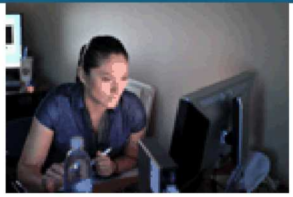
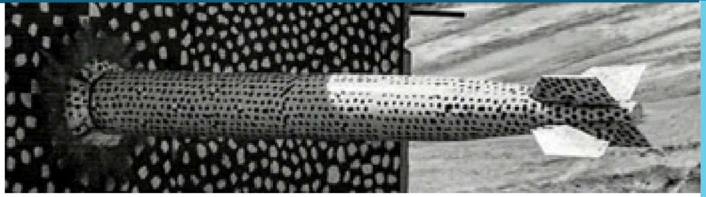
SAND2019-13733C

Scalable Models for Multi-Sector Resources





PRESENTED BY

Ben Knueven

Talk Overview

EGRET: Electrical Grid Research and Engineering Tools

EGRET is a Python-based package for electrical grid optimization based on the Pyomo optimization modeling language. EGRET is designed to be friendly for performing high-level analysis (e.g., as an engine for solving different optimization formulations), while also providing flexibility for researchers to rapidly explore new optimization formulations.

Major features:

- Expression and solution of unit commitment problems, including full ancillary service stack
- Expression and solution of economic dispatch (optimal power flow) problems (e.g, DCOPF, ACOPF)
- Library of different problem formulations and approximations
- Generic handling of data across model formulations
- Declarative model representation to support formulation development

EGRET is available under the BSD License at https://github.com/grid-parity-exchange/Egret

EGRET Prerequisites

Python >= 3.6

Pyomo

- www.pyomo.org
- Version >= 5.6
- Use "conda install" or "pip install"

Solver

- Pyomo is an Algebraic Modeling Language (AML)
- EGRET expresses unit commitment and dispatch models using the Pyomo AML
- Need a mixed-integer linear (commitment) and linear (dispatch) solver
  - Commercial: Gurobi, CPLEX, Xpress
  - Academic: CBC, GLPK
  - In Between: SCIP

EGRET: Commentary

Development of the commitment and dispatch models in EGRET started in 2012
- Funded by an ARPA-E GENI project (scalable stochastic unit commitment)

Predecessor models were validated against models in Alstom's Market Management System
- 2 months on-site effort at Alstom

C. Laird, M. Bynum, and A. Castillo (SNL) have led ACOPF-related efforts in EGRET
- Not the topic of today's talk

Documentation is lagging capability by a significant margin
- All function, no flash – yes, we are terrible at PR

Still migrating advanced models and capabilities into EGRET
- E.g., from ASU/Sandia/Nexant ARPA-E NODES project

Unit Commitment Modeling

The Unit Commitment Problem (1)

The Unit Commitment Problem (UC) is a large-scale mixed-integer nonlinear program for finding a low-cost operating schedule for power generators.

These problems typically have quadratic objective functions and non-linear, non-convex transmission constraints.

- Typically both of these are linearized

Starting in 2005 with PJM, market operators in the United States have transitioned from using Lagrangian relaxation to solve UC to using mixed-integer programming (MIP) and a commercial solver such as CPLEX, Gurobi, or Xpress.

- MIPs usually have many equivalent formulations, and UC is no exception.

The day-ahead problem has an hourly time horizon which is solved for 36 to 48 hours ahead to prevent end-of-horizon effects, and has hundreds to thousands of generators and up to tens of thousands of buses.

In practice, it is desirable to have a UC solution in 10 to 15 minutes.

The Unit Commitment Problem (2)

$$
\begin{aligned}
\min \sum_{g\in\mathcal{G}} \sum_{t\in\mathcal{T}} c_g(t) & \\
\text{s.t.} \ \sum_{g\in\mathcal{G}} A_g(p_g, \overline{p}_g, u_g) + N(s) = L & \\
(p_g, \overline{p}_g, u_g, c_g) \in \Pi_g & \qquad \forall g \in \mathcal{G}.
\end{aligned}
$$

UC is that of minimizing system operating costs subject to the system constraints and the technical constraints of the generators.

Generator technical constraints

- Convex (piecewise linear) production costs
- Minimum and maximum output levels
- Ramping constraints
- Minimum up/down time
- Downtime dependent startup costs

Using a performant formulation for all of these components is critically important for solving these problems efficiently!

