SUNPOWER

Carousel Model Simulation Framework
PyData San Francisco 2016
Galvanize

Mark Mikofski, SunPower Corp. | August 12th, 2016

About me

- @bwanamarko, @breaking_bytes, https://poquitopicante.blogspot.com
- PhD in ME from UC Berkeley
- Numerical modeling of physical phenomena for 15 years
- Lead architect of performance prediction software at SunPower
  - SunPower has been making the most efficient commercial PV solar panels for 30 years
  - Installed over 2 gigawatts of renewable energy on residential and commercial rooftops as well as utility power plants worldwide

July 28th Solar Impulse 2 became the first plane ever to circumnavigate the world using only solar energy from SunPower solar cells

I make models to simulate PV performance at SunPower.

Carousel Model Simulation Framework

Agenda

- Challenges and Requirements for Model Simulation (5 minutes)
  - Why did SunPower invest in a model simulation framework?
- Carousel Introduction (5 minutes)
  - A Python Model Simulation Framework
- PVLIB demonstration (20 minutes)
  - An example of implementing a model with Carousel
- Roadmap (3 minutes)
  - What's next?
- UncertaintyWrapper (2 minutes)

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

10:50

Challenges and Requirements for Model Simulation

SUNPOWER

Model Simulation Requirements

- Performance Model Simulation Overview:
  - Predicted hourly PV system energy production of both proposed and existing projects.
  - Large models involving many factors and many mathematical relations and conditions.
- SunPower Requirements:
  - Stable, performant, accurate, tested, documented, bankable net energy production models.
  - A turnkey approach to rapidly develop, maintain, and update models frequently.
  - The ability to quickly train new modelers and other collaborators.
  - Deploy and scale models in production and integrate with other applications.

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

Challenges of Current State

- Our existing models were complex and complicated …
  - Several coding paradigms and styles were mixed with no common guidelines.
  - 100% of the code was proprietary; 0% of the code was from open source projects.
  - Boilerplate procedures were implemented multiple times instead of leveraging reusable code.
  - No unit-tests. Profiling and debugging was very hard.
  - Implementing new features, making updates and bug fixes were difficult and error prone.
  - No one knew all of the the code well.
  - Training new modelers and collaborators was challenging.
  - Modelers were focused on implementing algorithms so there was no overall architecture or structure.

There was no framework, no common style, no best practices, no OSS.

Decisions

- Approach:
  - Establish a set of common guidelines for coding idioms, paradigms and styles.
  - Implement basic coding best practices (SCM, unit-testing, auto-documentation, CI, packaging, ...)
  - Leverage mature, popular, actively developed open source tools.
  - Utilize 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.
- Outcome:
  - Adopt or develop a model simulation framework using an established computer language with powerful numerical libraries and all the batteries included, *ie*: Python.

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

10:55

Carousel Introduction

SUNPOWER

What is a software framework?

- Design Patterns: Elements of Reusable Object-Oriented Software (1994) by Erich Gamma
  - "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
- Wikipedia
  - 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]

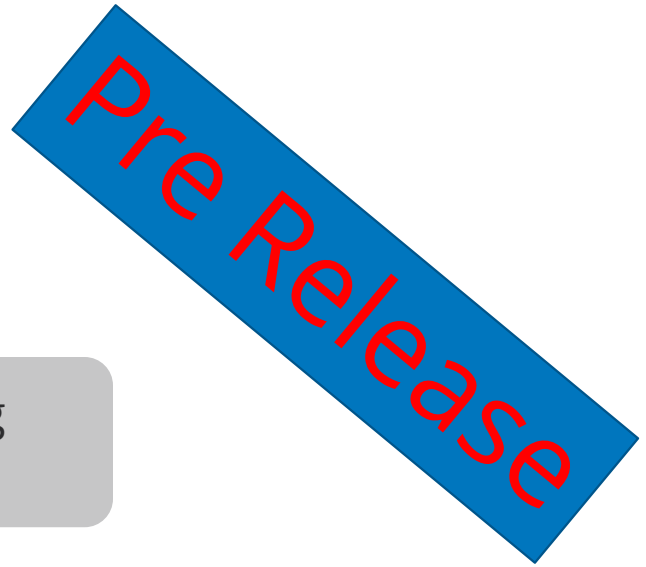
[1] Dirk Riehle (2000), *Framework Design: A Role Modeling Approach*, Swiss Federal Institute of Technology

