PVLIB_Python Documentation

Release 0+untagged.65.g94f2dd4.dirty

Sandia National Labs, Rob Andrews, University of Arizona, github

Jun 17, 2020

Contents

pvlib python is a community supported tool that provides a set of functions and classes for simulating the performance of photovoltaic energy systems. pvlib python was originally ported from the PVLIB MATLAB toolbox developed at Sandia National Laboratories and it implements many of the models and methods developed at the Labs. More information on Sandia Labs PV performance modeling programs can be found at https://pvpmc.sandia.gov/. We collaborate with the PVLIB MATLAB project, but operate independently of it.

The source code for pvlib python is hosted on github.

Please see the *Installation* page for installation help.

For examples of how to use pvlib python, please see *Package Overview* and our Jupyter Notebook tutorials. The documentation assumes general familiarity with Python, NumPy, and Pandas. Google searches will yield many excellent tutorials for these packages.

The pvlib python GitHub wiki has a Projects and publications that use pvlib python page for inspiration and listing of your application.

There is a *variable naming convention* to ensure consistency throughout the library.

CHAPTER 1

Citing pvlib python

Many of the contributors to pvlib-python work in institutions where citation metrics are used in performance or career evaluations. If you use pvlib python in a published work, please cite:

> William F. Holmgren, Clifford W. Hansen, and Mark A. Mikofski. "pvlib python: a python package for modeling solar energy systems." Journal of Open Source Software, 3(29), 884, (2018). https://doi.org/10.21105/joss.00884

Please also cite the DOI corresponding to the specific version of pvlib python that you used. pvlib python DOIs are listed at Zenodo.org

Additional pvlib python publications include:

- J. S. Stein, "The photovoltaic performance modeling collaborative (PVPMC)," in Photovoltaic Specialists Conference, 2012.
- R.W. Andrews, J.S. Stein, C. Hansen, and D. Riley, "Introduction to the open source pvlib for python photovoltaic system modelling package," in 40th IEEE Photovoltaic Specialist Conference, 2014. (paper)
- W.F. Holmgren, R.W. Andrews, A.T. Lorenzo, and J.S. Stein, "PVLIB Python 2015," in 42nd Photovoltaic Specialists Conference, 2015. (paper and the notebook to reproduce the figures)
- J.S. Stein, W.F. Holmgren, J. Forbess, and C.W. Hansen, "PVLIB: Open Source Photovoltaic Performance Modeling Functions for Matlab and Python," in 43rd Photovoltaic Specialists Conference, 2016.
- W.F. Holmgren and D.G. Groenendyk, "An Open Source Solar Power Forecasting Tool Using PVLIB-Python," in 43rd Photovoltaic Specialists Conference, 2016.

CHAPTER 2

NumFOCUS

pvlib python is a NumFOCUS Affiliated Project

CHAPTER 3

Contents

3.1 Package Overview

3.1.1 Introduction

The core mission of pvlib-python is to provide open, reliable, interoperable, and benchmark implementations of PV system models.

There are at least as many opinions about how to model PV systems as there are modelers of PV systems, so pvlib-python provides several modeling paradigms: functions, the Location/PVSystem classes, and the ModelChain class. Read more about this in the *Intro Tutorial* section.

3.1.2 User extensions

There are many other ways to organize PV modeling code. We encourage you to build on these paradigms and to share your experiences with the pvlib community via issues and pull requests.

3.1.3 Getting support

pvlib usage questions can be asked on Stack Overflow and tagged with the pvlib tag.

The pvlib-python google group is used for discussing various topics of interest to the pvlib-python community. We also make new version announcements on the google group.

If you suspect that you may have discovered a bug or if you'd like to change something about pvlib, then please make an issue on our GitHub issues page .

3.1.4 How do I contribute?

We're so glad you asked! Please see *Contributing* for information and instructions on how to contribute. We really appreciate it!

3.1.5 Credits

The pvlib-python community thanks Sandia National Lab for developing PVLIB Matlab and for supporting Rob Andrews of Calama Consulting to port the library to Python. Will Holmgren thanks the Department of Energy's Energy Efficiency and Renewable Energy Postdoctoral Fellowship Program (2014-2016), the University of Arizona Institute for Energy Solutions (2017-2018), and the DOE Solar Forecasting 2 program (2018). The pvlib-python maintainers thank all of pvlib's contributors of issues and especially pull requests. The pvlib-python community thanks all of the maintainers and contributors to the PyData stack.

3.2 Intro Tutorial

This page contains introductory examples of pvlib python usage.

3.2.1 Modeling paradigms

The backbone of pvlib-python is well-tested procedural code that implements PV system models. pvlib-python also provides a collection of classes for users that prefer object-oriented programming. These classes can help users keep track of data in a more organized way, provide some "smart" functions with more flexible inputs, and simplify the modeling process for common situations. The classes do not add any algorithms beyond what's available in the procedural code, and most of the object methods are simple wrappers around the corresponding procedural code.

Let's use each of these pvlib modeling paradigms to calculate the yearly energy yield for a given hardware configuration at a handful of sites listed below.

```
In [1]: import pandas as pd

In [2]: import matplotlib.pyplot as plt

In [3]: naive_times = pd.date_range(start='2015', end='2016', freq='1h')

# very approximate
# latitude, longitude, name, altitude, timezone
In [4]: coordinates = [(30, -110, 'Tucson', 700, 'Etc/GMT+7'),
   ...:                (35, -105, 'Albuquerque', 1500, 'Etc/GMT+7'),
   ...:                (40, -120, 'San Francisco', 10, 'Etc/GMT+8'),
   ...:                (50, 10, 'Berlin', 34, 'Etc/GMT-1')]
   ...:

In [5]: import pvlib

# get the module and inverter specifications from SAM
In [6]: sandia_modules = pvlib.pvsystem.retrieve_sam('SandiaMod')

In [7]: sapm_inverters = pvlib.pvsystem.retrieve_sam('cecinverter')

In [8]: module = sandia_modules['Canadian_Solar_CS5P_220M___2009_']

In [9]: inverter = sapm_inverters['ABB__MICRO_0_25_I_OUTD_US_208__208V_']

In [10]: temperature_model_parameters = pvlib.temperature.TEMPERATURE_MODEL_
↪PARAMETERS['sapm']['open_rack_glass_glass']

# specify constant ambient air temp and wind for simplicity
In [11]: temp_air = 20
```

(continues on next page)

(continued from previous page)

```
In [12]: wind_speed = 0
```

Procedural

The straightforward procedural code can be used for all modeling steps in pvlib-python.

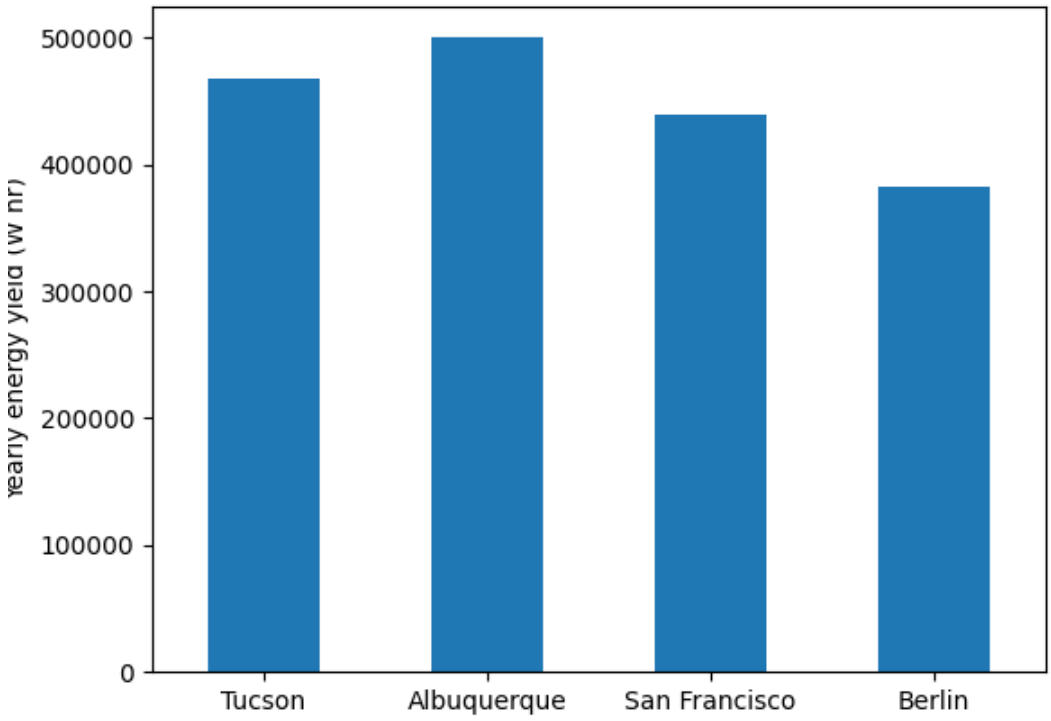
The following code demonstrates how to use the procedural code to accomplish our system modeling goal:

```
In [13]: system = {'module': module, 'inverter': inverter,
   ....:           'surface_azimuth': 180}
   ....:

In [14]: energies = {}

In [15]: for latitude, longitude, name, altitude, timezone in coordinates:
   ....:     times = naive_times.tz_localize(timezone)
   ....:     system['surface_tilt'] = latitude
   ....:     solpos = pvlib.solarposition.get_solarposition(times, latitude,␣
↪longitude)
   ....:     dni_extra = pvlib.irradiance.get_extra_radiation(times)
   ....:     airmass = pvlib.atmosphere.get_relative_airmass(solpos['apparent_zenith
↪'])
   ....:     pressure = pvlib.atmosphere.alt2pres(altitude)
   ....:     am_abs = pvlib.atmosphere.get_absolute_airmass(airmass, pressure)
   ....:     tl = pvlib.clearsky.lookup_linke_turbidity(times, latitude, longitude)
   ....:     cs = pvlib.clearsky.ineichen(solpos['apparent_zenith'], am_abs, tl,
   ....:                                  dni_extra=dni_extra, altitude=altitude)
   ....:     aoi = pvlib.irradiance.aoi(system['surface_tilt'], system['surface_
↪azimuth'],
   ....:                                solpos['apparent_zenith'], solpos['azimuth'])
   ....:     total_irrad = pvlib.irradiance.get_total_irradiance(system['surface_tilt
↪'],
   ....:                                                         system['surface_
↪azimuth'],
   ....:                                                         solpos['apparent_
↪zenith'],
   ....:                                                         solpos['azimuth'],
   ....:                                                         cs['dni'], cs['ghi'],
↪ cs['dhi'],
   ....:                                                         dni_extra=dni_extra,
   ....:                                                         model='haydavies')
   ....:     tcell = pvlib.temperature.sapm_cell(total_irrad['poa_global'],
   ....:                                         temp_air, wind_speed,
   ....:                                         **temperature_model_parameters)
   ....:     effective_irradiance = pvlib.pvsystem.sapm_effective_irradiance(
   ....:         total_irrad['poa_direct'], total_irrad['poa_diffuse'],
   ....:         am_abs, aoi, module)
   ....:     dc = pvlib.pvsystem.sapm(effective_irradiance, tcell, module)
   ....:     ac = pvlib.pvsystem.snlinverter(dc['v_mp'], dc['p_mp'], inverter)
   ....:     annual_energy = ac.sum()
   ....:     energies[name] = annual_energy
   ....:

In [16]: energies = pd.Series(energies)
```

(continues on next page)

(continued from previous page)

```
# based on the parameters specified above, these are in W*hrs
In [17]: print(energies.round(0))
Tucson           467494.0
Albuquerque      500230.0
San Francisco    439787.0
Berlin           383203.0
dtype: float64

In [18]: energies.plot(kind='bar', rot=0)
Out[18]: <matplotlib.axes._subplots.AxesSubplot at 0x7fa77a823278>

In [19]: plt.ylabel('Yearly energy yield (W hr)')
Out[19]: Text(0, 0.5, 'Yearly energy yield (W hr)')
```



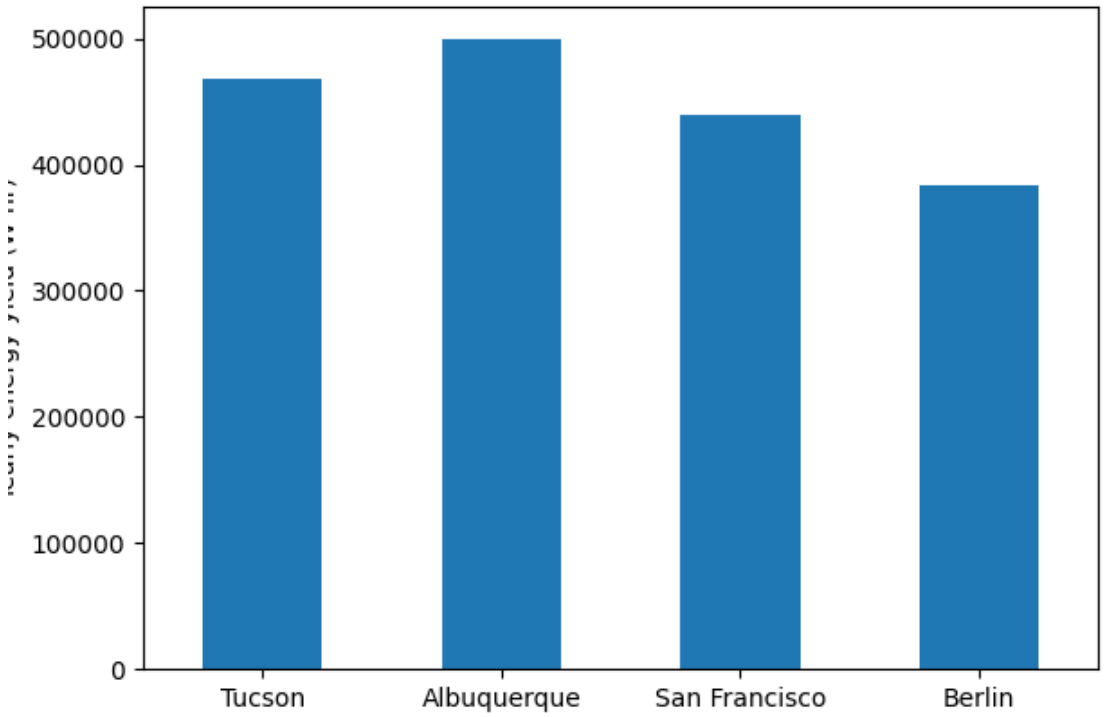
Object oriented (Location, PVSystem, ModelChain)

The first object oriented paradigm uses a model where a *PVSystem* object represents an assembled collection of modules, inverters, etc., a *Location* object represents a particular place on the planet, and a *ModelChain* object describes the modeling chain used to calculate PV output at that Location. This can be a useful paradigm if you prefer to think about the PV system and its location as separate concepts or if you develop your own ModelChain subclasses. It can also be helpful if you make extensive use of Location-specific methods for other calculations. pvlib-python also includes a *SingleAxisTracker* class that is a subclass of *PVSystem*.

The following code demonstrates how to use *Location*, *PVSystem*, and *ModelChain* objects to accomplish

our system modeling goal. ModelChain objects provide convenience methods that can provide default selections for models and can also fill necessary input with modeled data. For example, no air temperature or wind speed data is provided in the input *weather* DataFrame, so the ModelChain object defaults to 20 C and 0 m/s. Also, no irradiance transposition model is specified (keyword argument *transposition* for ModelChain) so the ModelChain defaults to the *haydavies* model. In this example, ModelChain infers the DC power model from the module provided by examining the parameters defined for the module.

```
In [20]: from pvlib.pvsystem import PVSystem

In [21]: from pvlib.location import Location

In [22]: from pvlib.modelchain import ModelChain

In [23]: system = PVSystem(module_parameters=module,
   ....:                   inverter_parameters=inverter,
   ....:                   temperature_model_parameters=temperature_model_parameters)
   ....:

In [24]: energies = {}

In [25]: for latitude, longitude, name, altitude, timezone in coordinates:
   ....:     times = naive_times.tz_localize(timezone)
   ....:     location = Location(latitude, longitude, name=name, altitude=altitude,
   ....:                         tz=timezone)
   ....:     weather = location.get_clearsky(times)
   ....:     mc = ModelChain(system, location,
   ....:                     orientation_strategy='south_at_latitude_tilt')
   ....:     mc.run_model(weather)
   ....:     annual_energy = mc.ac.sum()
   ....:     energies[name] = annual_energy
   ....:

In [26]: energies = pd.Series(energies)

# based on the parameters specified above, these are in W*hrs
In [27]: print(energies.round(0))
Tucson           467459.0
Albuquerque      500151.0
San Francisco    439786.0
Berlin           383200.0
dtype: float64

In [28]: energies.plot(kind='bar', rot=0)
Out[28]: <matplotlib.axes._subplots.AxesSubplot at 0x7fa77ee400f0>

In [29]: plt.ylabel('Yearly energy yield (W hr)')
Out[29]: Text(0, 0.5, 'Yearly energy yield (W hr)')
```



Object oriented (LocalizedPVSystem)

The second object oriented paradigm uses a model where a *LocalizedPVSystem* represents a PV system at a particular place on the planet. This can be a useful paradigm if you're thinking about a power plant that already exists.

The *LocalizedPVSystem* inherits from both *PVSystem* and *Location*, while the *LocalizedSingleAxisTracker* inherits from *SingleAxisTracker* (itself a subclass of *PVSystem*) and *Location*. The *LocalizedPVSystem* and *LocalizedSingleAxisTracker* classes may contain bugs due to the relative difficulty of implementing multiple inheritance. The *LocalizedPVSystem* and *LocalizedSingleAxisTracker* may be deprecated in a future release. We recommend that most modeling workflows implement *Location*, *PVSystem*, and *ModelChain*.

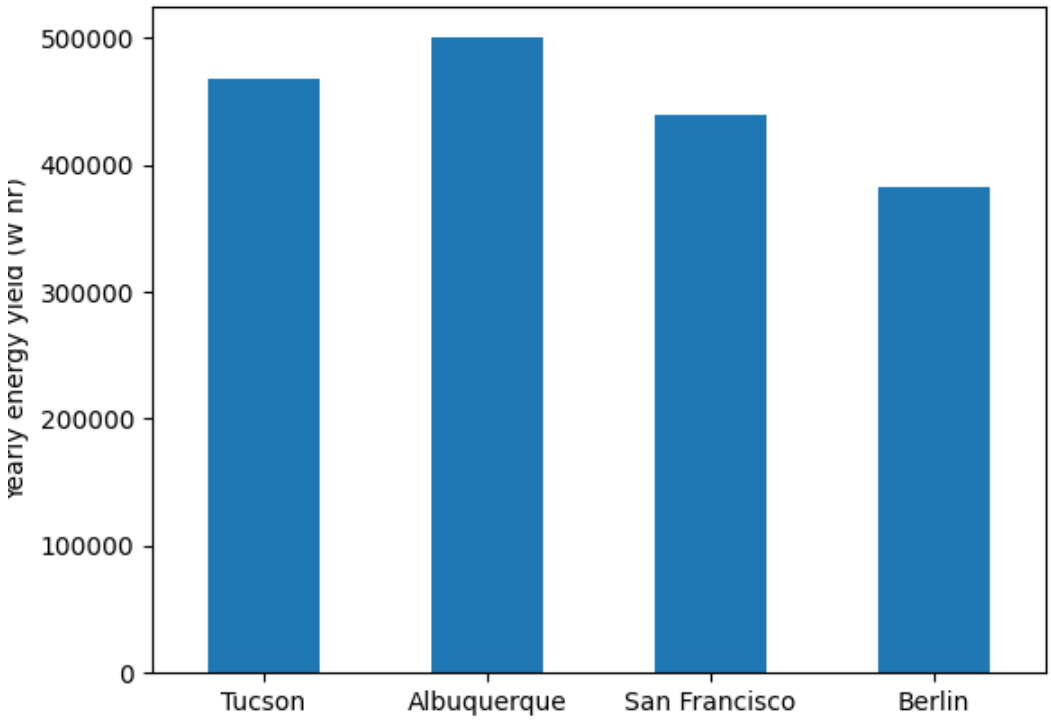
The following code demonstrates how to use a *LocalizedPVSystem* object to accomplish our modeling goal:

```
In [30]: from pvlib.pvsystem import LocalizedPVSystem

In [31]: energies = {}

In [32]: for latitude, longitude, name, altitude, timezone in coordinates:
   ....:     localized_system = LocalizedPVSystem(module_parameters=module,
   ....:                                          inverter_parameters=inverter,
   ....:                                          temperature_model_
↪parameters=temperature_model_parameters,
   ....:                                          surface_tilt=latitude,
   ....:                                          surface_azimuth=180,
   ....:                                          latitude=latitude,
```

(continues on next page)

(continued from previous page)

```
   ....:                                         longitude=longitude,
   ....:                                         name=name,
   ....:                                         altitude=altitude,
   ....:                                         tz=timezone)
   ....:     times = naive_times.tz_localize(timezone)
   ....:     clearsky = localized_system.get_clearsky(times)
   ....:     solar_position = localized_system.get_solarposition(times)
   ....:     total_irrad = localized_system.get_irradiance(solar_position['apparent_
↪zenith'],
   ....:                                                   solar_position['azimuth'],
   ....:                                                   clearsky['dni'],
   ....:                                                   clearsky['ghi'],
   ....:                                                   clearsky['dhi'])
   ....:     tcell = localized_system.sapm_celltemp(total_irrad['poa_global'],
   ....:                                            temp_air, wind_speed)
   ....:     aoi = localized_system.get_aoi(solar_position['apparent_zenith'],
   ....:                                    solar_position['azimuth'])
   ....:     airmass = localized_system.get_airmass(solar_position=solar_position)
   ....:     effective_irradiance = localized_system.sapm_effective_irradiance(
   ....:         total_irrad['poa_direct'], total_irrad['poa_diffuse'],
   ....:         airmass['airmass_absolute'], aoi)
   ....:     dc = localized_system.sapm(effective_irradiance, tcell)
   ....:     ac = localized_system.snlinverter(dc['v_mp'], dc['p_mp'])
   ....:     annual_energy = ac.sum()
   ....:     energies[name] = annual_energy
   ....:

In [33]: energies = pd.Series(energies)

# based on the parameters specified above, these are in W*hrs
In [34]: print(energies.round(0))
Tucson           467459.0
Albuquerque      500151.0
San Francisco    439786.0
Berlin           383200.0
dtype: float64

In [35]: energies.plot(kind='bar', rot=0)
Out[35]: <matplotlib.axes._subplots.AxesSubplot at 0x7fa77a779630>

In [36]: plt.ylabel('Yearly energy yield (W hr)')
Out[36]: Text(0, 0.5, 'Yearly energy yield (W hr)')
```



3.3 Example Gallery

This gallery shows examples of pvlib functionality. Community contributions are welcome!

3.3.1 Kimber Soiling Model

Examples of soiling using the Kimber model.

This example shows basic usage of pvlib's Kimber Soiling model[1] with `pvlib.soiling.kimber()`.

References

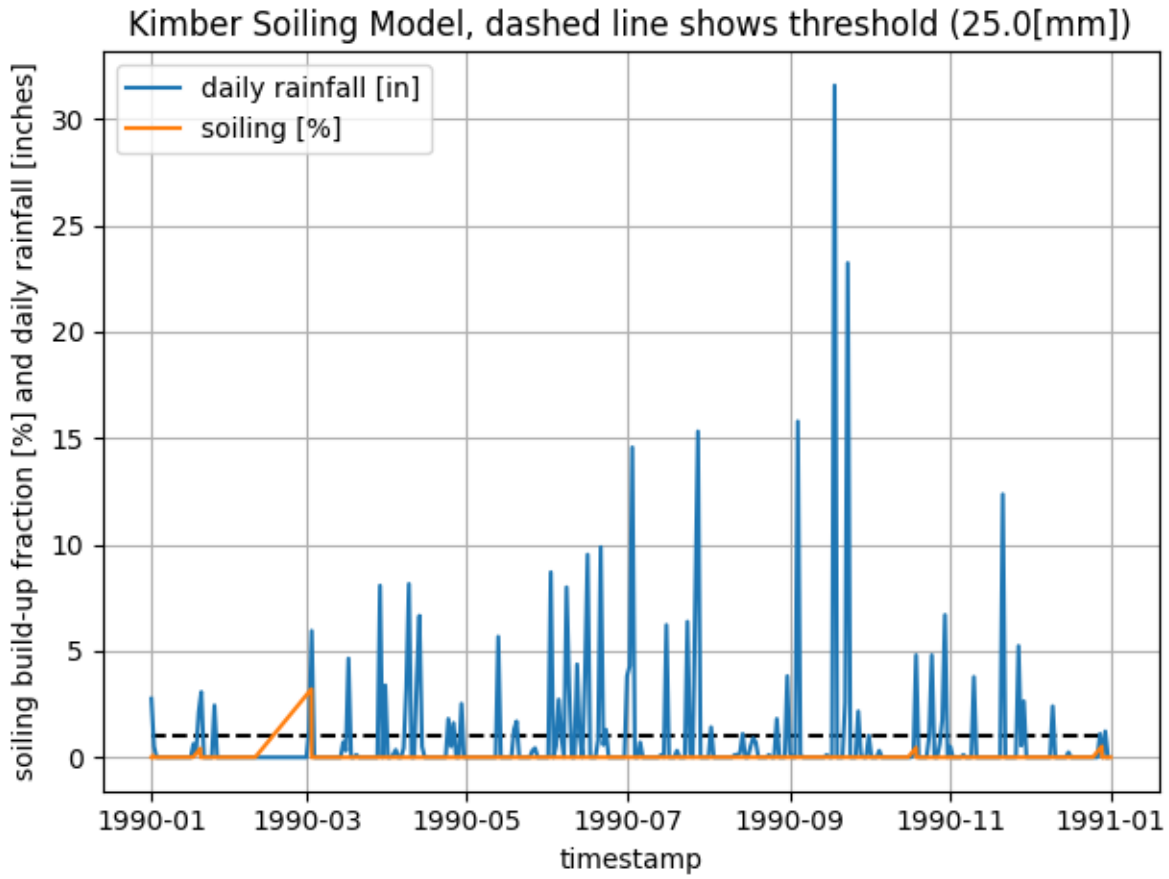
The Kimber Soiling model assumes that soiling builds up at a constant rate until cleaned either manually or by rain. The rain must reach a threshold to clean the panels. When rains exceeds the threshold, it's assumed the earth is damp for a grace period before it begins to soil again. There is a maximum soiling build up that cannot be exceeded even if there's no rain or manual cleaning.

[1] "The Effect of Soiling on Large Grid-Connected Photovoltaic Systems in California and the Southwest Region of the United States," Adrianne Kimber, et al., IEEE 4th World Conference on Photovoltaic Energy Conference, 2006, DOI: 10.1109/WCPEC.2006.279690

Threshold

The example shown here demonstrates how the threshold affects soiling. Because soiling depends on rainfall, loading weather data is always the first step.

```
from datetime import datetime
import pathlib
from matplotlib import pyplot as plt
from pvlib.iotools import read_tmy3
from pvlib.soiling import kimber
import pvlib

# get full path to the data directory
DATA_DIR = pathlib.Path(pvlib.__file__).parent / 'data'

# get TMY3 data with rain
greensboro, _ = read_tmy3(DATA_DIR / '723170TYA.CSV', coerce_year=1990)
# get the rain data
greensboro_rain = greensboro.Lprecipdepth
# calculate soiling with no wash dates and cleaning threshold of 25-mm of rain
THRESHOLD = 25.0
soiling_no_wash = kimber(greensboro_rain, cleaning_threshold=THRESHOLD)
soiling_no_wash.name = 'soiling'
# daily rain totals
daily_rain = greensboro_rain.iloc[:-1].resample('D').sum()
plt.plot(
    daily_rain.index.to_pydatetime(), daily_rain.values/25.4,
    soiling_no_wash.index.to_pydatetime(), soiling_no_wash.values*100.0)
plt.hlines(
    THRESHOLD/25.4, xmin=datetime(1990, 1, 1), xmax=datetime(1990, 12, 31),
    linestyles='--')
plt.grid()
plt.title(
    f'Kimber Soiling Model, dashed line shows threshold ({THRESHOLD}[mm])')
plt.xlabel('timestamp')
plt.ylabel('soiling build-up fraction [%] and daily rainfall [inches]')
plt.legend(['daily rainfall [in]', 'soiling [%]'])
plt.tight_layout()

plt.show()
```



Total running time of the script: (0 minutes 0.434 seconds)

3.3.2 Single-axis tracking

Examples of modeling tilt angles for single-axis tracker arrays.

This example shows basic usage of pvlib's tracker position calculations with *pvlib.tracking.singleaxis()*. The examples shown here demonstrate how the tracker parameters affect the generated tilt angles.

Because tracker angle is based on where the sun is in the sky, calculating solar position is always the first step.

True-tracking

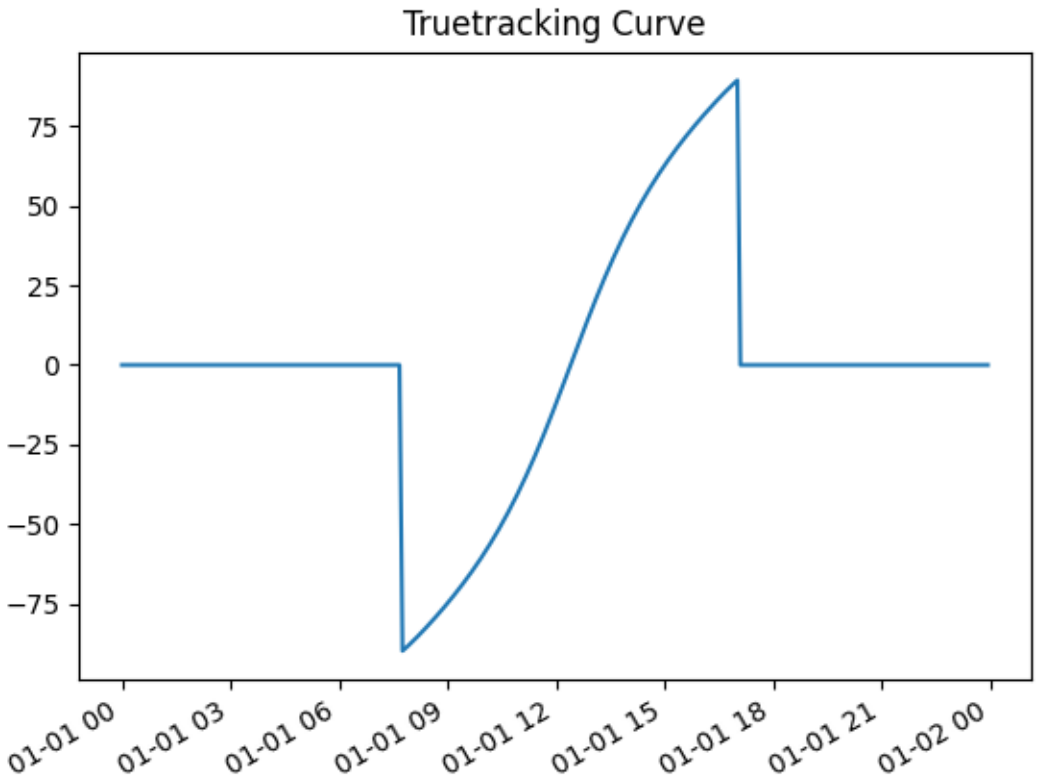
The basic tracking algorithm is called "true-tracking". It orients the panels towards the sun as much as possible in order to maximize the cross section presented towards incoming beam irradiance.

```
from pvlib import solarposition, tracking
import pandas as pd
import matplotlib.pyplot as plt

tz = 'US/Eastern'
lat, lon = 40, -80

times = pd.date_range('2019-01-01', '2019-01-02', closed='left', freq='5min',
```

(continues on next page)

(continued from previous page)

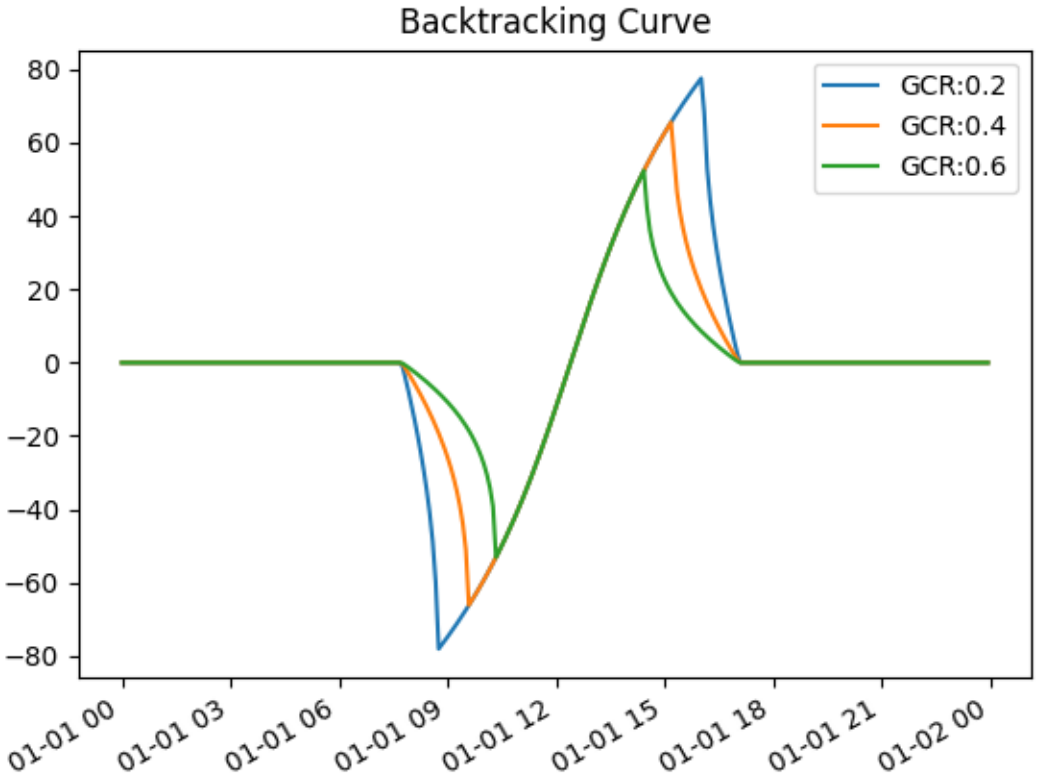
```
                       tz=tz)
solpos = solarposition.get_solarposition(times, lat, lon)

truetracking_angles = tracking.singleaxis(
    apparent_zenith=solpos['apparent_zenith'],
    apparent_azimuth=solpos['azimuth'],
    axis_tilt=0,
    axis_azimuth=180,
    max_angle=90,
    backtrack=False,  # for true-tracking
    gcr=0.5)  # irrelevant for true-tracking

truetracking_position = truetracking_angles['tracker_theta'].fillna(0)
truetracking_position.plot(title='Truetracking Curve')

plt.show()
```



Backtracking

Because truetracking yields steep tilt angle in morning and afternoon, it will cause row to row shading as the shadows from adjacent rows fall on each other. To prevent this, the trackers can rotate backwards when the sun is near the horizon – "backtracking". The shading angle depends on row geometry, so the gcr parameter must be specified. The greater the gcr, the tighter the row spacing and the more aggressively the array must backtrack.

```
fig, ax = plt.subplots()

for gcr in [0.2, 0.4, 0.6]:
    backtracking_angles = tracking.singleaxis(
        apparent_zenith=solpos['apparent_zenith'],
        apparent_azimuth=solpos['azimuth'],
        axis_tilt=0,
        axis_azimuth=180,
        max_angle=90,
        backtrack=True,
        gcr=gcr)

    backtracking_position = backtracking_angles['tracker_theta'].fillna(0)
    backtracking_position.plot(title='Backtracking Curve',
                               label='GCR:{:0.01f}'.format(gcr),
                               ax=ax)

plt.legend()
plt.show()
```



Total running time of the script: (0 minutes 0.328 seconds)

3.3.3 GHI to POA Transposition

Example of generating clearsky GHI and POA irradiance.

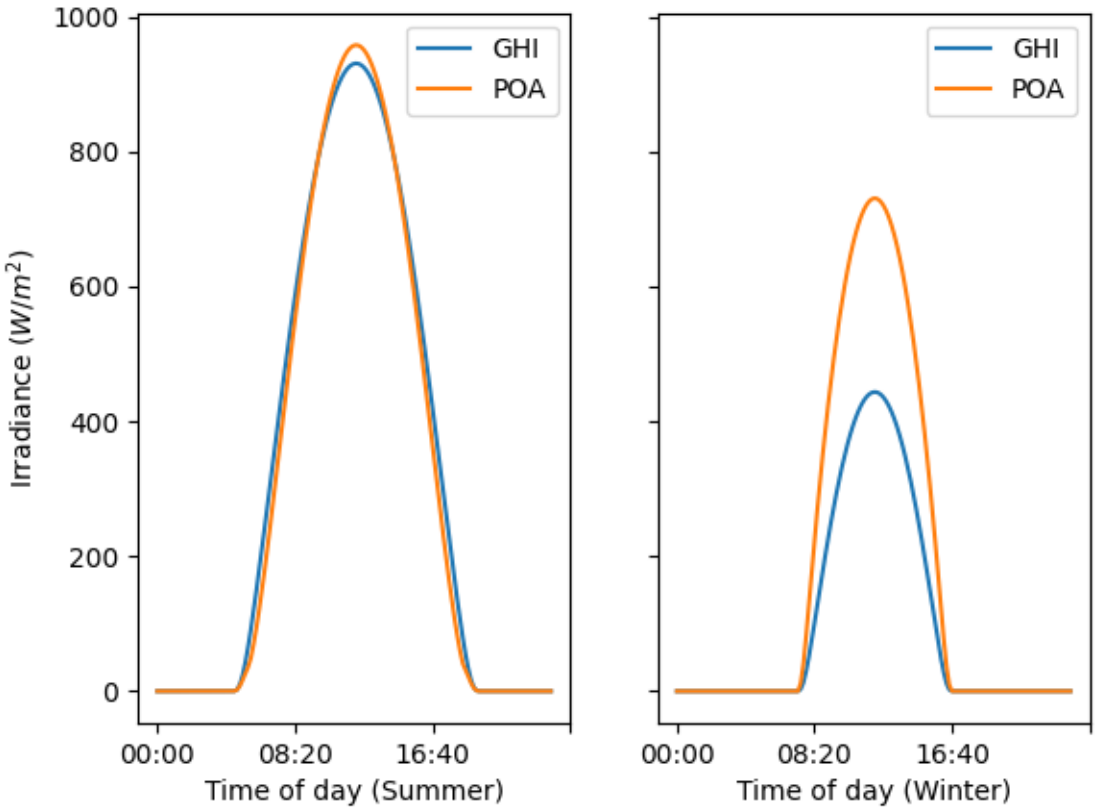
This example shows how to use the *pvlib.location.Location.get_clearsky()* method to generate clearsky GHI data as well as how to use the *pvlib.irradiance.get_total_irradiance()* function to transpose GHI data to Plane of Array (POA) irradiance.

```
from pvlib import location
from pvlib import irradiance
import pandas as pd
from matplotlib import pyplot as plt

# For this example, we will be using Golden, Colorado
tz = 'MST'
lat, lon = 39.755, -105.221

# Create location object to store lat, lon, timezone
site = location.Location(lat, lon, tz=tz)


# Calculate clear-sky GHI and transpose to plane of array
# Define a function so that we can re-use the sequence of operations with
# different locations
def get_irradiance(site_location, date, tilt, surface_azimuth):
    # Creates one day's worth of 10 min intervals
    times = pd.date_range(date, freq='10min', periods=6*24,
                          tz=site_location.tz)
    # Generate clearsky data using the Ineichen model, which is the default
    # The get_clearsky method returns a dataframe with values for GHI, DNI,
    # and DHI
    clearsky = site_location.get_clearsky(times)
    # Get solar azimuth and zenith to pass to the transposition function
    solar_position = site_location.get_solarposition(times=times)
    # Use the get_total_irradiance function to transpose the GHI to POA
    POA_irradiance = irradiance.get_total_irradiance(
        surface_tilt=tilt,
        surface_azimuth=surface_azimuth,
        dni=clearsky['dni'],
        ghi=clearsky['ghi'],
        dhi=clearsky['dhi'],
        solar_zenith=solar_position['apparent_zenith'],
        solar_azimuth=solar_position['azimuth'])
    # Return DataFrame with only GHI and POA
    return pd.DataFrame({'GHI': clearsky['ghi'],
                         'POA': POA_irradiance['poa_global']})


# Get irradiance data for summer and winter solstice, assuming 25 degree tilt
# and a south facing array
summer_irradiance = get_irradiance(site, '06-20-2020', 25, 180)
winter_irradiance = get_irradiance(site, '12-21-2020', 25, 180)

# Convert Dataframe Indexes to Hour:Minute format to make plotting easier
summer_irradiance.index = summer_irradiance.index.strftime("%H:%M")
winter_irradiance.index = winter_irradiance.index.strftime("%H:%M")

# Plot GHI vs. POA for winter and summer
fig, (ax1, ax2) = plt.subplots(1, 2, sharey=True)
summer_irradiance['GHI'].plot(ax=ax1, label='GHI')
summer_irradiance['POA'].plot(ax=ax1, label='POA')
winter_irradiance['GHI'].plot(ax=ax2, label='GHI')
```

(continues on next page)

(continued from previous page)

```
winter_irradiance['POA'].plot(ax=ax2, label='POA')
ax1.set_xlabel('Time of day (Summer)')
ax2.set_xlabel('Time of day (Winter)')
ax1.set_ylabel('Irradiance ($W/m^2$)')
ax1.legend()
ax2.legend()
plt.show()
```



Note that in Summer, there is not much gain when comparing POA irradiance to GHI. In the winter, however, POA irradiance is significantly higher than GHI. This is because, in winter, the sun is much lower in the sky, so a tilted array will be at a more optimal angle compared to a flat array. In summer, the sun gets much higher in the sky, and there is very little gain for a tilted array compared to a flat array.

Total running time of the script: (0 minutes 0.610 seconds)

3.3.4 Calculating a module's IV curves

Examples of modeling IV curves using a single-diode circuit equivalent model.

Calculating a module IV curve for certain operating conditions is a two-step process. Multiple methods exist for both parts of the process. Here we use the De Soto model[1] to calculate the electrical parameters for an IV curve at a certain irradiance and temperature using the module's base characteristics at reference conditions. Those parameters are then used to calculate the module's IV curve by solving the single-diode equation using the Lambert W method.

[1] W. De Soto et al., "Improvement and validation of a model for photovoltaic array performance", Solar Energy, vol 80, pp. 78-88, 2006.

The single-diode equation is a circuit-equivalent model of a PV cell and has five electrical parameters that depend on the operating conditions. For more details on the single-diode equation and the five parameters, see the PVPMC single diode page.

References

Calculating IV Curves

This example uses *pvlib.pvsystem.calcparams_desoto()* to calculate the 5 electrical parameters needed to solve the single-diode equation. *pvlib.pvsystem.singlediode()* is then used to generate the IV curves.

```
from pvlib import pvsystem
import pandas as pd
import matplotlib.pyplot as plt

# Example module parameters for the Canadian Solar CS5P-220M:
parameters = {
    'Name': 'Canadian Solar CS5P-220M',
    'BIPV': 'N',
    'Date': '10/5/2009',
    'T_NOCT': 42.4,
    'A_c': 1.7,
    'N_s': 96,
    'I_sc_ref': 5.1,
    'V_oc_ref': 59.4,
    'I_mp_ref': 4.69,
    'V_mp_ref': 46.9,
    'alpha_sc': 0.004539,
    'beta_oc': -0.22216,
    'a_ref': 2.6373,
    'I_L_ref': 5.114,
    'I_o_ref': 8.196e-10,
    'R_s': 1.065,
    'R_sh_ref': 381.68,
    'Adjust': 8.7,
    'gamma_r': -0.476,
    'Version': 'MM106',
    'PTC': 200.1,
    'Technology': 'Mono-c-Si',
}

cases = [
    (1000, 55),
    (800, 55),
    (600, 55),
    (400, 25),
    (400, 40),
    (400, 55)
]

conditions = pd.DataFrame(cases, columns=['Geff', 'Tcell'])

# adjust the reference parameters according to the operating
# conditions using the De Soto model:
```

(continues on next page)

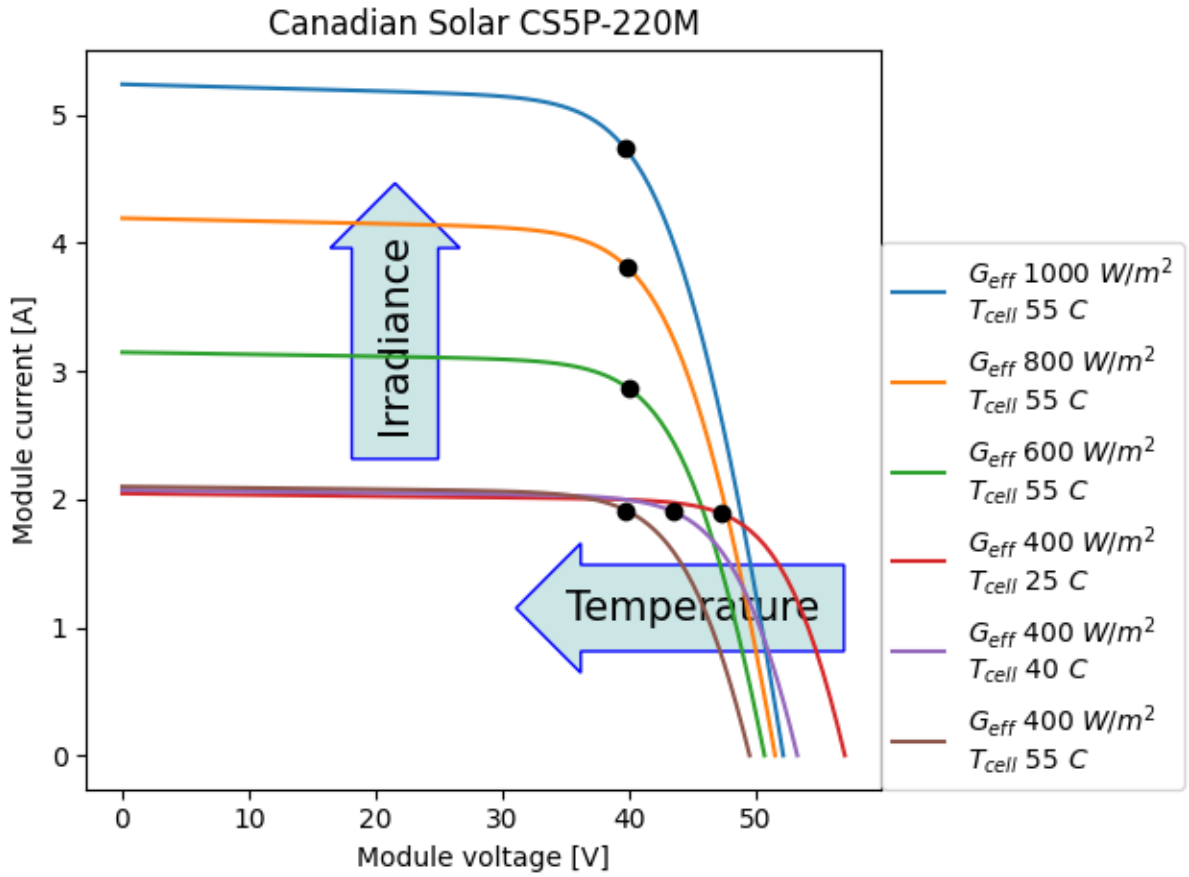
(continued from previous page)

```
IL, I0, Rs, Rsh, nNsVth = pvsystem.calcparams_desoto(
    conditions['Geff'],
    conditions['Tcell'],
    alpha_sc=parameters['alpha_sc'],
    a_ref=parameters['a_ref'],
    I_L_ref=parameters['I_L_ref'],
    I_o_ref=parameters['I_o_ref'],
    R_sh_ref=parameters['R_sh_ref'],
    R_s=parameters['R_s'],
    EgRef=1.121,
    dEgdT=-0.0002677
)

# plug the parameters into the SDE and solve for IV curves:
curve_info = pvsystem.singlediode(
    photocurrent=IL,
    saturation_current=I0,
    resistance_series=Rs,
    resistance_shunt=Rsh,
    nNsVth=nNsVth,
    ivcurve_pnts=100,
    method='lambertw'
)

# plot the calculated curves:
plt.figure()
for i, case in conditions.iterrows():
    label = (
        "$G_{eff}$ " + f"{case['Geff']} $W/m^2$\n"
        "$T_{cell}$ " + f"{case['Tcell']} $C$"
    )
    plt.plot(curve_info['v'][i], curve_info['i'][i], label=label)
    v_mp = curve_info['v_mp'][i]
    i_mp = curve_info['i_mp'][i]
    # mark the MPP
    plt.plot([v_mp], [i_mp], ls='', marker='o', c='k')

plt.legend(loc=(1.0, 0))
plt.xlabel('Module voltage [V]')
plt.ylabel('Module current [A]')
plt.title(parameters['Name'])
plt.show()
plt.gcf().set_tight_layout(True)


# draw trend arrows
def draw_arrow(ax, label, x0, y0, rotation, size, direction):
    style = direction + 'arrow'
    bbox_props = dict(boxstyle=style, fc=(0.8, 0.9, 0.9), ec="b", lw=1)
    t = ax.text(x0, y0, label, ha="left", va="bottom", rotation=rotation,
                size=size, bbox=bbox_props, zorder=-1)

    bb = t.get_bbox_patch()
    bb.set_boxstyle(style, pad=0.6)


ax = plt.gca()
```

(continues on next page)

(continued from previous page)

```
draw_arrow(ax, 'Irradiance', 20, 2.5, 90, 15, 'r')
draw_arrow(ax, 'Temperature', 35, 1, 0, 15, 'l')

print(pd.DataFrame({
    'i_sc': curve_info['i_sc'],
    'v_oc': curve_info['v_oc'],
    'i_mp': curve_info['i_mp'],
    'v_mp': curve_info['v_mp'],
    'p_mp': curve_info['p_mp'],
}))
```



Out:

```
      i_sc       v_oc      i_mp       v_mp        p_mp
0  5.235561  52.129782  4.742078  39.617325  187.868460
1  4.190781  51.483032  3.805385  39.871330  151.725745
2  3.144837  50.649227  2.862258  39.952853  114.355370
3  2.043319  56.987478  1.886511  47.285357   89.204359
4  2.070523  53.238566  1.900933  43.492748   82.676788
5  2.097727  49.474043  1.911758  39.742278   75.977636
```

Total running time of the script: (0 minutes 0.302 seconds)

3.3.5 Sun path diagram

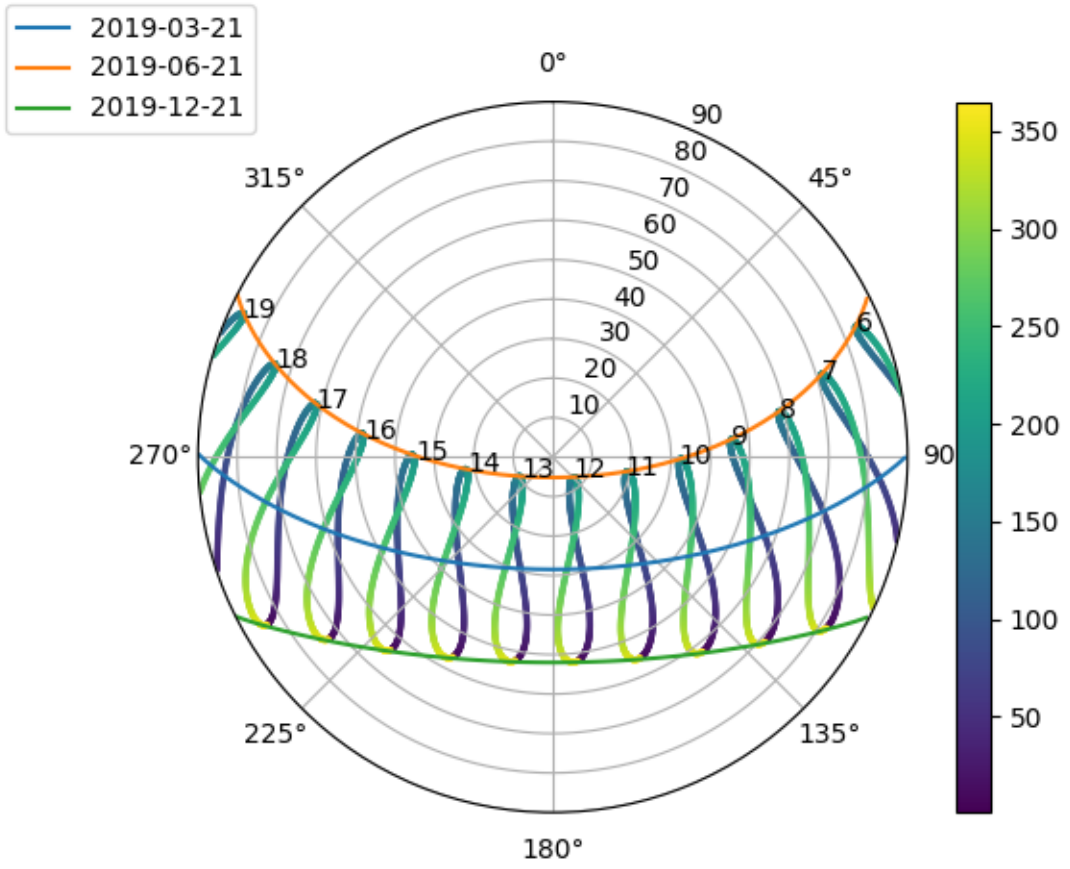
Examples of generating sunpath diagrams.

This example shows basic usage of pvlib's solar position calculations with *pvlib.solarposition.get_solarposition()*. The examples shown here will generate sunpath diagrams that shows solar position over a year.