Some Code Examples…

Example Code: Constructing and Solving a UC Problem

```
from egret.parsers.pglib_uc_parser import create_ModelData
from egret.models.unit_commitment import solve_unit_commitment

md = create_ModelData('./download/pglib-uc-master/ca/2014-09-01_reserves_0.json')

md_sol = solve_unit_commitment(md, 'gurobi', mipgap=0.0, timelimit=300)

print('Objective value:', md_sol.data['system']['total_cost'])
```

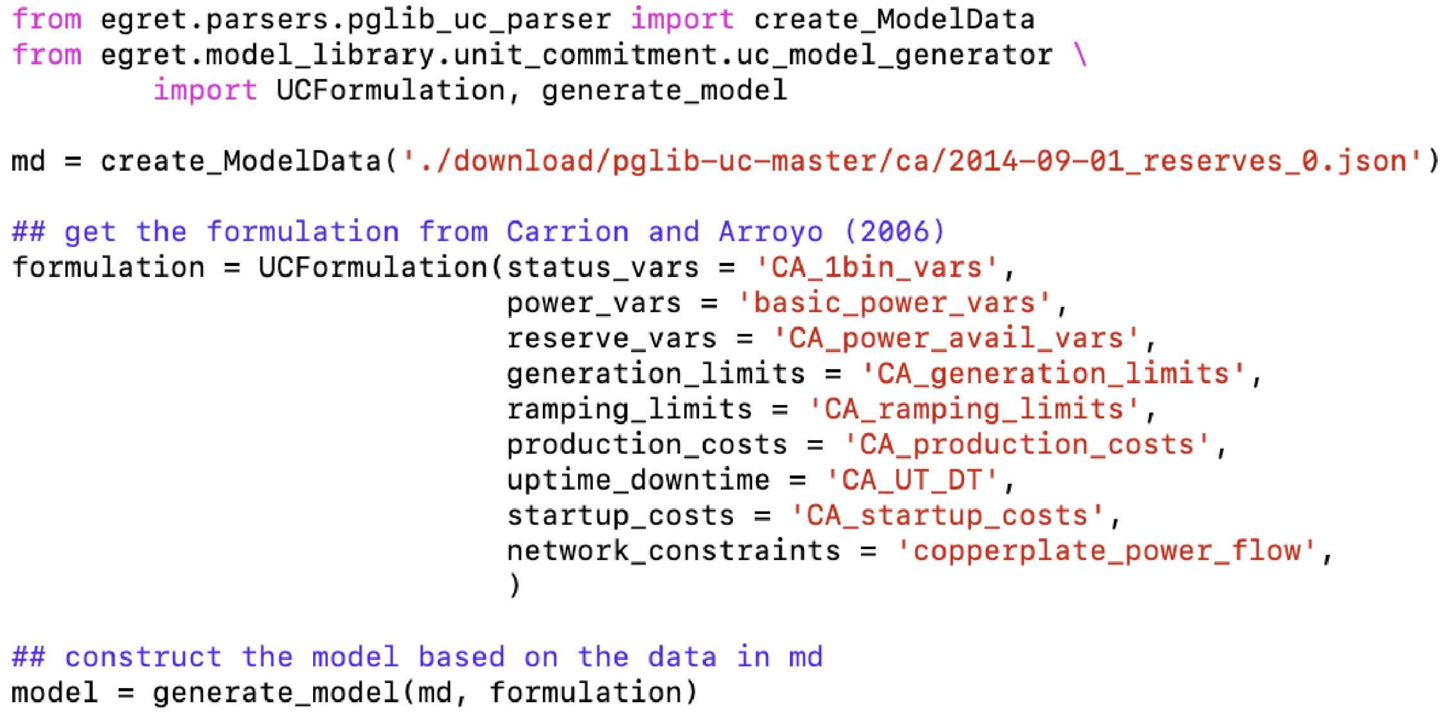
Example Code: Selecting an included formulation

```
from egret.parsers.pglib_uc_parser import create_ModelData
from egret.models.unit_commitment import solve_unit_commitment, \
        create_KOW_unit_commitment_model

md = create_ModelData('./download/pglib-uc-master/ca/2014-09-01_reserves_0.json')

md_sol = solve_unit_commitment(md, 'gurobi', mipgap=0.0, timelimit=300,
                                uc_model_generator=create_KOW_unit_commitment_model)

print('Objective value:', md_sol.data['system']['total_cost'])
```

