Production Cost Modeling with PowerSimulations.jl

Clayton Barrows,
National Renewable Energy Lab,
clayton.barrows@nrel.gov

This work is licensed under a Creative Commons Attribution 4.0 International License.
Introduction

PowerSimulations.jl supports simulations that consist of sequential optimization problems where results from previous problems inform subsequent problems. Otherwise known as production cost modeling. Additionally, several PowerSimulations.jl supports several other types of power system simulations:
Dependencies
Dependencies

In [17]:
1. `using SIIPExamples`
2. `using PowerSystems`
3. `using PowerSimulations`
4. `using Xpress`
5. `solver = optimizer_with_attributes(Xpress.Optimizer, "MIPRELSTOP" => 0.05, "OUTPU`
Data

PowerSystems.jl supports parsers for a few standard power system data formats:

- MATPOWER
- PTI network files in the .raw format that follow the PSS(R)E v33
- Tabular data (CSV)

The RTS-GMLC is published as a set of .csv files. So we can use the tabular data parsing support of PowerSystems.jl to read it.
```python
load_rts();
sys
```

Out[5]:

**System**

**Base Power**: 100.0

**Components**

**Num components**: 434

15 rows × 3 columns

<table>
<thead>
<tr>
<th>ConcreteType</th>
<th>SuperTypes</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>AggregationTopology &lt;: Topology &lt;: Component &lt;: PowerSystemType &lt;: InfrastructureSystemsType &lt;: Any</td>
<td>3</td>
</tr>
<tr>
<td>Area</td>
<td>Area</td>
<td>3</td>
</tr>
<tr>
<td>Bus</td>
<td>Topology &lt;: Component &lt;: PowerSystemType &lt;: InfrastructureSystemsType &lt;: Any</td>
<td>73</td>
</tr>
<tr>
<td>GenericBattery</td>
<td>Storage &lt;: StaticInjection &lt;: Device &lt;: Component &lt;: PowerSystemType &lt;: InfrastructureSystemsType &lt;: Any</td>
<td>1</td>
</tr>
<tr>
<td>HVDCLine</td>
<td>DCBranch &lt;: Branch &lt;: Device &lt;: Component &lt;: PowerSystemType &lt;: InfrastructureSystemsType &lt;: Any</td>
<td>1</td>
</tr>
</tbody>
</table>
Production Cost Modeling

PowerSimulations.jl is designed to flexibly build and execute sequential optimization problems. This example shows a straightforward representation of a day-ahead market clearing simulation with unit commitment. More complex examples are available in SIIPExamples.jl.
Define the problem formulation

First, we need to define how to represent each device type in the System using an OperationsProblemTemplate:
Define the problem formulation

First, we need to define how to represent each device type in the System using an OperationsProblemTemplate:

```
In [7]: uc_template = make_uc_template(network = DCPowerModel)

Out[7]:
Operations Problem Specification

transmission: DCPowerModel

devices:
  ILoads:
    device_type = InterruptibleLoad
    formulation = InterruptiblePowerLoad
  HydroROR:
    device_type = HydroDispatch
```
Define the day-ahead market model

- A **Stage** defines a model using the `OperationsProblemTemplate` and the `System` data.
- Users can create any number of **Stage**s along with control over how information flows inter and intra stage executions.
Define the day-ahead market model

- A **Stage** defines a model using the **OperationsProblemTemplate** and the **System** data.
- Users can create any number of **Stages** along with control over how information flows inter and intra stage executions.

```python
In [8]: stage_def = Dict("UC" => Stage(UnitCommitmentProblem, uc_template, sys, solver))

Out[8]: Dict(String,Stage{UnitCommitmentProblem}) with 1 entry:
"UC" => Stage()...
```
Sequencing

The stage problem length, look-ahead, and other details surrounding the temporal sequencing of stages are controlled using the `order`, `horizons`, and `intervals` arguments.

