Production Cost Modeling with PowerSimulations.jl

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

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]:
using SIIPExamples
using PowerSystems
using PowerSimulations
using Xpress
solver = optimizer_with_attributes(Xpress.Optimizer, "MIPRELSTOP" => 0.05, "OUTPU
using PowerGraphics
```

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.

In [5]:

```
load_rts();
sys
```

Out[5]:

System

Base Power: 100.0

Components

Num components: 434

15 rows × 3 columns

	ConcreteType	SuperTypes	Count
	String	String	Int64
1	Area	AggregationTopology <: Topology <: Component <: PowerSystemType <: InfrastructureSystemsType <: Any	3
2	Bus	Topology <: Component <: PowerSystemType <: InfrastructureSystemsType <: Any	73
3	GenericBattery	Storage <: StaticInjection <: Device <: Component <: PowerSystemType <: InfrastructureSystemsType <: Any	1
4	HVDCLine	DCBranch <: Branch <: Device <: Component <: PowerSystemType <: InfrastructureSystemsType <: Any	1

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]:  1  uc_template = make_uc_template(network = DCPPowerModel)

Out[7]:
         Operations Problem Specification
         ============================================

           transmission:  DCPPowerModel
         ============================================
           devices:
               ILoads:
                 device_type = InterruptibleLoad
                 formulation = InterruptiblePowerLoad
               HydroROR:
                 device_type = HydroDispatch
                 formulation = HydroFixed
```

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 `Stage`s along with control over how information flows inter and intra stage executions.

```
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 stage problems advance 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 `Stage`s that we've defined.

Simulation

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

```
In [10]:
sim = Simulation(name = "rts-test",
                 steps = 2,
                 stages = stage_def,
                 stages_sequence = DA_sequence,
                 simulation_folder = rts_dir,
                 initial_time = Dates.DateTime("2020-04-07T00:00:00"))
build!(sim)
```

Execute simulation

Execute simulation

In [11]:
```julia
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 w
hich 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 col
umns - too large
Cbc0045I Mini branch and bound defined values for remaining variables in 0.11 s
econds
```

Analysis

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

Analysis

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

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

Out[12]:

Results

P__ThermalStandard

48 rows × 77 columns (omitted printing of 70 columns)

	Time	322_CT_6	321_CC_1	202_STEAM_3	315_STEAM_1	223_CT_4	123_STEAM_2
	DateTime	Float64	Float64	Float64	Float64	Float64	Float64
1	2020-04-07T00:00:00	0.0	1.7	0.3	0.0	0.0	0.62
2	2020-04-07T01:00:00	0.0	1.7	0.3	0.0	0.0	0.62
3	2020-04-07T02:00:00	0.0	1.7	0.3	0.0	0.0	0.62

Plotting

The (new) PowerGraphics.jl package has some standard plotting capabilitites based on the results produduced by PowerSimulations.jl

Plotting

The (new) PowerGraphics.jl package has some standard plotting capabilitites based on the results produduced by PowerSimulations.jl

```
In [16]:  fuel_plot(uc_results, sys)
```

What's Next?