'**)
Neutron Flux by Cell	flux|cell2|gr7	only if **<printSpatialFlux>** is **'True'**)
Neutron Flux by Material	flux|mat3|gr4	only if **<printSpatialFlux>** is **'True'**)
Total n2n Reaction Rate	n2n|Total	only if **<printSpatialRR>** is **'False'**)
Total Power reaction Rate	power|Total	only if **<printSpatialRR>** is **'False'**)
Total Absorption Reaction Rate	absorption|Total	only if **<printSpatialRR>** is **'False'**)
Total Fission Reaction Rate	fission|Total	only if **<printSpatialRR>** is **'False'**)
Total Neutron Flux	flux|Total	only if **<printSpatialRR>** is **'False'**)
Total ν Fis. Reaction Rate	neutron|Total	only if **<printSpatialRR>** is **'False'**)
Decay Heat	decay|Fuel1|gr4	only if the decay heat flag is turned on in PHISICS
Cross sections	fuel1|xe135|npxs|8	only if cross sections are perturbed
PHISIC CPU time	cpuTime	

- the isotope number densities;
- the decay heat.

The variable template is provided in Table 28. In the table, the region number is taken equal to 4, the group number is taken equal to 7, the cell number equal to 2 and the material number equal to 3. Those values are only examples and can be adapted to the user's convenience.
ν is the average number of neutrons generated after fission. Note that the material number in the neutron flux by material corresponds to the material ID number in the PHISICS csv output, while the material string ID in the decay heat corresponds to the material name given by the user in the xml material file. Hence, the neutron flux by material will always have the format matX, where X is an integer. The decay heat material is user-defined in the xml material file via the attribute **id** of the **<mat>** node.

19.18 PHISICS/RELAP5 Interface

19.18.1 General Information

This section covers the input specification for running PHISICS/RELAP5 through RAVEN. This interface can be used to perturb the PHISICS and/or RELAP5 input files. This interface is strongly

Table 29: Example of RELAP5 type attributes in coupled PHISICS/RELAP5 mode

Type Attribute	*Corresponding RELAP5 input*	*Perturbable*
relapFluid	fluid properties	No
relapInp	Relap input File	yes
relapLicense	license for the RELAP5 executable	No

built around the PHISICS and RELAP5 standalone interfaces, hence this sections covers the additional cautions to take care of to run the coupled PHISICS/RELAP5 code. The user will find additional information regarding PHISICS in section 19.17 or RELAP5 in section 19.3.

19.18.2 Files

`<Files>` includes two attributes **`name`** and **`type`** entries, identically as other interfaces. It also includes two optional attributes **`perturbable`** and **`subDirectory`**.
The **`name`** attribute is a user-defined internal name for the file contained in the node. Default: None (required entry).
For the files parsed by the PHISICS interface or the PHISICS interface's parsers, some of the **`type`** attributes are hardcoded. The accepted PHISICS **`type`** attributes are given in Table 26 and are not repeated here. Additional information can be found in section 19.17.2. All the necessary RELAP5 input files need to have a **`type`** attribute starting with the string 'relap'. The necessary RELAP5 files for use in the coupled PHISICS/RELAP mode within RAVEN are given in Table 29. The files associated with a **`type`** that does not start with the string 'relap' will be treated by the PHISICS interface. The **`type`** attributes are case-incensitive.
Example of acceptable RELAP5 entries within PHISICS/RELAP5:

```
<File>
  <Input name="H2O"        type="relaph2o"
     ↪ perturbable="False">tph2o</Input>
  <Input name="H2"         type="relaph2"
     ↪ perturbable="False">tph2</Input>
  <Input name="inputDeck" type="relapInput"   perturbable="True"
     ↪ >inp.i</Input>
  <Input name="lic"        type="relapLicence"
     ↪ perturbable="False">license.bin</Input>
</File>
```

19.18.3 Models

The user has to provide the paths to executables for the sampled variables within the **`<Models>`** block.
The **`<Code>`** block will contain attributes **`name`** and **`subType`**. The **`name`** identifies the particular **`<Code>`** model within RAVEN, and **`subType`** specifies which code interface the model will use.

subType='PhisicsRelap5' is the class name currently used for PHISICS/RELAP5 coupled calculations.
The **<executable>** block contains the absolute or relative path (with respect to the current working directory) to PHISICS/RELAP5 that RAVEN will use to run the code. The additional nodes in the **<Models>** applicable to PHISICS standalone and RELAP5 standalone are valid in coupled mode and can be consulted in section 19.17.3 and section 19.3.5 respectively. Exception: the use of MrTau in standalone mode (i.e. **<mrtauStandAlone>** set to **'True'**) is not allowed in PHISICS/RELAP5 coupled calculations.
An example of the **<Models>** block is given below:

```
<Models>
  <Code name="PHISICS_RELAP5" subType="PhisicsRelap5">
     <executable>./path/to/instant/executable</executable>
  </Code>
</Models>
```

19.18.4 Distributions

The **<Distributions>** block defines all distributions used to sample variables in the current RAVEN run.
For all the possible distributions and their possible inputs please refer to the Distributions chapter (see 9).

19.18.5 Samplers

The **<Samplers>** block defines the variables to be sampled. After defining a sampling scheme, the variables to be sampled and their distributions are identified in the **<variable>** blocks. The **name** must be formatted according to the PHISICS library which the variable belongs to. Information relative to PHISICS distributions are in section 19.17.5, as well as specifications on PHISICS variable names. An example of a **<Samplers>** block is given below:

```
<Samplers>
   <MonteCarlo name="MC_samp">
     <samplerInit>
       <limit>10</limit>
     </samplerInit>
     <variable name="DENSITY|FUEL1|U238">
       <distribution>DENSITY|FUEL1|U238_distrib</distribution>
     </variable>
     <variable name="20100154:2">
       <distribution>heat_capacity_154</distribution>
     </variable>
   </MonteCarlo>
 </Samplers>
```

In this example, the variable **'DENSITY|FUEL1|U238'** is relative to PHISICS and the variable **'20100154:2'** is relative to RELAP5.

19.18.6 Steps

The tasks performed by RAVEN need to be defined in the **<Steps>** block. Each task needs to be defined with a **name**. This **name** is later on used in the the **<Sequence>** block. In the example, the step is called **'testDummyStep'**.

```
<Steps>
  <MultiRun name='testDummyStep' verbosity='debug'>
    <Input class='Files' type='decay'>decay.dat</Input>
    <Input class='Files' type='inp'>inp.xml</Input>
    <Input class='Files' type='XS'>xs.xml</Input>
    <Input class="Files" type="relapFluid">tpfhe</Input>
    <Input class="Files" type="relapExec">relap5Exec.x</Input>
    <Model class="Models" type="Code">PHISICS_RELAP5</Model>
    <Sampler class="Samplers" type="MonteCarlo">MC_samp</Sampler>
    <Output class="Databases" type="HDF5">DataB_REL5_1</Output>
    <Output class="DataObjects" type="PointSet">collset</Output>
    </MultiRun>
  </MultiRun>
</Steps>
```

19.18.7 Additional Input

The PHISICS additional inputs are described in section 19.17.7. The RELAP5 additional inputs are described in section 19.3.5.

19.18.8 Output Files Conversion

The PHISICS output available for RAVEN post-processing are described in section 19.17.8. The output printed from PHISICS and RELAP5 are synchronized in the RAVEN csv output. The synchronization scheme is explained in this section.
At t = 0 seconds, the RELAP5 initialized output are printed in the csv output, while the output variables from PHISICS are taken equal to 0. Then, the RAVEN/PHISICS/RELAP5 post-processor finds the time step number at the end of each PHISICS burn step based on the **<tab_time_step>** values, and prints the RELAP minor edits according to the **<TH_between_BURN>** values.
Let's consider the following example:

- **<tab_time_step>** **'5 3 2'** **</tab_time_step>** (in the PHISICS depletion file);
- **<TH_between_BURN>** **'1.0 2.0'** **<TH_between_BURN>**() in the PHISICS input file);

– **<tabulation_boundaries>** **'5.0 35.0 45.0'** **<tabulation_boundaries>** (upper burn step boundaries in the PHISICS depletion file).

– in the RELAP5 input, a 3.0 seconds steady state is considered, with minor edits every 0.5 seconds.

