INL/RPT-22-69190-Revision-0

Real-Time Optimization Workflow Status Update

September 2022

Daniel Garrett, Takanori Kajihara, Junyung Kim, Paul W Talbot

Changing the World's Energy Future

INL is a U.S. Department of Energy National Laboratory operated by Battelle Energy Alliance, LLC

INL/RPT-22-69190-Revision-0

Real-Time Optimization Workflow Status Update

Daniel Garrett, Takanori Kajihara, Junyung Kim, Paul W Talbot

September 2022

Idaho National Laboratory
Idaho Falls, Idaho 83415

http://www.inl.gov

Prepared for the
U.S. Department of Energy
Under DOE Idaho Operations Office
Contract DE-AC07-05ID14517

INL/RPT-22-69190
Revision 0

Real-time Optimization Workflow Status Update

September 2022

Daniel Garrett
Takanori Kajihara
Junyung Kim
Paul W. Talbot

IES
Integrated Energy Systems

INL is a U.S. Department of Energy National Laboratory operated by Battelle Energy Alliance, LLC

INL/RPT-22-69190
Revision 0

Real-time Optimization Workflow Status Update

Daniel Garrett
Takanori Kajihara
Junyung Kim
Paul W. Talbot

September 2022

Idaho National Laboratory
Integrated Energy Systems
Idaho Falls, Idaho 83415

http://www.ies.inl.gov

Prepared for the
U.S. Department of Energy
Office of Nuclear Energy
Under DOE Idaho Operations Office
Contract DE-AC07-05ID14517

Page intentionally left blank

ABSTRACT

Economically optimal and safe operation of integrated energy systems (IES) requires optimization at many different time scales. A real-time optimization (RTO) workflow will attempt to maximize revenue and minimize operational costs on a time scale of minutes to hours. Such a workflow requires the use of a digital twin (DT), which is a virtual representation of a physical system. The DT is updated using real-time data from the physical system, and serves as a model in an optimization framework. The optimization results are then sent back to the physical system to complete the loop. This report details the progress made in developing building blocks for a DT/RTO framework.

The Risk Analysis Virtual Environment (RAVEN) platform within the Framework for Optimization of Resources and Economics (FORCE) tool suite can perform many of the tasks required for building a DT and performing RTO. The first item of this report details RAVEN enhancements that enable RAVEN workflows to be run in various environments.

Data communication between the physical system and its DT is essential for successful RTO. This includes preprocessing real-time data, loading data into a data warehouse, and querying the stored data. The second section of this report describes the progress made in implementing an adapter in Python in order for Deep Lynx to handle the data communication.

Typical dispatch optimization frameworks are built on linear programming (LP). The prototype RTO workflow developed in this report uses an LP problem as a part of a receding-horizon- or economic model predictive control (EMPC)-based optimization. The third section of this report details the framework of an RTO workflow in which the system consists of a simple electrical storage device.

A DT can be built from a reduced-order model (ROM). Integrating a ROM into a typical LP optimization framework has been challenging because most optimization packages require the user to write algebraic expressions for the system model. The final section of this report shows how an externally built RAVEN ROM can be integrated in an RTO framework by using the Python package Pyomo. This demonstrates the RTO workflow capability from a software-only perspective and is an important step in demonstrating the capability to implement an RTO workflow for a physical system.

Page intentionally left blank

CONTENTS

FIGURES

TABLES

Page intentionally left blank

ACRONYMS

API	application programing interface
BOP	balance of plant
CPS	cyber-physical system
CSV	comma-separated values
DIAMOND	Data Integration Aggregated Model and Ontology
DT	digital twin
EMPC	economic model predictive control
FORCE	Framework for Optimization of Resources and Economics
GUI	graphical user interface
IES	integrated energy systems
ISO	Independent System Operator
JSON	JavaScript Object Notation
LMP	locational marginal pricing
LP	linear programming
NHES	nuclear hybrid energy systems
NLP	nonlinear programming
NPP	nuclear power plant
PJM	Pennsylvania-New-Jersey-Maryland
RAVEN	Risk Analysis Virtual Environment
ROM	reduced-order model
RTO	real-time optimization
SDK	Software Development Kit
SQL	Structured Query Language
TBV	turbine bypass valve
TEDS	Thermal Energy Delivery System
TRANSFORM	Transient Simulation Framework of Reconfigurable Modules
XML	Extensible Markup Language

Page intentionally left blank

Real-time Optimization Workflow Status Update

1. INTRODUCTION

1.1 Background

Complex industrial processes such as integrated energy systems (IES) consist of many operations and components that each have their own objectives and are interconnected with other components. Optimal and safe operation of an IES requires optimization at many different time scales. As depicted in Figure 1, typical optimization and control hierarchies span time scales ranging from weeks to seconds. Real-time optimization (RTO) enables economic optimization by maximizing revenue and minimizing operational costs, typically on a time scale of minutes to hours.

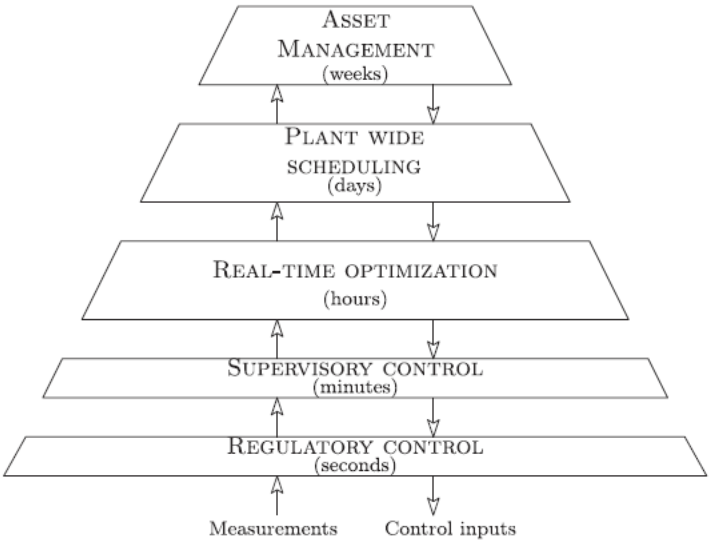


Figure 1. Typical hierarchy for process control.[1] RTO typically occurs on a time scale ranging from minutes to hours.

RTO is a workflow in which decision variables (also called setpoints) are optimized, using a system model and an economic model with process constraints, by solving a mathematical optimization problem.[1] The classical approach from the late 1980s was to use a steady-state model to describe the process behavior and optimize an economical objective function subject to the steady-state process model.[2] Numerous approaches to RTO have been applied, including model parameter adaptation,[3] integrated system optimization and parameter estimation,[4] modifier adaptation,[5] and economic model predictive control (EMPC).[6]

Regardless of the approach for solving the optimization problem, RTO workflows require a digital twin (DT). A DT consists of the physical system, its representation as a computational model (physics-based, data-informed, machine learning and artificial intelligence, etc.), data flowing from the physical system to the computational model, and data flowing back from the computational model to the physical system.[7] Data flowing from the physical system to the computational model are used to update or calibrate the computational model for use in the RTO workflow. Once the optimization is performed using the updated or calibrated computational model, the results are sent back to the physical system as recommended optimal setpoints for physical system operation. An implementation of the basic building blocks of a DT/RTO workflow are presented in the following sections of this report.

1.2 RAVEN in a Python Environment

The Risk Analysis Virtual Environment[8,9] (RAVEN) platform can perform many of the tasks required for building a DT and performing RTO. Such tasks could include building a computational model by training a reduced-order model (ROM) from a high-fidelity physics model, running optimization routines, or postprocessing statistics from coupled simulations. RAVEN was primarily designed to be executed from the command line/terminal. Once RAVEN has been executed, it reads an input Extensible Markup Language (XML) file detailing the analyses to be performed, conducts the analysis, and returns the desired results. Under this workflow, there is no capability to modify data during the analysis. Once RAVEN has finished running, any intermediate-step data not explicitly saved to a database or file are lost. The ability to access or modify data from intermediate steps and to rerun RAVEN workflows without rereading the input XML file, reinitializing, and running the entire workflow were capabilities missing in RAVEN.

This work included updating RAVEN to run in various environments, including Python scripts and Jupyter notebooks.[10,11] These updates allow RAVEN analyses to live in computer memory, thus enabling users to modify inputs, access data from intermediate steps, and rerun analyses without reinitializing and running the entire analysis from beginning to end. For real-time DT workflows, this is highly beneficial because the RAVEN analysis need only be instantiated once, then merely updated each time the analysis is run. This reduces computational costs by avoiding the overhead of having to instantiate RAVEN each and every time, and it allows RAVEN to run faster than real-time.

1.3 Deep Lynx Integration

Deep Lynx[12] is a unique open-source data warehouse software that allows users to store their data under a graph database governed by a custom ontology. It enables large projects to embrace digital engineering, and introduces DTs to projects via integration with various software systems. The data are stored in nodes in the graph, and their relationships represent how the nodes are all associated with each other. Users can query data by using the nodes and relationships, and this method fosters strong performance in querying complicated relationships, flexibility in expanding a given database, and agility in changing system requirements—as compared to a traditional relational database such as the Structured Query Language (SQL) database.

Deep Lynx governs its data format via the Data Integration Aggregated Model and Ontology (DIAMOND).[13,14] DIAMOND was developed as the ontology for nuclear power plant (NPP) and relevant nuclear application domains. Ontology consists of metatypes, relationship types, and relationships with their properties such as id, name, description, etc. Relationships are combinations of metatypes and relationship types. The metatypes and relationships are referenced when creating a node and an edge, respectively.

This work developed an adapter to integrate RTO with Deep Lynx, which then work in tandem to preprocess the given data, load them to Deep Lynx, and query stored data. Currently, the process of creating type-mapping transformations is controlled via the Deep Lynx graphical user interface (GUI), and other processes are controlled via the adapter. This work also modifies DIAMOND to tailor metatypes to specific data during RTO. In this report, operational sensor data from the Thermal Energy Delivery System (TEDS)[15] are used as a test dataset, similar to those data planned for application to RTO.

1.4 Real-time Optimization Workflow

In deregulated energy markets, the Independent System Operator (ISO) establishes markets for energy and ancillary services. In the day-ahead market, power generators bid capacity and price for each hour of the next 24-hour period, then submit these bids to the ISO once per day. The ISO matches up the projected electricity demand with the capacity bids, determines the clearing price, and selects which generators will participate in the power generation for each 1-hour period. The real-time market is a spot market in which the demand not covered by the day-ahead market is met. Generators submit bids up to about 75 minutes before the start of the trading hour, and, as with the day-ahead market, the ISO matches the demand with the capacity bids, determines the clearing price, and selects which generators will participate in the real-time period, which varies per ISO from five to 15 minutes. The ancillary service markets include markets for frequency regulation and reserves.

Energy storage opens up the possibility for energy arbitrage, which entails purchasing or storing (charging) energy when prices are low, then selling (discharging) energy when prices are high. The typical approach to energy arbitrage and dispatch optimization is to formulate revenue maximization as a linear programming (LP) or mixed-integer programming problem. This approach has been demonstrated for the day-ahead market,[16,17] day-ahead and real-time markets,[18,19] and day-ahead and frequency regulation markets.[20,21]

This work included developing an LP framework for optimizing dispatch from energy storage in a real-time market. The model includes a NPP that supplies a constant capacity, along with an electrical storage device that can charge from the NPP and discharge to the grid. The optimization uses the locational marginal pricing information to determine when it is economically beneficial to charge or discharge the electrical storage in order to maximize the revenue generated.

1.5 Digital Twin Development for Real-time Optimization

A DT is a virtual representation of a system, and is updated based on real-time data. It uses simulation, machine learning, and reasoning to aid in decision making. One application of DT is to conduct RTO on the control and operations of a cyber-physical system (CPS), and this requires system integration, analysis, exploration, and optimization to all be completed in the digital space.

The present work developed a workflow/optimization of a DT of a CPS. A system-level simulation model in Modelica[22] and RAVEN was mobilized to create a ROM that serves as a DT of the system. An optimization framework receives the ROM and provides optimal inputs for maximizing or minimizing an objective function.

In general, most optimization packages in Python (e.g., Pyomo,[23,24,25] PuLP,[26] cvxpy,[27,28] and ortools[29]) require the user to define inputs/outputs, objective functions, and constraints as algebraic expressions. PyNumero, a module within Pyomo, is a package for developing parallel algorithms for nonlinear programming (NLP). The unique aspect of PyNumero is that it provides users a DT interface as a block in a Pyomo model, enabling them to obtain inputs/outputs, constraints, and objective functions directly from the DT. In addition, PyNumero can be used alongside Pyomo to provide a unified Python platform for both modeling and solving optimization problems. Figure 2 shows a conceptual schematic of the optimization workflow using a DT and grey-box modeling in the PyNumero framework.

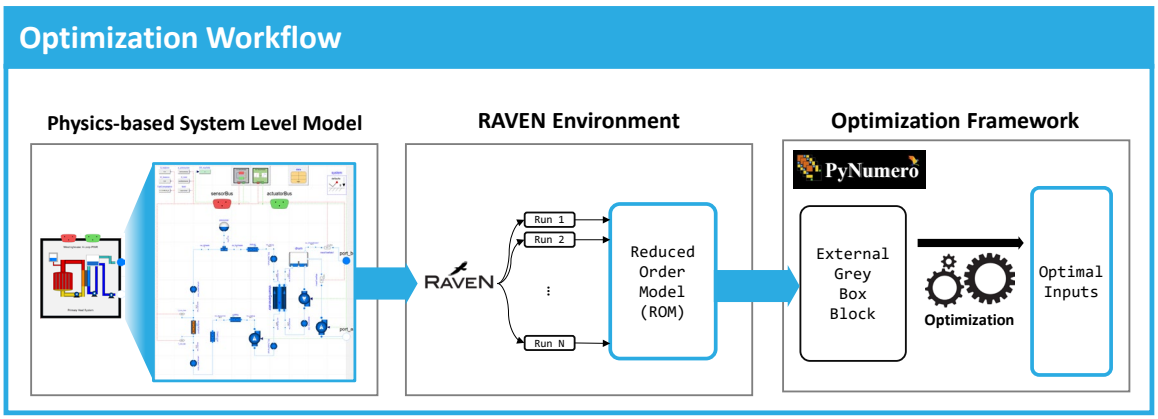


Figure 2. DT-to-PyNumero optimization workflow.

2. METHODS

This section summarizes the development of the building blocks for an IES DT and RTO workflow. These building blocks consist of a custom driver to enable RAVEN to run outside the command line/terminal, Deep Lynx integration with Python, receding-horizon-based optimization for dispatch RTO, and grey-box modeling integration with Pyomo.

2.1 RAVEN in a Python Environment

The typical RAVEN workflow execution from the command line/terminal consists of running the command:

```
raven_framework input_file.xml
```

The `raven_framework` command is a script that builds the RAVEN steps and analysis, and `input_file.xml` is the XML file containing all the settings and instructions for the RAVEN steps and analysis. The `raven_framework` script performs the following tasks:

1. Loads the virtual environment containing all the libraries needed for RAVEN workflows
2. Executes a Python script that does the following:
   a. Checks and applies all additional command line inputs and settings
   b. Loads the input XML file
   c. Instantiates, initializes, and runs the workflow.

Running RAVEN workflows in Python scripts or Jupyter notebooks requires methods for replicating the same steps as the command line/terminal version of executing the `raven_workflow` script. Implementing these steps in a Python or Jupyter notebook environment occurred over three phases: adding a Python driver instance to wrap the command line/terminal execution (RAVEN pull request #1685),[8] enabling RAVEN-running-RAVEN workflows (RAVEN pull request #1816),[8] and enabling re-running RAVEN workflows (RAVEN pull request #1843).[8]

Running RAVEN workflows in Python scripts requires that the user load the virtual environment `raven_libraries`, which contains all the libraries needed for running RAVEN workflows. The current RAVEN installation procedure uses the conda package manager to install and manage the required libraries for running RAVEN workflows. To activate the virtual environment, the user enters the following command in the command line/terminal:

```
conda activate raven_libraries
```

Once `raven_libraries` has been activated, the Python script can be run as:

```
python python_RAVEN_script.py
```

Running RAVEN workflows in Jupyter notebooks requires some additional setup steps because the current version of RAVEN does not include the Jupyter notebook library in the `raven_libraries` environment. To ensure that the `raven_libraries` environment is available to the Jupyter kernel, the user should perform the following tasks from the command line/terminal:

1. Ensure that the base conda environment is activated:

   ```
   conda activate base
   ```

2. Install the Jupyter notebook:

   ```
   conda install -c conda-forge notebook
   ```

3. Install nb_conda_kernels:

   ```
   conda install -c conda-forge nb_conda_kernels
   ```

4. Activate raven_libraries:

   ```
   conda activate raven_libraries
   ```

5. Install `ipykernel`:

   ```
   conda install -c conda-forge ipykernel.
   ```

Starting up a Jupyter notebook can be done from the command line/terminal in any conda environment by using the command:

```
jupyter notebook
```

All the user's conda environments that have the `ipykernel` package installed will be available for use in notebooks by selecting the appropriate kernel.

In RAVEN pull request #1643, a new custom driver for RAVEN was written to enable running RAVEN workflows in Python scripts or Jupyter notebooks. This custom driver can be found at `raven/ravenframework/CustomDrivers/PythonRaven.py`.[8] `PythonRaven.py` includes a class called `Raven` that contains methods for loading a RAVEN input XML file, running the RAVEN workflow, and interrogating RAVEN entities. When `ravenframework` is on the Python path, this class can be imported in a Python script or Jupyter notebook as:

```
from ravenframework import Raven
```

Instantiating a `Raven` object results in checking that all of the libraries needed to run a RAVEN workflow are installed, as well as initiating the setup required to run a RAVEN workflow. Once the `Raven` object has been instantiated, the RAVEN input file can be loaded using the `loadWorkflowFromFile` method, then the RAVEN workflow can be run via the `runWorkflow` method. After the RAVEN workflow is complete, the user can interrogate various RAVEN entities by applying the `getEntity` method. Section 3.1 demonstrates how to do the full workflow, from instantiating the `Raven` object to running the workflow, for a Python script and Jupyter notebook.

More complicated RAVEN workflows require RAVEN to run itself as the model (e.g., multilevel optimization with inner and outer loops). RAVEN pull request #1816 enabled these types of workflows by modifying the code interface in `raven/ravenframework/CodeInterfaces/RAVEN/RAVENInterface.py` and thus enabling RAVEN to use itself as an external model. These modifications included pointing to a Python file that will run the RAVEN workflow inside of another RAVEN workflow. Changing the executable file to a Python file instead of a shell script allows the RAVEN-running-RAVEN workflow to run successfully in a Python script or Jupyter notebook environment, independent of the user's operating system.

RAVEN was initially set up to run a workflow from beginning to end. Rerunning a workflow without rereading the input file and reinitializing all the steps and components needed for the workflow was originally impossible. To enable a RAVEN workflow to be rerun without reading the input file and re-initializing everything, many changes were necessary to the RAVEN source code, resulting in RAVEN pull request #1843. The most user-facing change was made to `raven/ravenframework/CustomDrivers/PythonRaven.py`. As part of the `runWorkflow` method, `PythonRaven.py` checks whether the workflow has been run previously. If yes, it resets the workflow and reruns it without reloading the input file.

When running a workflow, all the RAVEN entities and objects are collected in `raven/ravenframework/Simulation.py`. To enable the rerunning of RAVEN workflows, many new changes were introduced to this file. A binary variable called `ranPreviously` was introduced to track whether the workflow had been run previously (`ranPreviously = True`) or not (`ranPreviously = False`). When each step of the workflow is initiated, `Simulation.py` checks to see if `ranPreviously` is `True`. If yes, many entities are reset to their original state so they can be rerun. For `Steps(IOStep, MultiRun, and SingleRun)`, a `flushStep` method was introduced to return the `Step` entity to its original state. For `DataObjects`, `Optimizers`, and `Samplers`, a `flush` method was introduced to reset the entity to its original state before running the workflow. When the workflow completes, the method `resetSimulation` is run to reset warnings so that only those warnings generated when running the current workflow are shown, and not the warnings from the current workflow plus all the previous times the workflow was run. An example of rerunning a RAVEN workflow is given in Section 3.1.

2.2 Deep Lynx Integration

Deep Lynx offers a Python Software Development Kit (SDK)[30] for integrating Deep Lynx and Python-coded projects. The SDK creates the application programming interface (API), then the RTO adapter for Deep Lynx imports this SDK in order to communicate with a Deep Lynx instance.

The whole Deep Lynx process is conducted to create a container with an ontology, modify the ontology in the container, create a data source, upload data, create type-mapping transformations, run transformations based on the type mapping, and query data. In Deep Lynx, a container is created from an existing DIAMOND ontology file. Next, metatypes and relationships not listed in the original DIAMOND file—but that are necessary for storing experimental datasets—are added to the ontology. Only the relationship types from the original DIAMOND file are used. The process of adding metatypes and relationships is conducted using the Deep Lynx GUI. Lists of the created new metatypes and relationships are shown in Table 1 and Table 2, respectively. The meaning and specific examples of relationship type are described in Section 3.2.

Table 1. Metatypes added to the original DIAMOND ontology.

Metatypes
DifferentialPressureMeter
FlowMeter
PressureTransducer

Table 2. Relationships added to the original DIAMOND ontology.

