SoftwareX 15 (2021) 100747

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

PowerSystems.jl — A power system data management package for large scale modeling

José Daniel Lara [a,b], Clayton Barrows [b,*], Daniel Thom [b], Dheepak Krishnamurthy [b], Duncan Callaway [a]

[a] Energy and Resources Group, University of California, Berkeley, CA 94720, United States of America
[b] National Renewable Energy Laboratory, Golden, CO 80401, United States of America

ARTICLE INFO

Article history:
Received 30 September 2020
Received in revised form 28 May 2021
Accepted 8 June 2021

Keywords:
Power Systems
Julia
Energy

ABSTRACT

PowerSystems.jl is a package to organize and manipulate data for the study of energy systems with diverse modeling requirements. This software serves two main purposes: to reduce the burden of large power system data set development, and to promote reproducible research and simulation. PowerSystems.jl implements an abstract hierarchy to represent and customize power systems data and includes data containers for quasi-static and dynamic simulation applications. Key features include efficient management of large quantities of time series data, optimized serialization, and comprehensive validation capabilities. The paper includes detailed discussion and examples for reference.

Code metadata

Current code version	v1.6.0
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX_2020_107
Code Ocean compute capsule	none
Legal Code License	BSD-3-Clause
Code versioning system used	Git
Software code languages, tools, and services used	Julia, JSON, HDF5
Compilation requirements, operating environments & dependencies	Julia ~1.6 InfrastructureSystems.jl ~1.7
If available Link to developer documentation/manual	https://nrel-siip.github.io/PowerSystems.jl/stable
Support email for questions	NREL.SIIP@nrel.gov
Zenodo DOI references	DOI: 10.5281/zenodo.4818561

1. Motivation and significance

Adequate access to the data sets used in scientific models leads to more transparency and reproducibility [1], improves scientific discovery and allows third parties to verify model results [2]. Specifically, there is a common concern that computational experiments in energy systems need to define, document, and automate the processing and generating of data separate from the modeling [1,3]. The data workflows need to distinguish between the raw input data in a variety of formats and obtained from many different sources, and the analytical data for the model. PowerSystems.jl enables a consistent data model that can be populated from diverse sources of information. PowerSystems.jl is designed to address these challenges based on the principles laid out in [4–6]; namely: software should always return the same outputs when given the same inputs, functionalities should be easy to use, extend, prototype, and integrate into other packages, and code should be written for people to understand, not just for efficient computer resource use.

Applying scientific computing principles to the data processes used for energy sector models introduces several requirements:

- **Intuitive data creation scripts:** The syntax to create data sets should be easy to interpret and maintain.
- **Flexible interfaces for data intake:** Data sources for electric energy systems can be heterogeneous, requiring flexible interfaces to manipulate the data.

* Corresponding author.
E-mail address: clayton.barrows@nrel.gov (Clayton Barrows).

https://doi.org/10.1016/j.softx.2021.100747

- **Straightforward extension for new data layouts:** With the addition of new technologies and operational modes, the software needs to be extensible in order to add new data representations.
- **Dedicated public interface for extension and integration:** User extensions and integration as a dependency should not require modifications to the source code (also known as the delegation pattern [7]).[1]
- **Optimized memory use for large data sets:** The implementation should not overwhelm system memory when handling large data sets.

However, the traditional practice in energy modeling has been to develop *ad-hoc* data structures, containers, and interfaces related to the underlying mathematical model [8]. However, by virtue of relying on custom data structures and utilities, modeling packages can discourage data sharing and create barriers to validation, comparison and introduction of models. As a consequence model developers currently devote significant resources to parsing and converting between data models. In most cases, these efforts serve the specific scope of the analytical model and do not result in reusable code. Supplemental Information S-1 discusses in the data modeling characteristics of other software packages in more detail.

New initiatives are currently under development that focus on multi-sectorial energy system planning data management. PowerGenome [9], and Open Energy Platform [10] are efforts that focus on data curation, acquisition, and provenance for energy system planning models. For example, PowerGenome is used to generate inputs for the Gen-X model [11]. Spine-Toolbox [12] is an application with a graphical user interface for multi-sector energy system data and model workflow management. Efforts are underway to develop linkages to `PowerSystems.jl` from PowerGenome and Spine-Toolbox. These integrations will complement existing `PowerSystems.jl` capabilities and provide mechanisms to explore the operational impacts of long term planning (years to decades) results, and enable a graphical interface for editing data and managing model workflows.