Polar plot

Below is an example plot of solar position in polar coordinates.

```
from pvlib import solarposition
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

tz = 'Asia/Calcutta'
lat, lon = 28.6, 77.2

times = pd.date_range('2019-01-01 00:00:00', '2020-01-01', closed='left',
                      freq='H', tz=tz)
solpos = solarposition.get_solarposition(times, lat, lon)
# remove nighttime
solpos = solpos.loc[solpos['apparent_elevation'] > 0, :]

ax = plt.subplot(1, 1, 1, projection='polar')
# draw the analemma loops
points = ax.scatter(np.radians(solpos.azimuth), solpos.apparent_zenith,
                    s=2, label=None, c=solpos.index.dayofyear)
ax.figure.colorbar(points)

# draw hour labels
for hour in np.unique(solpos.index.hour):
    # choose label position by the smallest radius for each hour
    subset = solpos.loc[solpos.index.hour == hour, :]
    r = subset.apparent_zenith
    pos = solpos.loc[r.idxmin(), :]
    ax.text(np.radians(pos['azimuth']), pos['apparent_zenith'], str(hour))

# draw individual days
for date in pd.to_datetime(['2019-03-21', '2019-06-21', '2019-12-21']):
    times = pd.date_range(date, date+pd.Timedelta('24h'), freq='5min', tz=tz)
    solpos = solarposition.get_solarposition(times, lat, lon)
    solpos = solpos.loc[solpos['apparent_elevation'] > 0, :]
    label = date.strftime('%Y-%m-%d')
    ax.plot(np.radians(solpos.azimuth), solpos.apparent_zenith, label=label)

ax.figure.legend(loc='upper left')

# change coordinates to be like a compass
ax.set_theta_zero_location('N')
ax.set_theta_direction(-1)
ax.set_rmax(90)

plt.show()
```



This is a polar plot of hourly solar zenith and azimuth. The figure-8 patterns are called analemmas and show how the sun's path slowly shifts over the course of the year . The colored lines show the single-day sun paths for the winter and summer solstices as well as the spring equinox.

The soltice paths mark the boundary of the sky area that the sun traverses over a year. The diagram shows that there is no point in the year when is the sun directly overhead (zenith=0) – note that this location is north of the Tropic of Cancer.

Examining the sun path for the summer solstice in particular shows that the sun rises north of east, crosses into the southern sky around 10 AM for a few hours before crossing back into the northern sky around 3 PM and setting north of west. In contrast, the winter solstice sun path remains in the southern sky the entire day. Moreover, the diagram shows that the winter solstice is a shorter day than the summer soltice – in December, the sun rises after 7 AM and sets before 6 PM, whereas in June the sun is up before 6 AM and sets after 7 PM.

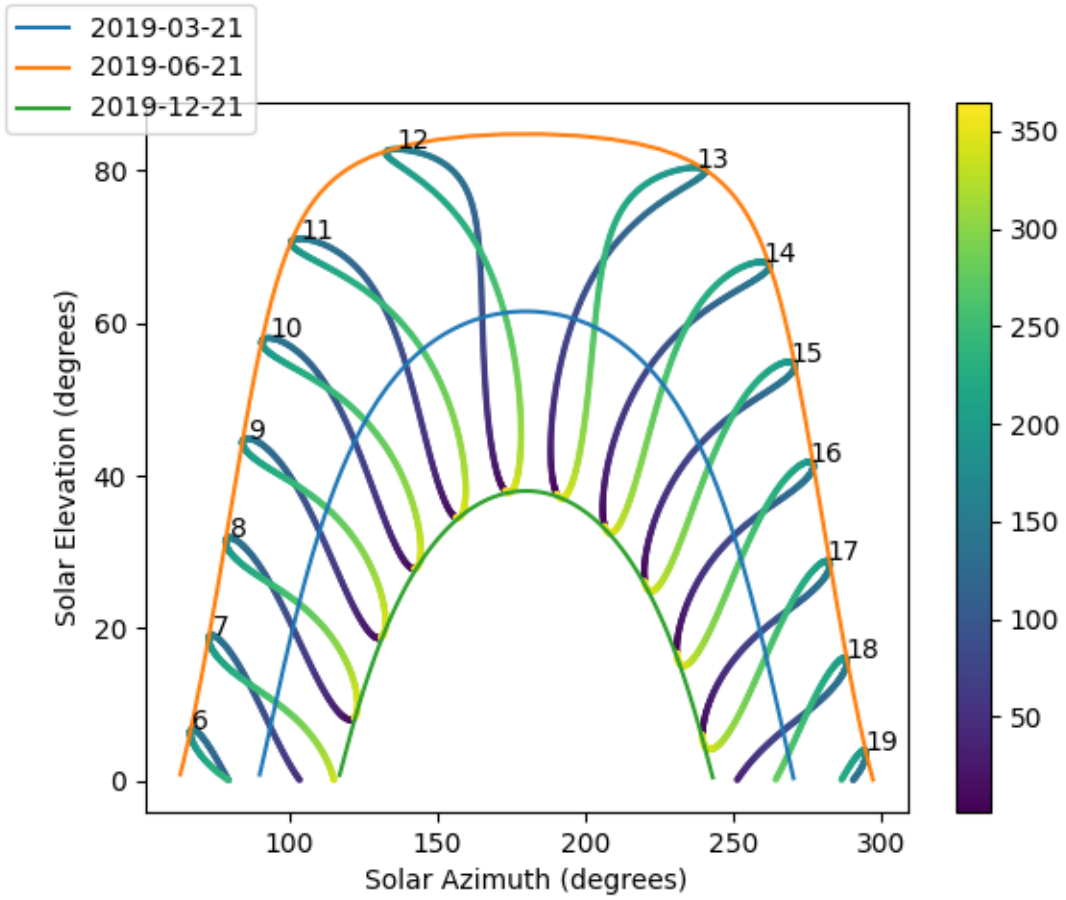
Another use of this diagram is to determine what times of year the sun is blocked by obstacles. For instance, for a mountain range on the western side of an array that extends 10 degrees above the horizon, the sun is blocked:

- after about 6:30 PM on the summer solstice
- after about 5:30 PM on the spring equinox
- after about 4:30 PM on the winter solstice

PVSyst Plot

PVSyst users will be more familiar with sunpath diagrams in Cartesian coordinates:

```
from pvlib import solarposition
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

tz = 'Asia/Calcutta'
lat, lon = 28.6, 77.2
times = pd.date_range('2019-01-01 00:00:00', '2020-01-01', closed='left',
                      freq='H', tz=tz)

solpos = solarposition.get_solarposition(times, lat, lon)
# remove nighttime
solpos = solpos.loc[solpos['apparent_elevation'] > 0, :]

fig, ax = plt.subplots()
points = ax.scatter(solpos.azimuth, solpos.apparent_elevation, s=2,
                    c=solpos.index.dayofyear, label=None)
fig.colorbar(points)

for hour in np.unique(solpos.index.hour):
    # choose label position by the largest elevation for each hour
    subset = solpos.loc[solpos.index.hour == hour, :]
    height = subset.apparent_elevation
    pos = solpos.loc[height.idxmax(), :]
    ax.text(pos['azimuth'], pos['apparent_elevation'], str(hour))

for date in pd.to_datetime(['2019-03-21', '2019-06-21', '2019-12-21']):
    times = pd.date_range(date, date+pd.Timedelta('24h'), freq='5min', tz=tz)
    solpos = solarposition.get_solarposition(times, lat, lon)
    solpos = solpos.loc[solpos['apparent_elevation'] > 0, :]
    label = date.strftime('%Y-%m-%d')
    ax.plot(solpos.azimuth, solpos.apparent_elevation, label=label)

ax.figure.legend(loc='upper left')
ax.set_xlabel('Solar Azimuth (degrees)')
ax.set_ylabel('Solar Elevation (degrees)')

plt.show()
```



Total running time of the script: (0 minutes 1.031 seconds)

3.4 What's New

These are new features and improvements of note in each release.

3.4.1 v0.8.0 (Month day, year)

API Changes

Enhancements

Bug fixes

- Fixed unit and default value errors in *pvlib.soiling.hsu()*. (GHXXX)

Testing

- Decorator `pvlib.conftest.fail_on_pvlib_version()` can now be applied to functions that require args or kwargs. (GH973)

Documentation

- Improved formatting and content of docstrings in `pvlib.atmosphere`. (GH969)
- Fix LaTeX rendering in *`pvlib.singlediode.bishop88()`*. (GH967)
- Clarify units for heat loss factors in *`pvlib.temperature.pvsyst_cell()`* and *`pvlib.temperature.faiman()`*. (GH960)

Requirements

Contributors

- Cliff Hansen (@cwhanse)
- Kevin Anderson (@kanderso-nrel)
- Mark Mikofski (@mikofski)
- Joshua S. Stein (@jsstein)

3.4.2 v0.7.2 (April 22, 2020)

API Changes

- `pvlib.forecast.ForecastModel` now requires `start` and `end` arguments to be tz-localized. (GH877, GH879)
- *`pvlib.iotools.read_tmy3()`* when coerced to a single year now returns indices that are monotonically increasing. Therefore, the last index will be January 1, 00:00 of the *next* year. (GH910)
- Renamed `pvlib.losses` to `pvlib.soiling`. Additional loss models will go into code modules named for the loss or effect type. (GH935, GH891)
- Renamed `pvlib.losses.soiling_hsu` to *`pvlib.soiling.hsu()`* (GH935)

Enhancements

- TMY3 dataframe returned by *`read_tmy3()`* now contains the original `Date (MM/DD/YYYY)` and `Time (HH:MM)` columns that the indices were parsed from. (GH866)
- Add `pvlib.pvsystem.PVSystem.faiman()` and added `temperature_model='faiman'` option to *`ModelChain`* (GH897) (GH836).
- Add Kimber soiling model `pvlib.losses.soiling_kimber()`. (GH860)
- Add *`pvlib.iotools.read_pvgis_tmy()`* for files downloaded using the
- **Add Kimber soiling model *`pvlib.soiling.kimber()`*. (GH860,** :issue‘935‘)
- Add *`read_pvgis_tmy()`* for files downloaded using the PVGIS tool. (GH880)
- Add *`pvlib.temperature.sapm_cell_from_module()`* to convert back of module temperature to cell temperature (GH927)
- Add new module `pvlib.snow` to contain models related to snow coverage and effects on a PV system. (GH764)

- Add snow coverage model *pvlib.snow.coverage_nrel()* and function to identify when modules are fully covered by snow *pvlib.snow.fully_covered_nrel()*. (GH577)
- Add function *pvlib.snow.dc_loss_nrel()* for effect of snow coverage on DC output. (GH764)
- Add capability to calculate current at reverse bias using an avalanche breakdown model, affects *pvlib.singlediode.bishop88()*, *pvlib.singlediode.bishop88_i_from_v()*, *pvlib.singlediode.bishop88_v_from_i()*, *pvlib.singlediode.bishop88_mpp()*. (GH948)
- Add weather data arguments in get_solarpostion method of modelchain.ModelChain.prepare_inputs `modelchain.ModelChain.prepare_inputs()` (GH936)

Bug fixes

- Fix *read_tmy3()* parsing when February contains a leap year. (GH866)
- Implement NREL Developer Network API key for consistent success with API calls in `pvlib.tests.iotools.test_psm3`. (GH873)
- Fix issue with *pvlib.location.Location* creation when passing `tz=datetime.timezone.utc`. (GH879)
- Fix documentation homepage title to "pvlib python" based on first heading on the page. (GH890) (GH888)
- Fix missing 0.7.0 what's new entries about changes to `PVSystem.pvwatts_ac`. Delete unreleased 0.6.4 what's new file. (GH898)
- Compatibility with cftime 1.1. (GH895)
- Minor implemention changes to avoid runtime and deprecation warnings in *detect_clearsky()*, *martin_ruiz_diffuse()*, `soiling_hsu()`, and various test functions.
- Fix *read_tmy3()* so that when coerced to a single year the TMY3 index will be monotonically increasing. (GH910)
- Fix `pvlib.spa.julian_day_dt()` so that microseconds are scaled correctly (GH940) (GH942)

Testing

- Rename *system* fixture to *sapm_dc_snl_ac_system* in model chain tests. (GH908, GH915).
- Implement pytest-remotedata to increase test suite speed. Requires `--remote-data` pytest flag to execute data retrieval tests over a network. (GH882)(GH896)
- Add Python3.8 to Azure Pipelines CI. (GH903)(GH904)
- Add documentation build test to Azure Pipelines CI. (GH909)
- Implement the *pytest.mark.flaky* decorator from *pytest-rurunfailures* https://github.com/pytest-dev/pytest-rerunfailures on all network dependent iotools tests to repeat them on failure. (GH919)
- Separate azure-pipelines.yml platform-specific tests to their own templates located in `./ci/azure/`. (GH926)

Documentation

- Add NumFOCUS affiliation to Sphinx documentation. (GH862)
- Add example of IV curve generation. (GH872)
- Add section about gallery examples to Contributing guide. (GH905)

- Add section with link to Code of Conduct in Contributing guide. (GH922)
- Add example of GHI to POA transposition (GH933)

Requirements

- nrel-pysam (optional) minimum set to v1.0.0 (GH874)
- cftime (optional) minimum set to >=1.1.1. Use of *only_use_python_datetimes* kwarg in *netCDF4.num2date* in forecast.py requires >=1.1.1 which is >=Python3.6. (GH947)

Contributors

- Mark Mikofski (@mikofski)
- Cliff Hansen (@cwhanse)
- Cameron T. Stark (@camerontstark)
- Will Holmgren (@wholmgren)
- Kevin Anderson (@kanderso-nrel)
- Karthikeyan Singaravelan (@tirkarthi)
- Siyan (Veronica) Guo (@veronicaguo)
- Eric Fitch (@ericf900)
- Joseph Palakapilly (@JPalakapilly)
- Auguste Colle(@augustecolle)
- Ahan M R (@Ahanmr)

3.4.3 v0.7.1 (January 17, 2020)

Enhancements

- Added *read_psm3()* to read local NSRDB PSM3 files and *parse_psm3()* to parse local NSRDB PSM3 file-like objects. (GH841)
- Added *leap_day* parameter to *iotools.get_psm3* instead of hardcoding it as False.
- Added *get_pvgis_tmy()* to get PVGIS TMY datasets. (GH845)
- Added *parse_epw()* to parse a file-like buffer containing weather data in the EPW format.
- Added a new module *pvlib.losses* for various loss models.
- Added the Humboldt State University soiling model `soiling_hsu()`. (GH739)

Bug fixes

- Fix error in logic for emitting deprecation warning in *sapm()* (GH844)
- Changed the PSM3 API endpoint for TMY requests in *iotools.get_psm3*.

Testing

- Added single-year PSM3 API test for *iotools.get_psm3*.
- Added tests for *iotools.parse_psm3* and *iotools.read_psm3*.
- Change *pvlib/test* folder to *pvlib/tests* and reorganize tests into subfolders, *e.g.*: created *pvlib/tests/iotools* (GH859)
- replace *os.path* with *pathlib* and stringify path objects for Python<=3.5

Documentation

- Created an Example Gallery. (GH846)
- Updated list of allowed years for *iotools.get_psm3*.

Contributors

- Kevin Anderson (@kanderso-nrel)
- Mark Mikofski (@mikofski)
- @dzimmanck
- Will Holmgren (@wholmgren)
- Cliff Hansen (@cwhanse)
- Valliappan CA (@nappaillav)
- Anton Driesse (@adriesse)

3.4.4 v0.7.0 (December 18, 2019)

This is a major release that drops support for Python 2 and Python 3.4. We recommend all users of v0.6.3 upgrade to this release after checking API compatibility notes.

Python 2.7 support ended on June 1, 2019. (GH501) **Minimum numpy version is now 1.12.0. Minimum pandas version is now 0.18.1.** (GH830, GH748)

API Breaking Changes

- The *effective_irradiance* argument for `pvsystem.sapm()` now requires units of W/m^2. Previously, units for this input were suns. A RuntimeWarning warning is raised if all *effective_irradiance* < *2.0*.
- The output of `pvsystem.sapm_effective_irradiance()` is now in units of W/m2 rather than suns.
- Calling *`pvlib.pvsystem.retrieve_sam()`* with no parameters will raise an exception instead of displaying a dialog.
- The *modelchain.ModelChain.diode_params* attribute is now formatted in a pandas.DataFrame with *DatetimeIndex*, rather than in a tuple.
- `PVSystem.pvwatts_ac` now uses inverter DC input limit `PVSystem.inverter_parameters['pdc0']` instead of module nameplate capacity `PVSystem.module_parameters['pdc0']`. (GH734)

- `ModelChain.infer_ac_model` now uses the presence of the key `'pdc0'` `PVSystem.inverter_parameters` to determine if the *pvwatts_ac* inverter model should be used. The inference method previously looked for the key in `PVSystem.module_parameters`. (GH734)

API Changes with Deprecations

- **Changes related to cell temperature models (GH678):**
  - **Changes to functions**
    - Moved functions for cell temperature from *pvsystem.py* to *temperature.py*.
    - Renamed *pvsystem.sapm_celltemp* and *pvsystem.pvsyst_celltemp* to *temperature.sapm_cell* and *temperature.pvsyst_cell*.
    - *temperature.sapm_cell* returns only the cell temperature, whereas the old *pvsystem.sapm_celltemp* returned a *DataFrame* with both cell and module temperatures.
    - Created *temperature.sapm_module* to return module temperature using the SAPM temperature model.
    - Changed the order of arguments for *pvsystem.sapm_celltemp*, *pvsystem.pvsyst_celltemp* and *PVSystem.sapm_celltemp* to be consistent among cell temperature model functions.
    - Removed *model* as a kwarg from *temperature.sapm_cell* and *temperature.pvsyst_cell*. These functions now require model-specific parameters.
    - Added the argument *irrad_ref*, default value 1000, to *temperature.sapm_cell*.
  - **Changes to named temperature model parameter sets**
    - Renamed *pvsystem.TEMP_MODEL_PARAMS* to *temperature.TEMPERATURE_MODEL_PARAMETERS*.
    - *temperature.TEMPERATURE_MODEL_PARAMETERS* uses dict rather than tuple for a parameter set.
    - Names for parameter sets in *temperature.TEMPERATURE_MODEL_PARAMETERS* have changed.
    - Parameter sets for the SAPM cell temperature model named 'open_rack_polymer_thinfilm_steel' and '22x_concentrator_tracker' are considered obsolete and have been removed.
  - **Changes to *PVSystem* class**
    - Changed the *model* kwarg in *PVSystem.sapm_celltemp* and *PVSystem.pvsyst_celltemp* to *parameter_set*. *parameter_set* expects a str which is a valid key for *temperature.TEMPERATURE_MODEL_PARAMETERS* for the corresponding temperature model.
    - Added an attribute *PVSystem.module_type* (str) to record module front and back materials, default is *glass_polymer*.
    - Changed meaning of *PVSystem.racking_model* to describe racking only, e.g., default is *open_rack*.
    - Added an attribute *PVSystem.temperature_model_parameters* (dict). to contain temperature model parameters.
    - If *PVSystem.temperature_model_parameters* is not specified and *PVSystem.racking_model* and *PVSystem.module_type* combine to a valid parameter set name for the SAPM cell temperature model, that parameter set is assigned to *PVSystem.temperature_model_parameters*. Otherwise *PVSystem.temperature_model_parameters* is assigned an empty dict. The result is that the

default parameter set for SAPM cell temperature model is *open_rack_glass_polymer*; the old default was *open_rack_glass_glass*.

- **Changes to *ModelChain* class**
  - *ModelChain.temp_model* renamed to *ModelChain.temperature_model*.
  - *ModelChain.temperature_model* now defaults to *None*. The temperature model can be inferred from *PVSystem.temperature_model_parameters*.
  - *ModelChain.temperature_model_parameters* now defaults to *None*. The temperature model can be inferred from *PVSystem.temperature_model_parameters*.
  - *ModelChain.temps* attribute renamed to *ModelChain.cell_temperature*, and its datatype is now *numeric* rather than *DataFrame*.
  - If *PVSystem.temperature_model_parameters* is not specified, *ModelChain* defaults to old behavior, using the SAPM temperature model with parameter set *open_rack_glass_glass*. This behavior is deprecated, and will be removed in v0.8. In v0.8 *PVSystem.temperature_model_parameters* will be required for *ModelChain*.
  - Implemented *pvsyst* as an option for *ModelChain.temperature_model*.
  - *modelchain.basic_chain* has a new required argument *temperature_model_parameters*.

- **Changes related to IAM (AOI loss) functions (GH680):**
  - **Changes to functions**
    - Moved functions from *pvsystem.py* to *iam.py*. *pvsystem* IAM functions are deprecated and will be removed in v0.8.
    - **Functions are renamed to a consistent pattern:**
      - *pvsystem.physicaliam* is *iam.physical*
      - *pvsystem.ashraeiam* is *iam.ashrae*
      - *pvsystem.sapm_aoi_loss* is *iam.sapm*
  - **Changes to *PVSystem* class**
    - IAM models are provided by *PVSystem.get_iam* with kwarg *iam_model*.
    - Methods *PVSystem.ashraeiam*, *PVSystem.physicaliam* and *PVSystem.sapm_aoi_loss* are deprecated and will be removed in v0.8.

- **Changes related to spectral modifier (GH782):**
  - **Changes to functions**
    - Added the argument *pw_min* and *pw_max*, default values 0.1 and 8 resp., to *atmosphere.first_solar_spectral_correction*. This function now returns NaN if pw value higher than *pw_max*.

- The *times* keyword argument has been deprecated in the `pvlib.modelchain.ModelChain.run_model()`, `pvlib.modelchain.ModelChain.prepare_inputs()`, and `pvlib.modelchain.ModelChain.complete_irradiance()` methods. Model times are now determined by the input *weather*. *DataFrame*. Therefore, the *weather* DataFrame must have a *DatetimeIndex*. The *weather* argument of the above methods is now the first, required positional argument and the *times* argument is kept as the second keyword argument for capability during the deprecation period.
- Parameter *pvsystem.DC_MODEL_PARAMS* is renamed to *pvsystem._DC_MODEL_PARAMS*. Users should not rely on this dictionary's existence or structure.

Other API Changes

- `pvlib.iotools.midc.read_midc()` now passes additional keyword arguments to *pandas.read_csv*
- Add *timeout* argument to `pvlib.iotools.midc.read_midc_raw_data_from_nrel()`
- `pvlib.bifacial` is now imported when `pvlib` is imported. (GH766)

Enhancements

- Created one new temperature model function: *pvlib.temperature.faiman()*. (GH750)
- Created two new incidence angle modifier (IAM) functions: *pvlib.iam.martin_ruiz()* and *pvlib.iam.interp()*. (GH751)
- Created one new incidence angle modifier (IAM) function for diffuse irradiance: *pvlib.iam.martin_ruiz_diffuse()*. (GH751)
- Add the *martin_ruiz* IAM function as an option for *ModelChain.aoi_model*.
- Updated the file for module parameters for the CEC model, from the SAM file dated 2017-6-5 to the SAM file dated 2019-03-05. (GH761)
- Updated the file for inverter parameters for the CEC model, from the SAM file dated 2018-3-18 to the SAM file dated 2019-03-05. (GH761)
- Added recombination current parameters to bishop88 single-diode functions and also to *pvlib.pvsystem.max_power_point()*. (GH762)
- Add *ivtools* module to contain functions for IV model fitting.
- Add *fit_sde_sandia()*, a simple method to fit the single diode equation to an IV curve.
- Add *fit_sdm_cec_sam()*, a wrapper for the CEC single diode model fitting function '6parsolve' from NREL's System Advisor Model.
- Add *fit_sdm_desoto()*, a method to fit the De Soto single diode model to the typical specifications given in manufacturers datasheets.
- Add *timeout* to *pvlib.iotools.get_psm3()*.
- Add *wvm()*, a port of the wavelet variability model for computing reductions in variability due to a spatially distributed plant.
- Add *from_epw()*, a method to create a Location object from epw metadata, typically coming from *pvlib.iotools.epw.read_epw*.

Bug fixes

- Fix handling of keyword arguments in *forecasts.get_processed_data*. (GH745)
- Fix output as Series feature in `pvlib.pvsystem.ashraeiam()`.
- Fix rounding issue in *clearsky._linearly_scale*, a function that converts longitude or latitude degree to an index number in a Linke turbidity lookup table. Also rename the function to *clearsky._degrees_to_index*. (GH754)
- Fix reading raw MIDC CSV files from NREL where the number of header columns does not match the number of data columns.
- Fix installation issue due to missing `requests` dependency. (GH725)

- `PVSystem.pvwatts_ac` now uses inverter DC input limit `PVSystem.inverter_parameters['pdc0']` instead of module nameplate capacity `PVSystem.module_parameters['pdc0']`. (GH734)

Testing

- Added 30 minutes to timestamps in *test_psm3.csv* to match change in NSRDB (GH733)
- Added tests for methods in bifacial.py.
- Added tests for changes to cell temperature models.
- Add tests configuration for bare python environment (no conda). (GH727)
- Added tests for changes to IAM models.
- Added test for *ModelChain.infer_aoi_model*.

Documentation

- Corrected docstring for *pvsystem.PVSystem.sapm*
- Fixed broken ipython examples from CEC data updates
- Edited docstring for *pvsystem.sapm* to remove DataFrame option for input *module*. The DataFrame option was never tested and would cause an error if used. (GH785)
- Note warning about _TMY3.epw files retrieved from energyplus.net in docstring of *epw.read_epw*
- Improved sphinx rendering of API reference entries for *clearsky.ineichen*, *clearsky.haurwitz*, *tracking.singleaxis*, *iotools.read_midc*, *Location.from_tmy*, *ModelChain.run_model*, *ModelChain.complete_irradiance*, and *ModelChain.prepare_inputs*
- Removed duplicate *pvwatts_losses* entry in *api.rst*

Removal of prior version deprecations

- Removed *irradiance.extraradiation*.
- Removed *irradiance.grounddiffuse*.
- Removed *irradiance.total_irrad*.
- Removed *irradiance.globalinplane*.
- Removed *atmosphere.relativeairmass*.
- Removed *atmosphere.relativeairmass*.
- Removed *solarposition.get_sun_rise_set_transit*.
- Removed *tmy* module.
- Removed *ModelChain.singlediode* method.
- Removed *ModelChain.prepare_inputs* clearsky assumption when no irradiance data was provided.

Requirements

- numpy minimum increased to v1.12.0, released in 2017. (GH830)
- pandas minimum increased to v1.18.1, released in 2016. (GH748)

Contributors

- Mark Campanellli (@markcampanelli)
- Will Holmgren (@wholmgren)
- Cliff Hansen (@cwhanse)
- Oscar Dowson (@odow)
- Anton Driesse (@adriesse)
- Alexander Morgan (@alexandermorgan)
- Miguel Sánchez de León Peque (@Peque)
- Tanguy Lunel (@tylunel)
- Veronica Guo (@veronicaguo)
- Joseph Ranalli (@jranalli)
- Tony Lorenzo (@alorenzo175)
- Todd Karin (@toddkarin)
- Mark Mikofski (@mikofski)
- Kevin Anderson (@kevinsa5)
- Cameron Stark (@camerontstark)
- Janine Freeman (@janinefreeman)
- Roel Loonen (@roelloonen)
- Birgit Schachler (@birgits)
- Hamilton Kibbe (@hamiltonkibbe)
- Adam Peretti (@aperetti)
- Cedric Leroy (@cedricleroy)
- Joseph Palakapilly (@JPalakapillyKWH)
- (@shall-resurety)

3.4.5 v0.6.3 (May 15, 2019)

This is a minor release on top of v0.6.2 to fix an installation issue. We recommend that all users of v0.6.1 and v0.6.2 upgrade to this release.

Python 2.7 support will end on June 1, 2019. Releases made after this date will require Python 3. This release is likely to be the last that supports Python 2.7. (GH501)

Bug fixes

- Fix installation issue due to missing `requests` dependency. (GH725)

Contributors

- Will Holmgren (@wholmgren)

3.4.6 v0.6.2 (May 15, 2019)

This is a minor release. We recommend all users of v0.6.1 upgrade to this release.

Python 2.7 support will end on June 1, 2019. Releases made after this date will require Python 3. This release is likely to be the last that supports Python 2.7. (GH501)

Minimum pandas requirement bumped 0.15.0=>0.16.0

API Changes

- *erbs()* *doy* argument changed to *datetime_or_doy* to be consistent with allowed types and similar functions (*disc()*, *get_extra_radiation()*). (GH681)
- *erbs()* DataFrame vs. OrderedDict return behavior now determined by type of *datetime_or_doy* instead of *ghi* or *zenith*. (GH681)
- Added *min_cos_zenith* and *max_zenith* keyword arguments to *erbs()*. (GH681)
- Deprecated *prepare_inputs()* assumption of clear sky if no irradiance fields were provided. (GH705, GH707)
- Remove automatic column name mapping from *read_midc()* and *read_midc_raw_data_from_nrel()* and added optional keyword argument *variable_map* to map columns. (GH721)
- Update *pvfactors_timeseries()* and tests to use `pvfactors` v1.0.1 (GH699)

Enhancements

- Add US CRN data reader to *IO Tools*. (GH666)
- Add SOLRAD data reader to *IO Tools*. (GH667)
- Add EPW data reader to *IO Tools*. (GH591)
- Add PSM3 reader to *IO Tools*. (GH592)
- Improve ModelChain inference method error text. (GH621)

Bug fixes

- Compatibility with pandas 0.24 deprecations. (GH659)
- *pvwatts_ac()* raised `ZeroDivisionError` when called with scalar `pdc=0` and a `RuntimeWarning` for `array(0)` input. Now correctly returns 0s of the appropriate type. (GH675)
- Fixed *erbs()* behavior when zenith is near 90 degrees. (GH681)

- *dni()* now referenced in API under Decomposing and Combining irradiance header. (GH686)
- Fixed NaN output from *singleaxis()* when sun near horizon. (GH656)
- Fixed numpy warnings in *singleaxis()* when comparing NaN values to limits. (GH622)
- Change ModelChain to apply `pvwatts_losses` to `mc.dc` instead of `mc.ac`. (GH696)
- Fixed a bug in the day angle equation for the ASCE extraterrestrial irradiance model. (GH211)
- Silenced divide by 0 irradiance warnings in *klucher()* and *calcparams_desoto()*. (GH698)
- Fix *NDFD* model by updating variables.
- Fix `format_index()` to parse non one-minute data correctly. (GH709)

Testing

- Remove most expected warnings emitted by test suite. (GH698)

Contributors

- Cliff Hansen (@cwhanse)
- Will Holmgren (@wholmgren)
- Roel Loonen (@roelloonen)
- Todd Hendricks (@tahentx)
- Kevin Anderson (@kevinsa5)
- @bentomlinson
- @yxh289
- Jonathan Gaffiot (@jgaffiot)
- Leland Boeman (@lboeman)
- Marc Anoma (@anomam)

3.4.7 v0.6.1 (January 31, 2019)

This is a minor release. We recommend all users of v0.6.0 upgrade to this release.

Python 2.7 support will end on June 1, 2019. Releases made after this date will require Python 3. (GH501)

Minimum pandas requirement bumped 0.14.0=>0.15.0

API Changes

- Created the `pvlib.iotools` subpackage. (GH29, GH261)
- Deprecated `tmy`, `tmy.readtmy2` and `tmy.readtmy3`; they will be removed in v0.7. Use the new *pvlib.iotools.read_tmy2()* and *pvlib.iotools.read_tmy3()* instead. (GH261)
- Added keyword argument `horizon` to *pyephem()* and *calc_time()* with default value `'+0:00'`. (GH588)

- Add *max_airmass* keyword argument to `pvlib.irradiance.disc()`. Default value (*max_airmass=12*) is consistent with polynomial fit in original paper describing the model. This change may result in different output of functions that use the *disc Kn* calculation for times when input zenith angles approach 90 degrees. This includes `pvlib.irradiance.dirint()` and `pvlib.irradiance.dirindex()` when *min_cos_zenith* and *max_zenith* kwargs are used, as well as `pvlib.irradiance.gti_dirint()`. (GH450)
- Changed key names for *components* returned from `pvlib.clearsky.detect_clearsky()`. (GH596)
- Changed function name from *pvlib.solarposition.get_rise_set_transit* (deprecated) to `pvlib.solarposition.sun_rise_set_transit_spa. `sun_rise_set_transit_spa()` requires time input to be localized to the specified latitude/longitude. (GH316)
- Created new bifacial section for *pvfactors* limited implementation (GH421)

Enhancements

- Add `sun_rise_set_transit_ephem`to calculate sunrise, sunset and transit times using pyephem (:issue:`114())`
- Add geometric functions for sunrise, sunset, and sun transit times, `sun_rise_set_transit_geometric()` (GH114)
- Add *Location* class method `get_sun_rise_set_transit()`
- Created `pvlib.iotools.read_srml()` and `pvlib.iotools.read_srml_month_from_solardat()` to read University of Oregon Solar Radiation Monitoring Laboratory data. (GH589)
- Created `pvlib.iotools.read_surfrad()` to read NOAA SURFRAD data. (GH590)
- Created `pvlib.iotools.read_midc()` and `pvlib.iotools.read_midc_raw_data_from_nrel()` to read NREL MIDC data. (GH601)
- Created `pvlib.iotools.get_ecmwf_macc()` and `pvlib.iotools.read_ecmwf_macc()` to get and read ECMWF MACC data. (GH602)
- Use HRRR modeled surface temperature values instead of inferring from isobaric values and modeled wind speed instead of inferring from gust. (GH604)
- Change `pvlib.pvsystem.sapm_spectral_loss()` to avoid numpy warning.
- Add warning message when `pvlib.spa()` is reloaded. (GH401)
- Add option for `pvlib.irradiance.disc()` to use relative airmass by supplying *pressure=None*. (GH449)
- Created `pvlib.pvsystem.pvsyst_celltemp()` to implement PVsyst's cell temperature model. (GH552)
- Created `pvlib.bifacial.pvfactors_timeseries()` to use open-source *pvfactors* package to calculate back surface irradiance (GH421)
- Add *PVSystem* class method `pvsyst_celltemp()` (GH633)
- Add `pvlib.irradiance.clearsky_index()` to calculate clear-sky index from measured GHI and modeled clear-sky GHI. (GH551)

Bug fixes

- Fix when building documentation using Matplotlib 3.0 or greater.

- ~*pvlib.spa.calculate_deltat*: Fix constant coefficient of the polynomial expression for years >= 1860 and < 1900, fix year 2050 which was returning 0. (GH600)
- Fix and improve *hour_angle()* (GH598)
- Fix error in *pvlib.clearsky.detect_clearsky()* (GH506)
- Fix documentation errors when using IPython >= 7.0.
- Fix error in *pvlib.modelchain.ModelChain.infer_spectral_model()* (GH619)
- Fix error in `pvlib.spa` when using Python 3.7 on some platforms.
- Fix error in `pvlib.irradiance._delta_kt_prime_dirint()` (GH637). The error affects the first and last values of DNI calculated by the function *pvlib.irradiance.dirint()*
- Fix errors on Python 2.7 and Numpy 1.6. (GH642)
- Replace deprecated *np.asscalar* with *array.item()*. (GH642)

Testing

- Add test for *hour_angle()* (GH597)
- Update tests to be compatible with pytest 4.0. (GH623)
- Add tests for *pvlib.bifacial.pvfactors_timeseries()* implementation (GH421)

Contributors

- Will Holmgren (@wholmgren)
- Leland Boeman (@lboeman)
- Cedric Leroy (@cedricleroy)
- Ben Ellis (@bhellis725)
- Cliff Hansen (@cwhanse)
- Mark Mikofski (@mikofski)
- Anton Driesse (@adriesse)
- Cameron Stark (@camerontstark)
- Jonathan Gaffiot (@jgaffiot)
- Marc Anoma (@anomam)
- Anton Driesse (@adriesse)
- Kevin Anderson (@kevinsa5)

3.4.8 v0.6.0 (September 17, 2018)

This is a major release and contains a large number of API changes, new features, and bug fixes. Users should carefully read the changelog below before upgrading.

Python 2.7 support will end on June 1, 2019. Releases made after this date will require Python 3. (GH501)

API Changes

- pvlib python is changing a handful of function names. In general, functions that can calculate a quantity using multiple algorithms now start with the prefix `get_`. For example, `relativeairmass` can calculate airmass using one of many `model` arguments. Its name has been changed to *`get_relative_airmass()`*. The old function names remain in this release, but will emit a `PVLibDeprecationWarning` when called. The old functions will be removed in the v0.7 release. Functions composed of multiple words jammed together have been renamed with underscores separating the words (see above). Each change is detailed below. (GH427)
- Deprecated `relativeairmass`; it will be removed in v0.7. Use the new *`get_relative_airmass()`* instead. (GH427)
- Deprecated `absoluteairmass`; it will be removed in v0.7. Use the new *`get_absolute_airmass()`* instead. (GH427)
- Deprecated `irradiance.globalinplane`; it will be removed in v0.7. Use the new *`poa_components()`* instead. (GH427)
- Added *`poa_components()`*. Function is the same as the now-deprecated `irradiance.globalinplane`, but adds `'poa_sky_diffuse'` and `'poa_ground_diffuse'` to the output. (GH427)
- Deprecated `irradiance.extraradiation`; it will be removed in v0.7. Use *`pvlib.irradiance.get_extra_radiation()`* instead. (GH427)
- Deprecated `irradiance.grounddiffuse`; it will be removed in v0.7. Use *`get_ground_diffuse()`* instead. (GH427)
- Added `get_poa_sky_diffuse()`. (GH427)
- Deprecated `irradiance.total_irrad`; it will be removed in v0.7. Use `get_total_poa_irradiance()` instead. (GH427)
- Removed `'klutcher'` from `get_sky_diffuse`/`total_irrad`. This misspelling was deprecated long ago but never removed. (GH97)
- *`calcparams_desoto()`* now requires arguments for each module model parameter. (GH462)
- Add losses_parameters attribute to PVSystem objects and remove the kwargs support from PVSystem.pvwatts_losses. Enables custom losses specification in ModelChain calculations. (GH484)
- removed irradiance parameter from ModelChain.run_model and ModelChain.prepare_inputs
- Add `perez_enhancement` keyword argument to clearsky.ineichen to control whether or not the "perez enhancement factor" is applied. The enhancement factor was always applied until now. Now it is turned off by default. The enhancement factor can yield unphysical results, especially for latitudes closer to the poles and especially in the winter months. It may yield improved results under other conditions. (GH435)
- Add min_cos_zenith, max_zenith keyword arguments to disc, dirint, and dirindex functions. (GH311, GH396)
- Method ModelChain.infer_dc_model now returns a tuple (function handle, model name string) instead of only the function handle (GH417)
- Add DC model methods desoto and pvsyst to ModelChain, and deprecates DC model method singlediode (singlediode defaults to desoto until v0.7.0) (GH487)
- Add the CEC module model in pvsystem.calcparams_cec and ModelChain.cec. The CEC model differs from the desoto model by using the parameter Adjust. Modules selected from the SAM CEC library sam-library-cec-modules-2017-6-5.csv include the Adjust parameter and ModelChain.infer_dc_model will now select the cec model rather than the desoto model. (GH463)

- The behavior of irradiance.perez(return_components=True) has changed. The function previously returned a tuple of total sky diffuse and an OrderedDict/DataFrame of components. The function now returns an OrderedDict/DataFrame with total sky diffuse and each component. The behavior for return_components=False remains unchanged. (GH434)

Enhancements

- Add sea surface albedo in `irradiance.py` (GH458)
- Implement *`first_solar_spectral_loss()`* in `modelchain.py` (GH359)
- Clarify arguments `Egref` and `dEgdT` for *`calcparams_desoto()`* (GH462)
- Add pvsystem.calcparams_pvsyst to compute values for the single diode equation using the PVsyst v6 model (GH470)
- Extend *`singlediode()`* with an additional keyword argument `method` in `('lambertw', 'newton', 'brentq')`, default is `'lambertw'`, to select a method to solve the single diode equation for points on the IV curve. Selecting either `'brentq'` or `'newton'` as the method uses *`bishop88()`* with the corresponding method. (GH410)
- Implement new methods `'brentq'` and `'newton'` for solving the single diode equation for points on the IV curve. `'brentq'` uses a bisection method (Brent, 1973) that may be slow but guarantees a solution. `'newton'` uses the Newton-Raphson method and may be faster but is not guaranteed to converge. However, `'newton'` should be safe for well-behaved IV curves. (GH408)
- Implement *`bishop88()`* for explicit calculation of arbitrary IV curve points using diode voltage instead of cell voltage. If `method` is either `'newton'` or `'brentq'` and `ivcurve_pnts` in *`singlediode()`* is provided, the IV curve points will be log spaced instead of linear.
- Implement *`estimate_voc()`* to estimate open circuit voltage by assuming $R_{sh} \to \infty$ and $R_s = 0$ as an upper bound in bisection method for *`singlediode()`* when method is either `'newton'` or `'brentq'`.
- Add *`max_power_point()`* method to compute the max power point using the new `'brentq'` method.
- Add new module `pvlib.singlediode` with low-level functions for solving the single diode equation such as: *`bishop88()`*, *`estimate_voc()`*, *`bishop88_i_from_v()`*, *`bishop88_v_from_i()`*, and *`bishop88_mpp()`*.
- Add PVSyst thin-film recombination losses for CdTe and a:Si (GH163)
- Python 3.7 officially supported. (GH496)
- Improve performance of solarposition.ephemeris. (GH512)
- Improve performance of Location.get_airmass. Most noticeable when solar position is supplied, time index length is less than 10000, and method is looped over. (GH502)
- Add irradiance.gti_dirint function. (GH396)
- Add irradiance.clearness_index function. (GH396)
- Add irradiance.clearness_index_zenith_independent function. (GH396)
- Add checking for consistency between module_parameters and dc_model. (GH417)
- Add DC model methods `'desoto'` and `'pvsyst'` to ModelChain (GH487)
- Add the CEC module model in *pvsystem.calcparams_cec* and *ModelChain.cec*. (GH463)
- Add DC model methods desoto and pvsyst to ModelChain (GH487)

- pvlib now ships with a pvlib[optional] installation option to automatically install packages needed to support additional pvlib features: `pip install pvlib[optional]`. Additional installation options include *doc* (requirements for minimal documentation build), *test* (requirements for testing), and *all* (optional + doc + test). (GH553, GH483)
- Set default alpha to 1.14 in *`angstrom_aod_at_lambda()`* (GH563)
- tracking.singleaxis now accepts scalar and 1D-array input.

Bug fixes

- Unset executable bits of irradiance.py and test_irradiance.py (GH460)
- Fix failing tests due to column order on Python 3.6+ and Pandas 0.23+ (GH464)
- ModelChain.prepare_inputs failed to pass solar_position and airmass to Location.get_clearsky. Fixed. (GH481)
- Add User-Agent specification to TMY3 remote requests to avoid rejection. (GH493)
- Fix `pvlib.irradiance.klucher` output is different for Pandas Series vs. floats and NumPy arrays. (GH508)
- Make GitHub recognize the license, add AUTHORS.md, clarify shared copyright. (GH503)
- Fix issue with non-zero direct irradiance contribution to Reindl, Klucher, and Hay-Davies diffuse sky algorithms when the sun is behind the array. (GH526)
- Fix issue with dividing by near-0 cos(solar_zenith) values in Reindl and Hay-Davies diffuse sky algorithms. (GH432)
- Fix argument order of longitude and latitude when querying weather forecasts by lonlat bounding box (GH521)
- Fix issue with unbounded clearness index calculation in disc. (GH540)
- Limit pvwatts_ac results to be greater than or equal to 0. (GH541)
- Fix bug in get_relative_airmass(model='youngirvine1967'). (GH545)
- Fix bug in variable names returned by forecast.py's HRRR_ESRL model. (GH557)
- Fixed bug in tracking.singleaxis that mistakenly assigned nan values when the Sun was still above the horizon. No effect on systems with axis_tilt=0. (GH569)
- Source distribution did not contain LICENSE file. Added LICENSE, AUTHORS.md, and some docs to MANIFEST. (GH579)
- Patch SPA C-files to fix timezone macro name clash with *pyconfig.h*. (GH168)

Documentation

- Expand testing section with guidelines for functions, PVSystem/Location objects, and ModelChain.
- Updated several incorrect statements in ModelChain documentation regarding implementation status and default values. (GH480)
- Expanded general contributing and pull request guidelines.
- Added section on single diode equation with some detail on solutions used in pvlib-python (GH518)
- Minor improvements and updates to installation documentation. (GH531)
- Improve LocalizedPVSystem and LocalizedSingleAxisTracker documentation. (GH532)
- Move the "Getting Started"/"Modeling Paradigms" section to a new top-level "Intro Examples" page.

- Copy pvlib documentation's "Getting support" section to README.md.
- Add PVSystem documentation page. (GH514, GH319)
- Add example of Kasten Linke Turbidity calculation, discuss broadband AOD and Angstrom Turbidity Model. (GH302)
- Add JOSS paper to "Citing pvlib-python" section.

Testing

- Add pytest-mock dependency
- Use pytest-mock to ensure that PVSystem and ModelChain methods call corresponding functions correctly. Removes implicit dependence on precise return values of some function/methods. (GH394)
- Additional test refactoring to limit test result dependence to a single function per test. (GH394)
- Use pytest-mock to ensure that ModelChain DC model is set up correctly.
- Add Python 3.7 to build matrix
- Make test_forecast.py more robust. (GH293)
- Improve test_atmosphere.py. (GH158)
- Add LGTM.com integration. (GH554)
- Add SticklerCI integration.
- Add codecov integration.

Contributors

- Will Holmgren (@wholmgren)
- Yu Cao (@tsaoyu)
- Cliff Hansen (@cwhanse)
- Mark Mikofski (@mikofski)
- Alan Mathew (@alamathe1)
- Xavier Rene-Corail (@xcorail)
- Anton Driesse (@adriesse)
- Mark Campanelli (@thunderfish24)
- Cedric Leroy (@cedricleroy)
- Jessica Forbess (@jforbess)
- Jeff Newmiller (@jdnewmil)
- @josricha
- Marc A. Anoma (@anomam)
- William C Grisaitis (@grisaitis)
- Karel De Brabandere (@kdebrab)
- @tadatoshi

3.4.9 v0.5.2 (May 13, 2018)

API Changes

- removed unused 'pressure' arg from irradiance.liujordan function (GH386)
- replaced logging.warning calls with warnings.warn calls, and removed logging.debug calls. We encourage users to explore tools such as pdb and trackback in place of the logging.debug calls. Fixes (GH447).

Enhancements

- Improve clearsky.lookup_linke_turbidity speed, changing .mat source file to .h5 (GH437)
- Updated libraries for CEC module parameters to SAM release 2017.9.5 (library dated 2017.6.30) and CEC inverter parameters to file posted to www.github.com/NREL/SAM on 2018.3.18, with one entry removed due to a missing parameter value. (:issue:'440')

Bug fixes

- fixed redeclaration of test_simplified_solis_series_elevation (GH387)
- physicaliam now returns a Series if called with a Series as an argument. (GH397)
- corrected docstring for irradiance.total_irrad (:issue: '423')
- modified solar_azimuth_analytical to handle some borderline cases, thereby avoiding the NaN values and/or warnings that were previously produced (:issue: '420')
- removed RuntimeWarnings due to divide by 0 or nans in input data within irradiance.perez, clearsky.simplified_solis, pvsystem.adrinverter, pvsystem.ashraeiam, pvsystem.physicaliam, pvsystem.sapm_aoi_loss, pvsystem.v_from_i. (GH428)

Documentation

- Improve physicaliam doc string. (GH397)

Testing

- Test Python 3.6 on Windows with Appveyor instead of 3.4. (GH392)
- Fix failing test on pandas 0.22 (GH406)
- Fix too large test tolerance (GH414)

Contributors

- Cliff Hansen
- Will Holmgren
- KonstantinTr
- Anton Driesse
- Cedric Leroy

3.4.10 v0.5.1 (October 17, 2017)

API Changes

- *pvsystem.v_from_i* and *pvsystem.i_from_v* functions now accept resistance_series = 0 and/or resistance_shunt = numpy.inf as inputs (GH340)

Enhancements

- Improve clearsky.lookup_linke_turbidity speed. (GH368)
- Ideal devices supported in single diode model, e.g., resistance_series = 0 and/or resistance_shunt = numpy.inf (GH340)
- *pvsystem.v_from_i* and *pvsystem.i_from_v* computations for near ideal devices are more numerically stable. However, very, very near ideal resistance_series and/or resistance_shunt may still cause issues with the implicit solver (GH340)

Bug fixes

- Remove condition causing Overflow warning from clearsky.haurwitz (GH363)
- modelchain.basic_chain now correctly passes 'solar_position_method' arg to solarposition.get_solarposition (GH370)
- Fixed: Variables and Symbols extra references not available (GH380)
- Removed unnecessary calculations of alpha_prime in spa.solar_position_numpy and spa.solar_position_loop (GH366)
- Fixed args mismatch for solarposition.pyephem call from solarposition.get_solarposition with method='pyephem' arg to solarposition.get_solarposition (GH370)
- ModelChain.prepare_inputs and ModelChain.complete_irradiance now correctly pass the 'solar_position_method' argument to solarposition.get_solarposition (GH377)
- Fixed usage of inplace parameter for tmy._recolumn (GH342)

Documentation

- Doc string of modelchain.basic_chain was updated to describe args more accurately. (GH370)
- Doc strings of *singlediode*, *pvsystem.v_from_i*, and *pvsystem.i_from_v* were updated to describe acceptable input arg ranges. (GH340)

Testing

- Changed test for clearsky.haurwitz to operate on zenith angles
- Significant new test cases added for *pvsystem.v_from_i* and *pvsystem.i_from_v* (GH340)

Contributors

- Cliff Hansen
- KonstantinTr
- Will Holmgren
- Mark Campanelli
- DaCoEx

3.4.11 v0.5.0 (August 11, 2017)

API Changes

- Removed parameter w from _calc_d (GH344)
- SingleAxisTracker.get_aoi and SingleAxisTracker.get_irradiance now require surface_zenith and surface_azimuth (GH351)
- Changes calculation of the Incidence Angle Modifier to return 0 instead of np.nan for angles >= 90°. This improves the calculation of effective irradiance close to sunrise and sunset. (GH338)
- Change the default ModelChain orientation strategy from 'south_at_latitude_tilt' to `None`. (GH290)

Bug fixes

- Method of multi-inheritance has changed to make it possible to use kwargs in the parent classes of LocalizedPVSystem and LocalizedSingleAxisTracker (GH330)
- Fix the *__repr__* method of *ModelChain*, crashing when *orientation_strategy* is set to *'None'* (GH352)
- Fix the *ModelChain*'s angle of incidence calculation for SingleAxisTracker objects (GH351)
- Fix issue with ForecastModel.cloud_cover_to_transmittance_linear method of forecast.py ignoring 'offset' parameter. (GH343)

Enhancements

- Added default values to docstrings of all functions (GH336)
- Added analytical method that calculates solar azimuth angle (GH291)

Documentation

- Added ModelChain documentation page
- Added nbsphinx to documentation build configuration.
- Added a pull request template file (GH354)

Testing

- Added explicit tests for aoi and aoi_projection functions.
- Update test of *ModelChain.__repr__* to take in account GH352
- Added a test for solar_azimuth_analytical function.

Contributors

- Johannes Kaufmann
- Will Holmgren
- Uwe Krien
- Alaina Kafkes
- Birgit Schachler
- Jonathan Gaffiot
- Siyan (Veronica) Guo
- KonstantinTr

3.4.12 v0.4.5 (June 5, 2017)

Bug fixes

- Fix pandas 0.20 incompatibilities in Location.get_clearsky, solarposition.ephemeris (GH325)
- Fixes timezone issue in solarposition spa_c function (GH237)
- Added NREL Bird clear sky model. (GH276)
- Added lower accuracy formulas for equation of time, declination, hour angle and solar zenith.
- Remove all instances of .ix (GH322)
- Update docstring in *pvlib.spa.solar_position* - change units of pressure to millibars. *NOTE*: units of pressure in *pvlib.solar_position.spa_python* and *pvlib.solar_position.spa_c* are still Pascals. This update should have no effect on most users, since it only applies to the low-level *spa.py* module. (GH327)

Enhancements

- Added irradiance.dni method that determines DNI from GHI and DHI and corrects unreasonable DNI values during sunrise/sunset transitions
- ForecastModel will now only connect to the Unidata server when necessary, rather than when the object is created. This supports offline work and speeds up analysis of previously downloaded data.

Contributors

- Will Holmgren
- Marc Anoma
- Mark Mikofski

- Birgit Schachler

3.4.13 v0.4.4 (February 18, 2017)

Enhancements

- Added Anton Driesse Inverter database and made compatible with pvsystem.retrieve_sam. (GH169)
- Ported Anton Driesse Inverter model from PV_LIB Toolbox. (GH160)
- Added Kasten pyrheliometric formula to calculate Linke turbidity factors with improvements by Ineichen and Perez to extend range of air mass (GH278)
- Added coefficients for CIGS and a-Si modules types to the first_solar_spectral_correction function (GH308)

API Changes

- Change PVSystem default module_parameters and inverter_parameters to empty dict. Code that relied on these attributes being None or raising a TypeError will need to be updated. (issue:*294*)

Documentation

- Fixes the Forecasting page's broken links to the tutorials.
- Fixes the Forecasting page's broken examples. (GH299)
- Fixes broken Classes link in the v0.3.0 documentation.

Bug fixes

- Resolved several issues with the forecast module tests. Library import errors were resolved by prioritizing the conda-forge channel over the default channel. Stalled ci runs were resolved by adding a timeout to the HRRR_ESRL test. (GH293)
- Fixed issue with irradiance jupyter notebook tutorial. (GH309)

Contributors

- Will Holmgren
- Volker Beutner
- Mark Mikofski
- Anton Driesse
- Mitchell Lee

3.4.14 v0.4.3 (December 28, 2016)

Enhancements

- Adding implementation of Perez's DIRINDEX model based on existing DIRINT model implementation. (GH282)

- Added clearsky.detect_clearsky function to determine the clear times in a GHI time series. (GH284)

Other

- Adds Python 3.6 to compatibility statement and pypi classifiers. (GH286)

Contributors

- Marc Anoma
- Will Holmgren
- Cliff Hansen
- Tony Lorenzo

3.4.15 v0.4.2 (December 7, 2016)

This is a minor release from 0.4.1.