Parent	Relationship type	Child
DifferentialPressureMeter	decomposes	System
DifferentialPressureMeter	relates	Pump
DifferentialPressureMeter	affectedBy	ControlValve
DifferentialPressureMeter	affectedBy	Thermocouple
FlowMeter	decomposes	System
FlowMeter	relates	Thermocouple
FlowMeter	affectedBy	Pump
FlowMeter	affectedBy	DifferentialPressureMeter
FlowMeter	affectedBy	Thermocouple
FlowMeter	affectedBy	ControlValve
FlowMeter	affectedBy	PressureTransducer
FlowMeter	affectedBy	FlowMeter
PressureTransducer	decomposes	System
PressureTransducer	relates	Thermocouple
PressureTransducer	affectedBy	Thermocouple
Pump	relates	DifferentialPressureMeter
System	decomposes	System
Thermocouple	relates	FlowMeter
Thermocouple	affectedBy	Pump
Thermocouple	affectedBy	DifferentialPressureMeter
Thermocouple	affectedBy	ControlValve
Thermocouple	affectedBy	FlowMeter
Thermocouple	relates	PressureTransducer
Thermocouple	affectedBy	Thermocouple
Thermocouple	affectedBy	ElectricHeater
Thermocouple	relates	FlowMeter
Thermocouple	affectedBy	PressureTransducer
ElectricalPowerSystem	decomposes	System

All data transferred between RTO and Deep Lynx are passed through the RTO adapter for Deep Lynx. Any given data from the RTO are preprocessed so as to be suitable for loading into Deep Lynx. When the Deep Lynx GUI is first run, an API key is generated and then paired with a Secret key. The adapter accesses Deep Lynx via an access token generated by these keys (Figure 3). Data are ingested in comma-separated values (CSV) format and processed through the adapter and Deep Lynx. First, a data source is created to identify a specific project and to instruct Deep Lynx on how and where to retrieve the data (Figure 4). The data are then uploaded to a selected data source (Figure 5).

```python
import deep_lynx
import requests

# initialize an ApiClient for use with deep_lynx APIs
configuration = deep_lynx.configuration.Configuration()
configuration.host = 'http://localhost:8090' # Your Base URL
apiClient = deep_lynx.ApiClient(configuration)

# authenticate via an API key and secret
authApi = deep_lynx.AuthenticationApi(apiClient)
apiKey = '' # Your API key
apiSecret = ''  # Your API Secret

token = authApi.retrieve_o_auth_token(x_api_key=apiKey, x_api_secret=apiSecret, x_api_expiry='24h')

# update apiClient with bearer token
apiClient = deep_lynx.ApiClient(configuration, "Authorization", F"Bearer {token}")

containerApi = deep_lynx.ContainersApi(apiClient)
datasourcesApi = deep_lynx.DataSourcesApi(apiClient)
```

Figure 3. Initialization of API client and authentication.

```python
def create_datasources(datasourcesApi: deep_lynx.DataSourcesApi, source_name: str, containerId: str):
    """
    Create new datasource.
    Args
        datasourcesApi (deep_lynx.DataSourcesApi): deep lynx data sources api
        source_name (str): name of a data source
        containerId (str): ID of the container where the data source is contained
    Returns
        datasourceId (str): datasource ID
        datasource (datasourceApi): deep lynx datasource api
    """
    source_body = deep_lynx.CreateDataSourceRequest(source_name, adapter_type='standard', active=True)
    datasource = datasourcesApi.create_data_source(source_body, containerId)
    datasourceId = datasource.value.id

    return datasourceId, datasource
```

Figure 4. Data source creation via the adapter.

```python
def upload_file(datasourcesApi: deep_lynx.DataSourcesApi, containerId: str, datasourceId: str, file_path: str):
    """
    Create new datasource.
    Args
        datasourcesApi (deep_lynx.DataSourcesApi): deep lynx data sources api
        containerId (str): ID of the container where the data source is contained
        datasourceId (str): ID of the data source where the file is uploaded
        file_path (str): the data file path to be uploaded
    Returns
        uploadedfile (datasourcesApi): data sources api containing uploaded file
    """
    uploadedfile = datasourcesApi.upload_file(containerId, datasourceId, file=file_path, async_req=False)
    if len(uploadedfile['value']) > 0:
        print('Data was imported to deep lynx.')
    else:
        print('ERROR: Data was not imported to deep lynx.')
    return uploadedfile
```

Figure 5. Data upload via the adapter.

The type-mapping transformation determines how to store the data in a graph database structured by ontology. The data are mapped to nodes, edges, their properties, and time-series tables defined by metatypes and relationships. Type-mapping transformation is currently only conducted using the Deep Lynx GUI. Once the transformation is set to encompass all the data from a given source, type mappings are constantly applied to automatically match the data. Thus, there is no need to create new type-mapping transformations. For the type-mapping setting, the information shown in Figure 6, Figure 7, and Figure 8 is filled out for the node(s), edge, and time-series table. For node type-mapping, a metatype and a unique identifier key, which selects what raw data are chosen for this mapping, are set. Furthermore, properties related to this node can be set from the raw data or be given a constant value (Figure 6). For relationship type-mapping, a relationship type and the information to identify the parent(s) and child(ren) are set. The parent or child ID key is equivalent to the unique identifier key for the node type-mapping (Figure 7). Through time-series data transformation, a time-series table is created and then attached to a node. The actual mapping is accomplished by using the node ID key to identify the target node and then create a table using the payload key. In this case, the node ID key and payload key cannot be the same. Furthermore, the first column must include the ‘date’ data type in the specified format (see Figure 8).

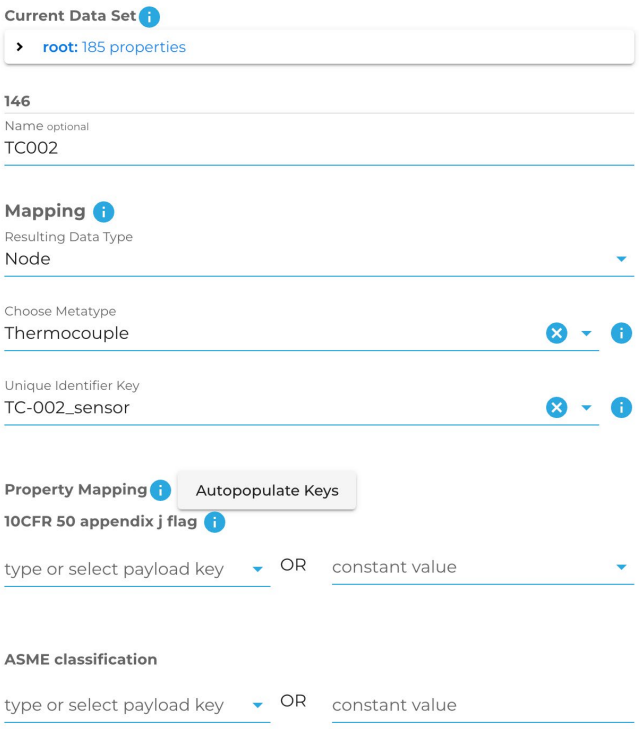


Figure 6. Node type-mapping transformation.

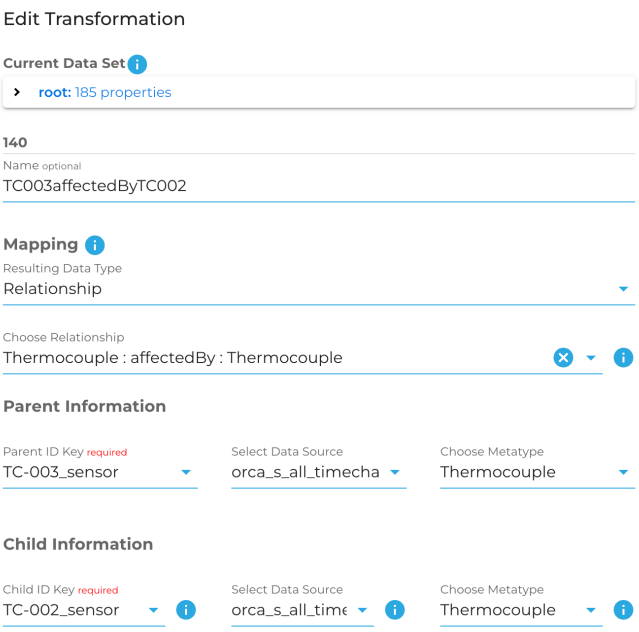


Figure 7. Relationship type-mapping transformation.

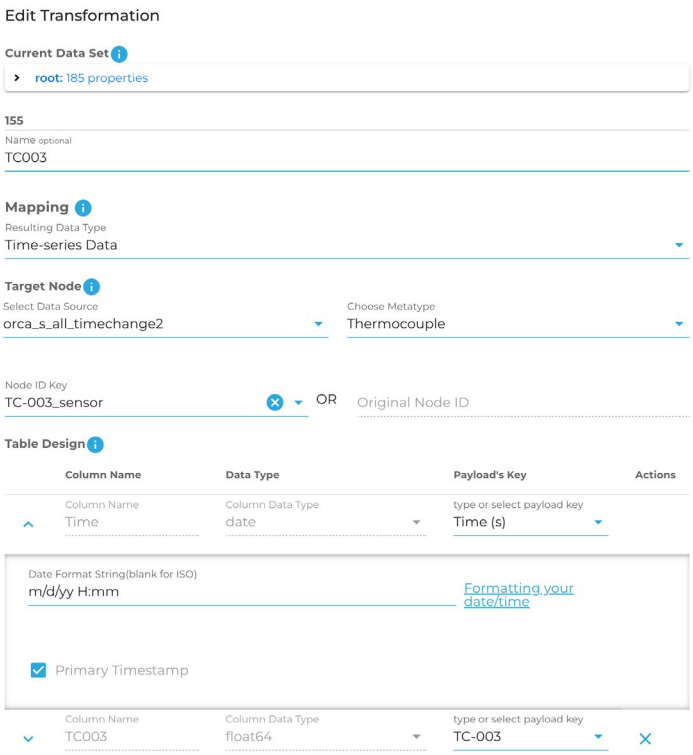


Figure 8. Time-series data mapping transformation.

Stored data in Deep Lynx can be queried using GraphQL[31] (Figure 9 and Figure 10). Currently, users can query data based on four different categories of information: metatypes, relationships, graph, and time. The metatype query can request all nodes labeled a specific metatype. The relationship query can request all edges labeled a specific relationship (i.e., two metatypes combined with a relationship type). The graph type query can specify a root node and depth. All nodes that are n-depth layers, apart from the root node, are selected. The time-series query can extract a specific time range for the selected columns in a time-series table attached to a node. These queries can also filter results by property values, using operators such as equal, greater than, and like. The query shown in Figure 10 will read time-series data from the TC-001 node (temperature measured with a thermocouple) between 10 and 12 p.m. on August 10, 2021.

In the RTO workflow, the stored data are assumed fixed during operation. Thus, the Deep Lynx settings (e.g., ontology and type-mapping transformations) are locked in before the operation starts. In addition, the preprocessing module and the query que in the RTO adapter are tailored to the specific operation. If the experimental system or any of the data sources change, so does the Deep Lynx and adapter settings.

```python
def query_data(body: str, requestURL: str = 'http://localhost:8090/containers/1/data'):
    """
    Query data.
    Args
        body (str): GraphQL Query
        requestURL (str): The endpoint where all GraphQL queries are sent
    Returns
        queried_data (json): the data queried from the database
    """
    queried_data = requests.post(requestURL, json={"query": body}, headers={"Authorization":"Bearer "+token}).json()
    return queried_data
```

Figure 9. Data query made via the adapter.

```
timeseries_query = ( """
{
    TC001 (Time: {
        operator: "between", value: ["2021-08-10 10:00:00", "2021-08-10 11:00:00"]
        }){
        TC001
        Time
        _record {
            count
        }
    }
}
"""
)

result = query_data(timeseries_query,'http://localhost:8090/containers/1/graphs/nodes/2375970/timeseries')
```

Figure 10. GraphQL query example for time-series data.

2.3 Real-time Optimization Workflow

The IES model used for this RTO workflow includes a NPP operating at constant capacity and an electrical storage device that can charge from the NPP and then discharge the energy to the real-time market. The NPP sends electricity either directly to the real-time market or to the electrical storage device. The electrical storage device is characterized by the following parameters:

- Power Rating (MW): The maximum amount of power that the electrical storage can charge or discharge
- Energy Capacity (MWh): The amount of energy the electrical storage device can store
- Efficiency (%): Conversion losses encountered when energy is stored during charge and released during discharge, implemented as round-trip efficiency.

The state of charge of the electrical storage device is modeled by the following difference equation:

$$S_t = S_{t-1} + \left(\sqrt{\gamma_{RTE}}\, q^C_{t-1} - \frac{q^D_{t-1}}{\sqrt{\gamma_{RTE}}}\right) \Delta t \times \frac{1hr}{60min} \quad (1)$$

where S_t is the current state of charge in MWh, S_{t-1} is the state of charge at the previous time step, γ_{RTE} is the round-trip efficiency, q^C_{t-1} is the charging power (MW) from the previous time step, q^D_{t-1} is the discharging power (MW) from the previous time step, Δt is the time step in minutes, and $\frac{1hr}{60min}$ is a conversion factor for ensuring that the state of charge is given in units of MWh. In other words, the state of charge at the current time step is the state of charge at the previous time step, plus the amount of charge

from the previous time step, minus the amount of discharge from the previous time step. The charging and discharging rates are assumed to be held constant throughout the duration of each time step.

The objective function for dispatch optimization is given by:

$$\max \sum_{t=1}^{T} P_t (q_{NPP} - q_t^C + q_t^D) \tag{2}$$

where T is the total number of time steps, P_t is the locational marginalized pricing (LMP) in \$/MW at time step t, q_{NPP} is the constant power generation of the NPP (MW), and q_t^C and q_t^D are the charging and discharging powers (MW), respectively, at the time step t. The objective function is optimized subject to both the state-of-charge model above and the following constraints:

$$0 \leq S_t \leq S_{max} \tag{3}$$

$$0 \leq q_t^C \leq q_{max}^C \tag{4}$$

$$0 \leq q_t^D \leq q_{max}^D \tag{5}$$

$$0 \leq q_t^D \leq \sqrt{\gamma_{RTE}} S_t \frac{60min}{\Delta t} + \gamma_{RTE} q_t^C \tag{6}$$

where S_{max} is the energy capacity (MWh) of the electrical storage device and q_{max}^C and q_{max}^D are the power ratings (MW) for charging and discharging, respectively. The final constraint says that the electrical storage device may discharge up to the total amount of energy stored in the device.

The RTO workflow is based on a receding-horizon- or EMPC-based optimization.[6] EMPC uses a dynamic model to optimize an objective function over a finite time horizon. The decision variable for the optimization problem is the input trajectory (charging and discharging) over the time horizon. The system (state of charge) model is used as an additional constraint to the optimization. EMPC is solved in a receding horizon fashion. At a given time, τ, EMPC receives a state measurement (state of charge at τ) that is used to initialize the dynamic model. An optimal piecewise input trajectory for the objective function and constraints is computed over the time horizon, corresponding to times $t \in [\tau, \tau + \Delta t, \tau + 2\Delta t, \ldots, \tau + N\Delta t]$, where N is the total number of time steps. The first of the optimized input trajectories (charging and discharging) is to be implemented over the next time period Δt. At the next time instance, $\tau + \Delta t$, the EMPC is re-solved for the optimal input trajectories.

2.4 Grey-box Modeling

As mentioned in Section 1.5, the `ExternalGreyBoxModel` in PyNumero enhances the interface between a DT and the RTO framework: one need not explicitly provide inputs, constraints, and outputs as algebraic expressions to the optimization framework, but simply plug in the ROM to the `ExternalGreyBox` block in the PyNumero framework.

Two files are required to solve the optimization problem in the PyNumero framework: a Python script and a pickled ROM file. The Python script is for interfacing the RAVEN ROM with PyNumero; it passes the pickled ROM file into the `ExternalGreyBoxBlock` in PyNumero.

This section describes how the Python script file (`rom_out.py`) generated by `<IOStep name="serialize">` and a ROM file generated by `<IOStep name="pickle">` in the RAVEN input file interface with the `ExternalGreyBoxModel` in PyNumero.

`rom_out.py` contains the infrastructure for inheriting the `ExternalGreyBoxModel` from PyNumero and other code blocks to print a file based on Pyomo syntax. The full code lines of `rom_out.py` are given in Appendix A. However, it is worth noting that, here, the Python script file

simply works as a template for users unfamiliar with solving optimization problems in the PyNumero framework. If desired, users can change the objective function, constraints, or other parameters.

First, the necessary packages are imported (Figure 11). Note that Pyomo objects exist within `Pyomo.environ`. In addition, `ExternalGreyBoxModel` and `ExternalGreyBoxBlock` must be imported from the PyNumero package. `ExternalGreyBoxBlock` automatically creates input and output variables corresponding to `ExternalGreyBoxModel`. It enables users to pass these in and use any provided variable data objects instead of having to create them. The `pickle` package is imported to load the pickled ROM file.

```
# MODEL GENERATED BY RAVEN (raven.inl.gov)
import pyomo.environ as pyo
from pyomo.contrib.pynumero.interfaces.external_grey_box import ExternalGreyBoxModel, ExternalGreyBoxBlock
from pyomo.contrib.pynumero.dependencies import (numpy as np)
from pyomo.contrib.pynumero.asl import AmplInterface
from pyomo.contrib.pynumero.algorithms.solvers.cyipopt_solver import CyIpoptSolver, CyIpoptNLP
import os, sys, pickle
from contextlib import redirect_stdout
```

Figure 11. `rom_out.py` Python package imports.

There are three major code blocks in `rom_out.py`:

1. The `ravenROM` class that inherits from `ExternalGreyBoxModel` (Figure 12)
2. Constructing a Pyomo model including an `ExternalGreyBoxBlock` (Figure 13)
3. Pyomo grey box for NLP (Figure 14).

The `ravenROM` class shown in Figure 12 is what allows an external model to be solved in PyNumero. It is for loading the pickled ROM file and collecting information from the external model loaded, including a list of names for inputs and outputs. This class contains a method for computing the derivatives and Jacobian of the output with respect to the inputs. The derivatives and Jacobian calculated will be used later for an optimization solver.

The function `pyomoModel` shown in Figure 13 sets up `ExternalGreyBoxBlock` for constructing a Pyomo model. It creates Pyomo variables to represent the inputs to and outputs from the external model. To set up the objective function, the `sense` option can be used as an argument of `pyo.Objective` to specify whether the objective function should be minimized or maximized. Users are given the flexibility to define the objective function based on the problem at hand, since an optimization problem could be set up in many ways.

Finally, the code in Figure 14 creates a Pyomo grey box NLP problem and provides Pyomo the model containing the external grey box block(s). This code block is how Pyomo actually solves an example optimization problem. Note that, in this code block, the external model (i.e., `ext_model`) is loaded into `pyomoModel`, and users can freely add inputs, constraints, and an objective in `concreteModel`. Currently, cyipopt, the Python wrapper for the IPOPT [32] optimization package, written in Cython, is the only solver supported.

```python
# RAVEN ROM PYOMO GREY MODEL CLASS
class ravenROM(ExternalGreyBoxModel):
  def __init__(self, **kwargs):
    self._rom_file = kwargs.get("rom_file")
    self._raven_framework = kwargs.get("raven_framework")
    self._raven_framework = os.path.abspath(self._raven_framework)
    if not os.path.exists(self._raven_framework):
      raise IOError('The RAVEN framework directory does not exist in location "' + str(self._raven_framework)+'" !')
    if os.path.isdir(os.path.join(self._raven_framework,"ravenframework")):
      sys.path.append(self._raven_framework)
    else:
      raise IOError('The RAVEN framework directory does not exist in location "' + str(self._raven_framework)+'" !')
    from ravenframework.CustomDrivers import DriverUtils as dutils
    dutils.doSetup()
    # de-serialize the ROM
    self._rom_file = os.path.abspath(self._rom_file)
    if not os.path.exists(self._rom_file):
      raise IOError('The serialized (binary) file has not been found in location "' + str(self._rom_file)+'" !')
    self.rom = pickle.load(open(self._rom_file, mode='rb'))
    #
    self.settings = self.rom.getInitParams()
    # get input names
    self._input_names = self.settings.get('Features')
    # n_inputs
    self._n_inputs = len(self._input_names)
    # get output names
    self._output_names = self.settings.get('Target')
    # n_outputs
    self._n_outputs = len(self._output_names)
    # input values storage
    self._input_values = np.zeros(self._n_inputs, dtype=np.float64)
  def return_train_values(self, feat):
    return self.rom.trainingSet.get(feat)
  def input_names(self):
    return self._input_names
  def output_names(self):
    return self._output_names
  def set_input_values(self, input_values):
    assert len(input_values) == self._n_inputs
    np.copyto(self._input_values, input_values)
  def evaluate_equality_constraints(self):
      raise NotImplementedError('This method should not be called for this model.')
  def evaluate_outputs(self):
    request = {k:np.asarray(v) for k,v in zip(self._input_names,self._input_values)}
    outs = self.rom.evaluate(request)
    eval_outputs = np.asarray([outs[k].flatten() for k in self._output_names], dtype=np.float64)
    return eval_outputs.flatten()
  def evaluate_jacobian_outputs(self):
    request = {k:np.asarray(v) for k,v in zip(self._input_names,self._input_values)}
    derivatives = self.rom.derivatives(request)
    jac = np.zeros((self._n_outputs, self._n_inputs))
    for tc, target in enumerate(self._output_names):
      for fc, feature in enumerate(self._input_names):
        jac[tc,fc] = derivatives['d{}|d{}'.format(target, feature)]
    return jac
```

Figure 12. `ravenROM` class from `rom_out.py`.

```python
def pyomoModel(ex_model, show_solver_log=False):
    m = pyo.ConcreteModel()
    m.egb = ExternalGreyBoxBlock()
    m.egb.set_external_model(ex_model)
    for inp in ex_model.input_names():
        m.egb.inputs[inp].value = np.mean(ex_model.return_train_values(inp))
        m.egb.inputs[inp].setlb(np.min(ex_model.return_train_values(inp)))
        m.egb.inputs[inp].setub(np.max(ex_model.return_train_values(inp)))
    for out in ex_model.output_names():
        m.egb.outputs[out].value = np.mean(ex_model.return_train_values(out))
        m.egb.outputs[out].setlb(np.min(ex_model.return_train_values(out)))
        m.egb.outputs[out].setub(np.max(ex_model.return_train_values(out)))
    m.obj = pyo.Objective(expr=m.egb.outputs[out], sense = pyo.minimize)
    return m
```

Figure 13. Function `pyomoModel` from `rom_out.py`.

```python
if __name__ == '__main__':
    for cnt, item in enumerate(sys.argv):
        if item.lower() == "-r":
            rom_file = sys.argv[cnt+1]
        if item.lower() == "-f":
            raven_framework = sys.argv[cnt+1]
    ext_model = ravenROM(**{'rom_file':rom_file,'raven_framework':raven_framework})
    concreteModel = pyomoModel(ext_model)
    ### here you should implement the optimization problem
    ###
    solver = pyo.SolverFactory('cyipopt')
    solver.config.options['hessian_approximation'] = 'limited-memory'
    results = solver.solve(concreteModel)
    print(results)
    count = 0
    lines = str(results).split("\n")
    with open ("GreyModelOutput_cyipopt.txt", "w") as textfile:
        with redirect_stdout(textfile):
            print("----------------------< Output Summary >----------------------" + "\n")
            for element in lines:
                textfile.write(element + "\n")
            print("---------------------< Optimization Results >---------------------" + "\n")
            concreteModel.pprint()
```

Figure 14. Solution of Pyomo grey box NLP.