Example Code: Specifying your own formulation from component library

```python
from egret.parsers.pglib_uc_parser import create_ModelData
from egret.models.unit_commitment import solve_unit_commitment
from egret.model_library.unit_commitment.uc_model_generator import \
        UCFormulation, generate_model

def create_my_unit_commitment_model(model_data,
                                    network_constraints='ptdf_power_flow',
                                    relaxed=False, **kwargs):

    my_uc_formulation = UCFormulation(status_vars = 'garver_3bin_vars',
                                      power_vars = 'garver_power_vars',
                                      reserve_vars = 'MLR_reserve_vars',
                                      generation_limits = 'pan_guan_gentile_KOW_generation_limits',
                                      ramping_limits = 'damcikurt_ramping',
                                      production_costs = 'KOW_production_costs_super_tight',
                                      uptime_downtime = 'rajan_takriti_UT_DT_2bin',
                                      startup_costs = 'KOW_startup_costs',
                                      network_constraints = network_constraints,
                                      )
    return generate_model(model_data, my_uc_formulation, relax_binaries=relaxed, **kwargs)

md = create_ModelData('./download/pglib-uc-master/ca/2014-09-01_reserves_0.json')

md_sol = solve_unit_commitment(md, 'gurobi', mipgap=0.0, timelimit=300,
                                uc_model_generator=create_my_unit_commitment_model)

print('Objective value:', md_sol.data['system']['total_cost'])
```

Example Code: A Modular Framework for UC Formulations



```
from egret.parsers.pglib_uc_parser import create_ModelData
from egret.model_library.unit_commitment.uc_model_generator \
        import UCFormulation, generate_model

md = create_ModelData('./download/pglib-uc-master/ca/2014-09-01_reserves_0.json')

## get the formulation from Carrion and Arroyo (2006)
formulation = UCFormulation(status_vars = 'CA_1bin_vars',
                            power_vars = 'basic_power_vars',
                            reserve_vars = 'CA_power_avail_vars',
                            generation_limits = 'CA_generation_limits',
                            ramping_limits = 'CA_ramping_limits',
                            production_costs = 'CA_production_costs',
                            uptime_downtime = 'CA_UT_DT',
                            startup_costs = 'CA_startup_costs',
                            network_constraints = 'copperplate_power_flow',
                            )

## construct the model based on the data in md
model = generate_model(md, formulation)
```

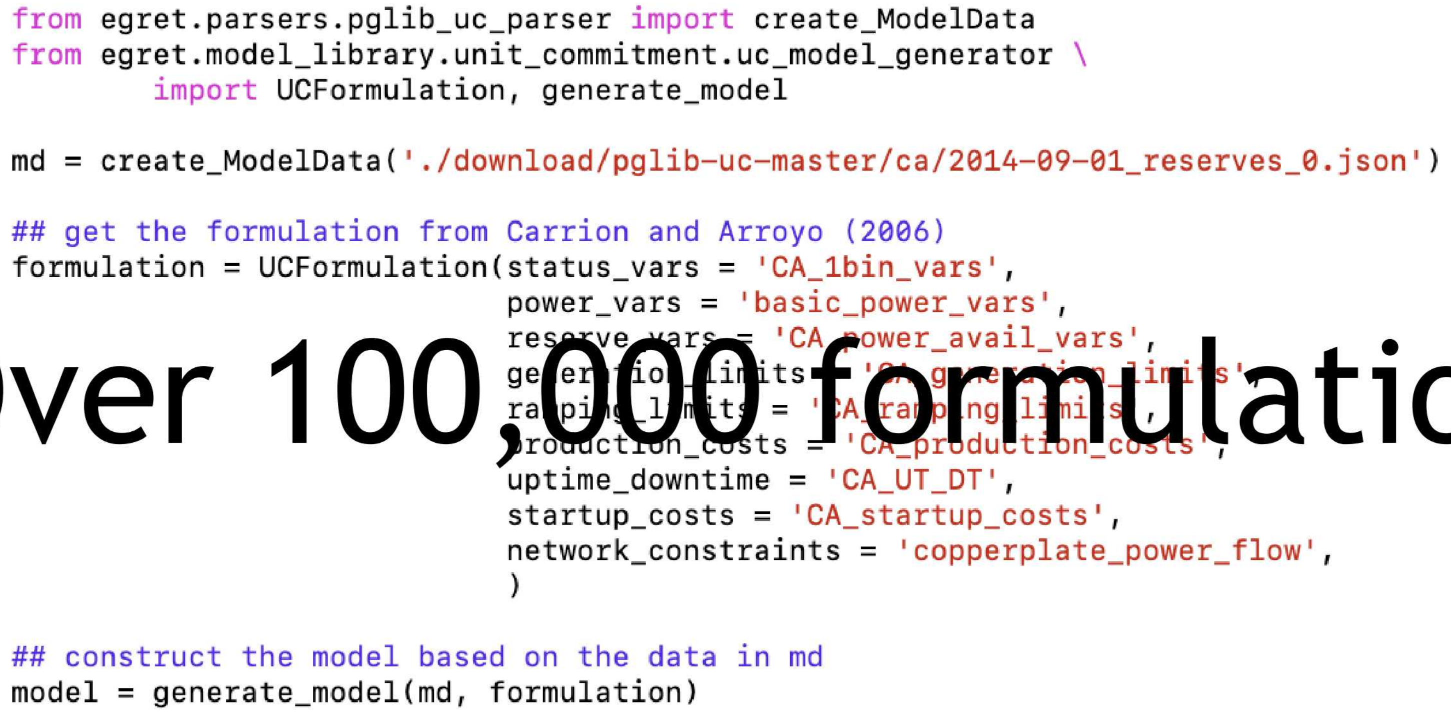
This instantiates a Pyomo `ConcreteModel` (`model`) based on the data provided in the object `md`, which can be used as part of a script.

