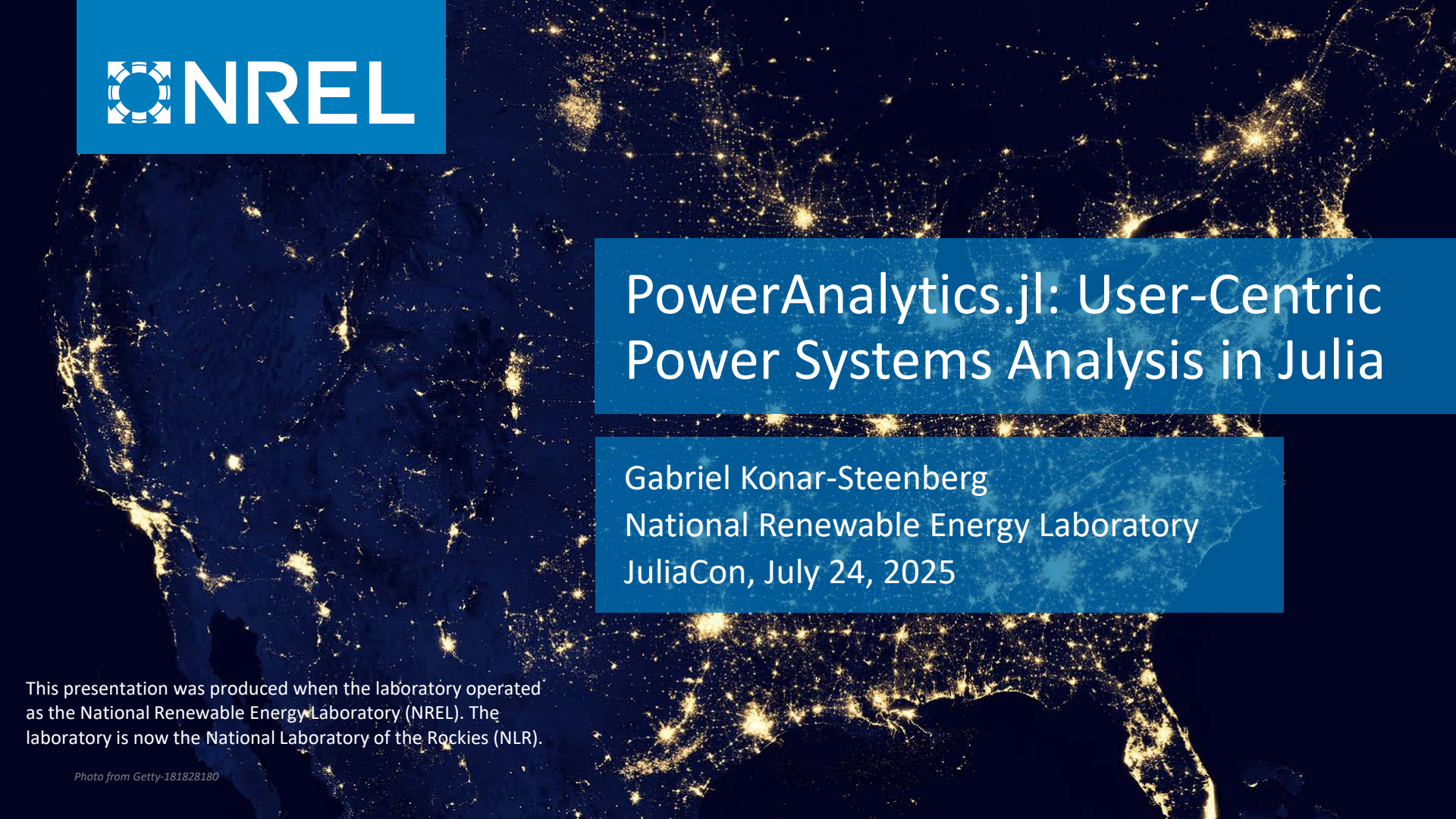


PowerAnalytics.jl: User-Centric Power Systems Analysis in Julia

Gabriel Konar-Steenberg
National Renewable Energy Laboratory
JuliaCon, July 24, 2025

This presentation was produced when the laboratory operated as the National Renewable Energy Laboratory (NREL). The laboratory is now the National Laboratory of the Rockies (NLR).