To the authors' knowledge, `PowerSystems.jl` is the only tool designed to provide model-agnostic data structures and model development capabilities as an independent library. Although other authors have advocated for a canonical data model in XML format [13], `PowerSystems.jl` also provides generic tools and interfaces required for data processing, verification, and extend the library of models. The principal use case for `PowerSystems.jl` is to provide efficient intake and utilization of power systems model input data.

The interfaces in `PowerSystems.jl` are designed with three types of users, and specific uses of a data modeling package in mind.

- **Modeler:** Develop standard data sets, share data, or generate reproducible computational experiments.
- **Model Developer:** Develop custom components, data sets, and models. Use `PowerSystems.jl` as a dependency on a modeling package.
- **Code Developer:** Contribute source code to `PowerSystems.jl` to implement new features.

This paper focuses on the first two categories: Modelers and Model Developers. PowerSystems.jl provides details for users in all categories, contains links to examples application code and a library section with all the supported models. Readers are encouraged to look through the tutorial sections as a starting point and the developer section when looking into the requirements to integrate custom data structures.

[1] In computer science, a design pattern is a reoccurring solution that is sufficiently general to warrant its own title and description.

2. Software description

`PowerSystems.jl` is developed in Julia [14] due to the inherent *composability* of the language and the extensive interoperability capabilities with other programming languages. `PowerSystems.jl` exploits the type system and multiple dispatch of the Julia programming language to promote the open development of energy data sets across domains.

The main features of `PowerSystems.jl` include:

- Comprehensive and extensible library of data structures for modeling electrical systems.
- Large scale data set development tools based on common text based data formats (PSS/e `.raw` and `.dyr`, and MATPOWER `.m`) and configurable tabular data (e.g. CSV) parsing capabilities.
- An optimized container for component data and time series supporting serialization to portable file formats and configurable validation routines

In `PowerSystems.jl`, a device is defined using a Julia structure embedded in a type hierarchy. Each device is discussed in detail in Section 3.1. This implementation enables categorizing the devices by their abstract operational characteristics. In principle, the generalization of each component is done at the categorical level, preventing the shortcomings of prescriptive data models. Representing all potential devices in energy modeling is not possible; neither is it desirable, as new technologies become available and make parts of the library obsolete. Thus, it is necessary to provide an extensible data model with simple rules such that different users can store custom data in an organized ontology and still use core other functionalities of the package.

The implementation of the data structures is *"method forward"*, which implies that the information stored in each object is accessible through the implementation of methods (e.g., `get_parameter_value(::DataStruct)`) and not by accessing specific fields (e.g., `datastruct.parameter_value`). Similarly, the requirements for extensions are described as interface implementations, not data fields, providing modelers with more flexibility. This design prevents known fragilities with classic implementations of inheritance [7].

Implementing data structures through interfaces is especially valuable for long-term code maintenance and for reproducible experiments. The use of *accessor functions* enables the modeler to manipulate the parameters without concerns about the underlying implementation. On the other hand, if the model accesses the data by field, subsequent implementation changes can generate unsustainable maintenance costs. The accessor interface also reduces the cost of integrating `PowerSystems.jl` into modeling applications. The model developer can re-use the data management methods already implemented and thereby minimize the development of custom code to handle data input.

In data sets for energy systems simulation, the largest demand on memory often comes from time series data. Given the size of this data, it can overwhelm system memory and must remain on persistent media (e.g., disk). However, it is also critical to maintain low read/write latency. `PowerSystems.jl` data container's implementation solves these issues by leveraging HDF5 storage to execute fast data loads into memory on demand.