- `order::Dict(Int, String)` : the hierarchical order of stages in the simulation
- `horizons::Dict(String, Int)` : defines the number of time periods in each stage (problem length)
- `intervals::Dict(String, Dates.Period)` : defines the interval with which each problem advances after each execution
Sequencing

The stage problem length, look-ahead, and other details surrounding the temporal sequencing of stages are controlled using the `order`, `horizons`, and `intervals` arguments.

- `order::Dict(Int, String)` : the hierarchical order of stages in the simulation
- `horizons::Dict(String, Int)` : defines the number of time periods in each stage (problem length)
- `intervals::Dict(String, Dates.Period)` : defines the interval with which stage problems advance after each execution
Simulation

Now, we can build and execute a simulation using the `SimulationSequence` and `Stages` that we've defined.
Simulation

Now, we can build and execute a simulation using the 
SimulationSequence
and Stages that we've defined.

In [10]:
1   sim = Simulation(name = "rts-test",
2                     steps = 2,
3                     stages = stage_def,
4                     stages_sequence = DA_sequence,
5                     simulation_folder = rts_dir,
6                     initial_time = Dates.DateTime("2020-04-07T00:00:00"))
7   build!(sim)
Execute simulation
Execute simulation

In [11]: `sim_results = execute!(sim)`

Executing Step 1
Executing Step 2
Welcome to the CBC MILP Solver
Version: 2.10.3
Build Date: Oct 7 2019

command line - Cbc_C_Interface -ratioGap 0.5 -logLevel 1 -solve -quit (default strategy 1)
ratioGap was changed from 0 to 0.5
Continuous objective value is 1.36059e+06 - 0.73 seconds
Cgl0004I processed model has 12968 rows, 26800 columns (5314 integer (5314 of which binary)) and 62564 elements
Cbc0045I Trying just fixing integer variables (and fixingish SOS).
Cbc0045I MIPStart solution provided values for 9060 of 5314 integer variables, 139 variables are still fractional.
Cbc0038I Full problem 12968 rows 26800 columns, reduced to 12968 rows 26800 columns - too large
Cbc0045I Mini branch and bound defined values for remaining variables in 0.11 seconds
Analysis

PowerSimulations.jl natively populates simulation results in a struct of DataFrames.
## Analysis

PowerSimulations.jl natively populates simulation results in a struct of DataFrames.

```python
In [12]: uc_results = load_simulation_results(sim_results, "UC")
```

### Results

<table>
<thead>
<tr>
<th>Time</th>
<th>322_CT_6</th>
<th>321_CC_1</th>
<th>202_STEAM_3</th>
<th>315_STEAM_1</th>
<th>223_CT_4</th>
<th>123_STEAM_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>DateTime</td>
<td>Float64</td>
<td>Float64</td>
<td>Float64</td>
<td>Float64</td>
<td>Float64</td>
<td>Float64</td>
</tr>
<tr>
<td>1</td>
<td>2020-04-07T00:00:00 0.0</td>
<td>1.7</td>
<td>0.3</td>
<td>0.0</td>
<td>0.0</td>
<td>0.62</td>
</tr>
<tr>
<td>2</td>
<td>2020-04-07T01:00:00 0.0</td>
<td>1.7</td>
<td>0.3</td>
<td>0.0</td>
<td>0.0</td>
<td>0.62</td>
</tr>
<tr>
<td>3</td>
<td>2020-04-07T02:00:00 0.0</td>
<td>1.7</td>
<td>0.3</td>
<td>0.0</td>
<td>0.0</td>
<td>0.62</td>
</tr>
</tbody>
</table>
Plotting

The (new) PowerGraphics.jl package has some standard plotting capabilities based on the results produced by PowerSimulations.jl.
Plotting

The (new) `PowerGraphics.jl` package has some standard plotting capabilities based on the results produced by `PowerSimulations.jl`.

```julia
fuel_plot(uc_results, sys)
```
What’s Next?