The first PHISICS/RELAP5 output line printed will be at t = 0 seconds. The PHISICS outputs are set to 0.0, the RELAP5 values are obtained at the end of the initialization. The second line printed will be at the PHISICS time step **'5'** (end of the first burn step, corresponding to t = 5.0 seconds), and prints the RELAP5 minor edits as long as the time from the minor edits is lower than the first **<TH_between_BURN>** value **'1.0'**. The RELAP5 minor edits are printed along with the PHISICS burn step **'5'** as long as time in the minor edits is smaller or equal to **'1.0'**. When the RELAP5 time in the minor edits time is greater than **'1.0'**, the end of the second PHISICS burn step is targetted, from the **<tab_time_step>**: **'3'**. This corresponds to a PHISICS time equal to 35.0 seconds. The RELAP5 minor edits are printed along with the PHISICS values at t = 35.0 s, as long as the minor data time is smaller than the **<TH_between_BURN>** equal to **'2.0'**. Finally, the last PHISICS time step at 45.0 seconds is printed along with the RELAP5 minor edits. Overall, the output variables printed will be:
mrtauTime, n2n|gr1|reg4, httemp_3001010_1, time
0.000, 0.000, 1.526, 0.000
5.000, 1.111, 1.859, 0.500
5.000, 1.111, 2.369, 1.000
35.00, 7.800, 3.666, 1.500
35.00, 7.800, 4.789, 2.000
45.00, 9.330, 4.225, 3.000

19.19 Neutrino Interface

This section covers the input specification for running Neutrino through RAVEN. It is important to notice that this explanation assumes that the reader already knows how to use Neutrino. The existing inteface can be used to modify the particle size. However, the interface can be modified to alter other parameters by using a similar method to the existing particle size modification included in the interface.

19.19.1 Files

In the **<Files>** section, as specified before, all the files needed for the code to run should be specified. In the case of Neutrino, the file needed is the following:

- Neutrino input file with file extension '.nescene';

The Neutrino input file name must be NeutrinoInput.nescene. Otherwise, the Neutrino interface must be modified. Example:

```
<Files>
  <Input name="neutrinoInput"
     ↪ type="">NeutrinoInput.nescene</Input>
</Files>
```

19.19.2 Models

In the **<Models>** block, the Neutrino executable needs to be specified. The entire path to the Neutrino executable must be included. Here is a standard example of what can be used:

```
<Models>
  <Code name="neutrinoCode" subType="Neutrino">
    <executable>"C:\Program␣
       ↪ Files\Neutrino_02_22_19\Neutrino.exe"</executable>
  </Code>
</Models>
```

The **<Code>** XML node contains the information needed to execute the specific External Code. This XML node accepts the following attributes:

- **name**, ***required string attribute***, user-defined identifier of this model. **Note:** As with other objects, this identifier can be used to reference this specific entity from other input blocks in the XML.
- **subType**, ***required string attribute***, specifies the code that needs to be associated to this Model.

This model can be initialized with the following children:

- **<executable>**, ***string, required field***, specifies the path of the executable to be used; **Note:** Either an absolute or relative path can be used.

19.19.3 Distributions

The **<Distributions>** block defines the distributions that are going to be used for the sampling of the variables defined in the **<Samplers>** block. For all the possible distributions and all their possible inputs, please see the chapter about Distributions (see 9). Here is an example of a Uniform distribution:

```
<Distributions>
  <Uniform name="uni">
      <lowerBound>0.1</lowerBound>
      <upperBound>0.2</upperBound>
  </Uniform>
</Distributions>
```

19.19.4 Samplers

The **<Samplers>** block defines the variables to be sampled. After defining a sampling scheme, the variables to be sampled and their distributions are identified in the **<variable>** blocks. The **name** must be formatted according to the Neutrino parameter name, which for the particle size is 'ParticleSize'. An example of a **<Samplers>** block is given below:

```
<Samplers>
   <MonteCarlo name="myMC">
     <samplerInit>
       <limit>5</limit>
     </samplerInit>
     <variable name='ParticleSize'>
       <distribution>uni</distribution>
     </variable>
   </MonteCarlo>
 </Samplers>
```

19.19.5 Steps

In this section, the **<MultiRun>** will be used. As shown in the following, a Neutrino input file is listed in **<Files>** and is linked here using **<Input>**, the **<Model>** and **<Sampler>** defined in previous sections will be used in this **<MultiRun>**. The outputs will be saved in the **DataObject** 'resultPointSet'.

```
  <Steps>
    <MultiRun name="run">
      <Input class="Files" type="">neutrinoInput</Input>
      <Model class="Models" type="Code">neutrinoCode</Model>
      <Sampler class="Samplers" type="MonteCarlo">myMC</Sampler>
      <Output class="DataObjects"
         ↪ type="PointSet">resultPointSet</Output>
    </MultiRun>
  </Steps>
```

19.19.6 Output File Conversion

The Neutrino measurement field output is a CSV output. However, labels must be added to the Neutrino output and it must be moved for RAVEN. These are both done in the Neutrino interface. The labels that are added to the output file are 'time' and 'result'. These labels would be used in the **<DataObjects>** specification. If different labels are wanted, they would need to be changed directly in the Neutrino interface.

19.19.7 Additional Information

The Neutrino interface is used to alter the particle size by modifying the Neutrino input file. The Neutrino interface searches for the default SPH solver parameter name: 'NIISphSolver_1'. If the SPH solver name is changed in the Neutrino input file, the Neutrino interface must also be changed. Similarly, the Neutrino interface searches for the output in the default Measurement field name: 'MeasurementField_1'. Again, this would need to modified in the interface if the measurement field name was changed.

19.20 Prescient Interface

19.20.1 General Information

The Prescient Interface is used to run the open source Prescient production cost modeling platform available from https://github.com/grid-parity-exchange/Prescient
This allows inputs to be perturbed and data to be read out.

19.20.2 Sampler

For perturbing inputs, the sampled variable needs to be placed inside of $( )$ like (var). The sampled variable can have a constant added or a multiplication factor like $(var+3.2)$ or $(var*2.1)$ or $(var*5.0+7.0)$ or $(a*-2.0)$ These can be placed in any of the .dat or .csv files that are listed in the **<Files>** section as **type="PrescientInput"** An example line could be: Abel 1 (var)

```
<Samplers>
  <Grid name="grid">
    <variable name="var">
      <distribution>dist</distribution>
      <grid construction="equal" steps="1" type="CDF">0.0
         ↪ 1.0</grid>
    </variable>
  </Grid>
</Samplers>
```

19.20.3 Files

There are two types of inputs in the **<Files>** section. The **type="PrescientRunnerInput"** ones are passed as an argument to the runner.py If multiple PrescientRunnerInput files are specified then runner.py will be called multiple times (which can be used to run a populate and then simulate command). The **type="PrescientInput"** are just used as additional inputs that have the data in them perturbed.

```
<Files>
<Input name="simulate" type="PrescientRunnerInput"
>simulate_day.txt</Input>
  <Input name="structure" type="PrescientInput"
  subDirectory="scenarios/pyspdir_twostage/2020-07-10/"
  >ScenarioStructure.dat</Input>
  <Input name="scenario_1" type="PrescientInput"
  subDirectory="scenarios/pyspdir_twostage/2020-07-10/"
  >Scenario_1.dat</Input>
  <Input name="actuals" type="PrescientInput"
  subDirectory="scenarios/pyspdir_twostage/2020-07-10/"
  >Scenario_actuals.dat</Input>
  <Input name="forcasts" type="PrescientInput"
  subDirectory="scenarios/pyspdir_twostage/2020-07-10/"
  >Scenario_forecasts.dat</Input>
  <Input name="scenarios" type="PrescientInput"
  subDirectory="scenarios/pyspdir_twostage/2020-07-10/"
  >scenarios.csv</Input>
</Files>
```

19.20.4 Models

The **<Code>** model can be used with the **subType="Prescient"** to run the Prescient Code Interface. The block currently does not have any option xml nodes.

```
<Models>
  <Code name="TestPrescient" subType="Prescient">
    <executable>
    </executable>
  </Code>
</Models>
```

19.20.5 Output Files Conversion

The code interface reads in the `hourly_summary.csv` and the `bus_detail.csv` files. It will generate a `Date_Hour` variable that can be used as the **<pivotParameter>** and is a string with the date and hour. It also generates an `Hour` variable that is the hour as an integer. From the hourly summary it will generate variables like TotalCosts and the other data that appears there. For each of the busses in the bus detail file it generates variables like `Clay_LMP` that can be used.
Exactly which variables will appear will vary depending on the Prescient input files, but typical ones include `TotalCosts`, `FixedCosts`, `VariableCosts`, `LoadShedding`, `OverGeneration`, `ReserveShortfall`, `RenewablesUsed`,

RenewablesCurtailment, Demand, Price, and NetDemand. Variables that can be included for a typical bus could include ones like Abel_LMP, Abel_LMP_DA, Abel_Shortfall, and Abel_Overgeneration.

```
<HistorySet name="samples">
  <Input>var</Input>
  <Output>Date_Hour, TotalCosts, FixedCosts, VariableCosts,
     ↪ LoadShedding, OverGeneration, ReserveShortfall,
     ↪ RenewablesUsed, RenewablesCurtailment, Demand,
     ↪ Price, NetDemand, Abel_LMP, Clay_LMP </Output>
  <options>
    <pivotParameter>Date_Hour</pivotParameter>
  </options>
</HistorySet>
```