Time series data is implemented using different formats and types depending on the modeling needs and originating process. For instance, data storage is optimized for forecast data dependent on the structure. `PowerSystems.jl` supports forecast data formatted as overlapping time windows (e.g. 4-h ahead forecasts created every 15-min) and contiguous time series of observations. The flexible representation of time series data allows for multiple

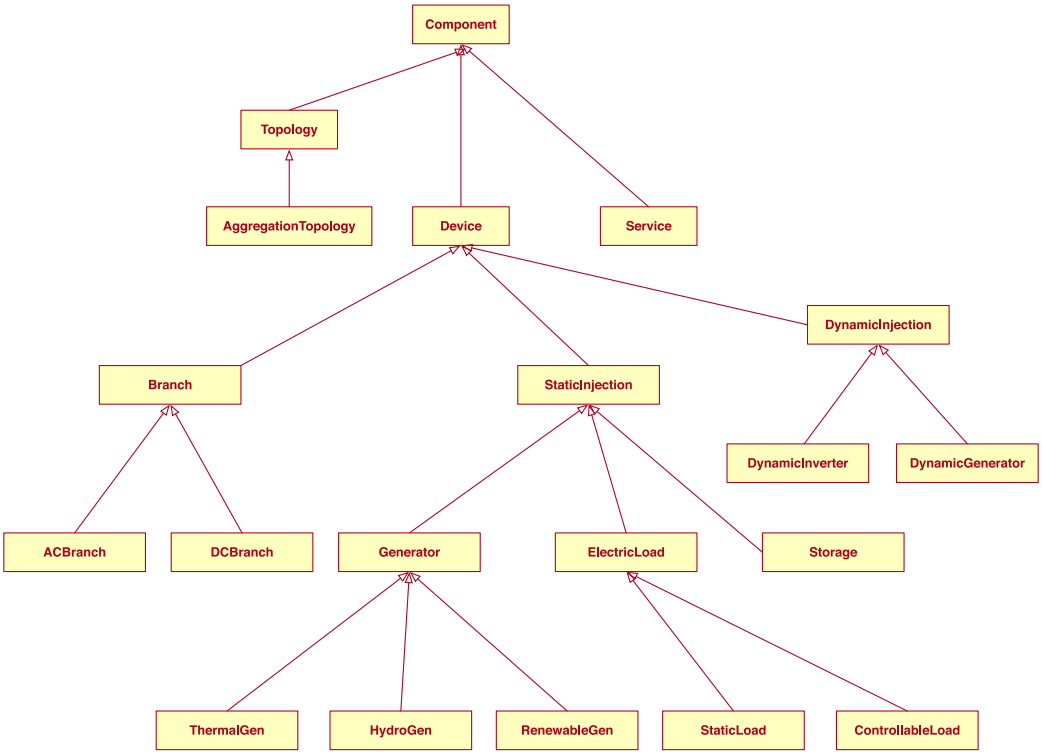

Fig. 1. Abstract tree hierarchy.

uses including Day-Ahead market models or slice selections to model limited operation periods. Also, `PowerSystems.jl` provides a mechanism to share time series data across components; this can greatly reduce primary storage requirements. Additional details about time series data management and implementation are included in Supplemental Information S-4.

3. Software architecture

`PowerSystems.jl` is structured to enable the requirements discussed in Section 2 through the implementation of the following features:

1. Abstract type hierarchy,
2. Optimized read/write data container (named `System`),
3. Utilities to facilitate modeling, extensions, and integration

The optimized container and generic extension interfaces in `PowerSystems.jl` are implemented through the utility library `InfrastructureSystems.jl` [15]. `PowerSystems.jl` contains the code, methods, and utilities specific to the electricity sector's physical representation and relationships. `InfrastructureSystems.jl` provides generic methods to handle components and manage data. This design follows from the recognition that several of the general features and requirements in `PowerSystems.jl` apply to any networked infrastructure data handling software, and in the future, an extension to water, gas, and similar infrastructure systems is possible.

3.1. Type hierarchy

The use of type trees (or taxononomies) to organize data classes or types is commonly used in libraries that implement an object oriented approach. For instance, the Common Information Model (CIM) [16] uses a taxonomy to represents all the major components in electric utility operations. However, the CIM is meant to facilitate the integration controls developed independently by different vendors and is not suitable for modeling. Also, initiatives like the Open Energy Ontology [17], which is currently under development, focus on organizing terms and relationships within energy system modeling.