The eights components of `UCFormulation` can be changed as easily as modifying a string in this file. Runtime checks to ensure incompatible components are not combined.

Number of implemented formulations per component:

- `status_vars`: 5
- `power_vars`: 3
- `reserve_vars`: 4
- `generation_limits`: 9
- `ramping_limits`: 8
- `production_costs`: 12
- `uptime_downtime`: 5
- `startup_costs`: 9
- `network_constraints`: 4

Example Code: A Modular Framework for UC Formulations



```
from egret.parsers.pglib_uc_parser import create_ModelData
from egret.model_library.unit_commitment.uc_model_generator \
        import UCFormulation, generate_model

md = create_ModelData('./download/pglib-uc-master/ca/2014-09-01_reserves_0.json')

## get the formulation from Carrion and Arroyo (2006)
formulation = UCFormulation(status_vars = 'CA_1bin_vars',
                            power_vars = 'basic_power_vars',
                            reserve_vars = 'CA_power_avail_vars',
                            generation_limits = 'CA_generation_limits',
                            ramping_limits = 'CA_ramping_limits',
                            production_costs = 'CA_production_costs',
                            uptime_downtime = 'CA_UT_DT',
                            startup_costs = 'CA_startup_costs',
                            network_constraints = 'copperplate_power_flow',
                            )

## construct the model based on the data in md
model = generate_model(md, formulation)
```

Over 100,000 formulations

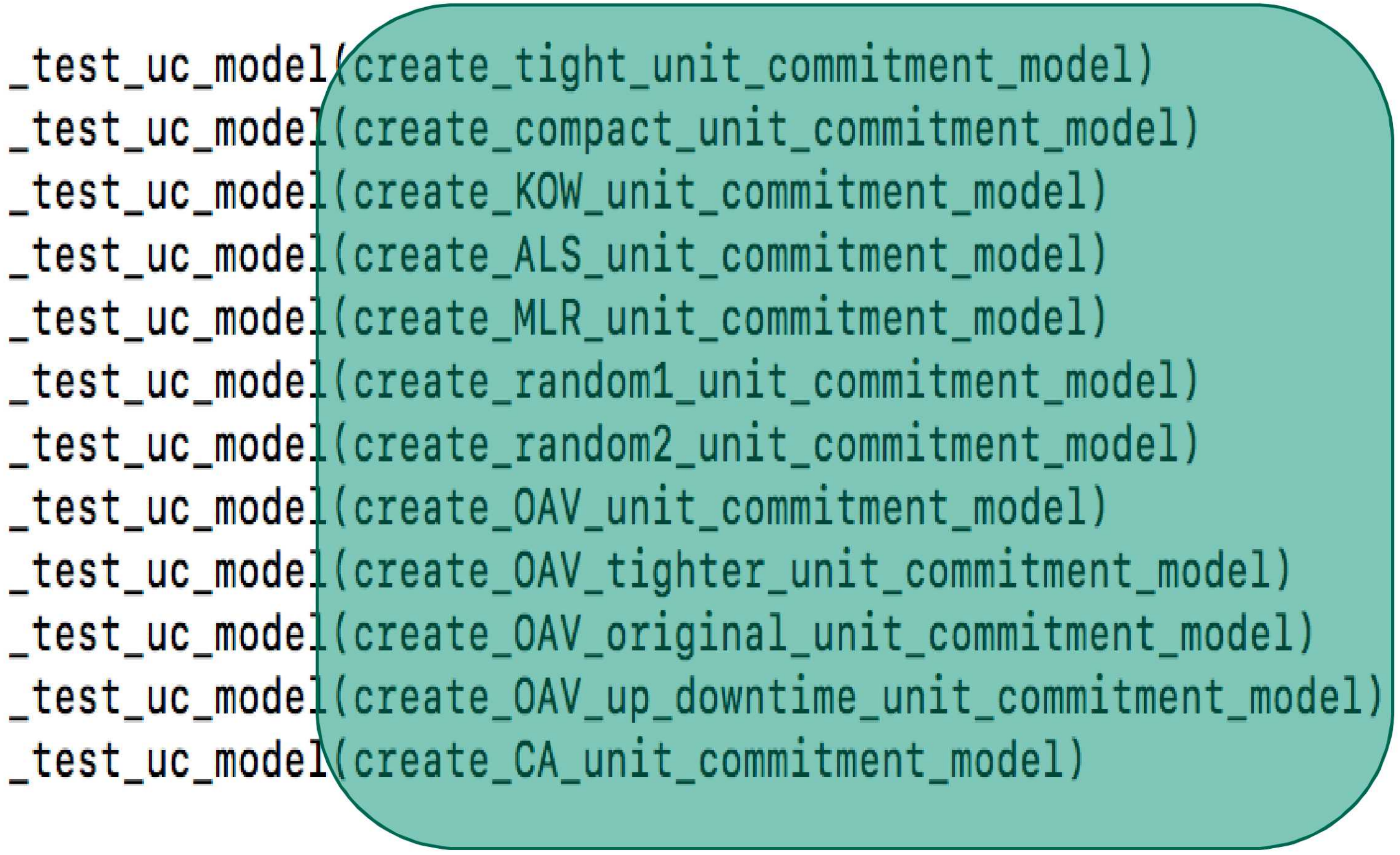
This instantiates a Pyomo `ConcreteModel` (`model`) based on the data provided in the object `md`, which can be used as part of a script.