19.20.6 Installation of Libraries

Installing Prescient so that RAVEN can run it requires that RAVEN and Prescient have a superset of the libraries that they use so that both can run. One way to set this up is to install RAVEN, and then source the conda load script and inside of the conda raven libraries environment do the Prescient and Egret install. This is shown in the following listing:

```
#first clone raven, Egret and Prescient into a directory
git clone git@github.com:idaholab/raven.git
git clone git@github.com:grid-parity-exchange/Prescient.git
git clone git@github.com:grid-parity-exchange/Egret.git
#Switch to raven directory
cd raven
#install raven libraries
./scripts/establish_conda_env.sh --install
#switch to using raven libraries
source ./scripts/establish_conda_env.sh --load
#Switch to Prescient and install
cd ../Prescient
python setup.py develop --user
conda install -c conda-forge coincbc
#Switch to Egret and install
cd ../Egret/
pip install --user -e .
```

Note that the path to runner.py may need to be added to the PATH variable via a command like: PATH="$PATH:$HOME/.local/bin"

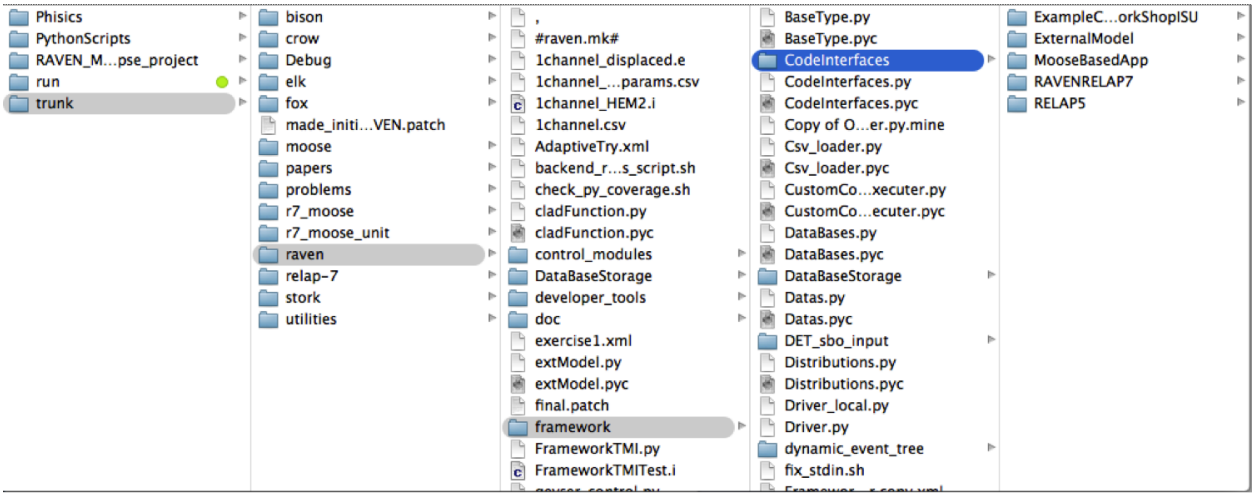

Figure 1: Code Interface Location.

20 Advanced Users: How to couple a new code

The procedure of coupling a new code/application with RAVEN is a straightforward process. For all the codes currently supported by RAVEN (e.g. RELAP-7, RELAP5-3D, BISON, MOOSE,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 coupling procedure does not require modifying RAVEN itself. Instead, the developer creates a new Python interface that is going to be embedded in RAVEN at run-time (no need to introduce hard-coded coupling statements). This interface needs to be placed in a folder (whatever name) located in (see figure 1):

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

At the initialization stage, RAVEN imports all the Interfaces that are contained in this directory and performs some preliminary cross-checks.
It is important to notice that the name of class in the Interface module is the one the user needs to specify when the new interface needs to be used. For example, if the Interface module contains the class “NewCode”, the *subType* in the **<Code>** block will be “NewCode”:

```
class NewCode(CodeInterfaceBase):
  ...
```

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

In the following sub-sections, a step-by-step procedure for coupling a code to RAVEN is outlined.

20.1 Pre-requisites.

In order to couple a newer application to the RAVEN code, some pre-requisites need to be satisfied.

Input

The first pre-requisite is the knowledge of the input syntax of the application the developer wants to couple. Indeed, RAVEN task "ends" at the Code Interface stage. RAVEN transfers the information needed to perturb the input space into the Code interface and expects that the newly developed Interface is able to perturb the input files based on the information passed through.
This means that the developer needs to code a Python-compatible parser of the system code input (a module that is able to read and modify the input of the code that needs to be coupled).
For example, let's suppose the input syntax of the code the developer needs to couple is as follows:

```
  kewword1 =  aValue1
  kewword2 =  aValue2
  kewword3 =  aValue3
  kewword4 =  aValue4
```

The Python input parser would be:

```
class simpleInputParser():
  def __init__(self,filename):
    #
    # @ In, string, filename, input file name (with path)
    #
    self.keywordDictionary = {}
    # open the file
    fileobject = open(filename)
    # store all the lines into a list
    lines = fileobject.readlines()
    # parse the list to construct
    # self.keywordDictionary dictionary
    for line in lines:
      # split the line with respect
      # to the symbol "=" and store the
      # outcomes into the dictionary
      # listSplitted[0] is the keword
      # listSplitted[1] is the value
      listSplitted = line.split("=")
      keyword = listSplitted[0]
      value   = listSplitted[1]
      self.keywordDictionary[keyword] = value
    # close the file
    fileobject.close()
```

```
def modifyInternalDictionary(self,inDictionary):
    #
    # @ In, dictionary {keyword:value},
    # inDictionary, dictionary containing
    # the keywords to perturb
    #

  # we just parse the dictionary and replace the
  # matching keywords
  for keyword,newvalue in inDictionary.items():
    self.keywordDictionary[keyword] = newvalue

def writeNewInput(self,filename):
  #
  # @ In, string, filename, newer input file name (with path)
  #

  # open the file
  fileobject = open(filename)
  # write line by line
  for keyword,newvalue in self.keywordDictionary.items():
    fileobject.write(keyword + ``='' + str(newvalue) + ``\n'')
  # close the file
  fileobject.close()
```

It is important to notice that for most of the codes, a wild-card approach can be used. In case this approach fits the user's needs, the RAVEN developer team suggests to inherit from the $GenericCode$ Interface (see section 19.1).

Output

RAVEN is able to handle Comma Separated Value (CSV) files (as outputs of the system code). In order make RAVEN able to retrieve the information from the newly coupled code, these files need to be either generated by the system code itself or the developer needs to code a Python-compatible module to convert the whatever code output format to a CSV one. This module can be directly called in the new code interface (see following section).

Let's suppose that the output format of the code (the same of the previous input parser example) is as follows:

```
result1 = aValue1
result2 = aValue2
result3 = aValue3
```

The Python output converter would be as simple as:

```
def convertOutputFileToCSV(outputfile):
    keywordDictionary = {}
    # open the original file
    fileobject = open(outputfile)
    outputCSVfile = open (outputfile + '.csv')
    # store all the lines into a list
    lines = fileobject.readlines()
    # parse the list to construct
    # self.keywordDictionary dictionary
    for line in lines:
      # split the line with respect
      # to the symbol "=" and store the
      # outcomes into the dictionary
      # listSplitted[0] is the keword
      # listSplitted[1] is the value
      listSplitted = line.split("=")
      keyword = listSplitted[0]
      value   = listSplitted[1]
      keywordDictionary[keyword] = value
    outputCSVfile.write(','.join(keywordDictionary.keys()))
    outputCSVfile.write(','.join(keywordDictionary.values()))
    outputCSVfile.close()
```

And the output CSV becomes:

```
  result1, result2, result3
  aValue1, aValue2, aValue3
```

Note that in general RAVEN is content with accepting floats or strings as data types in the CSV. However, if the CSV produced by running the code has a large number of columns (say, over 1000), it is necessary to include only floats and change the CSV loading utility. See more below (20.2.8)

20.2 Code Interface Creation

As already mentioned, RAVEN imports all the "Code Interfaces" at run-time, without actually knowing the syntax of the driven codes. In order to make RAVEN able to drive a newer software, the developer needs to code a Python module that will contain few methods (with strict syntax) that are called by RAVEN during the simulation.
When loading a "Code Interface", RAVEN expects to find, in the class representing the code, the following required methods:

```
from CodeInterfaceBaseClass import CodeInterfaceBase
class NewCode(CodeInterfaceBase):
```

```
  def generateCommand(self, inputFiles, executable, clargs=None,
     ↪ fargs=None, preExec=None)
  def createNewInput(self, currentInputFiles, oriInputFiles,
                                  samplerType, **Kwargs)
```

In addition, the following optional methods can be specified:

```
from CodeInterfaceBaseClass import CodeInterfaceBase
class NewCode(CodeInterfaceBase):
  ...
  def initialize(self, runInfoDict, oriInputFiles)
  def finalizeCodeOutput(self, command, output, workingDir)
  def getInputExtension(self)
  def checkForOutputFailure(self, output, workingDir)
```