[2] *Framework*, DocForge.org (archived)

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

Carousel Introduction

- Existing Frameworks
  - Many fields already implement component frameworks. Web frameworks such as Django, Flask and others have become the established method to implement web apps and APIs because they automate many boilerplate design patterns and result in stable, highly performant and feature rich web apps and APIs.
  - Model simulation frameworks do exist – most notably MathWorks Simulink and Modelica, but they are proprietary, and focused primarily on development and analysis, FOSS Modelica appears to be unsupported.
- Carousel - A Python Model Simulation Framework
  - PyPI: https://pypi.python.org/pypi/Carousel
  - GitHub (source, issues, wiki): https://github.com/SunPower/Carousel
  - Documentation: http://sunpower.github.io/Carousel/



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

11:00

PVLIB Demonstration

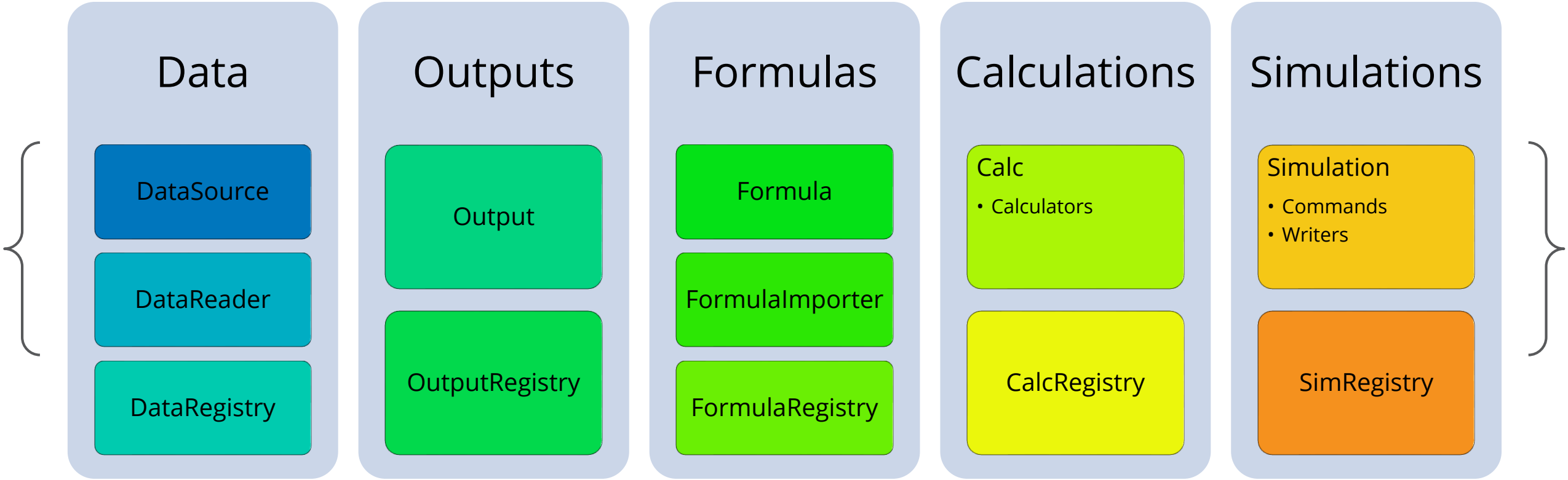
SUNPOWER

Carousel Introduction

- Carousel *defines* model components and *controls* the simulation
- Models are made of Layers → Layers are made of Sources → Sources contain data, algorithms and instructions.
  - **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.
  - **Registries**: Each layer stores data and attributes in a registry that the model controls. Items can only be registered once, but can be deleted.
- 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.