Bug fixes

- Fixed typo in __repr__ method of ModelChain and in its regarding test.
- PVSystem.pvwatts_ac could not use the eta_inv_ref kwarg and PVSystem.pvwatts_dc could not use the temp_ref kwarg. Fixed. (GH252)
- Fixed typo in ModelChain.infer_spectral_model error message. (GH251)
- Fixed Linke turbdity factor out of bounds error at 90-degree latitude or at 180-degree longitude (GH262)
- Fixed Linke turbidity factor grid spacing and centers (GH263)

API Changes

- The run_model method of the ModelChain will use the weather parameter of all weather data instead of splitting it to irradiation and weather. The irradiation parameter still works but will be removed soon. (GH239)
- delta_t kwarg is now 67.0 instead of None. IMPORTANT: Setting delta_t as None will break the code for the Numba accelerated calculations. This will be fixed in a future version. (GH165)

Enhancements

- Adding a complete_irradiance method to the ModelChain to make it possible to calculate missing irradiation data from the existing columns [beta]. (GH239)
- Added calculate_deltat method to the spa module to calculate the time difference between terrestrial time and UT1. Specifying a scalar is sufficient for most calculations. (GH165)
- Added more attributes to ModelChain, PVSystem, and Location printed representations. (GH254)
- Added name attribute to ModelChain and PVSystem. (GH254)
- Restructured API section of the documentation so that there are separate pages for each function, class, or method. (GH258)

- Improved Linke turbidity factor time interpolation with Python *calendar* month days and leap years (GH265)
- Added option to return diffuse components from Perez transposition model.

Other

- Typical modeling results could change by ~1%, depending on location, if they depend on the turbidity table
- Fixed issues with pvsystem, tracking, and tmy_to_power jupyter notebooks (GH267, GH273)

Code Contributors

- Uwe Krien
- Will Holmgren
- Volker Beutner
- Mark Mikofski
- Marc Anoma
- Giuseppe Peronato

3.4.16 v0.4.1 (October 5, 2016)

This is a minor release from 0.4.0. We recommend that all users upgrade to this version, especially if they want to use the latest versions of pandas.

Bug fixes

- Fixed an error in the irradiance.klucher transposition model. The error was introduced in version 0.4.0. (GH228)
- Update RAP forecast model variable names. (GH241)
- Fix incompatibility with pandas 0.19 and solar position calculations. (GH246)

Documentation

- Fixed a typo in the pvsystem.sapm returns description. (GH234)
- Replaced nosetests references with py.test. (GH232)
- Improve the rendering of the snlinverter doc string. (GH242)

Code Contributors

- Mark Mikofski
- Johannes Dollinger
- Will Holmgren

3.4.17 v0.4.0 (July 28, 2016)

This is a major release from 0.3.3. We recommend that all users upgrade to this version after reviewing the API Changes. Please see the Bug Fixes for changes that will result in slightly different modeling results.

API Changes

- Remove unneeded module argument from singlediode function. (GH200)
- In `pvlib.irradiance.perez`, renamed argument `modelt` to `model`. (GH196)
- Functions in the irradiance module now work with scalar inputs in addition to arrays and Series. Furthermore, these functions no longer promote scalar or array input to Series output. Also applies to atmosphere.relativeairmass. (GH201, GH214)
- Reorder the ashraeiam, physicaliam, and snlinverter arguments to put the most variable arguments first. Adds default arguments for the IAM functions. (GH197)
- The irradiance.extraradiation function input/output type consistency across different methods has been dramatically improved. (GH217, GH219)
- Updated to pvsystem.sapm to be consistent with the PVLIB MATLAB API. pvsystem.sapm now takes an effective irradiance argument instead of POA irradiances, airmass, and AOI. Implements closely related sapm_spectral_loss, sapm_aoi_loss, and sapm_effective_irradiance functions, as well as PVSystem methods. The sapm_aoi_loss function includes an optional argument to apply an upper limit to the output (output can be ~1% larger than 1 for AOI of ~30 degrees). (GH198, GH205, GH218)
- The pvsystem.retrieve_sam keyword argument `samfile` has been replaced with `path`. A selection dialog window is now activated by not supplying any arguments to the function. The API for typical usage remains unchanged, however, the data will be loaded from a local file rather than the SAM website. (GH52)

Enhancements

- Adds the First Solar spectral correction model. (GH115)
- Adds the Gueymard 1994 integrated precipitable water model. (GH115)
- Adds the PVWatts DC, AC, and system losses model. (GH195)
- Improve PEP8 conformity in irradiance module. (GH214)
- irradiance.disc is up to 10x faster. (GH214)
- Add solarposition.nrel_earthsun_distance function and option to calculate extraterrestrial radiation using the NREL solar position algorithm. (GH211, GH215)
- pvsystem.singlediode can now calculate IV curves if a user supplies an ivcurve_pnts keyword argument. (GH83)
- Includes SAM data files in the distribution. (GH52)
- ModelChain now implements SAPM, PVWatts, Single Diode and user-defined modeling options. See Will Holmgren's ModelChain refactor gist for more discussion about new features in ModelChain. (GH143, GH194)
- Added `forecast.py` module for solar power forecasts. (GH86, GH124, GH180)

Bug fixes

- Fixed an error in pvsystem.singlediode's i_mp, v_mp, and p_mp calculations when using array or Series input. The function wrongly returned solutions when any single point is within the error tolerance, rather than requiring that the solution for all points be within the error tolerance. Results in test scenarios changed by 1-10%. (GH221)
- Fixed a numerical overflow error in pvsystem.singlediode's v_oc determination for some combinations of parameters. (GH225)
- dirint function yielded the wrong results for non-sea-level pressures. Fixed. (GH212)
- Fixed a bug in the day angle calculation used by the 'spencer' and 'asce' extraterrestrial radiation options. Most modeling results will be changed by less than 1 part in 1000. (GH211)
- irradiance.extraradiation now raises a ValueError for invalid method input. It previously failed silently. (GH215)

Documentation

- Added new terms to the variables documentation. (GH195)
- Added clear sky documentation page.
- Fix documentation build warnings. (GH210)
- Removed an unneeded note in irradiance.extraradiation. (GH216)

Other

- pvlib-python is now available on the conda-forge channel: conda install pvlib-python -c conda-forge (GH154)
- Switch to the py.test testing framework. (GH204)
- Reconfigure Appveyor CI builds and resolve an issue in which the command line length was too long. (GH207)
- Manually build numpy and pandas for the min requirements test. This is needed to avoid Continuum's bad practice of bundling scipy with pandas. (GH214)

Requirements

- pvlib now requires pandas >= 0.14.0 and numpy >= 1.9.0, both released in 2014. Most of pvlib will work with lesser versions. (GH214)

Code Contributors

- Will Holmgren
- Jonathan Chambers
- Mitchell Lee
- Derek Groenendyk

3.4.18 v0.3.3 (June 15, 2016)

This is a minor release from 0.3.2. We recommend that all users upgrade to this version.

API Changes

- Renamed `series_modules` to `modules_per_string` and `parallel_modules` to `strings_per_inverter`. (GH176)
- Changed two of the TMY3 data reader fields for consistency with the rest of the fields and with PVLIB MATLAB. Changed 'PresWth source' to 'PresWthSource', and 'PresWth uncert' to 'PresWthUncertainty'. (GH193)

Enhancements

- Adds the Erbs model. (GH2)
- Adds the `scale_voltage_current_power` function and `PVSystem` method to support simple array modeling. (GH159)
- Adds support for `SingleAxisTracker` objects in `ModelChain`. (GH169)
- Add `__repr__` method to PVSystem, LocalizedPVSystem, ModelChain, SingleAxisTracker, Location. (GH142)
- Add `v_from_i` function for solving the single diode model. (GH190)
- Improve speed of `singlediode` function by using `v_from_i` to determine `v_oc`. Speed is ~2x faster. (GH190)
- Adds the Simplified Solis clear sky model. (GH148)

Bug fixes

- Fix another bug with the Appveyor continuous integration builds. (GH170)
- Add classifiers to setup.py. (GH181)
- Fix snlinverter and singlediode documentation. They incorrectly said that inverter/module must be a DataFrame, when in reality they can be any dict-like object. (GH157)
- Fix numpy 1.11 deprecation warnings caused by some functions using non-integer indices.
- Propagate airmass data through ModelChain's get_irradiance call so that the perez model can use it, if necessary. (GH172)
- Fix problem in which the perez function dropped nighttime values. Nighttime values are now set to 0. (GH191)

Documentation

- Localize datetime indices in package overview examples. (GH156)
- Clarify that `ModelChain` and `basic_chain` currently only supports SAPM. (GH177)
- Fix version number in 0.3.2 whatsnew file.
- Shorten README.md file and move information to official documentation. (GH182)
- Change authors to *PVLIB Python Developers* and clean up setup.py. (GH184)
- Document the PresWth, PresWthSource, and PresWthUncertainty fields in the TMY3 data reader. (GH193)

Other

- Removed test skip decorator functions for Linux + Python 3 and for pandas 0.18.0. (GH187)

Contributors

- Will Holmgren
- Mark Mikofski
- Johannes Oos
- Tony Lorenzo

3.4.19 v0.3.2 (May 3, 2016)

This is a minor release from 0.3.1. We recommend that all users upgrade to this version.

Bug fixes

- Updates the SAM file URL. (GH152)

Contributors

- Will Holmgren

3.4.20 v0.3.1 (April 19, 2016)

This is a minor release from 0.3.0. We recommend that all users upgrade to this version.

Enhancements

- Added versioneer to keep track of version changes instead of manually updating pvlib/version.py. This will aid developers because the version string includes the specific git commit of the library code currently imported. (issue:*150*)

Bug fixes

- Fixes night tare issue in snlinverter. When the DC input power (p_dc) to an inverter is below the inversion startup power (Ps0), the model should set the AC output (ac_power) to the night tare value (Pnt). The night tare power indicates the power consumed by the inverter to sense PV array voltage. The model was erroneously comparing Ps0 with the AC output power (ac_power), rather than the DC input power (p_dc). (GH140)
- Fixed the azimuth calculation of rotated PV panel in function pvlib.tracking.singleaxis(...) so that the results are consistent with PVsyst. (GH144)

Contributors

- ejmiller2
- Yudong Ma
- Tony Lorenzo
- Will Holmgren

3.4.21 v0.3.0 (March 21, 2016)

This is a major release from 0.2.2. It will almost certainly break your code, but it's worth it! We recommend that all users upgrade to this version after testing their code for compatibility and updating as necessary.

API changes

- The `location` argument in `solarposition.get_solarposition` and `clearsky.ineichen` has been replaced with `latitude`, `longitude`, `altitude`, and `tz` as appropriate. This separates the object-oriented API from the procedural API. (GH17)
- `Location` classes gain the `get_solarposition`, `get_clearsky`, and `get_airmass` functions.
- Adds `ModelChain`, `PVSystem`, `LocalizedPVSystem`, `SingleAxisTracker`, and `LocalizedSingleAxisTracker` classes. (GH17)
- `Location` objects can be created from TMY2/TMY3 metadata using the `from_tmy` constructor.
- Change default `Location` timezone to `'UTC'`.
- The solar position calculators now assume UTC time if the input time is not localized. The calculators previously tried to infer the timezone from the now defunct location argument.
- `pvsystem.sapm_celltemp` argument names now follow the variable conventions.
- `irradiance.total_irrad` now follows the variable conventions. (GH105)
- `atmosphere.relativeairmass` now raises a ValueError instead of assuming `'kastenyoung1989'` if an invalid model is supplied. (GH119)

Enhancements

- Added new sections to the documentation:
  - *Package Overview* (GH93)
  - *Installation* (GH135)
  - *Contributing* (GH46)
  - *Time and time zones* (GH47)
  - *Variables and Symbols* (GH102)
  - Classes (GH93) (Moved to API reference in GH258)
- Adds support for Appveyor, a Windows continuous integration service. (GH111)
- The readthedocs documentation build now uses conda packages instead of mock packages. This enables code to be run and figures to be generated during the documentation builds. (GH104)
- Reconfigures TravisCI builds and adds e.g. `has_numba` decorators to the test suite. The result is that the TravisCI test suite runs almost 10x faster and users do not have to install all optional dependencies to run the test suite. (GH109)
- Adds more unit tests that test that the return values are actually correct.
- Add `atmosphere.APPARENT_ZENITH_MODELS` and `atmosphere.TRUE_ZENITH_MODELS` to enable code that can automatically determine which type of zenith data to use e.g. `Location.get_airmass`.
- Modify `sapm` documentation to clarify that it does not work with the CEC database. (GH122)
- Adds citation information to the documentation. (GH73)

- Updates the *Comparison with PVLIB MATLAB* documentation. (GH116)

Bug fixes

- Fixed the metadata key specification in documentation of the `readtmy2` function.
- Fixes the import of tkinter on Python 3 (GH112)
- Add a decorator to skip `test_calcparams_desoto` on pandas 0.18.0. (GH130)
- Fixes `i_from_v` documentation. (GH126)
- Fixes two minor sphinx documentation errors: a too short heading underline in whatsnew/v0.2.2.txt and a table format in pvsystem. (GH123)

Contributors

- Will Holmgren
- pyElena21
- DaCoEx
- Uwe Krien

Will Holmgren, Jessica Forbess, bmu, Cliff Hansen, Tony Lorenzo, Uwe Krien, and bt- contributed to the object model discussion.

3.4.22 v0.2.2 (November 13, 2015)

This is a minor release from 0.2.1. We recommend that all users upgrade to this version.

Enhancements

- Adds Python 3.5 compatibility (GH87)
- Moves the Linke turbidity lookup into `clearsky.lookup_linke_turbidity`. The API for `clearsky.ineichen` remains the same. (GH95)

Bug fixes

- `irradiance.total_irrad` had a typo that required the Klucher model to be accessed with `'klutcher'`. Both spellings will work for the remaining 0.2.* versions of pvlib, but the misspelled method will be removed in 0.3. (GH97)
- Fixes an import and KeyError in the IPython notebook tutorials (GH94).
- Uses the `logging` module properly by replacing `format` calls with `args`. This results in a 5x speed increase for `tracking.singleaxis` (GH89).
- Adds a link to the 2015 PVSC paper (GH81)

Contributors

- Will Holmgren
- jetheurer
- dacoex

3.4.23 v0.2.1 (July 16, 2015)

This is a minor release from 0.2. It includes a large number of bug fixes for the IPython notebook tutorials. We recommend that all users upgrade to this version.

Enhancements

- Update component info from SAM (csvs dated 2015-6-30) (GH75)

Bug fixes

- Fix incorrect call to Perez irradiance function (GH76)
- Fix numerous bugs in the IPython notebook tutorials (GH30)

Contributors

- Will Holmgren
- Jessica Forbess

3.4.24 v0.2.0 (July 6, 2015)

This is a major release from 0.1 and includes a large number of API changes, several new features and enhancements along with a number of bug fixes. We recommend that all users upgrade to this version.

Due to the large number of API changes, you will probably need to update your code.

API changes

- Change variable names to conform with new Variables and style rules wiki. This impacts many function declarations and return values. Your existing code probably will not work! (GH37, GH54).
- Move `dirint` and `disc` algorithms from `clearsky.py` to `irradiance.py` (GH42)
- Mark some `pvsystem.py` methods as private (GH20)
- Make output of `pvsystem.sapm_celltemp` a DataFrame (GH54)

Enhancements

- Add conda installer
- PEP8 fixups to solarposition.py and spa.py (GH50)
- Add optional `projection_ratio` keyword argument to the `haydavies` calculator. Speeds calculations when irradiance changes but solar position remains the same (GH58)
- Improved installation instructions in README.

Bug fixes

- fix local build of the documentation (GH49, GH56)
- The release date of 0.1 was fixed in the documentation (see *v0.1.0 (April 20, 2015)*)
- fix casting of DateTimeIndex to int64 epoch timestamp on machines with 32 bit python int (GH63)
- fixed some docstrings with failing doctests (GH62)

Contributors

- Will Holmgren
- Rob Andrews
- bmu
- Tony Lorenzo

3.4.25 v0.1.0 (April 20, 2015)

This is the first official release of the pvlib-python project. As such, a "What's new" document is a little hard to write. There will be significant overlap with the to-be-written document that describes the differences between pvlib-python and PVLIB_Matlab.

API changes

- Remove `pvl_` from module names.
- Consolidation of similar modules. For example, functions from `pvl_clearsky_ineichen.py` and `pvl_clearsky_haurwitz.py` have been consolidated into `clearsky.py`.
- Return one DataFrame instead of a tuple of DataFrames.
- Change function and module names so that they do not conflict.

New features

- Library is Python 3.3 and 3.4 compatible
- Add What's New section to docs (GH10)
- Add PyEphem option to solar position calculations.
- Add a Python translation of NREL's SPA algorithm.

- `irradiance.py` has more AOI, projection, and irradiance sum and calculation functions
- TMY data import has a `coerce_year` option
- TMY data can be loaded from a url (GH5)
- Locations are now `pvlib.location.Location` objects, not "structs".
- Specify time zones using a string from the standard IANA Time Zone Database naming conventions or using a pytz.timezone instead of an integer GMT offset. We may add dateutils support in the future.
- `clearsky.ineichen` supports interpolating monthly Linke Turbidities to daily resolution.

Other changes

- Removed `Vars=Locals(); Expect...; var=pvl\_tools.Parse(Vars,Expect);` pattern. Very few tests of input valididity remain. Garbage in, garbage or `nan` out.
- Removing unnecssary and sometimes undesired behavior such as setting maximum zenith=90 or airmass=0. Instead, we make extensive use of `nan` values.
- Adding logging calls, removing print calls.
- Improved PEP8 compliance.
- Added `/pvlib/data` for lookup tables, test, and tutorial data.
- Limited the scope of `clearsky.py`'s `scipy` dependency. `clearsky.ineichen` will work without `scipy` so long as the Linke Turbidity is supplied as a keyword argument. (GH13)
- Removed NREL's SPA code to comply with their license (GH9).
- Revised the globalinplane function and added a test_globalinplane (GH21, GH33).

Documentation

- Using readthedocs for documentation hosting.
- Many typos and formatting errors corrected (GH16)
- Documentation source code and tutorials live in `/` rather than `/pvlib/docs`.
- Additional tutorials in `/docs/tutorials`.
- Clarify `pvsystem.systemdef` input (GH17)

Testing

- Tests are cleaner and more thorough. They are still nowhere near complete.
- Using Coveralls to measure test coverage.
- Using TravisCI for automated testing.
- Using `nosetests` for more concise test code.

Bug fixes

- Fixed DISC algorithm bugs concerning modifying input zenith Series (GH24), the `Kt` conditional evaluation (GH6), and ignoring the input pressure (GH25).
- Many more bug fixes were made, but you'll have to look at the detailed commit history.
- Fixed inconsistent azimuth angle in the ephemeris function (GH40)

Contributors

This list includes all (I hope) contributors to pvlib/pvlib-python, Sandia-Labs/PVLIB_Python, and UARENForecasting/PVLIB_Python.

- Rob Andrews
- Will Holmgren
- bmu
- Tony Lorenzo
- jforbess
- Jorissup
- dacoex
- alexisph
- Uwe Krien

3.5 Installation

Installing pvlib-python ranges from trivial to difficult depending on your python experience, how you want to use pvlib, and your system configuration.

Do you already have Python and the NumPy and Pandas libraries?

> If the answer to this is *No*, follow the *If you don't have Python* instructions to obtain the Anaconda Python distribution before proceeding.

Do you want to use the pvlib-python as-is, or do you want to be able to edit the source code?

> If you want to use pvlib-python *as-is*, follow the simple *Install standard release* instructions.
>
> If you want to be able to *edit the source code*, follow the *Install as an editable library* instructions.

Installing pvlib-python is similar to installing most scientific python packages, so see the *References* section for further help.

Please see the *Compatibility* section for information on the optional packages that are needed for some pvlib-python features.

3.5.1 If you don't have Python

There are many ways to install Python on your system, but the Anaconda Python distribution is the easiest way for most new users to get started. Anaconda includes all of the popular libraries that you'll need for pvlib, including Pandas, NumPy, and SciPy.

1. **Install** the Anaconda Python distribution available at Anaconda.com.

See What is Anaconda? and the Anaconda Documentation for more information.

You can now install pvlib-python by one of the methods below.

3.5.2 Install standard release

Users may install pvlib-python using either the conda or pip package manager. We recommend that most users install pvlib-python using the conda package manager in the Anaconda Python distribution. To install the most recent stable release of pvlib-python in a non-editable way, use one of the following commands to install pvlib-python:

```
# get the package from the pvlib conda channel
# best option for installing pvlib in the base Anaconda distribution
conda install -c pvlib pvlib

# get the package from the conda-forge conda channel
# best option if using pvlib.forecast module
# strongly recommend installing in a separate conda env as shown below
conda create -n pvlib -c conda-forge pvlib-python; conda activate pvlib

# get the package from the Python Package Index
# best option if you know what you are doing
pip install pvlib

# get pvlib and optional dependencies from the Python Package Index
# another option if you know what you are doing
pip install pvlib[optional]
```

Note: By default, pvlib will not install some infrequently used dependencies. If you run into an error such as *ModuleNotFoundError: No module named 'netCDF4'* you can either install pvlib with all optional dependencies using *pip install pvlib[optional]*, or you can install pvlib from conda-forge *conda create -n pvlib -c conda-forge pvlib-python; conda activate pvlib*.

If your system complains that you don't have access privileges or asks for a password then you're probably trying to install pvlib into your system's Python distribution. This is usually a bad idea and you should follow the *If you don't have Python* instructions before installing pvlib.

You may still want to download the Python source code so that you can easily get all of the Jupyter Notebook tutorials. Either clone the git repository or go to the Releases page to download the zip file of the most recent release. You can also use the nbviewer website to choose a tutorial to experiment with. Go to our nbviewer tutorial page.

3.5.3 Install as an editable library

Installing pvlib-python as an editable library involves 3 steps:

1. *Obtain the source code*
2. *Set up a virtual environment*
3. *Install the source code*

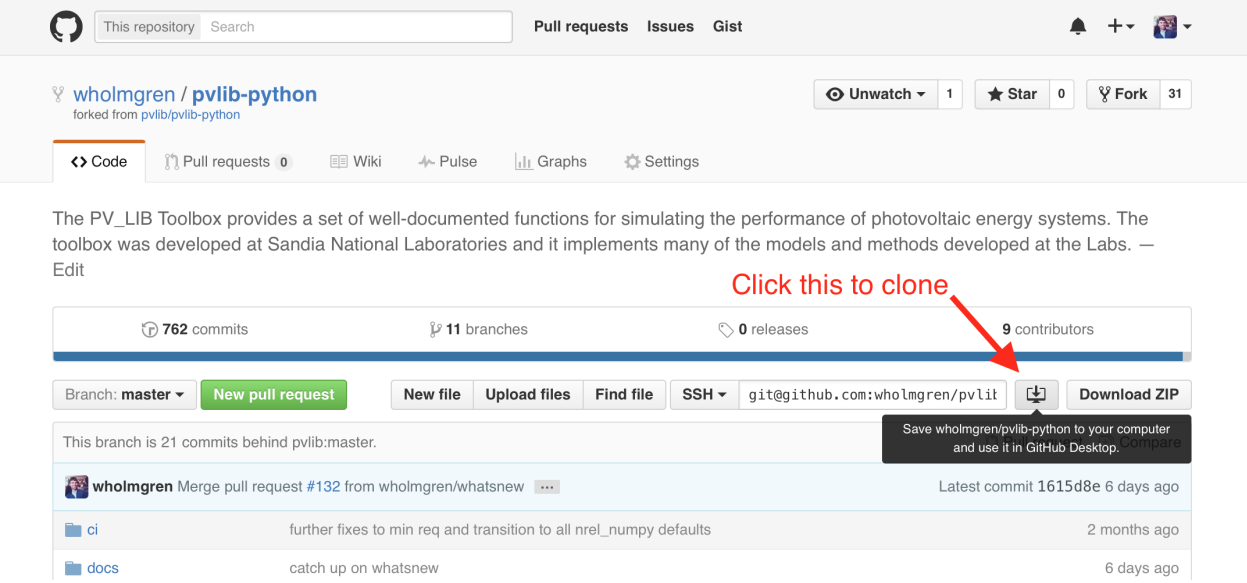
None of these steps are particularly challenging, but they become more difficult when combined. With a little bit of practice the process will be fast and easy. Experienced users can easily execute these steps in less than a minute. You'll get there.

Obtain the source code

We will briefly describe how to obtain the pvlib-python source code using the git/GitHub version control system. We strongly encourage users to learn how to use these powerful tools (see the *References*!), but we also recognize that they can be a substantial roadblock to getting started with pvlib-python. Therefore, you should know that you can download a zip file of the most recent development version of the source code by clicking on the **Download Zip** button on the right side of our GitHub page or download a zip file of any stable release from our Releases page.

Follow these steps to obtain the library using git/GitHub:

1. **Download** the GitHub Desktop application.
2. **Fork** the pvlib-python project by clicking on the "Fork" button on the upper right corner of the pvlib-python GitHub page.
3. **Clone** your fork to your computer using the GitHub Desktop application by clicking on the *Clone to Desktop* button on your fork's homepage. This button is circled in the image below. Remember the system path that you clone the library to.



Please see GitHub's Forking Projects, Fork A Repo, and the git-scm for more details.

Set up a virtual environment

We strongly recommend working in a virtual environment if you're going to use an editable version of the library. You can skip this step if:

1. You already have Anaconda or another scientific Python distribution
2. You don't mind polluting your Python installation with your development version of pvlib.
3. You don't want to work with multiple versions of pvlib.

There are many ways to use virtual environments in Python, but Anaconda again provides the easiest solution. These are often referred to as *conda environments*, but they're the same for our purposes.

1. **Create** a new conda environment for pvlib and pre-install the required packages into the environment: `conda create --name pvlibdev python pandas scipy`
2. **Activate** the new conda environment: `conda activate pvlibdev`

3. **Install** additional packages into your development environment: `conda install jupyter ipython matplotlib pytest nose flake8`

The conda documentation has more information on how to use conda virtual environments. You can also add `-h` to most pip and conda commands to get help (e.g. `conda -h` or `conda env -h`)

Install the source code

Good news – installing the source code is the easiest part! With your conda/virtual environment still active...

1. **Install** pvlib-python in "development mode" by running `pip install -e .` from within the directory you previously cloned. Consider installing pvlib using `pip install -e .[all]` so that you can run the unit tests and build the documentation. Your clone directory is probably similar to `C:\Users\%USER%\Documents\GitHub\pvlib-python``(Windows) or ``/Users/%USER%/Documents/pvlib-python` (Mac).
2. **Test** your installation by running `python -c 'import pvlib'`. You're good to go if it returns without an exception.

The version of pvlib-python that is on that path is now available as an installed package inside your conda/virtual environment.

Any changes that you make to this pvlib-python will be available inside your environment. If you run a git checkout, branch, or pull command the result will be applied to your pvlib-python installation. This is great for development. Note, however, that you will need to use Python's `reload` function (python 3) if you make changes to pvlib during an interactive Python session (including a Jupyter notebook). Restarting the Python interpreter will also work.

Remember to `conda activate pvlibdev` (or whatever you named your environment) when you start a new shell or terminal.

3.5.4 Compatibility

pvlib-python is compatible with Python 3.5 and above.

pvlib-python requires Pandas and Numpy. The minimum version requirements are specified in setup.py. They are typically releases from several years ago.

A handful of pvlib-python features require additional packages that must be installed separately using pip or conda. These packages/features include:

- scipy: single diode model, clear sky detection
- pytables (tables on PyPI): Linke turbidity look up for clear sky models
- numba: fastest solar position calculations
- pyephem: solar positions calculations using an astronomical library
- siphon: forecasting PV power using the pvlib.forecast module

The Anaconda distribution includes most of the above packages.

Alternatively, users may install all optional dependencies using

```
pip install pvlib[optional]
```

3.5.5 NREL SPA algorithm

pvlib-python is distributed with several validated, high-precision, and high-performance solar position calculators. We strongly recommend using the built-in solar position calculators.

pvlib-python also includes unsupported wrappers for the official NREL SPA algorithm. NREL's license does not allow redistribution of the source code, so you must jump through some hoops to use it with pvlib. You will need a C compiler to use this code.

To install the NREL SPA algorithm for use with pvlib:

1. Download the pvlib repository (as described in *Obtain the source code*)
2. Download the SPA files from NREL
3. Copy the SPA files into `pvlib-python/pvlib/spa_c_files`
4. From the `pvlib-python` directory, run `pip uninstall pvlib` followed by `pip install .`

3.5.6 References

Here are a few recommended references for installing Python packages:

- The Pandas installation page
- python4astronomers Modules, Packages, and all that
- Python Packaging User Guide
- Conda User Guide

Here are a few recommended references for git and GitHub:

- The git documentation: detailed explanations, videos, more links, and cheat sheets. Go here first!
- Forking Projects
- Fork A Repo
- Cloning a repository
- Aha! Moments When Learning Git

3.6 Contributing

Encouraging more people to help develop pvlib-python is essential to our success. Therefore, we want to make it easy and rewarding for you to contribute.

There is a lot of material in this section, aimed at a variety of contributors from novice to expert. Don't worry if you don't (yet) understand parts of it.

3.6.1 Easy ways to contribute

Here are a few ideas for how you can contribute, even if you are new to pvlib-python, git, or Python:

- Ask and answer pvlib questions on StackOverflow and participate in discussions in the pvlib-python google group.
- Make GitHub issues and contribute to the conversations about how to resolve them.

- Read issues and pull requests that other people created and contribute to the conversation about how to resolve them. Look for issues tagged with good first issue, easy, or help wanted.
- Improve the documentation and the unit tests.
- Improve the IPython/Jupyter Notebook tutorials or write new ones that demonstrate how to use pvlib-python in your area of expertise.
- If you have MATLAB experience, you can help us keep pvlib-python up to date with PVLIB_MATLAB or help us develop common unit tests. For more, see Issue #2 and Issue #3.
- Tell your friends and colleagues about pvlib-python.
- Add your project to our Projects and publications that use pvlib-python wiki.

3.6.2 How to contribute new code

The basics

Contributors to pvlib-python use GitHub's pull requests to add/modify its source code. The GitHub pull request process can be intimidating for new users, but you'll find that it becomes straightforward once you use it a few times. Please let us know if you get stuck at any point in the process. Here's an outline of the process:

1. Create a GitHub issue and get initial feedback from users and maintainers. If the issue is a bug report, please include the code needed to reproduce the problem.
2. Obtain the latest version of pvlib-python: Fork the pvlib-python project to your GitHub account, `git clone` your fork to your computer.
3. Make some or all of your changes/additions and `git commit` them to your local repository.
4. Share your changes with us via a pull request: `git push` your local changes to your GitHub fork, then go to GitHub make a pull request.

The Pandas project maintains an excellent contributing page that goes into detail on each of these steps. Also see GitHub's Set Up Git and Using Pull Requests.

We strongly recommend using virtual environments for development. Virtual environments make it trivial to switch between different versions of software. This astropy guide is a good reference for virtual environments. If this is your first pull request, don't worry about using a virtual environment.

You must include documentation and unit tests for any new or improved code. We can provide help and advice on this after you start the pull request. See the Testing section below.

Pull request scope

This section can be summed up as "less is more".

A pull request can quickly become unmanageable if too many lines are added or changed. "Too many" is hard to define, but as a rule of thumb, we encourage contributions that contain less than 50 lines of primary code. 50 lines of primary code will typically need at least 250 lines of documentation and testing. This is about the limit of what the maintainers can review on a regular basis.

A pull request can also quickly become unmanageable if it proposes changes to the API in order to implement another feature. Consider clearly and concisely documenting all proposed API changes before implementing any code. Modifying api.rst and/or the latest whatsnew file can help formalize this process.

Questions about related issues frequently come up in the process of addressing implementing code for a pull request. Please try to avoid expanding the scope of your pull request (this also applies to reviewers!). We'd rather see small,

well-documented additions to the project's technical debt than see a pull request languish because its scope expanded beyond what the reviewer community is capable of processing.

Of course, sometimes it is necessary to make a large pull request. We only ask that you take a few minutes to consider how to break it into smaller chunks before proceeding.

pvlib-python contains *3 "layers" of code*: functions, PVSystem/Location, and ModelChain. We recommend that contributors focus their work on only one or two of those layers in a single pull request. New models are *not* required to be available to the higher-level API!

When should I submit a pull request?

The short answer: anytime.

The long answer: it depends. If in doubt, go ahead and submit. You do not need to make all of your changes before creating a pull request. Your pull requests will automatically be updated when you commit new changes and push them to GitHub.

There are pros and cons to submitting incomplete pull-requests. On the plus side, it gives everybody an easy way to comment on the code and can make the process more efficient. On the minus side, it's easy for an incomplete pull request to grow into a multi-month saga that leaves everyone unhappy. If you submit an incomplete pull request, please be very clear about what you would like feedback on and what we should ignore. Alternatives to incomplete pull requests include creating a gist or experimental branch and linking to it in the corresponding issue.

The best way to ensure that a pull request will be reviewed and merged in a timely manner is to:

1. Start by creating an issue. The issue should be well-defined and actionable.
2. Ask the maintainers to tag the issue with the appropriate milestone.
3. Make a limited-scope pull request. It can be a lot of work to check all of the boxes in pull request guidelines, especially for pull requests with a lot of new primary code. See *Pull request scope*.
4. Tag pvlib community members or `@pvlib/maintainer` when the pull request is ready for review. (see *Pull request reviews*)

Pull request reviews

The pvlib community and maintainers will review your pull request in a timely fashion. Please "ping" `@pvlib/maintainer` if it seems that your pull request has been forgotten at any point in the pull request process.

Keep in mind that the PV modeling community is diverse and each pvlib community member brings a different perspective when reviewing code. Some reviewers bring years of expertise in the sub-field that your code contributes to and will focus on the details of the algorithm. Other reviewers will be more focused on integrating your code with the rest of pvlib, ensuring that it is feasible to maintain, that it meets the *code style* guidelines, and that it is *comprehensively tested*. Limiting the scope of the pull request makes it much more likely that all of these reviews can be conducted and any issues can be resolved in a timely fashion.

Sometimes it's hard for reviewers to be immediately available, so the right amount of patience is to be expected. That said, interested reviewers should do their best to not wait until the last minute to put in their two cents.

3.6.3 Code style

pvlib python generally follows the PEP 8 – Style Guide for Python Code. Maximum line length for code is 79 characters.

Code must be compatible with Python 3.5 and above.

pvlib python uses a mix of full and abbreviated variable names. See *Variables and Symbols*. We could be better about consistency. Prefer full names for new contributions. This is especially important for the API. Abbreviations can be used within a function to improve the readability of formulae.

Set your editor to strip extra whitespace from line endings. This prevents the git commit history from becoming cluttered with whitespace changes.

Please see *Documentation* for information specific to documentation style.

Remove any `logging` calls and `print` statements that you added during development. `warning` is ok.

We typically use GitHub's "squash and merge" feature to merge your pull request into pvlib. GitHub will condense the commit history of your branch into a single commit when merging into pvlib-python/master (the commit history on your branch remains unchanged). Therefore, you are free to make commits that are as big or small as you'd like while developing your pull request.

3.6.4 Documentation

Documentation must be written in numpydoc format format which is rendered using the Sphinx Napoleon extension.

The numpydoc format includes a specification for the allowable input types. Python's duck typing allows for multiple input types to work for many parameters. pvlib uses the following generic descriptors as short-hand to indicate which specific types may be used:

- dict-like : dict, OrderedDict, pd.Series
- numeric : scalar, np.array, pd.Series. Typically int or float dtype.
- array-like : np.array, pd.Series. Typically int or float dtype.

Parameters that specify a specific type require that specific input type.

Read the Docs will automatically build the documentation for each pull request. Please confirm the documentation renders correctly by following the `continuous-documentation/read-the-docs` link within the checks status box at the bottom of the pull request.

To build the docs locally, install the `doc` dependencies specified in the setup.py file. See *Installation* instructions for more information.

Example Gallery

The example gallery uses sphinx-gallery and is generated from script files in the docs/examples directory. sphinx-gallery will execute example files that start with `plot_` and capture the output.

Here is a starter template for new examples:

```
"""
Page Title
==========

A sentence describing the example.
"""

# %%
# Explanatory text about the example, what it does, why it does it, etc.
# Text in the comment block before the first line of code `import pvlib`
# will be printed to the example's webpage.

import pvlib
```

(continues on next page)

(continued from previous page)

```
import matplotlib.pyplot as plt

plt.scatter([1, 2, 3], [4, 5, 6])
plt.show()
```

For more details, see the sphinx-gallery docs.

3.6.5 Testing

Developers **must** include comprehensive tests for any additions or modifications to pvlib. New unit test code should be placed in the corresponding test module in the pvlib/tests directory.

A pull request will automatically run the tests for you on a variety of platforms (Linux, Mac, Windows) and python versions. However, it is typically more efficient to run and debug the tests in your own local environment.

To run the tests locally, install the `test` dependencies specified in the setup.py file. See *Installation* instructions for more information.

pvlib's unit tests can easily be run by executing `pytest` on the pvlib directory:

```
pytest pvlib
```

or, for a single module:

```
pytest pvlib/test/test_clearsky.py
```

or, for a single test:

```
pytest pvlib/test/test_clearsky.py::test_ineichen_nans
```

We suggest using pytest's `--pdb` flag to debug test failures rather than using `print` or `logging` calls. For example:

```
pytest pvlib --pdb
```

will drop you into the pdb debugger at the location of a test failure. As described in *Code style*, pvlib code does not use `print` or `logging` calls, and this also applies to the test suite (with rare exceptions).

To include all network-dependent tests, include the `--remote-data` flag to your `pytest` call:

```
pytest pvlib --remote-data
```

And consider adding `@pytest.mark.remote_data` to any network dependent test you submit for a PR.

pvlib-python contains 3 "layers" of code: functions, PVSystem/Location, and ModelChain. Contributors will need to add tests that correspond to the layers that they modify.

Functions

Tests of core pvlib functions should ensure that the function returns the desired output for a variety of function inputs. The tests should be independent of other pvlib functions (see GH394). The tests should ensure that all reasonable combinations of input types (floats, nans, arrays, series, scalars, etc) work as expected. Remember that your use case is likely not the only way that this function will be used, and your input data may not be generic enough to fully test the function. Write tests that cover the full range of validity of the algorithm. It is also important to write tests that assert the return value of the function or that the function throws an exception when input data is beyond the range of algorithm validity.

PVSystem/Location

The PVSystem and Location classes provide convenience wrappers around the core pvlib functions. The tests in test_pvsystem.py and test_location.py should ensure that the method calls correctly wrap the function calls. Many PVSystem/Location methods pass one or more of their object's attributes (e.g. PVSystem.module_parameters, Location.latitude) to a function. Tests should ensure that attributes are passed correctly. These tests should also ensure that the method returns some reasonable data, though the precise values of the data should be covered by function-specific tests discussed above.

We prefer to use the `pytest-mock` framework to write these tests. The test below shows an example of testing the `PVSystem.ashraeiam` method. `mocker` is a `pytest-mock` object. `mocker.spy` adds features to the `pvsystem.ashraeiam` *function* that keep track of how it was called. Then a `PVSystem` object is created and the `PVSystem.ashraeiam` *method* is called in the usual way. The `PVSystem.ashraeiam` method is supposed to call the `pvsystem.ashraeiam` function with the angles supplied to the method call and the value of `b` that we defined in `module_parameters`. The `pvsystem.ashraeiam.assert_called_once_with` tests that this does, in fact, happen. Finally, we check that the output of the method call is reasonable.

```
def test_PVSystem_ashraeiam(mocker):
    # mocker is a pytest-mock object.
    # mocker.spy adds code to a function to keep track of how it is called
    mocker.spy(pvsystem, 'ashraeiam')

    # set up inputs
    module_parameters = {'b': 0.05}
    system = pvsystem.PVSystem(module_parameters=module_parameters)
    thetas = 1

    # call the method
    iam = system.ashraeiam(thetas)

    # did the method call the function as we expected?
    # mocker.spy added assert_called_once_with to the function
    pvsystem.ashraeiam.assert_called_once_with(thetas, b=module_parameters['b'])

    # check that the output is reasonable, but no need to duplicate
    # the rigorous tests of the function
    assert iam < 1.
```

Avoid writing PVSystem/Location tests that depend sensitively on the return value of a statement as a substitute for using mock. These tests are sensitive to changes in the functions, which is *not* what we want to test here, and are difficult to maintain.

ModelChain

The tests in test_modelchain.py should ensure that `ModelChain.__init__` correctly configures the ModelChain object to eventually run the selected models. A test should ensure that the appropriate method is actually called in the course of `ModelChain.run_model`. A test should ensure that the model selection does have a reasonable effect on the subsequent calculations, though the precise values of the data should be covered by the function tests discussed above. `pytest-mock` can also be used for testing `ModelChain`.

The example below shows how mock can be used to assert that the correct PVSystem method is called through `ModelChain.run_model`.

```
def test_modelchain_dc_model(mocker):
    # set up location and system for model chain
    location = location.Location(32, -111)
```

(continues on next page)

(continued from previous page)

```
    system = pvsystem.PVSystem(module_parameters=some_sandia_mod_params,
                               inverter_parameters=some_cecinverter_params)

    # mocker.spy adds code to the system.sapm method to keep track of how
    # it is called. use returned mock object m to make assertion later,
    # but see example above for alternative
    m = mocker.spy(system, 'sapm')

    # make and run the model chain
    mc = ModelChain(system, location,
                    aoi_model='no_loss', spectral_model='no_loss')
    times = pd.date_range('20160101 1200-0700', periods=2, freq='6H')
    mc.run_model(times)

    # assertion fails if PVSystem.sapm is not called once
    # if using returned m, prefer this over m.assert_called_once()
    # for compatibility with python < 3.6
    assert m.call_count == 1

    # ensure that dc attribute now exists and is correct type
    assert isinstance(mc.dc, (pd.Series, pd.DataFrame))
```

3.6.6 This documentation

If this documentation is unclear, help us improve it! Consider looking at the pandas documentation for inspiration.

3.6.7 Code of Conduct

All contributors are expected to adhere to the Contributor Code of Conduct.

3.7 PVSystem

The *PVSystem* class wraps many of the functions in the `pvsystem` module. This simplifies the API by eliminating the need for a user to specify arguments such as module and inverter properties when calling PVSystem methods. *PVSystem* is not better or worse than the functions it wraps – it is simply an alternative way of organizing your data and calculations.

This guide aims to build understanding of the PVSystem class. It assumes basic familiarity with object-oriented code in Python, but most information should be understandable without a solid understanding of classes. Keep in mind that *functions* are independent of objects, while *methods* are attached to objects.

See *ModelChain* for an application of PVSystem to time series modeling.

3.7.1 Design philosophy

The PVSystem class allows modelers to easily separate the data that represents a PV system (e.g. tilt angle or module parameters) from the data that influences the PV system (e.g. the weather).

The data that represents the PV system is *intrinsic*. The data that influences the PV system is *extrinsic*.

Intrinsic data is stored in object attributes. For example, the data that describes a PV system's module parameters is stored in *PVSystem.module_parameters*.

```
In [1]: module_parameters = {'pdc0': 10, 'gamma_pdc': -0.004}

In [2]: system = pvsystem.PVSystem(module_parameters=module_parameters)

In [3]: print(system.module_parameters)
{'pdc0': 10, 'gamma_pdc': -0.004}
```

Extrinsic data is passed to a PVSystem as method arguments. For example, the `pvwatts_dc()` method accepts extrinsic data irradiance and temperature.

```
In [4]: pdc = system.pvwatts_dc(1000, 30)

In [5]: print(pdc)
9.8
```

Methods attached to a PVSystem object wrap corresponding functions in `pvsystem`. The methods simplify the argument list by using data stored in the PVSystem attributes. Compare the `pvwatts_dc()` method signature to the *`pvwatts_dc()`* function signature:

- `PVSystem.pvwatts_dc(g_poa_effective, temp_cell)`
- *`pvwatts_dc(g_poa_effective, temp_cell, pdc0, gamma_pdc, temp_ref=25.)`*

How does this work? The `pvwatts_dc()` method looks in *PVSystem.module_parameters* for the *pdc0*, and *gamma_pdc* arguments. Then the `PVSystem.pvwatts_dc` method calls the *`pvsystem.pvwatts_dc`* function with all of the arguments and returns the result to the user. Note that the function includes a default value for the parameter *temp_ref*. This default value may be overridden by specifying the *temp_ref* key in the *PVSystem.module_parameters* dictionary.

```
In [6]: system.module_parameters['temp_ref'] = 0

# lower temp_ref should lead to lower DC power than calculated above
In [7]: pdc = system.pvwatts_dc(1000, 30)

In [8]: print(pdc)
8.8
```

Multiple methods may pull data from the same attribute. For example, the *PVSystem.module_parameters* attribute is used by the DC model methods as well as the incidence angle modifier methods.

3.7.2 PVSystem attributes

Here we review the most commonly used PVSystem attributes. Please see the *`PVSystem`* class documentation for a comprehensive list.

The first PVSystem parameters are *surface_tilt* and *surface_azimuth*. These parameters are used in PVSystem methods such as *`get_aoi()`* and *`get_irradiance()`*. Angle of incidence (AOI) calculations require *surface_tilt*, *surface_azimuth* and also the sun position. The *`get_aoi()`* method uses the *surface_tilt* and *surface_azimuth* attributes in its PVSystem object, and so requires only *solar_zenith* and *solar_azimuth* as arguments.

```
# 20 deg tilt, south-facing
In [9]: system = pvsystem.PVSystem(surface_tilt=20, surface_azimuth=180)

In [10]: print(system.surface_tilt, system.surface_azimuth)
20 180
```

(continues on next page)

(continued from previous page)

```
# call get_aoi with solar_zenith, solar_azimuth
In [11]: aoi = system.get_aoi(30, 180)

In [12]: print(aoi)
9.999999999999975
```

module_parameters and *inverter_parameters* contain the data necessary for computing DC and AC power using one of the available PVSystem methods. These are typically specified using data from the *retrieve_sam()* function:

```
# retrieve_sam returns a dict. the dict keys are module names,
# and the values are model parameters for that module
In [13]: modules = pvsystem.retrieve_sam('cecmod')

In [14]: module_parameters = modules['Canadian_Solar_Inc__CS5P_220M']

In [15]: inverters = pvsystem.retrieve_sam('cecinverter')

In [16]: inverter_parameters = inverters['ABB__MICRO_0_25_I_OUTD_US_208__208V_']

In [17]: system = pvsystem.PVSystem(module_parameters=module_parameters, inverter_
↪parameters=inverter_parameters)
```

The module and/or inverter parameters can also be specified manually. This is useful for specifying modules and inverters that are not included in the supplied databases. It is also useful for specifying systems for use with the PVWatts models, as demonstrated in *Design philosophy*.

The *losses_parameters* attribute contains data that may be used with methods that calculate system losses. At present, these methods include only `PVSystem.pvwatts_losses` and *pvsystem.pvwatts_losses*, but we hope to add more related functions and methods in the future.

The attributes *modules_per_string* and *strings_per_inverter* are used in the `scale_voltage_current_power()` method. Some DC power models in *ModelChain* automatically call this method and make use of these attributes. As an example, consider a system with 35 modules arranged into 5 strings of 7 modules each.

```
In [18]: system = pvsystem.PVSystem(modules_per_string=7, strings_per_inverter=5)

# crude numbers from a single module
In [19]: data = pd.DataFrame({'v_mp': 8, 'v_oc': 10, 'i_mp': 5, 'i_x': 6,
   ....:                      'i_xx': 4, 'i_sc': 7, 'p_mp': 40}, index=[0])
   ....:

In [20]: data_scaled = system.scale_voltage_current_power(data)

In [21]: print(data_scaled)
   v_mp  v_oc  i_mp  i_x  i_xx  i_sc  p_mp
0    56    70    25   30    20    35  1400
```

3.7.3 SingleAxisTracker

The *SingleAxisTracker* is a subclass of *PVSystem*. The SingleAxisTracker class includes a few more keyword arguments and attributes that are specific to trackers, plus the *singleaxis()* method. It also overrides the *get_aoi* and *get_irradiance* methods.

3.8 ModelChain

The *ModelChain* class provides a high-level interface for standardized PV modeling. The class aims to automate much of the modeling process while providing user-control and remaining extensible. This guide aims to build users' understanding of the ModelChain class. It assumes some familiarity with object-oriented code in Python, but most information should be understandable even without a solid understanding of classes.

A *ModelChain* is composed of a *PVSystem* object and a *Location* object. A PVSystem object represents an assembled collection of modules, inverters, etc., a Location object represents a particular place on the planet, and a ModelChain object describes the modeling chain used to calculate a system's output at that location. The PVSystem and Location objects will be described in detail in another guide.

Modeling with a *ModelChain* typically involves 3 steps:

1. Creating the *ModelChain*.
2. Executing the *ModelChain.run_model()* method with prepared weather data.
3. Examining the model results that *run_model()* stored in attributes of the *ModelChain*.

3.8.1 A simple ModelChain example

Before delving into the intricacies of ModelChain, we provide a brief example of the modeling steps using ModelChain. First, we import pvlib's objects, module data, and inverter data.

```
In [1]: import pandas as pd

In [2]: import numpy as np

# pvlib imports
In [3]: import pvlib

In [4]: from pvlib.pvsystem import PVSystem

In [5]: from pvlib.location import Location

In [6]: from pvlib.modelchain import ModelChain

In [7]: from pvlib.temperature import TEMPERATURE_MODEL_PARAMETERS

In [8]: temperature_model_parameters = TEMPERATURE_MODEL_PARAMETERS['sapm']['open_
↪rack_glass_glass']

# load some module and inverter specifications
In [9]: sandia_modules = pvlib.pvsystem.retrieve_sam('SandiaMod')

In [10]: cec_inverters = pvlib.pvsystem.retrieve_sam('cecinverter')

In [11]: sandia_module = sandia_modules['Canadian_Solar_CS5P_220M___2009_']

In [12]: cec_inverter = cec_inverters['ABB__MICRO_0_25_I_OUTD_US_208__208V_']
```

Now we create a Location object, a PVSystem object, and a ModelChain object.

```
In [13]: location = Location(latitude=32.2, longitude=-110.9)

In [14]: system = PVSystem(surface_tilt=20, surface_azimuth=200,
```

(continues on next page)

(continued from previous page)

```
   ....:                     module_parameters=sandia_module,
   ....:                     inverter_parameters=cec_inverter,
   ....:                     temperature_model_parameters=temperature_model_parameters)
   ....:

In [15]: mc = ModelChain(system, location)
```

Printing a ModelChain object will display its models.

```
In [16]: print(mc)
ModelChain:
  name: None
  orientation_strategy: None
  clearsky_model: ineichen
  transposition_model: haydavies
  solar_position_method: nrel_numpy
  airmass_model: kastenyoung1989
  dc_model: sapm
  ac_model: snlinverter
  aoi_model: sapm_aoi_loss
  spectral_model: sapm_spectral_loss
  temperature_model: sapm_temp
  losses_model: no_extra_losses
```

Next, we run a model with some simple weather data.

```
In [17]: weather = pd.DataFrame([[1050, 1000, 100, 30, 5]],
   ....:                         columns=['ghi', 'dni', 'dhi', 'temp_air', 'wind_speed
↪'],
   ....:                         index=[pd.Timestamp('20170401 1200', tz='US/Arizona
↪')])
   ....:

In [18]: mc.run_model(weather);
```

ModelChain stores the modeling results on a series of attributes. A few examples are shown below.

```
In [19]: mc.aoi
Out[19]:
2017-04-01 12:00:00-07:00    15.929553
Name: aoi, dtype: float64
```

```
In [20]: mc.cell_temperature
Out[20]:
2017-04-01 12:00:00-07:00    58.087879
dtype: float64
```

```
In [21]: mc.dc
Out[21]:
                              i_sc      i_mp  ...       i_x      i_xx
2017-04-01 12:00:00-07:00  5.485953  4.860313  ...  5.363074  3.401312

[1 rows x 7 columns]
```

```
In [22]: mc.ac
Out[22]:
2017-04-01 12:00:00-07:00    189.990907
dtype: float64
```

The remainder of this guide examines the ModelChain functionality and explores common pitfalls.

3.8.2 Defining a ModelChain

A *ModelChain* object is defined by:

1. The properties of its *PVSystem* and *Location* objects
2. The keyword arguments passed to it at construction

ModelChain uses the keyword arguments passed to it to determine the models for the simulation. The documentation describes the allowed values for each keyword argument. If a keyword argument is not supplied, ModelChain will attempt to infer the correct set of models by inspecting the Location and PVSystem attributes.

Below, we show some examples of how to define a ModelChain.