In the following sub-sections all the methods are fully explained, providing examples (referring to the simple code used as example for the previous sections)

20.2.1 Method: `generateCommand`

```
def generateCommand(self, inputFiles, executable, clargs=None,
   ↪ fargs=None, preExec=None)
```

The **generateCommand** method is used to generate the commands (in `string` format) needed to launch the driven Code, as well as the root name of the output of the perturbed inputs (in `string` format). The return for this command is a two-part `Python` tuple. The first entry is a list of two-part tuples, each which specifies whether the corresponding command should be run exclusively in serial, or whether it can be run in parallel, as well as the command itself. For example, for a command where two successive commands are called, the first in serial and the second in parallel,

```
  def generateCommand(self,inputFiles,executable,clargs=None,
     ↪ fargs=None, preExec=None):
    . . .
    commands = [('serial',first_command), ('parallel',
       ↪ second_command)]
    return (commmands,outFileRoot)
```

For each command, the second entry in the tuple is a string containing the full command that the internal JobHandler is going to use to run the Code this interface refers to. The return data type must be a Python `tuple` with a list of tuples and a string: (`commands, outfileRoot`). Note that in most cases, only a single command needs to be run, so only a single command tuple is necessary. At run time, RAVEN will string together commands attached by double ampersands (&&), and each command labeled as parallel-compatible will be prepended with appropriate mpi arguments. For the example above, the command executed will be (with **`<NumMPI>`** equal to 4)

```
$  first_command && mpiexec -n 4 second_command
```

RAVEN is going to call the `generateCommand` function passing in the following arguments:

- **`inputFiles`**, data type = list: List of input files (length of the list depends on the number of inputs listed in the Step which is running this code);
- **`executable`**, data type = string, executable name with absolute path (e.g. /home/path_to_executable/code.exe);
- **`clargs`**, *optional*, data type = dictionary, a dictionary containing the command-line flags the user can specify in the input (e.g. under the node $< Code >< clargstype =' input' arg =' -i' extension =' .inp'/ >< /Code >$).
- **`fargs`**, *optional*, data type = dictionary, a dictionary containing the axuiliary input file variables the user can specify in the input (e.g. under the node $< Code >< clargstype =' input' arg =' aux' extension =' .aux'/ >< /Code >$).
- **`preExec`**, *optional*, data type = string, a string the command that needs to be pre-executed before the actual command. The user can specify in the input (e.g. under the node $< Code >< preexec > pre - executioncommand < /preexec >< /Code >$)
  Default: None

For the example referred to in the previous section, this method would be implemented as follows:

```
def generateCommand(self,inputFiles,executable,clargs=None,
    ↪ fargs=None, preExec=None):
  found = False
  for index, inputFile in enumerate(inputFiles):
    if inputFile.endswith(self.getInputExtension()):
      found = True
      break
  if not found: raise IOError(
        `None of the input files has one of the following
          ↪ extensions: ` +
        ` `.join(self.getInputExtension()))
  outputfile = 'out~'+os.path.split(inputFiles[index])[1].split
    ↪ ('.')[0]
  executeCommand = [('parallel',executable+ ` -i ` +os.path.
    ↪ split(inputFiles[index])[1])]
  return executeCommand,outputfile
```

20.2.2 Method: `createNewInput`

```
def createNewInput(self,currentInputFiles,oriInputFiles,
  ↪ samplerType,**Kwargs)
```

The **createNewInput** method is used to generate an input based on the information RAVEN passes in. In this function the developer needs to call the driven code input parser in order to modify the input file, accordingly with respect to the variables RAVEN is providing. This method needs to return a list containing the path and filenames of the modified input files. **Note:** RAVEN expects that at least one input file of the original list gets modified.
RAVEN is going to call this function passing in the following arguments:

- **`currentInputFiles`**, data type = list: List of current input files. This list of files is the one the code interface needs to use to print the new perturbed list of files. Indeed, RAVEN already changes the file location in sub-directories and the Code Interface does not need to change the filename or location of the files. For example, the files are going to have a absolute path as following: .
  $path_to_working_directory$
  $stepName$
  $anUniqueIdentifier$
  $filename.extension$. In case of sampling, the "*anUniqueIdentifier*" is going to be an integer (e.g. 1).

- **`oriInputFiles`** , data type = list, List of the original input files;

- **`samplerType`** , data type = string, Sampler type (e.g. MonteCarlo, Adaptive, etc.). **Note:** None if no Sampler has been used;

- **`Kwargs`** , data type = kwarded dictionary, dictionary of parameters. In this dictionary there is another dictionary called "SampledVars" where RAVEN stores the variables that got sampled (Kwargs['SampledVars'] = {'var1':10,'var2':40});

For the example referred in the previous section, this method would implemented as follows:

```
  def createNewInput(self,currentInputFiles,
        oriInputFiles,samplerType,**Kwargs):
    for index, inputFile in enumerate(oriInputFiles):
      if inputFile.endswith(self.getInputExtension()):
        break
    parser = simpleInputParser(currentInputFiles[index])
    parser.modifyInternalDictionary(**Kwargs['SampledVars'])
    parser.writeNewInput(newInputFiles[index])
    return newInputFiles
```

20.2.3 Method: `getInputExtension`

```
def getInputExtension(self)
```

The **getInputExtension** function is an optional method. If present, it is called by RAVEN code at run time. This function can be considered an utility method, since its main goal is to return a tuple of strings, where the developer can place all the input extensions the code interface needs to support (i.e. the extensions of the input(s) the code interface is going to "perturb"). If this method is not implemented, the default extensions are **(".i", ".inp", ".in"')**. This function does not accept any input argument. For the example referred in the previous section, this method would implemented as follows:

```
def getInputExtension(self):
    return (".i",".input")
```

20.2.4 Method: `initialize`

```
def initialize(self, runInfoDict, oriInputFiles)
```

The **initialize** function is an optional method. If present, it is called by RAVEN code at the begin of each Step (once per step) involving the particular Code Interface. This method is generally indicated to retrieve information from the RunInfo and/or the Input files.
RAVEN is going to call this function passing in the following arguments:

- **`runInfoDict`**, data type = dictionary: dictionary of the info stored in the run info XML block;
- **`oriInputFiles`**, data type = list, list of the original input files.

20.2.5 Method: `finalizeCodeOutput`

```
def finalizeCodeOutput(self, command, output, workingDir)
```

The **finalizeCodeOutput** function is an optional method. If present, it is called by RAVEN code at the end of each run. It can be used for those codes, that do not create CSV files as output to convert the whatever output format into a CSV. RAVEN checks if a string is returned; if so, RAVEN interprets that string as the new output file name (CSV).
RAVEN is going to call this function passing in the following arguments:

- **`command`**, data type = string: the command used to run the just ended job;
- **`output`**, data type = string, the Output name root;
- **`workingDir`**, data type = string, current working directory.

For the example referred in the previous section, this method would implemented as follows:

```
def finalizeCodeOutput(self, command, output, workingDir):
    outfile = os.path.join(workingDir,output+".o")
    convertOutputFileToCSV(outfile)
```

20.2.6 Method: `checkForOutputFailure`

```
def checkForOutputFailure(self, output, workingDir)
```

The **checkForOutputFailure** function is an optional method. If present, it is called by RAVEN code at the end of each run. This method needs to be implemented by the codes that, if a run fails, return a "returncode" = 0. This can happen in those codes that record the failure of a run (e.g. not converged, etc.) as normal termination (returncode == 0) This method can be used, for example, to parse the outputfile looking for a special keyword that testifies that a particular job failed (e.g. in RELAP5 would be the keyword "********"). This method MUST return a boolean (True if failed, False otherwise).
RAVEN is going to call this function passing in the following arguments:

- **`output`**, data type = string,the Output name root;
- **`workingDir`**, data type = string, current working directory.

For the example referred in the previous section, this method would implemented as follows:

```
def checkForOutputFailure(self, command, output, workingDir):
    from  __builtin__ import any
    errorWord = "ERROR"
    return any(errorWord in x for x in
      open(os.path.join(workingDir,output+'.o'),"r").readlines())
```

20.2.7 Method: `setRunOnShell`

```
self.setRunOnShell(shell=True)
```

The **setRunOnShell** function is an optional method. The default for shell is "True". In RAVEN, the *subprocess* module from Python is used to spawn new processes, connect to their input/output/error pipes, and obtain their return codes. To support a wide variety of use cases, the *Popen* constructor from *subprocess* is used to accept a large number of optional arguments. For most typical use cases, RAVEN will set these arguments automatically, and the code interface developers should not worry about the values for these arguments. However, in some specific use cases, the following argument may need to be setted by the code interface developers:

- shell, the default value is **True**. If shell is **True**, the specified command generated by RAVEN will be executed through the shell. This will allow RAVEN to have an enhanced control flow with convenient access to other shell features such as shell pipes, filename wildcards, environment variable expansion, and expansion of " " to a user's home directory. If shell is **False**, all the shell based features are disabled. In other words, the users could not use the shell features in the code inteface to generate the commands that are needed to lauch the driven code, i.e. the **generateCommand** method mentioned before should not use any shell features when constructs the commands. For more detailed description, please refer to the Python subprocess page `https://docs.python.org/2/library/subprocess.html` **Note:** If external codes can not run through "Shell", the code interface developers should call this function with "shell=False" in the "__init__" method. For example:

```
    def __init__(self):
      self.setRunOnShell(shell=False)
```

20.2.8 Method: `setCsvLoadUtil`

```
self.setCsvLoadUtil('pandas')
```

The default CSV loader in RAVEN is pandas, which allows arbitrary data types in the CSV, generally strings and floats. However, arbitrary data can be challenging to load if there are a large number of columns in the code's output CSV that RAVEN attempts to read in. As a rule of thumb, if there are over 1000 columns in a typical output CSV for your Code, the resulting values should only be floats and integers (not strings), and this method should be called during the CodeInterface construction or initialization to set the loading utility to `numpy`. While RAVEN's `numpy` CSV loading is notably faster than RAVEN's `pandas` CSV loading, it does not allow the flexibility of string entries except in the CSV header.