The abstract hierarchy implemented `PowerSystems.jl` in enables categorization of the devices by their operational characteristics and modeling requirements. Fig. 1 shows the abstract hierarchy of components[2]. For instance, generation is classified by the distinctive data requirements for modeling in three categories: Thermal, Renewable, and Hydropower. As a result of this design, developers can define model logic entirely based on abstract types and create generic code to support modeling technologies that are not yet implemented in the package. `PowerSystems.jl` has a category of topological components (e.g., Bus, Arc), separate from the physical components [8]. The hierarchy also includes components absent in standard data models, such as services. The services category includes reserves,transfers and Automatic Generation Control (AGC) (see, Supplemental Information S-2). The power of `PowerSystems.jl` lies in providing the abstraction without an implicit mathematical representation of the component in question.

Other abstractions provide flexible specification of parameters with distinct representations. An example is the representation of costs (see, Supplemental Information S-2). `PowerSystems.jl` implements several `DeviceParameters` to support composition patterns.

A comprehensive example of using methods and composition is the implementation of the `ThermalStandard`, an implementation a of thermal generator using common data fields, shown in Fig. 2. `ThermalStandard` implements several methods to retrieve device parameters such as the active power limits. It

[2] Due to the size of the library, it is not possible to depict all potential concrete components. Supplemental information S-2 contains more examples of concrete components

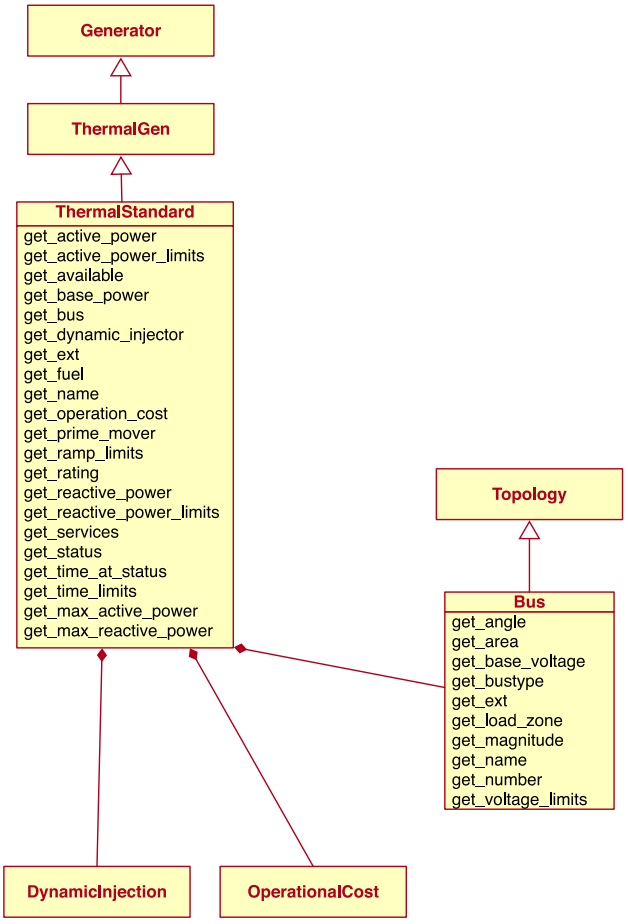

Fig. 2. Implementation of ThermalStandard.

is composed of the bus, operational cost, services, and the possibility to include a `DynamicInjection` to represent dynamic models for the same device. Supplemental Information S-5 contains details about the extensions of existing components and the addition of custom components.

Classically, data models have been separated dynamics and quasi-static analyses in different data sets. However, composition enables a joint representation of the components and eliminates the requirement to maintain two discrete databases. Detailed examples of building data sets for dynamic simulations is shown in the supplementary information S-3.

3.2. Data container

The `System` is the main container of components and the time series data references. `PowerSystems.jl` uses a hybrid approach to data storage, as shown in Fig. 3, where metadata (for describing package version compatibility, unique identifiers, and other system level meta-information), component data, and time series references are stored in volatile memory while the actual time series data is stored in an HDF5 file. This design loads into memory the portions of the data that are relevant at time of the query, and so avoids overwhelming the memory resources.