Photo from Getty-181828180

Contents

1 Introduction to Sienna

2 Gap and user needs

3 PowerAnalytics.jl package design

4 Design principles and techniques

5 Status and future work

6 Takeaways, acknowledgements

7 Q&A

Sienna

An open-source ecosystem for power system modeling, simulation and optimization

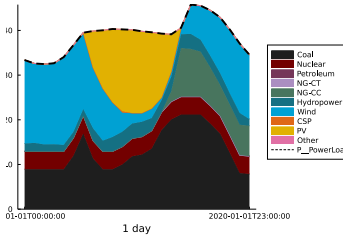
- 10,000+ downloads
- 30+ project usages
- 16 publications
- 200+ datasets

https://github.com/NREL-Sienna

Sienna\Data

Efficient intake and use of energy systems input data:

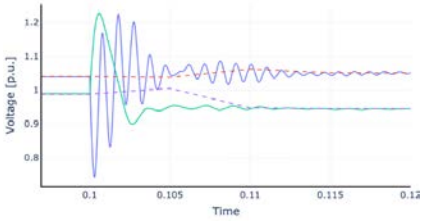
- Input data parsers
- Time series
- Quasi-static model data
- Dynamic model data
- DC and AC power-flow calculations
- Existing test systems



Sienna\Ops

Simulation of system scheduling, including multi-stage problems:

- Unit commitment, dispatch, AGC
- Reserve co-optimization
- Technology performance
- Market opportunities
- System reliability analysis
- Built-in analytics and graphics



Sienna\Dyn

Simulation of power system dynamic response to disturbances and contingencies:

- Small signal stability
- Full dynamic simulations
- Low inertia simulation capabilities
- Very fast computation

Existing Sienna Workflow

Bare bones schematic of a Sienna simulation:

```
using PowerSystems, PowerSystemCaseBuilder, PowerSimulations, StorageSystemsSimulations

# Set up a system
sys = build_system(PSISystems, "RTS_GMLC_DA_sys")
comp_storage = get_component(EnergyReservoirStorage, sys, "313_STORAGE_1")
set_input_active_power_limits!(comp_storage, (min = 0.0, max = 200.0))

# Set up a simulation
tplt = template_unit_commitment()
set_device_model!(tplt, DeviceModel(EnergyReservoirStorage, StorageDispatchWithReserves))
models = SimulationModels(;
  decision_models = [DecisionModel(tplt, sys; optimizer = solver, name = "UC")])
sequence = SimulationSequence(;
  models = models, ini_cond_chronology = InterProblemChronology())
sim = Simulation(; name = scenario, steps = 31, models = models, sequence = sequence,
  initial_time = DateTime("2020-07-01T00:00:00"), simulation_folder = output_folder)

# Build and run it
build!(sim)
execute!(sim)
```

Gap: PowerAnalytics Relationship to Existing Software

"Third-party" tools

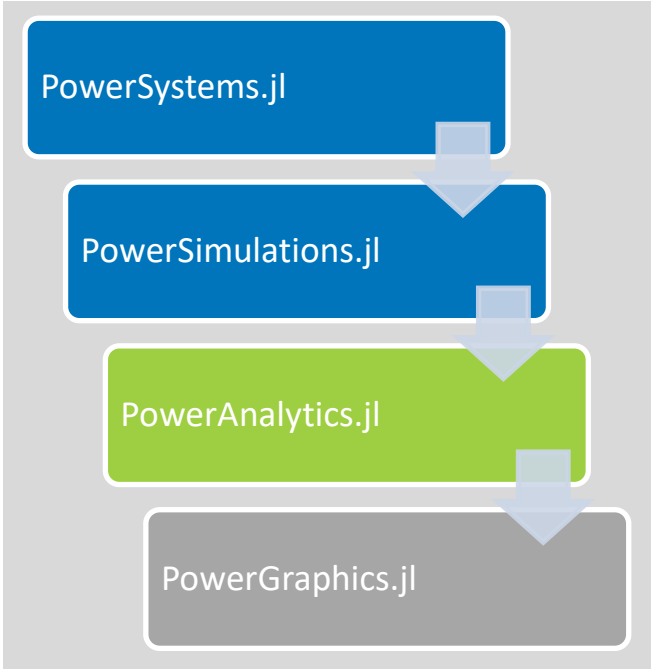
- Necessitates learning another "language"

