Carousel Model Simulation Framework
PVLIB Users Group Meeting
ERPI / Sandia PV System Symposium

Mark Mikofski, SunPower Corp | May 10th, 2016

Carousel Model Simulation Framework

Agenda

- Model Simulation Requirements (<5 minutes)
  - Why did SunPower invest in a model simulation framework?
- Carousel Introduction (5 minutes)
  - A model simulation framework
- PVLIB demonstration (15 minutes)
  - An example of implementing a model with Carousel
- Roadmap (0-5 minutes)
  - What’s next?

The goal of this talk is to inspire collaboration and the development of new applications with the Carousel Model Simulation Framework.

Model Simulation Requirements

SUNPOWER®

Model Simulation Requirements

- SunPower needs:
  - Stable, performant, accurate, tested, documented performance models
  - A turnkey approach to develop, maintain, and update complex models.
  - The ability to quickly train new modelers and downstream collaborators.
  - Deploy and scale models in production environment and integrate with established tools.
- Approaches:
  - Establishing a set of common guidelines for coding paradigms and styles.
  - Leveraging existing / proven open source tools.
  - Utilizing reusable code for boilerplate procedures (I/O, solvers, data visualization) as much as possible.
  - Use a framework that lets modelers focus on implementing algorithms instead of overall simulation structure.

SunPower needed an approach that considered the overall modeling structure and allowed frequent change but was simple to use.

Carousel Introduction

SUNPOWER

Carousel Introduction

- Carousel is a model simulation framework
  - Design Patterns: Elements of Reusable Object-Oriented Software (1994) by Erich Gamma *et al.*
    - "The framework dictates the architecture of your application. It will define the overall structure, its partitioning into classes and objects, the key responsibilities thereof, how the classes and objects collaborate, and the thread of control." pp. 26-27
- Existing Frameworks
  - Many fields already implement component frameworks. Web frameworks such as Ruby on Rails and Django have become the established method to implement database driven websites because they automate many boilerplate design patterns and result in stable, highly performant and feature rich web sites.
  - Model simulation frameworks do exist – most notably MathWorks Simulink and Modelica, but they are proprietary, focused on development and analysis, and Modelica appears to be unsupported.
- Carousel is open sourced, uses the Python Scientific Stack, integrates well with existing production environments (web based applications, cluster computing, all OS platforms)

Carousel is a model simulation framework that lets you focus on modeling while it takes care of complex but boilerplate tasks.

A framework predefines these design parameters so that you, the application designer/implementer, can concentrate on the specifics of your application. – *Design Patterns*