Once the code runs successfully, the file `GrayModelOutput_cyipopt.txt` (Figure 15) is generated in the same folder as the `rom_out.py` file. It contains a summary of the optimization problem, including which solver was used, whether the problem terminated successfully, and what the optimized input/output values are. In the Optimization Result in `GrayModelOutput_cyipopt.txt`, the output value, `ans`, is minimized (0.358796455415) when both y1 and y2 equal 0.938770350605.

Figure 15. Optimization results produced by `rom_out.py`.

3. RESULTS

3.1 RAVEN in a Python Environment

This section gives examples of how to run a RAVEN workflow in a Python script or Jupyter notebook. First, a simple Python script is presented to show what is required to run a RAVEN workflow and how to run the Python script. Next, an example of a Jupyter notebook is presented to demonstrate how to run and rerun RAVEN workflows in a Jupyter notebook.

3.1.1 Running RAVEN Workflows from a Python Script

Figure 16 shows `raven_test.py`, a Python script containing all the necessary code to run a RAVEN workflow from a Python environment. In the first portion of the script, all the necessary Python packages are loaded. These include `os` to work with paths and `sys` to append additional paths to the Python path. The function `runWorkflow` is defined for convenience in running the RAVEN workflow, and also lets users know whether the workflow ran successfully. Next, the path to the `ravenframework` directory is located so the `Raven` class may be imported. At this point, all the necessary setup has been completed. The procedure to run a RAVEN workflow is then:

1. Provide the path to the desired workflow input XML file; in this example, `raven/tests/framework/basic.xml`.
2. Change the working directory to the directory containing the input XML file (unnecessary if the script is run from the same directory as the desired input XML file).
3. Instantiate the `Raven` object.
4. Use the `Raven` object's `loadWorkflowFromFile` method to load the input XML file.
5. Use the `Raven` object's `runWorkflow` method to run the workflow; in this example, this is wrapped in the convenience function `runWorkflow`.

Execution of the Python script to run the RAVEN workflow is shown in Figure 17. The user must navigate to the directory where the Python script resides. In the example shown, the directory is `C:\Users\garrd\projects\JupyterRaven`. The `raven_libraries` environment must be activated with the following command:

```
conda activate raven_libraries
```

The script can then be executed by running the command:

```
python raven_test.py
```

Figure 18 shows the resulting messages that are printed to the screen. For demonstration purposes, the example shown in Figure 16, Figure 17, and Figure 18 involves running one of the RAVEN integration tests, located at `raven/tests/framework/basic.xml`. The user can specify any input file they wish to run as a part of the Python script.

```
import os, sys

# convenience function to run RAVEN workflow
# lets the user know if the workflow ran successfully or not
def runWorkflow(raven):
  returnCode = raven.runWorkflow()
  if returnCode != 0:
    print('RAVEN did not run successfully')
  else:
    print('RAVEN ran successfully')

# find the path to ravenframework so that it can be imported
home_path = os.path.expanduser('~')
for root, dirs, files in os.walk(home_path):
  if 'ravenframework' in dirs and 'build' not in root:
    frameworkDir = root

sys.path.append(frameworkDir)

# import ravenframework
from ravenframework import Raven

if __name__ == '__main__':
  # path to the target workflow XML file
  targetWorkflow = os.path.abspath(os.path.join(frameworkDir, 'tests', 'framework',
'basic.xml'))
  os.chdir(os.path.dirname(targetWorkflow))
  # instantiate Raven object
  raven = Raven()
  # load the input XML file
  raven.loadWorkflowFromFile(targetWorkflow)
  # run the workflow
  runWorkflow(raven)
```

Figure 16. Python script `raven_test.py` to demonstrate running a RAVEN workflow.

```
C:\Users\garrrd\projects\JupyterRaven>conda activate raven_libraries

(raven_libraries) C:\Users\garrrd\projects\JupyterRaven>python raven_test.py
```

Figure 17. Commands to run the `raven_test.py` script.

```
    Copyright 2017 Battelle Energy Alliance, LLC

    Licensed under the Apache License, Version 2.0 (the "License");
    you may not use this file except in compliance with the License.
    You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

    Unless required by applicable law or agreed to in writing, software
    distributed under the License is distributed on an "AS IS" BASIS,
    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
    See the License for the specific language governing permissions and
    limitations under the License.


    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

          .---.        .------######       #####     ###   ###  ########  ###    ###
         /     \  __   /    --###  ###    ###  ###   ###   ###  ###       #####  ###
        / /     \(  )/    --###  ###    ###   ###  ###   ###  ######    ### ######
       //////   ' \/ `   --#######     #########  ###   ###  ###       ###  #####
      //// / // :    :   -###   ###   ###   ###    ######   ####      ###   ####
     // /   /  /`    '---###    ###  ###   ###      ###    ########  ###    ###
    //           //..\\
    ===========UU====UU=============================================================
              '//||\\`
                ''``

    RAVEN Python dependencies located and checked.
    Loading plugin "ExamplePlugin" at C:\Users\garrrd\projects\raven\plugins\ExamplePlugin
     ... successfully imported "ExamplePlugin" ...
    Loading plugin "FARM" at C:\Users\garrrd\projects\raven\plugins\FARM
     ... successfully imported "FARM" ...
    Loading plugin "HERON" at C:\Users\garrrd\projects\raven\plugins\HERON
     ... successfully imported "HERON" ...
    Loading plugin "LOGOS" at C:\Users\garrrd\projects\raven\plugins\LOGOS
     ... successfully imported "LOGOS" ...
    Loading plugin "SR2ML" at C:\Users\garrrd\projects\raven\plugins\SR2ML
     ... successfully imported "SR2ML" ...
    Loading plugin "TEAL" at C:\Users\garrrd\projects\raven\plugins\TEAL
     ... successfully imported "TEAL" ...
    C:\Users\garrrd\Miniconda3\envs\raven_libraries\lib\site-
packages\shiboken2\files.dir\shibokensupport\feature.py:139: DeprecationWarning: the imp module is
deprecated in favour of importlib; see the module's documentation for alternative uses
      return original_import(name, *args, **kwargs)
    Target file found at "C:\Users\garrrd\projects\raven\tests\framework\basic.xml"
    (    0.00 sec) SIMULATION               : Message         -> Global verbosity level is "all"
    (           ) UTILS                    : Message         -> importing module
C:\Users\garrrd\projects\raven\tests\framework\AnalyticModels\projectile
    InputData: Using param spec "DataSet" to read XML node "PointSet.
    InputData: Using param spec "DataSet" to read XML node "PointSet.
    (    0.00 sec) SIMULATION               : Message         -> Simulation started at 2022-08-11 09:03:19
    (    0.00 sec) SIMULATION               : Message         -> -- Beginning MultiRun step "sample" ... --
    (    0.00 sec) STEP MULTIRUN            : Message         -> ***  Beginning initialization ***
    (    0.00 sec) SAMPLER MONTECARLO       : Message         -> No restart for SAMPLER MONTECARLO
    (    0.00 sec) STEP MULTIRUN            : Message         -> ***    Initialization done    ***
    (    0.00 sec) STEP MULTIRUN            : Message         -> ***       Beginning run       ***
    (    0.19 sec) STEP MULTIRUN            : Message         -> ***       Run finished        ***
    (    0.19 sec) STEP MULTIRUN            : Message         -> ***     Closing the step      ***
    (    0.19 sec) STEP MULTIRUN            : Message         -> ***        Step closed        ***
    (    0.19 sec) SIMULATION               : Message         -> -- End step sample of type: MultiRun --

    (    0.19 sec) SIMULATION               : Message         -> -- Beginning IOStep step "plot" ... --
    (    0.19 sec) STEP IOCOMBINED          : Message         -> ***  Beginning initialization ***
    (    0.25 sec) STEP IOCOMBINED          : Message         -> ***    Initialization done    ***
    (    0.25 sec) STEP IOCOMBINED          : Message         -> ***       Beginning run       ***
    (    0.43 sec) STEP IOCOMBINED          : Message         -> ***       Run finished        ***
    (    0.43 sec) STEP IOCOMBINED          : Message         -> ***     Closing the step      ***
    (    0.43 sec) STEP IOCOMBINED          : Message         -> ***        Step closed        ***
    (    0.43 sec) SIMULATION               : Message         -> -- End step plot of type: IOStep --

    (    0.44 sec) SIMULATION               : Message         -> Run complete!

RAVEN ran successfully
```

Figure 18. Output from running the `raven_test.py` script.

3.1.2 Running RAVEN Workflows in a Jupyter Notebook

A Jupyter notebook[10,11] is a locally run web application for creating computational documents. Jupyter notebooks support many programming languages, including Python. These notebooks can include text, code, plots, and many other elements. The main benefits of these notebooks are that they include interactive output and can be shared with others. They are very popular for performing data science analyses. As such, running RAVEN from a Jupyter notebook would be highly beneficial.

To run RAVEN workflows, a Jupyter notebook must be set up to access the `raven_libraries` environment (similar to the process outlined in Section 2.1). To verify this was performed correctly, inside the Jupyter notebook the kernel must be changed to `Python [conda env:raven_libriaries]`, as shown in the upper-right corner of Figure 19. The kernel can be changed by accessing "Change kernel" from the Kernel dropdown menu (shown in the middle of Figure 19). With the kernel set to `Python [conda env:raven_libraries]`, the notebook has access to all the libraries needed by RAVEN, and is ready to run a RAVEN workflow.

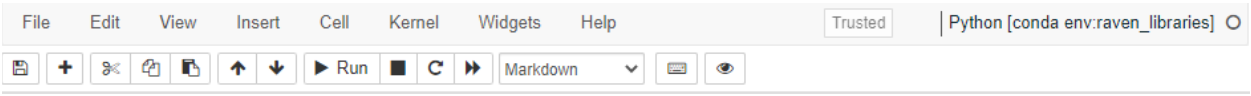


Figure 19. Jupyter notebook ribbon, with active kernel in the top-right corner: `Python [conda env:raven_libraries]`.

The first step in running a RAVEN workflow in a Jupyter notebook is to locate the folder containing the RAVEN source code on the user's system. This can be hard coded or involve using a generic search technique, as seen in Figure 20. The path to the folder containing the RAVEN source code must be appended to the Python path so that the Jupyter Python kernel knows where to look for the `ravenframework` Python package. The path to the RAVEN input XML file should also be provided.

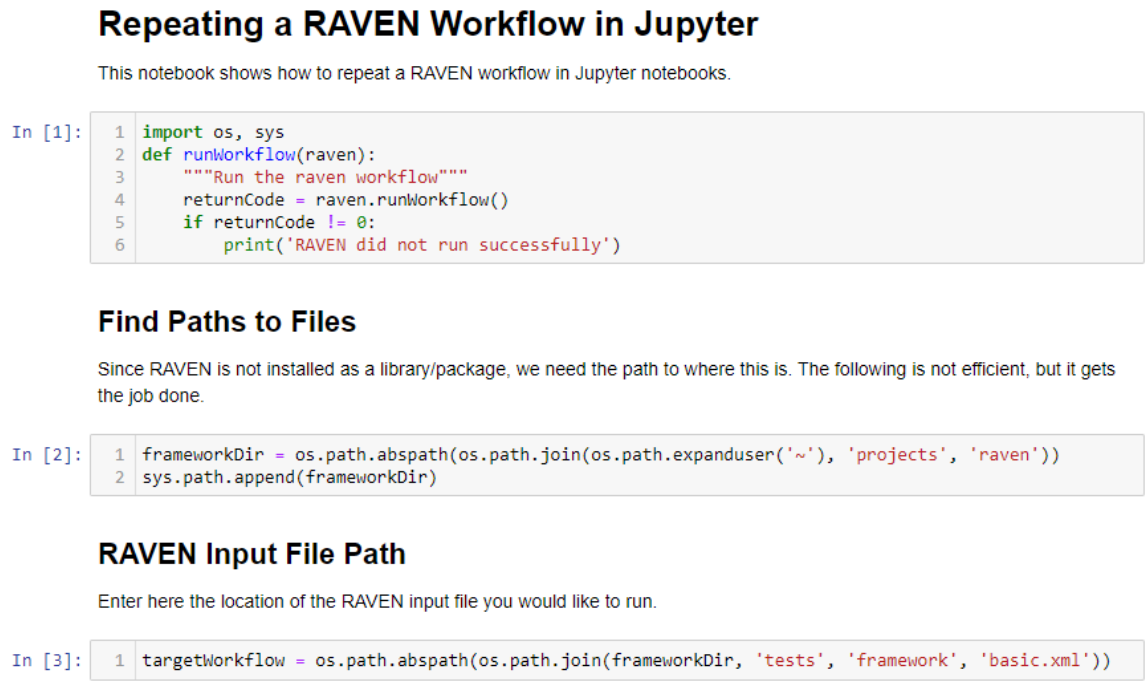


Figure 20. Jupyter notebook setup to run a RAVEN workflow.

Now that all the necessary setup is complete, the steps for running a RAVEN workflow can be found in Figure 21, Figure 22, and Figure 23. A `Raven` object is instantiated in Figure 21. The RAVEN workflow input XML file is loaded using the `loadWorkflowFromFile` method given in Figure 22. Finally, the workflow is run using the convenience function `runWorkflow`, which runs the `runWorkflow` method from the `Raven` object, as seen in Figure 23.

Instantiate a RAVEN instance

In [4]:

```python
from ravenframework import Raven
raven = Raven()
```

```
Copyright 2017 Battelle Energy Alliance, LLC

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

      .---.        .------######       #####     ###   ###  ########  ###    ###
     /     \  __  /    --###  ###    ###  ###   ###   ###  ###       #####  ###
    / /     \(  )/    --###  ###    ###   ###  ###   ###  ######    ### ######
   //////   ' \/ `   --#######     #########  ###   ###  ###       ###  #####
  //// / // :    :   -###   ###   ###   ###    ######   ####      ###   ####
 // /   /  /`    '---###    ###  ###   ###      ###    ########  ###    ###
//          //..\
===========UU====UU=============================================================
           '//||\`
             ''``

RAVEN Python dependencies located and checked.
```

Figure 21. Instantiating a `Raven` object in a Jupyter notebook.

Load workflow XML file

```
In [5]:  1 raven.loadWorkflowFromFile(targetWorkflow)
```

```
Loading plugin "ExamplePlugin" at C:\Users\garrrd\projects\raven\plugins\ExamplePlugin
 ... successfully imported "ExamplePlugin" ...
Loading plugin "FARM" at C:\Users\garrrd\projects\raven\plugins\FARM
 ... successfully imported "FARM" ...
Loading plugin "HERON" at C:\Users\garrrd\projects\raven\plugins\HERON
 ... successfully imported "HERON" ...
Loading plugin "LOGOS" at C:\Users\garrrd\projects\raven\plugins\LOGOS
 ... successfully imported "LOGOS" ...
Loading plugin "SR2ML" at C:\Users\garrrd\projects\raven\plugins\SR2ML
 ... successfully imported "SR2ML" ...
Loading plugin "TEAL" at C:\Users\garrrd\projects\raven\plugins\TEAL
 ... successfully imported "TEAL" ...
```

```
C:\Users\garrrd\Miniconda3\envs\raven_libraries\lib\site-packages\shiboken2\files.dir\shibokensupport
\feature.py:142: DeprecationWarning: the imp module is deprecated in favour of importlib; see the mod
ule's documentation for alternative uses
  return original_import(name, *args, **kwargs)
```

```
Target file found at "C:\Users\garrrd\projects\raven\tests\framework\basic.xml"
(    0.00 sec) SIMULATION              : Message         -> Global verbosity level is "all"
(           ) UTILS                    : Message         -> importing module C:\Users\garrrd\project
s\raven\tests\framework\AnalyticModels\projectile
InputData: Using param spec "DataSet" to read XML node "PointSet.
InputData: Using param spec "DataSet" to read XML node "PointSet.
```

Figure 22. Loading a RAVEN input XML file in a Jupyter notebook.

Run the workflow

```
In [6]:  1 runWorkflow(raven)
```

```
(    0.00 sec) SIMULATION              : Message         -> Simulation started at 2022-06-15 15:24:2
0
(    0.00 sec) SIMULATION              : Message         -> -- Beginning MultiRun step "sample" ...
--
(    0.00 sec) STEP MULTIRUN           : Message         -> ***  Beginning initialization ***
(    0.01 sec) SAMPLER MONTECARLO      : Message         -> No restart for SAMPLER MONTECARLO
(    0.01 sec) STEP MULTIRUN           : Message         -> ***    Initialization done    ***
(    0.01 sec) STEP MULTIRUN           : Message         -> ***       Beginning run       ***
(    0.37 sec) STEP MULTIRUN           : Message         -> ***       Run finished        ***
(    0.37 sec) STEP MULTIRUN           : Message         -> ***     Closing the step      ***
(    0.37 sec) STEP MULTIRUN           : Message         -> ***        Step closed        ***
(    0.37 sec) SIMULATION              : Message         -> -- End step sample of type: MultiRun --

(    0.38 sec) SIMULATION              : Message         -> -- Beginning IOStep step "plot" ... --
(    0.38 sec) STEP IOCOMBINED         : Message         -> ***  Beginning initialization ***
(    0.40 sec) STEP IOCOMBINED         : Message         -> ***    Initialization done    ***
(    0.40 sec) STEP IOCOMBINED         : Message         -> ***       Beginning run       ***
(    0.62 sec) STEP IOCOMBINED         : Message         -> ***       Run finished        ***
(    0.63 sec) STEP IOCOMBINED         : Message         -> ***     Closing the step      ***
(    0.64 sec) STEP IOCOMBINED         : Message         -> ***        Step closed        ***
(    0.64 sec) SIMULATION              : Message         -> -- End step plot of type: IOStep --

(    0.65 sec) SIMULATION              : Message         -> Run complete!
```

Figure 23. Running a RAVEN workflow in a Jupyter notebook.

To show that the same results are achieved when rerunning a RAVEN workflow in a Jupyter notebook, Figure 24 includes some additional code. In the [7] block, the workflow has been run once and the file name modified for comparison to the original output file after the workflow is rerun. The workflow is rerun using the convenience function `runWorkflow`, defined inside this notebook, but could also be rerun using the `runWorkflow` method of the `Raven` object. The [9] block gives some additional code to read the first and second output files, perform a comparison, then print out whether the two files contain the same information or not. Figure 24 shows that not only can a RAVEN workflow be run in a Jupyter notebook, it can also be rerun with the same inputs and give the same outputs. This shows that the Jupyter notebook process is repeatable and stable, opening up future development opportunities in which inputs can be changed dynamically without reloading the input XML file, and thus unlocking the potential for faster-than-real-time execution.

Show Same Results

For the `basic.xml` test, change the output file name to show that you get the same result.

In [7]:
```
printer = raven.getEntity('OutStreams', 'to_file')
printer.name = 'to_file2'
printer._printer.name = 'to_file2'
```

Run again

In [8]:
```
runWorkflow(raven)
```

```
(    0.00 sec) SIMULATION               : Message         -> Simulation started at 2022-06-15 15:24:2
0
(    0.00 sec) SIMULATION               : Message         -> -- Beginning MultiRun step "sample" ...
--
(    0.00 sec) STEP MULTIRUN            : Message         -> ***  Beginning initialization ***
(    0.00 sec) SAMPLER MONTECARLO       : Message         -> No restart for SAMPLER MONTECARLO
(    0.01 sec) STEP MULTIRUN            : Message         -> ***    Initialization done    ***
(    0.01 sec) STEP MULTIRUN            : Message         -> ***       Beginning run       ***
(    0.34 sec) STEP MULTIRUN            : Message         -> ***       Run finished        ***
(    0.34 sec) STEP MULTIRUN            : Message         -> ***     Closing the step      ***
(    0.34 sec) STEP MULTIRUN            : Message         -> ***        Step closed        ***
(    0.35 sec) SIMULATION               : Message         -> -- End step sample of type: MultiRun --

(    0.35 sec) SIMULATION               : Message         -> -- Beginning IOStep step "plot" ... --
(    0.35 sec) STEP IOCOMBINED          : Message         -> ***  Beginning initialization ***
(    0.38 sec) STEP IOCOMBINED          : Message         -> ***    Initialization done    ***
(    0.38 sec) STEP IOCOMBINED          : Message         -> ***       Beginning run       ***
(    0.60 sec) STEP IOCOMBINED          : Message         -> ***       Run finished        ***
(    0.60 sec) STEP IOCOMBINED          : Message         -> ***     Closing the step      ***
(    0.60 sec) STEP IOCOMBINED          : Message         -> ***        Step closed        ***
(    0.60 sec) SIMULATION               : Message         -> -- End step plot of type: IOStep --

(    0.61 sec) SIMULATION               : Message         -> Run complete!
```

Get the Same Results

Prove that the results from the first and second run are the same.