DIY analytics from scratch

- Duplication, bugs, etc.

Old PowerAnalytics.jl

- A few hard-coded sets of calculations
- Limited aggregation capability



Package Design

Select components, compute metrics, reshape results

Main Concepts

ComponentSelector: flexibly express what part of the power system to analyze

- All solar plants and, separately, all wind plants
- Each solar plant larger than 15 MW, individually
- Loads in a particular zone, all together

Metric: what to compute, how to compute it, how to aggregate across space and time

- Power: look up power in results; sum across generators and time
- Integration: calculate power ÷ total load; sum across generators, weighted mean across time

DataFrame metadata: enables easy time aggregation and plotting

- Aggregate to day of week, hour of day, etc.
- Plot using built-in metadata for labels

ComponentSelector

A **ComponentSelector** is a **lazy**, partitioned, named, **source-independent** collection of Components.

```
julia> sel1 = make_selector(RenewableGen)  # renewables partitioned individually (:each)
InfrastructureSystems.TypeComponentSelector(RenewableGen, :each, "RenewableGen")

julia> get_components(sel1, sys1) # get all the components that match in system 1
RenewableGen Counts:
RenewableDispatch: 30
RenewableNonDispatch: 31

julia> get_components(sel1, sys2)  # and in system 2
RenewableGen Counts:
RenewableDispatch: 29
RenewableNonDispatch: 31

julia> get_name.(get_groups(sel1, sys1))  # get the groups by which sel1 partitions sys1
61-element Vector{String}:
 "RenewableDispatch__122_WIND_1"
 "RenewableDispatch__324_PV_3"
 ⋮
 "RenewableNonDispatch__118_RTPV_4"
```

ComponentSelector

A **ComponentSelector** is a lazy, **partitioned**, named, source-independent collection of Components.

```
julia> # now partition all generators by prime mover type
julia> sel2 = make_selector(Generator; groupby = get_prime_mover_type)
InfrastructureSystems.TypeComponentSelector(Generator, get_prime_mover_type, "Generator")

julia> get_components(sel2, sys1)
Generator Counts:
HydroDispatch: 1
HydroEnergyReservoir: 19
RenewableDispatch: 30
RenewableNonDispatch: 31
ThermalStandard: 76

julia> println(get_name.(get_groups(sel2, sys1)) ) # now groups are by prime mover type
["CT", "CC", "ST", "OT", "HY", "WT", "PVe", "CP"]

julia> get_components(first(get_groups(sel2, sys1)), sys1) # components in first group
ThermalStandard Counts:
ThermalStandard: 39
```

ComponentSelector

A **ComponentSelector** is a lazy, partitioned, **named**, source-independent collection of Components.

```
julia> # all photovoltaic renewable generators with rating above 15 MW
julia> sel3 = make_selector(
  gen -> get_rating(gen) > 15 && get_prime_mover_type(gen) == PrimeMovers.PVe,
  RenewableGen; name = "PV > 15 MW")
InfrastructureSystems.FilterComponentSelector(RenewableGen, var"#39#40"(), :each, "PV >
  15 MW")

julia> sel4 = make_selector(ThermalStandard; name = "All thermal std")
InfrastructureSystems.TypeComponentSelector(ThermalStandard, :each, "All thermal std")

julia> sel5 = make_selector(sel3, sel4); # compose a new selector

julia> get_name.(get_groups(sel5, sys1)) # it is grouped by its constituent selectors
("PV > 15 MW", "All thermal std")
```

Other features: selectors from a config file, regrouping, etc.