`PowerSystems.jl` implements a wide variety of methods to search for components to aid in the development of models. Listing 1 shows an example of retrieving components through

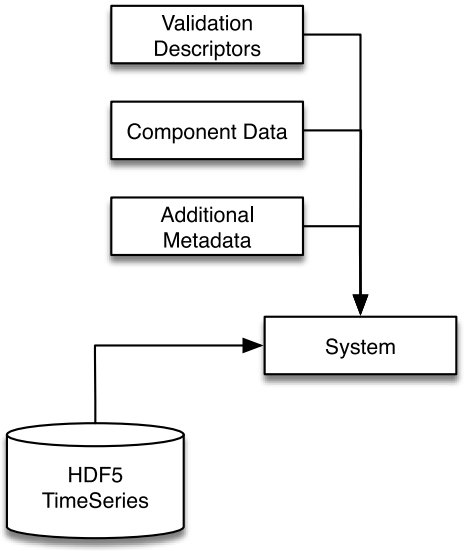

Fig. 3. `System` data container structure.

the type hierarchy with the `get_components` function and exploiting the type hierarchy for modeling purposes.[3] The default implementation of the function `get_components` takes the desired device type (concrete or abstract) and the system and it also accepts filter functions for a more refined search. The most common filtering requirement is by component name and for this case the method `get_component` returns a single component taking the device type, system and name as arguments. The container is optimized for iteration over abstract or concrete component types as described by the type hierarchy. Given the potential size of the return, `PowerSystems.jl` returns Julia iterators in order to avoid unnecessary memory allocations.

An essential workflow in energy systems modeling is developing data sets that combine existing data sources with new components and time series data. Listing 2 shows the code process to execute the following workflow:

1. Load component data from a power flow file,
2. Add a new component. The example creates and adds a single wind power plant. The component constructors use keyword arguments making data entries very explicit,
3. Load time series data from a pointers file and add it to the system components,
4. Load time series data specific to newly added component and attach it to a single component,
5. Serialize system for future use.

The data is serialized to a JSON file and the time series data to an HDF5 file. This feature has been developed to enable reproducible data workflows. Modelers can develop data sets as a separate exercise from the modeling and later use `PowerSystems.jl` to load the final data in a fraction of the time that takes to re-create and load using the raw data. This feature has proven to be particularly useful when the raw data has been generated from free form tables stored as CSV files [18,19].

Examples of data cases already available in `PowerSystems.jl` format include the data from the RTS-96 data set https://github.com/GridMod/RTS-GMLC [19] and we have made available some

[3] See Julia's punctuation to facilitate reading the listing https://docs.julialang.org/en/v1/base/punctuation/.

```julia
using PowerSystems

# Load the system from a power flow case
system_data = System("case_ACTIVSg70k.m")

function installed_capacity(system::System; technology::Type{T} =
↪  Generator) where T <: Generator
    installed_capacity = 0.0
    for g in get_components(T, system)
        installed_capacity += get_max_active_power(g)
    end
    return installed_capacity
end

installed_capacity(system_data)
installed_capacity(system_data; technology = RenewableGen)
installed_capacity(system_data; technology = ThermalStandard)
```

Listing 1: This example implements a function where the modeler can choose the technology by its type and use the different implementations of `get_max_active_power`. Note that in line 15 the function takes an abstract type and in line 16 a concrete type. This listing exemplifies the flexibility of the container interface to facilitate the development of models consistent with the ontology defined in Fig. 1.

simpler example data sets in a separate repository https://github.com/NREL-SIIP/PowerSystemCaseBuilder.jl that allows users to automatically build systems from a standard specification.

3.3. Additional utilities

Besides the core features discussed in previous sections, `PowerSystems.jl` provides additional utilities to aid in model development, data set analysis and integration. All these additional utilities exploit the type hierarchy and the data containers to reduce development overhead. The following additional features are detailed in the package documentation.

1. **Network Matrices**: Utilities to calculate the bus admittance (Ybus), power transfer and line outage distribution factor (PTDF, LODF) matrices. All the matrices implement custom array containers to facilitate the indexing of component names.
2. **Power flow solution**: This utility is used to serialize initialized systems or to check if the operating point of the components is valid in AC.
3. **Extended container printing methods**: Several print-to-REPL[4] methods are included to improve the visualization of the data.

4. Illustrative example

`PowerSystems.jl` is accompanied by an extensive repository SIIPExamples.jl[5] with common usage examples, several of which are included in the Supplemental Information.

Listing 3 shows a minimal example of `PowerSystems.jl` used to develop an Economic Dispatch (ED) model. The listing shows the stages explicitly:

1. Make the data set from power flow and time series data,
2. Serialize the data,
3. Pass the data and algorithm to the model.