PVLIB Demonstration

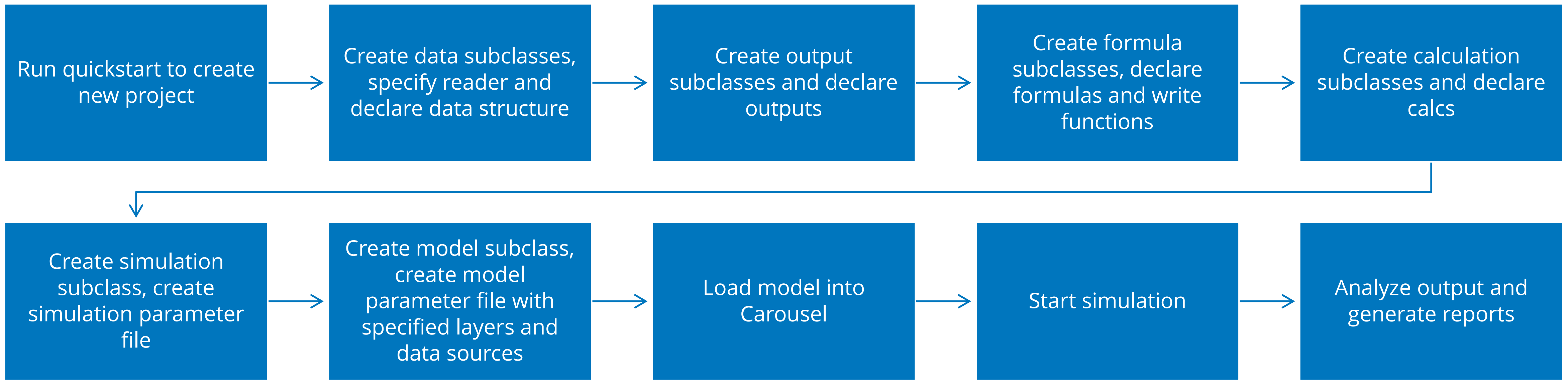
SUNPOWER®

Carousel Introduction

- Carousel *defines* model components and *controls* the simulation
  - **Data Inputs and Output**: collection from various sources, report generation, units and uncertainty propagation.
    - **DataReaders** are methods for retrieving data, EG: from csv, JSON, Excel, HDF5, REST API or database
  - **Formulas**: Mathematical equations and model algorithms are expressed as functions in a repeatable format. Units and uncertainty are automatically propagated.
  - **Calculations**: combine formulas with data and outputs and return new outputs.
  - **Models and Simulations**: Assembling everything together and running static & dynamic simulations.
- Users make models by subclassing the layers in their projects.
- Carousel is *extensible*:
  - Model components are generic base class called **layers** that can be used to implement new structural features.
  - Proposed additional layers include scrubbing, validation, optimization, visualization, cluster, debug and testing.

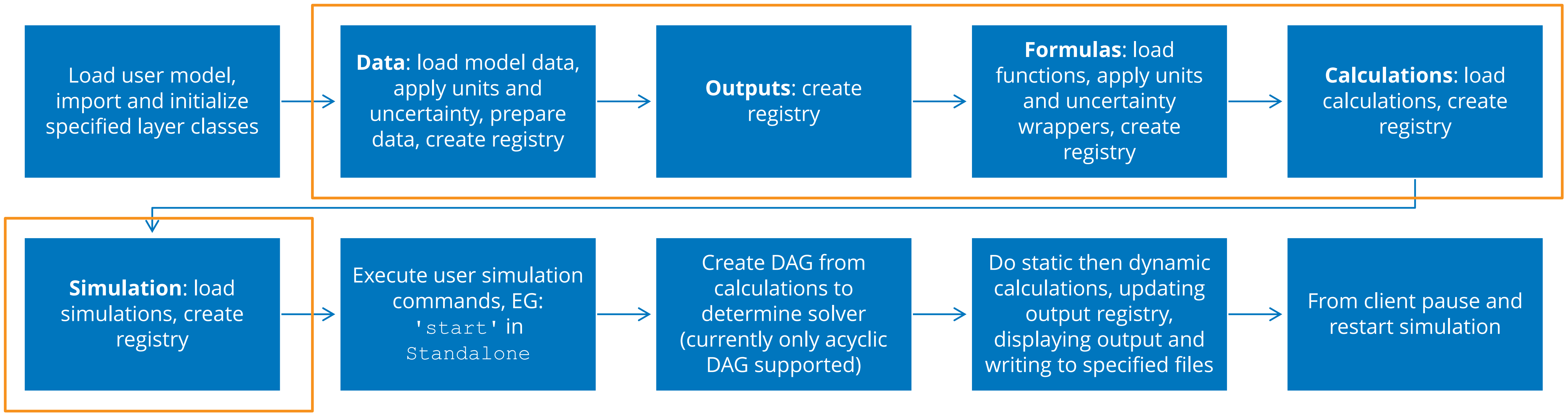
Carousel consists of five layers, new layers can be created to extend Carousel.

Carousel model simulation process


Run quickstart to create new project
Create data subclasses, specify reader and declare data structure
Create output subclasses and declare outputs
Create formula subclasses, declare formulas and write functions
Create calculation subclasses and declare calcs
Create simulation subclass, create simulation parameter file
Create model subclass, create model parameter file with specified layers and data sources
Load model into Carousel
Start simulation
Analyze output and generate reports

Model creation is a repeatable process with well defined component steps

Carousel process flow chart with `BasicModel` layers



Load user model, import and initialize specified layer classes

Data: load model data, apply units and uncertainty, prepare data, create registry

Outputs: create registry

Formulas: load functions, apply units and uncertainty wrappers, create registry

Calculations: load calculations, create registry

Simulation: load simulations, create registry

Execute user simulation commands, EG: `'start'` in `Standalone`

Create DAG from calculations to determine solver (currently only acyclic DAG supported)

Do static then dynamic calculations, updating output registry, displaying output and writing to specified files

From client pause and restart simulation

Carousel controls the flow of simulations, so you can focus on functions

SUNPOWER

PVLIB Demonstration

- This demonstration follows the Package Overview section of the PVLIB documentation. The annual energy is calculated using Sandia Array Performance Model.
- Requirements:
  - NumPy, SciPy, numexpr, Pint, XLRD, PVLIB, UncertaintiesWrapper, Python-Dateutil, PyTZ, Pandas, Nose, Sphinx
- Installation: https://pypi.python.org/pypi/Carousel
  - Carousel is at the Python Package Index. Install using pip from a shell or Windows command line.

```
$ pip install carousel
```

- Issues and Hacking: https://github.com/SunPower/Carousel
  - Issue tracker and source code are maintained on GitHub. Fork it and send a pull request.
- Documentation: http://sunpower.github.io/Carousel/
  - This demonstration is in the documentation tutorial section and the examples section of the source code on GitHub.