Metric

A **Metric** is an automatically aggregable computation on **results**, **often indexed** by a ComponentSelector.

```
julia> # list the available built-in metrics
julia> names(PowerAnalytics.Metrics)
25-element Vector{Symbol}:
 :Metrics
 :calc_active_power
 ⋮
 :calc_total_cost

# call metrics on ComponentSelector and results, aggregation happens automatically
julia> calc_active_power(
  make_selector(ThermalStandard; groupby = get_prime_mer_type), results1)
744×5 DataFrame
 Row │ DateTime            CT       CC       ST       OT
     │ DateTime            Float64  Float64  Float64  Float64
─────┼──────────────────────────────────────────────────────────
   1 │ 2020-07-01T00:00:00     0.0  836.767  2360.67      0.0
   2 │ 2020-07-01T01:00:00     0.0  680.0    2011.6       0.0
   ⋮ │          ⋮              ⋮        ⋮        ⋮        ⋮
 744 │ 2020-07-31T23:00:00     0.0  1436.82  2360.67      0.0
```

Metric

A **Metric** is an **automatically aggregable computation** on results, often indexed by a ComponentSelector.

```
julia> # aggregation behavior is configurable
julia> get_component_agg_fn(calc_active_power)
sum (generic function with 15 methods)

julia> calc_avg_active_power = rebuild_metric(calc_active_power;
  name = "AverageActivePower", component_agg_fn = mean);

julia> calc_avg_active_power(
  make_selector(ThermalStandard; groupby = get_prime_mover_type), results1)
744×5 DataFrame
 Row │ DateTime             CT       CC       ST       OT
     │ DateTime             Float64  Float64  Float64  Float64
─────┼──────────────────────────────────────────────────────────
   1 │ 2020-07-01T00:00:00      0.0  83.6767  98.3611      0.0
   2 │ 2020-07-01T01:00:00      0.0  68.0     83.8165      0.0
   ⋮ │          ⋮              ⋮        ⋮        ⋮        ⋮
 744 │ 2020-07-31T23:00:00      0.0  143.682  98.3611      0.0
```

Metric

Users can easily define new **Metrics**.

```
julia> function fast_discharge(results, comp; start_time=nothing, len=nothing, thres=0.1)
    val = PowerAnalytics.read_component_result(
      results, ActivePowerOutVariable, comp; start_time = start_time, len = len)
    soc_limits = get_storage_level_limits(comp)
    soc_range = get_storage_capacity(comp) * (soc_limits.max - soc_limits.min)
    val[!, first(get_data_cols(val))] = (get_data_vec(val) .>= thres*soc_range)
    return val
  end;

julia> "Result is `true` if discharging quickly in the given time period else `false`"
julia> calc_fast_discharge = ComponentTimedMetric(name = "FastDischarge",
  eval_fn = fast_discharge, component_agg_fn = sum, time_agg_fn = mean);

julia> calc_fast_discharge(make_selector(EnergyReservoirStorage, "313_STORAGE_1"), results1)
 Row | DateTime            EnergyReservoirStorage__313_STORAGE_1
-----+-----------------------------------------------------------
   1 | 2020-07-01T00:00:00                                false
   ⋮ |          ⋮                          ⋮
 744 | 2020-07-31T23:00:00                                false
```