Let's make the most basic Location and PVSystem objects and build from there.

```
In [23]: location = Location(32.2, -110.9)

In [24]: poorly_specified_system = PVSystem()

In [25]: print(location)
Location:
  name: None
  latitude: 32.2
  longitude: -110.9
  altitude: 0
  tz: UTC

In [26]: print(poorly_specified_system)
PVSystem:
  name: None
  surface_tilt: 0
  surface_azimuth: 180
  module: None
  inverter: None
  albedo: 0.25
  racking_model: open_rack
```

These basic objects do not have enough information for ModelChain to be able to automatically determine its set of models, so the ModelChain will throw an error when we try to create it.

```
In [27]: ModelChain(poorly_specified_system, location)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-27-4e9151e6ff63> in <module>
----> 1 ModelChain(poorly_specified_system, location)

~/checkouts/readthedocs.org/user_builds/tylunelpvlib-python/checkouts/latest/pvlib/
↪modelchain.py in __init__(self, system, location, orientation_strategy, clearsky_
↪model, transposition_model, solar_position_method, airmass_model, dc_model, ac_
↪model, aoi_model, spectral_model, temperature_model, losses_model, name, **kwargs)
```

(continues on next page)

(continued from previous page)

```
    319
    320         # calls setters
--> 321         self.dc_model = dc_model
    322         self.ac_model = ac_model
    323         self.aoi_model = aoi_model

~/checkouts/readthedocs.org/user_builds/tylunelpvlib-python/checkouts/latest/pvlib/
↪modelchain.py in dc_model(self, model)
    393         # guess at model if None
    394         if model is None:
--> 395             self._dc_model, model = self.infer_dc_model()
    396
    397         # Set model and validate parameters

~/checkouts/readthedocs.org/user_builds/tylunelpvlib-python/checkouts/latest/pvlib/
↪modelchain.py in infer_dc_model(self)
    437             return self.pvwatts_dc, 'pvwatts'
    438         else:
--> 439             raise ValueError('could not infer DC model from '
    440                              'system.module_parameters. Check '
    441                              'system.module_parameters or explicitly '

ValueError: could not infer DC model from system.module_parameters. Check system.
↪module_parameters or explicitly set the model with the dc_model kwarg.
```

Next, we define a PVSystem with a module from the SAPM database and an inverter from the CEC database. ModelChain will examine the PVSystem object's properties and determine that it should choose the SAPM DC model, AC model, AOI loss model, and spectral loss model.

```
In [28]: sapm_system = PVSystem(
   ....:     module_parameters=sandia_module,
   ....:     inverter_parameters=cec_inverter,
   ....:     temperature_model_parameters=temperature_model_parameters)
   ....:

In [29]: mc = ModelChain(sapm_system, location)

In [30]: print(mc)
ModelChain:
  name: None
  orientation_strategy: None
  clearsky_model: ineichen
  transposition_model: haydavies
  solar_position_method: nrel_numpy
  airmass_model: kastenyoung1989
  dc_model: sapm
  ac_model: snlinverter
  aoi_model: sapm_aoi_loss
  spectral_model: sapm_spectral_loss
  temperature_model: sapm_temp
  losses_model: no_extra_losses
```

```
In [31]: mc.run_model(weather);

In [32]: mc.ac
Out[32]:
```

(continues on next page)

(continued from previous page)

```
2017-04-01 12:00:00-07:00    176.649413
dtype: float64
```

Alternatively, we could have specified single diode or PVWatts related information in the PVSystem construction. Here we pass PVWatts data to the PVSystem. ModelChain will automatically determine that it should choose PVWatts DC and AC models. ModelChain still needs us to specify `aoi_model` and `spectral_model` keyword arguments because the `system.module_parameters` dictionary does not contain enough information to determine which of those models to choose.

```
In [33]: pvwatts_system = PVSystem(
   ....:     module_parameters={'pdc0': 240, 'gamma_pdc': -0.004},
   ....:     inverter_parameters={'pdc0': 240},
   ....:     temperature_model_parameters=temperature_model_parameters)
   ....:

In [34]: mc = ModelChain(pvwatts_system, location,
   ....:                 aoi_model='physical', spectral_model='no_loss')
   ....:

In [35]: print(mc)
ModelChain:
  name: None
  orientation_strategy: None
  clearsky_model: ineichen
  transposition_model: haydavies
  solar_position_method: nrel_numpy
  airmass_model: kastenyoung1989
  dc_model: pvwatts_dc
  ac_model: pvwatts_inverter
  aoi_model: physical_aoi_loss
  spectral_model: no_spectral_loss
  temperature_model: sapm_temp
  losses_model: no_extra_losses
```

```
In [36]: mc.run_model(weather);

In [37]: mc.ac
Out[37]:
2017-04-01 12:00:00-07:00    198.519201
dtype: float64
```

User-supplied keyword arguments override ModelChain's inspection methods. For example, we can tell ModelChain to use different loss functions for a PVSystem that contains SAPM-specific parameters.

```
In [38]: sapm_system = PVSystem(
   ....:     module_parameters=sandia_module,
   ....:     inverter_parameters=cec_inverter,
   ....:     temperature_model_parameters=temperature_model_parameters)
   ....:

In [39]: mc = ModelChain(sapm_system, location, aoi_model='physical', spectral_model=
↪'no_loss')

In [40]: print(mc)
ModelChain:
  name: None
```

(continues on next page)

(continued from previous page)

```
  orientation_strategy: None
  clearsky_model: ineichen
  transposition_model: haydavies
  solar_position_method: nrel_numpy
  airmass_model: kastenyoung1989
  dc_model: sapm
  ac_model: snlinverter
  aoi_model: physical_aoi_loss
  spectral_model: no_spectral_loss
  temperature_model: sapm_temp
  losses_model: no_extra_losses
```

```
In [41]: mc.run_model(weather);

In [42]: mc.ac
Out[42]:
2017-04-01 12:00:00-07:00    177.381377
dtype: float64
```

Of course, these choices can also lead to failure when executing *run_model()* if your system objects do not contain the required parameters for running the model.

3.8.3 Demystifying ModelChain internals

The ModelChain class has a lot going in inside it in order to make users' code as simple as possible.

The key parts of ModelChain are:

1. The *ModelChain.run_model()* method
2. A set of methods that wrap and call the PVSystem methods.
3. A set of methods that inspect user-supplied objects to determine the appropriate default models.

run_model

Most users will only interact with the *run_model()* method. The *run_model()* method, shown below, calls a series of methods to complete the modeling steps. The first method, *prepare_inputs()*, computes parameters such as solar position, airmass, angle of incidence, and plane of array irradiance. The *prepare_inputs()* method also assigns default values for temperature (20 C) and wind speed (0 m/s) if these inputs are not provided. *prepare_inputs()* requires all irradiance components (GHI, DNI, and DHI). See *complete_irradiance()* and *DNI estimation models* for methods and functions that can help fully define the irradiance inputs.

Next, *run_model()* calls the wrapper methods for AOI loss, spectral loss, effective irradiance, cell temperature, DC power, AC power, and other losses. These methods are assigned to standard names, as described in the next section.

The methods called by *run_model()* store their results in a series of ModelChain attributes: `times`, `solar_position`, `airmass`, `irradiance`, `total_irrad`, `effective_irradiance`, `weather`, `temps`, `aoi`, `aoi_modifier`, `spectral_modifier`, `dc`, `ac`, `losses`.

```
In [43]: mc.run_model??
Signature: mc.run_model(weather, times=None)
Source:
    def run_model(self, weather, times=None):
        """
```

(continues on next page)

(continued from previous page)

```
        Run the model.

        Parameters
        ----------
        weather : DataFrame
            Column names must be ``'dni'``, ``'ghi'``, ``'dhi'``,
            ``'wind_speed'``, ``'temp_air'``. All irradiance components
            are required. Air temperature of 20 C and wind speed
            of 0 m/s will be added to the DataFrame if not provided.
        times : None, deprecated
            Deprecated argument included for API compatibility, but not
            used internally. The index of the weather DataFrame is used
            for times.

        Returns
        -------
        self

        Assigns attributes: ``solar_position``, ``airmass``, ``irradiance``,
        ``total_irrad``, ``effective_irradiance``, ``weather``,
        ``cell_temperature``, ``aoi``, ``aoi_modifier``, ``spectral_modifier``,
        ``dc``, ``ac``, ``losses``,
        ``diode_params`` (if dc_model is a single diode model)
        """
        if times is not None:
            warnings.warn('times keyword argument is deprecated and will be '
                          'removed in 0.8. The index of the weather DataFrame '
                          'is used for times.', pvlibDeprecationWarning)

        self.prepare_inputs(weather)
        self.aoi_model()
        self.spectral_model()
        self.effective_irradiance_model()
        self.temperature_model()
        self.dc_model()
        self.losses_model()
        self.ac_model()

        return self
File:      ~/checkouts/readthedocs.org/user_builds/tylunelpvlib-python/checkouts/
↪latest/pvlib/modelchain.py
Type:      method
```

Finally, the *complete_irradiance()* method is available for calculating the full set of GHI, DNI, or DHI if only two of these three series are provided. The completed dataset can then be passed to *run_model()*.

Wrapping methods into a unified API

Readers may notice that the source code of the ModelChain.run_model method is model-agnostic. ModelChain.run_model calls generic methods such as `self.dc_model` rather than a specific model such as `singlediode`. So how does the ModelChain.run_model know what models it's supposed to run? The answer comes in two parts, and allows us to explore more of the ModelChain API along the way.

First, ModelChain has a set of methods that wrap the PVSystem methods that perform the calculations (or further wrap the pvsystem.py module's functions). Each of these methods takes the same arguments (`self`) and sets the same attributes, thus creating a uniform API. For example, the ModelChain.pvwatts_dc method is shown below. Its

only argument is `self`, and it sets the `dc` attribute.

```
In [44]: mc.pvwatts_dc??
Signature: mc.pvwatts_dc()
Docstring: <no docstring>
Source:
    def pvwatts_dc(self):
        self.dc = self.system.pvwatts_dc(self.effective_irradiance,
                                         self.cell_temperature)
        return self
File:      ~/checkouts/readthedocs.org/user_builds/tylunelpvlib-python/checkouts/
↪latest/pvlib/modelchain.py
Type:      method
```

The ModelChain.pvwatts_dc method calls the pvwatts_dc method of the PVSystem object that we supplied using data that is stored in its own `effective_irradiance` and `cell_temperature` attributes. Then it assigns the result to the `dc` attribute of the ModelChain object. The code below shows a simple example of this.

```
# make the objects
In [45]: pvwatts_system = PVSystem(
   ....:     module_parameters={'pdc0': 240, 'gamma_pdc': -0.004},
   ....:     inverter_parameters={'pdc0': 240},
   ....:     temperature_model_parameters=temperature_model_parameters)
   ....:

In [46]: mc = ModelChain(pvwatts_system, location,
   ....:                 aoi_model='no_loss', spectral_model='no_loss')
   ....:

# manually assign data to the attributes that ModelChain.pvwatts_dc will need.
# for standard workflows, run_model would assign these attributes.
In [47]: mc.effective_irradiance = pd.Series(1000, index=[pd.Timestamp('20170401 1200-
↪0700')])

In [48]: mc.cell_temperature = pd.Series(50, index=[pd.Timestamp('20170401 1200-0700
↪')])

# run ModelChain.pvwatts_dc and look at the result
In [49]: mc.pvwatts_dc();

In [50]: mc.dc
Out[50]:
2017-04-01 12:00:00-07:00    216.0
dtype: float64
```

The ModelChain.sapm method works similarly to the ModelChain.pvwatts_dc method. It calls the PVSystem.sapm method using stored data, then assigns the result to the `dc` attribute. The ModelChain.sapm method differs from the ModelChain.pvwatts_dc method in three notable ways. First, the PVSystem.sapm method expects different units for effective irradiance, so ModelChain handles the conversion for us. Second, the PVSystem.sapm method (and the PVSystem.singlediode method) returns a DataFrame with current, voltage, and power parameters rather than a simple Series of power. Finally, this current and voltage information allows the SAPM and single diode model paths to support the concept of modules in series and parallel, which is handled by the PVSystem.scale_voltage_current_power method.

```
In [51]: mc.sapm??
Signature: mc.sapm()
Docstring: <no docstring>
Source:
```

(continues on next page)

(continued from previous page)

```
    def sapm(self):
        self.dc = self.system.sapm(self.effective_irradiance,
                                   self.cell_temperature)

        self.dc = self.system.scale_voltage_current_power(self.dc)

        return self
File:      ~/checkouts/readthedocs.org/user_builds/tylunelpvlib-python/checkouts/
↪latest/pvlib/modelchain.py
Type:      method
```

```
# make the objects
In [52]: sapm_system = PVSystem(
   ....:     module_parameters=sandia_module,
   ....:     inverter_parameters=cec_inverter,
   ....:     temperature_model_parameters=temperature_model_parameters)
   ....:

In [53]: mc = ModelChain(sapm_system, location)

# manually assign data to the attributes that ModelChain.sapm will need.
# for standard workflows, run_model would assign these attributes.
In [54]: mc.effective_irradiance = pd.Series(1000, index=[pd.Timestamp('20170401 1200-
↪0700')])

In [55]: mc.cell_temperature = pd.Series(50, index=[pd.Timestamp('20170401 1200-0700
↪')])

# run ModelChain.sapm and look at the result
In [56]: mc.sapm();

In [57]: mc.dc
Out[57]:
                            i_sc      i_mp  ...       i_x      i_xx
2017-04-01 12:00:00-07:00  5.14168  4.566863  ...  5.025377  3.219662

[1 rows x 7 columns]
```

We've established that the `ModelChain.pvwatts_dc` and `ModelChain.sapm` have the same API: they take the same arugments (`self`) and they both set the `dc` attribute.* Because the methods have the same API, we can call them in the same way. ModelChain includes a large number of methods that perform the same API-unification roles for each modeling step.

Again, so how does the ModelChain.run_model know which models it's supposed to run?

At object construction, ModelChain assigns the desired model's method (e.g. `ModelChain.pvwatts_dc`) to the corresponding generic attribute (e.g. `ModelChain.dc_model`) using a method described in the next section.

```
In [58]: pvwatts_system = PVSystem(
   ....:     module_parameters={'pdc0': 240, 'gamma_pdc': -0.004},
   ....:     inverter_parameters={'pdc0': 240},
   ....:     temperature_model_parameters=temperature_model_parameters)
   ....:

In [59]: mc = ModelChain(pvwatts_system, location,
   ....:                 aoi_model='no_loss', spectral_model='no_loss')
   ....:
```

(continues on next page)

(continued from previous page)

```
In [60]: mc.dc_model.__func__
Out[60]: <function pvlib.modelchain.ModelChain.pvwatts_dc(self)>
```

The ModelChain.run_model method can ignorantly call `self.dc_module` because the API is the same for all methods that may be assigned to this attribute.

* some readers may object that the API is *not* actually the same because the type of the `dc` attribute is different (Series vs. DataFrame)!

Inferring models

How does ModelChain infer the appropriate model types? ModelChain uses a series of methods (ModelChain.infer_dc_model, ModelChain.infer_ac_model, etc.) that examine the user-supplied PVSystem object. The inference methods use set logic to assign one of the model-specific methods, such as ModelChain.sapm or ModelChain.snlinverter, to the universal method names ModelChain.dc_model and ModelChain.ac_model. A few examples are shown below.

```
In [61]: mc.infer_dc_model??
Signature: mc.infer_dc_model()
Docstring: <no docstring>
Source:
    def infer_dc_model(self):
        params = set(self.system.module_parameters.keys())
        if set(['A0', 'A1', 'C7']) <= params:
            return self.sapm, 'sapm'
        elif set(['a_ref', 'I_L_ref', 'I_o_ref', 'R_sh_ref',
                  'R_s', 'Adjust']) <= params:
            return self.cec, 'cec'
        elif set(['a_ref', 'I_L_ref', 'I_o_ref', 'R_sh_ref',
                  'R_s']) <= params:
            return self.desoto, 'desoto'
        elif set(['gamma_ref', 'mu_gamma', 'I_L_ref', 'I_o_ref',
                  'R_sh_ref', 'R_sh_0', 'R_sh_exp', 'R_s']) <= params:
            return self.pvsyst, 'pvsyst'
        elif set(['pdc0', 'gamma_pdc']) <= params:
            return self.pvwatts_dc, 'pvwatts'
        else:
            raise ValueError('could not infer DC model from '
                             'system.module_parameters. Check '
                             'system.module_parameters or explicitly '
                             'set the model with the dc_model kwarg.')
File:      ~/checkouts/readthedocs.org/user_builds/tylunelpvlib-python/checkouts/
↪latest/pvlib/modelchain.py
Type:      method
```

```
In [62]: mc.infer_ac_model??
Signature: mc.infer_ac_model()
Docstring: <no docstring>
Source:
    def infer_ac_model(self):
        inverter_params = set(self.system.inverter_parameters.keys())
        if set(['C0', 'C1', 'C2']) <= inverter_params:
            return self.snlinverter
        elif set(['ADRCoefficients']) <= inverter_params:
```

(continues on next page)

(continued from previous page)

```
            return self.adrinverter
        elif set(['pdc0']) <= inverter_params:
            return self.pvwatts_inverter
        else:
            raise ValueError('could not infer AC model from '
                             'system.inverter_parameters. Check '
                             'system.inverter_parameters or explicitly '
                             'set the model with the ac_model kwarg.')
File:      ~/checkouts/readthedocs.org/user_builds/tylunelpvlib-python/checkouts/
↪latest/pvlib/modelchain.py
Type:      method
```

3.8.4 User-defined models

Users may also write their own functions and pass them as arguments to ModelChain. The first argument of the function must be a ModelChain instance. For example, the functions below implement the PVUSA model and a wrapper function appropriate for use with ModelChain. This follows the pattern of implementing the core models using the simplest possible functions, and then implementing wrappers to make them easier to use in specific applications. Of course, you could implement it in a single function if you wanted to.

```
In [63]: def pvusa(poa_global, wind_speed, temp_air, a, b, c, d):
   ....:     """
   ....:     Calculates system power according to the PVUSA equation
   ....:     P = I * (a + b*I + c*W + d*T)
   ....:     where
   ....:     P is the output power,
   ....:     I is the plane of array irradiance,
   ....:     W is the wind speed, and
   ....:     T is the temperature
   ....:     a, b, c, d are empirically derived parameters.
   ....:     """
   ....:     return poa_global * (a + b*poa_global + c*wind_speed + d*temp_air)
   ....:

In [64]: def pvusa_mc_wrapper(mc):
   ....:     mc.dc = pvusa(mc.total_irrad['poa_global'], mc.weather['wind_speed'], mc.
↪weather['temp_air'],
   ....:                   mc.system.module_parameters['a'], mc.system.module_
↪parameters['b'],
   ....:                   mc.system.module_parameters['c'], mc.system.module_
↪parameters['d'])
   ....:

    # returning mc is optional, but enables method chaining
In [65]: def pvusa_ac_mc(mc):
   ....:     mc.ac = mc.dc
   ....:     return mc
   ....:

In [66]: def no_loss_temperature(mc):
   ....:     mc.cell_temperature = mc.weather['temp_air']
   ....:     return mc
   ....:
```

```
In [67]: module_parameters = {'a': 0.2, 'b': 0.00001, 'c': 0.001, 'd': -0.00005}

In [68]: pvusa_system = PVSystem(module_parameters=module_parameters)

In [69]: mc = ModelChain(pvusa_system, location,
   ....:                 dc_model=pvusa_mc_wrapper, ac_model=pvusa_ac_mc,
   ....:                 temperature_model=no_loss_temperature,
   ....:                 aoi_model='no_loss', spectral_model='no_loss')
   ....:
```

A ModelChain object uses Python's functools.partial function to assign itself as the argument to the user-supplied functions.

```
In [70]: mc.dc_model.func
Out[70]: <function __main__.pvusa_mc_wrapper(mc)>
```

The end result is that ModelChain.run_model works as expected!

```
In [71]: mc.run_model(weather);

In [72]: mc.dc
Out[72]:
2017-04-01 12:00:00-07:00    209.518773
dtype: float64
```

3.9 Time and time zones

Dealing with time and time zones can be a frustrating experience in any programming language and for any application. pvlib-python relies on `pandas` and pytz to handle time and time zones. Therefore, the vast majority of the information in this document applies to any time series analysis using pandas and is not specific to pvlib-python.

3.9.1 General functionality

pvlib makes extensive use of pandas due to its excellent time series functionality. Take the time to become familiar with pandas' Time Series / Date functionality page. It is also worthwhile to become familiar with pure Python's `datetime` module, although we usually recommend using the corresponding pandas functionality where possible.

First, we'll import the libraries that we'll use to explore the basic time and time zone functionality in python and pvlib.

```
In [1]: import datetime

In [2]: import pandas as pd

In [3]: import pytz
```

Finding a time zone

pytz is based on the Olson time zone database. You can obtain a list of all valid time zone strings with `pytz.all_timezones`. It's a long list, so we only print every 20th time zone.

```
In [4]: len(pytz.all_timezones)
Out[4]: 593

In [5]: pytz.all_timezones[::20]
Out[5]:
['Africa/Abidjan',
 'Africa/Douala',
 'Africa/Mbabane',
 'America/Argentina/Catamarca',
 'America/Belize',
 'America/Curacao',
 'America/Guatemala',
 'America/Kentucky/Louisville',
 'America/Mexico_City',
 'America/Phoenix',
 'America/Shiprock',
 'America/Yellowknife',
 'Asia/Ashgabat',
 'Asia/Dili',
 'Asia/Katmandu',
 'Asia/Pyongyang',
 'Asia/Tokyo',
 'Atlantic/Madeira',
 'Australia/Perth',
 'Canada/Saskatchewan',
 'Etc/GMT+6',
 'Etc/Greenwich',
 'Europe/Gibraltar',
 'Europe/Oslo',
 'Europe/Vatican',
 'Indian/Christmas',
 'Mexico/General',
 'Pacific/Guam',
 'Pacific/Saipan',
 'US/East-Indiana']
```

Wikipedia's List of tz database time zones is also good reference.

The `pytz.country_timezones` function is useful, too.

```
In [6]: pytz.country_timezones('US')
Out[6]:
['America/New_York',
 'America/Detroit',
 'America/Kentucky/Louisville',
 'America/Kentucky/Monticello',
 'America/Indiana/Indianapolis',
 'America/Indiana/Vincennes',
 'America/Indiana/Winamac',
 'America/Indiana/Marengo',
 'America/Indiana/Petersburg',
 'America/Indiana/Vevay',
 'America/Chicago',
 'America/Indiana/Tell_City',
 'America/Indiana/Knox',
 'America/Menominee',
 'America/North_Dakota/Center',
 'America/North_Dakota/New_Salem',
```

(continues on next page)

(continued from previous page)

```
 'America/North_Dakota/Beulah',
 'America/Denver',
 'America/Boise',
 'America/Phoenix',
 'America/Los_Angeles',
 'America/Anchorage',
 'America/Juneau',
 'America/Sitka',
 'America/Metlakatla',
 'America/Yakutat',
 'America/Nome',
 'America/Adak',
 'Pacific/Honolulu']
```

And don't forget about Python's `filter()` function.

```
In [7]: list(filter(lambda x: 'GMT' in x, pytz.all_timezones))
Out[7]:
['Etc/GMT',
 'Etc/GMT+0',
 'Etc/GMT+1',
 'Etc/GMT+10',
 'Etc/GMT+11',
 'Etc/GMT+12',
 'Etc/GMT+2',
 'Etc/GMT+3',
 'Etc/GMT+4',
 'Etc/GMT+5',
 'Etc/GMT+6',
 'Etc/GMT+7',
 'Etc/GMT+8',
 'Etc/GMT+9',
 'Etc/GMT-0',
 'Etc/GMT-1',
 'Etc/GMT-10',
 'Etc/GMT-11',
 'Etc/GMT-12',
 'Etc/GMT-13',
 'Etc/GMT-14',
 'Etc/GMT-2',
 'Etc/GMT-3',
 'Etc/GMT-4',
 'Etc/GMT-5',
 'Etc/GMT-6',
 'Etc/GMT-7',
 'Etc/GMT-8',
 'Etc/GMT-9',
 'Etc/GMT0',
 'GMT',
 'GMT+0',
 'GMT-0',
 'GMT0']
```

Note that while pytz has `'EST'` and `'MST'`, it does not have `'PST'`. Use `'Etc/GMT+8'` instead, or see *Fixed offsets*.

Timestamps

`pandas.Timestamp` and `pandas.DatetimeIndex` can be created in many ways. Here we focus on the time zone issues surrounding them; see the pandas documentation for more information.

First, create a time zone naive pandas.Timestamp.

```
In [8]: pd.Timestamp('2015-1-1 00:00')
Out[8]: Timestamp('2015-01-01 00:00:00')
```

You can specify the time zone using the `tz` keyword argument or the `tz_localize` method of Timestamp and DatetimeIndex objects.

```
In [9]: pd.Timestamp('2015-1-1 00:00', tz='America/Denver')
Out[9]: Timestamp('2015-01-01 00:00:00-0700', tz='America/Denver')

In [10]: pd.Timestamp('2015-1-1 00:00').tz_localize('America/Denver')
Out[10]: Timestamp('2015-01-01 00:00:00-0700', tz='America/Denver')
```

Localized Timestamps can be converted from one time zone to another.

```
In [11]: midnight_mst = pd.Timestamp('2015-1-1 00:00', tz='America/Denver')

In [12]: corresponding_utc = midnight_mst.tz_convert('UTC')  # returns a new Timestamp

In [13]: corresponding_utc
Out[13]: Timestamp('2015-01-01 07:00:00+0000', tz='UTC')
```

It does not make sense to convert a time stamp that has not been localized, and pandas will raise an exception if you try to do so.

```
In [14]: midnight = pd.Timestamp('2015-1-1 00:00')

In [15]: midnight.tz_convert('UTC')
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-15-f99dcf3885a1> in <module>
----> 1 midnight.tz_convert('UTC')

pandas/_libs/tslibs/timestamps.pyx in pandas._libs.tslibs.timestamps.Timestamp.tz_
↪convert()

TypeError: Cannot convert tz-naive Timestamp, use tz_localize to localize
```

The difference between `tz_localize` and `tz_convert` is a common source of confusion for new users. Just remember: localize first, convert later.

Daylight savings time

Some time zones are aware of daylight savings time and some are not. For example the winter time results are the same for US/Mountain and MST, but the summer time results are not.

Note the UTC offset in winter...

```
In [16]: pd.Timestamp('2015-1-1 00:00').tz_localize('US/Mountain')
Out[16]: Timestamp('2015-01-01 00:00:00-0700', tz='US/Mountain')
```

(continues on next page)

(continued from previous page)

```
In [17]: pd.Timestamp('2015-1-1 00:00').tz_localize('Etc/GMT+7')
Out[17]: Timestamp('2015-01-01 00:00:00-0700', tz='Etc/GMT+7')
```

vs. the UTC offset in summer...

```
In [18]: pd.Timestamp('2015-6-1 00:00').tz_localize('US/Mountain')
Out[18]: Timestamp('2015-06-01 00:00:00-0600', tz='US/Mountain')

In [19]: pd.Timestamp('2015-6-1 00:00').tz_localize('Etc/GMT+7')
Out[19]: Timestamp('2015-06-01 00:00:00-0700', tz='Etc/GMT+7')
```

pandas and pytz make this time zone handling possible because pandas stores all times as integer nanoseconds since January 1, 1970. Here is the pandas time representation of the integers 1 and 1e9.

```
In [20]: pd.Timestamp(1)
Out[20]: Timestamp('1970-01-01 00:00:00.000000001')

In [21]: pd.Timestamp(1e9)
Out[21]: Timestamp('1970-01-01 00:00:01')
```

So if we specify times consistent with the specified time zone, pandas will use the same integer to represent them.

```
# US/Mountain
In [22]: pd.Timestamp('2015-6-1 01:00', tz='US/Mountain').value
Out[22]: 1433142000000000000

# MST
In [23]: pd.Timestamp('2015-6-1 00:00', tz='Etc/GMT+7').value
Out[23]: 1433142000000000000

# Europe/Berlin
In [24]: pd.Timestamp('2015-6-1 09:00', tz='Europe/Berlin').value
Out[24]: 1433142000000000000

# UTC
In [25]: pd.Timestamp('2015-6-1 07:00', tz='UTC').value
Out[25]: 1433142000000000000

# UTC
In [26]: pd.Timestamp('2015-6-1 07:00').value
Out[26]: 1433142000000000000
```

It's ultimately these integers that are used when calculating quantities in pvlib such as solar position.

As stated above, pandas will assume UTC if you do not specify a time zone. This is dangerous, and we recommend using localized timeseries, even if it is UTC.

Fixed offsets

The `'Etc/GMT*'` time zones mentioned above provide fixed offset specifications, but watch out for the counter-intuitive sign convention.

```
In [27]: pd.Timestamp('2015-1-1 00:00', tz='Etc/GMT-2')
Out[27]: Timestamp('2015-01-01 00:00:00+0200', tz='Etc/GMT-2')
```

Fixed offset time zones can also be specified as offset minutes from UTC using `pytz.FixedOffset`.

```
In [28]: pd.Timestamp('2015-1-1 00:00', tz=pytz.FixedOffset(120))
Out[28]: Timestamp('2015-01-01 00:00:00+0200', tz='pytz.FixedOffset(120)')
```

You can also specify the fixed offset directly in the `tz_localize` method, however, be aware that this is not documented and that the offset must be in seconds, not minutes.

```
In [29]: pd.Timestamp('2015-1-1 00:00', tz=7200)
Out[29]: Timestamp('2015-01-01 00:00:00+0200', tz='pytz.FixedOffset(120)')
```

Yet another way to specify a time zone with a fixed offset is by using the string formulation.

```
In [30]: pd.Timestamp('2015-1-1 00:00+0200')
Out[30]: Timestamp('2015-01-01 00:00:00+0200', tz='pytz.FixedOffset(120)')
```

Native Python objects

Sometimes it's convenient to use native Python `datetime.date` and `datetime.datetime` objects, so we demonstrate their use next. pandas Timestamp objects can also be created from time zone aware or naive `datetime.datetime` objects. The behavior is as expected.

```
# tz naive python datetime.datetime object
In [31]: naive_python_dt = datetime.datetime(2015, 6, 1, 0)

# tz naive pandas Timestamp object
In [32]: pd.Timestamp(naive_python_dt)
Out[32]: Timestamp('2015-06-01 00:00:00')

# tz aware python datetime.datetime object
In [33]: aware_python_dt = pytz.timezone('US/Mountain').localize(naive_python_dt)

# tz aware pandas Timestamp object
In [34]: pd.Timestamp(aware_python_dt)
Out[34]: Timestamp('2015-06-01 00:00:00-0600', tz='US/Mountain')
```

One thing to watch out for is that python `datetime.date` objects gain time information when passed to `Timestamp`.

```
# tz naive python datetime.date object (no time info)
In [35]: naive_python_date = datetime.date(2015, 6, 1)

# tz naive pandas Timestamp object (time=midnight)
In [36]: pd.Timestamp(naive_python_date)
Out[36]: Timestamp('2015-06-01 00:00:00')
```

You cannot localize a native Python date object.

```
 # fail
In [37]: pytz.timezone('US/Mountain').localize(naive_python_date)
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-37-c4cced4ef9d9> in <module>
----> 1 pytz.timezone('US/Mountain').localize(naive_python_date)

~/checkouts/readthedocs.org/user_builds/tylunelpvlib-python/envs/latest/lib/python3.7/
↪site-packages/pytz/tzinfo.py in localize(self, dt, is_dst)
```

(continues on next page)

(continued from previous page)

```
    315         Non-existent
    316         '''
--> 317         if dt.tzinfo is not None:
    318             raise ValueError('Not naive datetime (tzinfo is already set)')
    319

AttributeError: 'datetime.date' object has no attribute 'tzinfo'
```

3.9.2 pvlib-specific functionality

Note: This section applies to pvlib >= 0.3. Version 0.2 of pvlib used a `Location` object's `tz` attribute to automagically correct for some time zone issues. This behavior was counter-intuitive to many users and was removed in version 0.3.

How does this general functionality interact with pvlib? Perhaps the two most common places to get tripped up with time and time zone issues in solar power analysis occur during data import and solar position calculations.

Data import

Let's first examine how pvlib handles time when it imports a TMY3 file.

```
In [38]: import os

In [39]: import inspect

In [40]: import pvlib

# some gymnastics to find the example file
In [41]: pvlib_abspath = os.path.dirname(os.path.abspath(inspect.getfile(pvlib)))

In [42]: file_abspath = os.path.join(pvlib_abspath, 'data', '703165TY.csv')

In [43]: tmy3_data, tmy3_metadata = pvlib.iotools.read_tmy3(file_abspath)

In [44]: tmy3_metadata
Out[44]:
{'USAF': 703165,
 'Name': '"SAND POINT"',
 'State': 'AK',
 'TZ': -9.0,
 'latitude': 55.317,
 'longitude': -160.517,
 'altitude': 7.0}
```

The metadata has a `'TZ'` key with a value of `-9.0`. This is the UTC offset in hours in which the data has been recorded. The `readtmy3()` function read the data in the file, created a `DataFrame` with that data, and then localized the DataFrame's index to have this fixed offset. Here, we print just a few of the rows and columns of the large dataframe.

```
In [45]: tmy3_data.index.tz
Out[45]: pytz.FixedOffset(-540)
```

(continues on next page)

(continued from previous page)

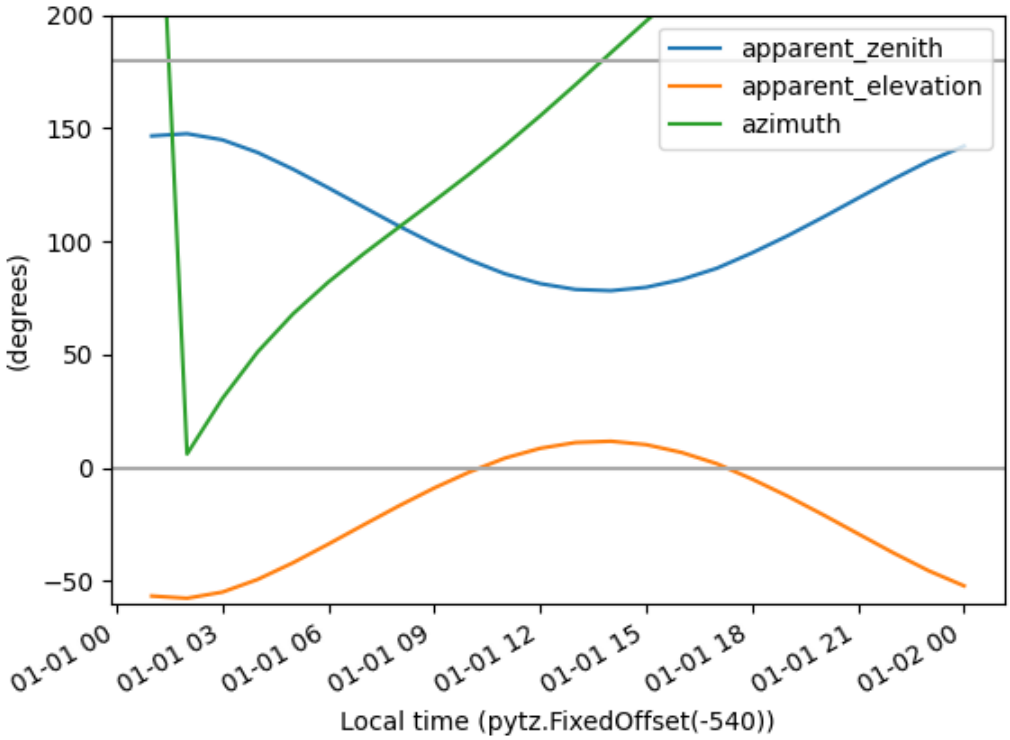
```
In [46]: tmy3_data.loc[tmy3_data.index[0:3], ['GHI', 'DNI', 'AOD']]
Out[46]:
                           GHI  DNI    AOD
1997-01-01 01:00:00-09:00    0    0  0.051
1997-01-01 02:00:00-09:00    0    0  0.051
1997-01-01 03:00:00-09:00    0    0  0.051
```

The `readtmy2()` function also returns a DataFrame with a localized DatetimeIndex.

Solar position

The correct solar position can be immediately calculated from the DataFrame's index since the index has been localized.

```
In [47]: solar_position = pvlib.solarposition.get_solarposition(tmy3_data.index,
   ....:                                                        tmy3_metadata[
↪'latitude'],
   ....:                                                        tmy3_metadata[
↪'longitude'])
   ....:

In [48]: ax = solar_position.loc[solar_position.index[0:24], ['apparent_zenith',
↪'apparent_elevation', 'azimuth']].plot()

In [49]: ax.legend(loc=1);

In [50]: ax.axhline(0, color='darkgray');  # add 0 deg line for sunrise/sunset

In [51]: ax.axhline(180, color='darkgray');  # add 180 deg line for azimuth at solar␣
↪noon

In [52]: ax.set_ylim(-60, 200);  # zoom in, but cuts off full azimuth range

In [53]: ax.set_xlabel('Local time ({})'.format(solar_position.index.tz));

In [54]: ax.set_ylabel('(degrees)');
```



According to the US Navy, on January 1, 1997 at Sand Point, Alaska, sunrise was at 10:09 am, solar noon was at 1:46 pm, and sunset was at 5:23 pm. This is consistent with the data plotted above (and depressing).

Solar position (assumed UTC)

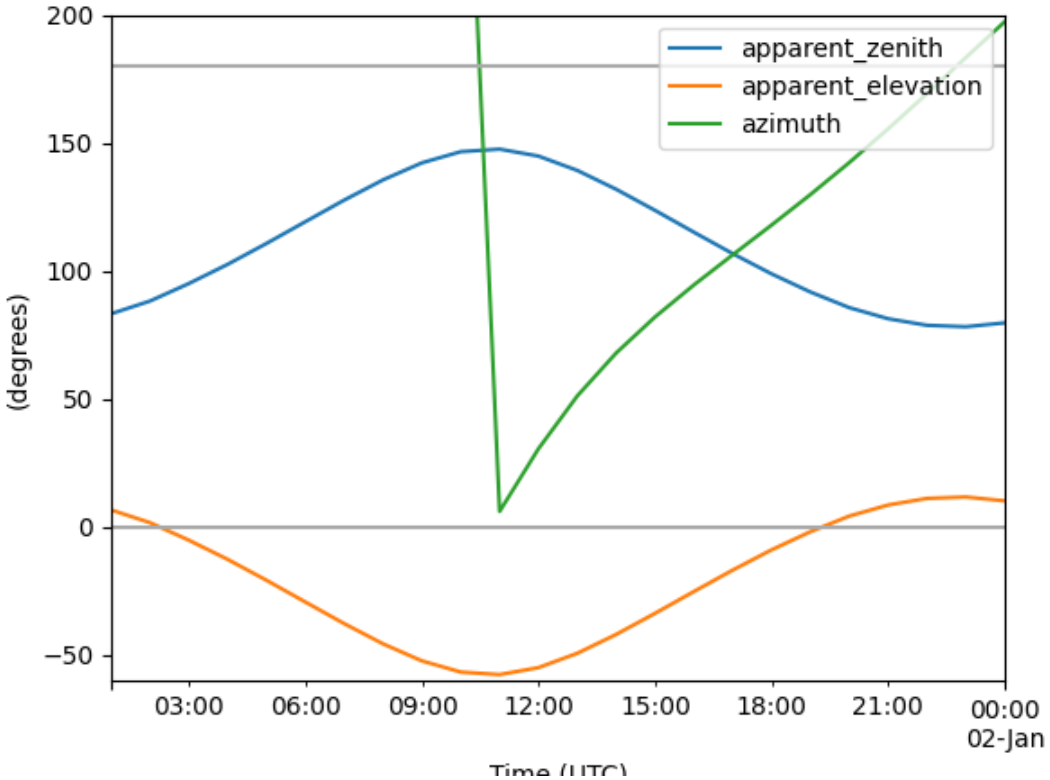
What if we had a DatetimeIndex that was not localized, such as the one below? The solar position calculator will assume UTC time.

```
In [55]: index = pd.date_range(start='1997-01-01 01:00', freq='1h', periods=24)

In [56]: index
Out[56]:
DatetimeIndex(['1997-01-01 01:00:00', '1997-01-01 02:00:00',
               '1997-01-01 03:00:00', '1997-01-01 04:00:00',
               '1997-01-01 05:00:00', '1997-01-01 06:00:00',
               '1997-01-01 07:00:00', '1997-01-01 08:00:00',
               '1997-01-01 09:00:00', '1997-01-01 10:00:00',
               '1997-01-01 11:00:00', '1997-01-01 12:00:00',
               '1997-01-01 13:00:00', '1997-01-01 14:00:00',
               '1997-01-01 15:00:00', '1997-01-01 16:00:00',
               '1997-01-01 17:00:00', '1997-01-01 18:00:00',
               '1997-01-01 19:00:00', '1997-01-01 20:00:00',
               '1997-01-01 21:00:00', '1997-01-01 22:00:00',
               '1997-01-01 23:00:00', '1997-01-02 00:00:00'],
              dtype='datetime64[ns]', freq='H')
```

(continues on next page)

(continued from previous page)

```
In [57]: solar_position_notz = pvlib.solarposition.get_solarposition(index,
   ....:                                                             tmy3_metadata[
↪'latitude'],
   ....:                                                             tmy3_metadata[
↪'longitude'])
   ....:

In [58]: ax = solar_position_notz.loc[solar_position_notz.index[0:24], ['apparent_
↪zenith', 'apparent_elevation', 'azimuth']].plot()

In [59]: ax.legend(loc=1);

In [60]: ax.axhline(0, color='darkgray');  # add 0 deg line for sunrise/sunset

In [61]: ax.axhline(180, color='darkgray');  # add 180 deg line for azimuth at solar␣
↪noon

In [62]: ax.set_ylim(-60, 200);  # zoom in, but cuts off full azimuth range

In [63]: ax.set_xlabel('Time (UTC)');

In [64]: ax.set_ylabel('(degrees)');
```

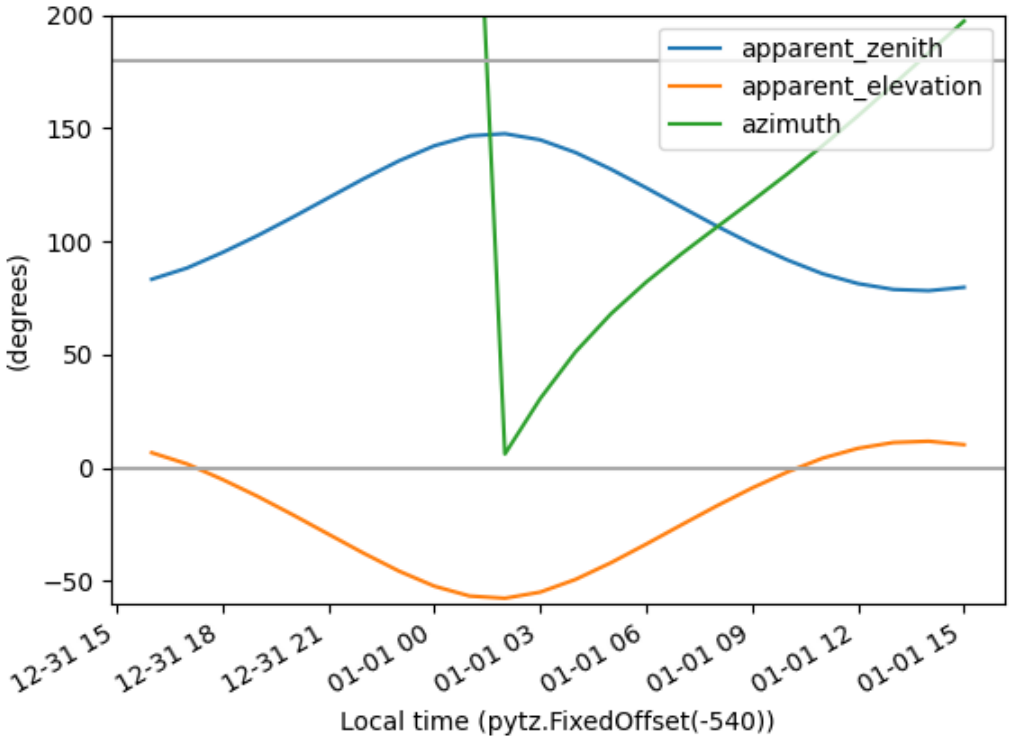


This looks like the plot above, but shifted by 9 hours.

Solar position (calculate and convert)

In principle, one could localize the tz-naive solar position data to UTC, and then convert it to the desired time zone.

```
In [65]: fixed_tz = pytz.FixedOffset(tmy3_metadata['TZ'] * 60)

In [66]: solar_position_hack = solar_position_notz.tz_localize('UTC').tz_
↪convert(fixed_tz)

In [67]: solar_position_hack.index
Out[67]:
DatetimeIndex(['1996-12-31 16:00:00-09:00', '1996-12-31 17:00:00-09:00',
               '1996-12-31 18:00:00-09:00', '1996-12-31 19:00:00-09:00',
               '1996-12-31 20:00:00-09:00', '1996-12-31 21:00:00-09:00',
               '1996-12-31 22:00:00-09:00', '1996-12-31 23:00:00-09:00',
               '1997-01-01 00:00:00-09:00', '1997-01-01 01:00:00-09:00',
               '1997-01-01 02:00:00-09:00', '1997-01-01 03:00:00-09:00',
               '1997-01-01 04:00:00-09:00', '1997-01-01 05:00:00-09:00',
               '1997-01-01 06:00:00-09:00', '1997-01-01 07:00:00-09:00',
               '1997-01-01 08:00:00-09:00', '1997-01-01 09:00:00-09:00',
               '1997-01-01 10:00:00-09:00', '1997-01-01 11:00:00-09:00',
               '1997-01-01 12:00:00-09:00', '1997-01-01 13:00:00-09:00',
               '1997-01-01 14:00:00-09:00', '1997-01-01 15:00:00-09:00'],
              dtype='datetime64[ns, pytz.FixedOffset(-540)]', freq='H')

In [68]: ax = solar_position_hack.loc[solar_position_hack.index[0:24], ['apparent_
↪zenith', 'apparent_elevation', 'azimuth']].plot()

In [69]: ax.legend(loc=1);

In [70]: ax.axhline(0, color='darkgray');  # add 0 deg line for sunrise/sunset

In [71]: ax.axhline(180, color='darkgray');  # add 180 deg line for azimuth at solar␣
↪noon

In [72]: ax.set_ylim(-60, 200);  # zoom in, but cuts off full azimuth range

In [73]: ax.set_xlabel('Local time ({})'.format(solar_position_hack.index.tz));

In [74]: ax.set_ylabel('(degrees)');
```



Note that the time has been correctly localized and converted, however, the calculation bounds still correspond to the original assumed-UTC range.

For this and other reasons, we recommend that users supply time zone information at the beginning of a calculation rather than localizing and converting the results at the end of a calculation.

3.10 Clear sky

This section reviews the clear sky modeling capabilities of pvlib-python.

pvlib-python supports two ways to generate clear sky irradiance:

1. A *Location* object's *get_clearsky()* method.
2. The functions contained in the `clearsky` module, including *ineichen()* and *simplified_solis()*.

Users that work with simple time series data may prefer to use *get_clearsky()*, while users that want finer control, more explicit code, or work with multidimensional data may prefer to use the basic functions in the `clearsky` module.

The *Location* subsection demonstrates the easiest way to obtain a time series of clear sky data for a location. The *Ineichen and Perez* and *Simplified Solis* subsections detail the clear sky algorithms and input data. The *Detect Clearsky* subsection demonstrates the use of the clear sky detection algorithm.

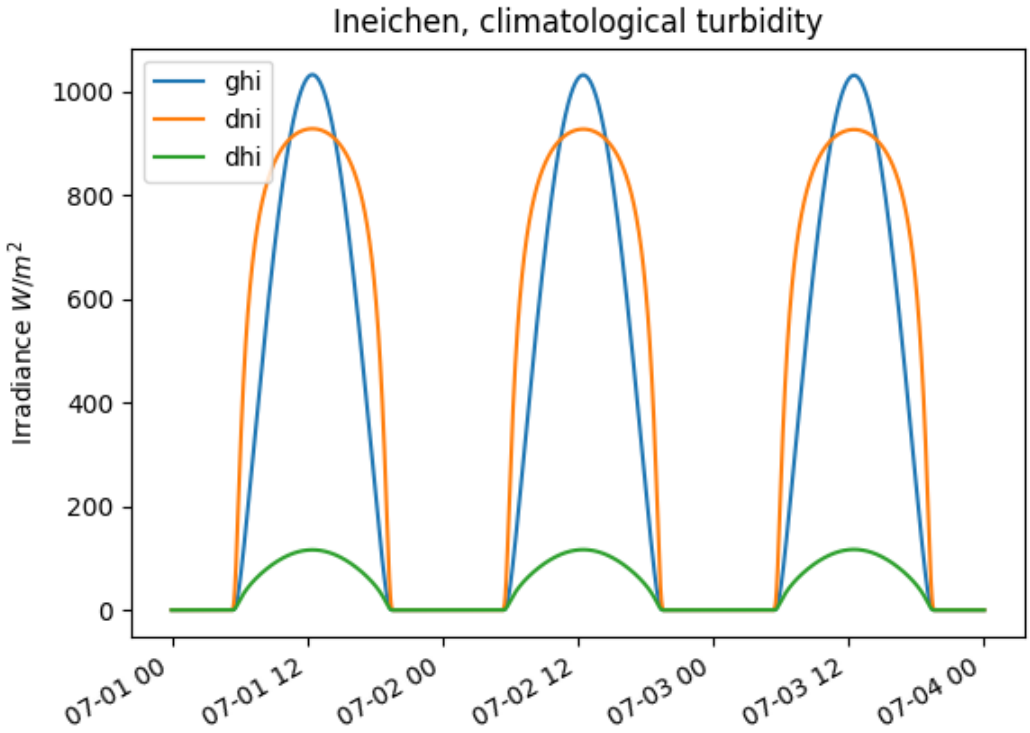
We'll need these imports for the examples below.

```
In [1]: import os

In [2]: import itertools

In [3]: import matplotlib.pyplot as plt

In [4]: import pandas as pd

In [5]: import pvlib

In [6]: from pvlib import clearsky, atmosphere, solarposition

In [7]: from pvlib.location import Location

In [8]: from pvlib.iotools import read_tmy3
```

3.10.1 Location

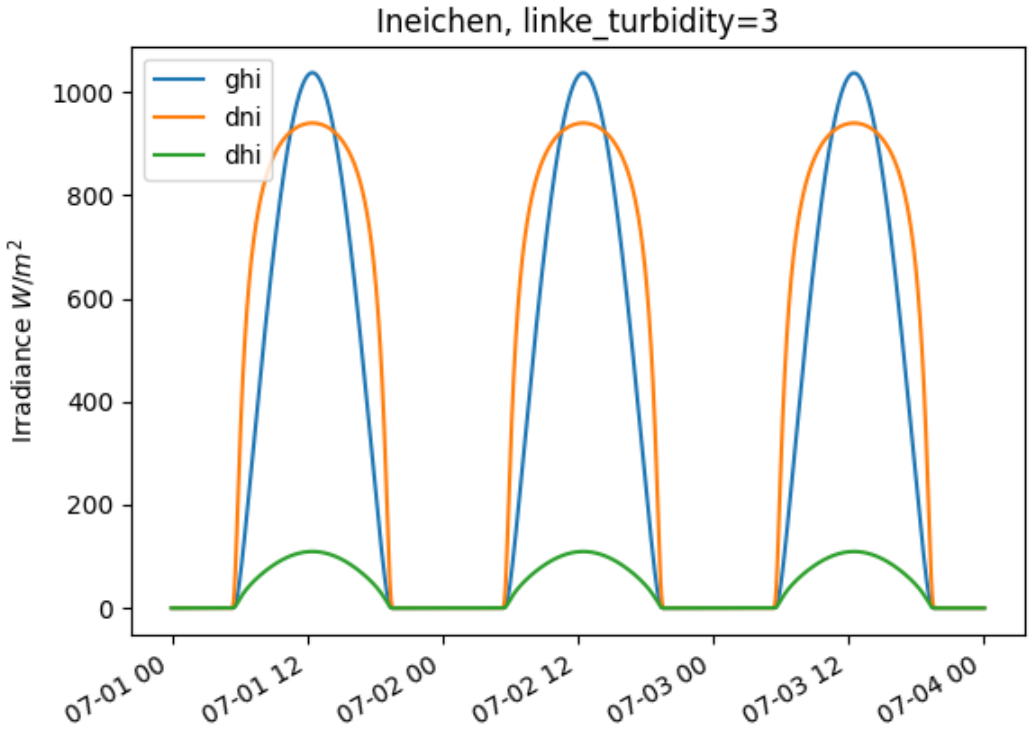
The easiest way to obtain a time series of clear sky irradiance is to use a *Location* object's *get_clearsky()* method. The *get_clearsky()* method does the dirty work of calculating solar position, extraterrestrial irradiance, airmass, and atmospheric pressure, as appropriate, leaving the user to only specify the most important parameters: time and atmospheric attenuation. The time input must be a pandas.DatetimeIndex, while the atmospheric attenuation inputs may be constants or arrays. The *get_clearsky()* method always returns a pandas.DataFrame.

```
In [9]: tus = Location(32.2, -111, 'US/Arizona', 700, 'Tucson')

In [10]: times = pd.date_range(start='2016-07-01', end='2016-07-04', freq='1min',␣
↪tz=tus.tz)

In [11]: cs = tus.get_clearsky(times)  # ineichen with climatology table by default

In [12]: cs.plot();

In [13]: plt.ylabel('Irradiance $W/m^2$');

In [14]: plt.title('Ineichen, climatological turbidity');
```

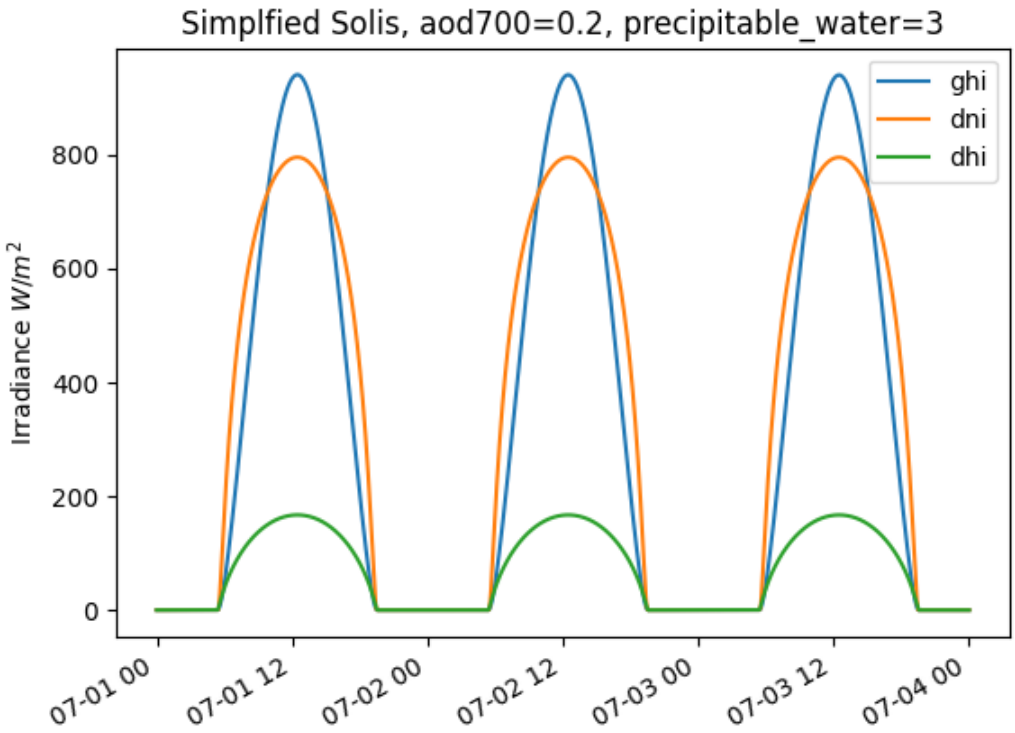


The *get_clearsky()* method accepts a model keyword argument and propagates additional arguments to the functions that do the computation.

```
In [15]: cs = tus.get_clearsky(times, model='ineichen', linke_turbidity=3)

In [16]: cs.plot();

In [17]: plt.title('Ineichen, linke_turbidity=3');

In [18]: plt.ylabel('Irradiance $W/m^2$');
```




```
In [19]: cs = tus.get_clearsky(times, model='simplified_solis', aod700=0.2,␣
↪precipitable_water=3)

In [20]: cs.plot();

In [21]: plt.title('Simplfied Solis, aod700=0.2, precipitable_water=3');

In [22]: plt.ylabel('Irradiance $W/m^2$');
```



See the sections below for more detail on the clear sky models.