The eights components of `UCFormulation` can be changed as easily as modifying a string in this file. Runtime checks to ensure incompatible components are not combined.

Number of implemented formulations per component:

- `status_vars`: 5
- `power_vars`: 3
- `reserve_vars`: 4
- `generation_limits`: 9
- `ramping_limits`: 8
- `production_costs`: 12
- `uptime_downtime`: 5
- `startup_costs`: 9
- `network_constraints`: 4

Example Code: Alternative UC Model Build Functions

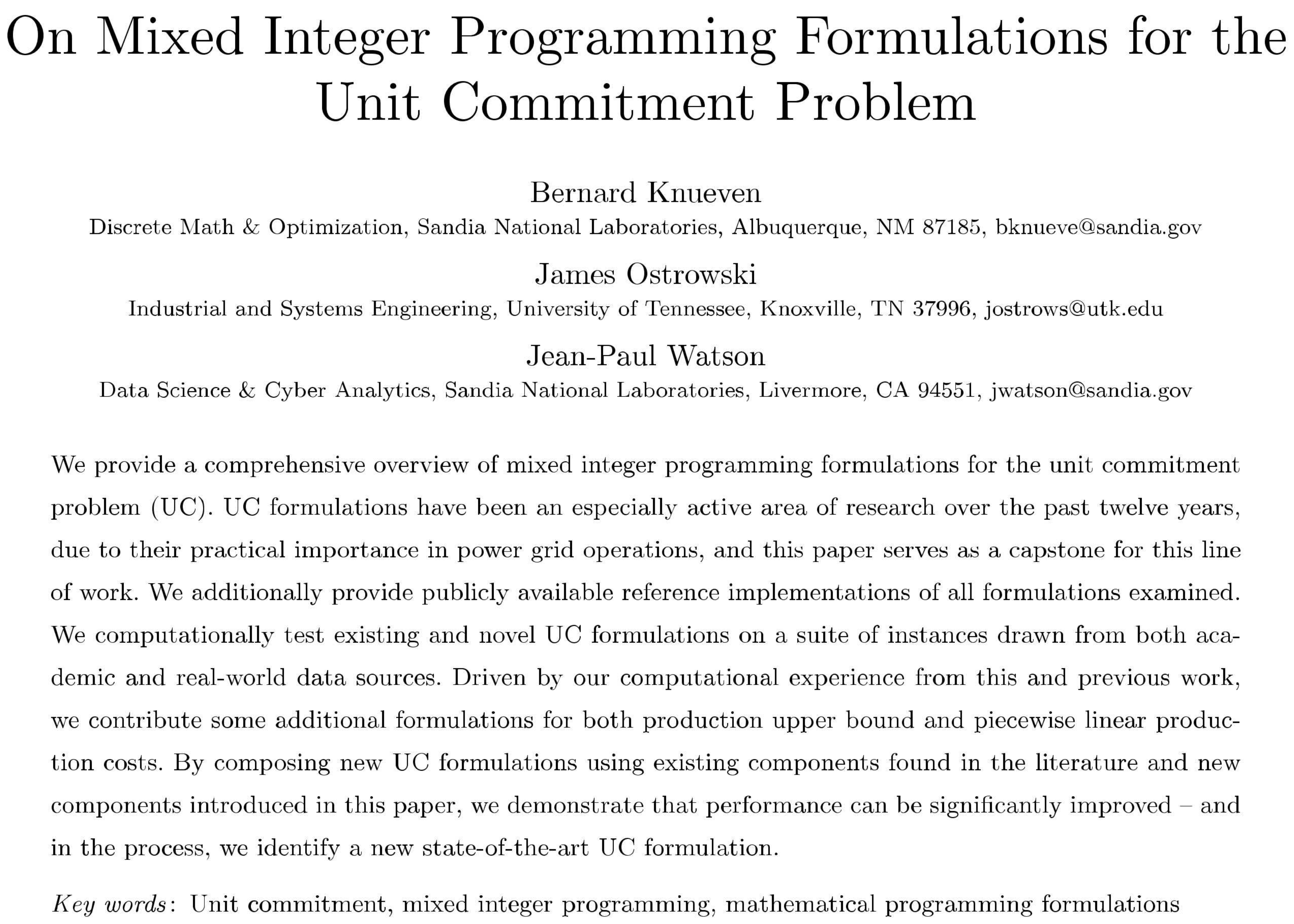


```
_test_uc_model(create_tight_unit_commitment_model)
_test_uc_model(create_compact_unit_commitment_model)
_test_uc_model(create_KOW_unit_commitment_model)
_test_uc_model(create_ALS_unit_commitment_model)
_test_uc_model(create_MLR_unit_commitment_model)
_test_uc_model(create_random1_unit_commitment_model)
_test_uc_model(create_random2_unit_commitment_model)
_test_uc_model(create_OAV_unit_commitment_model)
_test_uc_model(create_OAV_tighter_unit_commitment_model)
_test_uc_model(create_OAV_original_unit_commitment_model)
_test_uc_model(create_OAV_up_downtime_unit_commitment_model)
_test_uc_model(create_CA_unit_commitment_model)
```