One of the main uses of `PowerSystems.jl` is not having re-run lines 4–5 for every model execution. The model code shows an example of populating the constraints and cost functions using accessor functions inside the model function (lines 9–37). The example concludes by reading the data created earlier in line 40 and passing the algorithm with the data in line 41.

5. Impact

`PowerSystems.jl` is designed to account for the common workflows that analysts and engineers use to manipulate data sets, and develop new models and packages for the changing landscape of energy systems. The two main contributions are as follows:

1. Decouples data processing from the computations in the models.
2. Provides an inherently extensible data modeling framework to develop new models and software packages that can be shared across a variety of modeling objectives.

Existing power systems software packages have addressed a subset of the principles established in Section 1 but with the specific focus of providing a data model for the analytic objective of the underlying mathematical model. As a result, developers of new models are faced with the choice of either adapting their data model needs to the existing structures or develop custom ones. Often the underlying data is formatted in `MATPOWER` [20] developed in Matlab, and industrial tools like PSS/e. Both formats are based on fixed order, fixed position text files that require code development for data input. `PowerSystems.jl` goes beyond the implementation of the canonical data model and also provides manipulation methods and utilities for its integration into other packages.

[4] A REPL or read–evaluate–print loop is a simple programming environment that takes and executes inputs then returns them to the user.

[5] https://github.com/NREL-SIIP/SIIPExamples.jl.

```julia
using PowerSystems, CSV, TimeSeries, Dates

# (1) Load the system from a power flow case
system = System("src/case5.m")

# Define a new device
new_renewable = RenewableDispatch(
        name = "WindBusA",
        available = true,
        bus = get_component(Bus, system, "3"),
        active_power = 2.0,
        reactive_power = 1.0,
        rating = 1.2,
        prime_mover = PrimeMovers.WT,
        reactive_power_limits = (min = 0.0, max = 0.0),
        base_power = 100.0,
        operation_cost = TwoPartCost(22.0, 0.0),
        power_factor = 1.0
    )

# (2) Add that device as a new component
add_component!(system, new_renewable)

# Manually create a TimeSeriesData
time_series_data_raw = TimeArray(CSV.read("wind_data.csv"),
↪  timestamp=:timestamp)
ts_data = SingleTimeSeries(label = "active_power", data =
↪  time_series_data_raw)

# (3) Add the time series to the system and component
add_time_series!(system, new_renewable, ts_data)

# (3) Load time series from pointers file
add_time_series!(system, "timeseries_pointers_load.json")

to_json(system, "serialized_system.json")
```

Listing 2: Example of data set composition workflow

`PowerSystems.jl` data-handling improves the scientific integrity of power systems research and analysis and enables the implementation of scientific computing principles for several research communities that rely on electric power systems data. One recent example of its application in the analysis of the Cambodian grid [21], where the authors separated data manipulation from modeling, making transparent both processes individually.

High impact research in energy systems requires analysis of large amounts of data, `PowerSystems.jl` has several key features for modelers and analysts to handle and extend large data sets. `PowerSystems.jl` **enables easy data creation** with parsers for standard file formats, **optimizes in-memory data access** by creating methods for efficient parameter and time series access and iteration. The fast data access and efficient model instantiation features of `PowerSystems.jl` were leveraged to enumerate large contingency sets in numerous power system test cases [22,23].

For model developers, `PowerSystems.jl` provides a generic, reusable, and customizable data model applicable to multiple modeling objectives. Not only does it provide the computational improvement to handle data at large scales, but it also provides extension capabilities by design that make it easier to integrate into modeling packages.

`PowerSystems.jl` implementation in the Julia programming language **allows for fast development and prototyping, as well as fast compilation and runtime performance** [14]. In contrast to other object-oriented-programming, where inheritance is the only way to develop extensions or custom methods, it provides user flexibility and extensibility by providing type abstractions on which to implement methods using multiple dispatch.

The type-based and method-forward paradigm in `PowerSystems.jl` incentivizes the adoption of best practices in scientific computing [4] by creating accessible interfaces to enable code reuse in modeling and to create modular and reproducible scientific computing applications. Listing 3 summarizes the breakdown between data, model, and algorithm. `PowerSystems.jl` is used in [3] to implement a more comprehensive pipeline for scientific computing applied to operational simulations in power systems.