3.10.2 Ineichen and Perez

The Ineichen and Perez clear sky model parameterizes irradiance in terms of the Linke turbidity [Ine02]. pvlib-python implements this model in the *pvlib.clearsky.ineichen()* function.

Turbidity data

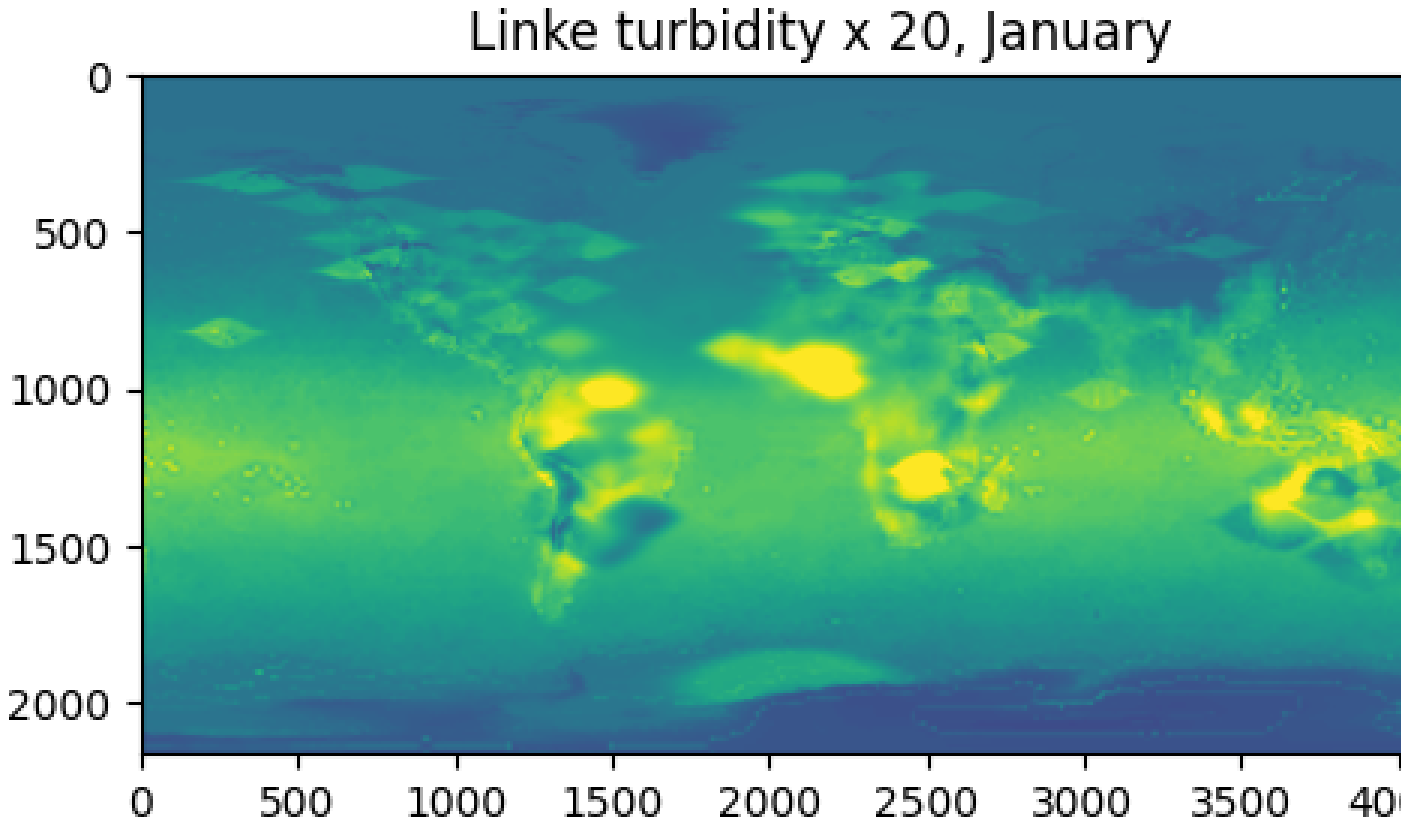
pvlib includes a file with monthly climatological turbidity values for the globe. The code below creates turbidity maps for a few months of the year. You could run it in a loop to create plots for all months.

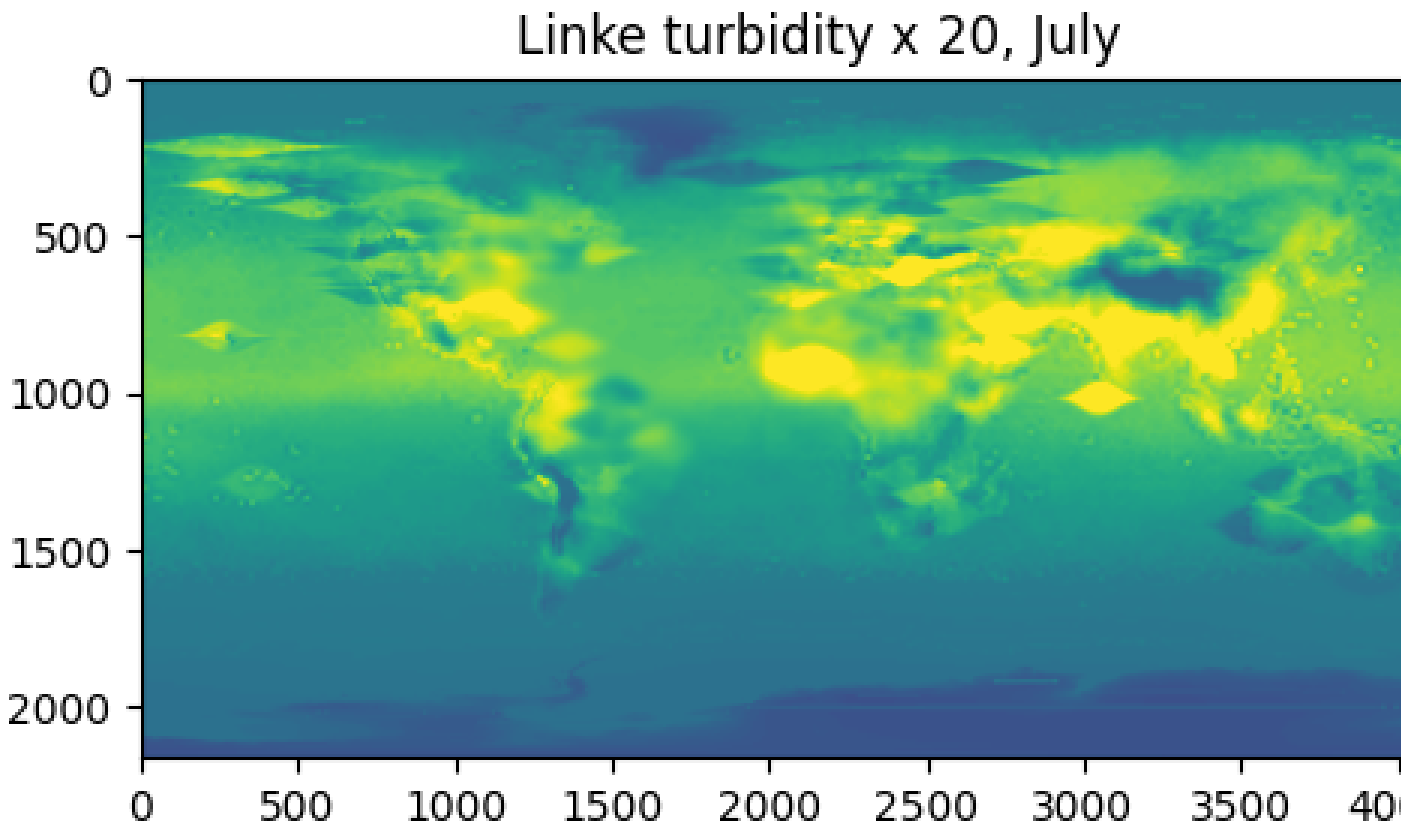
```
In [23]: import calendar

In [24]: import os

In [25]: import tables

In [26]: pvlib_path = os.path.dirname(os.path.abspath(pvlib.clearsky.__file__))

In [27]: filepath = os.path.join(pvlib_path, 'data', 'LinkeTurbidities.h5')

In [28]: def plot_turbidity_map(month, vmin=1, vmax=100):
   ....:     plt.figure();
   ....:     with tables.open_file(filepath) as lt_h5_file:
```

(continues on next page)

(continued from previous page)

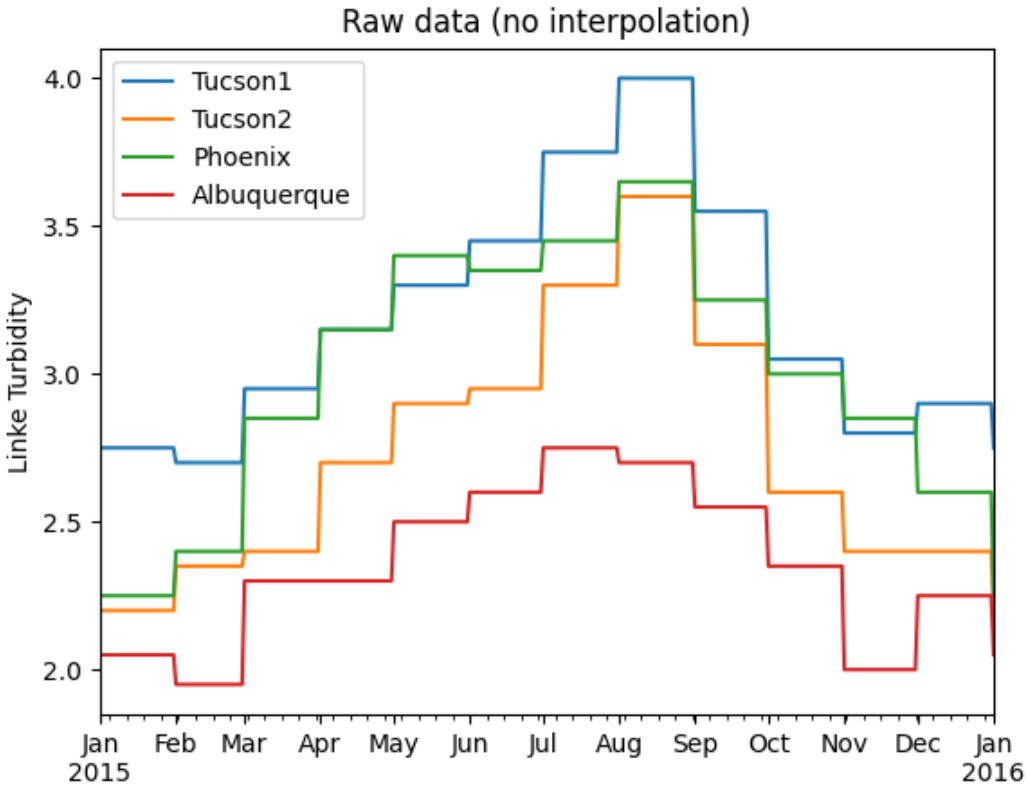
```
   ....:         ltdata = lt_h5_file.root.LinkeTurbidity[:, :, month-1]
   ....:     plt.imshow(ltdata, vmin=vmin, vmax=vmax);
   ....:     # data is in units of 20 x turbidity
   ....:     plt.title('Linke turbidity x 20, ' + calendar.month_name[month]);
   ....:     plt.colorbar(shrink=0.5);
   ....:     plt.tight_layout();
   ....:

In [29]: plot_turbidity_map(1)

In [30]: plot_turbidity_map(7)
```

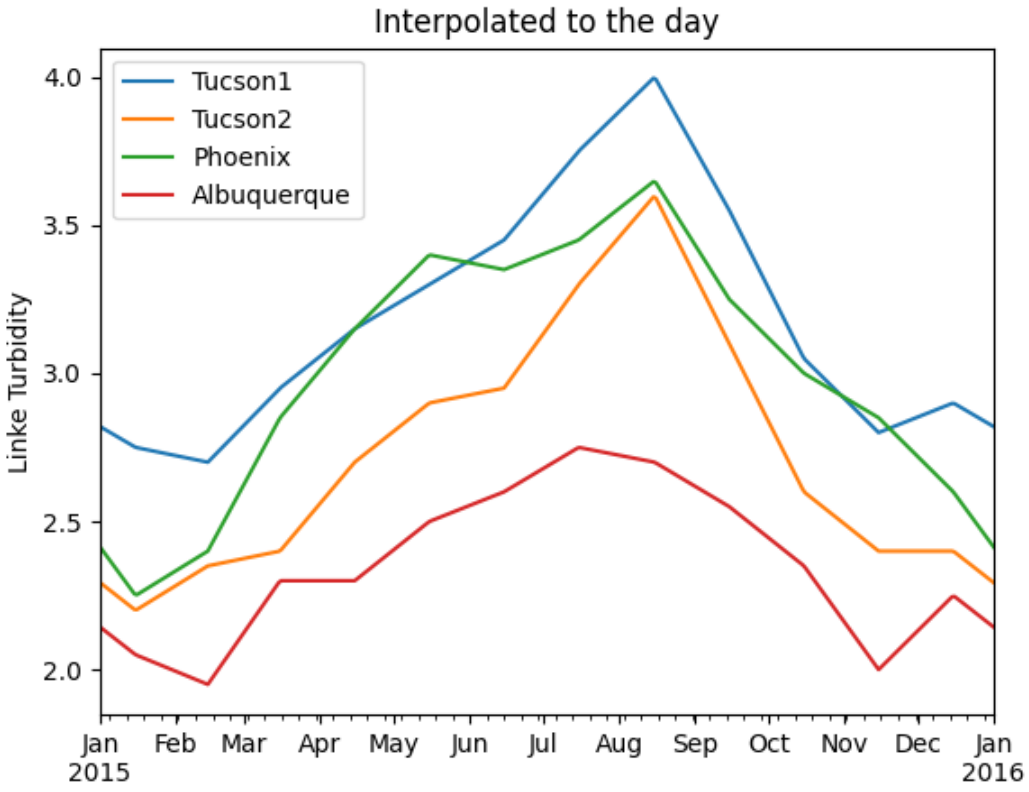

Linke turbidity x 20, January
0
500
1000
1500
2000
0
500
1000
1500
2000
2500
3000
3500



The `lookup_linke_turbidity()` function takes a time, latitude, and longitude and gets the corresponding climatological turbidity value for that time at those coordinates. By default, the `lookup_linke_turbidity()` function will linearly interpolate turbidity from month to month, assuming that the raw data is valid on 15th of each month. This interpolation removes discontinuities in multi-month PV models. Here's a plot of a few locations in the Southwest U.S. with and without interpolation. We chose points that are relatively close so that you can get a better sense of the spatial noise and variability of the data set. Note that the altitude of these sites varies from 300 m to 1500 m.

```
In [31]: times = pd.date_range(start='2015-01-01', end='2016-01-01', freq='1D')

In [32]: sites = [(32, -111, 'Tucson1'), (32.2, -110.9, 'Tucson2'),
   ....:          (33.5, -112.1, 'Phoenix'), (35.1, -106.6, 'Albuquerque')]
   ....:

In [33]: plt.figure();

In [34]: for lat, lon, name in sites:
   ....:     turbidity = pvlib.clearsky.lookup_linke_turbidity(times, lat, lon,␣
↪interp_turbidity=False)
   ....:     turbidity.plot(label=name)
   ....:

In [35]: plt.legend();

In [36]: plt.title('Raw data (no interpolation)');

In [37]: plt.ylabel('Linke Turbidity');

In [38]: plt.figure();

In [39]: for lat, lon, name in sites:
   ....:     turbidity = pvlib.clearsky.lookup_linke_turbidity(times, lat, lon)
   ....:     turbidity.plot(label=name)
   ....:

In [40]: plt.legend();

In [41]: plt.title('Interpolated to the day');

In [42]: plt.ylabel('Linke Turbidity');
```


Raw data (no interpolation)
Tucson1
Tucson2
Phoenix
Albuquerque
Linke Turbidity
4.0
3.5
3.0
2.5
2.0
Jan
2015
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
Jan
2016



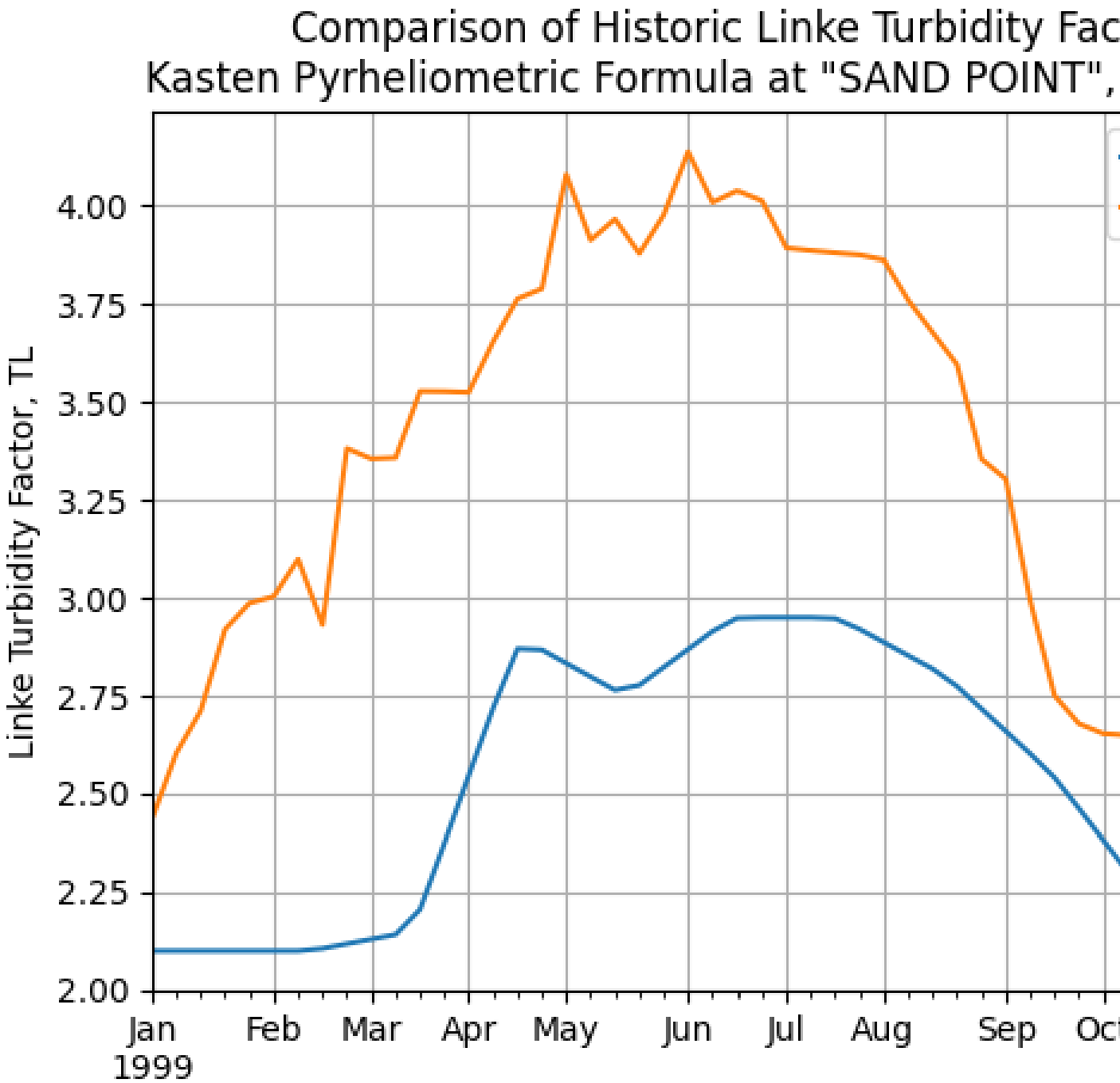
The *kasten96_lt()* function can be used to calculate Linke turbidity [Kas96] as input to the clear sky Ineichen and Perez function. The Kasten formulation requires precipitable water and broadband aerosol optical depth (AOD). According to Molineaux, broadband AOD can be approximated by a single measurement at 700-nm [Mol98]. An alternate broadband AOD approximation from Bird and Hulstrom combines AOD measured at two wavelengths [Bir80], and is implemented in *bird_hulstrom80_aod_bb()*.

```
In [43]: pvlib_data = os.path.join(os.path.dirname(pvlib.__file__), 'data')

In [44]: mbars = 100  # conversion factor from mbars to Pa

In [45]: tmy_file = os.path.join(pvlib_data, '703165TY.csv')  # TMY file

In [46]: tmy_data, tmy_header = read_tmy3(tmy_file, coerce_year=1999)  # read TMY data

In [47]: tl_historic = clearsky.lookup_linke_turbidity(time=tmy_data.index,
   ....:     latitude=tmy_header['latitude'], longitude=tmy_header['longitude'])
   ....:

In [48]: solpos = solarposition.get_solarposition(time=tmy_data.index,
   ....:     latitude=tmy_header['latitude'], longitude=tmy_header['longitude'],
   ....:     altitude=tmy_header['altitude'], pressure=tmy_data['Pressure']*mbars,
   ....:     temperature=tmy_data['DryBulb'])
   ....:

In [49]: am_rel = atmosphere.get_relative_airmass(solpos.apparent_zenith)

In [50]: am_abs = atmosphere.get_absolute_airmass(am_rel, tmy_data['Pressure']*mbars)
```

(continues on next page)

(continued from previous page)

```
In [51]: airmass = pd.concat([am_rel, am_abs], axis=1).rename(
   ....:     columns={0: 'airmass_relative', 1: 'airmass_absolute'})
   ....:

In [52]: tl_calculated = atmosphere.kasten96_lt(
   ....:     airmass.airmass_absolute, tmy_data['Pwat'], tmy_data['AOD'])
   ....:

In [53]: tl = pd.concat([tl_historic, tl_calculated], axis=1).rename(
   ....:     columns={0:'Historic', 1:'Calculated'})
   ....:

In [54]: tl.index = tmy_data.index.tz_convert(None)  # remove timezone

In [55]: tl.resample('W').mean().plot();

In [56]: plt.grid()

In [57]: plt.title('Comparison of Historic Linke Turbidity Factors vs. \n'
   ....:     'Kasten Pyrheliometric Formula at {name:s}, {state:s} ({usaf:d}TY)'.
↪format(
   ....:     name=tmy_header['Name'], state=tmy_header['State'], usaf=tmy_header['USAF
↪']));
   ....:

In [58]: plt.ylabel('Linke Turbidity Factor, TL');

In [59]: plt.tight_layout()
```



Examples

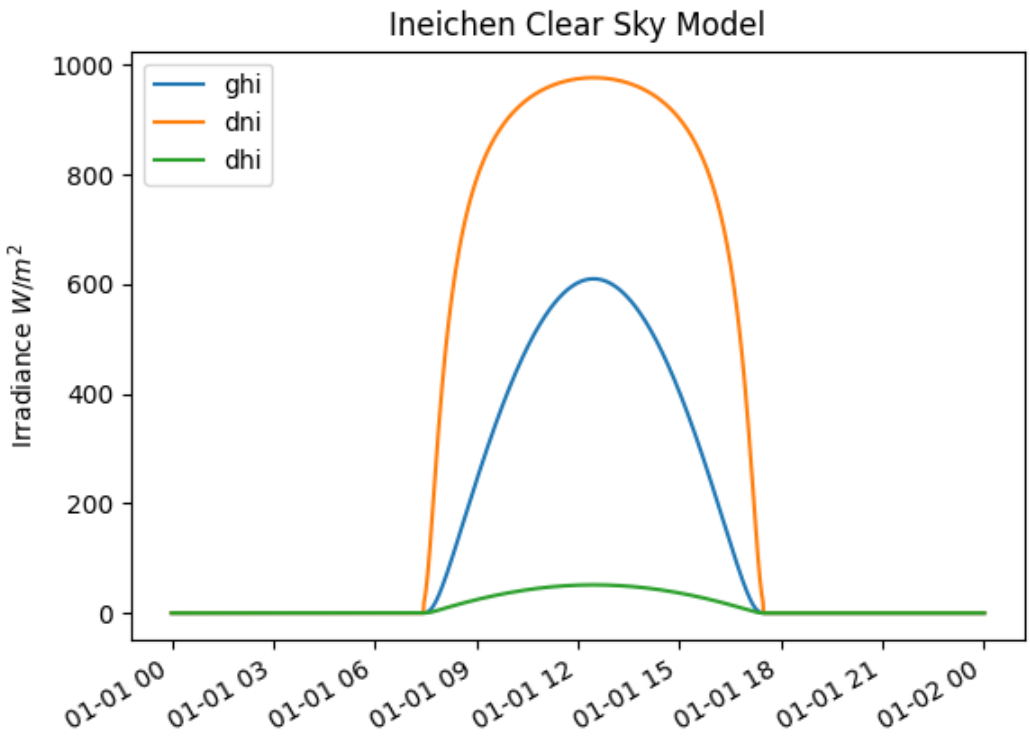
A clear sky time series using only basic pvlib functions.

```
In [60]: latitude, longitude, tz, altitude, name = 32.2, -111, 'US/Arizona', 700,
↪'Tucson'
```

(continues on next page)

(continued from previous page)

```
In [61]: times = pd.date_range(start='2014-01-01', end='2014-01-02', freq='1Min',␣
↪tz=tz)

In [62]: solpos = pvlib.solarposition.get_solarposition(times, latitude, longitude)

In [63]: apparent_zenith = solpos['apparent_zenith']

In [64]: airmass = pvlib.atmosphere.get_relative_airmass(apparent_zenith)

In [65]: pressure = pvlib.atmosphere.alt2pres(altitude)

In [66]: airmass = pvlib.atmosphere.get_absolute_airmass(airmass, pressure)

In [67]: linke_turbidity = pvlib.clearsky.lookup_linke_turbidity(times, latitude,␣
↪longitude)

In [68]: dni_extra = pvlib.irradiance.get_extra_radiation(times)

# an input is a pandas Series, so solis is a DataFrame
In [69]: ineichen = clearsky.ineichen(apparent_zenith, airmass, linke_turbidity,␣
↪altitude, dni_extra)

In [70]: plt.figure();

In [71]: ax = ineichen.plot()

In [72]: ax.set_ylabel('Irradiance $W/m^2$');

In [73]: ax.set_title('Ineichen Clear Sky Model');

In [74]: ax.legend(loc=2);
```



The input data types determine the returned output type. Array input results in an OrderedDict of array output, and Series input results in a DataFrame output. The keys are 'ghi', 'dni', and 'dhi'.

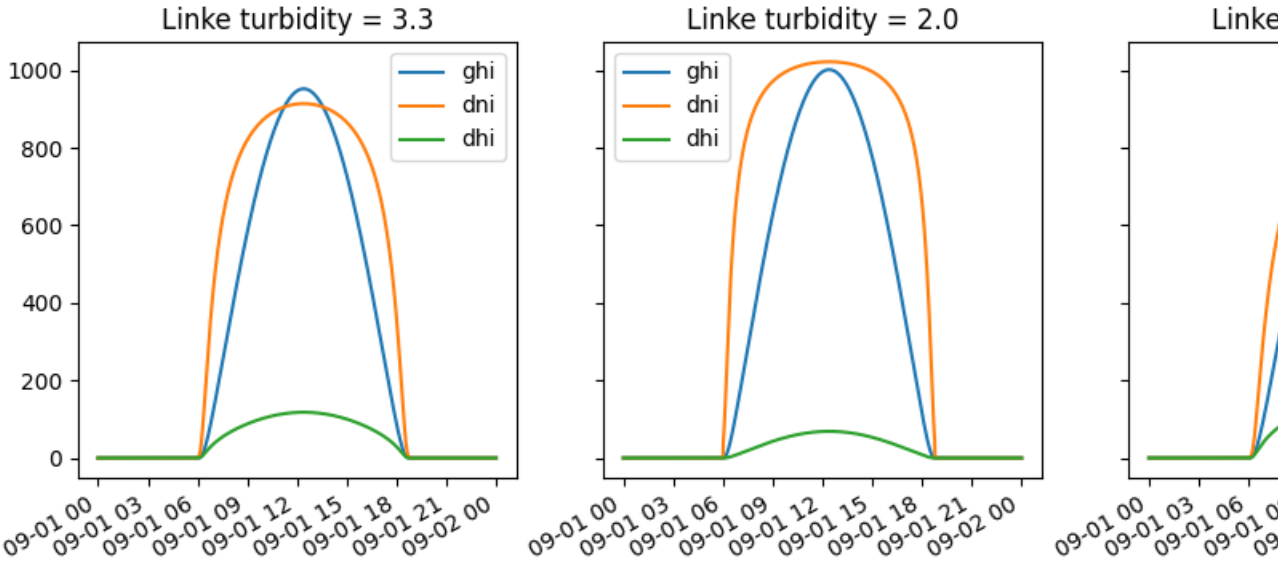
Grid with a clear sky irradiance for a few turbidity values.

```
In [75]: times = pd.date_range(start='2014-09-01', end='2014-09-02', freq='1Min',␣
↪tz=tz)

In [76]: solpos = pvlib.solarposition.get_solarposition(times, latitude, longitude)

In [77]: apparent_zenith = solpos['apparent_zenith']

In [78]: airmass = pvlib.atmosphere.get_relative_airmass(apparent_zenith)

In [79]: pressure = pvlib.atmosphere.alt2pres(altitude)

In [80]: airmass = pvlib.atmosphere.get_absolute_airmass(airmass, pressure)

In [81]: linke_turbidity = pvlib.clearsky.lookup_linke_turbidity(times, latitude,␣
↪longitude)

In [82]: print('climatological linke_turbidity = {}'.format(linke_turbidity.mean()))
climatological linke_turbidity = 3.329496820286459

In [83]: dni_extra = pvlib.irradiance.get_extra_radiation(times)

In [84]: linke_turbidities = [linke_turbidity.mean(), 2, 4]
```

(continues on next page)

(continued from previous page)

```
In [85]: fig, axes = plt.subplots(ncols=3, nrows=1, sharex=True, sharey=True,␣
↪squeeze=True, figsize=(12, 4))

In [86]: axes = axes.flatten()

In [87]: for linke_turbidity, ax in zip(linke_turbidities, axes):
   ....:     ineichen = clearsky.ineichen(apparent_zenith, airmass, linke_turbidity,␣
↪altitude, dni_extra)
   ....:     ineichen.plot(ax=ax, title='Linke turbidity = {:0.1f}'.format(linke_
↪turbidity));
   ....:

In [88]: ax.legend(loc=1);
```



Validation

See [Ine02], [Ren12].

Will Holmgren compared pvlib's Ineichen model and climatological turbidity to SoDa's McClear service in Arizona. Here are links to an ipynb notebook and its html rendering.

3.10.3 Simplified Solis

The Simplified Solis model parameterizes irradiance in terms of precipitable water and aerosol optical depth [Ine08ss]. pvlib-python implements this model in the *pvlib.clearsky.simplified_solis()* function.

Aerosol and precipitable water data

There are a number of sources for aerosol and precipitable water data of varying accuracy, global coverage, and temporal resolution. Ground based aerosol data can be obtained from Aeronet. Precipitable water can be obtained

from radiosondes, ESRL GPS-MET, or derived from surface relative humidity using functions such as *pvlib.atmosphere.gueymard94_pw()*. Numerous gridded products from satellites, weather models, and climate models contain one or both of aerosols and precipitable water. Consider data from the ECMWF and SoDa.

Aerosol optical depth (AOD) is a function of wavelength, and the Simplified Solis model requires AOD at 700 nm. *angstrom_aod_at_lambda()* is useful for converting AOD between different wavelengths using the Angstrom turbidity model. The Angstrom exponent, α, can be calculated from AOD at two wavelengths with *angstrom_alpha()*. [Ine08con], [Ine16], [Ang61].

```
In [89]: aod1240nm = 1.2  # fictitious AOD measured at 1240-nm

In [90]: aod550nm = 3.1  # fictitious AOD measured at 550-nm

In [91]: alpha_exponent = atmosphere.angstrom_alpha(aod1240nm, 1240, aod550nm, 550)

In [92]: aod700nm = atmosphere.angstrom_aod_at_lambda(aod1240nm, 1240, alpha_exponent,
    ↪ 700)

In [93]: aod380nm = atmosphere.angstrom_aod_at_lambda(aod550nm, 550, alpha_exponent,␣
    ↪380)

In [94]: aod500nm = atmosphere.angstrom_aod_at_lambda(aod550nm, 550, alpha_exponent,␣
    ↪500)

In [95]: aod_bb = atmosphere.bird_hulstrom80_aod_bb(aod380nm, aod500nm)

In [96]: print('compare AOD at 700-nm = {:g}, to estimated broadband AOD = {:g}, '
    ....:      'with alpha = {:g}'.format(aod700nm, aod_bb, alpha_exponent))
    ....:
compare AOD at 700-nm = 2.33931, to estimated broadband AOD = 2.52936, with alpha = 1.
    ↪16745
```

Examples

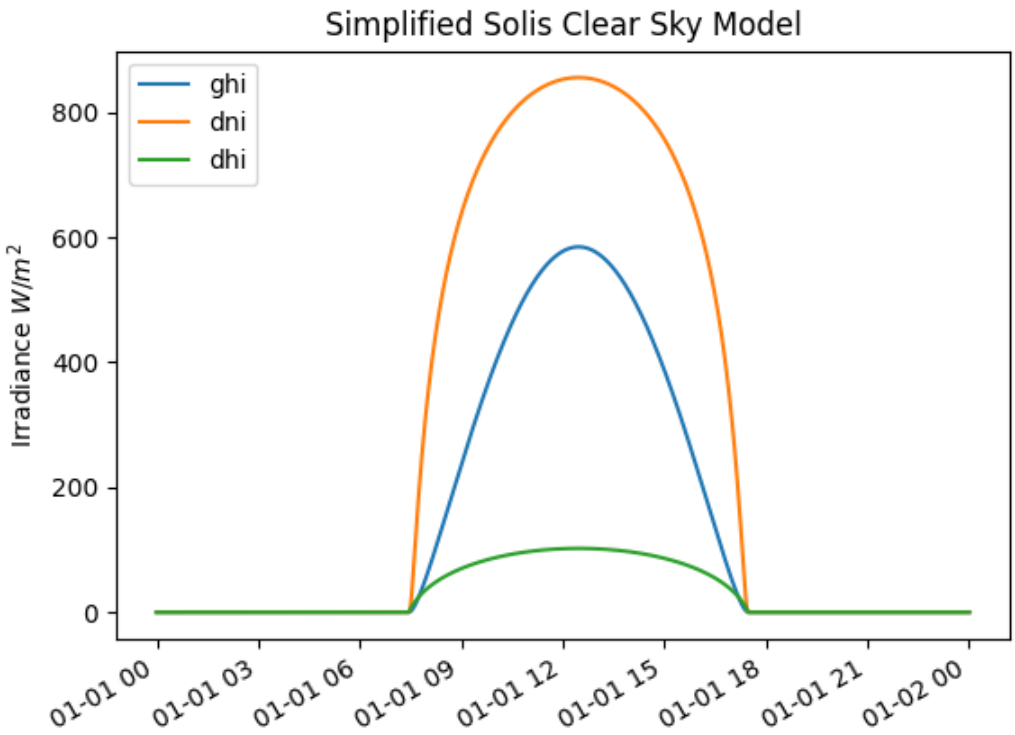
A clear sky time series using only basic pvlib functions.

```
In [97]: latitude, longitude, tz, altitude, name = 32.2, -111, 'US/Arizona', 700,
    ↪'Tucson'

In [98]: times = pd.date_range(start='2014-01-01', end='2014-01-02', freq='1Min',␣
    ↪tz=tz)

In [99]: solpos = pvlib.solarposition.get_solarposition(times, latitude, longitude)

In [100]: apparent_elevation = solpos['apparent_elevation']

In [101]: aod700 = 0.1

In [102]: precipitable_water = 1

In [103]: pressure = pvlib.atmosphere.alt2pres(altitude)

In [104]: dni_extra = pvlib.irradiance.get_extra_radiation(times)

# an input is a Series, so solis is a DataFrame
In [105]: solis = clearsky.simplified_solis(apparent_elevation, aod700, precipitable_
    ↪water,
```

(continues on next page)

(continued from previous page)

```
   .....:                                     pressure, dni_extra)
   .....:

In [106]: ax = solis.plot();

In [107]: ax.set_ylabel('Irradiance $W/m^2$');

In [108]: ax.set_title('Simplified Solis Clear Sky Model');

In [109]: ax.legend(loc=2);
```



The input data types determine the returned output type. Array input results in an OrderedDict of array output, and Series input results in a DataFrame output. The keys are 'ghi', 'dni', and 'dhi'.

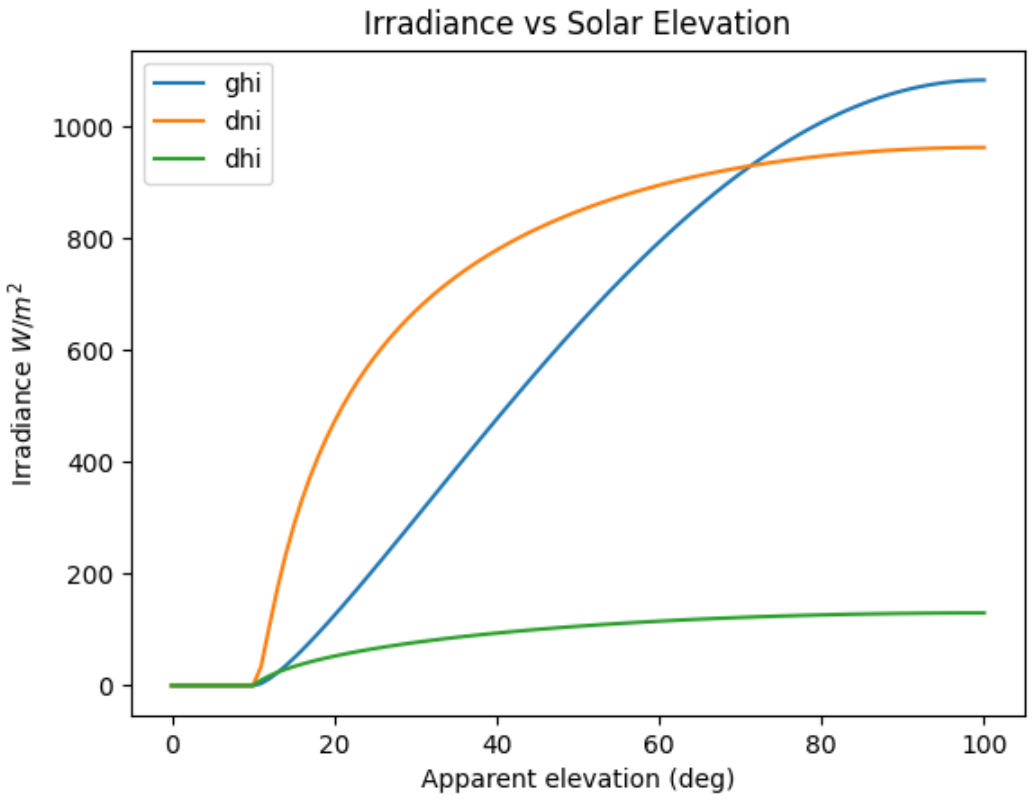
Irradiance as a function of solar elevation.

```
In [110]: apparent_elevation = pd.Series(np.linspace(-10, 90, 101))

In [111]: aod700 = 0.1

In [112]: precipitable_water = 1

In [113]: pressure = 101325

In [114]: dni_extra = 1364

In [115]: solis = clearsky.simplified_solis(apparent_elevation, aod700,
```

(continues on next page)

(continued from previous page)

```
   .....:                                     precipitable_water, pressure, dni_extra)
   .....:

In [116]: ax = solis.plot();

In [117]: ax.set_xlabel('Apparent elevation (deg)');

In [118]: ax.set_ylabel('Irradiance $W/m^2$');

In [119]: ax.set_title('Irradiance vs Solar Elevation')
Out[119]: Text(0.5, 1.0, 'Irradiance vs Solar Elevation')

In [120]: ax.legend(loc=2);
```



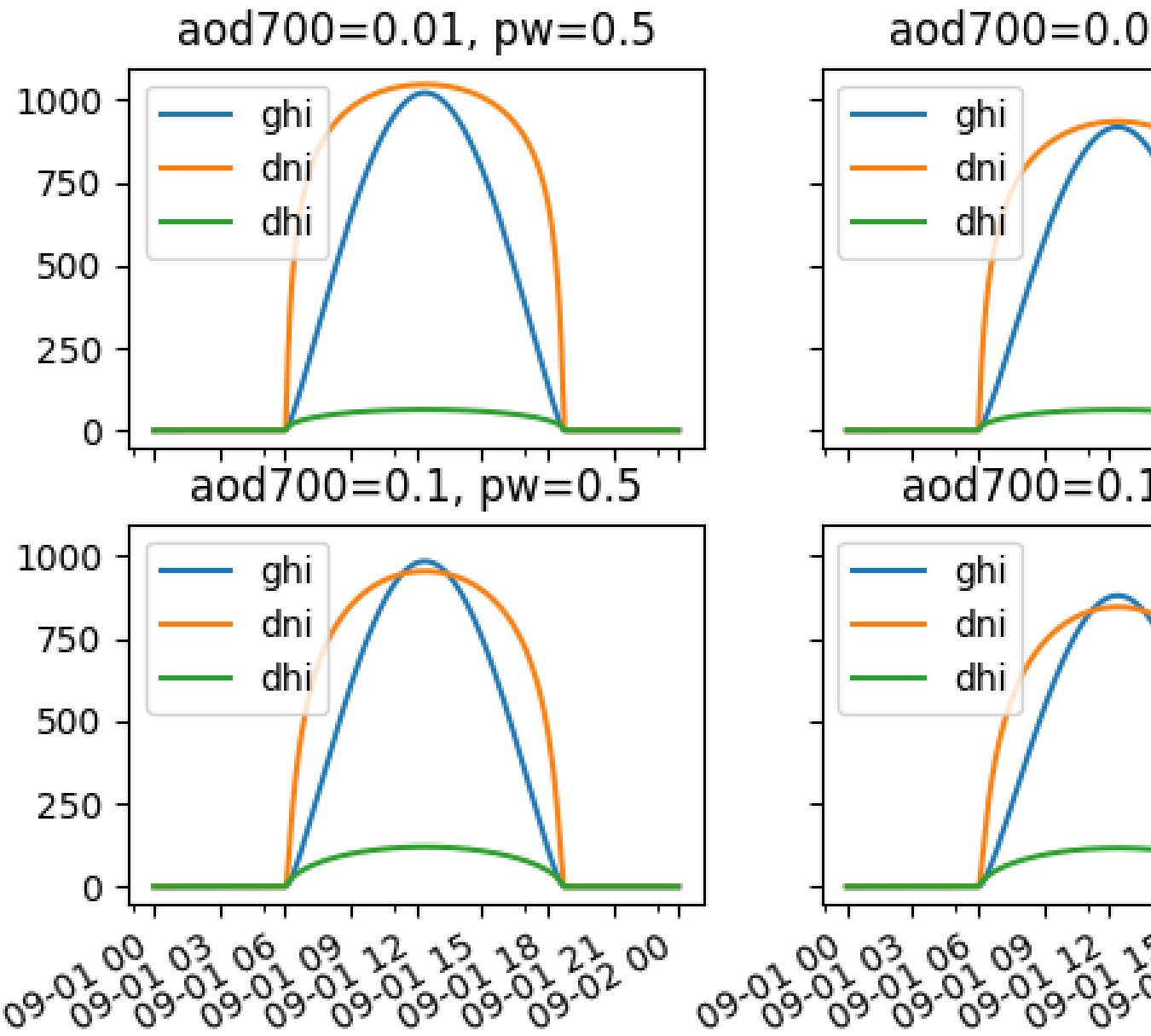
Grid with clear sky irradiance for a few PW and AOD values.

```
In [121]: times = pd.date_range(start='2014-09-01', end='2014-09-02', freq='1Min',␣
↪tz=tz)

In [122]: solpos = pvlib.solarposition.get_solarposition(times, latitude, longitude)

In [123]: apparent_elevation = solpos['apparent_elevation']

In [124]: pressure = pvlib.atmosphere.alt2pres(altitude)

In [125]: dni_extra = pvlib.irradiance.get_extra_radiation(times)
```

(continues on next page)

(continued from previous page)

```
In [126]: aod700 = [0.01, 0.1]

In [127]: precipitable_water = [0.5, 5]

In [128]: fig, axes = plt.subplots(ncols=2, nrows=2, sharex=True, sharey=True,␣
↪squeeze=True)

In [129]: axes = axes.flatten()

In [130]: for (aod, pw), ax in zip(itertools.chain(itertools.product(aod700,␣
↪precipitable_water)), axes):
   .....:     cs = clearsky.simplified_solis(apparent_elevation, aod, pw, pressure,␣
↪dni_extra)
   .....:     cs.plot(ax=ax, title='aod700={}, pw={}'.format(aod, pw))
   .....:
```



Contour plots of irradiance as a function of both PW and AOD.

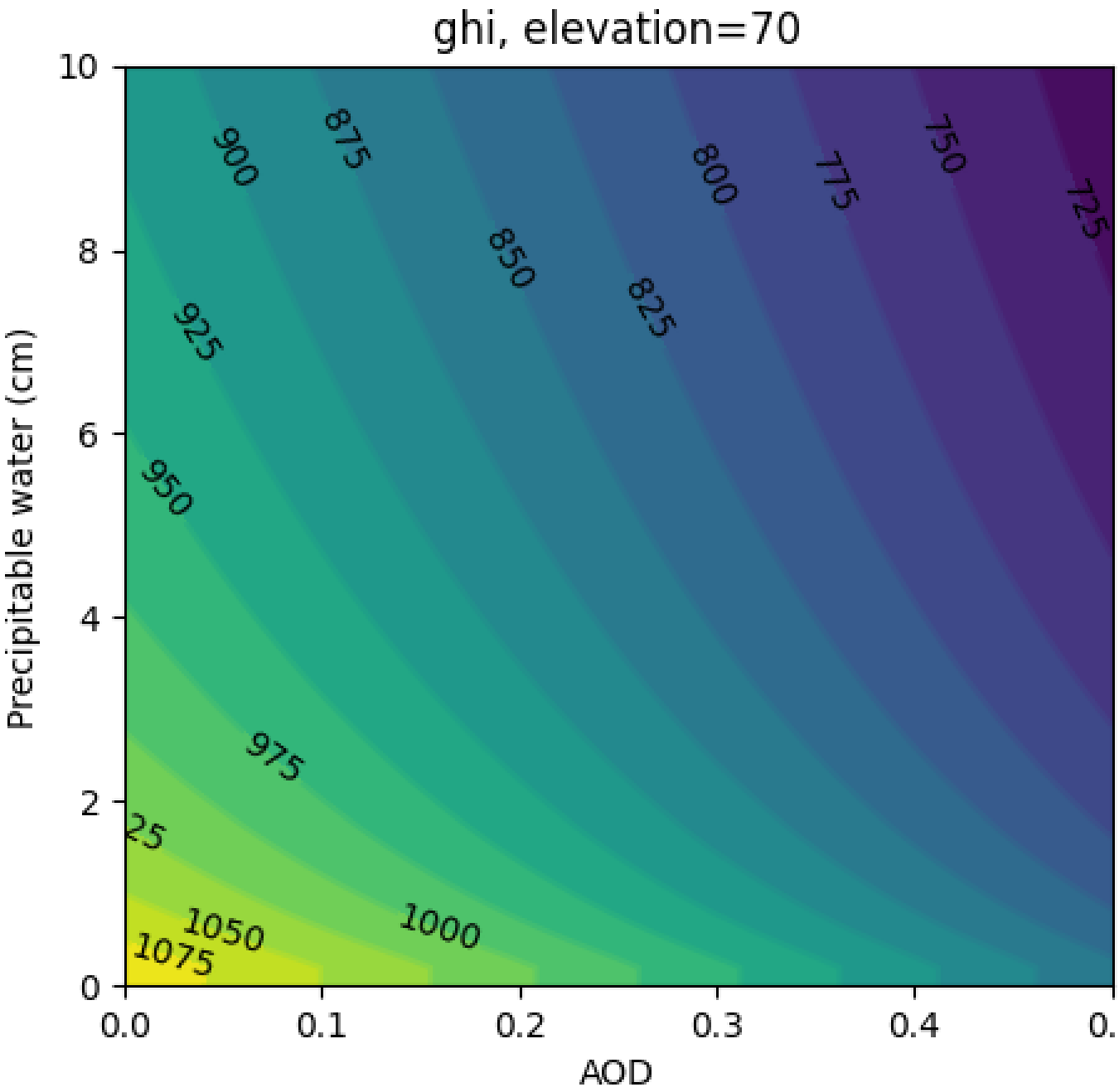
```
In [131]: aod700 = np.linspace(0, 0.5, 101)

In [132]: precipitable_water = np.linspace(0, 10, 101)

In [133]: apparent_elevation = 70
```

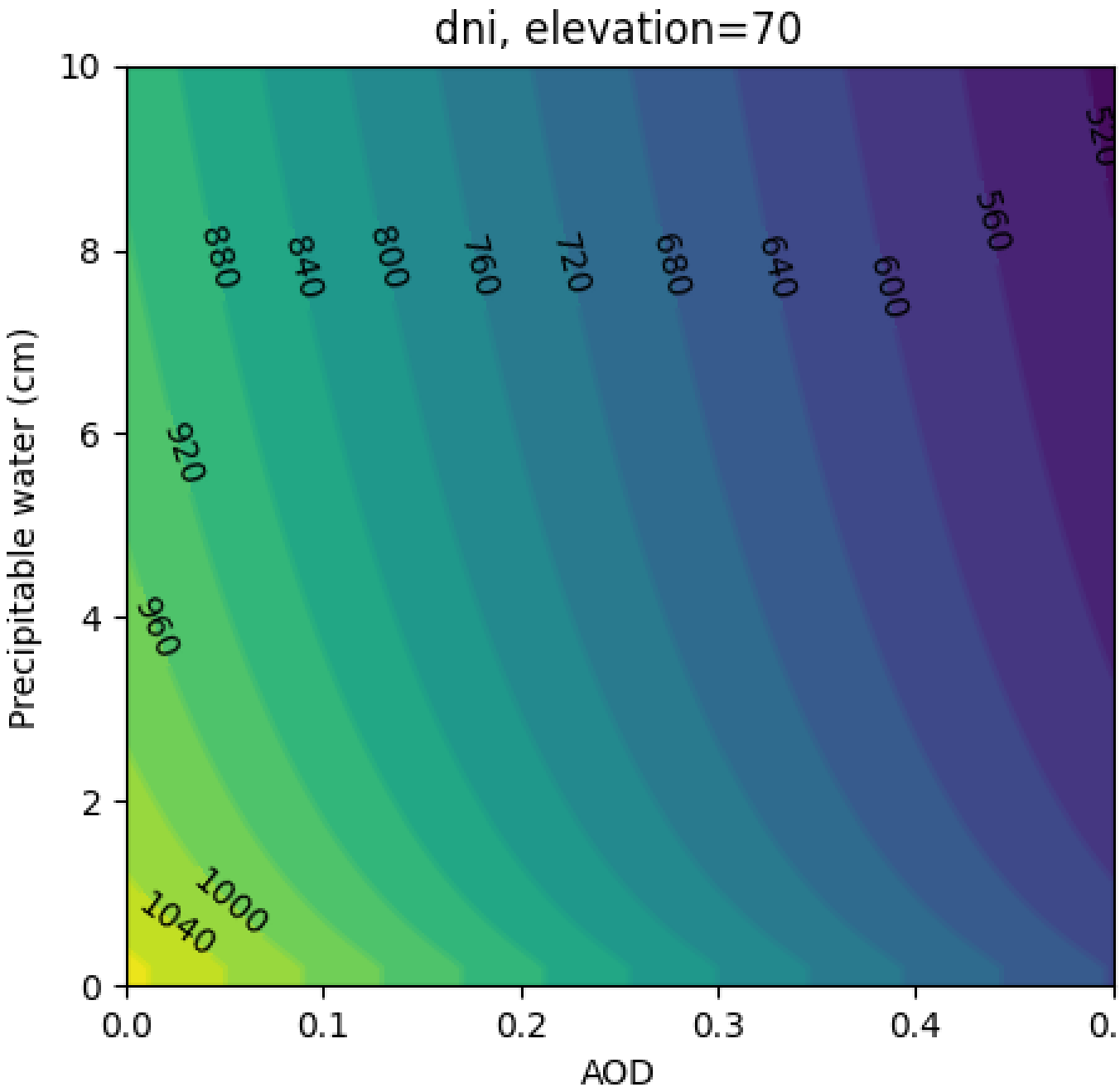
(continues on next page)

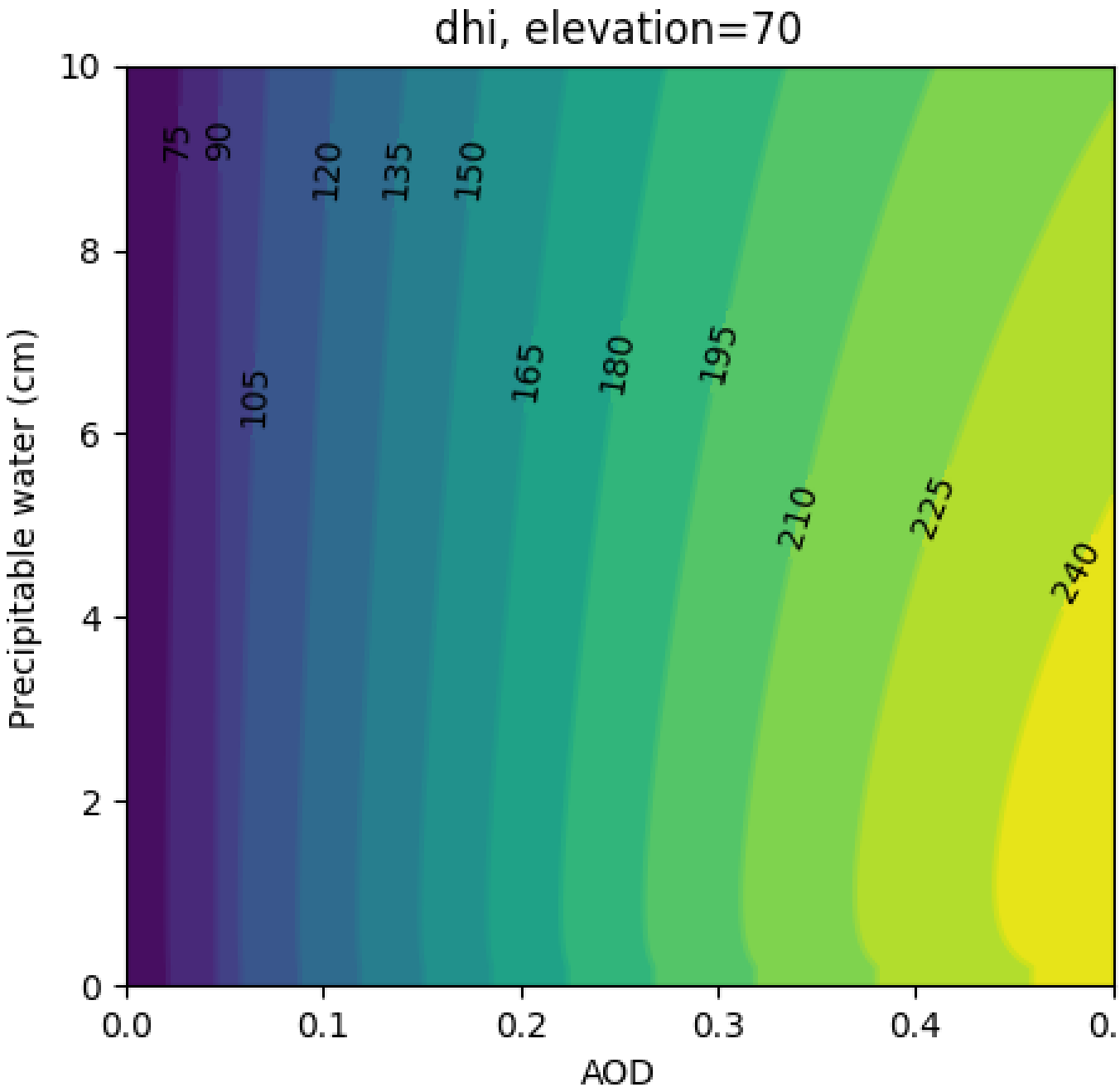
(continued from previous page)

```
In [134]: pressure = 101325

In [135]: dni_extra = 1364

In [136]: aod700, precipitable_water = np.meshgrid(aod700, precipitable_water)

# inputs are arrays, so solis is an OrderedDict
In [137]: solis = clearsky.simplified_solis(apparent_elevation, aod700,
   .....:                                   precipitable_water, pressure,
   .....:                                   dni_extra)
   .....:

In [138]: n = 15

In [139]: vmin = None

In [140]: vmax = None

In [141]: def plot_solis(key):
   .....:     irrad = solis[key]
   .....:     fig, ax = plt.subplots()
   .....:     im = ax.contour(aod700, precipitable_water, irrad[:, :], n, vmin=vmin,␣
↪vmax=vmax)
   .....:     imf = ax.contourf(aod700, precipitable_water, irrad[:, :], n, vmin=vmin,
↪ vmax=vmax)
   .....:     ax.set_xlabel('AOD')
   .....:     ax.set_ylabel('Precipitable water (cm)')
   .....:     ax.clabel(im, colors='k', fmt='%.0f')
   .....:     fig.colorbar(imf, label='{} (W/m**2)'.format(key))
   .....:     ax.set_title('{}, elevation={}'.format(key, apparent_elevation))
   .....:
```

```
In [142]: plot_solis('ghi')


In [143]: plot_solis('dni')


In [144]: plot_solis('dhi')
```


ghi, elevation=70
Precipitable water (cm)
AOD
10
8
6
4
2
0
0.0
0.1
0.2
0.3
0.4
900
875
850
825
800
775
750
725
925
950
975
25
1050
1075
1000


dni, elevation=70
Precipitable water (cm)
AOD
0
2
4
6
8
10
0.0
0.1
0.2
0.3
0.4
520
560
600
640
680
720
760
800
840
880
920
960
1000
1040



Validation

See [Ine16].