20.3 Tools for Developing Code Interfaces

To make generating a code interface as simple as possible, there are several tools RAVEN makes available within the Code Interface objects.

20.3.1 File Objects

RAVEN has created a wrapper for files within Python in order to carry along some additional information. This allows the user to tag particular files for reference in the Code Interface, using the **`type`** XML attribute in **`<Files>`** nodes. To differentiate, RAVEN file objects will use the capital Files, whereas typical files will use the lowercase files.
When the Files are passed in to `createNewInput`, they are passed in as Files objects. To access the xmlAttrtype of a file, use the method `getType`. For instance, instead of looking for an extension, a Code Interface might identify an input file by looking for a particular type, as shown in the example below. **Note:** RAVEN does not access a File's **`type`**; it is exclusively an optional tool for Code Interface developers.

```
found = False
for inFile in inputFiles:
  if inFile.getType()=='mainInput':
    found = True
    break
if not found:
  raise IOError('Desired file with type ``mainInput'' not found!'
    ↪ )
```

Using Files **type** attributes can especially help when multiple input files have the same extension. For example, say a Code execution command normally has the following appearance on the command line:

```
/home/path/to/executable/myexec.sh -i mainInp.xml -a auxInp.xml
  ↪ --mesh cube.e
```

The **<Files>** block in the RAVEN XML might appear as follows:

```
<Files>
  <Input name='main' type='base'>mainInp.xml</Input>
  <Input name='two'  type='aux' >auxInp.xml</Input>
  <Input name='cube' type='mesh' perturbable='False'>cube.e</
    ↪ Input>
</Files>
```

The search for these files in the Code Interface might then look like the example below, assuming one file per type:

```
# populate a type dictionary
typesDict={}
for inFile in inputFiles:
  typesDict[inFile.getType()]=inFile
# check all the necessary files are there
if 'base' not in typesDict.keys():
  raise IOError('File type ``base'' not listed in input file!')
if 'aux' not in typesDict.keys():
  raise IOError('File type ``aux'' not listed in input file!')
if 'mesh' not in typesDict.keys():
  raise IOError('File type ``mesh'' not listed in input file!')
mainFile = typesDict['base']
# do operations on file, etc.
```

Additionally, a Code Interface developer can access the **perturbable** through the `getPerturbable()` method of a Files object. This can be useful, for example, in preventing searching binary files for variable names when creating new input. For example,

```
for inFile in inputFiles:
  if not inFile.getPerturbable(): continue
  # etc
```

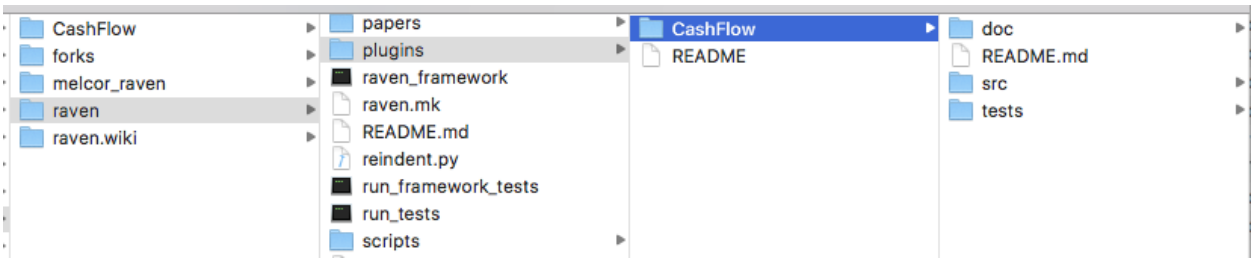

Figure 2: Plugins Location

21 Advanced Users: How to create a RAVEN ExternalModel plugin

The procedure of adding a plugin for the ExternalModel is a straightforward process. The addition of a plugin does not require modifying RAVEN itself. Instead, the developer creates a new Python module that is going to be embedded in RAVEN at run-time (no need to introduce hard-coded statements). This plugin needs to be placed in a folder (whatever name) located in (see figure 2):

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

In order to install a new plugin, the user can run the script contained in the RAVEN script folder:

```
python path/to/raven/scripts/install_plugins.py  **directory**
```

where $**directory**$ should be replaced with the absolute path to the plugin directory. (e.g. "path/to/my/plugins/folder"). If the plugin developer wants to make of his plugin an official supported plugin in RAVEN (by the submodule system), he needs to check the raven wiki under the "contribution" section).
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 "NewPlugin":

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

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

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

21.1 ExternalModel Plugin Input

When a new ExternalModel plugin is developed, its RAVEN input is almost identical to the general ExternalModel entity (see 15.4). 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 name of the new plugin the user wants to use (e.g. $NewPlugin$). **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 **`<variables>`** block:

- **`<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.

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

- **`<alias>`** ***string, optional field*** specifies alias for any variable of interest in the input or output space for the ExternalModel. These aliases can be used anywhere in the RAVEN input to refer to the ExternalModel variables. 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.
  Default: None

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)
```

21.2 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 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 ExternalModelPluginBase import ExternalModelPluginBase
class NewPlugin(ExternalModelPluginBase):
  def run (self, container, Inputs)
```

In addition, the following optional methods can be specified:

```
from 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.

21.2.1 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'']
```

21.2.2 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, thorough 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 12), 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']
```

21.2.3 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:

```
<Simulation>
  ...
  <Models>
     ...
    <ExternalModel name='AnExtModule' subType=''NewPlugin">
␣␣␣␣␣␣␣<variables>sigma,rho,outcome</variables>
␣␣␣␣␣␣␣<!--
␣␣␣␣␣␣␣␣␣␣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>
␣␣...
</Simulation>
```

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
```

21.2.4 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. For example, it can be used to compute a quantity needed by the "run" method before performing the actual calculation. If this method is implemented in the **ExternalModel** Plugin, RAVEN is going to call it at the initialization stage of each "Step" (see section 18). RAVEN will communicate, thorough 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.