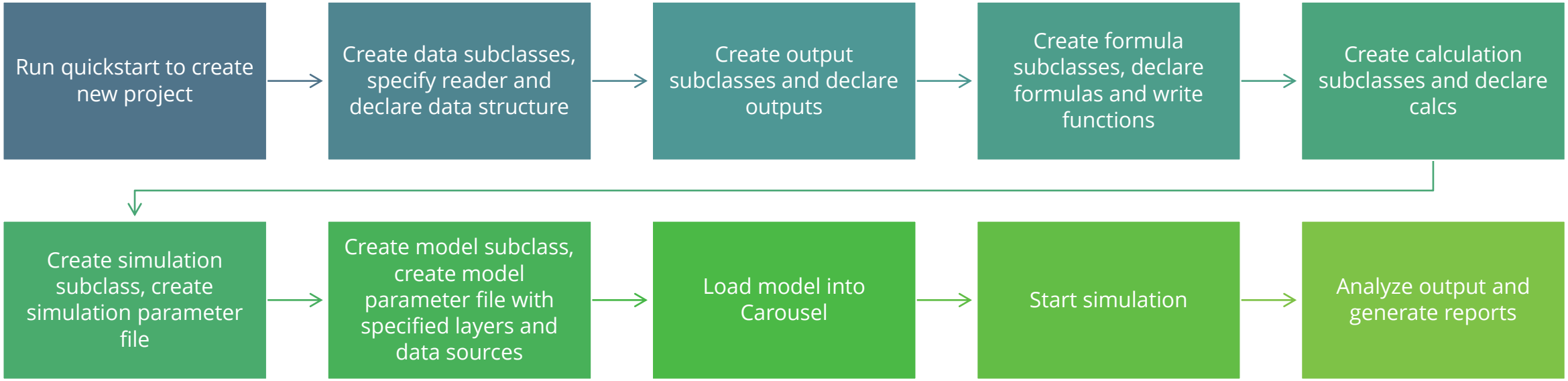
Carousel consists of five layers, new layers can be created to extend Carousel.

Carousel Structure



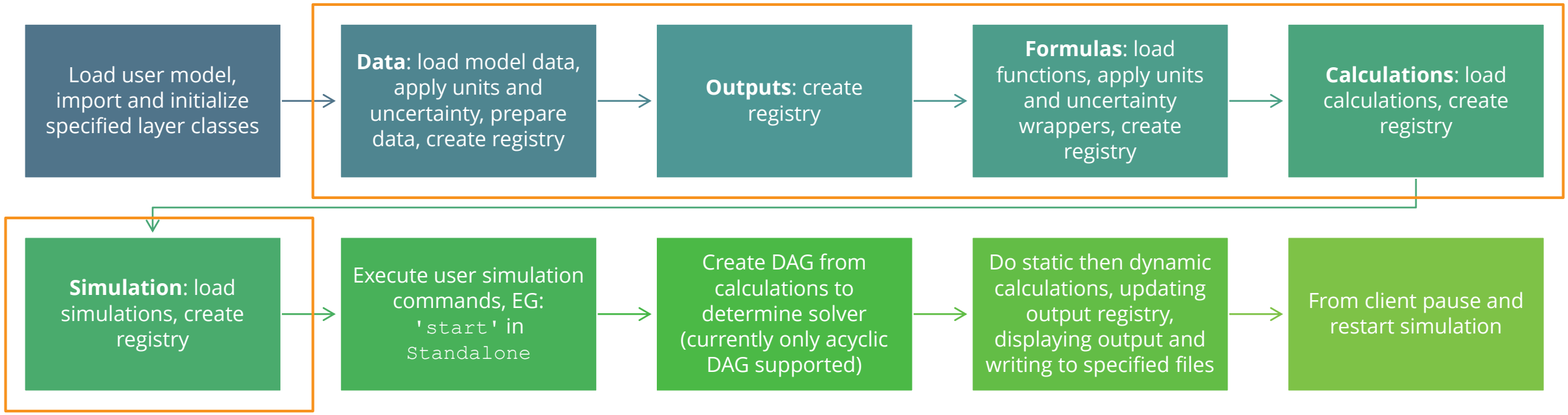
Sources form Layers and Layers form Models.

Carousel model simulation process for user



Model creation is a repeatable process with well defined component steps.

Carousel process flow chart with `BasicModel` layers



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

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.