We encourage users to compare the pvlib implementation to Ineichen's Excel tool.

3.10.4 Detect Clearsky

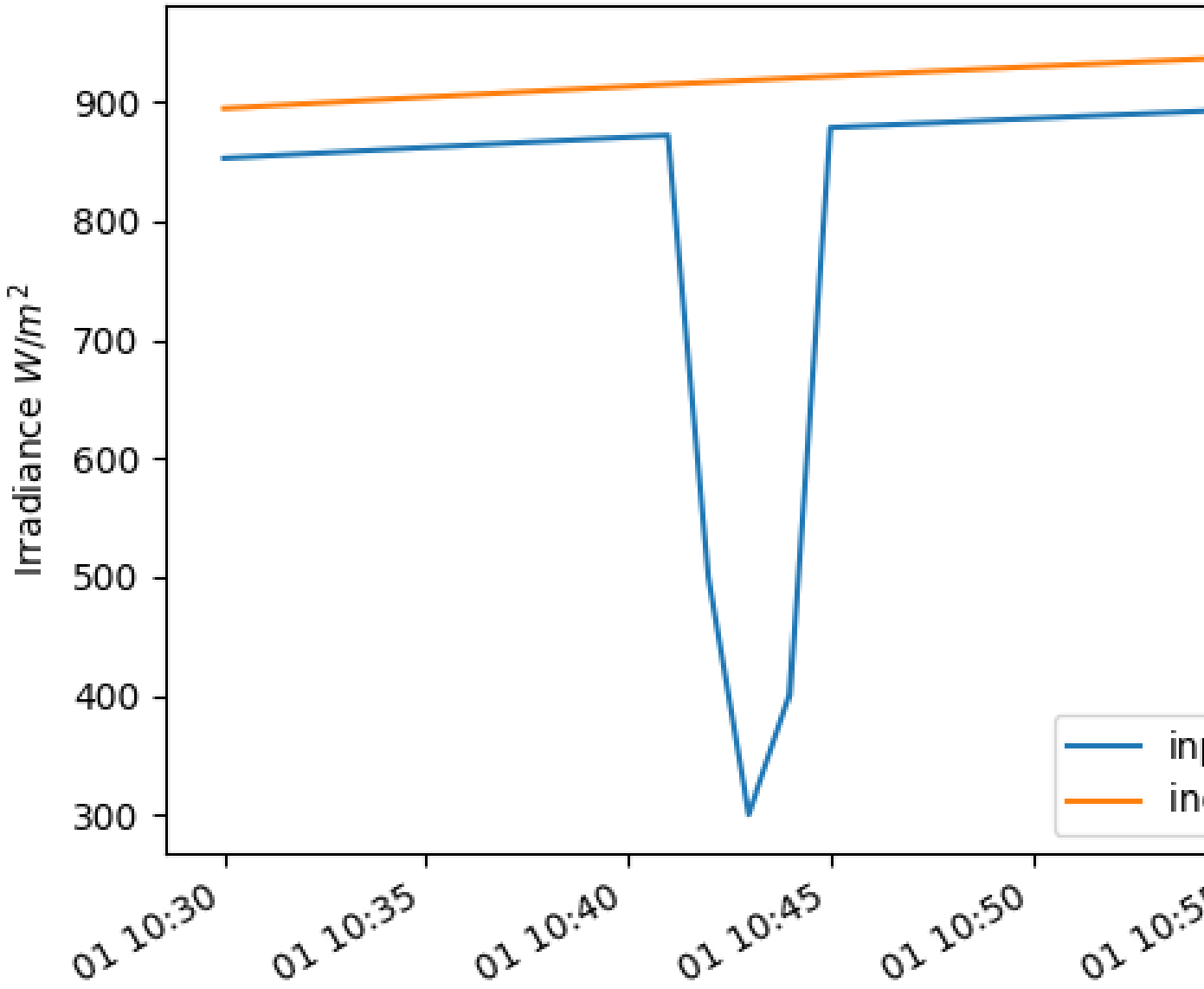
The *detect_clearsky()* function implements the [Ren16] algorithm to detect the clear and cloudy points of a time series. The algorithm was designed and validated for analyzing GHI time series only. Users may attempt to apply it to other types of time series data using different filter settings, but should be skeptical of the results.

The algorithm detects clear sky times by comparing statistics for a measured time series and an expected clearsky time series. Statistics are calculated using a sliding time window (e.g., 10 minutes). An iterative algorithm identifies clear periods, uses the identified periods to estimate bias in the clearsky data, scales the clearsky data and repeats.

Clear times are identified by meeting 5 criteria. Default values for these thresholds are appropriate for 10 minute windows of 1 minute GHI data.

Next, we show a simple example of applying the algorithm to synthetic GHI data. We first generate and plot the clear sky and measured data.

```
In [145]: abq = Location(35.04, -106.62, altitude=1619)

In [146]: times = pd.date_range(start='2012-04-01 10:30:00', tz='Etc/GMT+7',␣
↪periods=30, freq='1min')

In [147]: cs = abq.get_clearsky(times)

# scale clear sky data to account for possibility of different turbidity
In [148]: ghi = cs['ghi']*.953

# add a cloud event
In [149]: ghi['2012-04-01 10:42:00':'2012-04-01 10:44:00'] = [500, 300, 400]

# add an overirradiance event
In [150]: ghi['2012-04-01 10:56:00'] = 950

In [151]: fig, ax = plt.subplots()

In [152]: ghi.plot(label='input');

In [153]: cs['ghi'].plot(label='ineichen clear');

In [154]: ax.set_ylabel('Irradiance $W/m^2$');

In [155]: plt.legend(loc=4);
```



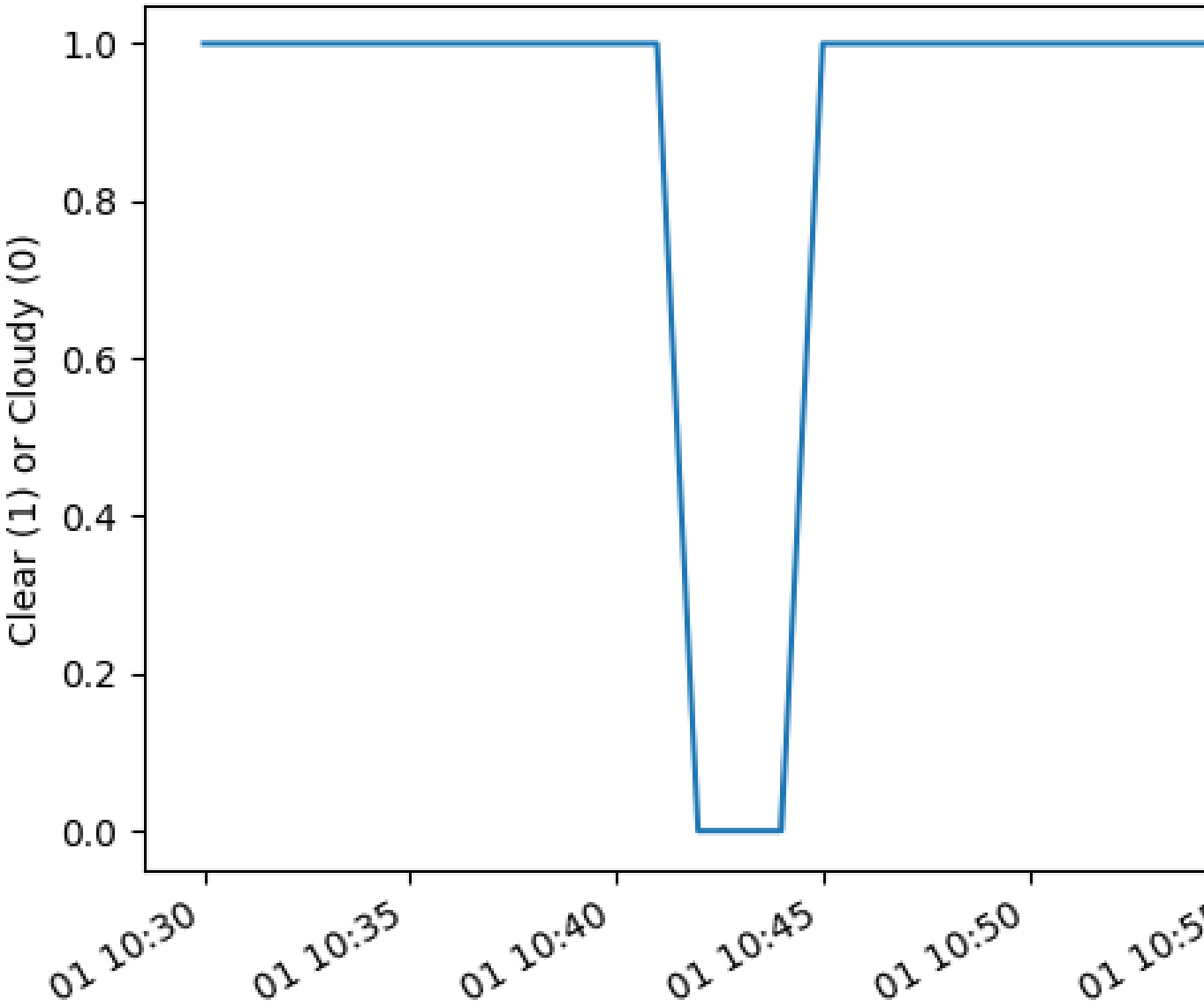
Now we run the synthetic data and clear sky estimate through the *detect_clearsky()* function.

```
In [156]: clear_samples = clearsky.detect_clearsky(ghi, cs['ghi'], cs.index, 10)

In [157]: fig, ax = plt.subplots()

In [158]: clear_samples.astype(int).plot();
```

(continues on next page)

(continued from previous page)

```
In [159]: ax.set_ylabel('Clear (1) or Cloudy (0)');
```



The algorithm detected the cloud event and the overirradiance event.

3.10.5 References

3.11 Forecasting

pvlib-python provides a set of functions and classes that make it easy to obtain weather forecast data and convert that data into a PV power forecast. Users can retrieve standardized weather forecast data relevant to PV power modeling from NOAA/NCEP/NWS models including the GFS, NAM, RAP, HRRR, and the NDFD. A PV power forecast can then be obtained using the weather data as inputs to the comprehensive modeling capabilities of PVLIB-Python. Standardized, open source, reference implementations of forecast methods using publicly available data may help advance the state-of-the-art of solar power forecasting.

pvlib-python uses Unidata's Siphon library to simplify access to real-time forecast data hosted on the Unidata THREDDS catalog. Siphon is great for programatic access of THREDDS data, but we also recommend using tools such as Panoply to easily browse the catalog and become more familiar with its contents.

We do not know of a similarly easy way to access archives of forecast data.

This document demonstrates how to use pvlib-python to create a PV power forecast using these tools. The forecast and forecast_to_power Jupyter notebooks provide additional example code.

Warning: The forecast module algorithms and features are highly experimental. The API may change, the functionality may be consolidated into an io module, or the module may be separated into its own package.

Note: This documentation is difficult to reliably build on readthedocs. If you do not see images, try building the documentation on your own machine or see the notebooks linked to above.

3.11.1 Accessing Forecast Data

The Siphon library provides access to, among others, forecasts from the Global Forecast System (GFS), North American Model (NAM), High Resolution Rapid Refresh (HRRR), Rapid Refresh (RAP), and National Digital Forecast Database (NDFD) on a Unidata THREDDS server. Unfortunately, many of these models use different names to describe the same quantity (or a very similar one), and not all variables are present in all models. For example, on the THREDDS server, the GFS has a field named `Total_cloud_cover_entire_atmosphere_Mixed_intervals_Average`, while the NAM has a field named `Total_cloud_cover_entire_atmosphere_single_layer`, and a similar field in the HRRR is named `Total_cloud_cover_entire_atmosphere`.

PVLIB-Python aims to simplify the access of the model fields relevant for solar power forecasts. Model data accessed with PVLIB-Python is returned as a pandas DataFrame with consistent column names: `temp_air`, `wind_speed`, `total_clouds`, `low_clouds`, `mid_clouds`, `high_clouds`, `dni`, `dhi`, `ghi`. To accomplish this, we use an object-oriented framework in which each weather model is represented by a class that inherits from a parent `ForecastModel` class. The parent `ForecastModel` class contains the common code for accessing and parsing the data using Siphon, while the child model-specific classes (*GFS*, *HRRR*, etc.) contain the code necessary to map and process that specific model's data to the standardized fields.

The code below demonstrates how simple it is to access and plot forecast data using PVLIB-Python. First, we set up make the basic imports and then set the location and time range data.

```
In [1]: import pandas as pd

In [2]: import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
In [3]: import datetime

# import pvlib forecast models
In [4]: from pvlib.forecast import GFS, NAM, NDFD, HRRR, RAP

# specify location (Tucson, AZ)
In [5]: latitude, longitude, tz = 32.2, -110.9, 'US/Arizona'

# specify time range.
In [6]: start = pd.Timestamp(datetime.date.today(), tz=tz)

In [7]: end = start + pd.Timedelta(days=7)

In [8]: irrad_vars = ['ghi', 'dni', 'dhi']
```

Next, we instantiate a GFS model object and get the forecast data from Unidata.

```
# GFS model, defaults to 0.5 degree resolution
# 0.25 deg available
In [9]: model = GFS()

# retrieve data. returns pandas.DataFrame object
In [10]: raw_data = model.get_data(latitude, longitude, start, end)

In [11]: print(raw_data.head())
                           Wind_speed_gust_surface  ...  u-component_of_wind_isobaric
2020-06-17 09:00:00-07:00                 4.515094  ...                      1.288459
2020-06-17 12:00:00-07:00                 3.351485  ...                     -0.223870
2020-06-17 15:00:00-07:00                 5.920943  ...                     -0.125232
2020-06-17 18:00:00-07:00                13.021213  ...                      6.285261
2020-06-17 21:00:00-07:00                11.483777  ...                      7.605815

[5 rows x 11 columns]
```

It will be useful to process this data before using it with pvlib. For example, the column names are non-standard, the temperature is in Kelvin, the wind speed is broken into east/west and north/south components, and most importantly, most of the irradiance data is missing. The forecast module provides a number of methods to fix these problems.

```
In [12]: data = raw_data

# rename the columns according the key/value pairs in model.variables.
In [13]: data = model.rename(data)

# convert temperature
In [14]: data['temp_air'] = model.kelvin_to_celsius(data['temp_air'])

# convert wind components to wind speed
In [15]: data['wind_speed'] = model.uv_to_speed(data)

# calculate irradiance estimates from cloud cover.
# uses a cloud_cover to ghi to dni model or a
# uses a cloud cover to transmittance to irradiance model.
# this step is discussed in more detail in the next section
In [16]: irrad_data = model.cloud_cover_to_irradiance(data['total_clouds'])
```

(continues on next page)

(continued from previous page)

```
In [17]: data = data.join(irrad_data, how='outer')

# keep only the final data
In [18]: data = data[model.output_variables]

In [19]: print(data.head())
                           temp_air  wind_speed  ...  mid_clouds  high_clouds
2020-06-17 09:00:00-07:00  20.652344    4.519079  ...         0.0          0.0
2020-06-17 12:00:00-07:00  18.554626    3.344021  ...         0.0          0.0
2020-06-17 15:00:00-07:00  33.791962    4.076262  ...         0.0          0.0
2020-06-17 18:00:00-07:00  47.350037   11.147929  ...         0.0          0.0
2020-06-17 21:00:00-07:00  50.227295   10.875139  ...         0.0          0.0

[5 rows x 9 columns]
```

Much better.

The GFS class's `process_data()` method combines these steps in a single function. In fact, each forecast model class implements its own `process_data` method since the data from each weather model is slightly different. The `process_data` functions are designed to be explicit about how the data is being processed, and users are **strongly** encouraged to read the source code of these methods.

```
In [20]: data = model.process_data(raw_data)

In [21]: print(data.head())
                           temp_air  wind_speed  ...  mid_clouds  high_clouds
2020-06-17 09:00:00-07:00  20.652344    4.519079  ...         0.0          0.0
2020-06-17 12:00:00-07:00  18.554626    3.344021  ...         0.0          0.0
2020-06-17 15:00:00-07:00  33.791962    4.076262  ...         0.0          0.0
2020-06-17 18:00:00-07:00  47.350037   11.147929  ...         0.0          0.0
2020-06-17 21:00:00-07:00  50.227295   10.875139  ...         0.0          0.0

[5 rows x 9 columns]
```

Users can easily implement their own `process_data` methods on inherited classes or implement similar stand-alone functions.

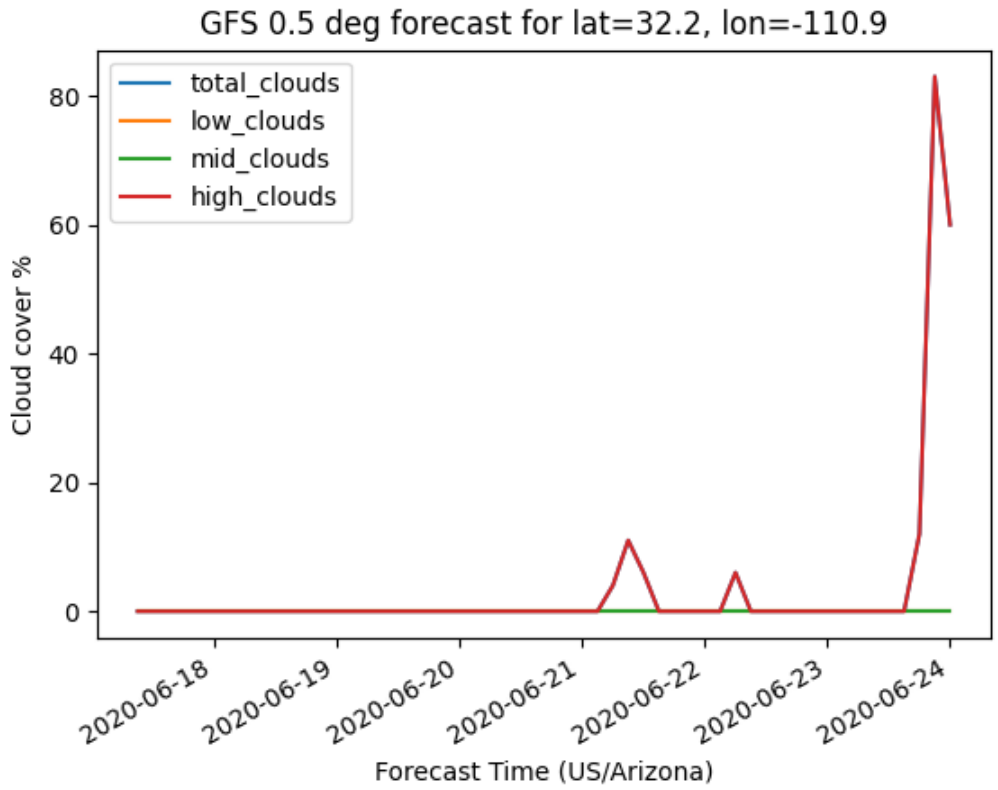
The forecast model classes also implement a *`get_processed_data()`* method that combines the *`get_data()`* and *`process_data()`* calls.

```
In [22]: data = model.get_processed_data(latitude, longitude, start, end)

In [23]: print(data.head())
                           temp_air  wind_speed  ...  mid_clouds  high_clouds
2020-06-17 09:00:00-07:00  20.652344    4.519079  ...         0.0          0.0
2020-06-17 12:00:00-07:00  18.554626    3.344021  ...         0.0          0.0
2020-06-17 15:00:00-07:00  33.791962    4.076262  ...         0.0          0.0
2020-06-17 18:00:00-07:00  47.350037   11.147929  ...         0.0          0.0
2020-06-17 21:00:00-07:00  50.227295   10.875139  ...         0.0          0.0

[5 rows x 9 columns]
```

3.11.2 Cloud cover and radiation

All of the weather models currently accessible by pvlib include one or more cloud cover forecasts. For example, below we plot the GFS cloud cover forecasts.

```
# plot cloud cover percentages
In [24]: cloud_vars = ['total_clouds', 'low_clouds',
   ....:               'mid_clouds', 'high_clouds']
   ....:

In [25]: data[cloud_vars].plot();

In [26]: plt.ylabel('Cloud cover %');

In [27]: plt.xlabel('Forecast Time ({})'.format(tz));

In [28]: plt.title('GFS 0.5 deg forecast for lat={}, lon={}'
   ....:           .format(latitude, longitude));
   ....:

In [29]: plt.legend();
```



However, many of forecast models do not include radiation components in their output fields, or if they do then the radiation fields suffer from poor solar position or radiative transfer algorithms. It is often more accurate to create empirically derived radiation forecasts from the weather models' cloud cover forecasts.

PVLIB-Python provides two basic ways to convert cloud cover forecasts to irradiance forecasts. One method assumes a linear relationship between cloud cover and GHI, applies the scaling to a clear sky climatology, and then uses the DISC model to calculate DNI. The second method assumes a linear relationship between cloud cover and atmospheric transmittance, and then uses the Liu-Jordan [Liu60] model to calculate GHI, DNI, and DHI.

Caveat emptor: these algorithms are not rigorously verified! The purpose of the forecast module is to provide a few exceedingly simple options for users to play with before they develop their own models. We strongly encourage pvlib

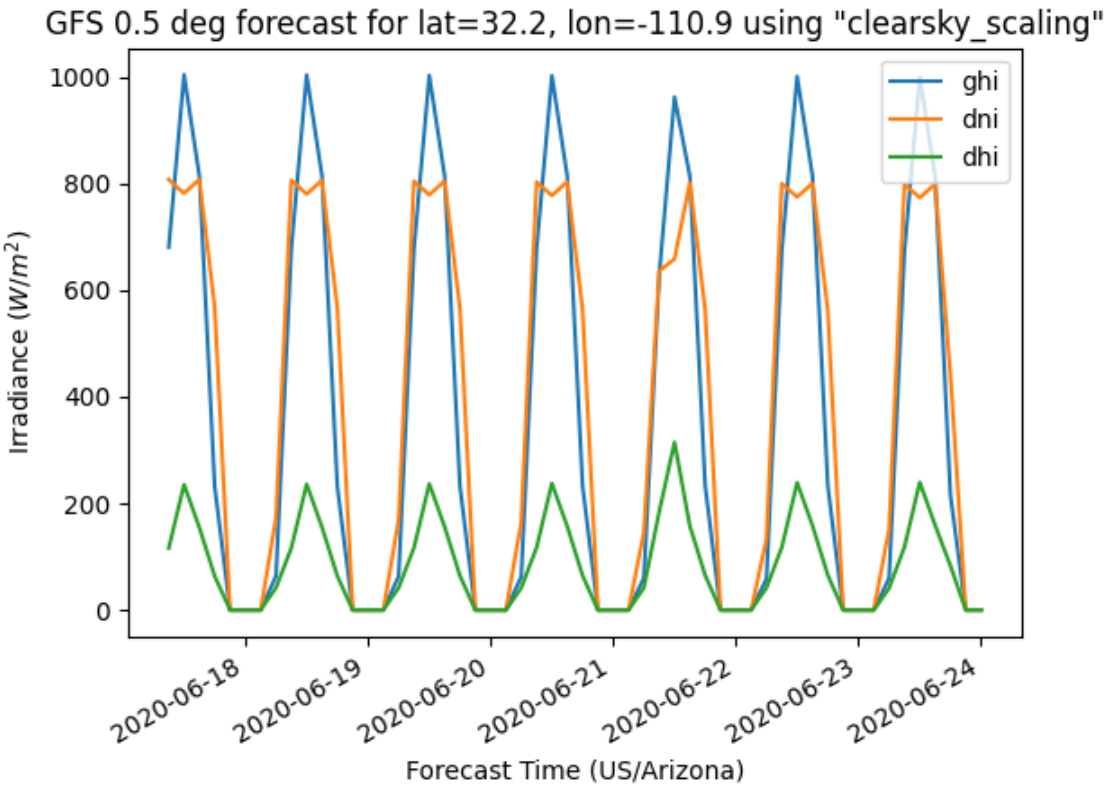
users first read the source code and second to implement new cloud cover to irradiance algorithms.

The essential parts of the clear sky scaling algorithm are as follows. Clear sky scaling of climatological GHI is also used in Larson et. al. [Lar16].

```
solpos = location.get_solarposition(cloud_cover.index)
cs = location.get_clearsky(cloud_cover.index, model='ineichen')
# offset and cloud cover in decimal units here
# larson et. al. use offset = 0.35
ghi = (offset + (1 - offset) * (1 - cloud_cover)) * ghi_clear
dni = disc(ghi, solpos['zenith'], cloud_cover.index)['dni']
dhi = ghi - dni * np.cos(np.radians(solpos['zenith']))
```

The figure below shows the result of the total cloud cover to irradiance conversion using the clear sky scaling algorithm.

```
# plot irradiance data
In [30]: data = model.rename(raw_data)

In [31]: irrads = model.cloud_cover_to_irradiance(data['total_clouds'], how='clearsky_
↪scaling')

In [32]: irrads.plot();

In [33]: plt.ylabel('Irradiance ($W/m^2$)');

In [34]: plt.xlabel('Forecast Time ({})'.format(tz));

In [35]: plt.title('GFS 0.5 deg forecast for lat={}, lon={} using "clearsky_scaling"'
   ....:           .format(latitude, longitude));
   ....:

In [36]: plt.legend();
```

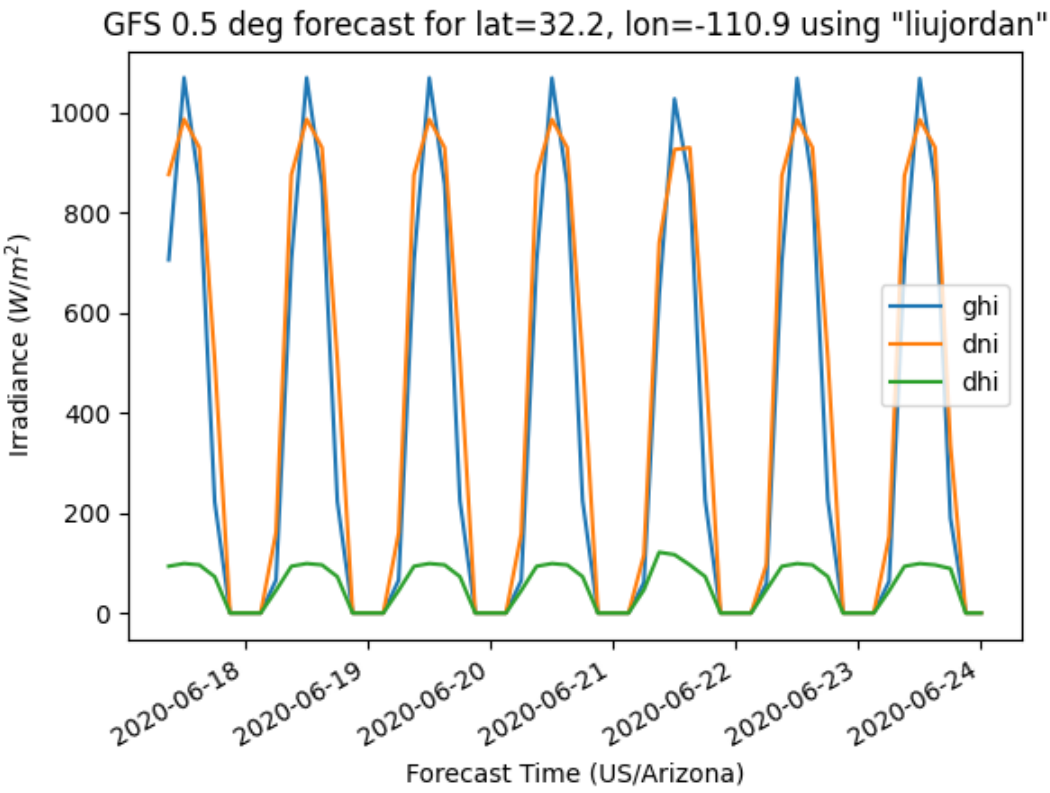


The essential parts of the Liu-Jordan cloud cover to irradiance algorithm are as follows.

```
# cloud cover in percentage units here
transmittance = ((100.0 - cloud_cover) / 100.0) * 0.75
# irrads is a DataFrame containing ghi, dni, dhi
irrads = liujordan(apparent_zenith, transmittance, airmass_absolute)
```

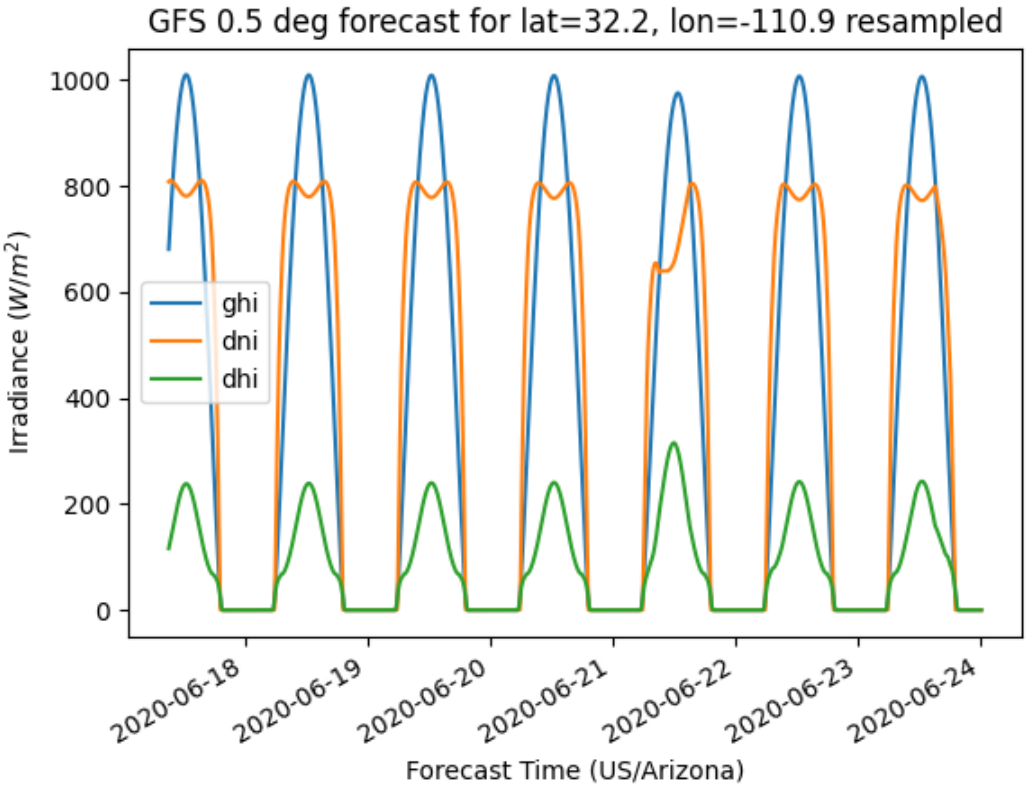
The figure below shows the result of the Liu-Jordan total cloud cover to irradiance conversion.

```
# plot irradiance data
In [37]: irrads = model.cloud_cover_to_irradiance(data['total_clouds'], how='liujordan
↪')

In [38]: irrads.plot();

In [39]: plt.ylabel('Irradiance ($W/m^2$)');

In [40]: plt.xlabel('Forecast Time ({})'.format(tz));

In [41]: plt.title('GFS 0.5 deg forecast for lat={}, lon={} using "liujordan"'
   ....:           .format(latitude, longitude));
   ....:

In [42]: plt.legend();
```



Most weather model output has a fairly coarse time resolution, at least an hour. The irradiance forecasts have the same time resolution as the weather data. However, it is straightforward to interpolate the cloud cover forecasts onto a higher resolution time domain, and then recalculate the irradiance.

```
In [43]: resampled_data = data.resample('5min').interpolate()

In [44]: resampled_irrads = model.cloud_cover_to_irradiance(resampled_data['total_
↪clouds'], how='clearsky_scaling')

In [45]: resampled_irrads.plot();

In [46]: plt.ylabel('Irradiance ($W/m^2$)');

In [47]: plt.xlabel('Forecast Time ({})'.format(tz));

In [48]: plt.title('GFS 0.5 deg forecast for lat={}, lon={} resampled'
   ....:           .format(latitude, longitude));
   ....:

In [49]: plt.legend();
```



Users may then recombine resampled_irrads and resampled_data using slicing `pandas.concat()` or `pandas.DataFrame.join()`.

We reiterate that the open source code enables users to customize the model processing to their liking.

3.11.3 Weather Models

Next, we provide a brief description of the weather models available to pvlib users. Note that the figures are generated when this documentation is compiled so they will vary over time.

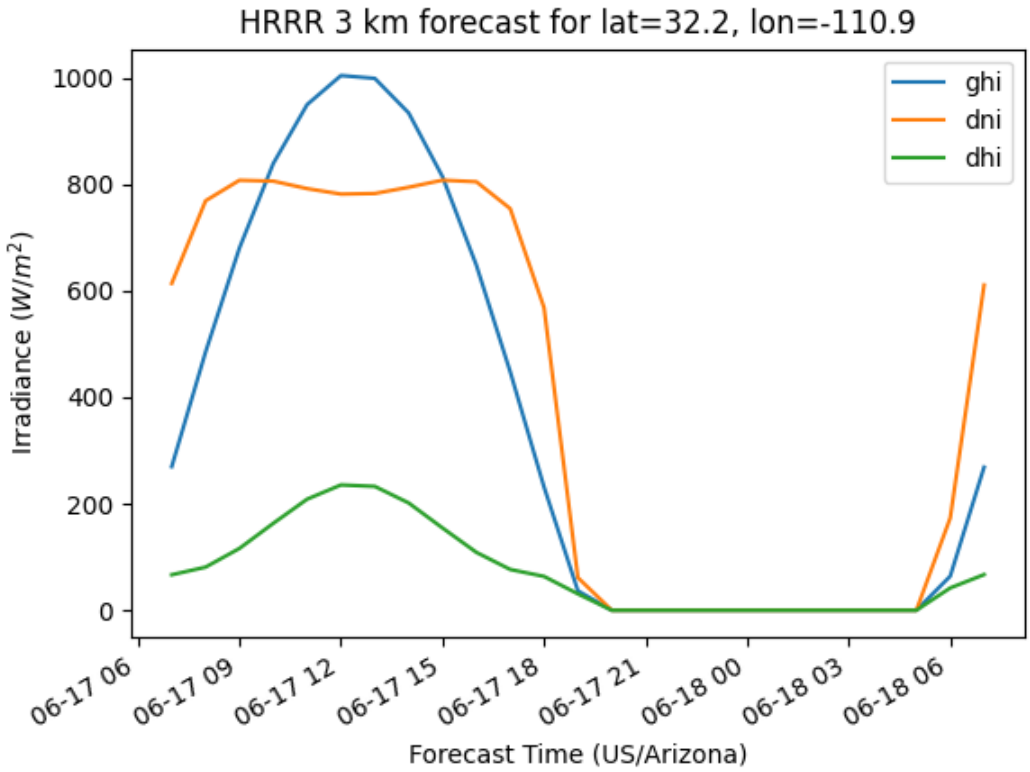
GFS

The Global Forecast System (GFS) is the US model that provides forecasts for the entire globe. The GFS is updated every 6 hours. The GFS is run at two resolutions, 0.25 deg and 0.5 deg, and is available with 3 hour time resolution. Forecasts from GFS model were shown above. Use the GFS, among others, if you want forecasts for 1-7 days or if you want forecasts for anywhere on Earth.

HRRR

The High Resolution Rapid Refresh (HRRR) model is perhaps the most accurate model, however, it is only available for ~15 hours. It is updated every hour and runs at 3 km resolution. The HRRR excels in severe weather situations. See the NOAA ESRL HRRR page for more information. Use the HRRR, among others, if you want forecasts for less than 24 hours. The HRRR model covers the continental United States.

```
In [50]: model = HRRR()

In [51]: data = model.get_processed_data(latitude, longitude, start, end)

In [52]: data[irrad_vars].plot();

In [53]: plt.ylabel('Irradiance ($W/m^2$)');

In [54]: plt.xlabel('Forecast Time ({})'.format(tz));

In [55]: plt.title('HRRR 3 km forecast for lat={}, lon={}'
   ....:           .format(latitude, longitude));
   ....:

In [56]: plt.legend();
```



RAP

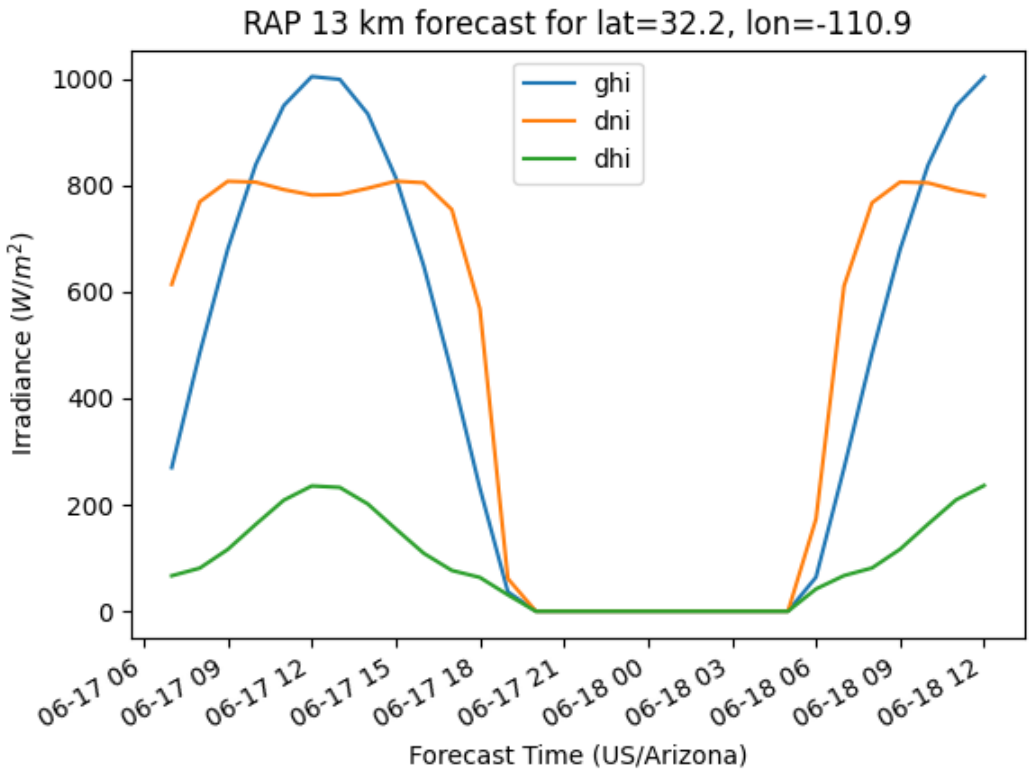
The Rapid Refresh (RAP) model is the parent model for the HRRR. It is updated every hour and runs at 40, 20, and 13 km resolutions. Only the 20 and 40 km resolutions are currently available in pvlib. It is also excels in severe weather situations. See the NOAA ESRL HRRR page for more information. Use the RAP, among others, if you want forecasts for less than 24 hours. The RAP model covers most of North America.

```
In [57]: model = RAP()
```

(continues on next page)

(continued from previous page)

```
In [58]: data = model.get_processed_data(latitude, longitude, start, end)

In [59]: data[irrad_vars].plot();

In [60]: plt.ylabel('Irradiance ($W/m^2$)');

In [61]: plt.xlabel('Forecast Time ({})'.format(tz));

In [62]: plt.title('RAP 13 km forecast for lat={}, lon={}'
   ....:           .format(latitude, longitude));
   ....:

In [63]: plt.legend();
```



NAM

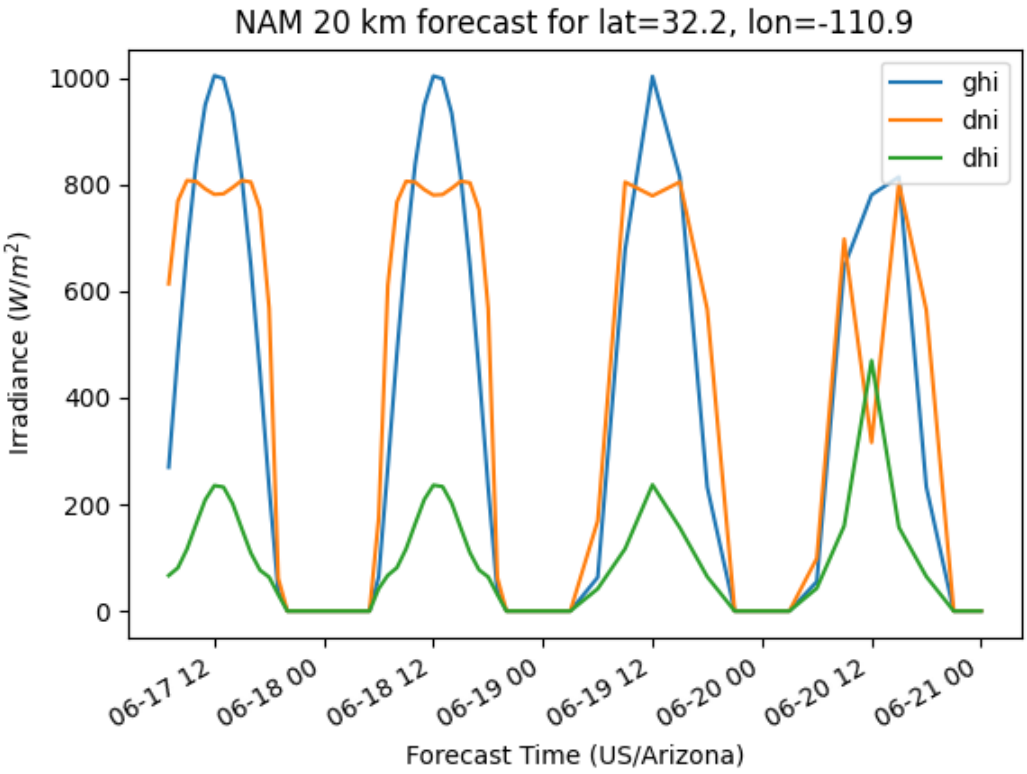
The North American Mesoscale model covers, not surprisingly, North America. It is updated every 6 hours. pvlib provides access to 20 km resolution NAM data with a time horizon of up to 4 days.

```
In [64]: model = NAM()

In [65]: data = model.get_processed_data(latitude, longitude, start, end)

In [66]: data[irrad_vars].plot();
```

(continues on next page)

(continued from previous page)

```
In [67]: plt.ylabel('Irradiance ($W/m^2$)');

In [68]: plt.xlabel('Forecast Time ({})'.format(tz));

In [69]: plt.title('NAM 20 km forecast for lat={}, lon={}'
   ....:           .format(latitude, longitude));
   ....:

In [70]: plt.legend();
```



NDFD

The National Digital Forecast Database is not a model, but rather a collection of forecasts made by National Weather Service offices across the country. It is updated every 6 hours. Use the NDFD, among others, for forecasts at all time horizons. The NDFD is available for the United States.

```
In [71]: model = NDFD()

In [72]: data = model.get_processed_data(latitude, longitude, start, end)

In [73]: data[irrad_vars].plot();

In [74]: plt.ylabel('Irradiance ($W/m^2$)');

In [75]: plt.xlabel('Forecast Time ({})'.format(tz));
```

(continues on next page)

(continued from previous page)

```
In [76]: plt.title('NDFD forecast for lat={}, lon={}'
   ....:             .format(latitude, longitude));
   ....:  plt.legend();
   ....:  plt.close();
   ....:
  File "<ipython-input-76-1c7cdda0217f>", line 3
    plt.legend();
    ^
IndentationError: unexpected indent
```

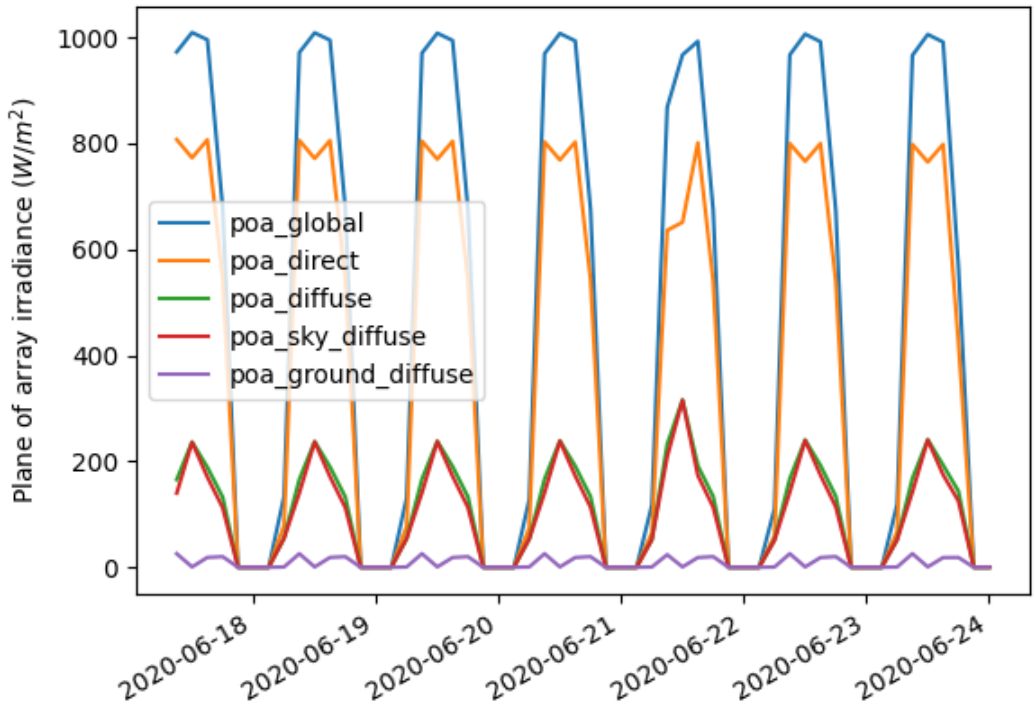
3.11.4 PV Power Forecast

Finally, we demonstrate the application of the weather forecast data to a PV power forecast. Please see the remainder of the pvlib documentation for details.

```
In [77]: from pvlib.pvsystem import PVSystem, retrieve_sam

In [78]: from pvlib.temperature import TEMPERATURE_MODEL_PARAMETERS

In [79]: from pvlib.tracking import SingleAxisTracker

In [80]: from pvlib.modelchain import ModelChain

In [81]: sandia_modules = retrieve_sam('sandiamod')

In [82]: cec_inverters = retrieve_sam('cecinverter')

In [83]: module = sandia_modules['Canadian_Solar_CS5P_220M___2009_']

In [84]: inverter = cec_inverters['SMA_America__SC630CP_US__with_ABB_EcoDry_Ultra_
↪transformer_']

In [85]: temperature_model_parameters = TEMPERATURE_MODEL_PARAMETERS['sapm']['open_
↪rack_glass_glass']

# model a big tracker for more fun
In [86]: system = SingleAxisTracker(module_parameters=module, inverter_
↪parameters=inverter, temperature_model_parameters=temperature_model_parameters,␣
↪modules_per_string=15, strings_per_inverter=300)

# fx is a common abbreviation for forecast
In [87]: fx_model = GFS()

In [88]: fx_data = fx_model.get_processed_data(latitude, longitude, start, end)

# use a ModelChain object to calculate modeling intermediates
In [89]: mc = ModelChain(system, fx_model.location)

# extract relevant data for model chain
In [90]: mc.run_model(fx_data);
```

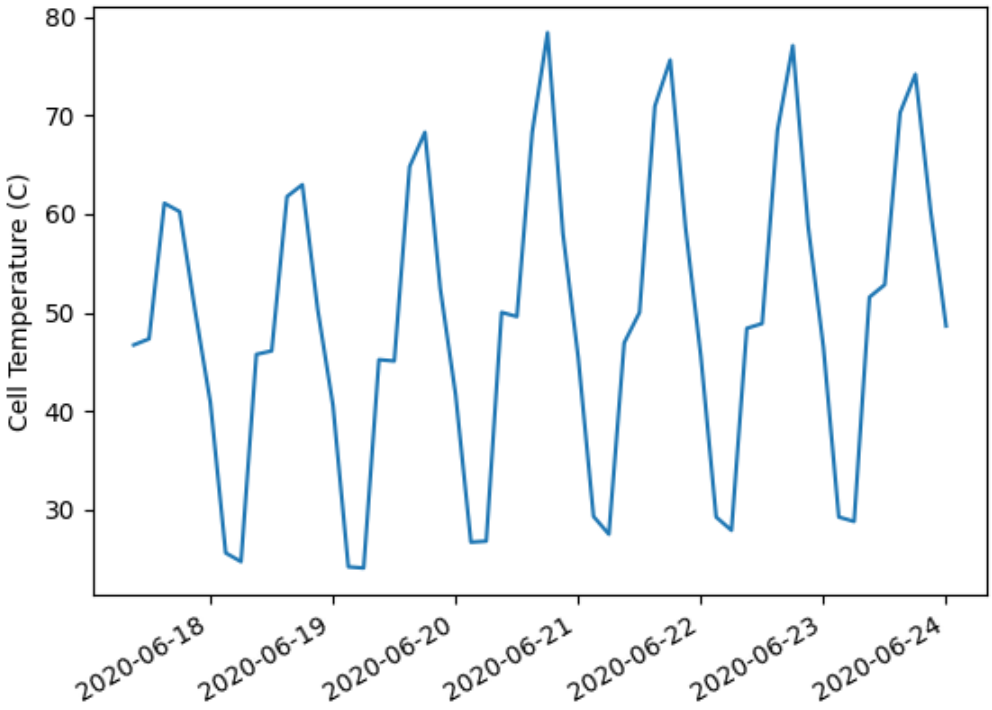
Now we plot a couple of modeling intermediates and the forecast power. Here's the forecast plane of array irradiance...

```
In [91]: mc.total_irrad.plot();

In [92]: plt.ylabel('Plane of array irradiance ($W/m^2$)');

In [93]: plt.legend(loc='best');
```



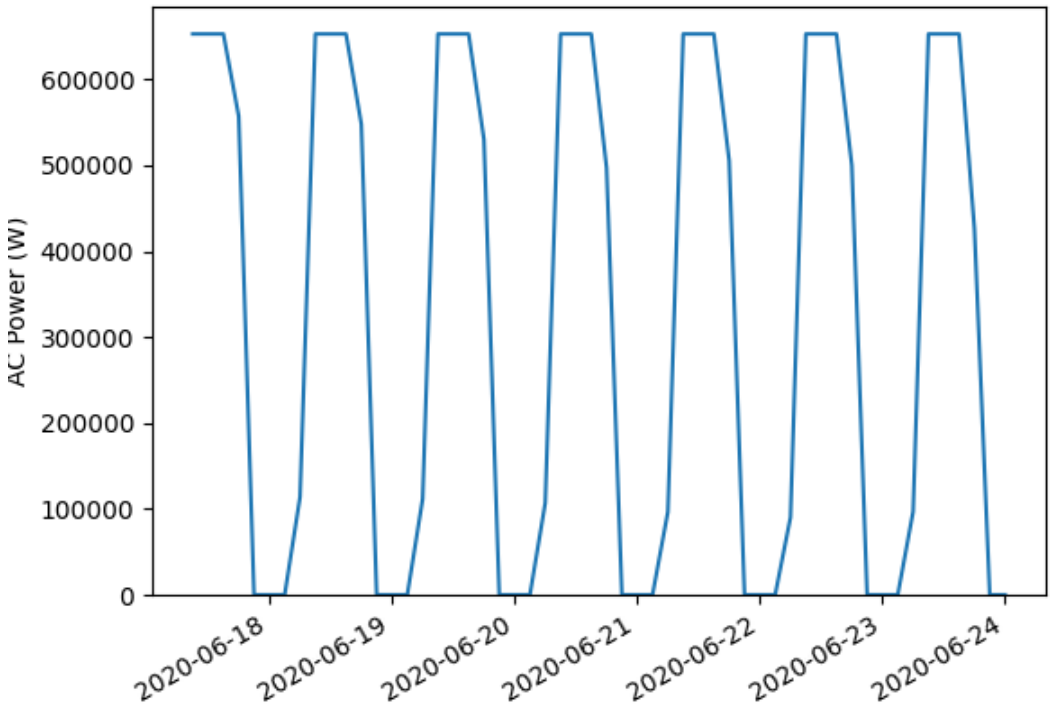
. . . the cell and module temperature. . .

```
In [94]: mc.cell_temperature.plot();

In [95]: plt.ylabel('Cell Temperature (C)');
```



. . . and finally AC power. . .

```
In [96]: mc.ac.fillna(0).plot();

In [97]: plt.ylim(0, None);

In [98]: plt.ylabel('AC Power (W)');
```



3.12 API reference

3.12.1 Classes

pvlib-python provides a collection of classes for users that prefer object-oriented programming. These classes can help users keep track of data in a more organized way, and can help to simplify the modeling process. The classes do not add any functionality beyond the procedural code. Most of the object methods are simple wrappers around the corresponding procedural code.

location.Location(latitude, longitude[, tz, ...])	Location objects are convenient containers for latitude, longitude, timezone, and altitude data associated with a particular geographic location.
pvsystem.PVSystem([surface_tilt, ...])	The PVSystem class defines a standard set of PV system attributes and modeling functions.
tracking.SingleAxisTracker([axis_tilt, ...])	Inherits the PV modeling methods from *PVSystem*.
modelchain.ModelChain(system, location[, ...])	The ModelChain class to provides a standardized, high-level interface for all of the modeling steps necessary for calculating PV power from a time series of weather inputs.
pvsystem.LocalizedPVSystem([pvsystem, location])	The LocalizedPVSystem class defines a standard set of installed PV system attributes and modeling functions.