Library of built-in UC model builders

From: `egret/models/tests/test_unit_commitment.py`

Paper



On Mixed Integer Programming Formulations for the Unit Commitment Problem

Bernard Knueven
Discrete Math & Optimization, Sandia National Laboratories, Albuquerque, NM 87185, bknueve@sandia.gov

James Ostrowski
Industrial and Systems Engineering, University of Tennessee, Knoxville, TN 37996, jostrows@utk.edu

Jean-Paul Watson
Data Science & Cyber Analytics, Sandia National Laboratories, Livermore, CA 94551, jwatson@sandia.gov

We provide a comprehensive overview of mixed integer programming formulations for the unit commitment problem (UC). UC formulations have been an especially active area of research over the past twelve years, due to their practical importance in power grid operations, and this paper serves as a capstone for this line of work. We additionally provide publicly available reference implementations of all formulations examined. We computationally test existing and novel UC formulations on a suite of instances drawn from both academic and real-world data sources. Driven by our computational experience from this and previous work, we contribute some additional formulations for both production upper bound and piecewise linear production costs. By composing new UC formulations using existing components found in the literature and new components introduced in this paper, we demonstrate that performance can be significantly improved – and in the process, we identify a new state-of-the-art UC formulation.

Key words: Unit commitment, mixed integer programming, mathematical programming formulations

Available on *Optimization Online*: http://www.optimization-online.org/DB_HTML/2018/11/6930.html

EGRET: Other Capabilities Included

Full ancillary service stack

- Seven products
  - Reg-up, reg-down, spinning, non-spinning, supplemental, flex-up, and flex-down reserves

Storage

- Generic, full-fidelity models

Zonal definition support

- Relevant for reserve modeling

Network

- B-theta
- PTDF, PTDF+Losses (full and lazy)

Fuel Constraints

- Dual-fuel generator model

Dual-Fuel and Fuel-Constrained Unit Commitment

Fuel Constrained Modeling in EGRET

Current capability is an "instantaneous fuel supply", which is to model fuel availability for a specific time interval.

Exemplar is a natural gas hub, but could also be used for regional natural gas availability.

An arbitrary subset of generators can be attached to a fuel supply, though a single-fuel generator can only be attached to one.