Recent publications [24] used `PowerSystems.jl` for the development of AGC simulation models following scientific computing practices to develop the experiment. Two recent contributions in low inertia power systems [25,26] exploited the flexibility of the dynamic models specification in `PowerSystems.jl` to implement the data manipulation code in the simulation experiments.

6. Conclusions

This paper introduces the release of `PowerSystems.jl`, the first tool dedicated exclusively to providing data management tools for electricity system modeling across research domains.

```julia
using PowerSystems, JuMP, Ipopt

########## Data Process ###########
system = System("src/5bus_ts/case5_re.m")
add_forecasts!(system, "timeseries_pointers.json")
to_json(system, "system_data.json")

######## Model Process #########
function ed_model(system::System, optimizer)
  m = Model(optimizer)
  time_periods = get_time_series_horizon(system)
  thermal_gens_names = get_name.(get_components(ThermalStandard, system))
  @variable(m, pg[g in thermal_gens_names, t in time_periods] >= 0)

  for g in get_components(ThermalStandard, system), t in time_periods
      name = get_name(g)
      @constraint(m, pg[name, t] >= get_active_power_limits(g).min)
      @constraint(m, pg[name, t] <= get_active_power_limits(g).max)
  end

  net_load = zeros(time_periods)
  for g in get_components(RenewableGen, system)
      net_load -= get_time_series_values(SingleTimeSeries, g, "max_active_power")
  end

  for g in get_components(StaticLoad, system)
      net_load += get_time_series_values(SingleTimeSeries, g, "max_active_power")
  end

  for t in time_periods
      @constraint(m, sum(pg[g, t] for g in thermal_gens_names) == net_load[t])
  end

  @objective(m, Min, sum(pg[get_name(g), t]^2 *
  get_cost(get_variable(get_operation_cost(g)))[1] +
  pg[get_name(g), t] *
  get_cost(get_variable(get_operation_cost(g)))[2] for g in
  get_components(ThermalGen, system), t in time_periods))

  return optimize!(m)
end

#### Execution ####
system_data = System("system_data.json")
results = ed_model(system_data, Ipopt.Optimizer)
```

Listing 3: Example usage of `PowerSystems.jl` to the development of a multi-time step Economic Dispatch (ED) formulation

The focus of `PowerSystems.jl` is to provide a structured data-scheme, efficient in-memory data handling, and parsing capabilities from popular file formats. The primary motivation is to provide model agnostic data structures that incentivize separation between the data processing code and the modeling code.

Ongoing work focuses on expanding input data formats and expand potential data sources to include available databases like OpenGenome, Spine and OpenEnergyPlatform. Improving interoperability with open data sources will facilitate the adoption by other fields interested in using electric power systems data models.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was authored by the National Renewable Energy Laboratory (NREL), operated by Alliance for Sustainable Energy, LLC, for the U.S. Department of Energy (DOE) under Contract No. DE-AC36-08GO28308. This work was supported by the Laboratory Directed Research and Development (LDRD), United States

of America Program at NREL. 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.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.softx.2021.100747.