Continued on next page

Table 1 – continued from previous page

tracking.LocalizedSingleAxisTracker([...])	The LocalizedSingleAxisTracker class defines a standard set of installed PV system attributes and modeling functions.

pvlib.location.Location

class pvlib.location.**Location**(*latitude*, *longitude*, *tz='UTC'*, *altitude=0*, *name=None*, ***kwargs*)

Location objects are convenient containers for latitude, longitude, timezone, and altitude data associated with a particular geographic location. You can also assign a name to a location object.

Location objects have two timezone attributes:

- `tz` is a IANA timezone string.
- `pytz` is a pytz timezone object.

Location objects support the print method.

Parameters

- **latitude** (*float.*) – Positive is north of the equator. Use decimal degrees notation.
- **longitude** (*float.*) – Positive is east of the prime meridian. Use decimal degrees notation.
- **tz** (*str, int, float, or pytz.timezone, default 'UTC'.*) – See http://en.wikipedia.org/wiki/List_of_tz_database_time_zones for a list of valid time zones. pytz.timezone objects will be converted to strings. ints and floats must be in hours from UTC.
- **altitude** (*float, default 0.*) – Altitude from sea level in meters.
- **name** (*None or string, default None.*) – Sets the name attribute of the Location object.
- ****kwargs** – Arbitrary keyword arguments. Included for compatibility, but not used.

See also:

pvlib.pvsystem.PVSystem

__init__(*latitude*, *longitude*, *tz='UTC'*, *altitude=0*, *name=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(latitude, longitude[, tz, ...])	Initialize self.
from_epw(metadata[, data])	Create a Location object based on a metadata dictionary from epw data readers.
from_tmy(tmy_metadata[, tmy_data])	Create an object based on a metadata dictionary from tmy2 or tmy3 data readers.
get_airmass([times, solar_position, model])	Calculate the relative and absolute airmass.
get_clearsky(times[, model, solar_position, ...])	Calculate the clear sky estimates of GHI, DNI, and/or DHI at this location.

Continued on next page

Table 2 – continued from previous page

get_solarposition(times[, pressure, temperature])	Uses the `solarposition.get_solarposition()` function to calculate the solar zenith, azimuth, etc.
get_sun_rise_set_transit(times[, method])	Calculate sunrise, sunset and transit times.

pvlib.pvsystem.PVSystem

class pvlib.pvsystem.**PVSystem**(*surface_tilt=0*, *surface_azimuth=180*, *albedo=None*, *surface_type=None*, *module=None*, *module_type='glass_polymer'*, *module_parameters=None*, *temperature_model_parameters=None*, *modules_per_string=1*, *strings_per_inverter=1*, *inverter=None*, *inverter_parameters=None*, *racking_model='open_rack'*, *losses_parameters=None*, *name=None*, ***kwargs*)

The PVSystem class defines a standard set of PV system attributes and modeling functions. This class describes the collection and interactions of PV system components rather than an installed system on the ground. It is typically used in combination with *Location* and *ModelChain* objects.

See the *LocalizedPVSystem* class for an object model that describes an installed PV system.

The class supports basic system topologies consisting of:

- *N* total modules arranged in series (*modules_per_string=N*, *strings_per_inverter=1*).
- *M* total modules arranged in parallel (*modules_per_string=1*, *strings_per_inverter=M*).
- *NxM* total modules arranged in *M* strings of *N* modules each (*modules_per_string=N*, *strings_per_inverter=M*).

The class is complementary to the module-level functions.

The attributes should generally be things that don't change about the system, such the type of module and the inverter. The instance methods accept arguments for things that do change, such as irradiance and temperature.

Parameters

- **surface_tilt** (*float or array-like, default 0*) – Surface tilt angles in decimal degrees. The tilt angle is defined as degrees from horizontal (e.g. surface facing up = 0, surface facing horizon = 90)
- **surface_azimuth** (*float or array-like, default 180*) – Azimuth angle of the module surface. North=0, East=90, South=180, West=270.
- **albedo** (*None or float, default None*) – The ground albedo. If `None`, will attempt to use `surface_type` and `irradiance.SURFACE_ALBEDOS` to lookup albedo.
- **surface_type** (*None or string, default None*) – The ground surface type. See `irradiance.SURFACE_ALBEDOS` for valid values.
- **module** (*None or string, default None*) – The model name of the modules. May be used to look up the module_parameters dictionary via some other method.
- **module_type** (*None or string, default 'glass_polymer'*) – Describes the module's construction. Valid strings are 'glass_polymer' and 'glass_glass'. Used for cell and module temperature calculations.
- **module_parameters** (*None, dict or Series, default None*) – Module parameters as defined by the SAPM, CEC, or other.

- **temperature_model_parameters** (*None, dict or Series, default None.*) – Temperature model parameters as defined by the SAPM, Pvsyst, or other.
- **modules_per_string** (*int or float, default 1*) – See system topology discussion above.
- **strings_per_inverter** (*int or float, default 1*) – See system topology discussion above.
- **inverter** (*None or string, default None*) – The model name of the inverters. May be used to look up the inverter_parameters dictionary via some other method.
- **inverter_parameters** (*None, dict or Series, default None*) – Inverter parameters as defined by the SAPM, CEC, or other.
- **racking_model** (*None or string, default 'open_rack'*) – Valid strings are 'open_rack', 'close_mount', and 'insulated_back'. Used to identify a parameter set for the SAPM cell temperature model.
- **losses_parameters** (*None, dict or Series, default None*) – Losses parameters as defined by PVWatts or other.
- **name** (*None or string, default None*) –
- ****kwargs** – Arbitrary keyword arguments. Included for compatibility, but not used.

See also:

pvlib.location.Location, *pvlib.tracking.SingleAxisTracker*, *pvlib.pvsystem.LocalizedPVSystem*

__init__(*surface_tilt=0*, *surface_azimuth=180*, *albedo=None*, *surface_type=None*, *module=None*, *module_type='glass_polymer'*, *module_parameters=None*, *temperature_model_parameters=None*, *modules_per_string=1*, *strings_per_inverter=1*, *inverter=None*, *inverter_parameters=None*, *racking_model='open_rack'*, *losses_parameters=None*, *name=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([surface_tilt, surface_azimuth, ...])	Initialize self.
adrinverter(v_dc, p_dc)	
ashraeiam(aoi)	Deprecated.
calcparams_cec(effective_irradiance, ...)	Use the *calcparams_cec()* function, the input parameters and self.module_parameters to calculate the module currents and resistances.
calcparams_desoto(effective_irradiance, ...)	Use the *calcparams_desoto()* function, the input parameters and self.module_parameters to calculate the module currents and resistances.
calcparams_pvsyst(effective_irradiance, ...)	Use the *calcparams_pvsyst()* function, the input parameters and self.module_parameters to calculate the module currents and resistances.
faiman_celltemp(poa_global, temp_air[, ...])	Use temperature.faiman() to calculate cell temperature.

Continued on next page

Table 3 – continued from previous page

first_solar_spectral_loss(pw, airmass_absolute)	Use the first_solar_spectral_correction() function to calculate the spectral loss modifier.
get_aoi(solar_zenith, solar_azimuth)	Get the angle of incidence on the system.
get_iam(aoi[, iam_model])	Determine the incidence angle modifier using the method specified by iam_model.
get_irradiance(solar_zenith, solar_azimuth, ...)	Uses the irradiance.get_total_irradiance() function to calculate the plane of array irradiance components on a tilted surface defined by self.surface_tilt, self.surface_azimuth, and self.albedo.
i_from_v(resistance_shunt, ...)	Wrapper around the i_from_v() function.
localize([location, latitude, longitude])	Creates a LocalizedPVSystem object using this object and location data.
physicaliam(aoi)	Deprecated.
pvsyst_celltemp(poa_global, temp_air[, ...])	Uses temperature.pvsyst_cell() to calculate cell temperature.
pvwatts_ac(pdc)	Calculates AC power according to the PVWatts model using pvwatts_ac(), *self.module_parameters['pdc0']*, and *eta_inv_nom=self.inverter_parameters['eta_inv_nom']*.
pvwatts_dc(g_poa_effective, temp_cell)	Calcuates DC power according to the PVWatts model using pvwatts_dc(), *self.module_parameters['pdc0']*, and *self.module_parameters['gamma_pdc']*.
pvwatts_losses()	Calculates DC power losses according the PVwatts model using pvwatts_losses() and self.losses_parameters.‘
sapm(effective_irradiance, temp_cell, **kwargs)	Use the sapm() function, the input parameters, and self.module_parameters to calculate Voc, Isc, Ix, Ixx, Vmp, and Imp.
sapm_aoi_loss(aoi)	Deprecated.
sapm_celltemp(poa_global, temp_air, wind_speed)	Uses temperature.sapm_cell() to calculate cell temperatures.
sapm_effective_irradiance(poa_direct, ...[, ...])	Use the sapm_effective_irradiance() function, the input parameters, and self.module_parameters to calculate effective irradiance.
sapm_spectral_loss(airmass_absolute)	Use the sapm_spectral_loss() function, the input parameters, and self.module_parameters to calculate F1.
scale_voltage_current_power(data)	Scales the voltage, current, and power of the DataFrames returned by singlediode() and sapm() by *self.modules_per_string* and *self.strings_per_inverter*.
singlediode(photocurrent, ...[, ivcurve_pnts])	Wrapper around the singlediode() function.
snlinverter(v_dc, p_dc)	Uses snlinverter() to calculate AC power based on self.inverter_parameters and the input parameters.

pvlib.tracking.SingleAxisTracker

class pvlib.tracking.**SingleAxisTracker**(*axis_tilt=0*, *axis_azimuth=0*, *max_angle=90*, *backtrack=True*, *gcr=0.2857142857142857*, ***kwargs*)

Inherits the PV modeling methods from *PVSystem*.

Parameters

- **axis_tilt** (*float, default 0*) – The tilt of the axis of rotation (i.e, the y-axis defined by axis_azimuth) with respect to horizontal, in decimal degrees.
- **axis_azimuth** (*float, default 0*) – A value denoting the compass direction along which the axis of rotation lies. Measured in decimal degrees East of North.
- **max_angle** (*float, default 90*) – A value denoting the maximum rotation angle, in decimal degrees, of the one-axis tracker from its horizontal position (horizontal if axis_tilt = 0). A max_angle of 90 degrees allows the tracker to rotate to a vertical position to point the panel towards a horizon. max_angle of 180 degrees allows for full rotation.
- **backtrack** (*bool, default True*) – Controls whether the tracker has the capability to "backtrack" to avoid row-to-row shading. False denotes no backtrack capability. True denotes backtrack capability.
- **gcr** (*float, default 2.0/7.0*) – A value denoting the ground coverage ratio of a tracker system which utilizes backtracking; i.e. the ratio between the PV array surface area to total ground area. A tracker system with modules 2 meters wide, centered on the tracking axis, with 6 meters between the tracking axes has a gcr of 2/6=0.333. If gcr is not provided, a gcr of 2/7 is default. gcr must be <=1.

__init__(*axis_tilt=0*, *axis_azimuth=0*, *max_angle=90*, *backtrack=True*, *gcr=0.2857142857142857*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([axis_tilt, axis_azimuth, ...])	Initialize self.
adrinverter(v_dc, p_dc)	
ashraeiam(aoi)	Deprecated.
calcparams_cec(effective_irradiance, ...)	Use the calcparams_cec() function, the input parameters and self.module_parameters to calculate the module currents and resistances.
calcparams_desoto(effective_irradiance, ...)	Use the calcparams_desoto() function, the input parameters and self.module_parameters to calculate the module currents and resistances.
calcparams_pvsyst(effective_irradiance, ...)	Use the calcparams_pvsyst() function, the input parameters and self.module_parameters to calculate the module currents and resistances.
faiman_celltemp(poa_global, temp_air[, ...])	Use temperature.faiman() to calculate cell temperature.
first_solar_spectral_loss(pw, airmass_absolute)	Use the first_solar_spectral_correction() function to calculate the spectral loss modifier.
get_aoi(surface_tilt, surface_azimuth, ...)	Get the angle of incidence on the system.

Continued on next page

Table 4 – continued from previous page

get_iam(aoi[, iam_model])	Determine the incidence angle modifier using the method specified by iam_model.
get_irradiance(surface_tilt, ...[, ...])	Uses the irradiance.get_total_irradiance() function to calculate the plane of array irradiance components on a tilted surface defined by the input data and self.albedo.
i_from_v(resistance_shunt, ...)	Wrapper around the i_from_v() function.
localize([location, latitude, longitude])	Creates a LocalizedSingleAxisTracker object using this object and location data.
physicaliam(aoi)	Deprecated.
pvsyst_celltemp(poa_global, temp_air[, ...])	Uses temperature.pvsyst_cell() to calculate cell temperature.
pvwatts_ac(pdc)	Calculates AC power according to the PVWatts model using pvwatts_ac(), *self.module_parameters['pdc0']*, and *eta_inv_nom=self.inverter_parameters['eta_inv_nom']*.
pvwatts_dc(g_poa_effective, temp_cell)	Calcuates DC power according to the PVWatts model using pvwatts_dc(), *self.module_parameters['pdc0']*, and *self.module_parameters['gamma_pdc']*.
pvwatts_losses()	Calculates DC power losses according the PVwatts model using pvwatts_losses() and self.losses_parameters.‘
sapm(effective_irradiance, temp_cell, **kwargs)	Use the sapm() function, the input parameters, and self.module_parameters to calculate Voc, Isc, Ix, Ixx, Vmp, and Imp.
sapm_aoi_loss(aoi)	Deprecated.
sapm_celltemp(poa_global, temp_air, wind_speed)	Uses temperature.sapm_cell() to calculate cell temperatures.
sapm_effective_irradiance(poa_direct, ...[, ...])	Use the sapm_effective_irradiance() function, the input parameters, and self.module_parameters to calculate effective irradiance.
sapm_spectral_loss(airmass_absolute)	Use the sapm_spectral_loss() function, the input parameters, and self.module_parameters to calculate F1.
scale_voltage_current_power(data)	Scales the voltage, current, and power of the DataFrames returned by singlediode() and sapm() by *self.modules_per_string* and *self.strings_per_inverter*.
singleaxis(apparent_zenith, apparent_azimuth)	Get tracking data.
singlediode(photocurrent, ...[, ivcurve_pnts])	Wrapper around the singlediode() function.
snlinverter(v_dc, p_dc)	Uses snlinverter() to calculate AC power based on self.inverter_parameters and the input parameters.

pvlib.modelchain.ModelChain

class pvlib.modelchain.**ModelChain**(*system*, *location*, *orientation_strategy=None*, *clearsky_model='ineichen'*, *transposition_model='haydavies'*, *solar_position_method='nrel_numpy'*, *airmass_model='kastenyoung1989'*, *dc_model=None*, *ac_model=None*, *aoi_model=None*, *spectral_model=None*, *temperature_model=None*, *losses_model='no_loss'*, *name=None*, ***kwargs*)

The ModelChain class to provides a standardized, high-level interface for all of the modeling steps necessary for calculating PV power from a time series of weather inputs.

See https://pvlib-python.readthedocs.io/en/stable/modelchain.html for examples.

Parameters

- **system** (PVSystem) – A *PVSystem* object that represents the connected set of modules, inverters, etc.
- **location** (Location) – A *Location* object that represents the physical location at which to evaluate the model.
- **orientation_strategy** (*None or str, default None*) – The strategy for aligning the modules. If not None, sets the `surface_azimuth` and `surface_tilt` properties of the `system`. Allowed strategies include 'flat', 'south_at_latitude_tilt'. Ignored for SingleAxisTracker systems.
- **clearsky_model** (*str, default 'ineichen'*) – Passed to location.get_clearsky.
- **transposition_model** (*str, default 'haydavies'*) – Passed to system.get_irradiance.
- **solar_position_method** (*str, default 'nrel_numpy'*) – Passed to location.get_solarposition.
- **airmass_model** (*str, default 'kastenyoung1989'*) – Passed to location.get_airmass.
- **dc_model** (*None, str, or function, default None*) – If None, the model will be inferred from the contents of system.module_parameters. Valid strings are 'sapm', 'desoto', 'cec', 'pvsyst', 'pvwatts'. The ModelChain instance will be passed as the first argument to a user-defined function.
- **ac_model** (*None, str, or function, default None*) – If None, the model will be inferred from the contents of system.inverter_parameters and system.module_parameters. Valid strings are 'snlinverter', 'adrinverter', 'pvwatts'. The ModelChain instance will be passed as the first argument to a user-defined function.
- **aoi_model** (*None, str, or function, default None*) – If None, the model will be inferred from the contents of system.module_parameters. Valid strings are 'physical', 'ashrae', 'sapm', 'martin_ruiz', 'no_loss'. The ModelChain instance will be passed as the first argument to a user-defined function.
- **spectral_model** (*None, str, or function, default None*) – If None, the model will be inferred from the contents of system.module_parameters. Valid strings are 'sapm', 'first_solar', 'no_loss'. The ModelChain instance will be passed as the first argument to a user-defined function.

- **temperature_model** (*None, str or function, default None*) – Valid strings are 'sapm', 'pvsyst', and 'faiman'. The ModelChain instance will be passed as the first argument to a user-defined function.
- **losses_model** (*str or function, default 'no_loss'*) – Valid strings are 'pvwatts', 'no_loss'. The ModelChain instance will be passed as the first argument to a user-defined function.
- **name** (*None or str, default None*) – Name of ModelChain instance.
- ****kwargs** – Arbitrary keyword arguments. Included for compatibility, but not used.

__init__(*system*, *location*, *orientation_strategy=None*, *clearsky_model='ineichen'*, *transposition_model='haydavies'*, *solar_position_method='nrel_numpy'*, *airmass_model='kastenyoung1989'*, *dc_model=None*, *ac_model=None*, *aoi_model=None*, *spectral_model=None*, *temperature_model=None*, *losses_model='no_loss'*, *name=None*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(system, location[, ...])	Initialize self.
adrinverter()	
ashrae_aoi_loss()	
cec()	
complete_irradiance(weather[, times])	Determine the missing irradiation columns.
desoto()	
effective_irradiance_model()	
faiman_temp()	
first_solar_spectral_loss()	
infer_ac_model()	
infer_aoi_model()	
infer_dc_model()	
infer_losses_model()	
infer_spectral_model()	
infer_temperature_model()	
martin_ruiz_aoi_loss()	
no_aoi_loss()	
no_extra_losses()	
no_spectral_loss()	
physical_aoi_loss()	
prepare_inputs(weather[, times])	Prepare the solar position, irradiance, and weather inputs to the model.
pvsyst()	
pvsyst_temp()	
pvwatts_dc()	
pvwatts_inverter()	
pvwatts_losses()	
run_model(weather[, times])	Run the model.
sapm()	
sapm_aoi_loss()	
sapm_spectral_loss()	
sapm_temp()	

Continued on next page

Table 5 – continued from previous page

snlinverter()	

Attributes

ac_model
aoi_model
dc_model
losses_model
orientation_strategy
spectral_model
temperature_model

pvlib.pvsystem.LocalizedPVSystem

class pvlib.pvsystem.**LocalizedPVSystem**(*pvsystem=None*, *location=None*, ***kwargs*)

The LocalizedPVSystem class defines a standard set of installed PV system attributes and modeling functions. This class combines the attributes and methods of the PVSystem and Location classes.

The LocalizedPVSystem may have bugs due to the difficulty of robustly implementing multiple inheritance. See *ModelChain* for an alternative paradigm for modeling PV systems at specific locations.

__init__(*pvsystem=None*, *location=None*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([pvsystem, location])	Initialize self.
adrinverter(v_dc, p_dc)	
ashraeiam(aoi)	Deprecated.
calcparams_cec(effective_irradiance, ...)	Use the *calcparams_cec()* function, the input parameters and self.module_parameters to calculate the module currents and resistances.
calcparams_desoto(effective_irradiance, ...)	Use the *calcparams_desoto()* function, the input parameters and self.module_parameters to calculate the module currents and resistances.
calcparams_pvsyst(effective_irradiance, ...)	Use the *calcparams_pvsyst()* function, the input parameters and self.module_parameters to calculate the module currents and resistances.
faiman_celltemp(poa_global, temp_air[, ...])	Use temperature.faiman() to calculate cell temperature.
first_solar_spectral_loss(pw, airmass_absolute)	Use the first_solar_spectral_correction() function to calculate the spectral loss modifier.
from_epw(metadata[, data])	Create a Location object based on a metadata dictionary from epw data readers.
from_tmy(tmy_metadata[, tmy_data])	Create an object based on a metadata dictionary from tmy2 or tmy3 data readers.
get_airmass([times, solar_position, model])	Calculate the relative and absolute airmass.
get_aoi(solar_zenith, solar_azimuth)	Get the angle of incidence on the system.

Continued on next page

Table 7 – continued from previous page

get_clearsky(times[, model, solar_position, ...])	Calculate the clear sky estimates of GHI, DNI, and/or DHI at this location.
get_iam(aoi[, iam_model])	Determine the incidence angle modifier using the method specified by iam_model.
get_irradiance(solar_zenith, solar_azimuth, ...)	Uses the irradiance.get_total_irradiance() function to calculate the plane of array irradiance components on a tilted surface defined by self.surface_tilt, self.surface_azimuth, and self.albedo.
get_solarposition(times[, pressure, temperature])	Uses the solarposition.get_solarposition() function to calculate the solar zenith, azimuth, etc.
get_sun_rise_set_transit(times[, method])	Calculate sunrise, sunset and transit times.
i_from_v(resistance_shunt, ...)	Wrapper around the *i_from_v()* function.
localize([location, latitude, longitude])	Creates a LocalizedPVSystem object using this object and location data.
physicaliam(aoi)	Deprecated.
pvsyst_celltemp(poa_global, temp_air[, ...])	Uses temperature.pvsyst_cell() to calculate cell temperature.
pvwatts_ac(pdc)	Calculates AC power according to the PVWatts model using *pvwatts_ac()*, *self.module_parameters['pdc0']*, and *eta_inv_nom=self.inverter_parameters['eta_inv_nom']*.
pvwatts_dc(g_poa_effective, temp_cell)	Calcuates DC power according to the PVWatts model using *pvwatts_dc()*, *self.module_parameters['pdc0']*, and *self.module_parameters['gamma_pdc']*.
pvwatts_losses()	Calculates DC power losses according the PVwatts model using *pvwatts_losses()* and self.losses_parameters.'
sapm(effective_irradiance, temp_cell, **kwargs)	Use the *sapm()* function, the input parameters, and self.module_parameters to calculate Voc, Isc, Ix, Ixx, Vmp, and Imp.
sapm_aoi_loss(aoi)	Deprecated.
sapm_celltemp(poa_global, temp_air, wind_speed)	Uses temperature.sapm_cell() to calculate cell temperatures.
sapm_effective_irradiance(poa_direct, ...[, ...])	Use the *sapm_effective_irradiance()* function, the input parameters, and self.module_parameters to calculate effective irradiance.
sapm_spectral_loss(airmass_absolute)	Use the *sapm_spectral_loss()* function, the input parameters, and self.module_parameters to calculate F1.
scale_voltage_current_power(data)	Scales the voltage, current, and power of the DataFrames returned by *singlediode()* and *sapm()* by *self.modules_per_string* and *self.strings_per_inverter*.
singlediode(photocurrent, ... [, ivcurve_pnts])	Wrapper around the *singlediode()* function.

Continued on next page

Table 7 – continued from previous page

snlinverter(v_dc, p_dc)	Uses *snlinverter()* to calculate AC power based on self.inverter_parameters and the input parameters.

pvlib.tracking.LocalizedSingleAxisTracker

class pvlib.tracking.**LocalizedSingleAxisTracker**(*pvsystem=None*, *location=None*, ***kwargs*)

The LocalizedSingleAxisTracker class defines a standard set of installed PV system attributes and modeling functions. This class combines the attributes and methods of the SingleAxisTracker (a subclass of PVSystem) and Location classes.

The LocalizedSingleAxisTracker may have bugs due to the difficulty of robustly implementing multiple inheritance. See *ModelChain* for an alternative paradigm for modeling PV systems at specific locations.

__init__(*pvsystem=None*, *location=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([pvsystem, location])	Initialize self.
adrinverter(v_dc, p_dc)	
ashraeiam(aoi)	Deprecated.
calcparams_cec(effective_irradiance, ...)	Use the calcparams_cec() function, the input parameters and self.module_parameters to calculate the module currents and resistances.
calcparams_desoto(effective_irradiance, ...)	Use the calcparams_desoto() function, the input parameters and self.module_parameters to calculate the module currents and resistances.
calcparams_pvsyst(effective_irradiance, ...)	Use the calcparams_pvsyst() function, the input parameters and self.module_parameters to calculate the module currents and resistances.
faiman_celltemp(poa_global, temp_air[, ...])	Use temperature.faiman() to calculate cell temperature.
first_solar_spectral_loss(pw, airmass_absolute)	Use the first_solar_spectral_correction() function to calculate the spectral loss modifier.
from_epw(metadata[, data])	Create a Location object based on a metadata dictionary from epw data readers.
from_tmy(tmy_metadata[, tmy_data])	Create an object based on a metadata dictionary from tmy2 or tmy3 data readers.
get_airmass([times, solar_position, model])	Calculate the relative and absolute airmass.
get_aoi(surface_tilt, surface_azimuth, ...)	Get the angle of incidence on the system.
get_clearsky(times[, model, solar_position, ...])	Calculate the clear sky estimates of GHI, DNI, and/or DHI at this location.
get_iam(aoi[, iam_model])	Determine the incidence angle modifier using the method specified by iam_model.

Continued on next page

Table 8 – continued from previous page

get_irradiance(surface_tilt, ..., [, ...])	Uses the irradiance.get_total_irradiance() function to calculate the plane of array irradiance components on a tilted surface defined by the input data and self.albedo.
get_solarposition(times[, pressure, temperature])	Uses the solarposition.get_solarposition() function to calculate the solar zenith, azimuth, etc.
get_sun_rise_set_transit(times[, method])	Calculate sunrise, sunset and transit times.
i_from_v(resistance_shunt, ...)	Wrapper around the i_from_v() function.
localize([location, latitude, longitude])	Creates a *LocalizedSingleAxisTracker* object using this object and location data.
physicaliam(aoi)	Deprecated.
pvsyst_celltemp(poa_global, temp_air[, ...])	Uses temperature.pvsyst_cell() to calculate cell temperature.
pvwatts_ac(pdc)	Calculates AC power according to the PVWatts model using pvwatts_ac(), *self.module_parameters['pdc0']*, and *eta_inv_nom=self.inverter_parameters['eta_inv_nom']*.
pvwatts_dc(g_poa_effective, temp_cell)	Calcuates DC power according to the PVWatts model using pvwatts_dc(), *self.module_parameters['pdc0']*, and *self.module_parameters['gamma_pdc']*.
pvwatts_losses()	Calculates DC power losses according the PVwatts model using pvwatts_losses() and self.losses_parameters.'
sapm(effective_irradiance, temp_cell, **kwargs)	Use the sapm() function, the input parameters, and self.module_parameters to calculate Voc, Isc, Ix, Ixx, Vmp, and Imp.
sapm_aoi_loss(aoi)	Deprecated.
sapm_celltemp(poa_global, temp_air, wind_speed)	Uses temperature.sapm_cell() to calculate cell temperatures.
sapm_effective_irradiance(poa_direct, ...[, ...])	Use the sapm_effective_irradiance() function, the input parameters, and self.module_parameters to calculate effective irradiance.
sapm_spectral_loss(airmass_absolute)	Use the sapm_spectral_loss() function, the input parameters, and self.module_parameters to calculate F1.
scale_voltage_current_power(data)	Scales the voltage, current, and power of the DataFrames returned by singlediode() and sapm() by *self.modules_per_string* and *self.strings_per_inverter*.
singleaxis(apparent_zenith, apparent_azimuth)	Get tracking data.
singlediode(photocurrent, ... [, ivcurve_pnts])	Wrapper around the singlediode() function.
snlinverter(v_dc, p_dc)	Uses snlinverter() to calculate AC power based on self.inverter_parameters and the input parameters.

3.12.2 Solar Position

Functions and methods for calculating solar position.

The *location.Location.get_solarposition()* method and the *solarposition.get_solarposition()* function with default parameters are fast and accurate. We recommend using these functions unless you know that you need a different function.

location.Location.get_solarposition(times[, ...])	Uses the *solarposition.get_solarposition()* function to calculate the solar zenith, azimuth, etc.
solarposition.get_solarposition(time, ...[, ...])	A convenience wrapper for the solar position calculators.
solarposition.spa_python(time, latitude, ...)	Calculate the solar position using a python implementation of the NREL SPA algorithm.
solarposition.ephemeris(time, latitude, ...)	Python-native solar position calculator.
solarposition.pyephem(time, latitude, longitude)	Calculate the solar position using the PyEphem package.
solarposition.spa_c(time, latitude, longitude)	Calculate the solar position using the C implementation of the NREL SPA code.

pvlib.location.Location.get_solarposition

Location.**get_solarposition**(*times*, *pressure=None*, *temperature=12*, ***kwargs*)

Uses the `solarposition.get_solarposition()` function to calculate the solar zenith, azimuth, etc. at this location.

Parameters

- **times** (*pandas.DatetimeIndex*) – Must be localized or UTC will be assumed.
- **pressure** (*None, float, or array-like, default None*) – If None, pressure will be calculated using `atmosphere.alt2pres()` and `self.altitude`.
- **temperature** (*None, float, or array-like, default 12*) –
- **kwargs** – passed to `solarposition.get_solarposition()`

Returns **solar_position** (*DataFrame*) – Columns depend on the `method` kwarg, but always include `zenith` and `azimuth`.

pvlib.solarposition.get_solarposition

pvlib.solarposition.**get_solarposition**(*time*, *latitude*, *longitude*, *altitude=None*, *pressure=None*, *method='nrel_numpy'*, *temperature=12*, ***kwargs*)

A convenience wrapper for the solar position calculators.

Parameters

- **time** (*pandas.DatetimeIndex*) – Must be localized or UTC will be assumed.
- **latitude** (*float*) – Latitude in decimal degrees. Positive north of equator, negative to south.

- **longitude** (*float*) – Longitude in decimal degrees. Positive east of prime meridian, negative to west.
- **altitude** (*None or float, default None*) – If None, computed from pressure. Assumed to be 0 m if pressure is also None.
- **pressure** (*None or float, default None*) – If None, computed from altitude. Assumed to be 101325 Pa if altitude is also None.
- **method** (*string, default 'nrel_numpy'*) – 'nrel_numpy' uses an implementation of the NREL SPA algorithm described in [1] (default, recommended): *spa_python()*

  'nrel_numba' uses an implementation of the NREL SPA algorithm described in [1], but also compiles the code first: *spa_python()*

  'pyephem' uses the PyEphem package: *pyephem()*

  'ephemeris' uses the pvlib ephemeris code: *ephemeris()*

  'nrel_c' uses the NREL SPA C code [3]: *spa_c()*
- **temperature** (*float, default 12*) – Degrees C.
- **kwargs** – Other keywords are passed to the solar position function specified by the `method` argument.

References

pvlib.solarposition.spa_python

pvlib.solarposition.**spa_python**(*time*, *latitude*, *longitude*, *altitude=0*, *pressure=101325*, *temperature=12*, *delta_t=67.0*, *atmos_refract=None*, *how='numpy'*, *numthreads=4*, ***kwargs*)

Calculate the solar position using a python implementation of the NREL SPA algorithm.

The details of the NREL SPA algorithm are described in[1].

If numba is installed, the functions can be compiled to machine code and the function can be multithreaded. Without numba, the function evaluates via numpy with a slight performance hit.

Parameters

- **time** (*pandas.DatetimeIndex*) – Must be localized or UTC will be assumed.
- **latitude** (*float*) – Latitude in decimal degrees. Positive north of equator, negative to south.
- **longitude** (*float*) – Longitude in decimal degrees. Positive east of prime meridian, negative to west.
- **altitude** (*float, default 0*) – Distance above sea level.
- **pressure** (*int or float, optional, default 101325*) – avg. yearly air pressure in Pascals.
- **temperature** (*int or float, optional, default 12*) – avg. yearly air temperature in degrees C.

[1] I. Reda and A. Andreas, Solar position algorithm for solar radiation applications. Solar Energy, vol. 76, no. 5, pp. 577-589, 2004.

- **delta_t** (*float, optional, default 67.0*) – If delta_t is None, uses spa.calculate_deltat using time.year and time.month from pandas.DatetimeIndex. For most simulations specifing delta_t is sufficient. Difference between terrestrial time and UT1. *Note: delta_t = None will break code using nrel_numba, this will be fixed in a future version.* The USNO has historical and forecasted delta_t [3].
- **atmos_refrac** (*None or float, optional, default None*) – The approximate atmospheric refraction (in degrees) at sunrise and sunset.
- **how** (*str, optional, default 'numpy'*) – Options are 'numpy' or 'numba'. If numba >= 0.17.0 is installed, how='numba' will compile the spa functions to machine code and run them multithreaded.
- **numthreads** (*int, optional, default 4*) – Number of threads to use if how == 'numba'.

Returns *DataFrame* – The DataFrame will have the following columns: apparent_zenith (degrees), zenith (degrees), apparent_elevation (degrees), elevation (degrees), azimuth (degrees), equation_of_time (minutes).

References

See also:

pyephem(), *spa_c()*, *ephemeris()*

pvlib.solarposition.ephemeris

pvlib.solarposition.**ephemeris**(*time*, *latitude*, *longitude*, *pressure=101325*, *temperature=12*)

Python-native solar position calculator. The accuracy of this code is not guaranteed. Consider using the built-in spa_c code or the PyEphem library.

Parameters

- **time** (*pandas.DatetimeIndex*) – Must be localized or UTC will be assumed.
- **latitude** (*float*) – Latitude in decimal degrees. Positive north of equator, negative to south.
- **longitude** (*float*) – Longitude in decimal degrees. Positive east of prime meridian, negative to west.
- **pressure** (*float or Series, default 101325*) – Ambient pressure (Pascals)
- **temperature** (*float or Series, default 12*) – Ambient temperature (C)

Returns

DataFrame with the following columns –

- apparent_elevation : apparent sun elevation accounting for atmospheric refraction.
- elevation : actual elevation (not accounting for refraction) of the sun in decimal degrees, 0 = on horizon. The complement of the zenith angle.
- azimuth : Azimuth of the sun in decimal degrees East of North. This is the complement of the apparent zenith angle.
- apparent_zenith : apparent sun zenith accounting for atmospheric refraction.
- zenith : Solar zenith angle

- solar_time : Solar time in decimal hours (solar noon is 12.00).

References

See also:

pyephem(), *spa_c()*, *spa_python()*

pvlib.solarposition.pyephem

pvlib.solarposition.**pyephem**(*time*, *latitude*, *longitude*, *altitude=0*, *pressure=101325*, *temperature=12*, *horizon='+0:00'*)

Calculate the solar position using the PyEphem package.

Parameters

- **time** (*pandas.DatetimeIndex*) – Must be localized or UTC will be assumed.
- **latitude** (*float*) – Latitude in decimal degrees. Positive north of equator, negative to south.
- **longitude** (*float*) – Longitude in decimal degrees. Positive east of prime meridian, negative to west.
- **altitude** (*float, default 0*) – Height above sea level in meters. [m]
- **pressure** (*int or float, optional, default 101325*) – air pressure in Pascals.
- **temperature** (*int or float, optional, default 12*) – air temperature in degrees C.
- **horizon** (*string, optional, default '+0:00'*) – arc degrees:arc minutes from geometrical horizon for sunrise and sunset, e.g., horizon='+0:00' to use sun center crossing the geometrical horizon to define sunrise and sunset, horizon='-0:34' for when the sun's upper edge crosses the geometrical horizon

Returns *pandas.DataFrame* – index is the same as input *time* argument The DataFrame will have the following columns: apparent_elevation, elevation, apparent_azimuth, azimuth, apparent_zenith, zenith.

See also:

spa_python(), *spa_c()*, *ephemeris()*

pvlib.solarposition.spa_c

pvlib.solarposition.**spa_c**(*time*, *latitude*, *longitude*, *pressure=101325*, *altitude=0*, *temperature=12*, *delta_t=67.0*, *raw_spa_output=False*)

Calculate the solar position using the C implementation of the NREL SPA code.

The source files for this code are located in './spa_c_files/', along with a README file which describes how the C code is wrapped in Python. Due to license restrictions, the C code must be downloaded seperately and used in accordance with it's license.

This function is slower and no more accurate than *spa_python()*.

Parameters

- **time** (*pandas.DatetimeIndex*) – Must be localized or UTC will be assumed.

- **latitude** (*float*) – Latitude in decimal degrees. Positive north of equator, negative to south.
- **longitude** (*float*) – Longitude in decimal degrees. Positive east of prime meridian, negative to west.
- **pressure** (*float, default 101325*) – Pressure in Pascals
- **altitude** (*float, default 0*) – Height above sea level. [m]
- **temperature** (*float, default 12*) – Temperature in C
- **delta_t** (*float, default 67.0*) – Difference between terrestrial time and UT1. USNO has previous values and predictions.
- **raw_spa_output** (*bool, default False*) – If true, returns the raw SPA output.

Returns *DataFrame* – The DataFrame will have the following columns: elevation, azimuth, zenith, apparent_elevation, apparent_zenith.

References

Note: The `timezone` field in the SPA C files is replaced with `time_zone` to avoid a nameclash with the function `__timezone` that is redefined by Python>=3.5. This issue is Python bug 24643.

See also:

pyephem(), *spa_python()*, *ephemeris()*

Additional functions for quantities closely related to solar position.

solarposition.calc_time(lower_bound, ..., ...)	Calculate the time between lower_bound and upper_bound where the attribute is equal to value.
solarposition.pyephem_earthsun_distance(time)	Calculates the distance from the earth to the sun using pyephem.
solarposition.nrel_earthsun_distance(time, ...)	Calculates the distance from the earth to the sun using the NREL SPA algorithm.
spa.calculate_deltat(year, month)	Calculate the difference between Terrestrial Dynamical Time (TD) and Universal Time (UT).

pvlib.solarposition.calc_time

pvlib.solarposition.**calc_time**(*lower_bound*, *upper_bound*, *latitude*, *longitude*, *attribute*, *value*, *altitude=0*, *pressure=101325*, *temperature=12*, *horizon='+0:00'*, *xtol=1e-12*)

Calculate the time between lower_bound and upper_bound where the attribute is equal to value. Uses PyEphem for solar position calculations.

Parameters

- **lower_bound** (*datetime.datetime*) –
- **upper_bound** (*datetime.datetime*) –
- **latitude** (*float*) – Latitude in decimal degrees. Positive north of equator, negative to south.
- **longitude** (*float*) – Longitude in decimal degrees. Positive east of prime meridian, negative to west.

- **attribute** (*str*) – The attribute of a pyephem.Sun object that you want to solve for. Likely options are 'alt' and 'az' (which must be given in radians).
- **value** (*int or float*) – The value of the attribute to solve for
- **altitude** (*float, default 0*) – Distance above sea level.
- **pressure** (*int or float, optional, default 101325*) – Air pressure in Pascals. Set to 0 for no atmospheric correction.
- **temperature** (*int or float, optional, default 12*) – Air temperature in degrees C.
- **horizon** (*string, optional, default '+0:00'*) – arc degrees:arc minutes from geometrical horizon for sunrise and sunset, e.g., horizon='+0:00' to use sun center crossing the geometrical horizon to define sunrise and sunset, horizon='-0:34' for when the sun's upper edge crosses the geometrical horizon
- **xtol** (*float, optional, default 1.0e-12*) – The allowed error in the result from value

Returns *datetime.datetime*

Raises

- ValueError – If the value is not contained between the bounds.
- AttributeError – If the given attribute is not an attribute of a PyEphem.Sun object.

pvlib.solarposition.pyephem_earthsun_distance

pvlib.solarposition.**pyephem_earthsun_distance**(*time*)

Calculates the distance from the earth to the sun using pyephem.

Parameters **time** (*pandas.DatetimeIndex*) – Must be localized or UTC will be assumed.

Returns *pd.Series. Earth-sun distance in AU.*

pvlib.solarposition.nrel_earthsun_distance

pvlib.solarposition.**nrel_earthsun_distance**(*time*, *how='numpy'*, *delta_t=67.0*, *numthreads=4*)

Calculates the distance from the earth to the sun using the NREL SPA algorithm.

The details of the NREL SPA algorithm are described in[1].

Parameters

- **time** (*pandas.DatetimeIndex*) – Must be localized or UTC will be assumed.
- **how** (*str, optional, default 'numpy'*) – Options are 'numpy' or 'numba'. If numba >= 0.17.0 is installed, how='numba' will compile the spa functions to machine code and run them multithreaded.
- **delta_t** (*float, optional, default 67.0*) – If delta_t is None, uses spa.calculate_deltat using time.year and time.month from pandas.DatetimeIndex. For most simulations specifing delta_t is sufficient. Difference between terrestrial time and UT1. *Note: delta_t = None will break code using nrel_numba, this will be fixed in a future version.* By default, use USNO historical data and predictions

[1] Reda, I., Andreas, A., 2003. Solar position algorithm for solar radiation applications. Technical report: NREL/TP-560- 34302. Golden, USA, http://www.nrel.gov.

- **numthreads** (*int, optional, default 4*) – Number of threads to use if how == 'numba'.

Returns **dist** (*pd.Series*) – Earth-sun distance in AU.

References

pvlib.spa.calculate_deltat

pvlib.spa.**calculate_deltat**(*year*, *month*)

Calculate the difference between Terrestrial Dynamical Time (TD) and Universal Time (UT).

Note: This function is not yet compatible for calculations using Numba.

Equations taken from http://eclipse.gsfc.nasa.gov/SEcat5/deltatpoly.html

Functions for calculating sunrise, sunset and transit times.

location.Location.get_sun_rise_set_transit(times)	Calculate sunrise, sunset and transit times.
solarposition.sun_rise_set_transit_ephem(...)	Calculate the next sunrise and sunset times using the PyEphem package.
solarposition.sun_rise_set_transit_spa(...)	Calculate the sunrise, sunset, and sun transit times using the NREL SPA algorithm.
solarposition.sun_rise_set_transit_geometric(...)	Geometric calculation of solar sunrise, sunset, and transit.

pvlib.location.Location.get_sun_rise_set_transit

Location.**get_sun_rise_set_transit**(*times*, *method='pyephem'*, ***kwargs*)

Calculate sunrise, sunset and transit times.

Parameters

- **times** (*DatetimeIndex*) – Must be localized to the Location
- **method** (*str, default 'pyephem'*) – 'pyephem', 'spa', or 'geometric'
- **are passed to the relevant functions. See** (*kwargs*) –
- **for details.** (*solarposition.sun_rise_set_transit_<method>*) –

Returns **result** (*DataFrame*) – Column names are: sunrise, sunset, transit.

pvlib.solarposition.sun_rise_set_transit_ephem

pvlib.solarposition.**sun_rise_set_transit_ephem**(*times*, *latitude*, *longitude*, *next_or_previous='next'*, *altitude=0*, *pressure=101325*, *temperature=12*, *horizon='0:00'*)

Calculate the next sunrise and sunset times using the PyEphem package.

Parameters

- **time** (*pandas.DatetimeIndex*) – Must be localized
- **latitude** (*float*) – Latitude in degrees, positive north of equator, negative to south

- **longitude** (*float*) – Longitude in degrees, positive east of prime meridian, negative to west
- **next_or_previous** (*str*) – 'next' or 'previous' sunrise and sunset relative to time
- **altitude** (*float, default 0*) – distance above sea level in meters.
- **pressure** (*int or float, optional, default 101325*) – air pressure in Pascals.
- **temperature** (*int or float, optional, default 12*) – air temperature in degrees C.
- **horizon** (*string, format +/-X:YY*) – arc degrees:arc minutes from geometrical horizon for sunrise and sunset, e.g., horizon='+0:00' to use sun center crossing the geometrical horizon to define sunrise and sunset, horizon='-0:34' for when the sun's upper edge crosses the geometrical horizon

Returns *pandas.DataFrame* – index is the same as input *time* argument columns are 'sunrise', 'sunset', and 'transit'

See also:

pyephem()

pvlib.solarposition.sun_rise_set_transit_spa

pvlib.solarposition.**sun_rise_set_transit_spa**(*times*, *latitude*, *longitude*, *how='numpy'*, *delta_t=67.0*, *numthreads=4*)

Calculate the sunrise, sunset, and sun transit times using the NREL SPA algorithm.

The details of the NREL SPA algorithm are described in[1].

If numba is installed, the functions can be compiled to machine code and the function can be multithreaded. Without numba, the function evaluates via numpy with a slight performance hit.

Parameters

- **times** (*pandas.DatetimeIndex*) – Must be localized to the timezone for `latitude` and `longitude`.
- **latitude** (*float*) – Latitude in degrees, positive north of equator, negative to south
- **longitude** (*float*) – Longitude in degrees, positive east of prime meridian, negative to west
- **delta_t** (*float, optional*) – If delta_t is None, uses spa.calculate_deltat using times.year and times.month from pandas.DatetimeIndex. For most simulations specifing delta_t is sufficient. Difference between terrestrial time and UT1. delta_t = None will break code using nrel_numba, this will be fixed in a future version. By default, use USNO historical data and predictions
- **how** (*str, optional, default 'numpy'*) – Options are 'numpy' or 'numba'. If numba >= 0.17.0 is installed, how='numba' will compile the spa functions to machine code and run them multithreaded.
- **numthreads** (*int, optional, default 4*) – Number of threads to use if how == 'numba'.

[1] Reda, I., Andreas, A., 2003. Solar position algorithm for solar radiation applications. Technical report: NREL/TP-560- 34302. Golden, USA, http://www.nrel.gov.

Returns *pandas.DataFrame* – index is the same as input *times* argument columns are 'sunrise', 'sunset', and 'transit'

References

pvlib.solarposition.sun_rise_set_transit_geometric

pvlib.solarposition.**sun_rise_set_transit_geometric**(*times*, *latitude*, *longitude*, *declination*, *equation_of_time*)

Geometric calculation of solar sunrise, sunset, and transit.

> **Warning:** The geometric calculation assumes a circular earth orbit with the sun as a point source at its center, and neglects the effect of atmospheric refraction on zenith. The error depends on location and time of year but is of order 10 minutes.

Parameters

- **times** (*pandas.DatetimeIndex*) – Corresponding timestamps, must be localized to the timezone for the `latitude` and `longitude`.
- **latitude** (*float*) – Latitude in degrees, positive north of equator, negative to south
- **longitude** (*float*) – Longitude in degrees, positive east of prime meridian, negative to west
- **declination** (*numeric*) – declination angle in radians at `times`
- **equation_of_time** (*numeric*) – difference in time between solar time and mean solar time in minutes

Returns

- **sunrise** (*datetime*) – localized sunrise time
- **sunset** (*datetime*) – localized sunset time
- **transit** (*datetime*) – localized sun transit time

References

The spa module contains the implementation of the built-in NREL SPA algorithm.

spa	Calculate the solar position using the NREL SPA algorithm either using numpy arrays or compiling the code to machine language with numba.

pvlib.spa

Calculate the solar position using the NREL SPA algorithm either using numpy arrays or compiling the code to machine language with numba.

Functions

aberration_correction(earth_radius_vector)	
apparent_sidereal_time(mean_sidereal_time, ...)	
apparent_sun_longitude(geocentric_longitude, ...)	
atmospheric_refraction_correction(...)	
calculate_deltat(year, month)	Calculate the difference between Terrestrial Dynamical Time (TD) and Universal Time (UT).
earthsun_distance(unixtime, delta_t, numthreads)	Calculates the distance from the earth to the sun using the NREL SPA algorithm described in [1].
equation_of_time(sun_mean_longitude, ...)	
equatorial_horizontal_parallax(...)	
geocentric_latitude(heliocentric_latitude)	
geocentric_longitude(heliocentric_longitude)	
geocentric_sun_declination(...)	
geocentric_sun_right_ascension(...)	
heliocentric_latitude(jme)	
heliocentric_longitude(jme)	
heliocentric_radius_vector(jme)	
jcompile(*args, **kwargs)	
julian_century(julian_day)	
julian_day(unixtime)	
julian_day_dt(year, month, day, hour, ...)	This is the original way to calculate the julian day from the NREL paper.
julian_ephemeris_century(julian_ephemeris_day)	
julian_ephemeris_day(julian_day, delta_t)	
julian_ephemeris_millennium(...)	
local_hour_angle(apparent_sidereal_time, ...)	Measured westward from south
longitude_nutation(julian_ephemeris_century, ...)	
mean_anomaly_moon(julian_ephemeris_century)	
mean_anomaly_sun(julian_ephemeris_century)	
mean_ecliptic_obliquity(...)	
mean_elongation(julian_ephemeris_century)	
mean_sidereal_time(julian_day, julian_century)	
moon_argument_latitude(julian_ephemeris_century)	
moon_ascending_longitude(...)	
nocompile(*args, **kwargs)	
obliquity_nutation(julian_ephemeris_century, ...)	
parallax_sun_right_ascension(xterm, ...)	
solar_position(unixtime, lat, lon, elev, ...)	Calculate the solar position using the NREL SPA algorithm described in [1].
solar_position_loop(unixtime, loc_args, out)	Loop through the time array and calculate the solar position
solar_position_numba(unixtime, lat, lon, ...)	Calculate the solar position using the numba compiled functions and multiple threads.
solar_position_numpy(unixtime, lat, lon, ...)	Calculate the solar position assuming unixtime is a numpy array.
sun_mean_longitude(julian_ephemeris_millennium)	
topocentric_astronomers_azimuth(...)	