Fuel consumed by generators (primary and auxiliary) at time *t* must be less than availability

$$\sum_{g \in \mathcal{G}_P(i)} f_g^P(t) + \sum_{g \in \mathcal{G}_A(i)} f_g^A(t) \leq S_i(t)$$

Modular modeling paradigm allows the easy addition of other fuel supply models, e.g., on-site oil, higher fidelity NG model.

Dual Fuel Modeling in EGRET

Heat rates define the fuel consumption (regardless of fuel type) for production, start-up, and no-load

Tracks fuel consumption for both the primary (P) and auxiliary (A) fuel, which then gets costed in the objective by the fuel cost

$$f_g^P(t) + f_g^A(t) = \Delta(t) F_g^0 u_g(t) + \sum_{l \in \mathcal{L}_g} \Delta(t) F_g^l p_g^l(t) + \sum_{s \in \mathcal{S}_g} F_g^s \delta_g^s(t)$$

$$C_g^{F,P} f_g^P(t) + C_g^{F,A} f_g^A(t) + C_g^{u,NF} u_g(t) + C_g^{v,NF} v_g(t)$$

Can model both on-line fuel switching and off-line only fuel switching – the above model is enough for a unit which can do arbitrary fuel blending.

Dual-fuel units have their base start-up costs, no-load costs, and piecewise production costs replaced with this specialized dual-fuel model.

- Could replace other components as well, e.g., ramp-rates, min/max generation levels.

If $f_g^A(t)$ is 0, this is a model for tracking fuel-consumption for a normal single-fuel generator

Single-Fire Dual Fuel Units

If a unit cannot co-fire fuels, we can add binary variables $u_g^P(t)$ and $u_g^A(t)$ for fuel modes primary (P) and auxiliary (A), along with constraints to restrict the generator to using no more than one fuel at a time.

$$u_g(t) = u_g^P(t) + u_g^A(t)$$

$$f_g^P(t) \leq \overline{F}_g u_g^P(t)$$

$$f_g^A(t) \leq \overline{F}_g u_g^A(t)$$

If the unit can only switch fuels while offline, we can add additional variables and constraint which track the start-up and shut-down fuel, thereby forcing one of the binary fuel indicators to be selected for the entire running cycle.