As all the others method in the ExternalModel Plugin, the information *must* be stored in the "self-like" object "container". 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
```

22 Advanced Users: How and When to create a RAVEN Template

One of the great strengths of the RAVEN input is its flexibility; an enormous number of different types of workflows can be constructed with the components outlined in this manual. Sometimes, this flexibility is not required when standard or predefined workflows need to be employed changing just few settings. For example, in classical uncertainty quantification, sometimes only a few variables or the model needs to be changed, while the rest of the workflow stays the same.

As a tool to focus RAVEN on particular workflows, we introduce the RAVEN Templated Input Files. The intention of this system is to allow a single user to develop a template RAVEN input file along with a template interface, thereby simplifying inputs for any number of users that only need to make minor changes to the templated workflow in order to perform their analysis.

Note: A RAVEN Template is a wrapper for creating RAVEN input files; it is not part of the RAVEN core code and is usually s specific to a particular application.

22.1 When to use a RAVEN Template

By design, a RAVEN Template simplifies the user experience at the cost of flexibility. The amount of streamlining is adjustable and specific to each template. At one extreme, a Template takes no modifications at all and always produces the same workflow; at the other extreme the Template duplicates entirely the RAVEN input syntax. Neither of those options is desirable; Templates should find ground in-between.

There are some times where using a RAVEN Template can be highly beneficial:

- The workflow in question is highly complex, involving some advanced RAVEN usage to perform unorthodox calculations,
- The workflow is mostly the same for each user, requiring only a small number of changes to use repeatedly.

There are some times when using a RAVEN Template is unlikely to be useful:

- The workflow needs to be flexible enough to accommodate many unpredictable changes,
- The workflow has few entries and can be changed manually quite easily.

22.2 How to create a RAVEN Template

A RAVEN Template consists of three main pieces: a Templated Workflow, a Template Class, and a Template Interface. The Interface is the main driver, and uses an input file to inform the Template Class on how to modify the Templated Workflow in order to create a new RAVEN input file.

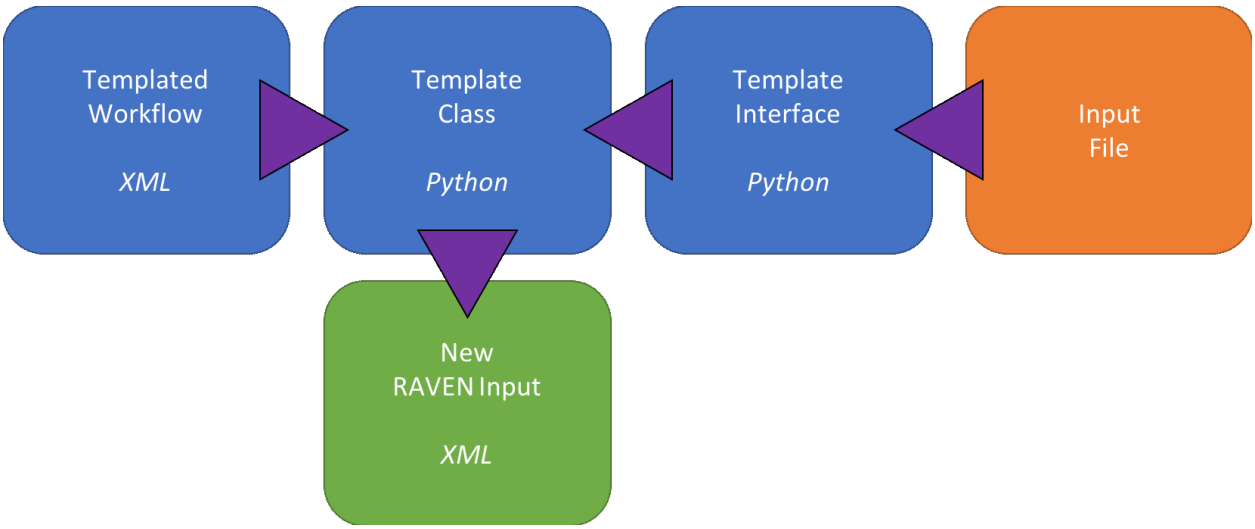

Figure 3: Information Flow for RAVEN Templates

Refer to Figure 3. Each box represents a file in a RAVEN template system. The three boxes in blue (Templated Workflow, Template Class, and Template Interface) are developed collectively as the Template by a designer familiar with RAVEN input files and Python. The development of this template only needs to occur once. The orange box (Input File) is in a format determined by the Template Interface, and is the only portion of the Template that a user will interact with repeatedly for any given workflow. The green box (New RAVEN Input) is the result of reading a particular input file with the Template and is written by the Template. This new input can either be run automatically with RAVEN or left to run at the user's convenience, based on what the Template Interface is designed to do. Note that running RAVEN from within a python script on Windows within MinGW is particularly tricky.

The three portions of the Template are discussed individually in the following sections.

22.2.1 Templated Workflows

A Templated Workflow begins with a traditional RAVEN input file that is run to do a particular analysis. It is highly recommended that this workflow is run with RAVEN and the inputs and outputs are well understood before beginning templating. Keep a copy of the original workflow before modifying the Templated Workflow.

Next, consider the parts of the workflow that are common to anyone who will want to perform a similar analysis, and which are specific to individual runs. For example, perhaps the **`<Sequence>`** and **`<Steps>`** are always the same, but the **`<Model>`** and sampled **`<variable>`** nodes may change for each analysis. Note those parts of the original workflow that need to be flexible, and remove them from the Templated Workflow. These will be filled in by the Template Class for this workflow when the Template Interface is run.

22.2.2 Template Class

The Template Class is a bridge between the template designer and the Templated Workflow. The Template Class knows every detail about the Templated Workflow and knows how to modify it to create a working RAVEN input. It does so through a set of standardized calls from the Template Interface.
A Template Base Class is provided in the RAVEN repository to be inherited by your new Template Class. It is located in

```
raven/framework/InputTemplates/TemplateBaseClass.py
```

We recommend you locate your new Template Class near your project where the workflows are run, and not in the RAVEN repository.
There are several required methods in the API of the Template Base Class that are important.

```
loadTemplate(self, filename, path)
```

The `loadTemplate` method is how the Template Class knows how to load the Template Workflow. The default implementation in the Template Base Class is probably sufficient for most Template Classes, where given the filename and path to the file, the template is loaded into `self._template`. Of course, this behavior can be modified however suits a project by overloading this method in the Template Class.

```
createWorkflow(self, **kwargs)
```

The `createWorkflow` method is the main method of the Template Class. The Template Interface calls this method when it wants to use a series of modifications to write a new RAVEN input file. Note the Template Base Class implementation of `createWorkflow` accepts arbitrary keyword arguments as `**kwargs`. This allows the inheriting Template Class to define its own required arguments necessary to write a new input file. These may be lists, dictionaries, or any other Python object. All of the necessary information for the Template Class to convert a Template Workflow into a valid RAVEN input file should be passed through these arguments.

The rest of `createWorkflow` is open to do any operations necessary to modify the XML in the Template Workflow until it becomes a valid RAVEN input that performs the desired analysis. RAVEN offers a plethora of handy XML tools in `raven/framework/xmlUtils`, which is imported in the base class and can be imported in your Template Class as well. In addition, the Python standard library has an excellent `xml.etree.ElementTree` package for manipulating XML. Note that any `createWorkflow` should start by deepcopying the template XML, to assure a clean copy is available each time it is called. The `createWorkflow` ends by returning the modified XML element.

```
writeWorkflow(self, template, destination, run=False)
```

Once `createWorkflow` is called, the resulting XML element can be supplied to the `writeWorkflow` method, which writes the XML to a file. The Template Base Class implementation will likely cover the needs of most Template Classes, and shouldn't require significant

modification. Note that an optional argument `run` instructs the Template Class to attempt to run the workflow in RAVEN once it is written to file. Note this currently works on Mac and Linux systems, but is not yet consistent on Windows.

Other optional methods also exist in the Template Base Class and may be of use to individual templates.

```
addNamingTemplates(cls, templates)
```

Note that the Template Base Class has a class-level dictionary member called `namingTemplates`. The intention of this method is to store common ways to name items in the RAVEN input in a format method so that later they are always consistent. To extend this method, call `BaseClass.addNamingTemplates` at the class level in the inheriting Template Class.

Finally, commonly-used shortcuts are included at the end of the Template Base Class to perform actions that are repetitively used in modifying RAVEN inputs. We recommend you add your own to your Template Class to help keep `createWorkflow` clean and easily maintainable.

22.2.3 Template Interface

The Template Interface is the code that actually gets called by users once the Template Class and Template Workflow are complete. In its simplest form, the Template Interface is a Python script that reads the data needed for the `createWorkflow` method and calls the methods on the Template Class in order:

1. `loadTemplate`
2. `createWorkflow`
3. `writeWorkflow`

Template Interfaces read a simplified input file so that users can provide their parameters for the Templated Workflow in an easy manner. Whatever enables the use of the RAVEN workflow with minimal effort on the part of the users is ideal for the Template Interface.

22.3 Example

For testing and as an example of implementation, a simple Template was created to perform basic uncertainty quantification (UQ) analysis on external models. The example can be found in

```
raven/tests/framework/TemplateInputs
```

The following files are part of this template under the directory given above:

- Templated Workflow: `TemplateInputs/UQTemplate/uq_template.xml`

- Template Class: `TemplateInputs/UQTemplate/UQTemplate.py`
- Template Interface: `TemplateInputs/uq_maker.py`
- Input File: `TemplateInputs/UQTemplate/uq_template_input.i`

The original workflow from which the Templated Workflow was created involved a simple Monte Carlo sampling of an external model and then postprocessing with BasicStatistics to find the mean, standard deviation, skewness, and kurtosis of the results. The Template designer determined that this workflow could be used for many similar analyses with only small changes, and decided to template it. The designer determined that things that could be changed include the model sampled, the outputs of the model, the inputs to the model (with their distributions), and the number of Monte Carlo samples to take. The designer also decided that each case should have its own WorkingDir to keep analyses separate. We will consider the resulting Template files that the designer wrote one at a time in the following sections.

22.3.1 Example Templated Workflow

The file `uq_template.xml` looks much like a typical RAVEN input file with some key pieces missing. The **`<Sequence>`** shows that the two steps are **`'sample'`** and **`'stats'`**, which are for sampling a model using Monte Carlo sampling and then performing some statistics on the results. Note however the missing contents in the **`<WorkingDir>`**, the empty nodes in the **`<DataObjects>`**, the lack of any **`<Distributions>`**, and the missing variable lists in the xmlNodePostProcessor. All the missing contents are filled in by the Template Class. For the results of the filled-in workflow, see in