In [9]:
```
import pandas as pd
run1 = pd.read_csv('to_file.csv')
run2 = pd.read_csv('to_file2.csv')

if not (run1[run1.columns].equals(run2[run1.columns]) and
        run2[run2.columns].equals(run1[run2.columns])):
    print('Did not get correct values')
else:
    print('Same files written')
```

```
Same files written
```

Figure 24. Rerunning a RAVEN workflow in a Jupyter notebook and generating the same results.

3.2 Deep Lynx Integration

The time-series sensor data from the TEDS experiment were used to demonstrate the Deep Lynx data transformation. Table 3 lists the time-series sensor data obtained from the TEDS experiment, along with their metatype assigned to them for storage in Deep Lynx. The sensor name abbreviations in the table are defined as follows: Dynamic Energy Transport and Integration Laboratory (DETAIL), differential pressure meter (DP), flow meter (FM), globe valve (GBV), set point (SP), pressure transducer (PT), thermal energy storage (TES), and high-temperature steam electrolysis (HTSE). Raw experimental data were converted into CSV format, and sensor ID columns were created for the type-mapping process. Next, the uploaded CSV file was mapped to the graph structure, consisting of nodes, edges, and time-series tables. Figure 25 shows a graphical view of the mapped data. Each sensor is represented as a node, and a time-series table is attached to each node. In the figure, each metatype is given a different color (e.g., ControlValve is purple), and the relationships between sensors are expressed by arrow symbols. Three relationship types were used for edges. Adjacent sensors are represented as "relates" to each other, sensors or other entities directly connected to other sensors are represented as "affectedBy," and the parent relation of an entity is represented as "decomposes." When the data are queried from the graph database, time-series data are output in the JavaScript Object Notation (JSON) format, as shown in Figure 26.

Table 3. List of sensor data in the TEDS experiment.

Sensor	Metatype	Sensor	Metatype
Ambient Temp	Thermometer	Master Alarm	Warning
DETAIL Control	Switch	Master Warning	Warning
DP-001	DifferentialPressureMeter	PT-001	PressureTransducer
FM-001	FlowMeter	PT-201	PressureTransducer
FM-002	FlowMeter	Pump Frequency	Pump
FM-003	FlowMeter	Pump SP	Pump
FM-201	FlowMeter	TC-001	Thermocouple
FM-202	FlowMeter	TC-002	Thermocouple
GBV-002	ControlValve	TC-003	Thermocouple
GBV-004	ControlValve	TC-004	Thermocouple
GBV-006	ControlValve	TC-005	Thermocouple
GBV-008	ControlValve	TC-006	Thermocouple
GBV-009	ControlValve	TC-201	Thermocouple
GBV-012	ControlValve	TC-202	Thermocouple
GBV-201	ControlValve	TEDS Power Input	ElectricalPowerSystem
GBV-202	ControlValve	TEDS Power Output	ElectricalPowerSystem
GBV-203	ControlValve	TEDS	System
GBV-204	ControlValve	TES	System
Heater SP	ElectricHeater	HTSE	System
Heater Temp	ElectricHeater		

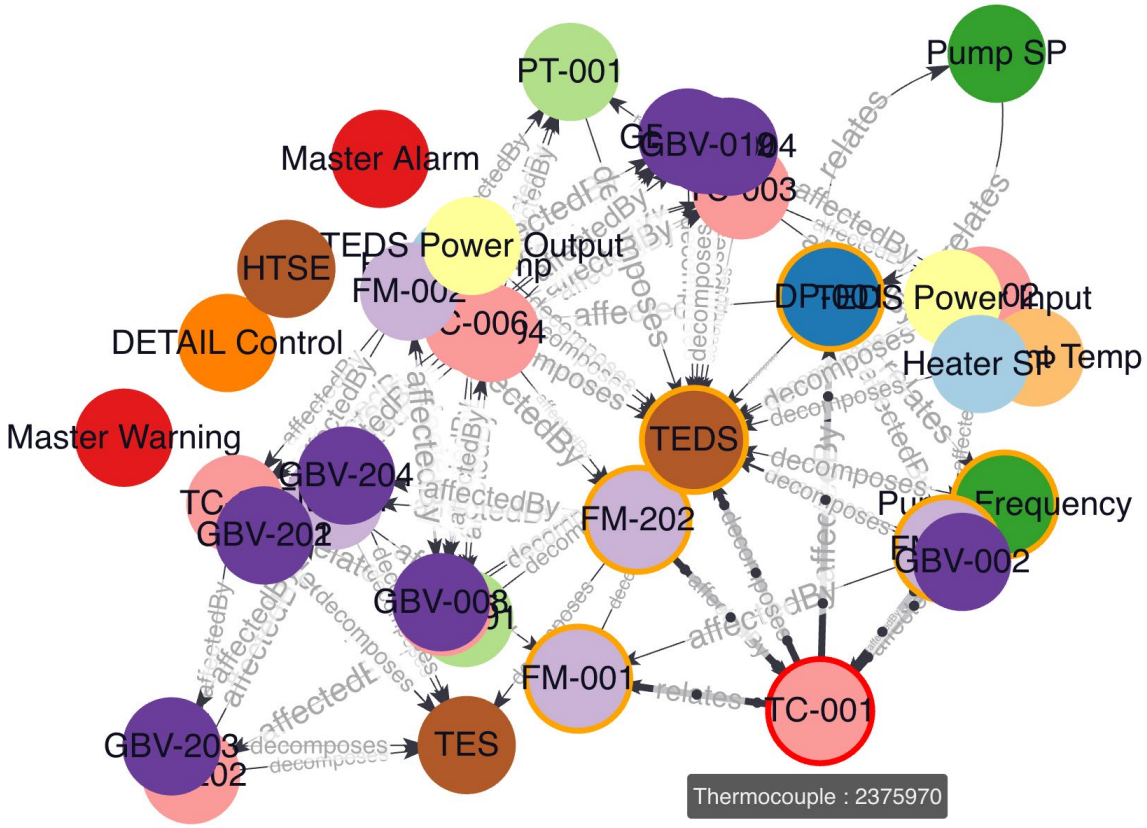


Figure 25. Graphical view of the mapped graph database.


```
{
    "data": {
        "TC001": [
            {
                "TC001": "68.362981",
                "Time": "1293849000000",
                "_record": {
                    "count": null
                }
            },
            {
                "TC001": "70.380862",
                "Time": "1293868800000",
                "_record": {
                    "count": null
                }
            },
            {
                "TC001": "70.003984",
                "Time": "1293865200000",
                "_record": {
                    "count": null
                }
            },
            {
                "TC001": "67.832335",
                "Time": "1293843600000",
                "_record": {
                    "count": null
                }
            },
```


Figure 26. Queried data in JSON format.

3.3 Real-time Optimization Workflow

A demonstration of an electrical storage device RTO workflow was developed using the parameters in Table 4, with the naming convention seen in Section 2.3. S_0, which did not appear in Section 2.3, represents the initial state of charge of the electrical storage device in MWh. Δt was chosen to be 5 minutes because that is the frequency of the real-time market LMP for the Pennsylvania-New Jersey-Maryland (PJM) Interconnection.[33] PJM LMP data from May 31, 2022, to June 7, 2022, for Pricing Node 1 (PJM-RTO) was downloaded to a CSV file for use in the RTO workflow. The LMP for this time period shows a number of sharp price increases that are attractive for energy arbitrage (see Figure 27).

Table 4. Parameters used in the RTO workflow.

Parameter	Value
Δt	5 minutes
γ_{RTE}	0.8
q_{NPP}	50 MW
S_{max}	20 MWh
S_0	0 MWh
q^C_{max}	20 MW
q^D_{max}	20 MW

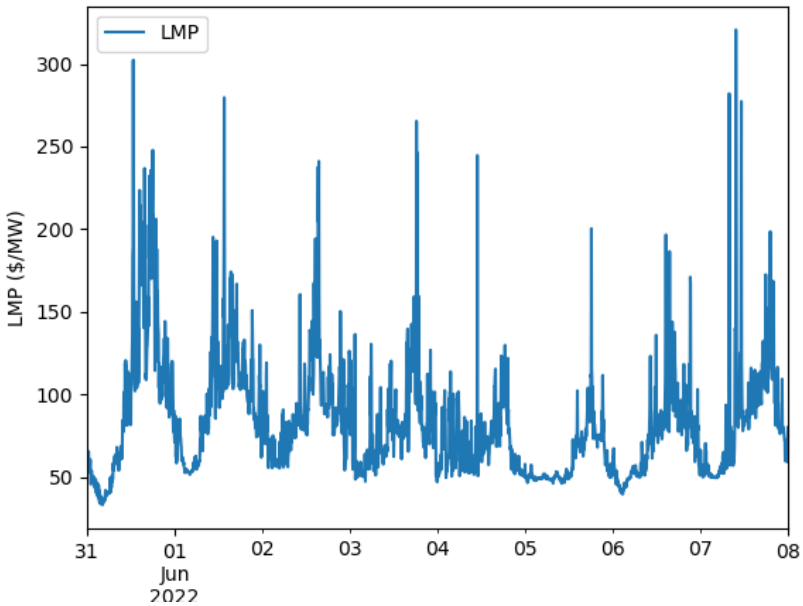


Figure 27. LMP from PJM Pricing Node 1.

The dispatch optimization portion of the RTO workflow was set up as a LP problem. The open-source optimization modeling language package Pyomo[23,24,25] was used to solve the LP problem. Pyomo is a Python package that allows developers to write LP, mixed integer programming, or NLP in a standard way and then connect the problem to an external solver. The variables and parameters of the LP problem are defined using Pyomo objects. Relations—including equations, inequalities, or other mathematical relationships—define how the variables and parameters are connected to each other in constraints or the objective function. These relations are explicitly implemented within Pyomo as algebraic expressions. Once all the variables, parameters, and relations are defined, Pyomo sends the LP problem to an external solver such as CBC[34,35] or GLPK.[36]

The process of the RTO workflow using EMPC and LP optimization is shown in the following Jupyter notebook sections. LMP data from PJM are used, along with the parameters from Table 4. For the EMPC process, a 12-hour time horizon from the downloaded LMP data gives a perfect knowledge forecast with 144 time steps. For a real-world implementation, a forecasting method would need to be implemented, as the actual future LMP data are unknown. Since only the charging and discharging inputs for the next time step are required, it is not too important to have a highly accurate LMP forecast. Simply knowing the direction of the LMP movements will suffice.

The necessary Python packages are imported at the beginning of the Jupyter notebook, as seen in Figure 28. The package `time` is imported to determine how long the workflow will run. `pandas` and `numpy` are used to load the PJM LMP data from a CSV file and store the results. Pyomo is imported to set up and solve the LP problem. Matplotlib is imported to build plots. The final line of code sets the solver to be CBC. A different available solver (e.g., GLPK) could also have been chosen to solve the LP problem.

```
import time
import pandas as pd
import numpy as np
import pyomo.environ as pyo
import matplotlib.pyplot as plt

opt = pyo.SolverFactory('cbc')
```

Figure 28. Jupyter notebook setup to run the RTO workflow.

The section of code in Figure 29 shows how the PJM LMP data are loaded from a saved CSV file by using `pandas`. For convenience, the LMP data are reshaped and renamed. At the end of this section, the LMP data are plotted, resulting in Figure 27 above.

```
# load PJM LMP data
lmp = pd.read_csv('rt_fivemin_hrl_lmps.csv')
# simplify dataset to just time and LMP
lmp = lmp[['datetime_beginning_ept', 'total_lmp_rt']]
lmp = lmp.rename(columns={'datetime_beginning_ept': 'time', 'total_lmp_rt': 'LMP'})
lmp['time'] = pd.to_datetime(lmp['time'])
lmp = lmp.set_index('time')
# plot the LMP data for fun
ax = lmp.plot()
ax.set_ylabel('LMP ($/MW)')
# plt.savefig('LMP.png')
```

Figure 29. Loading LMP data into a Jupyter notebook for the RTO workflow.

For convenience, two constraint functions are defined in Figure 30. The first is the state-of-charge model defined in Section 2.3. The second implements the final constraint from Section 2.3, which is that the electric storage device cannot discharge more energy than is currently stored. Both constraint functions were defined using a Pyomo `ConcreteModel` object named `model`—containing the variables, parameters, and relations—and the time step index variable `t`.

```python
def state_of_charge(model, t):
    """State of charge increment"""
    if t == model.t.first():
        # initial storage level
        return model.SOC[t] == model.SOCbegin
    else:
        # current level
        return model.SOC[t] == model.SOC[t-1] + (model.sqrtgammaRTE*model.qC[t-1] -
                                                 model.qD[t-1]/model.sqrtgammaRTE)*model.dt/60.0

def discharge(model, t):
    """Discharge cannot be greater than what is stored"""
    if t == model.t.first():
        return model.qD[t] <= model.sqrtgammaRTE*model.SOCbegin*60.0/model.dt
    else:
        return model.qD[t] <= model.sqrtgammaRTE*model.SOC[t]*60.0/model.dt
                              + model.gammaRTE*model.qC[t]
```

Figure 30. Constraint functions implemented in a Jupyter notebook for the RTO workflow.

The next section of code, shown in Figure 31, defines the Pyomo `ConcreteModel` and solves the LP problem. A Python dictionary containing the specifications for the LP is passed into the `solve_model` function as `specs`. If the model has been defined previously and only needs updating, it can be passed in as well. The number of time steps was determined by dividing the specified `t_window` (in minutes) by the time step length `dt` (in minutes). For this workflow demonstration, `t_window` was set to 720 minutes (12 hours), and the time step length `dt` was set to 5 minutes, resulting in 144 time steps. Each of the system parameters from Table 4 was then implemented, as well as the time index. The LMP data were also initialized as a parameter. The variables for charging, discharging, and state of charge were then set up. The two constraint functions defined in Figure 30 were added to the model. The objective function, defined in Section 2.3, was implemented to complete the Pyomo LP model. If the model had been defined previously, the LMP values and initial state of charge of the electrical storage device had to be updated. Finally, the model was then solved.

```python
def solve_model(specs, model=None):
    """Set up and solve pyomo model

    Parameters
    ----------
        specs : dict
            dictionary containing parameters for solve
        model : pyomo.environment.ConcreteModel
            previously solved pyomo model

    Returns
    -------
        model : pyomo.environment.ConcreteModel, optional
            solved pyomo model
    """
    # parameters needed for solve
    n = int(specs['t_window']/specs['dt']) # number of steps to take

    if model is None:
        # pyomo setup
        model = pyo.ConcreteModel()

        # system parameters
        model.dt = pyo.Param(initialize=specs['dt'])
        model.qNPP = pyo.Param(initialize=specs['q_NPP'])
        model.Smax = pyo.Param(initialize=specs['S_max'])
        model.qDmax = pyo.Param(initialize=specs['q_Dmax'])
        model.qCmax = pyo.Param(initialize=specs['q_Cmax'])
        model.gammaRTE = pyo.Param(initialize=specs['gamma_RTE'])
        model.sqrtgammaRTE = pyo.Param(initialize=np.sqrt(specs['gamma_RTE']))
        model.SOCbegin = pyo.Param(initialize=specs['S_init'], mutable=True)

        # time index
        model.t = pyo.Set(initialize=np.arange(0, n, dtype=int))

        # LMP price
        model.P = pyo.Param(model.t,
                            initialize={x: specs['lmp']['LMP'].values[x] for x in model.t},
                            mutable=True)

        # variables
        model.qD = pyo.Var(model.t, domain=pyo.NonNegativeReals, bounds=(0.0, model.qDmax))
        model.qC = pyo.Var(model.t, domain=pyo.NonNegativeReals, bounds=(0.0, model.qCmax))
        model.SOC = pyo.Var(model.t, domain=pyo.NonNegativeReals, bounds=(0.0, model.Smax))

        # constraints
        model.state = pyo.Constraint(model.t, rule=state_of_charge)
        model.discharge = pyo.Constraint(model.t, rule=discharge)

        # objective
        obj = sum(model.P[t]*(model.qNPP - model.qC[t] + model.qD[t]) for t in model.t)
        model.objective = pyo.Objective(rule=obj, sense=pyo.maximize)
    else:
        # don't need to set up problem again, just modify the LMP data and SOCbegin
        for x in model.t:
            model.P[x] = specs['lmp']['LMP'].values[x]
        model.SOCbegin = specs['S_init']

    results = opt.solve(model)

    return model
```

Figure 31. Pyomo `ConcreteModel` setup for the RTO workflow in the Jupyter notebook.

To get the optimized dispatch of the next time step from a solved LP model, the function `gen_next_dispatch` is defined, as shown in Figure 32. This function takes in the Python dictionary `specs` and, optionally, the Pyomo `ConcreteModel` model. The `solve_model` function defined in Figure 31 solves the LP problem; pulls the optimal charge, discharge, and state of charge for the next time step from the solved model; and returns these values along with the solved model.

```python
def get_next_dispatch(specs, model=None):
    """Gets dispatch of next time step using receding horizon optimization

    Parameters
    ----------
        specs : dict
            dictionary containing parameters for solve
        model : pyomo.environ.ConcreteModel
            previously solved pyomo model

    Returns
    -------
        qD : float
            dispatch discharged from storage to real time market (MW)
        qC : float
            dispatch to storage (charging) (MW)
        SOC : float
            state of charge (MWh)
        model : pyomo.environ.ConcreteModel
            solved pyomo model
    """
    model = solve_model(specs, model=model)
    qD = pyo.value(model.qD[1])
    qC = pyo.value(model.qC[1])
    SOC = pyo.value(model.SOC[1])

    return qD, qC, SOC, model
```

Figure 32. Function to retrieve the next time step's optimal values for the EMPC approach to the RTO workflow implemented in a Jupyter notebook.

To demonstrate solving the LP problem via the methods described above, the next section of code (see Figure 33) shows how the parameters from Table 4 are implemented as the Python dictionary `specs`. The initial instance of the LP problem is solved using the function `solve_model`. The results of the solution to this single instance of the LP problem are shown in Figure 34. The blue circles in the top subfigure show the electrical power from the NPP going to the real-time market. The orange squares and green triangles are the power from the NPP charging the electrical storage device and the power discharged from the storage device to the real-time market, respectively. As expected, the electrical storage device charges when the LMP is relatively low, and discharges near the end of the 12-hour period, when the LMP is higher. Another feature of this single-instance LP solve is that the electrical storage device completely discharges before the end of the 12-hour period. With perfect foresight, the optimal arbitrage solution will leave nothing remaining in the storage device. This is avoided for the RTO workflow thanks to the receding horizon approach from EMPC. With the rolling time horizon seeking only the inputs for the next time step, the electrical storage device will charge, discharge, or patiently wait for the optimal time to take an action, rather than fully discharge by the end of the time horizon.

```python
specs = {
    't_window': 60*12, # forward projecting time window (minutes)
    'dt': 5,           # time step (minutes)
    'q_NPP': 50.0,     # Nuclear power plant capacity (MW)
    'S_max': 20.0,     # energy capacity (MWh)
    'q_Dmax': 20.0,    # maximum discharge power (MW)
    'q_Cmax': 20.0,    # maximum charging power (MW)
    'gamma_RTE': 0.8,  # round trip efficiency
    'S_init': 0.0,     # initial state of charge (MWh)
    'lmp': lmp         # Local Marginal Price ($/MW)
}

model2 = solve_model(specs)
```

Figure 33. Solving a single LP instance in a Jupyter notebook.

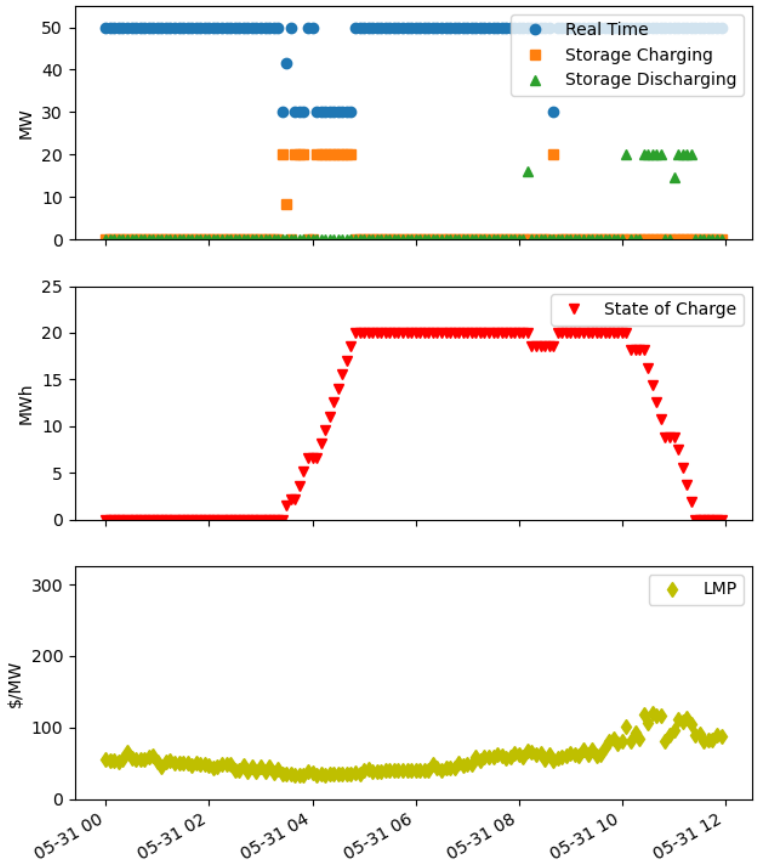


Figure 34. Results from the initial LP instance from the RTO workflow.