$$v_g(t) = v_g^P(t) + v_g^A(t)$$

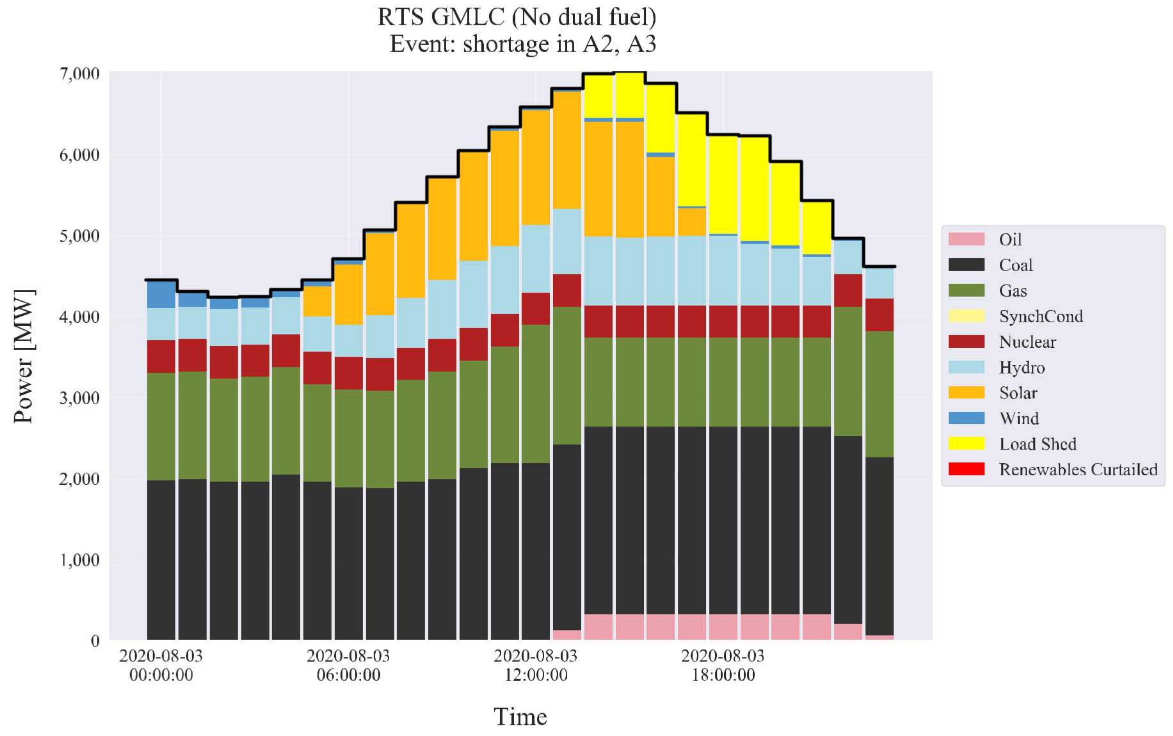
$$w_g(t) = w_g^P(t) + w_g^A(t)$$

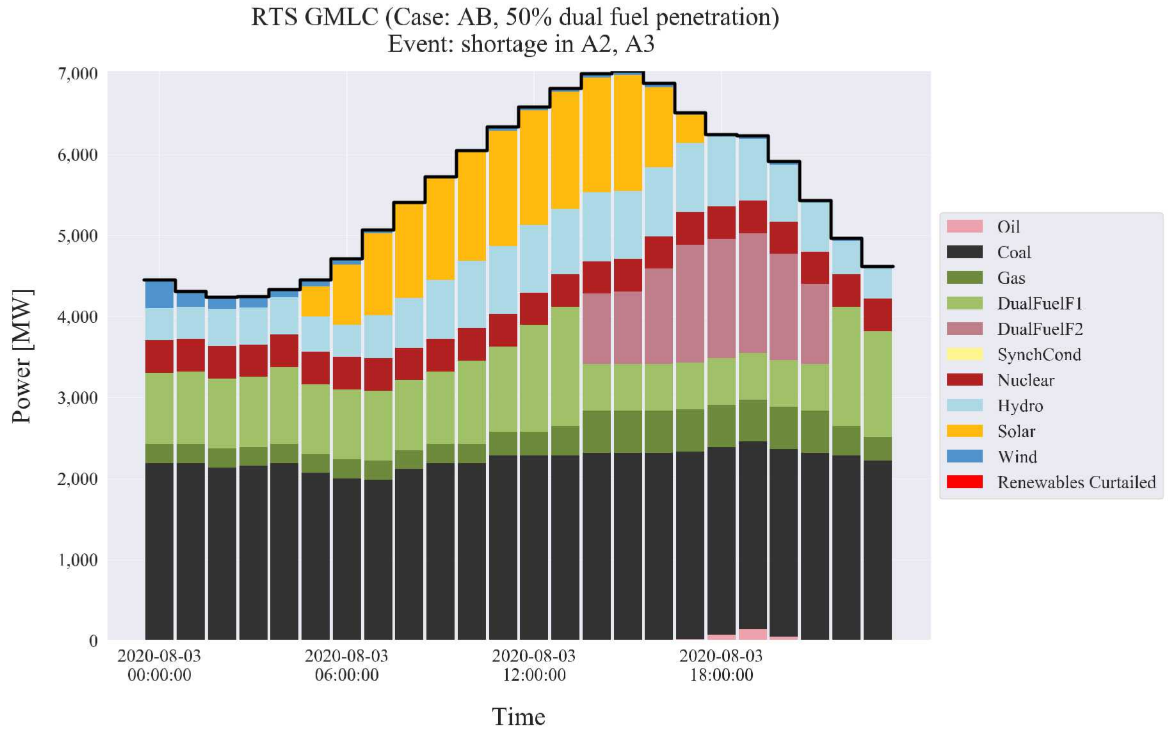
$$u_g^P(t) - u_g^P(t-1) = v_g^P(t) - w_g^P(t)$$

$$u_g^A(t) - u_g^A(t-1) = v_g^A(t) - w_g^A(t)$$

Some Preliminary Results

Augmented RTS-GMLC – suppose a NG shortage in two areas. Adding dual-fuel capability enhances resilience of the system to a NG shortage.





Current Practice for Dual-Fuel Units

Current ISO/RTO practices allows dual-fuel units to bid-in different cost curves, but the units (or market participants) decide when to execute their fuel switch.

This can be inefficient, as natural gas generators share some gas resources.

With some (moderate) insight into the gas supply network, the ISO/RTO could instruct market participants when to fuel switch based on shared fuel supply constraints.

- This can enhance reliability and resilience, especially when fuel supply is a binding constraint.

Future Development Directions

Intellectual investment in commitment / dispatch libraries is massive

- Decades of person-years invested in aggregate
- Very intricate knowledge of mathematical programming required for performance advances

Security Constraints

- N-1, sophisticated transmission constraint filtering

Drive toward open and transparent models is critical

- Necessary condition for V&V of power systems operations simulation models

Enhanced Fuel Supply Constrained Models

- Co-optimization of NG and BES
- Avoiding non-linearities of full NG system models