```
raven/tests/framework/TemplateInputs/gold/UQTemplate/new_uq.xml
```

22.3.2 Example Template Class

The file `UQTemplate.py` demonstrates inheritance of the Template Base Class and customization for the logic to fill in the Templated Workflow.

Note that the Template Class adds three formatted strings to the class-level name templates, one each for step names, distributions, and metric variables. These are called later in the code to assure the naming conventions are always the same. These formatted strings employ Python's inherent string formatting tools.

The Template Base Class implementation of `loadWorfklow` does everything this Template needs to load the Templated Workflow, so there is no need to modify it in the custom Template Class. Similarly, the `writeWorkflows` does everything this Template needs to write a newly-created input, so there is no need to modify it in the custom Template Class. Since there was no need to overload the Template Base Class implementations of `loadWorfklow` and `writeWorkflows`, the only main method changed in the Template Class is the essential `createWorkflow` method. Note that we've added several keywords to the argument list:

```
def createWorkflow(self, model=None, variables=None, samples=
    ↪ None, case=None, **kwargs):
```

In order to correctly modify the Templated Workflow, the Template Class needs to know about what model is being sampled, the input variables to the model and how they're distributed, how many Monte Carlo samples to take, and the name of the case being run. It requires all of these to be provided by the Template Interface in order to write a new RAVEN input. In this case, the method arguments `model` and `variables` are dictionaries, while the `samples` are an integer and the `case` is a string.

Note that `UQTemplate.createWorkflow` calls the Template Base Class's implementation first in order to preserve inheritance. Since the deepcopy happens in the base class, we don't perform it again in the custom Template Class.

Throughout the remainder of the workflow creation, a series of XML manipulations are performed based on the inputs provided from the Template Interface. For example, the module to load for the Model is changed, the working directory is set, and the input and output variables are propagated throughout the input file. Note also that several input construction shortcut methods have been added for this particular template to simplify maintenance of the template.

22.3.3 Example Template Interface

The file `uq_maker.py` contains the basic logic needed to read a user input file, load the Template Class, and generate new inputs. It follows the sequence of events outlined above, first instructing the Template Class to load the template, then reading in the user input, then instructing the Template Class to create the workflow, then to write the workflow.

Beause the input needs for this Template are simple, we use Python's standard library `configparser` to read in the user input file, `uq_template_input.i`. This simple input structure uses sections (model, variables, and settings) with keyword and value pairs in each section. In order to change the RAVEN workflow created, the user only needs to make changes to the existing input file and run the interface, then run RAVEN on the new input.

Note that we chose to provide the information to the Template Class mostly through dictionaries, where the essential pieces of information can be provided. In particular note that the variables provide only a mean and standard deviation; one reduction in flexibility is that we assume the variables are normally distributed, disallowing other distributions.

The Template can be run with Python from the command line:

```
> python uq_maker.py
> cd UQTemplate
> ~/projects/raven/raven_framework new_uq.xml
```

It reads in the user input, modifies the template, writes the new input file, and finally runs RAVEN. Note that, if desired, the interface can be extended to perform additional operations after RAVEN has finished creating the workflow.

A Appendix: Example Primer

In this Appendix, a set of examples are reported. In order to be as general as possible, the *Model* type "ExternalModel" has been used.

A.1 Example 1.

This simple example is about the construction of a "Lorentz attractor", sampling the relative input space. The parameters that are sampled represent the initial coordinate (x0,y0,z0) of the attractor origin.

```
<?xml version="1.0" encoding="UTF-8"?>
<Simulation verbosity="debug">
<!-- RUNINFO -->
<RunInfo>
    <WorkingDir>externalModel</WorkingDir>
    <Sequence>FirstMRun</Sequence>
    <batchSize>3</batchSize>
</RunInfo>
<!-- Files -->
<Files>
    <Input name='lorentzAttractor.py'
       ↪ type=''>lorentzAttractor</Input>
</Files>
<!-- STEPS -->
<Steps>
    <MultiRun name='FirstMRun'  re-seeding='25061978'>
        <Input   class='Files'     type=''
           ↪ >lorentzAttractor.py</Input>
        <Model   class='Models'    type='ExternalModel'
           ↪ >PythonModule</Model>
        <Sampler class='Samplers'  type='MonteCarlo'
           ↪ >MC_external</Sampler>
        <Output  class='DataObjects'     type='HistorySet'
           ↪ >testPrintHistorySet</Output>
        <Output  class='Databases' type='HDF5'
           ↪ >test_external_db</Output>
        <Output  class='OutStreams' type='Print'
           ↪ >testPrintHistorySet_dump</Output>
    </MultiRun >
</Steps>
<!-- MODELS -->
<Models>
```

```
    <ExternalModel name='PythonModule' subType=''
       ↪ ModuleToLoad='externalModel/lorentzAttractor'>
       <variables>sigma,rho,beta,x,y,z,time,x0,y0,z0</variables>
    </ExternalModel>
</Models>
<!-- DISTRIBUTIONS -->
<Distributions>
    <Normal name='x0_distrib'>
        <mean>4</mean>
        <sigma>1</sigma>
    </Normal>
    <Normal name='y0_distrib'>
        <mean>4</mean>
        <sigma>1</sigma>
    </Normal>
    <Normal name='z0_distrib'>
        <mean>4</mean>
        <sigma>1</sigma>
    </Normal>
</Distributions>
<!-- SAMPLERS -->
<Samplers>
    <MonteCarlo name='MC_external'>
      <samplerInit>
        <limit>3</limit>
      </samplerInit>
      <variable name='x0' >
        <distribution  >x0_distrib</distribution>
      </variable>
      <variable name='y0' >
        <distribution  >y0_distrib</distribution>
      </variable>
      <variable name='z0' >
        <distribution  >z0_distrib</distribution>
      </variable>
    </MonteCarlo>
</Samplers>
<!-- DATABASES -->
<Databases>
  <HDF5 name="test_external_db"/>
</Databases>
<!-- OUTSTREAMS -->
```

```
<OutStreams>
  <Print name='testPrintHistorySet_dump'>
    <type>csv</type>
    <source>testPrintHistorySet</source>
  </Print>
</OutStreams>
<!-- DATA OBJECTS -->
<DataObjects>
    <HistorySet name='testPrintHistorySet'>
        <Input>x0,y0,z0</Input>
        <Output>time,x,y,z</Output>
   </HistorySet>
</DataObjects>
</Simulation>
```

The Python *ExternalModel* is reported below:

```
import numpy as np

def run(self,Input):
  max_time = 0.03
  t_step = 0.01

  numberTimeSteps = int(max_time/t_step)

  self.x = np.zeros(numberTimeSteps)
  self.y = np.zeros(numberTimeSteps)
  self.z = np.zeros(numberTimeSteps)
  self.time = np.zeros(numberTimeSteps)

  self.x0 = Input['x0']
  self.y0 = Input['y0']
  self.z0 = Input['z0']

  self.x[0] = Input['x0']
  self.y[0] = Input['y0']
  self.z[0] = Input['z0']
  self.time[0]= 0

  for t in range (numberTimeSteps-1):
    self.time[t+1] = self.time[t] + t_step
    self.x[t+1]    = self.x[t] +  self.sigma*
                      (self.y[t]-self.x[t]) * t_step
    self.y[t+1]    = self.y[t] + (self.x[t]*
```

```
                        (self.rho-self.z[t])-self.y[t]) * t_step
    self.z[t+1]     = self.z[t] + (self.x[t]*
                            self.y[t]-self.beta*self.z[t]) * t_step
```

A.2 Example 2.

This example shows a slightly more complicated example, that employs the usage of:

- *Samplers:* Grid and Adaptive;
- *Models:* External, Reduce Order Models and Post-Processors;
- *OutStreams:* Prints and Plots;
- *Data Objects:* PointSets;
- *Functions:* ExternalFunctions.

The goal of this input is to compute the "SafestPoint". It provides the coordinates of the farthest point from the limit surface that is given as an input. The safest point coordinates are expected values of the coordinates of the farthest points from the limit surface in the space of the "controllable" variables based on the probability distributions of the "non-controllable" variables.
The term "controllable" identifies those variables that are under control during the system operation, while the "non-controllable" variables are stochastic parameters affecting the system behaviour randomly.
The "SafestPoint" post-processor requires the set of points belonging to the limit surface, which must be given as an input.