The final section of code in Figure 35 implements the EMPC approach to the RTO workflow. The workflow runs for the week beginning on May 31, 2022, and ending on June 7, 2022. The parameters from Table 4 were again implemented in the Python dictionary `specs`. For each time step, the LMP for the 12-hour period beginning with the current time step is updated, and `get_next_dispatch` is called to get the optimal charge, discharge, and state-of-charge dispatch by solving the LP problem. These values are stored in a `pandas DataFrame`. When moving to the next time step, the initial state of charge is set to the state of charge of the current time step, and the time index is incremented. When the results for the entire week are collected, the code prints out how long it took to run. For this demonstration case, the workflow took 410.78 seconds to complete on a computer with a Windows operating system.

```
tic = time.time()
ind = 0
lmp2 = lmp.reset_index()
end_date = pd.to_datetime('2022-06-07 00:05:00')
specs = {
    't_window': 60*12, # forward projecting time window (minutes)
    'dt': 5, # time step (minutes)
    'q_NPP': 50.0, # Nuclear power plant capacity (MW)
    'S_max': 20.0, # storage capacity (MWh)
    'q_Dmax': 20.0, # maximum discharge power (MW)
    'q_Cmax': 20.0, # maximum charging power (MW)
    'gamma_RTE': 0.8, # round trip efficiency
    'S_init': 0.0, # initial state of charge (MWh)
    'lmp': lmp2 # Local Marginal Price ($/MW)
}
n = int(specs['t_window']/specs['dt'])
model = None
while lmp2['time'][ind] < end_date:
    print(f'ind: {ind} time: {lmp2["time"][ind]}')
    # forward price information
    specs['lmp'] = lmp2[ind:ind+n]

    # get next time step dispatch
    qD, qC, SOC, model = get_next_dispatch(specs, model=model)
    qRT = specs['q_NPP'] - qC

    # store new results
    new_df = pd.DataFrame(data={'time': [lmp2['time'][ind+1]],
                                'qRT': [qRT],
                                'qD': [qD],
                                'qC': [qC],
                                'SOC': [SOC],
                                'LMP': [lmp2['LMP'][ind+1]]})
    if ind == 0:
        result_df = new_df
    else:
        result_df = pd.concat([result_df, new_df], ignore_index=True)

    # move to next time step
    specs['S_init'] = SOC
    ind += 1
print(f'Total solve time: {time.time() - tic} s')
```

Figure 35. EMPC methodology for the RTO workflow in a Jupyter notebook.

Figure 36 shows the results of running the code from Figure 35. As with Figure 34, the top subfigure shows the electricity dispatched from the NPP to the real-time market (in blue), the charging power from the NPP to the electrical storage device (in orange), and the electricity discharged from the electrical storage device to the real-time market (in green). The state of charge of the electrical storage device is shown in the middle subfigure, and the LMP is shown in the bottom subfigure. As expected, the spikes in LMP correlate with discharging electricity from the electrical storage device to the real-time market. Times of low LMP reflect when the electrical storage device is charged. For comparison, the revenue earned over this 1-week period was computed for a NPP operating at a constant 50 MW and for the integrated NPP-electrical storage device system in the real-time market. In looking at the standalone NPP, the revenue was $8,265,764.33. For the integrated NPP-electrical storage device system, the revenue was $8,511,996.26. Including the electrical storage device and using the RTO workflow for energy arbitrage increased the week's worth of revenue by $246,231.93.

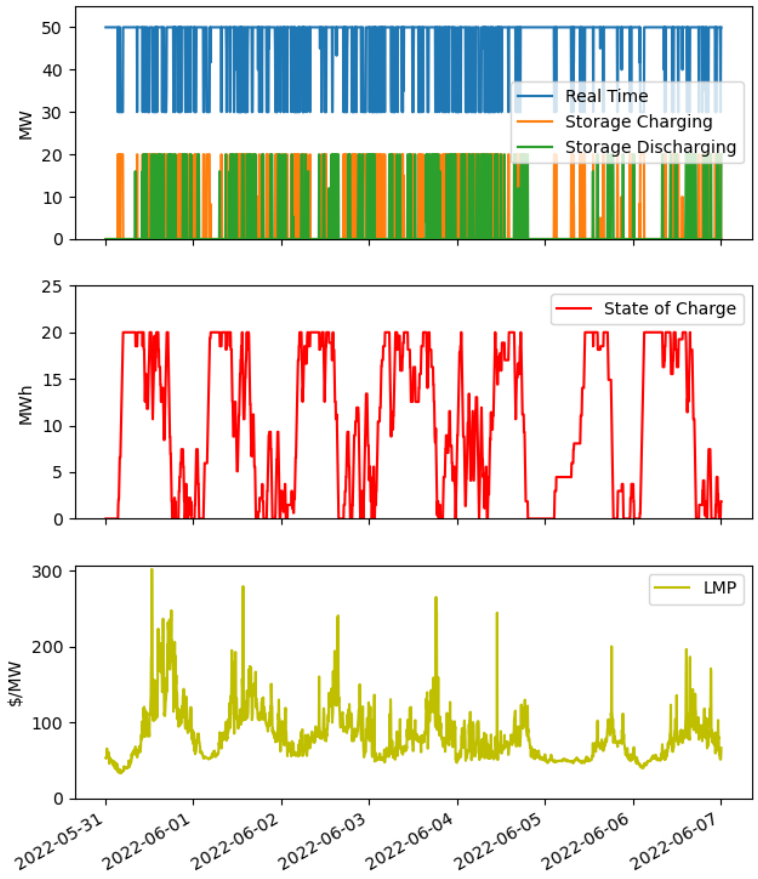


Figure 36. Results of running the EMPC-based RTO workflow for 1 week.

3.4 Grey-Box Modeling

3.4.1 External System-level Simulation Model in Modelica

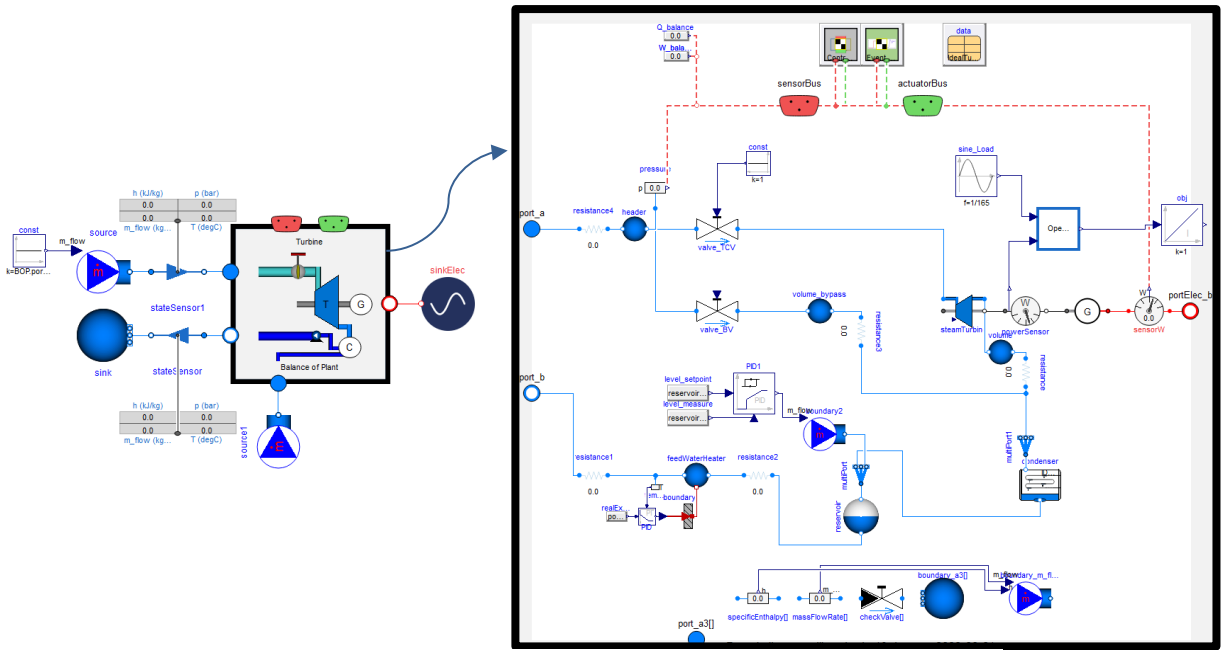
Idaho National Laboratory has been working to develop its Nuclear Hybrid Energy Systems (NHES) package,[37,38,39,40] a library of high-fidelity process models in Dymola,[41] which is a commercial modeling and simulation environment based on Modelica.[22] Modelica is an object-oriented, equation-based language used to model complex CPSs containing mechanical, electrical, electronic, hydraulic, thermal, control, electric power, or process-oriented subcomponents. It is an inherently time-dependent modeling language that enables swift interconnection of independently developed models.[42]

The NHES library employs both third-party components from the Modelica Standard Library and the Transient Simulation Framework of Reconfigurable Modules (TRANSFORM),[43] as well as components developed internally to the project for specific subsystems. For example, the NHES library contains a large variety of models for developing a high-temperature steam electrolysis plant, a gas turbine, a single-stage Rankine cycle balance of plant (BOP), and a light-water nuclear reactor. In this milestone report, a single-stage Rankine cycle BOP is used to demonstrate optimization problem solving.

First, the NHES package must be installed. It can be downloaded from the HYBRID[44] GitHub repository (https://github.com/idaholab/HYBRID). After successfully installing the package, a single-stage Rankine cycle BOP can be found at:

```
NHES.Systems.BalanceOfPlant.Turbine.SteamTurbine_L1_boundaries
```

Figure 37 shows the Dymola models in graphics mode: a steam turbine testing model (top left), which contains the steam turbine model, and a Rankine cycle BOP model (top right). Note that three additional components are added to the BOP model. `Modelica.Blocks.Sources.Sine` and `Modelica.Blocks.Continuous.Integrator` components are used, along with a newly defined math block for calculating the objective function. In the top-right of Figure 37, one finds that the `Sine` component replicates the external electricity load demand, while the math block receives two inputs from the `sine` and `powerSensor` components and sends an output to the `Integrator` component. In complying with the variable naming convention in the RAVEN framework (no commas, no spaces in a variable name), use of a `combiTable` component in Dymola is limited. Rather than using any type of table component in Dymola, a vectorization approach for the time-dependent RAVEN input was introduced. The Modelica code snippet developed for the vectorization of time and input variables is also shown. The full Modelica codes of the steam turbine and BOP models are given in Appendix B.




```
  parameter Integer n = 11;
  Integer count(start=1);

  //*************************************************//
  //New vectorization for time-dependent RAVEN input;
  parameter Real valve_area_vec[n] = {0.20,0.15,0.12,0.08,0.05,
                                      0.05,0.07,0.10,0.15,0.21,0.33};

  parameter Real time_setpoint_vec[n] = {30,40, 50, 60, 70,
                                         80,90,100,110,120,130};
  Real time_step;
  //*************************************************//

initial equation
  time_step = time_setpoint_vec[1];

equation
  when time>time_step then
    count = pre(count)+1;
    time_step = time_setpoint_vec[count];
  end when;
    valve_BV.opening = valve_area_vec[count];
```

Figure 37. Steam turbine and BOP Modelica models.

3.4.2 External Simulation Model Interface with RAVEN

This section shows key code blocks of a RAVEN input XML file that interfaces with a Modelica model and constructs a ROM. The full RAVEN input file used in this section can be found in Appendix C.

The first section to point out in the RAVEN input XML file is the <Steps> node. The <Steps> node consists of multiple sub-nodes to detail each step in the RAVEN workflow (Figure 38):

1. <MultiRun>, named "sample", is used to run the multiple instances of the driven code and to collect the output in a DataObjects node. <Sampler> is inputted to communicate to the Step that the driven code must be perturbed via the MonteCarlo sampling strategy.
2. <RomTrainer>, named "trainROM", is used to construct the ROM by using a DataObjects node called samples. Note that samples is the output from <MultiRun> and becomes the input for <RomTrainer>.
3. Two inputs are required to solve an optimization problem using PyNumero: (1) a Python script file for optimization and (2) a pickled ROM file. Two <IOStep> blocks work separately to create these inputs. Note that both <IOStep name="serialize"> and <IOStep name="dumpROM"> receive as input a ROM model called ROM, which is the output from the <Models> node.

```
<Simulation>
  ...
  <Steps>
    <MultiRun name="sample">
      <Input class="Files" type="DymolaInitialisation">BOPInput</Input>
      <Model class="Models" type="Code">BOP</Model>
      <Sampler class="Samplers" type="MonteCarlo">my_mc</Sampler>
      <Output class="DataObjects" type="PointSet">samples</Output>
      <Output class="DataObjects" type="HistorySet">histories</Output>
    </MultiRun>
    <MultiRun name="sampleROM">
      <Input class="DataObjects" type="PointSet">inputPlaceHolder</Input>
      <Model class="Models" type="ROM">ROM</Model>
      <Sampler class="Samplers" type="MonteCarlo">my_mc</Sampler>
      <Output class="DataObjects" type="PointSet">samplesROM</Output>
    </MultiRun>
    <RomTrainer name="trainROM">
      <Input class="DataObjects" type="PointSet">samples</Input>
      <Output class="Models" type="ROM">ROM</Output>
    </RomTrainer>

    <IOStep name="serialize">
      <Input class="Models" type="ROM">ROM</Input>
      <Output class="Files" type="">rom_out.py</Output>
    </IOStep>

    <IOStep name="dumpROM">
      <Input class="Models" type="ROM">ROM</Input>
      <Output class="Files" type="">rom_pickle</Output>
    </IOStep>
      <IOStep name="writeHistories" pauseAtEnd="True">
      <Input class="DataObjects" type="HistorySet">histories</Input>
      <Input class="DataObjects" type="PointSet">samples</Input>
      <Output class="OutStreams" type="Print">histories</Output>
      <Output class="OutStreams" type="Print">samples</Output>
    </IOStep>
  </Steps>
  …
</Simulation>
```

Figure 38. Steps node of the RAVEN input XML for Pyomo integration with Dymola.

When a Modelica model is implemented in Dymola, the platform-dependent C-code from the Modelica model and the corresponding executable code (i.e., `dymosim.exe` on Windows) are generated for conducting the simulation. `dsin.txt`, which contains model parameters and initial conditions, is also generated as part of the build process. Since the RAVEN-Dymola interface modifies input parameters by changing copies of this text file, the names of both the executable and text file (by default, `dymosim.exe` and `dsin.txt`, respectively) must be provided to RAVEN. In the `<Files>` node, the location of the initialization file (`dsin.txt`), the name of Python script file, and the name of the pickled file that will be saved are all specified to RAVEN. The `<Files>` node of the RAVEN input file is shown in Figure 39.

```
<Simulation>
  ...
  <Files>
    <Input name="BOPInput" type="DymolaInitialisation">C:\Users\<UserName_Here>\Documents\Dymola
\dsin.txt</Input>
    <Input name="rom_out.py" type="Pyomo">rom_out.py</Input>
    <Input name="rom_pickle" type="">BOP.pk</Input>
  </Files>
  ...
</Simulation>
```

Figure 39. Files node of the RAVEN input XML for Pyomo integration with Dymola.

In the `<Models>` node shown in Figure 40, the simulation code to be used and the ROM construction are both specified. A list of variables that users can access in the Modelica model is shown in the `[ # Names of initial variables ]` section in `dsin.txt`. To refer to variables from `dsin.txt` that are given different names inside of RAVEN, an `<alias>` node can be used for each variable. In this demonstration, the N-dimensional inverse distance weighting algorithm `"NDinvDistWeight"` is used to construct a ROM with 10 feature variables for the turbine bypass valve (TBV) opening area at different time steps, as well as one target variable (i.e., the objective function value, `BOP.obj.y`, defined in the Modelica model). The subtype of the ROM model can be that of any available RAVEN ROM. The ROM selection will affect the computational time when running the optimization problem in the PyNumero framework.

```
<Simulation>
  ...
  <Models>
    <Code name="BOP" subType = "Dymola">
      <executable> c:/Users/<User_ID>/Documents/Dymola/dymosim.exe</executable>
      <alias type="input" variable="valve_area_t30_40">BOP.valve_area_vec[2]</alias>
      <alias type="input" variable="valve_area_t40_50">BOP.valve_area_vec[3]</alias>
      <alias type="input" variable="valve_area_t50_60">BOP.valve_area_vec[4]</alias>
      <alias type="input" variable="valve_area_t60_70">BOP.valve_area_vec[5]</alias>
      <alias type="input" variable="valve_area_t70_80">BOP.valve_area_vec[6]</alias>
      <alias type="input" variable="valve_area_t80_90">BOP.valve_area_vec[7]</alias>
      <alias type="input" variable="valve_area_t90_100">BOP.valve_area_vec[8]</alias>
      <alias type="input" variable="valve_area_t100_110">BOP.valve_area_vec[9]</alias>
      <alias type="input" variable="valve_area_t110_120">BOP.valve_area_vec[10]</alias>
      <alias type="input" variable="valve_area_t120_130">BOP.valve_area_vec[11]</alias>
    </Code>
    <ROM name="ROM" subType="NDinvDistWeight">
      <pivotParameter>Time</pivotParameter>
      <Features> valve_area_t30_40, valve_area_t40_50, valve_area_t50_60, valve_area_t60_70,
      valve_area_t70_80, valve_area_t80_90, valve_area_t90_100, valve_area_t100_110,
      valve_area_t110_120, valve_area_t120_130 </Features>
      <Target>BOP.obj.y</Target>
      <p>1</p>
    </ROM>
  </Models>
  ...
</Simulation>
```

Figure 40. Models node of the RAVEN input XML for Pyomo integration with Dymola.

3.4.3 Optimization in PyNumero

The optimization example shown here aims to match the demand load by controlling the TBV opening area to control how much electrical power is generated. This is done for a time domain of 0–130 seconds. The optimization portion occurs after 30 seconds so that the system can achieve near-steady-state operation. The load demand is modeled as a sine function. The objective function to be minimized is given as:

$$y = \int_{30}^{130} \left(\frac{l_t - g_t(x_t)}{1e8}\right)^2 dt \tag{7}$$

where l_t is the load demand at time t (W), x_t is the percent TBV opening area at time t, and $g_t(x_t)$ is the electricity generated at time t (W). The $1e8$ factor scales the objective function so that its magnitude is not too large. Scaling can be important for numerical solutions, especially if the optimization method is not scaling invariant.

In this example, the required Python packages were imported just as in Figure 11, the `ravenROM` class from Figure 12 was used, and the `pyomoModel` function from Figure 13 was applied to construct the Pyomo `ConcreteModel` by using `ExternalGreyBoxBlock` components. Setting up and solving the optimization problem required an additional five lines of code, shown in Figure 41. The grey box model was loaded using the `ravenROM` class. Next, the Pyomo `ConcreteModel` was built using the `pyomoModel` function. Finally, the solver was configured, and the model was solved in the final line.

```
ext_model = ravenROM(**{'rom_file':"C:/Users/KIMJ5/projects/raven/Rankin_Model/BOP.pk",
                        'raven_framework': 'C:/Users/KIMJ5/projects/raven'})
[12]                                                                          Python

concreteModel = pyomoModel(ext_model)
solver = pyo.SolverFactory('cyipopt')
solver.config.options['hessian_approximation'] = 'limited-memory'
results = solver.solve(concreteModel)
[12]                                                                          Python
```

Figure 41. Python code to solve the optimization problem in Pyomo, using a grey-box model from PyNumero.

Executing the command `concreteModel.pprint()` generated the PyNumero results shown in Figure 42. The optimized TBV opening areas at each time step are bordered by the red box, and the objective function is shown in the blue box. To visually show that the optimal values have been found, the optimal inputs were simulated in Dymola, and the resulting plots are shown in Figure 43. The top plot shows the time history of the TBV opening area, with the optimal TBV opening areas in red. The middle plot shows the time history of the load demand (in green), with the electricity generated via the optimal TBV opening areas (in red). The bottom plot shows the time history of the objective function, with the optimal TBV opening areas again shown in red. The optimized values brought the electricity generated close to the load demand, and the objective function with the optimal TBV opening areas was close to zero (zero being the optimal result). Were the time step decreased, the optimal solution would more closely match the load demand, since the TBV opening areas are constant for the duration of the time step.

```
1 Objective Declarations
    obj : Size=1, Index=None, Active=True
        Key  : Active : Sense    : Expression
        None :   True : minimize : egb.outputs['BOP.obj.y']

1 ExternalGreyBoxBlock Declarations
    egb : Size=1, Index=None, Active=True
        2 Set Declarations
            _input_names_set : Size=1, Index=None, Ordered=Insertion
                Key  : Dimen : Domain : Size : Members
                None :     1 :    Any :   10 : {'valve_area_t30_40', 'valve_area_t40_50', 'valve_area_t50_60', 'valve_area_t60_70', 'valv
            _output_names_set : Size=1, Index=None, Ordered=Insertion
                Key  : Dimen : Domain : Size : Members
                None :     1 :    Any :    1 : {'BOP.obj.y',}

        2 Var Declarations
                                                          Optimized input values
            inputs : Size=10, Index=egb._input_names_set
                Key                  : Lower          : Value            : Upper          : Fixed : Stale : Domain
                valve_area_t100_110 :  0.100022703595 :   0.15219250539 :  0.199890470109 : False : False :  Reals
                valve_area_t110_120 :  0.200052037649 :  0.219024168197 :   0.29997138076 : False : False :  Reals
                valve_area_t120_130 :  0.250003071874 :  0.332950145461 :  0.349971767492 : False : False :  Reals
                 valve_area_t30_40 :  0.100033251173 :  0.154657874271 :  0.199831875903 : False : False :  Reals
                 valve_area_t40_50 :  0.100000581797 :  0.120968757709 :  0.149975259404 : False : False :  Reals
                 valve_area_t50_60 : 0.0500093263225 : 0.0855123411679 : 0.0999924619682 : False : False :  Reals
                 valve_area_t60_70 : 0.0500200772192 : 0.0597456433164 :  0.099973034102 : False : False :  Reals
                 valve_area_t70_80 : 0.0500092598493 : 0.0525886075982 : 0.0999875780544 : False : False :  Reals
                 valve_area_t80_90 : 0.0500134698581 : 0.0783750020986 :  0.149955769791 : False : False :  Reals
                valve_area_t90_100 :  0.100011876156 :  0.103754730728 :  0.149985381879 : False : False :  Reals
            outputs : Size=1, Index=egb._output_names_set
                Key       : Lower          : Value           : Upper          : Fixed : Stale : Domain
                BOP.obj.y : 0.730359256268 : 0.740140140055 : 7.51498651505 : False : False :  Reals
                                                  Optimized output value

        4 Declarations: _input_names_set inputs _output_names_set outputs

2 Declarations: egb obj
```

Figure 42. PyNumero results for the TBV opening area optimization.