Output Manipulation

Resulting DataFrames have metadata enabling post-Metric **time aggregation** and plotting.

```
julia> full_power_df = calc_active_power(
  make_selector(Generator; groupby = typeof), results1); # 755x5 DataFrame

julia> aggregate_time(full_power_df)
1×5 DataFrame
 Row │ DateTime            ThermalStandard  HydroDispatch  RenewableDispatch  ⋯
─────┼──────────────────────────────────────────────────────────────────────────
   1 │ 2020-07-01T00:00:00       2.78707e6        21626.0          7.09695e5  ⋯

julia> aggregate_time(full_power_df; groupby_fn = dayofweek, groupby_col = "agg_day")
7×6 DataFrame
 Row │ DateTime            agg_day  ThermalStandard  HydroDispatch  RenewableDispatch  ⋯
─────┼──────────────────────────────────────────────────────────────────────────────────
   1 │ 2020-07-06T00:00:00        1        3.59081e5        2811.12           1.04591e5  ⋯
   2 │ 2020-07-07T00:00:00        2        3.43765e5        2745.08          106733.0
   3 │ 2020-07-01T00:00:00        3        4.20078e5        3490.63           1.32703e5
   4 │ 2020-07-02T00:00:00        4        4.44829e5        3493.3            1.1759e5
   5 │ 2020-07-03T00:00:00        5        4.63554e5        3425.49           1.14584e5  ⋯
   6 │ 2020-07-04T00:00:00        6        3.83284e5        2944.23           67607.7
   7 │ 2020-07-05T00:00:00        7        3.72475e5        2716.16           65886.6
```

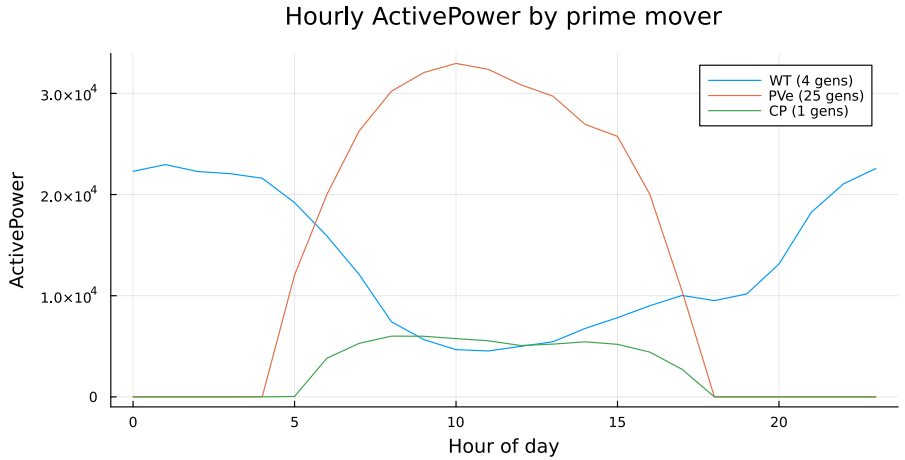
Output Manipulation

Resulting **DataFrames have metadata** enabling post-Metric time aggregation and **plotting**.

```
julia> df = aggregate_time(calc_active_power(
  make_selector(RenewableGen; groupby = get_prime_mover_type), results1);
  groupby_fn = hour, groupby_col = "hour"); # 24x5 DataFrame

julia> labels = permutedims(["$label ($(length(colmetadata(df)[label]["components"])) gens)"
  for label in Symbol.(get_data_cols(df))]);
julia> title = "Hourly $(metadata(df)["title"]) by prime mover";

julia> plot(
  df[!, "hour"], get_data_mat(df),
  labels = labels
  title = title,
  xlabel = "Hour of day",
  ylabel = metadata(df)["title"]
)
```



Miscellaneous

Other features not shown:

- Easy results loading
- Easy computation of multiple metrics

Check out the tutorial online!

We run the `post_processing` routine with our multi-scenario results we previously defined. We can see the final summary table including all scenarios below:

```
julia> df = post_processing(results_all);
Computing for scenario: Scenario_1
Computing for scenario: Scenario_2

julia> show(df; allcols = true)
2×10 DataFrame
```

Row	Scenario	DateTime	Thermal Generation (MWh)	Renewables Curtailment (MWh)
	String	DateTime	Float64	Float64
1	Scenario_1	2020-07-01T00:00:00	2.78707e6	31740.3
2	Scenario_2	2020-07-01T00:00:00	2.78266e6	27960.7

Looking at the final dataframe, we can now easily compare the aggregated results of the selected metrics between the two simulation scenarios.

Focusing on the thermal generation and renewable curtailment results for example, we observe a decrease of approximately 0.15% in total thermal generation and approximately 11.90% in renewable curtailment across the full simulation horizon and the entire system. The increased energy capacity of the storage device in the second scenario enables it to store more excess renewable energy rather than curtail it, which in turn reduces the need for thermal generation.