Carousel Quickstart

- The first step in creating a Carousel project is to run carousel-quickstart from the command line. It creates a folder structure for a basic model and a python package to hold your subclasses.

```
$ carousel-quickstart MyProject
```

- This creates a folder called MyProject with 6 subfolders: data, outputs, formulas, calculations, simulations and models
- It also creates a Python package called the same name as your project in lower case, eg: myproject

Use quickstart script to generate the folder structure for a new project.

Data Layer

- Specify the data structure by subclassing `DataSource` from Carousels `core` package.
- Data fields can be declared directly in the subclass as dictionaries of data attributes or in a separate JSON parameter file.
- Each `DataSource` subclass requires a `DataReader`, such as `XLRDReader`, that knows how to get data from files, databases and REST APIs. The default is `JSONReader` which is also used to read cached data.
- Methods: use `__prepare_data__()` to manipulate data further after loaded from source by reader
- Attributes:
  - `units`: Carousel uses Python Pint units wrapper.
  - `isconstant`: in dynamic calculations, determines whether data is indexed at each timestep or used in its entirety.
  - `uncertainty`: Carousel uses UncertaintyWrapper to propagate variance and sensitivity.
  - `timeseries`: another data or output field that indexes this data field.
- See `examples/PVPower/data/pvpower.json`, `pvpower/sandia_performance_model.py` & `carousel/tests/test_data.py`

Data definition is routine and consistent. Follows similar style as Django.

Outputs Layer

- Desired outputs must be specified in advance. Create a subclass of `Outputs` and add the desired output fields as either dictionaries or in a JSON parameter file.
- Attributes:
  - `units`: desired units of output
  - `isconstant`: indicates index for dynamic calculations, otherwise is constant
  - `size`: for dynamic calculations indicates the width of output per timestep
    - *eg*: a spectral output might be 122 wide corresponding to different wavelength bands at each timestep
  - `timeseries`: the data or output that indexes this output
  - `isproperty`: in dynamic calculations determines if output goes to zero when not calculated
  - `initial_value`: in dynamic calculations specifies the initial value
- See `examples/PVPower/outputs/irradiance.json`, `pvpower/sandia_performance_model.py` & `carousel/tests/test_outputs.py`

Outputs and data declaration provide clear information about structure.

Formula Layer

- Formula can be written as functions in python modules or as numerical expressions strings using Python numexpr.
- Formula can be specific or generic to be used by any model, eg: an integration function.
- Formula can be short or long – whatever is meaningful to you. Here are some information to guide you:
  - Return values of all formulas is stored in the outputs registry
  - Units, if given, are checked and converted going into functions, but inside only magnitudes are used and then given units are applied to returns.
  - Uncertainty, if given, is propagated across functions using 1st order Taylor series expansion and central difference approximation of sensitivity:

$$df = Jac \cdot Cov \cdot Jac^T = \sum \left(\frac{\partial f_i}{\partial x_j} \frac{\partial f_i}{\partial x_k} \right) (\Delta x_j \Delta x_k)$$

- Create a subclass of `Formulas` in your project package to declare your formulas.
- Formulas also have attributes: `args`, `units`, `isconstant` and `islinear`
- See `examples/PVPower/formulas/irradiance.json`, `examples/PVPower/formulas/irradiance.py`, `pvpower/sandia_performance_model.py` & `carousel/tests/test_formulas.py`

Separation of formulas from their usage clarifies their function and allows them to be reused in different context. Units/uncertainty are automatic.

Calculations

- Create in your project package to declare calculations.
- Calculations can be declared in a JSON parameter file or directly in your `Calc` subclasses consisting of the following:
  - An ordered list of functions to execute.
  - Argument names for each function mapped to corresponding data and output names.
  - Name of outputs to use for returns.
- Calculations attributes are: `dependencies` and `always_calc`
- See examples

Calculations combine formulas and arguments from data and/or outputs to return new outputs.

Simulations

- Subclass `Standalone` simulation in your project package.
- The `Simulation` class control the flow of the simulation. Should call `calc_static` and `calc_dynamic` methods, implement commands, write and display methods and accept progress hooks and registries.
- Sim attributes: `commands` – a list of commands that simulation accepts from model.
- Is given access to all registries by model.
- Specify parameters in JSON file.
- See examples

The `Simulation` class controls the flow of the simulation.