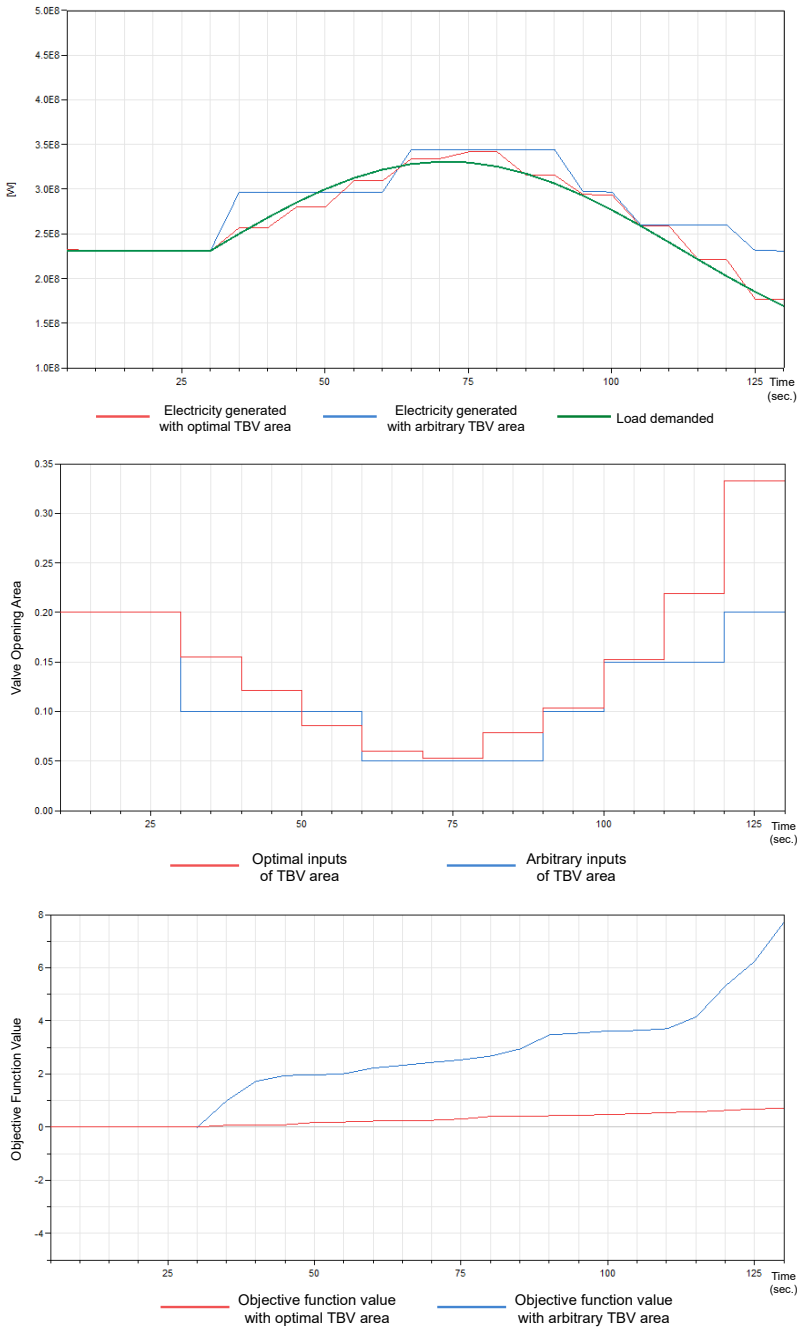


Figure 43. Dymola simulation using the optimized TBV opening area from PyNumero.

4. CONCLUSIONS

4.1 RAVEN in a Python Environment

This work included updating RAVEN to run in various environments, such as Python scripts and Jupyter notebooks. These updates allow RAVEN analyses to live in computer memory, and for analyses to be rerun without reinitializing and running the entire analysis from beginning to end. This is the necessary first step in running RAVEN in a dynamic environment in which inputs may be modified on the fly. For real-time DT workflows, this will be highly beneficial because the RAVEN analysis only need be instantiated once, then merely updated each time the analysis is run. This saves computational time by avoiding the overhead of having to reinstantiate RAVEN every time, and it enables RAVEN to run faster than real-time.

4.2 Deep Lynx Integration

This work showed the potential for Deep Lynx to efficiently store and query data for the RTO framework. The data stored in the graph database can potentially be processed faster and more flexibly than in a conventional relational database such as SQL. Queries can be executed according to metatype, relationship, graph, and time-series information. It does not require an additional join query to return related information simultaneously. In addition, the RTO Deep Lynx adapter has realized preliminary data processing and communication between Deep Lynx and TEDS operational data.

4.3 Real-time Optimization Workflow

This work included the development of an RTO workflow for a simple IES, consisting of an electrical storage device coupled to a NPP. The approach was to combine EMPC with LP. Building and solving an LP model in Pyomo and extracting the optimal results was demonstrated as a part of the EMPC workflow process. Creating this framework was the first step in generating RTO workflows for more complicated systems. Coupling this RTO workflow framework to a grey-box model (DT) will add the flexibility needed to accommodate more complex IES in the future. Additionally, the RTO workflow framework is in place to integrate with a DT to perform dispatch optimization in real-time with an operating physical IES.

4.4 Grey-box Modeling

An easy-to-use interface between a DT and an optimization framework, as well as a rapid optimization time, are desirable for RTO workflows. In this report, DT development for RTO was performed and the development concept, workflow, and technical characteristics of the optimization framework discussed in terms of a simple case study. A system-level simulation model in Modelica was used with RAVEN to create a ROM of the physical system, with the ROM functioning as a DT of that physical system. PyNumero enhanced the interface between the DT and the optimization framework by using the `ExternalGreyBoxModel` in the Pyomo problem setup instead of requiring the use of algebraic expressions. The RAVEN ROM was used directly in Pyomo as an external model, and optimal inputs minimizing the objective function were found.

5. FUTURE WORK

This report presented the basic building blocks of a DT and RTO workflow. Additional improvements must be made to each of these building blocks, including running RAVEN in Python environments, integrating Deep Lynx with the RTO workflow, the RTO workflow itself, and optimizing the grey-box modeling for integrating external models with the RTO workflow.

Running RAVEN in Python environments has been demonstrated; however, this is simply the necessary first step. A typical workflow for a user would include making updates to inputs—something that can only currently be accomplished by making changes to the RAVEN input XML file. Implementing a method for updating various RAVEN settings and inputs would enhance the user experience and enable RAVEN to run in a real-time interactive mode. Running RAVEN in a Jupyter notebook would generate additional challenges. Most of the RAVEN integration tests will run and rerun in a Jupyter notebook, but certain cases will cause the Jupyter kernel to crash. This problem does not appear to be caused by RAVEN itself. Most likely, it is an issue with Jupyter. These compatibility issues must be further investigated to ensure that all RAVEN workflows can be run successfully in a Jupyter notebook and can create a more user-friendly experience.

In the scope of this report, Deep Lynx only dealt with specific historical experimental data. Once a new experiment is ready, the RTO adapter will be required to update to process the new experimental data on a real-time basis. Also, the current style of type-mapping transformations in Deep Lynx can only be created manually through the Deep Lynx GUI, a process that is time-consuming and easily leads to errors as a result of the data processing. It is especially impractical for processing a large-sized graph database. Either the Deep Lynx adapter or Deep Lynx itself are required to implement the function in order to systematically create the large-sized type-mapping transformations. Deep Lynx is still in active development. New features are being deployed, and the interfaces become more sophisticated on a weekly basis. The RTO adapter will be updated in response to the Deep Lynx update.

A simple electrical storage device was demonstrated as an example of an RTO workflow. The RTO workflow should be expanded to handle more complex systems. To do so, the RTO workflow must be integrated with a grey-box model. For future RTO workflows, the workflow demonstrated here will be implemented as a Python package to abstract out the Pyomo model building. This Python package will also contain additional optimization strategies beyond the EMPC method used in this report. Users will then be able to create bespoke RTO workflows by using their own external grey-box models.

An RTO workflow must complete in time to implement the control actions before the economic opportunity has passed. The computational time for RTO workflows depends on the ROM type used and the how the optimization problem was set up. Future work will include testing several ROM types to determine the optimal type for RTO workflows. In addition, other approaches to developing the optimization model in Pyomo will be explored so as to give users more flexibility during the modeling process.

Ultimately, the RTO workflow will be demonstrated on a physical IES. An appropriate IES will be identified and a DT created. The data connections will need to be configured and the optimization framework set up to optimize the correct inputs and work with an external grey-box model (the DT) of the system.

6. REFERENCES

1. Krishnamoorthy, D., B. Foss, and S. Skogestad. 2018. “Steady-state real-time optimization using transient measurements.” *Computers and Chemical Engineering* 115:34-45. https://doi.org/10.1016/j.compchemeng.2018.03.021.
2. Mendoza, D., J. Graciano, F. Liporace, and G. Le Roux. 2016. “Assessing the reliability of different real-time optimization methodologies.” *The Canadian Journal of Chemical Engineering* 94(3):485-497. https://doi.org/10.1002/cjce.22402.
3. Miletic, I., and T. Marlin. 1998. “On-line Statistical Results Analysis in Real-Time Operations Optimization.” *Industrial & Engineering Chemistry Research* 37(9):3670-3684. https://doi.org/10.1021/IE9707376.

4. Roberts, P. 1979. "An algorithm for steady-state system optimization and parameter estimation." *International Journal of Systems Science* 10(7):719-734. https://doi.org/10.1080/00207727908941614.

5. Marchetti, A., B. Chachuat, and D. Bonvin. 2009. "Modifier-Adaptation Methodology for Real-Time Optimization." *Industrial & Engineering Chemistry Research* 48(13):6022-6033. https://doi.org/10.1021/ie801352x.

6. Ellis, M., H. Durand, and P. Christofides. 2014. "A tutorial review of economic model predictive control methods." *Journal of Process Control* 24(8):1156-1178. https://doi.org/10.1016/j.jprocont.2014.03.010.

7. Yadav, V., et al. 2021 "Technical Challenges and Gaps in Digital-Twin-Enabling Technologies for Nuclear Reactor Applications." TLR/RES-DE-REB-2021-17. U.S. Nuclear Regulatory Commission. https://adamswebsearch2.nrc.gov/webSearch2/main.jsp?AccessionNumber=ML21361A261.

8. Idaho National Laboratory. "RAVEN." Accessed September 2022. https://github.com/idaholab/raven.

9. Rabiti, C., et al. 2021. "RAVEN User Manual." INL/EXT-15-34123, Revision 7, Idaho National Laboratory. https://doi.org/10.2172/1784874.

10. Jupyter Project. "Jupyter." Accessed September 2022. https://jupyter.org/.

11. Kluyver, T., et al. 2016. "Jupyter Notebooks – a publishing format for reproducible computational workflows." In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, edited by F. Loizides and B. Scmidt. IOS Press. 87-90. https://doi.org/10.3233/978-1-61499-649-1-87.

12. Darrington, J., J. Browning, C. Ritter, and USDOE Office of Energy Efficiency and Renewable Energy. "Deep Lynx: Digital Engineering Integration Hub." Computer software. July 2, 2020. https://doi.org/10.11578/dc.20200929.1. https://github.com/idaholab/Deep-Lynx.

13. Browning, J., A. Al Rashdan, C. Ritter, J. Settle, and J. Gracely. "Data Integration Aggregated Model For Nuclear Deployment." Computer software. February 5, 2020. https://doi.org/10.11578/dc.20200501.1. https://github.com/idaholab/DIAMOND.

14. Al Rashdan, A., J. Browning, and C. Ritter. 2019. "Data Integration Aggregated Model and Ontology for Nuclear Deployment (DIAMOND): Preliminary Model and Ontology." INL/EXT-19-55610, Idaho National Laboratory.

15. Stoots, C., A. Duenas, P. Sabharwall, J. O'Brien, J. Yoo, and S. Bragg-Sitton. "Thermal Energy Delivery System Design Basis Report." INL/EXT-18-51351, Idaho National Laboratory. https://doi.org/10.2172/1756571.

16. Byrne, R., and C. Silva-Monroy. 2015. "Potential revenue from electrical energy storage in ERCOT: The impact of location and recent trends." *2015 IEEE Power & Energy Society General Meeting* 1-5. https://doi.org/10.1109/PESGM.2015.7286145.

17. Byrne, R., R. Concepcion, and C. Silva-Monroy. 2016. "Estimating potential revenue from electrical energy storage in PJM." *2016 IEEE Power and Energy Society General Meeting* 1-5. https://doi.org/10.1109/PESGM.2016.7741915.

18. Nguyen, T., R. Byrne, R. Concepcion, and I. Gyuk. 2017. "Maximizing revenue from electrical energy storage in MISO energy & frequency regulation markets." *2017 IEEE Power and Energy Society General Meeting* 1-5. https://doi.org/10.1109/PESGM.2017.8274348.

19. Byrne, R., T. Nguyen, D. Copp, R. Concepcion, B. Chalamala, and I. Gyuk. 2018. "Opportunities for Energy Storage in CAISO: Day-Ahead and Real-Time Market Arbitrage." *2018 International Symposium on Power Electronics, Electrical Drives, Automation and Motion (SPEEDAM)* 63-68. https://doi.org/10.1109/SPEEDAM.2018.8445408.

20. Wilches-Bernal, F., R. Concepcion, and R. Byrne. 2019. “Participation of Electric Storage Resources in the NYISO Electricity and Frequency Regulation Markets.” *2019 IEEE Power & Energy Society General Meeting (PESGM)* 1-5. https://doi.org/10.1109/PESGM40551.2019.8973512.

21. Concepcion, R., F. Wilches-Bernal, and R. Byrne. 2019. “Revenue Opportunities for Electric Storage Resources in the Southwest Power Pool Integrated Marketplace.” *2019 IEEE Power & Energy Society General Meeting (PESGM)* 1-5. https://doi.org/10.1109/PESGM40551.2019.8974047.

22. Tiller, M. 2001. *Introduction to Physical Modeling with Modelica*. Boston: Kluwer Academic Publishing.

23. Hart, W., J. Watson, and D. Woodruff. 2011. “Pyomo: modeling and solving mathematical programs in Python.” *Mathematical Programming Computation* 3:219-260. https://doi.org/10.1007/s12532-011-0026-8.

24. Hart, W., et al. 2017. *Pyomo – Optimization Modeling in Python.* Second Edition. Vol. 67. Springer.

25. Sandia National Laboratory. 2017. “Pyomo Documentation 6.4.2.” https://pyomo.readthedocs.io/en/stable/.

26. Dunning, I., S. Mitchell, and M. O’Sullivan. 2011. PuLP: “A Linear Programming Toolkit for Python.” Department of Engineering Science, The University of Auckland. https://optimization-online.org/2011/09/3178/.

27. Diamond, S., and S. Boyd. 2016. “CVXPY: A Python-Embedded Modeling Language for Convex Optimization.” *Journal of Machine Learning Research* 17:1-5. https://www.jmlr.org/papers/volume17/15-408/15-408.pdf.

28. Agrawal, A., R. Verschueren, S. Diamond, and S. Boyd. 2018. “A rewriting system for convex optimization problems.” *Journal of Control and Decision* 5(1):42-60. https://doi.org/10.1080/23307706.2017.1397554.

29. Perron, L., and V. Furnon. 2022. “OR-Tools.” Google. Accessed September 2022. https://developers.google.com/optimization.

30. Browning, J. M., and USDOE Office of Nuclear Energy. “Deep-Lynx-Python-Package.” Computer software. June 16, 2021. https://doi.org/10.11578/dc.20210629.1. https://github.com/idaholab/Deep-Lynx-Python-Package.

31. Ooms, J., Facebook, Inc. 2022. “graphql: A GraphQL Query Parser.” https://docs.ropensci.org/graphql, http://graphql.org (upstream), https://github.com/ropensci/graphql (devel).

32. Biegler, L., and V. Zavala. 2009. “Large-scale nonlinear programming using IPOPT: An integrating framework for enterprise-wide dynamic optimization.” *Computers & Chemical Engineering* 33(3):575-582. https://doi.org/10.1016/j.compchemeng.2008.08.006.

33. PJM. “Data Miner 2.” Accessed September 2022. https://dataminer2.pjm.com/list.

34. Forrest, J., et al. 2022. “coin-or/Cbc: Release releases/2.10.8.” Zenodo. https://doi.org/10.5281/zenodo.6522795.

35. Forrest, J., and R. Lougee-Heimer. 2005. “CBC User Guide.” IBM Corporation. https://www.coin-or.org/Cbc/.

36. Makhorin, A. 2012. “GLPK (GNU Linear Programming Kit). ” https://www.gnu.org/software/glpk/.

37. Rabiti, C., A. Epiney, P. Talbot, J. Kim, S. Bragg-Sitton, A. Alfonsi, A. Yigitoglu, S. Greenwood, S. Cetiner, F. Ganda, and G. Maronati. 2017. "Status Report on Modelling and Simulation Capabilities for Nuclear-Renewable Hybrid Energy Systems." INL/EXT-17-43441, Revision 1, Idaho National Laboratory. https://doi.org/10.2172/1408526.

38. Kim, J. S., M. McKellar, S. Bragg-Sitton, and R. Boardman. 2016. "Status on the Component Models Developed in the Modelica Framework: High-Temperature Steam Electrolysis Plant & Gas Turbine Power Plant." INL/EXT-16-40305, Idaho National Laboratory. https://doi.org/10.2172/1333156.

39. Kim, J. S., and K. Frick. 2018. "Status Report on the Component Models Developed in the Modelica Framework: Reverse Osmosis Desalination Plant & Thermal Energy Storage." INL/EXT-18-45505, Idaho National Laboratory. https://doi.org/10.2172/1468648.

40. Frick, K. 2019. "Status Report on the NuScale Module Developed in the Modelica Framework." INL/EXT-19-55520, Idaho National Laboratory. https://doi.org/10.2172/1569288.

41. Dempsey, M. 2006. "Dymola for Multi-Engineering Modelling and Simulation." *2006 IEEE Vehicle Power and Propulsion Conference* 1-6. https://doi.org/10.1109/VPPC.2006.364294.

42. Frick, K., S. Bragg-Sitton, and C. Rabiti. 2020. "Modeling the Idaho National Laboratory Thermal-Energy Distribution System (TEDS) in the Modelica Ecosystem." *Energies* 13(23):6353-6378. https://doi.org/10.3390/en13236353.

43. Greenwood, M., M. Cetiner, D. Fugate, R. Hale, T. Harrison, and A. Qualls. 2017. "TRANSFORM – TRANsient Simulation Framework of Reconfigurable Models." Computer software. Oak Ridge National Laboratory. https://www.osti.gov//servlets/purl/1395459.

44. Frick, K., A. Alfonsi, C. Rabiti, and D. Mikkelson. 2022. "Hybrid User Manual." Idaho National Laboratory. https://www.osti.gov/biblio/1863262.