References

[1] Pfenninger S, Hirth L, Schlecht I, Schmid E, Wiese F, Brown T, et al. Opening the black box of energy modelling: Strategies and lessons learned. Energy Strategy Rev 2018;19:63–71.
[2] DeCarolis JF, Hunter K, Sreepathi S. The case for repeatable analysis with energy economy optimization models. Energy Econ 2012;34(6):1845–53.
[3] Lara JD, Lee JT, Callaway DS, Hodge B-M. Computational experiment design for operations model simulation. Electr Power Syst Res 2020;189:106680. http://dx.doi.org/10.1016/j.epsr.2020.106680, http://www.sciencedirect.com/science/article/pii/S0378779620304831.
[4] Wilson G, Aruliah DA, Brown CT, Hong NPC, Davis M, Guy RT, et al. Best practices for scientific computing. PLoS Biol 2014;12(1):e1001745.
[5] Wilson G, Bryan J, Cranston K, Kitzes J, Nederbragt L, Teal TK. Good enough practices in scientific computing. PLoS Comput Biol 2017;13(6):e1005510.
[6] Sandve GK, Nekrutenko A, Taylor J, Hovig E. Ten simple rules for reproducible computational research. PLoS Comput Biol 2013;9(10):e1003285.
[7] Kwong T. Hands-on design patterns and best practices with julia. Packt Publishing; 2020.
[8] Milano F. Power system modelling and scripting. Springer Science & Business Media; 2010.
[9] Schivley G, Welty E, Patankar N. PowerGenome/PowerGenome: v0.4.1. Zenodo; 2021, http://dx.doi.org/10.5281/zenodo.4552835.
[10] OpenEnergyPlatform/oeplatform. Open Energy Family; 2021, original-date: 2015-11-20T14:21:02Z, https://github.com/OpenEnergyPlatform/oeplatform.
[11] Jenkins JD, Sepulveda NA. Enhanced decision support for a changing electricity landscape: The genx configurable electricity resource capacity expansion model. p. 67.
[12] Marin M, PekkaSavolainen, Vennström P, Rinne E, jkiviluo, sundelld, et al. Spine-project/Spine-Toolbox: v0.5.0-final.1. Zenodo; 2021, http://dx.doi.org/10.5281/zenodo.4501197.
[13] Milano F, Zhou M, Hou G. Open model for exchanging power system data. In: 2009 IEEE power & energy society general meeting. IEEE; 2009, p. 1–7.
[14] Bezanson J, Edelman A, Karpinski S, Shah VB. Julia: A fresh approach to numerical computing. SIAM Rev 2017;59(1):65–98.
[15] Thom D, Lara JD, Barrows CP. NREL-SIIP/InfrastructureSystems.jl: v1.7.4. Zenodo; 2021, http://dx.doi.org/10.5281/zenodo.4780333.
[16] Uslar M, Specht M, Rohjans S, Trefke J, González JM. The common information model CIM: IEC 61968/61970 and 62325-A practical introduction to the CIM. Springer Science & Business Media; 2012.
[17] Glauer M, Booshehri M, Emele L, Fluegel S, Förster H, Frey J, et al. The open energy ontology. 2020.
[18] Xu Y, Myhrvold N, Sivam D, Mueller K, Olsen DJ, Xia B, et al. US test system with high spatial and temporal resolution for renewable integration studies. 2020, ArXiv Preprint arXiv:2002.06155.
[19] Barrows C, Bloom A, Ehlen A, Ikäheimo J, Jorgenson J, Krishnamurthy D, et al. The IEEE reliability test system: A proposed 2019 update. IEEE Trans Power Syst 2019;35(1):119–27.
[20] Zimmerman RD, Murillo-Sánchez CE, Thomas RJ, et al. MATPOWER: Steady-state operations, planning, and analysis tools for power systems research and education. IEEE Trans Power Syst 2011;26(1):12–9.
[21] Barrows CP, Chernyakhovskiy I. PSI-Cambodia: Analysis Release:1.0.0. Zenodo; 2020, http://dx.doi.org/10.5281/zenodo.4009566.
[22] Bush B, Chen Y, Ofori-Boateng D, Gei Y. Topological machine learning methods for power system responses to contingencies. In: The thirty-third annual conference on innovative applications of artificial intelligence, association for the advancement of artificial intelligence; 2020.
[23] Bush B. Topology-based machine-learning for modeling power-system responses to contingencies. In: American statistical association 2020 joint statistical meetings; 2020.
[24] Lara JD, Henriquez-Auba R, Callaway DS, Hodge B-M. AGC Simulation model for large renewable energy penetration studies. In: 2020 52nd North American Power Symposium (NAPS). 2021, p. 1–6. http://dx.doi.org/10.1109/NAPS50074.2021.9449687.
[25] Roberts C, Lara JD, Henriquez-Auba R, Poolla BK, Callaway DS. Grid-coupled dynamic response of battery-driven voltage source converters. In: 2020 IEEE international conference on communications, control, and computing technologies for smart grids (SmartGridComm). IEEE; 2020, p. 1–6.
[26] Henriquez-Auba R, Lara JD, Roberts C, Callaway DS. Grid forming inverter small signal stability: Examining role of line and voltage dynamics. In: IECON 2020 the 46th annual conference of the IEEE industrial electronics society. 2020, p. 4063–8. http://dx.doi.org/10.1109/IECON43393.2020.9255030.