Design Principles

A user-centered interface with a powerful, performant API enabled by Julia language features

User-centered design

- Designed with real analysis scripts in mind
  - Reduced lines of code by 2/3 from 293 to 98
- Ruthlessly eliminates complexity
  - Why not SQL?
  - Why not custom Tables.jl type?
- In the Sienna “design language”

Performance and power

- Extensibility
  - ComponentSelector
  - Metrics
- Performs on Eastern Interconnection-sized datasets, ready for HPC
  - 1.4s for power, curtailment of thousands of generators
- Isomorphism with SQL on unpivoted data (next slide)
  - Adds arbitrary code execution

https://www.nrel.gov/hpc/kestrel-countdown

Isomorphism with SQL

Up to pivoting and given precomputation,

```
SELECT component_group, timestamp_group,
  AVG(component_value) AS agg_value
FROM (
  SELECT component_group, result_timestamp,
    timestamp_group,
    SUM(value) AS component_value
  FROM metric_table
  WHERE component_filter = 1
  GROUP BY component_group, result_timestamp
)
GROUP BY component_group, timestamp_group;
```

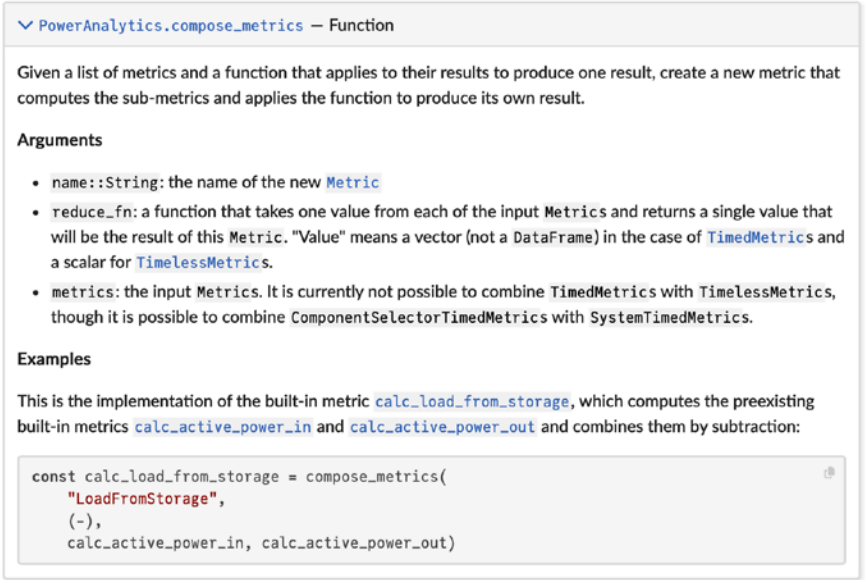
component	component_group	component_filter	result_timestamp	timestamp_group	value
C1	G1	1	T1	D1	3
C1	G1	1	T2	D1	1
C1	G1	1	T3	D2	4
C1	G1	1	T4	D2	1
C2	G1	1	T1	D1	5
C2	G1	1	T2	D1	9
C2	G1	1	T3	D2	2
C2	G1	1	T4	D2	6
C3	G2	1	T1	D1	5
C3	G2	1	T2	D1	3
C3	G2	1	T3	D2	5
C3	G2	1	T4	D2	8
C4	G2	0	T1	D1	9
C4	G2	0	T2	D1	7
C4	G2	0	T3	D2	9
C4	G2	0	T4	D2	3

becomes:

```
aggregate_time(calc_metric(
  make_selector(component_filter, Component; groupby = component_group), results);
  groupby_fn = timestamp_group, groupby_col = "timestamp_group")
```

Julia language features

- Functional paradigm
  - Functor pattern
  - Functional Metric definition and editing (compose_metrics)
- Multiple dispatch
  - ComponentSelector extensibility
  - Metric computation implementation
- Compilation for performance



`PowerAnalytics.compose_metrics` — Function

Given a list of metrics and a function that applies to their results to produce one result, create a new metric that computes the sub-metrics and applies the function to produce its own result.