Model

- A model collects all layers: `data`, `outputs`, `formulas`, `calculations` and `simulations`
- Subclass the `BasicModel` in your project package
- Specify parameters in JSON file
- Can call simulations commands
- See examples

The model class is the top level. Its methods are limited to loading parameters, initializing layers and passing commands and registries to simulations.

Load the model and start the simulation

- In your project folder start python interpreter and import the model subclass from your project package.

```
~ $ cd MyProject
~/MyProject /$ python
>>> from pvpower.sandia_performance_model import SAPM
```

- Load your model parameters and instantiate the model.

```
>>> m = SAPM('models/sandia_performance_model-Tuscon.json‘)
```

- Start the model simulation(s). If you provide a simulation, then only that simulation will run, if you provide a list then those sims execute, if you leave simulation empty, then all start.

```
>>> m.command('start')
```

- In the future you can use either the carousel command line or the graphical tk client start a server and spawn model simulations. Currently simulations run in the main Python process.

Your model contains all the information need to run your simulations.

Roadmap

Roadmap

- Release plan and feature wish list.
  - Validation layer: check model integrity and validate data
  - Data scrubber and PECOS integration layer
  - Carousel client
  - Field class instead of dictionaries for layer declarations – eg: data fields like FloatField, etc.
    - Integration with mature serializer package such as Marshmallow to provide fields will also provide data validation
  - Name clean up and base class reorg: move common elements from layer subclasses to base class, eg: registry
  - Database and shared memory layer for stateless sharing of data between multiprocessing subprocesses.
  - Replace threading with multiprocessing with communication over TCP sockets and Carousel server (related to client)
  - Additional data readers such as REST API, HDF5 and DB and integration with Pandas and SQL alchemy.

Carousel is a work in progress. We hope others will collaborate to develop it.

PVFree

PVFree

A public database of CEC module and inverter and SAPM module parameters and REST API.

- REST API currently yields CEC module data from SAM csv file from 2014
- Directions and browsable datatables of PVFree available from demo site.
  - http://pvfree.alwaysdata.net/
- Collaboration to implement CEC and Sandia Perf. Model module parameters desired.
- Open PR to utilize PVFree in PVLIB instead of statically linked static SAM library csv files.
  - Collaborators desired to implement conversion of API JSON response to PVLIB inverter and module structure.
- SunPower will host this database and web server with community oversite by PVLIB users.
- SunPower will reserve the right to terminate database and server at will, but all effort possible will be made to facilitate the xfer of PVFree to new ownership so long as the PVLIB community maintains oversite.

PVFree is a public REST API and database maintained by PVLIB community for storage, retrieval, update and maintenance of module and inverter data.

SUNPOWER®

Thank You

Let’s change the way our world is powered.

Challenges of Current State

- Our existing models were complex
  - Implementing new features, making updates and bug fixes were difficult and error prone.
  - Profiling and debugging was very hard.
  - Training new modelers and downstream collaborators was challenging.
  - No one knew the code well.
  - Several coding paradigms and styles were mixed with no common guidelines.
  - 100% of the code was proprietary; 0% of the code was sourced from open source projects.
  - Modelers were focused on implementing algorithms and not on overall architecture or structure.
  - Boilerplate procedures were implemented multiple times instead of leveraging reusable code.

- There was no framework.

Model Simulation Requirements

- Provide a turnkey approach to develop, maintain, and update complex models efficiently.
- Provide a set of common guidelines for coding paradigms and styles.
- Enable detailed profiling and debugging.
- Facilitate training of new modelers and downstream collaborators.
- Easier for more modelers to become experts on the model implementation.
- Leverage widely known proven open source tools as much as possible.
- Allow modelers to focus on implementing algorithms instead of overall structure.
- Utilize reusable code for boilerplate procedures (I/O, solvers, data viz) as much as possible.

We need a framework for model simulation so we can focus on making models.

What is a software framework?

- Wikipedia
  - "A software framework is a universal, reusable software environment that provides particular functionality as part of a larger software platform to facilitate development of software applications, products and solutions."
  - Frameworks contain key distinguishing features that separate them from normal libraries:
    - *inversion of control*: In a framework, unlike in libraries or normal user applications, the overall program's flow of control is not dictated by the caller, but by the framework.[1]
    - *extensibility*: A framework can be extended by the user usually by selective overriding or specialized by user code to provide specific functionality.
  - Rationale: The designers of software frameworks aim to facilitate software development by allowing designers and programmers to devote their time to meeting software requirements rather than dealing with the more standard low-level details of providing a working system, thereby reducing overall development time.[2]