```
<Simulation verbosity='debug'>

<!-- RUNINFO -->
<RunInfo>
  <WorkingDir>SafestPointPP</WorkingDir>
  <Sequence>pth1,pth2,pth3,pth4</Sequence>
  <batchSize>50</batchSize>
</RunInfo>

<!-- STEPS -->
<Steps>
  <MultiRun name = 'pth1' pauseAtEnd = 'False'>
    <Sampler  class = 'Samplers'  type = 'Grid'
      ↪ >grd_vl_ql_smp_dpt</Sampler>
    <Input    class = 'DataObjects'     type = 'PointSet'
      ↪ >grd_vl_ql_smp_dpt_dt</Input>
```

```
    <Model     class = 'Models'     type = 'ExternalModel'
       ↪ >xtr_mdl</Model>
    <Output    class = 'DataObjects'      type = 'PointSet'
       ↪ >nt_phy_dpt_dt</Output>
  </MultiRun >

  <MultiRun name = 'pth2' pauseAtEnd = 'True'>
    <Sampler           class = 'Samplers'  type = 'Adaptive'
       ↪ >dpt_smp</Sampler>
    <Input             class = 'DataObjects'      type =
       ↪ 'PointSet'  >bln_smp_dt</Input>
    <Model             class = 'Models'     type = 'ExternalModel'
       ↪ >xtr_mdl</Model>
    <Output            class = 'DataObjects'      type =
       ↪ 'PointSet'  >nt_phy_dpt_dt</Output>
    <SolutionExport    class = 'DataObjects'      type =
       ↪ 'PointSet'  >lmt_srf_dt</SolutionExport>
  </MultiRun>

  <PostProcess name='pth3' pauseAtEnd = 'False'>
    <Input     class = 'DataObjects'           type = 'PointSet'
       ↪        >lmt_srf_dt</Input>
    <Model     class = 'Models'          type = 'PostProcessor'
       ↪ >SP</Model>
    <Output    class = 'DataObjects'           type = 'PointSet'
       ↪      >sfs_pnt_dt</Output>
  </PostProcess>

  <OutStreamStep name = 'pth4' pauseAtEnd = 'True'>
        <Input  class = 'DataObjects'             type =
           ↪ 'PointSet'  >lmt_srf_dt</Input>
        <Output class = 'OutStreams' type = 'Print'
           ↪ >lmt_srf_dmp</Output>
    <Input  class = 'DataObjects'             type = 'PointSet'
       ↪ >sfs_pnt_dt</Input>
        <Output class = 'OutStreams' type = 'Print'
           ↪ >sfs_pnt_dmp</Output>
  </OutStreamStep>
</Steps>

<!-- DATA OBJECTS -->
<DataObjects>
```

```
  <PointSet name = 'grd_vl_ql_smp_dpt_dt'>
    <Input>x1,x2,gammay</Input>
    <Output>OutputPlaceHolder</Output>
  </PointSet>

  <PointSet name = 'nt_phy_dpt_dt'>
    <Input>x1,x2,gammay</Input>
    <Output>g</Output>
  </PointSet>

  <PointSet name = 'bln_smp_dt'>
    <Input>x1,x2,gammay</Input>
    <Output>OutputPlaceHolder</Output>
  </PointSet>

  <PointSet name = 'lmt_srf_dt'>
    <Input>x1,x2,gammay</Input>
    <Output>g_zr</Output>
  </PointSet>

  <PointSet name = 'sfs_pnt_dt'>
    <Input>x1,x2,gammay</Input>
    <Output>p</Output>
  </PointSet>
</DataObjects>

<!-- DISTRIBUTIONS -->
<Distributions>
  <Normal name = 'x1_dst'>
    <upperBound>10</upperBound>
    <lowerBound>-10</lowerBound>
        <mean>0.5</mean>
    <sigma>0.1</sigma>
  </Normal>

  <Normal name = 'x2_dst'>
    <upperBound>10</upperBound>
    <lowerBound>-10</lowerBound>
    <mean>-0.15</mean>
    <sigma>0.05</sigma>
  </Normal>
```

```
  <Normal name = 'gammay_dst'>
    <upperBound>20</upperBound>
    <lowerBound>-20</lowerBound>
    <mean>0</mean>
    <sigma>15</sigma>
  </Normal>
</Distributions>

<!-- SAMPLERS -->
<Samplers>
  <Grid name = 'grd_vl_ql_smp_dpt'>
    <variable name = 'x1' >
      <distribution>x1_dst</distribution>
      <grid type = 'value' construction = 'equal' steps = '10'
         ↪ upperBound = '10'>2</grid>
    </variable>
    <variable name='x2' >
      <distribution>x2_dst</distribution>
      <grid type = 'value' construction = 'equal' steps = '10'
         ↪ upperBound = '10'>2</grid>
    </variable>
    <variable name='gammay' >
      <distribution>gammay_dst</distribution>
      <grid type = 'value' construction = 'equal' steps = '10'
         ↪ lowerBound = '-20'>4</grid>
    </variable>
  </Grid>

  <Adaptive name = 'dpt_smp' verbosity='debug'>
    <ROM              class = 'Models'    type = 'ROM'
       ↪ >accelerated_ROM</ROM>
    <Function         class = 'Functions' type = 'External'
       ↪ >g_zr</Function>
    <TargetEvaluation class = 'DataObjects'     type =
       ↪ 'PointSet'  >nt_phy_dpt_dt</TargetEvaluation>
    <Convergence limit = '3000' forceIteration = 'False' weight
       ↪ = 'none' persistence = '5'>1e-2</Convergence>
      <variable name = 'x1'>
        <distribution>x1_dst</distribution>
      </variable>
      <variable name = 'x2'>
        <distribution>x2_dst</distribution>
```

```
      </variable>
      <variable name = 'gammay'>
        <distribution>gammay_dst</distribution>
      </variable>
  </Adaptive>
</Samplers>

<!-- MODELS -->
<Models>
  <ExternalModel name = 'xtr_mdl' subType = '' ModuleToLoad =
     ↪ 'SafestPointPP/safest_point_test_xtr_mdl'>
    <variables>x1,x2,gammay,g</variables>
  </ExternalModel>

  <ROM name = 'accelerated_ROM' subType = 'SciKitLearn'>
    <Features>x1,x2,gammay</Features>
    <Target>g_zr</Target>
    <SKLtype>svm|SVC</SKLtype>
    <kernel>rbf</kernel>
    <gamma>10</gamma>
    <tol>1e-5</tol>
    <C>50</C>
  </ROM>

  <PostProcessor name='SP' subType='SafestPoint'>
    <!-- List of Objects (external with respect to this PP)
       ↪ needed by this post-processor -->
    <Distribution     class = 'Distributions'  type =
       ↪ 'Normal'>x1_dst</Distribution>
    <Distribution     class = 'Distributions'  type =
       ↪ 'Normal'>x2_dst</Distribution>
    <Distribution     class = 'Distributions'  type =
       ↪ 'Normal'>gammay_dst</Distribution>
    <!- end of the list -->
    <controllable>
        <variable name = 'x1'>
                <distribution>x1_dst</distribution>
                <grid type = 'value' steps = '20'>1</grid>
        </variable>
        <variable name = 'x2'>
                <distribution>x2_dst</distribution>
                <grid type = 'value' steps = '20'>1</grid>
```

```
        </variable>
    </controllable>
    <non-controllable>
        <variable name = 'gammay'>
                <distribution>gammay_dst</distribution>
                <grid type = 'value' steps = '20'>2</grid>
        </variable>
    </non-controllable>
  </PostProcessor>
</Models>

<!-- FUNCTIONS -->
<Functions>
  <External name='g_zr'
     ↪ file='SafestPointPP/safest_point_test_g_zr.py'>
    <variable>g</variable>
  </External>
</Functions>

<!-- OUT-STREAMS -->
<OutStreams>
  <Print name = 'lmt_srf_dmp'>
        <type>csv</type>
        <source>lmt_srf_dt</source>
  </Print>

  <Print name = 'sfs_pnt_dmp'>
        <type>csv</type>
        <source>sfs_pnt_dt</source>
  </Print>
</OutStreams>

</Simulation>
```

The Python *ExternalModel* is reported below:

```
def run(self,Input):
  self.g = self.x1+4*self.x2-self.gammay
```

The "Goal Function",the function that defines the transitions with respect the input space coordinates, is as follows:

```
def __residuumSign(self):
  if self.g<0 : return  1
  else        : return -1
```

Document Version Information

tag_number_23-1193-gdb26e9baf
7a5c160535d4d5ffea28ce50f8414d519a335029 alfoa
Wed, 12 May 2021 22:39:09 -0600

References

[1] M. P. Forum, "Mpi: A message-passing interface standard," tech. rep., Knoxville, TN, USA, 1994.

[2] "Portable batch system." http://www.pbsworks.com.

[3] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing In Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[4] D. Cournapeau, "Scikit-learn library for machine learning." http://scikit-learn.org/stable/user_guide.html.

[5] P. Schmid, "Dynamic mode decomposition of numerical and experimental data," *Journal of Fluid Mechanics*, vol. 656, pp. 5–28, 2010.

[6] S. L. Clainche and J. M. Vega, "Higher order dynamic mode decomposition," *SIAM Journal on Applied Dynamical Systems*, vol. 16, no. 2, p. 882–925, 2017.

[7] S. Wilcox and W. Marion, *Users manual for TMY3 data sets*. National Renewable Energy Laboratory Golden, CO, 2008.

[8] J. M. Finkelstein and R. E. Schafer, "Improved goodness-of-fit tests," *Biometrika*, vol. 58, no. 3, pp. 641–645, 1971.

Idaho National Laboratory