Arguments

- `name::String`: the name of the new `Metric`
- `reduce_fn`: a function that takes one value from each of the input `Metrics` and returns a single value that will be the result of this `Metric`. "Value" means a vector (not a `DataFrame`) in the case of `TimedMetrics` and a scalar for `TimelessMetrics`.
- `metrics`: the input `Metrics`. It is currently not possible to combine `TimedMetrics` with `TimelessMetrics`, though it is possible to combine `ComponentSelectorTimedMetrics` with `SystemTimedMetrics`.

Examples

This is the implementation of the built-in metric `calc_load_from_storage`, which computes the preexisting built-in metrics `calc_active_power_in` and `calc_active_power_out` and combines them by subtraction:

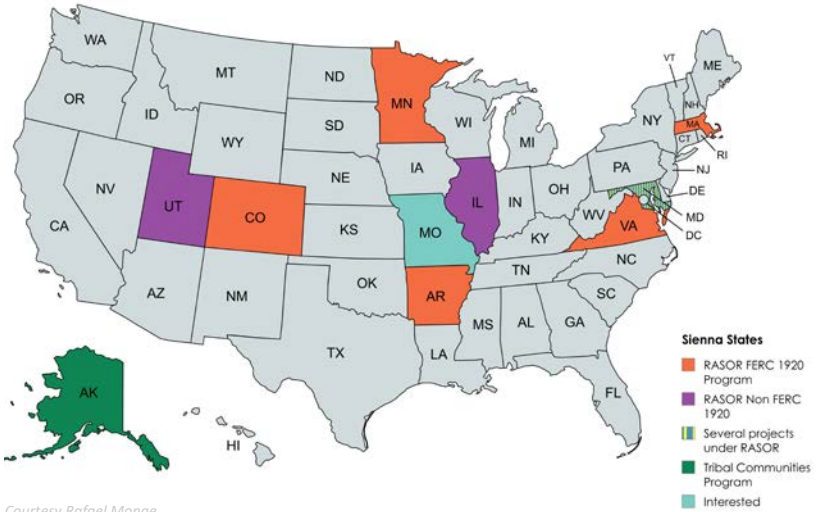
```
const calc_load_from_storage = compose_metrics(
    "LoadFromStorage",
    (-),
    calc_active_power_in, calc_active_power_out)
```

Usage and future work

Used at NREL for regional study analytics, extreme events research

Future work:

- Greater composability of ComponentSel
- More streamlined Metric creation
- Continued performance improvements
- PowerGraphics.jl analogous redesign



Courtesy Rafael Monge

Takeaways for package designers

- Meet your users where they are
- Develop a “theory” of intuitive abstractions
- Manage expectations

Acknowledgements

- Sourabh Dalvi and Clayton Barrows for project conceptualization and funding support
- Dan Thom, José Daniel Lara, and Joseph McKinsey for design and code review
- Akrivi Liakopoulou, Kate Doubleday, and Anna Casavant for documentation and tutorial support
- Tengis Dashmunkh for implementation support

Documentation (reference, tutorial, GitHub, etc.):

https://nrel-sienna.github.io/PowerAnalytics.jl/stable

Q&A

www.nrel.gov

NLR/PR-6A40-96062

This work was authored by NREL for the U.S. Department of Energy (DOE), operated under Contract No. DE-AC36-08GO28308. Funding provided by U.S. Department of Energy Office of Energy Efficiency and Renewable Energy Solar Energy Technologies Office. A portion of this research was performed using computational resources sponsored by the U.S. Department of Energy's Office of Energy Efficiency and Renewable Energy and located at the National Renewable Energy Laboratory. The views expressed in the article do not necessarily represent the views of the DOE or the U.S. Government. The U.S. Government retains and the publisher, by accepting the article for publication, acknowledges that the U.S. Government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this work, or allow others to do so, for U.S. Government purposes.