Appendix A
Python Code `rom_out.py`

The Python code `rom_out.py`, discussed in Section 2.4, is presented here.

```
# MODEL GENERATED BY RAVEN
import pyomo.environ as pyo
from pyomo.contrib.pynumero.interfaces.external_grey_box import ExternalGreyBoxModel,
ExternalGreyBoxBlock
from pyomo.contrib.pynumero.dependencies import (numpy as np)
from pyomo.contrib.pynumero.asl import AmplInterface
from pyomo.contrib.pynumero.algorithms.solvers.cyipopt_solver import CyIpoptSolver, CyIpoptNLP
import os, sys, pickle
from contextlib import redirect_stdout

# RAVEN ROM PYOMO GREY MODEL CLASS
class ravenROM(ExternalGreyBoxModel):
  def __init__(self, **kwargs):
    self._rom_file = kwargs.get("rom_file")
    self._raven_framework = kwargs.get("raven_framework")
    self._raven_framework = os.path.abspath(self._raven_framework)
    if not os.path.exists(self._raven_framework):
      raise IOError('The RAVEN framework directory does not exist in location "' +
str(self._raven_framework)+'" !')
    if os.path.isdir(os.path.join(self._raven_framework,"ravenframework")):
      sys.path.append(self._raven_framework)
    else:
      raise IOError('The RAVEN framework directory does not exist in location "' +
str(self._raven_framework)+'" !')
    from ravenframework.CustomDrivers import DriverUtils as dutils
    dutils.doSetup()
    # de-serialize the ROM
    self._rom_file = os.path.abspath(self._rom_file)
    if not os.path.exists(self._rom_file):
      raise IOError('The serialized (binary) file has not been found in location "' +
str(self._rom_file)+'" !')
    self.rom = pickle.load(open(self._rom_file, mode='rb'))
    #
    self.settings = self.rom.getInitParams()
    # get input names
    self._input_names = self.settings.get('Features')
    # n_inputs
    self._n_inputs = len(self._input_names)
    # get output names
    self._output_names = self.settings.get('Target')
    # n_outputs
    self._n_outputs = len(self._output_names)
    # input values storage
    self._input_values = np.zeros(self._n_inputs, dtype=np.float64)

  def return_train_values(self, feat):
    return self.rom.trainingSet.get(feat)

  def input_names(self):
    return self._input_names

  def output_names(self):
    return self._output_names

  def set_input_values(self, input_values):
    assert len(input_values) == self._n_inputs
    np.copyto(self._input_values, input_values)
```

```
  def evaluate_equality_constraints(self):
      raise NotImplementedError('This method should not be called for this model.')

  def evaluate_outputs(self):
    request = {k:np.asarray(v) for k,v in zip(self._input_names,self._input_values)}
    outs = self.rom.evaluate(request)
    eval_outputs = np.asarray([outs[k].flatten() for k in self._output_names], dtype=np.float64)
    return eval_outputs.flatten()

  def evaluate_jacobian_outputs(self):
    request = {k:np.asarray(v) for k,v in zip(self._input_names,self._input_values)}
    derivatives = self.rom.derivatives(request)
    jac = np.zeros((self._n_outputs, self._n_inputs))
    for tc, target in enumerate(self._output_names):
      for fc, feature in enumerate(self._input_names):
        jac[tc,fc] = derivatives['d{}|d{}'.format(target, feature)]
    return jac

def pyomoModel(ex_model):
  m = pyo.ConcreteModel()
  m.egb = ExternalGreyBoxBlock()
  m.egb.set_external_model(ex_model)
  for inp in ex_model.input_names():
    m.egb.inputs[inp].value = np.mean(ex_model.return_train_values(inp))
    m.egb.inputs[inp].setlb(np.min(ex_model.return_train_values(inp)))
    m.egb.inputs[inp].setub(np.max(ex_model.return_train_values(inp)))
  for out in ex_model.output_names():
    m.egb.outputs[out].value = np.mean(ex_model.return_train_values(out))
    m.egb.outputs[out].setlb(np.min(ex_model.return_train_values(out)))
    m.egb.outputs[out].setub(np.max(ex_model.return_train_values(out)))
  m.obj = pyo.Objective(expr=m.egb.outputs[out])

  return m

if __name__ == '__main__':
  for cnt, item in enumerate(sys.argv):
    if item.lower() == "-r":
      rom_file = sys.argv[cnt+1]
    if item.lower() == "-f":
      raven_framework = sys.argv[cnt+1]
  ext_model = ravenROM(**{'rom_file':rom_file,'raven_framework':raven_framework})
  concreteModel = pyomoModel(ext_model)
  ### here you should implement the optimization problem
  ###
  solver = pyo.SolverFactory('cyipopt')
  solver.config.options['hessian_approximation'] = 'limited-memory'
  results = solver.solve(concreteModel)
  print(results)
  count = 0
  lines = str(results).split("\n")
  with open ("GreyModelOutput_cyipopt.txt", "w") as textfile:
    with redirect_stdout(textfile):
      print("--------------------< Output Summary >--------------------" + "\n")
      for element in lines:
        textfile.write(element + "\n")
      print("--------------------< Optimization Results >--------------------" + "\n")
      concreteModel.pprint()
```

Appendix B
Modelica Models

The Modelica models discussed in Section 3.4 are presented here in full.

B-1. Steam Turbine Model Modelica Code

```
model SteamTurbine_L1_boundaries_Test_Opt_b_v7_0to130
  import NHES;
  extends Modelica.Icons.Example;
  SteamTurbine_L1_boundaries_valve_area_opt_With_Scaler BOP(
    nPorts_a3=1,
    port_a3_nominal_m_flow={10},
    port_a_nominal(
      m_flow=493.7058,
      p=5550000,
      h=BOP.Medium.specificEnthalpy_pT(BOP.port_a_nominal.p, 591)),
    port_b_nominal(p=1000000, h=BOP.Medium.specificEnthalpy_pT(BOP.port_b_nominal.p,
          318.95)),
    redeclare
      NHES.Systems.BalanceOfPlant.Turbine.ControlSystems.CS_PressureAndPowerControl
      CS(
      delayStartTCV=600,
      p_nominal=BOP.port_a_nominal.p,
      W_totalSetpoint=sine.y),
    const(k=1.0),
    const1(k=0.1))
    annotation (Placement(transformation(extent={{-30,-30},{30,30}})));
  TRANSFORM.Electrical.Sources.FrequencySource
                                      sinkElec(f=60)
    annotation (Placement(transformation(extent={{62,-10},{42,10}})));
  Modelica.Fluid.Sources.Boundary_pT sink(
    redeclare package Medium = Modelica.Media.Water.StandardWater,
    nPorts=1,
    p(displayUnit="MPa") = BOP.port_b_nominal.p,
    T(displayUnit="K") = BOP.port_b_nominal.T)
    annotation (Placement(transformation(extent={{-88,-2},{-68,-22}})));
  NHES.Fluid.Sensors.stateSensor stateSensor(redeclare package Medium =
        Modelica.Media.Water.StandardWater)
    annotation (Placement(transformation(extent={{-40,-22},{-60,-2}})));
  NHES.Fluid.Sensors.stateSensor stateSensor1(redeclare package Medium =
        Modelica.Media.Water.StandardWater)
    annotation (Placement(transformation(extent={{-60,2},{-40,22}})));
  NHES.Fluid.Sensors.stateDisplay stateDisplay
    annotation (Placement(transformation(extent={{-72,-60},{-28,-30}})));
  NHES.Fluid.Sensors.stateDisplay stateDisplay1
    annotation (Placement(transformation(extent={{-72,20},{-28,50}})));
  Modelica.Fluid.Sources.MassFlowSource_h source1(
    redeclare package Medium = Modelica.Media.Water.StandardWater,
    nPorts=1,
    use_m_flow_in=false,
    m_flow=10,
    h=3e6)
    annotation (Placement(transformation(extent={{-10,-10},{10,10}},
        rotation=90,
        origin={-12,-52})));
  Modelica.Fluid.Sources.MassFlowSource_T source(
    redeclare package Medium = Modelica.Media.Water.StandardWater,
    nPorts=1,
    m_flow=BOP.port_a_nominal.m_flow,
    T(displayUnit="K") = BOP.port_a_nominal.T,
    use_m_flow_in=true)
    annotation (Placement(transformation(extent={{-88,2},{-68,22}})));
  Modelica.Blocks.Sources.Constant
```

```
                                  const(k=BOP.port_a_nominal.m_flow)
    annotation (Placement(transformation(extent={{-112,14},{-100,26}})));
equation

  connect(stateDisplay1.statePort, stateSensor1.statePort) annotation (Line(
        points={{-50,31.1},{-50,31.1},{-50,12.05},{-49.95,12.05}}, color={0,0,0}));
  connect(stateDisplay.statePort, stateSensor.statePort) annotation (Line(
        points={{-50,-48.9},{-50,-11.95},{-50.05,-11.95}}, color={0,0,0}));
  connect(sink.ports[1], stateSensor.port_b) annotation (Line(points={{-68,-12},
          {-64,-12},{-60,-12}}, color={0,127,255}));
  connect(stateSensor.port_a, BOP.port_b)
    annotation (Line(points={{-40,-12},{-30,-12}}, color={0,127,255}));
  connect(stateSensor1.port_b, BOP.port_a)
    annotation (Line(points={{-40,12},{-30,12}}, color={0,127,255}));
  connect(source1.ports[1], BOP.port_a3[1]) annotation (Line(points={{-12,-42},{
        -12,-30}},            color={0,127,255}));
  connect(source.ports[1], stateSensor1.port_a)
    annotation (Line(points={{-68,12},{-60,12}}, color={0,127,255}));
  connect(const.y, source.m_flow_in)
    annotation (Line(points={{-99.4,20},{-88,20}},
                                                 color={0,0,127}));
  connect(BOP.portElec_b, sinkElec.port)
    annotation (Line(points={{30,0},{42,0}}, color={255,0,0}));
  annotation (experiment(
      StopTime=130,
      Interval=5,
      __Dymola_Algorithm="Dassl"),
    Diagram(coordinateSystem(extent={{-120,-100},{100,100}})),
    Icon(coordinateSystem(extent={{-120,-100},{100,100}})));
end SteamTurbine_L1_boundaries_Test_Opt_b_v7_0to130;
```

B-2. BOP Model Modelica Code

```
model SteamTurbine_L1_boundaries_valve_area_opt_With_Scaler

  extends BaseClasses.Partial_SubSystem_B(
    redeclare replaceable ControlSystems.CS_Dummy CS,
    redeclare replaceable ControlSystems.ED_Dummy ED,
    redeclare Data.IdealTurbine data);

  parameter Modelica.Units.SI.Pressure p_condenser=1e4
    "Condenser operating pressure";
  parameter Modelica.Units.SI.Pressure p_reservoir=port_b_nominal.p
    "Reservoir operating pressure";

  TRANSFORM.Fluid.Machines.SteamTurbine steamTurbine(
    redeclare package Medium = Medium,
    use_T_start=false,
    h_a_start=port_a_start.h,
    m_flow_start=port_a_start.m_flow,
    m_flow_nominal=port_a_nominal.m_flow,
    use_T_nominal=false,
    nUnits=2,
    energyDynamics=TRANSFORM.Types.Dynamics.DynamicFreeInitial,
    p_b_start=p_condenser,
    p_outlet_nominal=p_condenser,
    d_nominal=Medium.density_ph(steamTurbine.p_inlet_nominal, port_a_nominal.h),
    p_a_start=header.p_start -valve_TCV.dp_start,
    p_inlet_nominal=port_a_nominal.p -valve_TCV.dp_nominal)
    annotation (Placement(transformation(extent={{40,-10},{60,10}})));
  NHES.Electrical.Generator generator1(J=1e4)
    annotation (Placement(transformation(extent={{100,-10},{120,10}})));
  TRANSFORM.Electrical.Sensors.PowerSensor sensorW
    annotation (Placement(transformation(extent={{130,-10},{150,10}})));
  TRANSFORM.Fluid.FittingsAndResistances.SpecifiedResistance resistance(
      redeclare package Medium = Medium, R=1)
```

```
    annotation (Placement(transformation(extent={{10,-10},{-10,10}},
        rotation=90,
        origin={80,-30})));
  TRANSFORM.Fluid.Volumes.IdealCondenser
                                      condenser(
    redeclare package Medium = Medium,
    set_m_flow=true,
    p=p_condenser)
    annotation (Placement(transformation(extent={{77,-94},{97,-74}})));
  TRANSFORM.Fluid.Volumes.SimpleVolume volume(
    redeclare package Medium = Medium,
    use_T_start=false,
    redeclare model Geometry =
        TRANSFORM.Fluid.ClosureRelations.Geometry.Models.LumpedVolume.GenericVolume
        (V=0.01),
    p_start=p_condenser,
    h_start=steamTurbine.h_b_start)
    annotation (Placement(transformation(extent={{58,-30},{78,-10}})));
  TRANSFORM.Fluid.Volumes.DumpTank reservoir(
    redeclare package Medium = Medium,
    A=10,
    p_start=p_reservoir,
    level_start=10)
    annotation (Placement(transformation(extent={{-10,-110},{10,-90}})));
  Modelica.Fluid.Fittings.MultiPort multiPort(redeclare package Medium =
        Medium, nPorts_b=2)
                        annotation (Placement(transformation(
        extent={{-4,-10},{4,10}},
        rotation=90,
        origin={0,-80})));
  TRANSFORM.Fluid.Volumes.SimpleVolume feedWaterHeater(
    use_HeatPort=true,
    redeclare package Medium = Medium,
    p_start=port_b_start.p,
    redeclare model Geometry =
        TRANSFORM.Fluid.ClosureRelations.Geometry.Models.LumpedVolume.GenericVolume
        (V=5),
    use_T_start=false,
    h_start=port_b_start.h)
    annotation (Placement(transformation(extent={{-70,-90},{-90,-70}})));
  TRANSFORM.HeatAndMassTransfer.BoundaryConditions.Heat.HeatFlow    boundary(use_port=
        true)
    annotation (Placement(transformation(extent={{-102,-110},{-82,-90}})));
  TRANSFORM.Controls.LimPID PID(
    controllerType=Modelica.Blocks.Types.SimpleController.PI,
    yb=1e8,
    k=1e8,
    k_s=1/port_b_nominal.T,
    k_m=1/port_b_nominal.T)
    annotation (Placement(transformation(extent={{-108,-96},{-100,-104}})));
  Modelica.Blocks.Sources.RealExpression realExpression(y=port_b_nominal.T)
    annotation (Placement(transformation(extent={{-124,-106},{-114,-94}})));
  TRANSFORM.Fluid.Sensors.Temperature temperature(redeclare package Medium =
        Medium)
    annotation (Placement(transformation(extent={{-96,-82},{-104,-90}})));
  Modelica.Fluid.Fittings.MultiPort multiPort1(redeclare package Medium =
        Medium, nPorts_b=if nPorts_a3 > 0 then nPorts_a3+2 else 2)
                        annotation (Placement(transformation(
        extent={{-4,-10},{4,10}},
        rotation=90,
        origin={80,-66})));
  TRANSFORM.Fluid.FittingsAndResistances.SpecifiedResistance resistance1(
      redeclare package Medium = Medium, R=1)
    annotation (Placement(transformation(extent={{-10,-10},{10,10}},
        rotation=180,
        origin={-130,-80})));
  TRANSFORM.Fluid.Sensors.Pressure pressure(redeclare package Medium = Medium)
```

```
    annotation (Placement(transformation(extent={{-110,50},{-90,70}})));
  TRANSFORM.Fluid.Valves.ValveCompressible valve_BV(
    rho_nominal=Medium.density_ph(port_a_nominal.p, port_a_nominal.h),
    p_nominal=port_a_nominal.p,
    redeclare package Medium = Medium,
    m_flow_nominal=port_a_nominal.m_flow,
    dp_nominal=100000)
    annotation (Placement(transformation(extent={{-80,-10},{-60,10}})));
  TRANSFORM.Fluid.Volumes.SimpleVolume volume_bypass(
    use_T_start=false,
    h_start=port_a_start.h,
    redeclare model Geometry =
        TRANSFORM.Fluid.ClosureRelations.Geometry.Models.LumpedVolume.GenericVolume
        (V=0.01),
    redeclare package Medium = Medium,
    p_start=p_condenser)
                  "included for numeric purposes"
    annotation (Placement(transformation(extent={{-30,-10},{-10,10}})));
  TRANSFORM.Fluid.FittingsAndResistances.SpecifiedResistance resistance3(R=1000,
      redeclare package Medium = Medium)
    annotation (Placement(transformation(extent={{10,-10},{-10,10}},
        rotation=90,
        origin={0,-10})));
  Modelica.Blocks.Sources.RealExpression W_balance1
    "Electricity loss/gain not accounted for in connections (e.g., heating/cooling, pumps
, etc.) [W]"
    annotation (Placement(transformation(extent={{-96,118},{-84,130}})));
  TRANSFORM.Fluid.Valves.ValveCompressible valve_TCV(
    rho_nominal=Medium.density_ph(port_a_nominal.p, port_a_nominal.h),
    p_nominal=port_a_nominal.p,
    redeclare package Medium = Medium,
    m_flow_nominal=port_a_nominal.m_flow,
    dp_nominal=100000)
    annotation (Placement(transformation(extent={{-80,30},{-60,50}})));
  TRANSFORM.Fluid.Volumes.MixingVolume header(
    use_T_start=false,
    h_start=port_a_start.h,
    p_start=port_a_start.p,
    nPorts_a=1,
    nPorts_b=3,
    redeclare model Geometry =
        TRANSFORM.Fluid.ClosureRelations.Geometry.Models.LumpedVolume.GenericVolume
        (V=1),
    redeclare package Medium = Medium)
    annotation (Placement(transformation(extent={{-120,30},{-100,50}})));
  TRANSFORM.Fluid.FittingsAndResistances.SpecifiedResistance resistance4(R=1000,
      redeclare package Medium = Medium)
    annotation (Placement(transformation(extent={{10,-10},{-10,10}},
        rotation=180,
        origin={-130,40})));
  TRANSFORM.Fluid.BoundaryConditions.Boundary_ph boundary_a3[nPorts_a3](
    redeclare package Medium = Medium,
    each nPorts=1,
    p=port_a3_nominal_p,
    h=port_a3_nominal_h) if nPorts_a3 > 0
    annotation (Placement(transformation(extent={{50,-150},{30,-130}})));
   TRANSFORM.Fluid.Valves.CheckValve checkValve[nPorts_a3](redeclare package
      Medium =  Medium, m_flow_start=port_a3_start_m_flow) if nPorts_a3 > 0
     annotation (Placement(transformation(extent={{0,-150},{20,-130}})));
   TRANSFORM.Fluid.BoundaryConditions.MassFlowSource_h boundary_m_flow_a3[
     nPorts_a3](
     redeclare package Medium = Medium,
     each nPorts=1,
    each use_m_flow_in=true,
    each use_h_in=true)  if nPorts_a3 > 0
     annotation (Placement(transformation(extent={{72,-150},{92,-130}})));
  TRANSFORM.Fluid.Sensors.MassFlowRate massFlowRate[nPorts_a3](redeclare
```

```
    package Medium = Medium) if nPorts_a3 > 0
  annotation (Placement(transformation(extent={{-30,-150},{-10,-130}})));
TRANSFORM.Fluid.Sensors.SpecificEnthalpyTwoPort
                                        specificEnthalpy[nPorts_a3](
    redeclare package Medium = Medium) if nPorts_a3 > 0
  annotation (Placement(transformation(extent={{-60,-150},{-40,-130}})));
Modelica.Mechanics.Rotational.Sensors.PowerSensor powerSensor
  annotation (Placement(transformation(extent={{70,10},{90,-10}})));
TRANSFORM.Fluid.FittingsAndResistances.SpecifiedResistance resistance2(R=1,
    redeclare package Medium = Medium)
  annotation (Placement(transformation(extent={{-10,-10},{10,10}},
      rotation=180,
      origin={-50,-80})));
TRANSFORM.Controls.LimPID_Hysteresis PID1(
  controllerType=Modelica.Blocks.Types.SimpleController.PI,
  k_s=1/reservoir.level_start,
  k_m=1/reservoir.level_start,
  k=1e2,
  yMin=0,
  eOn=0.1*reservoir.level_start)
  annotation (Placement(transformation(extent={{-62,-50},{-42,-30}})));
Modelica.Blocks.Sources.RealExpression level_setpoint(y=reservoir.level_start)
  annotation (Placement(transformation(extent={{-94,-50},{-74,-30}})));
Modelica.Blocks.Sources.RealExpression level_measure(y=reservoir.level)
  "noEvent(if time < 10 then reservoir.level_start else reservoir.level)"
  annotation (Placement(transformation(extent={{-94,-62},{-74,-42}})));
TRANSFORM.Fluid.BoundaryConditions.MassFlowSource_T boundary2(
  redeclare package Medium = Modelica.Media.Water.StandardWater,
  use_m_flow_in=true,
  T=298.15,
  nPorts=1)
  annotation (Placement(transformation(extent={{-28,-70},{-8,-50}})));
Modelica.Blocks.Sources.Sine sine_Load(
  f=1/165,
  offset=2.3090397e8,
  startTime=30,
  amplitude=1e8)
  annotation (Placement(transformation(extent={{46,56},{66,76}})));
Modelica.Blocks.Sources.CombiTimeTable combiTimeTable_TCV(table=[0,0.5;
      25,0.75; 50,1; 75,0.75; 100,0.5])
  annotation (Placement(transformation(extent={{-8,62},{-28,82}})));
Modelica.Blocks.Sources.Constant const(k=1)
  annotation (Placement(transformation(extent={{-44,66},{-56,78}})));
Modelica.Blocks.Sources.Constant const1(k=0)
  annotation (Placement(transformation(extent={{-96,14},{-84,26}})));
Modelica.Blocks.Continuous.Integrator obj(y_start=-3.399617066)
  annotation (Placement(transformation(extent={{146,32},{166,52}})));

parameter Integer n = 11;

//************************************************//
//New vectorization for time-dependent RAVEN input;

parameter Real valve_area_vec[n] = {0.20,
                                    0.23,
                                    0.25,
                                    0.20,
                                    0.20,
                                    0.20,
                                    0.15,
                                    0.20,
                                    0.30,
                                    0.35,
                                    0.45};

Integer count(start=1);
```

```
  parameter Real time_setpoint_vec[n] = {30,
                                         40,
                                         50,
                                         60,
                                         70,
                                         80,
                                         90,
                                         100,
                                         110,
                                         120,
                                         130};

  Real time_step;
  //************************************************//

  Examples.Math_Exp_With_Scaler math_Exp
    annotation (Placement(transformation(extent={{86,30},{106,50}})));
initial equation
  time_step = time_setpoint_vec[1];

equation
  //Code to step in time along our vectors
  when time>time_step then
    count = pre(count)+1;
time_step = pre(time_step) + time_setpoint_vec[count]-pre(time_step);
//Must be within the when() loop and must use pre() stuff for Modelica reasons.
  end when;
    valve_BV.opening = valve_area_vec[count];
  //End of new code

for i in 1:nPorts_a3 loop
    connect(specificEnthalpy[i].port_a, port_a3[i]);
    connect(specificEnthalpy[i].port_b, massFlowRate[i].port_a);
     connect(checkValve[i].port_a, massFlowRate[i].port_b);
     connect(checkValve[i].port_b, boundary_a3[i].ports[1]);
connect(boundary_m_flow_a3[i].ports[1], multiPort1.ports_b[i+2]);

end for;

  connect(generator1.portElec, sensorW.port_a)
    annotation (Line(points={{120,0},{130,0}}, color={255,0,0}));
  connect(sensorW.port_b, portElec_b)
    annotation (Line(points={{150,0},{160,0}}, color={255,0,0}));
  connect(steamTurbine.portLP, volume.port_a)
    annotation (Line(points={{60,6},{60,-20},{62,-20}},   color={0,127,255}));
  connect(boundary.port,feedWaterHeater. heatPort)
    annotation (Line(points={{-82,-100},{-80,-100},{-80,-86}},
                                                        color={191,0,0}));
  connect(temperature.T,PID. u_m) annotation (Line(points={{-102.4,-86},{-104,
          -86},{-104,-95.2}},
                        color={0,0,127}));
  connect(realExpression.y,PID. u_s)
    annotation (Line(points={{-113.5,-100},{-108.8,-100}},
                                                      color={0,0,127}));
  connect(PID.y,boundary. Q_flow_ext)
    annotation (Line(points={{-99.6,-100},{-96,-100}}, color={0,0,127}));
  connect(temperature.port,feedWaterHeater. port_b) annotation (Line(points={{-100,
          -82},{-100,-80},{-86,-80}},color={0,127,255}));
  connect(multiPort.port_a, reservoir.port_a)
    annotation (Line(points={{0,-84},{0,-91.6}}, color={0,127,255}));
  connect(volume.port_b, resistance.port_a)
    annotation (Line(points={{74,-20},{80,-20},{80,-23}}, color={0,127,255}));
  connect(multiPort1.port_a, condenser.port_a)
    annotation (Line(points={{80,-70},{80,-77}}, color={0,127,255}));
  connect(condenser.port_b, multiPort.ports_b[1]) annotation (Line(points={{87,-92},
          {87,-100},{20,-100},{20,-68},{-2,-68},{-2,-76}}, color={0,127,255}));
  connect(sensorBus.p_inlet_steamTurbine, pressure.p)
```

```
    annotation (Line(
      points={{-29.9,100.1},{-94,100.1},{-94,60}},
      color={239,82,82},
      pattern=LinePattern.Dash,
      thickness=0.5));
  connect(volume_bypass.port_a, valve_BV.port_b)
    annotation (Line(points={{-26,0},{-60,0}}, color={0,127,255}));
  connect(volume_bypass.port_b, resistance3.port_a)
    annotation (Line(points={{-14,0},{4.44089e-16,0},{4.44089e-16,-3}},
                                                       color={0,127,255}));
  connect(resistance3.port_b, multiPort1.ports_b[1]) annotation (Line(points={{
          -4.44089e-16,-17},{-4.44089e-16,-28},{0,-28},{0,-40},{80,-40},{80,-62}},
                                                     color={0,127,255}));
  connect(resistance.port_b, multiPort1.ports_b[2]) annotation (Line(points={{80,-37},
          {80,-62}},                         color={0,127,255}));
  connect(valve_BV.port_a, header.port_b[1]) annotation (Line(points={{-80,0},{-100,
          0},{-100,39.3333},{-104,39.3333}},      color={0,127,255}));
  connect(valve_TCV.port_a, header.port_b[2]) annotation (Line(points={{-80,40},
          {-92,40},{-92,40},{-104,40}},     color={0,127,255}));
  connect(port_a, resistance4.port_a)
    annotation (Line(points={{-160,40},{-137,40}}, color={0,127,255}));
  connect(resistance4.port_b, header.port_a[1])
    annotation (Line(points={{-123,40},{-116,40}}, color={0,127,255}));
  connect(resistance1.port_a, feedWaterHeater.port_b)
    annotation (Line(points={{-123,-80},{-86,-80}}, color={0,127,255}));
  connect(resistance1.port_b, port_b) annotation (Line(points={{-137,-80},{-140,
          -80},{-140,-40},{-160,-40}}, color={0,127,255}));
  connect(massFlowRate.m_flow, boundary_m_flow_a3.m_flow_in) annotation (Line(
        points={{-20,-136.4},{-20,-124},{62,-124},{62,-132},{72,-132}},
                                                                 color={0,0,
          127}));
  connect(specificEnthalpy.h_out, boundary_m_flow_a3.h_in) annotation (Line(
        points={{-50,-136.4},{-50,-126},{60,-126},{60,-136},{70,-136}},
                                                                 color={0,0,
          127}));
  connect(steamTurbine.shaft_b, powerSensor.flange_a)
    annotation (Line(points={{60,0},{70,0}}, color={0,0,0}));
  connect(powerSensor.flange_b, generator1.shaft_a)
    annotation (Line(points={{90,0},{100,0}}, color={0,0,0}));
  connect(valve_TCV.port_b, steamTurbine.portHP)
    annotation (Line(points={{-60,40},{40,40},{40,6}}, color={0,127,255}));
  connect(pressure.port, header.port_b[3]) annotation (Line(points={{-100,50},{-100,
          40},{-104,40},{-104,40.6667}},          color={0,127,255}));
  connect(feedWaterHeater.port_a, resistance2.port_b)
    annotation (Line(points={{-74,-80},{-57,-80}}, color={0,127,255}));
  connect(resistance2.port_a, reservoir.port_b) annotation (Line(points={{-43,
          -80},{-40,-80},{-40,-114},{0,-114},{0,-108.4}}, color={0,127,255}));
  connect(sensorBus.W_total, sensorW.W) annotation (Line(
      points={{-29.9,100.1},{140,100.1},{140,11}},
      color={239,82,82},
      pattern=LinePattern.Dash,
      thickness=0.5));
  connect(level_measure.y, PID1.u_m)
    annotation (Line(points={{-73,-52},{-52,-52}}, color={0,0,127}));
  connect(level_setpoint.y, PID1.u_s)
    annotation (Line(points={{-73,-40},{-64,-40}}, color={0,0,127}));
  connect(PID1.y, boundary2.m_flow_in) annotation (Line(points={{-41,-40},{-38,
          -40},{-38,-52},{-28,-52}}, color={0,0,127}));
  connect(boundary2.ports[1], multiPort.ports_b[2])
    annotation (Line(points={{-8,-60},{2,-60},{2,-76}}, color={0,127,255}));
  connect(const.y, valve_TCV.opening) annotation (Line(points={{-56.6,72},{
          -70,72},{-70,48}}, color={0,0,127}));
  connect(sine_Load.y, math_Exp.u1) annotation (Line(points={{67,66},{78,66},
          {78,46},{84,46}}, color={0,0,127}));
  connect(powerSensor.power, math_Exp.u2)
    annotation (Line(points={{72,11},{72,34},{84,34}}, color={0,0,127}));
  connect(math_Exp.y, obj.u) annotation (Line(points={{107,40},{138,40},{138,42},
```

```
          {144,42}}, color={0,0,127}));
  annotation (defaultComponentName="BOP", Icon(coordinateSystem(extent={{-100,-100},
            {100,100}}),                        graphics={
        Rectangle(
          extent={{-2.09756,2},{83.9024,-2}},
          lineColor={0,0,0},
          origin={-39.9024,-64},
          rotation=360,
          fillColor={0,0,255},
          fillPattern=FillPattern.HorizontalCylinder),
        Rectangle(
          extent={{-0.578156,2.1722},{23.1262,-2.1722}},
          lineColor={0,0,0},
          origin={27.4218,-39.8278},
          rotation=180,
          fillColor={0,0,255},
          fillPattern=FillPattern.HorizontalCylinder),
        Rectangle(
          extent={{-1.81332,3},{66.1869,-3}},
          lineColor={0,0,0},
          origin={-14.1867,-1},
          rotation=0,
          fillColor={135,135,135},
          fillPattern=FillPattern.HorizontalCylinder),
        Rectangle(
          extent={{-0.373344,2},{13.6267,-2}},
          lineColor={0,0,0},
          origin={24.3733,-56},
          rotation=0,
          fillColor={0,0,255},
          fillPattern=FillPattern.HorizontalCylinder),
        Rectangle(
          extent={{-0.4,3},{15.5,-3}},
          lineColor={0,0,0},
          origin={36.4272,-29},
          rotation=0,
          fillColor={0,128,255},
          fillPattern=FillPattern.HorizontalCylinder),
        Rectangle(
          extent={{-14,46},{12,34}},
          lineColor={0,0,0},
          fillColor={66,200,200},
          fillPattern=FillPattern.HorizontalCylinder),
        Rectangle(
          extent={{-64,46},{-16,34}},
          lineColor={0,0,0},
          fillColor={66,200,200},
          fillPattern=FillPattern.HorizontalCylinder),
        Ellipse(
          extent={{-24,49},{-6,31}},
          lineColor={95,95,95},
          fillColor={175,175,175},
          fillPattern=FillPattern.Sphere),
        Ellipse(
          extent={{-13,49},{-17,31}},
          lineColor={0,0,0},
          fillPattern=FillPattern.VerticalCylinder,
          fillColor={162,162,0}),
        Rectangle(
          extent={{-24,63},{-6,61}},
          lineColor={0,0,0},
          fillColor={181,0,0},
          fillPattern=FillPattern.HorizontalCylinder),
        Rectangle(
          extent={{-14,49},{-16,61}},
          lineColor={0,0,0},
          fillColor={95,95,95},
```

```
          fillPattern=FillPattern.VerticalCylinder),
        Rectangle(
          extent={{-16,3},{16,-3}},
          lineColor={0,0,0},
          fillColor={66,200,200},
          fillPattern=FillPattern.HorizontalCylinder,
          origin={10,30},
          rotation=-90),
        Polygon(
          points={{6,16},{6,-14},{36,-32},{36,36},{6,16}},
          lineColor={0,0,0},
          fillColor={0,114,208},
          fillPattern=FillPattern.Solid),
        Rectangle(
          extent={{-1.81329,5},{66.1867,-5}},
          lineColor={0,0,0},
          origin={-62.1867,-41},
          rotation=0,
          fillColor={0,0,255},
          fillPattern=FillPattern.HorizontalCylinder),
        Line(points={{10,-42}}, color={0,0,0}),
        Rectangle(
          extent={{-0.43805,2.7864},{15.9886,-2.7864}},
          lineColor={0,0,0},
          origin={49.2136,-41.9886},
          rotation=90,
          fillColor={0,128,255},
          fillPattern=FillPattern.HorizontalCylinder),
        Text(
          extent={{15,-8},{25,6}},
          lineColor={0,0,0},
          fillColor={255,255,255},
          fillPattern=FillPattern.Solid,
          textString="T"),
        Polygon(
          points={{4,-44},{0,-48},{16,-48},{12,-44},{4,-44}},
          lineColor={0,0,255},
          pattern=LinePattern.None,
          fillColor={0,0,0},
          fillPattern=FillPattern.VerticalCylinder),
        Ellipse(
          extent={{2,-34},{14,-46}},
          lineColor={0,0,0},
          fillPattern=FillPattern.Sphere,
          fillColor={0,100,199}),
        Polygon(
          points={{9,-37},{9,-43},{5,-40},{9,-37}},
          lineColor={0,0,0},
          pattern=LinePattern.None,
          fillPattern=FillPattern.HorizontalCylinder,
          fillColor={255,255,255}),
        Ellipse(
          extent={{50,12},{78,-14}},
          lineColor={0,0,0},
          fillColor={255,255,255},
          fillPattern=FillPattern.Solid),
        Text(
          extent={{59,-8},{69,6}},
          lineColor={0,0,0},
          fillColor={255,255,255},
          fillPattern=FillPattern.Solid,
          textString="G"),
        Rectangle(
          extent={{-0.487802,2},{19.5122,-2}},
          lineColor={0,0,0},
          origin={26,-38.4878},
          rotation=-90,
```

```
          fillColor={0,0,255},
          fillPattern=FillPattern.HorizontalCylinder),
        Ellipse(
          extent={{36,-42},{64,-68}},
          lineColor={0,0,0},
          fillColor={255,255,255},
          fillPattern=FillPattern.Solid),
        Text(
          extent={{45,-62},{55,-48}},
          lineColor={0,0,0},
          fillColor={255,255,255},
          fillPattern=FillPattern.Solid,
          textString="C"),
        Text(
          extent={{-94,82},{94,74}},
          lineColor={0,0,0},
          lineThickness=1,
          fillColor={255,255,237},
          fillPattern=FillPattern.Solid,
          textString="Turbine"),
        Rectangle(
          extent={{-0.243902,2},{9.7562,-2}},
          lineColor={0,0,0},
          origin={-40,-62.2438},
          rotation=-90,
          fillColor={0,0,255},
          fillPattern=FillPattern.HorizontalCylinder)}),
    Diagram(coordinateSystem(extent={{-160,-160},{160,140}})),
    experiment(StopTime=1000));
end SteamTurbine_L1_boundaries_valve_area_opt_With_Scaler;
```