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

Example Data Source

```
from carousel.core.data_sources import DataSource
from simeng2.data.readers import DjangoModelReader
from simengapi_app.models import PVModule, PVInverter, Weather


class PVModuleData(DataSource):
    data_reader = DjangoModelReader
    data_cache_enabled = False
    # parameters
    Isco = {'units': 'A'}
    Voco = {'units': 'V'}
    Impo = {'units': 'A'}
    Vmpo = {'units': 'V'}

    class Meta:
        model = PVModule  # sets parameters
        exclude = (
            'alias', 'vintage', 'vintageEstimated', 'parentID', 'isInActive'
        )

    def __prepare_data__(self):
        pass
```

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

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.
- Default `FormulaImporter` is `PythonImporter`.
- Formulas also have attributes: `args`, `units`, `isconstant` and `islinear`

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

Example Formulas and Parameters

```python
import pvlib
import pandas as pd
import numpy as np


def f_clearsky(times, latitude, longitude, altitude):
    times = pd.DatetimeIndex(times)
    # latitude and longitude must be scalar or else linke turbidity lookup fails
    latitude, longitude = latitude.item(), longitude.item()
    cs = pvlib.clearsky.ineichen(times, latitude, longitude, altitude)
    return cs['dni'].values, cs['ghi'].values, cs['dhi'].values


def f_solpos(times, latitude, longitude):
    times = pd.DatetimeIndex(times)
    solpos = pvlib.solarposition.get_solarposition(times, latitude, longitude)
    return solpos['apparent_zenith'].values, solpos['azimuth'].values


def f_dni_extra(times):
    times = pd.DatetimeIndex(times)
    return pvlib.irradiance.extraradiation(times)
```

```json
{
  "module": ".irradiance",
  "package": "formulas",
  "formulas": {
    "f_clearsky": {
      "args": ["times", "latitude", "longitude", "altitude"],
      "units": [["W/m**2", "W/m**2", "W/m**2"], [null, "deg", "deg", "m"]],
      "isconstant": ["times"]
    },
    "f_solpos": {
      "args": ["times", "latitude", "longitude"],
      "units": [["deg", "deg"], [null, "deg", "deg"]],
      "isconstant": ["times"]
    },
    "f_dni_extra": {"args": ["times"], "units": ["W/m**2", [null]]},
    "f_airmass": {
      "args": ["solar_zenith"], "units": ["dimensionless", ["deg"]],
      "isconstant": []
    },
    "f_pressure": {
      "args": ["altitude"], "units": ["Pa", ["m"]], "isconstant": []
    },
```

Formulas are stateless functions. Parameters provide additional attributes.

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`

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

Example Calculations

```
{
  "static": [
    {
      "formula": "f_daterange",
      "args": {
        "data": {
          "freq": "HOURLY", "dtstart": "timestamp_start",
          "count": "timestamp_count"
        }
      },
      "returns": ["timestamps"]
    },
    {
      "formula": "f_clearsky",
      "args": {
        "data": {
          "latitude": "latitude", "longitude": "longitude",
          "altitude": "elevation"
        },
        "outputs": {"times": "timestamps"}
      },
      "returns": ["dni", "ghi", "dhi"]
    },
```

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.

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

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

Example Model

```
{
  "outputs": {
  "formulas": {
    "UtilityFormulas": {
    "PerformanceFormulas": {
    "IrradianceFormulas": {
      "module": ".sandia_performance_model",
      "package": "pvpower"
    }
  },
  "data": {
    "PVPowerData": {
  },
  "calculations": {
    "UtilityCalcs": {
    "PerformanceCalcs": {
    "IrradianceCalcs": {
  },
  "simulations": {
    "Standalone": {
  }
}
```

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.

11:20

Roadmap

SUNPOWER

Roadmap

- Release plan and feature wish list is listed on Wiki
  - Validation layer: check model integrity and validate data
  - Test layer: help automate testing of models, functions, calculations, etc – all layers
  - Data scrubber layer, simplified API to integrate 3rd party apps into new layers
  - Carousel client/server to run and connect to simulations in clusters or multiprocessing
  - 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
  - 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.

11:22

UncertaintyWrapper

SUNPOWER

UncertaintyWrapper

A fast wrapper to propagate uncertainty using estimated Jacobian and linear combinations.

- Does not require any code changes. Wraps Python C-extensions and `ctypes`.
- Inputs 1st order covariance → outputs 1st order covariance and sensitivity.
- Faster than auto-differentiation. Vetted against Uncertainties, AlgoPy, NumDiffTools & others.
- Available on PyPI: https://pypi.python.org/pypi/UncertaintyWrapper
- Documenation: http://sunpower.github.io/UncertaintyWrapper/
- Collaborate and send issues at GitHub: https://github.com/SunPower/UncertaintyWrapper

UncertaintyWrapper is fast and accurate for most uses, doesn't require code changes and can be used with C-extensions and other external libraries.

SUNPOWER

Thank You

Let’s change the way our world is powered.