Appendix C
RAVEN Input XML File

This appendix contains the RAVEN input XML file used for the optimization problem regarding a Rankine cycle BOP model.

```
<?xml version="1.0" ?>
<Simulation verbosity="debug">
  <RunInfo>
    <JobName>BOP_CreatePickle_</JobName>
    <Sequence>
      sample, trainROM, sampleROM, serialize, dumpROM, writeHistories
    </Sequence>
    <WorkingDir>BOP_Opt</WorkingDir>
    <batchSize>6</batchSize>
  </RunInfo>

  <TestInfo>
    <name> RAVEN Input Code for Optimization Problem Solving of a Rankin Cycle BOP model </name>
    <author>Junyung Kim</author>
    <created>08_17_2022</created>
    <classesTested></classesTested>
    <description>
      RAVEN input deck for PyNumero optimization test.
    </description>
  </TestInfo>

  <Steps>
    <MultiRun name="sample">
      <Input class="Files" type="DymolaInitialisation">BOPInput</Input>
      <Model class="Models" type="Code">BOP</Model>
      <Sampler class="Samplers" type="MonteCarlo">my_mc</Sampler>
      <Output class="DataObjects" type="PointSet">samples</Output>
      <Output class="DataObjects" type="HistorySet">histories</Output>
    </MultiRun>
    <MultiRun name="sampleROM">
      <Input class="DataObjects" type="PointSet">inputPlaceHolder</Input>
      <Model class="Models" type="ROM">ROM</Model>
      <Sampler class="Samplers" type="MonteCarlo">my_mc</Sampler>
      <Output class="DataObjects" type="PointSet">samplesROM</Output>
    </MultiRun>
    <RomTrainer name="trainROM">
      <Input class="DataObjects" type="PointSet">samples</Input>
      <Output class="Models" type="ROM">ROM</Output>
    </RomTrainer>

    <IOStep name="serialize">
      <Input class="Models" type="ROM">ROM</Input>
      <Output class="Files" type="">rom_out.py</Output>
    </IOStep>

    <IOStep name="dumpROM">
      <Input class="Models" type="ROM">ROM</Input>
      <Output class="Files" type="">rom_pickle</Output>
    </IOStep>
      <IOStep name="writeHistories" pauseAtEnd="True">
      <Input class="DataObjects" type="HistorySet">histories</Input>
      <Input class="DataObjects" type="PointSet">samples</Input>
      <Output class="OutStreams" type="Print">histories</Output>
      <Output class="OutStreams" type="Print">samples</Output>
    </IOStep>
  </Steps>
```

```
  <Files>
    <Input name="BOPInput" type="DymolaInitialisation">C:\Users\<UserName_Here>\Documents\Dymola
\dsin.txt</Input>
    <Input name="rom_out.py" type="Pyomo">rom_out.py</Input>
    <Input name="rom_pickle" type="">BOP.pk</Input>
  </Files>

  <Models>
    <Code name="BOP" subType = "Dymola">
      <executable> C:\Users\<UserName_Here>\Documents\Dymola\dymosim.exe</executable>
      <alias type="input" variable="valve_area_t30_40">BOP.valve_area_vec[2]</alias>
      <alias type="input" variable="valve_area_t40_50">BOP.valve_area_vec[3]</alias>
      <alias type="input" variable="valve_area_t50_60">BOP.valve_area_vec[4]</alias>
      <alias type="input" variable="valve_area_t60_70">BOP.valve_area_vec[5]</alias>
      <alias type="input" variable="valve_area_t70_80">BOP.valve_area_vec[6]</alias>
      <alias type="input" variable="valve_area_t80_90">BOP.valve_area_vec[7]</alias>
      <alias type="input" variable="valve_area_t90_100">BOP.valve_area_vec[8]</alias>
      <alias type="input" variable="valve_area_t100_110">BOP.valve_area_vec[9]</alias>
      <alias type="input" variable="valve_area_t110_120">BOP.valve_area_vec[10]</alias>
      <alias type="input" variable="valve_area_t120_130">BOP.valve_area_vec[11]</alias>

    </Code>
    <ROM name="ROM" subType="NDinvDistWeight">
      <pivotParameter>Time</pivotParameter>
      <Features>valve_area_t30_40, valve_area_t40_50, valve_area_t50_60, valve_area_t60_70,
valve_area_t70_80, valve_area_t80_90, valve_area_t90_100, valve_area_t100_110,
valve_area_t110_120, valve_area_t120_130 </Features>
      <Target>BOP.obj.y</Target>
      <p>1</p>
    </ROM>
  </Models>

  <Distributions>
    <Uniform name="va_t30_40_dist">
      <lowerBound>0.10</lowerBound>
      <upperBound>0.20</upperBound>
    </Uniform>
    <Uniform name="va_t40_50_dist">
      <lowerBound>0.10</lowerBound>
      <upperBound>0.15</upperBound>
    </Uniform>
    <Uniform name="va_t50_60_dist">
      <lowerBound>0.05</lowerBound>
      <upperBound>0.10</upperBound>
    </Uniform>
    <Uniform name="va_t60_70_dist">
      <lowerBound>0.05</lowerBound>
      <upperBound>0.10</upperBound>
    </Uniform>
    <Uniform name="va_t70_80_dist">
      <lowerBound>0.05</lowerBound>
      <upperBound>0.10</upperBound>
    </Uniform>-->
    <Uniform name="va_t80_90_dist">
      <lowerBound>0.05</lowerBound>
      <upperBound>0.15</upperBound>
    </Uniform>
    <Uniform name="va_t90_100_dist">
      <lowerBound>0.10</lowerBound>
      <upperBound>0.15</upperBound>
    </Uniform>
    <Uniform name="va_t100_110_dist">
      <lowerBound>0.10</lowerBound>
      <upperBound>0.20</upperBound>
    </Uniform>
```

```
    <Uniform name="va_t110_120_dist">
      <lowerBound>0.20</lowerBound>
      <upperBound>0.30</upperBound>
    </Uniform>-->
    <Uniform name="va_t120_130_dist">
      <lowerBound>0.25</lowerBound>
      <upperBound>0.35</upperBound>
    </Uniform>
  </Distributions>

  <Samplers>
    <MonteCarlo name="my_mc">
      <samplerInit>
        <limit>3</limit>
        <initialSeed>42</initialSeed>
      </samplerInit>
      <variable name="valve_area_t30_40">
        <distribution>va_t30_40_dist</distribution>
      </variable>
      <variable name="valve_area_t40_50">
        <distribution>va_t40_50_dist</distribution>
      </variable>
      <variable name="valve_area_t50_60">
        <distribution>va_t50_60_dist</distribution>
      </variable>
      <variable name="valve_area_t60_70">
        <distribution>va_t60_70_dist</distribution>
      </variable>
      <variable name="valve_area_t70_80">
        <distribution>va_t70_80_dist</distribution>
      </variable>-->
      <variable name="valve_area_t80_90">
        <distribution>va_t80_90_dist</distribution>
      </variable>
      <variable name="valve_area_t90_100">
        <distribution>va_t90_100_dist</distribution>
      </variable>
      <variable name="valve_area_t100_110">
        <distribution>va_t100_110_dist</distribution>
      </variable>
      <variable name="valve_area_t110_120">
        <distribution>va_t110_120_dist</distribution>
      </variable>
      <variable name="valve_area_t120_130">
        <distribution>va_t120_130_dist</distribution>
      </variable>
    </MonteCarlo>
  </Samplers>

  <DataObjects>
    <PointSet name="inputPlaceHolder">
      <Input>valve_area_t30_40, valve_area_t40_50, valve_area_t50_60, valve_area_t60_70,
valve_area_t70_80, valve_area_t80_90, valve_area_t90_100, valve_area_t100_110,
valve_area_t110_120, valve_area_t120_130</Input>
      <Output>OutputPlaceHolder</Output>
    </PointSet>
    <PointSet name="samples">
      <Input>valve_area_t30_40, valve_area_t40_50, valve_area_t50_60, valve_area_t60_70,
valve_area_t70_80, valve_area_t80_90, valve_area_t90_100, valve_area_t100_110,
valve_area_t110_120, valve_area_t120_130</Input>
      <Output>BOP.obj.y, Time</Output>
    </PointSet>
    <PointSet name="samplesROM">
```

```
      <Input>valve_area_t30_40, valve_area_t40_50, valve_area_t50_60, valve_area_t60_70,
valve_area_t70_80, valve_area_t80_90, valve_area_t90_100, valve_area_t100_110,
valve_area_t110_120, valve_area_t120_130</Input>
      <Output>BOP.obj.y</Output>
    </PointSet>
    <HistorySet name="histories">
      <Input>valve_area_t30_40, valve_area_t40_50, valve_area_t50_60, valve_area_t60_70,
valve_area_t70_80, valve_area_t80_90, valve_area_t90_100, valve_area_t100_110,
valve_area_t110_120, valve_area_t120_130</Input>
      <Output>BOP.obj.y, BOP.math_Exp.y, Time</Output>
      <options>
        <pivotParameter>Time</pivotParameter>
      </options>
    </HistorySet>
  </DataObjects>

  <OutStreams>
    <Print name="samples">
      <type>csv</type>
      <source>samples</source>
    </Print>
    <Print name="histories">
      <type>csv</type>
      <source>histories</source>
    </Print>
  </OutStreams>
</Simulation>
```