Continued on next page

Table 13 – continued from previous page

topocentric_azimuth_angle(...)	
topocentric_elevation_angle(...)	
topocentric_elevation_angle_without_atmosphere(...)	
topocentric_local_hour_angle(...)	
topocentric_sun_declination(...)	
topocentric_sun_right_ascension(...)	
topocentric_zenith_angle(...)	
transit_sunrise_sunset(dates, lat, lon, ...)	Calculate the sun transit, sunrise, and sunset for a set of dates at a given location.
true_ecliptic_obliquity(...)	
uterm(observer_latitude)	
xterm(u, observer_latitude, observer_elevation)	
yterm(u, observer_latitude, observer_elevation)	

Correlations and analytical expressions for low precision solar position calculations.

solarposition.solar_zenith_analytical(...)	Analytical expression of solar zenith angle based on spherical trigonometry.
solarposition.solar_azimuth_analytical(...)	Analytical expression of solar azimuth angle based on spherical trigonometry.
solarposition.declination_spencer71(dayofyear)	Solar declination from Duffie & Beckman and attributed to Spencer (1971) and Iqbal (1983).
solarposition.declination_cooper69(dayofyear)	Solar declination from Duffie & Beckman and attributed to Cooper (1969).
solarposition.equation_of_time_spencer71(...)	Equation of time from Duffie & Beckman and attributed to Spencer (1971) and Iqbal (1983).
solarposition.equation_of_time_pvcdrom(dayofyear)	Equation of time from PVCDROM.
solarposition.hour_angle(times, longitude, ...)	Hour angle in local solar time.
solarposition.sun_rise_set_transit_geometric(...)	Geometric calculation of solar sunrise, sunset, and transit.

pvlib.solarposition.solar_zenith_analytical

pvlib.solarposition.**solar_zenith_analytical**(*latitude*, *hourangle*, *declination*)

Analytical expression of solar zenith angle based on spherical trigonometry.

> **Warning:** The analytic form neglects the effect of atmospheric refraction.

Parameters

- **latitude** (*numeric*) – Latitude of location in radians.
- **hourangle** (*numeric*) – Hour angle in the local solar time in radians.
- **declination** (*numeric*) – Declination of the sun in radians.

Returns **zenith** (*numeric*) – Solar zenith angle in radians.

References

See also:

declination_spencer71(), *declination_cooper69()*, *hour_angle()*

pvlib.solarposition.solar_azimuth_analytical

pvlib.solarposition.**solar_azimuth_analytical**(*latitude*, *hourangle*, *declination*, *zenith*)

Analytical expression of solar azimuth angle based on spherical trigonometry.

Parameters

- **latitude** (*numeric*) – Latitude of location in radians.
- **hourangle** (*numeric*) – Hour angle in the local solar time in radians.
- **declination** (*numeric*) – Declination of the sun in radians.
- **zenith** (*numeric*) – Solar zenith angle in radians.

Returns **azimuth** (*numeric*) – Solar azimuth angle in radians.

References

See also:

declination_spencer71(), *declination_cooper69()*, *hour_angle()*, *solar_zenith_analytical()*

pvlib.solarposition.declination_spencer71

pvlib.solarposition.**declination_spencer71**(*dayofyear*)

Solar declination from Duffie & Beckman and attributed to Spencer (1971) and Iqbal (1983).

See[1] for details.

Warning: Return units are radians, not degrees.

Parameters **dayofyear** (*numeric*) –

Returns **declination (radians)** (*numeric*) – Angular position of the sun at solar noon relative to the plane of the equator, approximately between +/-23.45 (degrees).

References

See also:

declination_cooper69()

[1] J. A. Duffie and W. A. Beckman, "Solar Engineering of Thermal Processes, 3rd Edition" pp. 13-14, J. Wiley and Sons, New York (2006)

pvlib.solarposition.declination_cooper69

pvlib.solarposition.**declination_cooper69**(*dayofyear*)

Solar declination from Duffie & Beckman and attributed to Cooper (1969).

See[1] for details.

Warning: Return units are radians, not degrees.

Declination can be expressed using either sine or cosine:

$$\delta = 23.45 \sin\left(\frac{2\pi}{365}\left(n_{day} + 284\right)\right) = -23.45 \cos\left(\frac{2\pi}{365}\left(n_{day} + 10\right)\right)$$

Parameters **dayofyear** (*numeric*) –

Returns **declination (radians)** (*numeric*) – Angular position of the sun at solar noon relative to the plane of the equator, approximately between +/-23.45 (degrees).

References

See also:

declination_spencer71()

pvlib.solarposition.equation_of_time_spencer71

pvlib.solarposition.**equation_of_time_spencer71**(*dayofyear*)

Equation of time from Duffie & Beckman and attributed to Spencer (1971) and Iqbal (1983).

The coefficients correspond to the online copy of the Fourier paper[1] in the Sundial Mailing list that was posted in 1998 by Mac Oglesby from his correspondence with Macquarie University Prof. John Pickard who added the following note.

> In the early 1970s, I contacted Dr Spencer about this method because I was trying to use a hand calculator for calculating solar positions, etc. He was extremely helpful and gave me a reprint of this paper. He also pointed out an error in the original: in the series for E, the constant was printed as 0.000075 rather than 0.0000075. I have corrected the error in this version.

There appears to be another error in formula as printed in both Duffie & Beckman's[2] and Frank Vignola's[3] books in which the coefficient 0.04089 is printed instead of 0.040849, corresponding to the value used in the Bird Clear Sky model implemented by Daryl Myers[4] and printed in both the Fourier paper from the Sundial Mailing List and R. Hulstrom's[5] book.

Parameters **dayofyear** (*numeric*) –

Returns **equation_of_time** (*numeric*) – Difference in time between solar time and mean solar time in minutes.

[1] J. A. Duffie and W. A. Beckman, "Solar Engineering of Thermal Processes, 3rd Edition" pp. 13-14, J. Wiley and Sons, New York (2006)
[1] J. W. Spencer, "Fourier series representation of the position of the sun" in Search 2 (5), p. 172 (1971)
[2] J. A. Duffie and W. A. Beckman, "Solar Engineering of Thermal Processes, 3rd Edition" pp. 9-11, J. Wiley and Sons, New York (2006)
[3] Frank Vignola et al., "Solar And Infrared Radiation Measurements", p. 13, CRC Press (2012)
[4] Daryl R. Myers, "Solar Radiation: Practical Modeling for Renewable Energy Applications", p. 5 CRC Press (2013)
[5] Roland Hulstrom, "Solar Resources" p. 66, MIT Press (1989)

References

See also:

equation_of_time_pvcdrom()

pvlib.solarposition.equation_of_time_pvcdrom

pvlib.solarposition.**equation_of_time_pvcdrom**(*dayofyear*)
Equation of time from PVCDROM.

PVCDROM is a website by Solar Power Lab at Arizona State University (ASU)

Parameters **dayofyear** (*numeric*) –

Returns **equation_of_time** (*numeric*) – Difference in time between solar time and mean solar time in minutes.

References

See also:

equation_of_time_spencer71()

pvlib.solarposition.hour_angle

pvlib.solarposition.**hour_angle**(*times*, *longitude*, *equation_of_time*)
Hour angle in local solar time. Zero at local solar noon.

Parameters

- **times** (pandas.DatetimeIndex) – Corresponding timestamps, must be localized to the timezone for the longitude.
- **longitude** (*numeric*) – Longitude in degrees
- **equation_of_time** (*numeric*) – Equation of time in minutes.

Returns **hour_angle** (*numeric*) – Hour angle in local solar time in degrees.

References

See also:

equation_of_time_spencer71(), *equation_of_time_pvcdrom()*

3.12.3 Clear sky

location.Location.get_clearsky(times[, ...])	Calculate the clear sky estimates of GHI, DNI, and/or DHI at this location.
clearsky.ineichen(apparent_zenith, ...[, ...])	Determine clear sky GHI, DNI, and DHI from Ineichen/Perez model.

Continued on next page

Table 15 – continued from previous page

clearsky.lookup_linke_turbidity(time, ...[, ...])	Look up the Linke Turbidity from the `LinkeTurbidities.h5` data file supplied with pvlib.
clearsky.simplified_solis(apparent_elevation)	Calculate the clear sky GHI, DNI, and DHI according to the simplified Solis model.
clearsky.haurwitz(apparent_zenith)	Determine clear sky GHI using the Haurwitz model.
clearsky.detect_clearsky(measured, clearsky, ...)	Detects clear sky times according to the algorithm developed by Reno and Hansen for GHI measurements.
clearsky.bird(zenith, airmass_relative, ...)	Bird Simple Clear Sky Broadband Solar Radiation Model

pvlib.location.Location.get_clearsky

`Location.`**`get_clearsky`**(*times*, *model='ineichen'*, *solar_position=None*, *dni_extra=None*, ***kwargs*)

Calculate the clear sky estimates of GHI, DNI, and/or DHI at this location.

Parameters

- **times** (*DatetimeIndex*) –
- **model** (*str, default 'ineichen'*) – The clear sky model to use. Must be one of 'ineichen', 'haurwitz', 'simplified_solis'.
- **solar_position** (*None or DataFrame, default None*) – DataFrame with columns 'apparent_zenith', 'zenith', 'apparent_elevation'.
- **dni_extra** (*None or numeric, default None*) – If None, will be calculated from times.
- **kwargs** – Extra parameters passed to the relevant functions. Climatological values are assumed in many cases. See source code for details!

Returns **clearsky** (*DataFrame*) – Column names are: `ghi, dni, dhi`.

pvlib.clearsky.ineichen

`pvlib.clearsky.`**`ineichen`**(*apparent_zenith*, *airmass_absolute*, *linke_turbidity*, *altitude=0*, *dni_extra=1364.0*, *perez_enhancement=False*)

Determine clear sky GHI, DNI, and DHI from Ineichen/Perez model.

Implements the Ineichen and Perez clear sky model for global horizontal irradiance (GHI), direct normal irradiance (DNI), and calculates the clear-sky diffuse horizontal (DHI) component as the difference between GHI and DNI*cos(zenith) as presented in [1, 2]. A report on clear sky models found the Ineichen/Perez model to have excellent performance with a minimal input data set [3].

Default values for monthly Linke turbidity provided by SoDa [4, 5].

Parameters

- **apparent_zenith** (*numeric*) – Refraction corrected solar zenith angle in degrees.
- **airmass_absolute** (*numeric*) – Pressure corrected airmass.
- **linke_turbidity** (*numeric*) – Linke Turbidity.
- **altitude** (*numeric, default 0*) – Altitude above sea level in meters.
- **dni_extra** (*numeric, default 1364*) – Extraterrestrial irradiance. The units of `dni_extra` determine the units of the output.

- **perez_enhancement** (*bool, default False*) – Controls if the Perez enhancement factor should be applied. Setting to True may produce spurious results for times when the Sun is near the horizon and the airmass is high. See https://github.com/pvlib/pvlib-python/issues/435

Returns **clearsky** (*DataFrame (if Series input) or OrderedDict of arrays*) – DataFrame/OrderedDict contains the columns/keys `'dhi', 'dni', 'ghi'`.

See also:

lookup_linke_turbidity(), *pvlib.location.Location.get_clearsky()*

References

pvlib.clearsky.lookup_linke_turbidity

pvlib.clearsky.**lookup_linke_turbidity**(*time*, *latitude*, *longitude*, *filepath=None*, *interp_turbidity=True*)

Look up the Linke Turibidity from the `LinkeTurbidities.h5` data file supplied with pvlib.

Parameters

- **time** (*pandas.DatetimeIndex*) –
- **latitude** (*float or int*) –
- **longitude** (*float or int*) –
- **filepath** (*None or string, default None*) – The path to the `.h5` file.
- **interp_turbidity** (*bool, default True*) – If `True`, interpolates the monthly Linke turbidity values found in `LinkeTurbidities.h5` to daily values.

Returns **turbidity** (*Series*)

pvlib.clearsky.simplified_solis

pvlib.clearsky.**simplified_solis**(*apparent_elevation*, *aod700=0.1*, *precipitable_water=1.0*, *pressure=101325.0*, *dni_extra=1364.0*)

Calculate the clear sky GHI, DNI, and DHI according to the simplified Solis model.

Reference[1] describes the accuracy of the model as being 15, 20, and 18 W/m^2 for the beam, global, and diffuse components. Reference[2] provides comparisons with other clear sky models.

Parameters

- **apparent_elevation** (*numeric*) – The apparent elevation of the sun above the horizon (deg).
- **aod700** (*numeric, default 0.1*) – The aerosol optical depth at 700 nm (unitless). Algorithm derived for values between 0 and 0.45.
- **precipitable_water** (*numeric, default 1.0*) – The precipitable water of the atmosphere (cm). Algorithm derived for values between 0.2 and 10 cm. Values less than 0.2 will be assumed to be equal to 0.2.
- **pressure** (*numeric, default 101325.0*) – The atmospheric pressure (Pascals). Algorithm derived for altitudes between sea level and 7000 m, or 101325 and 41000 Pascals.

[1] P. Ineichen, "A broadband simplified version of the Solis clear sky model," Solar Energy, 82, 758-762 (2008).
[2] P. Ineichen, "Validation of models that estimate the clear sky global and beam solar irradiance," Solar Energy, 132, 332-344 (2016).

- **dni_extra** (*numeric, default 1364.0*) – Extraterrestrial irradiance. The units of `dni_extra` determine the units of the output.

Returns **clearsky** (*DataFrame (if Series input) or OrderedDict of arrays*) – DataFrame/OrderedDict contains the columns/keys `'dhi', 'dni', 'ghi'`.

References

pvlib.clearsky.haurwitz

`pvlib.clearsky.`**`haurwitz`**(*apparent_zenith*)

Determine clear sky GHI using the Haurwitz model.

Implements the Haurwitz clear sky model for global horizontal irradiance (GHI) as presented in [1, 2]. A report on clear sky models found the Haurwitz model to have the best performance in terms of average monthly error among models which require only zenith angle [3].

Parameters **apparent_zenith** (*Series*) – The apparent (refraction corrected) sun zenith angle in degrees.

Returns **ghi** (*DataFrame*) – The modeled global horizonal irradiance in W/m^2 provided by the Haurwitz clear-sky model.

References

pvlib.clearsky.detect_clearsky

`pvlib.clearsky.`**`detect_clearsky`**(*measured*, *clearsky*, *times*, *window_length*, *mean_diff=75*, *max_diff=75*, *lower_line_length=-5*, *upper_line_length=10*, *var_diff=0.005*, *slope_dev=8*, *max_iterations=20*, *return_components=False*)

Detects clear sky times according to the algorithm developed by Reno and Hansen for GHI measurements. The algorithm[1] was designed and validated for analyzing GHI time series only. Users may attempt to apply it to other types of time series data using different filter settings, but should be skeptical of the results.

The algorithm detects clear sky times by comparing statistics for a measured time series and an expected clearsky time series. Statistics are calculated using a sliding time window (e.g., 10 minutes). An iterative algorithm identifies clear periods, uses the identified periods to estimate bias in the clearsky data, scales the clearsky data and repeats.

Clear times are identified by meeting 5 criteria. Default values for these thresholds are appropriate for 10 minute windows of 1 minute GHI data.

Parameters

- **measured** (*array or Series*) – Time series of measured values.
- **clearsky** (*array or Series*) – Time series of the expected clearsky values.
- **times** (*DatetimeIndex*) – Times of measured and clearsky values.
- **window_length** (*int*) – Length of sliding time window in minutes. Must be greater than 2 periods.
- **mean_diff** (*float, default 75*) – Threshold value for agreement between mean values of measured and clearsky in each interval, see Eq. 6 in [1].

[1] Reno, M.J. and C.W. Hansen, "Identification of periods of clear sky irradiance in time series of GHI measurements" Renewable Energy, v90, p. 520-531, 2016.

- **max_diff** (*float, default 75*) – Threshold value for agreement between maxima of measured and clearsky values in each interval, see Eq. 7 in [1].
- **lower_line_length** (*float, default -5*) – Lower limit of line length criterion from Eq. 8 in [1]. Criterion satisfied when lower_line_length < line length difference < upper_line_length
- **upper_line_length** (*float, default 10*) – Upper limit of line length criterion from Eq. 8 in [1].
- **var_diff** (*float, default 0.005*) – Threshold value in Hz for the agreement between normalized standard deviations of rate of change in irradiance, see Eqs. 9 through 11 in [1].
- **slope_dev** (*float, default 8*) – Threshold value for agreement between the largest magnitude of change in successive values, see Eqs. 12 through 14 in [1].
- **max_iterations** (*int, default 20*) – Maximum number of times to apply a different scaling factor to the clearsky and redetermine clear_samples. Must be 1 or larger.
- **return_components** (*bool, default False*) – Controls if additional output should be returned. See below.

Returns

- **clear_samples** (*array or Series*) – Boolean array or Series of whether or not the given time is clear. Return type is the same as the input type.
- **components** (*OrderedDict, optional*) – Dict of arrays of whether or not the given time window is clear for each condition. Only provided if return_components is True.
- **alpha** (*scalar, optional*) – Scaling factor applied to the clearsky_ghi to obtain the detected clear_samples. Only provided if return_components is True.

References

Notes

Initial implementation in MATLAB by Matthew Reno. Modifications for computational efficiency by Joshua Patrick and Curtis Martin. Ported to Python by Will Holmgren, Tony Lorenzo, and Cliff Hansen.

Differences from MATLAB version:

- no support for unequal times
- automatically determines sample_interval
- requires a reference clear sky series instead calculating one from a user supplied location and UTCoffset
- parameters are controllable via keyword arguments
- option to return individual test components and clearsky scaling parameter

pvlib.clearsky.bird

pvlib.clearsky.**bird**(*zenith*, *airmass_relative*, *aod380*, *aod500*, *precipitable_water*, *ozone=0.3*, *pressure=101325.0*, *dni_extra=1364.0*, *asymmetry=0.85*, *albedo=0.2*)

Bird Simple Clear Sky Broadband Solar Radiation Model

Based on NREL Excel implementation by Daryl R. Myers [1, 2].

Bird and Hulstrom define the zenith as the "angle between a line to the sun and the local zenith". There is no distinction in the paper between solar zenith and apparent (or refracted) zenith, but the relative airmass is defined using the Kasten 1966 expression, which requires apparent zenith. Although the formulation for calculated zenith is never explicitly defined in the report, since the purpose was to compare existing clear sky models with "rigorous radiative transfer models" (RTM) it is possible that apparent zenith was obtained as output from the RTM. However, the implentation presented in PVLIB is tested against the NREL Excel implementation by Daryl Myers which uses an analytical expression for solar zenith instead of apparent zenith.

Parameters

- **zenith** (*numeric*) – Solar or apparent zenith angle in degrees - see note above
- **airmass_relative** (*numeric*) – Relative airmass
- **aod380** (*numeric*) – Aerosol optical depth [cm] measured at 380[nm]
- **aod500** (*numeric*) – Aerosol optical depth [cm] measured at 500[nm]
- **precipitable_water** (*numeric*) – Precipitable water [cm]
- **ozone** (*numeric*) – Atmospheric ozone [cm], defaults to 0.3[cm]
- **pressure** (*numeric*) – Ambient pressure [Pa], defaults to 101325[Pa]
- **dni_extra** (*numeric*) – Extraterrestrial radiation [W/m^2], defaults to 1364[W/m^2]
- **asymmetry** (*numeric*) – Asymmetry factor, defaults to 0.85
- **albedo** (*numeric*) – Albedo, defaults to 0.2

Returns clearsky (*DataFrame (if Series input) or OrderedDict of arrays*) – DataFrame/OrderedDict contains the columns/keys `'dhi', 'dni', 'ghi', 'direct_horizontal'` in [W/m^2].

See also:

pvlib.atmosphere.bird_hulstrom80_aod_bb(), *pvlib.atmosphere.get_relative_airmass()*

References

3.12.4 Airmass and atmospheric models

location.Location.get_airmass([times, ...])	Calculate the relative and absolute airmass.
atmosphere.get_absolute_airmass(airmass_relative)	Determine absolute (pressure-adjusted) airmass from relative airmass and pressure.
atmosphere.get_relative_airmass(zenith[, model])	Calculate relative (not pressure-adjusted) airmass at sea level.
atmosphere.pres2alt(pressure)	Determine altitude from site pressure.
atmosphere.alt2pres(altitude)	Determine site pressure from altitude.
atmosphere.gueymard94_pw(temp_air, ...)	Calculates precipitable water (cm) from ambient air temperature (C) and relatively humidity (%) using an empirical model.
atmosphere.first_solar_spectral_correction(pw, ...)	Spectral mismatch modifier based on precipitable water and absolute (pressure-adjusted) airmass.
atmosphere.bird_hulstrom80_aod_bb(aod380, aod500)	Approximate broadband aerosol optical depth.

Continued on next page

Table 16 – continued from previous page

atmosphere.kasten96_lt(airmass_absolute, ...)	Calculate Linke turbidity using Kasten pyrheliometric formula.
atmosphere.angstrom_aod_at_lambda(aod0, lambda0)	Get AOD at specified wavelength using Angstrom turbidity model.
atmosphere.angstrom_alpha(aod1, lambda1, ...)	Calculate Angstrom alpha exponent.

pvlib.location.Location.get_airmass

Location.**get_airmass**(*times=None*, *solar_position=None*, *model='kastenyoung1989'*)

Calculate the relative and absolute airmass.

Automatically chooses zenith or apparant zenith depending on the selected model.

Parameters

- **times** (*None or DatetimeIndex, default None*) – Only used if solar_position is not provided.
- **solar_position** (*None or DataFrame, default None*) – DataFrame with with columns 'apparent_zenith', 'zenith'.
- **model** (*str, default 'kastenyoung1989'*) – Relative airmass model. See *pvlib.atmosphere.get_relative_airmass()* for a list of available models.

Returns **airmass** (*DataFrame*) – Columns are 'airmass_relative', 'airmass_absolute'

See also:

pvlib.atmosphere.get_relative_airmass()

pvlib.atmosphere.get_absolute_airmass

pvlib.atmosphere.**get_absolute_airmass**(*airmass_relative*, *pressure=101325.0*)

Determine absolute (pressure-adjusted) airmass from relative airmass and pressure.

The calculation for absolute airmass (AM_a) is

$$AM_a = AM_r \frac{P}{101325}$$

where AM_r is relative air mass at sea level and P is atmospheric pressure.

Parameters

- **airmass_relative** (*numeric*) – The airmass at sea level. [unitless]
- **pressure** (*numeric, default 101325*) – Atmospheric pressure. [Pa]

Returns **airmass_absolute** (*numeric*) – Absolute (pressure-adjusted) airmass

References

pvlib.atmosphere.get_relative_airmass

pvlib.atmosphere.**get_relative_airmass**(*zenith*, *model='kastenyoung1989'*)

Calculate relative (not pressure-adjusted) airmass at sea level.

Parameter `model` allows selection of different airmass models.

Parameters

- **zenith** (*numeric*) – Zenith angle of the sun. [degrees]
- **model** (*string, default 'kastenyoung1989'*) – Available models include the following:
  - 'simple' - secant(apparent zenith angle) - Note that this gives -Inf at zenith=90
  - 'kasten1966' - See reference [1] - requires apparent sun zenith
  - 'youngirvine1967' - See reference [2] - requires true sun zenith
  - 'kastenyoung1989' (default) - See reference [3] - requires apparent sun zenith
  - 'gueymard1993' - See reference [4] - requires apparent sun zenith
  - 'young1994' - See reference [5] - requries true sun zenith
  - 'pickering2002' - See reference [6] - requires apparent sun zenith

Returns **airmass_relative** (*numeric*) – Relative airmass at sea level. Returns NaN values for any zenith angle greater than 90 degrees. [unitless]

Notes

Some models use apparent (refraction-adjusted) zenith angle while other models use true (not refraction-adjusted) zenith angle. Apparent zenith angles should be calculated at sea level.

References

pvlib.atmosphere.pres2alt

pvlib.atmosphere.**pres2alt**(*pressure*)

Determine altitude from site pressure.

Parameters **pressure** (*numeric*) – Atmospheric pressure. [Pa]

Returns **altitude** (*numeric*) – Altitude above sea level. [m]

Notes

The following assumptions are made

Parameter	Value
Base pressure	101325 Pa
Temperature at zero altitude	288.15 K
Gravitational acceleration	9.80665 m/s^2
Lapse rate	-6.5E-3 K/m
Gas constant for air	287.053 J/(kg K)
Relative Humidity	0%

References

pvlib.atmosphere.alt2pres

pvlib.atmosphere.**alt2pres**(*altitude*)

Determine site pressure from altitude.

Parameters **altitude** (*numeric*) – Altitude above sea level. [m]

Returns **pressure** (*numeric*) – Atmospheric pressure. [Pa]

Notes

The following assumptions are made

Parameter	Value
Base pressure	101325 Pa
Temperature at zero altitude	288.15 K
Gravitational acceleration	9.80665 m/s^2
Lapse rate	-6.5E-3 K/m
Gas constant for air	287.053 J/(kg K)
Relative Humidity	0%

References

pvlib.atmosphere.gueymard94_pw

pvlib.atmosphere.**gueymard94_pw**(*temp_air*, *relative_humidity*)

Calculates precipitable water (cm) from ambient air temperature (C) and relatively humidity (%) using an empirical model. The accuracy of this method is approximately 20% for moderate PW (1-3 cm) and less accurate otherwise.

The model was developed by expanding Eq. 1 in[2]:

$$Pw = 0.1 H_v \rho_v$$

using Eq. 2 in[2]

$$\rho_v = 216.7 R_H e_s / T$$

Pw is the precipitable water (cm), H_v is the apparent water vapor scale height (km) and ρ_v is the surface water vapor density (g/m^3). . The expression for H_v is Eq. 4 in[2]:

$$H_v = 0.4976 + 1.5265 \frac{T}{273.15} + \exp\left(13.6897 \frac{T}{273.15} - 14.9188 \left(\frac{T}{273.15}\right)^3\right)$$

In the expression for ρ_v, e_s is the saturation water vapor pressure (millibar). The expression for e_s is Eq. 1 in[3]

$$e_s = \exp\left(22.330 - 49.140 \frac{100}{T} - 10.922 \left(\frac{100}{T}\right)^2 - 0.39015 \frac{T}{100}\right)$$

[2] C. Gueymard, Analysis of Monthly Average Atmospheric Precipitable Water and Turbidity in Canada and Northern United States, Solar Energy vol 53(1), pp. 57-71, 1994.

[3] C. Gueymard, Assessment of the Accuracy and Computing Speed of simplified saturation vapor equations using a new reference dataset, J. of Applied Meteorology 1993, vol. 32(7), pp. 1294-1300.

Parameters

- **temp_air** (*numeric*) – ambient air temperature T at the surface. [C]
- **relative_humidity** (*numeric*) – relative humidity R_H at the surface. [%]

Returns **pw** (*numeric*) – precipitable water. [cm]

References

pvlib.atmosphere.first_solar_spectral_correction

pvlib.atmosphere.**first_solar_spectral_correction**(*pw*, *airmass_absolute*, *module_type=None*, *coefficients=None*, *min_pw=0.1*, *max_pw=8*)

Spectral mismatch modifier based on precipitable water and absolute (pressure-adjusted) airmass.

Estimates a spectral mismatch modifier M representing the effect on module short circuit current of variation in the spectral irradiance. M is estimated from absolute (pressure currected) air mass, AM_a, and precipitable water, Pw, using the following function:

$$M = c_1 + c_2 AM_a + c_3 Pw + c_4 AM_a^{0.5} + c_5 Pw^{0.5} + c_6 \frac{AM_a}{Pw^{0.5}}$$

Default coefficients are determined for several cell types with known quantum efficiency curves, by using the Simple Model of the Atmospheric Radiative Transfer of Sunshine (SMARTS)[1]. Using SMARTS, spectrums are simulated with all combinations of AMa and Pw where:

- 0.5cm <= Pw <= 5cm
- 1.0 <= AM_a <= 5.0
- Spectral range is limited to that of CMP11 (280 nm to 2800 nm)
- spectrum simulated on a plane normal to the sun
- All other parameters fixed at G173 standard

From these simulated spectra, M is calculated using the known quantum efficiency curves. Multiple linear regression is then applied to fit Eq. 1 to determine the coefficients for each module.

Based on the PVLIB Matlab function `pvl_FSspeccorr` by Mitchell Lee and Alex Panchula of First Solar, 2016[2].

Parameters

- **pw** (*array-like*) – atmospheric precipitable water. [cm]
- **airmass_absolute** (*array-like*) – absolute (pressure-adjusted) airmass. [unitless]
- **min_pw** (*float, default 0.1*) – minimum atmospheric precipitable water. Any pw value lower than min_pw is set to min_pw to avoid model divergence. [cm]
- **max_pw** (*float, default 8*) – maximum atmospheric precipitable water. Any pw value higher than max_pw is set to NaN to avoid model divergence. [cm]

[1] Gueymard, Christian. SMARTS2: a simple model of the atmospheric radiative transfer of sunshine: algorithms and performance assessment. Cocoa, FL: Florida Solar Energy Center, 1995.

[2] Lee, Mitchell, and Panchula, Alex. "Spectral Correction for Photovoltaic Module Performance Based on Air Mass and Precipitable Water." IEEE Photovoltaic Specialists Conference, Portland, 2016

- **module_type** (*None or string, default None*) – a string specifying a cell type. Values of 'cdte', 'monosi', 'xsi', 'multisi', and 'polysi' (can be lower or upper case). If provided, module_type selects default coefficients for the following modules:
  - 'cdte' - First Solar Series 4-2 CdTe module.
  - 'monosi', 'xsi' - First Solar TetraSun module.
  - 'multisi', 'polysi' - anonymous multi-crystalline silicon module.
  - 'cigs' - anonymous copper indium gallium selenide module.
  - 'asi' - anonymous amorphous silicon module.

  The module used to calculate the spectral correction coefficients corresponds to the Multi-crystalline silicon Manufacturer 2 Model C from[3]. The spectral response (SR) of CIGS and a-Si modules used to derive coefficients can be found in[4]
- **coefficients** (*None or array-like, default None*) – Allows for entry of user-defined spectral correction coefficients. Coefficients must be of length 6. Derivation of coefficients requires use of SMARTS and PV module quantum efficiency curve. Useful for modeling PV module types which are not included as defaults, or to fine tune the spectral correction to a particular PV module. Note that the parameters for modules with very similar quantum efficiency should be similar, in most cases limiting the need for module specific coefficients.

Returns **modifier** (*array-like*) – spectral mismatch factor (unitless) which is can be multiplied with broadband irradiance reaching a module's cells to estimate effective irradiance, i.e., the irradiance that is converted to electrical current.

References

pvlib.atmosphere.bird_hulstrom80_aod_bb

pvlib.atmosphere.**bird_hulstrom80_aod_bb**(*aod380*, *aod500*)

Approximate broadband aerosol optical depth.

Bird and Hulstrom developed a correlation for broadband aerosol optical depth (AOD) using two wavelengths, 380 nm and 500 nm.

Parameters

- **aod380** (*numeric*) – AOD measured at 380 nm. [unitless]
- **aod500** (*numeric*) – AOD measured at 500 nm. [unitless]

Returns **aod_bb** (*numeric*) – Broadband AOD. [unitless]

See also:

pvlib.atmosphere.kasten96_lt()

References

[3] Marion, William F., et al. User's Manual for Data for Validating Models for PV Module Performance. National Renewable Energy Laboratory, 2014. http://www.nrel.gov/docs/fy14osti/61610.pdf

[4] Schweiger, M. and Hermann, W, Influence of Spectral Effects on Energy Yield of Different PV Modules: Comparison of Pwat and MMF Approach, TUV Rheinland Energy GmbH report 21237296.003, January 2017

pvlib.atmosphere.kasten96_lt

pvlib.atmosphere.**kasten96_lt**(*airmass_absolute*, *precipitable_water*, *aod_bb*)
Calculate Linke turbidity using Kasten pyrheliometric formula.

Note that broadband aerosol optical depth (AOD) can be approximated by AOD measured at 700 nm according to Molineaux [4] . Bird and Hulstrom offer an alternate approximation using AOD measured at 380 nm and 500 nm.

Based on original implementation by Armel Oumbe.

Warning: These calculations are only valid for airmass less than 5 and precipitable water less than 5 cm.

Parameters

- **airmass_absolute** (*numeric*) – Pressure-adjusted airmass. [unitless]
- **precipitable_water** (*numeric*) – Precipitable water. [cm]
- **aod_bb** (*numeric*) – broadband AOD. [unitless]

Returns **lt** (*numeric*) – Linke turbidity. [unitless]

See also:

pvlib.atmosphere.bird_hulstrom80_aod_bb(), *pvlib.atmosphere.angstrom_aod_at_lambda()*

References

pvlib.atmosphere.angstrom_aod_at_lambda

pvlib.atmosphere.**angstrom_aod_at_lambda**(*aod0*, *lambda0*, *alpha=1.14*, *lambda1=700.0*)
Get AOD at specified wavelength using Angstrom turbidity model.

Parameters

- **aod0** (*numeric*) – Aerosol optical depth (AOD) measured at wavelength `lambda0`. [unitless]
- **lambda0** (*numeric*) – Wavelength corresponding to `aod0`. [nm]
- **alpha** (*numeric, default 1.14*) – Angstrom α exponent corresponding to `aod0`. [unitless]
- **lambda1** (*numeric, default 700*) – Desired wavelength. [nm]

Returns **aod1** (*numeric*) – AOD at desired wavelength `lambda1`. [unitless]

See also:

pvlib.atmosphere.angstrom_alpha()

References

pvlib.atmosphere.angstrom_alpha

pvlib.atmosphere.**angstrom_alpha**(*aod1*, *lambda1*, *aod2*, *lambda2*)
Calculate Angstrom alpha exponent.

Parameters

- **aod1** (*numeric*) – Aerosol optical depth at wavelength `lambda1`. [unitless]
- **lambda1** (*numeric*) – Wavelength corresponding to `aod1`. [nm]
- **aod2** (*numeric*) – Aerosol optical depth at wavelength `lambda2`. [unitless]
- **lambda2** (*numeric*) – Wavelength corresponding to `aod2`. [nm]

Returns **alpha** (*numeric*) – Angstrom α exponent for wavelength in `(lambda1, lambda2)`. [unitless]

See also:

pvlib.atmosphere.angstrom_aod_at_lambda()

3.12.5 Irradiance

Methods for irradiance calculations

pvsystem.PVSystem.get_irradiance(...[, ...])	Uses the *irradiance.get_total_irradiance()* function to calculate the plane of array irradiance components on a tilted surface defined by `self.surface_tilt`, `self.surface_azimuth`, and `self.albedo`.
pvsystem.PVSystem.get_aoi(solar_zenith, ...)	Get the angle of incidence on the system.
tracking.SingleAxisTracker.get_irradiance(...)	Uses the *irradiance.get_total_irradiance()* function to calculate the plane of array irradiance components on a tilted surface defined by the input data and `self.albedo`.

pvlib.pvsystem.PVSystem.get_irradiance

PVSystem.**get_irradiance**(*solar_zenith*, *solar_azimuth*, *dni*, *ghi*, *dhi*, *dni_extra=None*, *airmass=None*, *model='haydavies'*, ***kwargs*)

Uses the `irradiance.get_total_irradiance()` function to calculate the plane of array irradiance components on a tilted surface defined by `self.surface_tilt`, `self.surface_azimuth`, and `self.albedo`.

Parameters

- **solar_zenith** (*float or Series.*) – Solar zenith angle.
- **solar_azimuth** (*float or Series.*) – Solar azimuth angle.
- **dni** (*float or Series*) – Direct Normal Irradiance
- **ghi** (*float or Series*) – Global horizontal irradiance
- **dhi** (*float or Series*) – Diffuse horizontal irradiance
- **dni_extra** (*None, float or Series, default None*) – Extraterrestrial direct normal irradiance
- **airmass** (*None, float or Series, default None*) – Airmass
- **model** (*String, default 'haydavies'*) – Irradiance model.
- **kwargs** – Extra parameters passed to `irradiance.get_total_irradiance()`.

Returns **poa_irradiance** (*DataFrame*) – Column names are: `total, beam, sky, ground`.

pvlib.pvsystem.PVSystem.get_aoi

`PVSystem.`**`get_aoi`**(*solar_zenith*, *solar_azimuth*)

Get the angle of incidence on the system.

Parameters

- **`solar_zenith`** (*`float or Series.`*) – Solar zenith angle.
- **`solar_azimuth`** (*`float or Series.`*) – Solar azimuth angle.

Returns **aoi** (*Series*) – The angle of incidence

pvlib.tracking.SingleAxisTracker.get_irradiance

`SingleAxisTracker.`**`get_irradiance`**(*surface_tilt*, *surface_azimuth*, *solar_zenith*, *solar_azimuth*, *dni*, *ghi*, *dhi*, *dni_extra=None*, *airmass=None*, *model='haydavies'*, ***kwargs*)

Uses the `irradiance.get_total_irradiance()` function to calculate the plane of array irradiance components on a tilted surface defined by the input data and `self.albedo`.

For a given set of solar zenith and azimuth angles, the surface tilt and azimuth parameters are typically determined by *`singleaxis()`*.

Parameters

- **`surface_tilt`** (*`numeric`*) – Panel tilt from horizontal.
- **`surface_azimuth`** (*`numeric`*) – Panel azimuth from north
- **`solar_zenith`** (*`numeric`*) – Solar zenith angle.
- **`solar_azimuth`** (*`numeric`*) – Solar azimuth angle.
- **`dni`** (*`float or Series`*) – Direct Normal Irradiance
- **`ghi`** (*`float or Series`*) – Global horizontal irradiance
- **`dhi`** (*`float or Series`*) – Diffuse horizontal irradiance
- **`dni_extra`** (*`float or Series, default None`*) – Extraterrestrial direct normal irradiance
- **`airmass`** (*`float or Series, default None`*) – Airmass
- **`model`** (*`String, default 'haydavies'`*) – Irradiance model.
- **`**kwargs`** – Passed to `irradiance.get_total_irradiance()`.

Returns **poa_irradiance** (*DataFrame*) – Column names are: `total, beam, sky, ground`.

Decomposing and combining irradiance

`irradiance.get_extra_radiation`(datetime_or_doy)	Determine extraterrestrial radiation from day of year.
`irradiance.aoi`(surface_tilt, ...)	Calculates the angle of incidence of the solar vector on a surface.

Continued on next page

Table 18 – continued from previous page

irradiance.aoi_projection(surface_tilt, ...)	Calculates the dot product of the sun position unit vector and the surface normal unit vector; in other words, the cosine of the angle of incidence.
irradiance.poa_horizontal_ratio(...)	Calculates the ratio of the beam components of the plane of array irradiance and the horizontal irradiance.
irradiance.beam_component(surface_tilt, ...)	Calculates the beam component of the plane of array irradiance.
irradiance.poa_components(aoi, dni, ...)	Determine in-plane irradiance components.
irradiance.get_ground_diffuse(surface_tilt, ghi)	Estimate diffuse irradiance from ground reflections given irradiance, albedo, and surface tilt
irradiance.dni(ghi, dhi, zenith[, ...])	Determine DNI from GHI and DHI.

pvlib.irradiance.get_extra_radiation

pvlib.irradiance.**get_extra_radiation**(*datetime_or_doy*, *solar_constant=1366.1*, *method='spencer'*, *epoch_year=2014*, ***kwargs*)

Determine extraterrestrial radiation from day of year.

Parameters

- **datetime_or_doy** (*numeric, array, date, datetime, Timestamp, DatetimeIndex*) – Day of year, array of days of year, or datetime-like object
- **solar_constant** (*float, default 1366.1*) – The solar constant.
- **method** (*string, default 'spencer'*) – The method by which the ET radiation should be calculated. Options include `'pyephem'`, `'spencer'`, `'asce'`, `'nrel'`.
- **epoch_year** (*int, default 2014*) – The year in which a day of year input will be calculated. Only applies to day of year input used with the pyephem or nrel methods.
- **kwargs** – Passed to solarposition.nrel_earthsun_distance

Returns **dni_extra** (*float, array, or Series*) – The extraterrestrial radiation present in watts per square meter on a surface which is normal to the sun. Pandas Timestamp and DatetimeIndex inputs will yield a Pandas TimeSeries. All other inputs will yield a float or an array of floats.

References

pvlib.irradiance.aoi

pvlib.irradiance.**aoi**(*surface_tilt*, *surface_azimuth*, *solar_zenith*, *solar_azimuth*)

Calculates the angle of incidence of the solar vector on a surface. This is the angle between the solar vector and the surface normal.

Input all angles in degrees.

Parameters

- **surface_tilt** (*numeric*) – Panel tilt from horizontal.
- **surface_azimuth** (*numeric*) – Panel azimuth from north.
- **solar_zenith** (*numeric*) – Solar zenith angle.
- **solar_azimuth** (*numeric*) – Solar azimuth angle.

Returns **aoi** (*numeric*) – Angle of incidence in degrees.

pvlib.irradiance.aoi_projection

pvlib.irradiance.**aoi_projection**(*surface_tilt*, *surface_azimuth*, *solar_zenith*, *solar_azimuth*)

Calculates the dot product of the sun position unit vector and the surface normal unit vector; in other words, the cosine of the angle of incidence.

Usage note: When the sun is behind the surface the value returned is negative. For many uses negative values must be set to zero.

Input all angles in degrees.

Parameters

- **surface_tilt** (*numeric*) – Panel tilt from horizontal.
- **surface_azimuth** (*numeric*) – Panel azimuth from north.
- **solar_zenith** (*numeric*) – Solar zenith angle.
- **solar_azimuth** (*numeric*) – Solar azimuth angle.

Returns **projection** (*numeric*) – Dot product of panel normal and solar angle.

pvlib.irradiance.poa_horizontal_ratio

pvlib.irradiance.**poa_horizontal_ratio**(*surface_tilt*, *surface_azimuth*, *solar_zenith*, *solar_azimuth*)

Calculates the ratio of the beam components of the plane of array irradiance and the horizontal irradiance.

Input all angles in degrees.

Parameters

- **surface_tilt** (*numeric*) – Panel tilt from horizontal.
- **surface_azimuth** (*numeric*) – Panel azimuth from north.
- **solar_zenith** (*numeric*) – Solar zenith angle.
- **solar_azimuth** (*numeric*) – Solar azimuth angle.

Returns **ratio** (*numeric*) – Ratio of the plane of array irradiance to the horizontal plane irradiance

pvlib.irradiance.beam_component

pvlib.irradiance.**beam_component**(*surface_tilt*, *surface_azimuth*, *solar_zenith*, *solar_azimuth*, *dni*)

Calculates the beam component of the plane of array irradiance.

Parameters

- **surface_tilt** (*numeric*) – Panel tilt from horizontal.
- **surface_azimuth** (*numeric*) – Panel azimuth from north.
- **solar_zenith** (*numeric*) – Solar zenith angle.
- **solar_azimuth** (*numeric*) – Solar azimuth angle.
- **dni** (*numeric*) – Direct Normal Irradiance

Returns **beam** (*numeric*) – Beam component

pvlib.irradiance.poa_components

pvlib.irradiance.**poa_components**(*aoi*, *dni*, *poa_sky_diffuse*, *poa_ground_diffuse*)

Determine in-plane irradiance components.

Combines DNI with sky diffuse and ground-reflected irradiance to calculate total, direct and diffuse irradiance components in the plane of array.

Parameters

- **aoi** (*numeric*) – Angle of incidence of solar rays with respect to the module surface, from *aoi()*.
- **dni** (*numeric*) – Direct normal irradiance (W/m^2), as measured from a TMY file or calculated with a clearsky model.
- **poa_sky_diffuse** (*numeric*) – Diffuse irradiance (W/m^2) in the plane of the modules, as calculated by a diffuse irradiance translation function
- **poa_ground_diffuse** (*numeric*) – Ground reflected irradiance (W/m^2) in the plane of the modules, as calculated by an albedo model (eg. `grounddiffuse()`)

Returns

irrads (*OrderedDict or DataFrame*) – Contains the following keys:

- `poa_global` : Total in-plane irradiance (W/m^2)
- `poa_direct` : Total in-plane beam irradiance (W/m^2)
- `poa_diffuse` : Total in-plane diffuse irradiance (W/m^2)
- `poa_sky_diffuse` : In-plane diffuse irradiance from sky (W/m^2)
- `poa_ground_diffuse` : In-plane diffuse irradiance from ground (W/m^2)

Notes

Negative beam irradiation due to aoi $> 90°$ or AOI $< 0°$ is set to zero.

pvlib.irradiance.get_ground_diffuse

pvlib.irradiance.**get_ground_diffuse**(*surface_tilt*, *ghi*, *albedo=0.25*, *surface_type=None*)

Estimate diffuse irradiance from ground reflections given irradiance, albedo, and surface tilt

Function to determine the portion of irradiance on a tilted surface due to ground reflections. Any of the inputs may be DataFrames or scalars.

Parameters

- **surface_tilt** (*numeric*) – Surface tilt angles in decimal degrees. Tilt must be >=0 and <=180. The tilt angle is defined as degrees from horizontal (e.g. surface facing up = 0, surface facing horizon = 90).
- **ghi** (*numeric*) – Global horizontal irradiance in W/m^2.

- **albedo** (*numeric, default 0.25*) – Ground reflectance, typically 0.1-0.4 for surfaces on Earth (land), may increase over snow, ice, etc. May also be known as the reflection coefficient. Must be >=0 and <=1. Will be overridden if surface_type is supplied.
- **surface_type** (*None or string, default None*) – If not None, overrides albedo. String can be one of 'urban', 'grass', 'fresh grass', 'snow', 'fresh snow', 'asphalt', 'concrete', 'aluminum', 'copper', 'fresh steel', 'dirty steel', 'sea'.

Returns **grounddiffuse** (*numeric*) – Ground reflected irradiances in W/m^2.

References

The calculation is the last term of equations 3, 4, 7, 8, 10, 11, and 12.

pvlib.irradiance.dni

pvlib.irradiance.**dni**(*ghi*, *dhi*, *zenith*, *clearsky_dni=None*, *clearsky_tolerance=1.1*, *zenith_threshold_for_zero_dni=88.0*, *zenith_threshold_for_clearsky_limit=80.0*)

Determine DNI from GHI and DHI.

When calculating the DNI from GHI and DHI the calculated DNI may be unreasonably high or negative for zenith angles close to 90 degrees (sunrise/sunset transitions). This function identifies unreasonable DNI values and sets them to NaN. If the clearsky DNI is given unreasonably high values are cut off.

Parameters

- **ghi** (*Series*) – Global horizontal irradiance.
- **dhi** (*Series*) – Diffuse horizontal irradiance.
- **zenith** (*Series*) – True (not refraction-corrected) zenith angles in decimal degrees. Angles must be >=0 and <=180.
- **clearsky_dni** (*None or Series, default None*) – Clearsky direct normal irradiance.
- **clearsky_tolerance** (*float, default 1.1*) – If 'clearsky_dni' is given this parameter can be used to allow a tolerance by how much the calculated DNI value can be greater than the clearsky value before it is identified as an unreasonable value.
- **zenith_threshold_for_zero_dni** (*float, default 88.0*) – Non-zero DNI values for zenith angles greater than or equal to 'zenith_threshold_for_zero_dni' will be set to NaN.
- **zenith_threshold_for_clearsky_limit** (*float, default 80.0*) – DNI values for zenith angles greater than or equal to 'zenith_threshold_for_clearsky_limit' and smaller the 'zenith_threshold_for_zero_dni' that are greater than the clearsky DNI (times allowed tolerance) will be corrected. Only applies if 'clearsky_dni' is not None.

Returns **dni** (*Series*) – The modeled direct normal irradiance.

Transposition models

irradiance.get_total_irradiance(... [, ...])	Determine total in-plane irradiance and its beam, sky diffuse and ground reflected components, using the specified sky diffuse irradiance model.
irradiance.get_sky_diffuse(surface_tilt, ...)	Determine in-plane sky diffuse irradiance component using the specified sky diffuse irradiance model.
irradiance.isotropic(surface_tilt, dhi)	Determine diffuse irradiance from the sky on a tilted surface using the isotropic sky model.
irradiance.perez(surface_tilt, ... [, model, ...])	Determine diffuse irradiance from the sky on a tilted surface using one of the Perez models.
irradiance.haydavies(surface_tilt, ... [, ...])	Determine diffuse irradiance from the sky on a tilted surface using Hay & Davies' 1980 model
irradiance.klucher(surface_tilt, ...)	Determine diffuse irradiance from the sky on a tilted surface using Klucher's 1979 model
irradiance.reindl(surface_tilt, ...)	Determine diffuse irradiance from the sky on a tilted surface using Reindl's 1990 model
irradiance.king(surface_tilt, dhi, ghi, ...)	Determine diffuse irradiance from the sky on a tilted surface using the King model.

pvlib.irradiance.get_total_irradiance

pvlib.irradiance.**get_total_irradiance**(*surface_tilt*, *surface_azimuth*, *solar_zenith*, *solar_azimuth*, *dni*, *ghi*, *dhi*, *dni_extra=None*, *airmass=None*, *albedo=0.25*, *surface_type=None*, *model='isotropic'*, *model_perez='allsitescomposite1990'*, ***kwargs*)

Determine total in-plane irradiance and its beam, sky diffuse and ground reflected components, using the specified sky diffuse irradiance model.

$$I_{tot} = I_{beam} + I_{skydiffuse} + I_{ground}$$

Sky diffuse models include:

- isotropic (default)
- klucher
- haydavies
- reindl
- king
- perez

Parameters

- **surface_tilt** (*numeric*) – Panel tilt from horizontal.
- **surface_azimuth** (*numeric*) – Panel azimuth from north.
- **solar_zenith** (*numeric*) – Solar zenith angle.
- **solar_azimuth** (*numeric*) – Solar azimuth angle.
- **dni** (*numeric*) – Direct Normal Irradiance
- **ghi** (*numeric*) – Global horizontal irradiance
- **dhi** (*numeric*) – Diffuse horizontal irradiance

- **dni_extra** (*None or numeric, default None*) – Extraterrestrial direct normal irradiance
- **airmass** (*None or numeric, default None*) – Airmass
- **albedo** (*numeric, default 0.25*) – Surface albedo
- **surface_type** (*None or String, default None*) – Surface type. See grounddiffuse.
- **model** (*String, default 'isotropic'*) – Irradiance model.
- **model_perez** (*String, default 'allsitescomposite1990'*) – Used only if model='perez'. See *perez()*.

Returns **total_irrad** (*OrderedDict or DataFrame*) – Contains keys/columns `'poa_global'`, `'poa_direct'`, `'poa_diffuse'`, `'poa_sky_diffuse'`, `'poa_ground_diffuse'`.

pvlib.irradiance.get_sky_diffuse

pvlib.irradiance.**get_sky_diffuse**(*surface_tilt*, *surface_azimuth*, *solar_zenith*, *solar_azimuth*, *dni*, *ghi*, *dhi*, *dni_extra=None*, *airmass=None*, *model='isotropic'*, *model_perez='allsitescomposite1990'*)

Determine in-plane sky diffuse irradiance component using the specified sky diffuse irradiance model.

Sky diffuse models include:

- isotropic (default)
- klucher
- haydavies
- reindl
- king
- perez

Parameters

- **surface_tilt** (*numeric*) – Panel tilt from horizontal.
- **surface_azimuth** (*numeric*) – Panel azimuth from north.
- **solar_zenith** (*numeric*) – Solar zenith angle.
- **solar_azimuth** (*numeric*) – Solar azimuth angle.
- **dni** (*numeric*) – Direct Normal Irradiance
- **ghi** (*numeric*) – Global horizontal irradiance
- **dhi** (*numeric*) – Diffuse horizontal irradiance
- **dni_extra** (*None or numeric, default None*) – Extraterrestrial direct normal irradiance
- **airmass** (*None or numeric, default None*) – Airmass
- **model** (*String, default 'isotropic'*) – Irradiance model.
- **model_perez** (*String, default 'allsitescomposite1990'*) – See perez.

Returns **poa_sky_diffuse** (*numeric*)

pvlib.irradiance.isotropic

pvlib.irradiance.**isotropic**(*surface_tilt*, *dhi*)

Determine diffuse irradiance from the sky on a tilted surface using the isotropic sky model.

$$I_d = DHI \frac{1 + \cos\beta}{2}$$

Hottel and Woertz's model treats the sky as a uniform source of diffuse irradiance. Thus the diffuse irradiance from the sky (ground reflected irradiance is not included in this algorithm) on a tilted surface can be found from the diffuse horizontal irradiance and the tilt angle of the surface.

Parameters

- **surface_tilt** (*numeric*) – Surface tilt angle in decimal degrees. Tilt must be >=0 and <=180. The tilt angle is defined as degrees from horizontal (e.g. surface facing up = 0, surface facing horizon = 90)
- **dhi** (*numeric*) – Diffuse horizontal irradiance in W/m^2. DHI must be >=0.

Returns **diffuse** (*numeric*) – The sky diffuse component of the solar radiation.

References

pvlib.irradiance.perez

pvlib.irradiance.**perez**(*surface_tilt*, *surface_azimuth*, *dhi*, *dni*, *dni_extra*, *solar_zenith*, *solar_azimuth*, *airmass*, *model='allsitescomposite1990'*, *return_components=False*)

Determine diffuse irradiance from the sky on a tilted surface using one of the Perez models.

Perez models determine the diffuse irradiance from the sky (ground reflected irradiance is not included in this algorithm) on a tilted surface using the surface tilt angle, surface azimuth angle, diffuse horizontal irradiance, direct normal irradiance, extraterrestrial irradiance, sun zenith angle, sun azimuth angle, and relative (not pressure-corrected) airmass. Optionally a selector may be used to use any of Perez's model coefficient sets.

Parameters

- **surface_tilt** (*numeric*) – Surface tilt angles in decimal degrees. surface_tilt must be >=0 and <=180. The tilt angle is defined as degrees from horizontal (e.g. surface facing up = 0, surface facing horizon = 90)
- **surface_azimuth** (*numeric*) – Surface azimuth angles in decimal degrees. surface_azimuth must be >=0 and <=360. The azimuth convention is defined as degrees east of north (e.g. North = 0, South=180 East = 90, West = 270).
- **dhi** (*numeric*) – Diffuse horizontal irradiance in W/m^2. DHI must be >=0.
- **dni** (*numeric*) – Direct normal irradiance in W/m^2. DNI must be >=0.
- **dni_extra** (*numeric*) – Extraterrestrial normal irradiance in W/m^2.
- **solar_zenith** (*numeric*) – apparent (refraction-corrected) zenith angles in decimal degrees. solar_zenith must be >=0 and <=180.
- **solar_azimuth** (*numeric*) – Sun azimuth angles in decimal degrees. solar_azimuth must be >=0 and <=360. The azimuth convention is defined as degrees east of north (e.g. North = 0, East = 90, West = 270).

- **airmass** (*numeric*) – Relative (not pressure-corrected) airmass values. If AM is a DataFrame it must be of the same size as all other DataFrame inputs. AM must be >=0 (careful using the 1/sec(z) model of AM generation)
- **model** (*string (optional, default='allsitescomposite1990')*) – A string which selects the desired set of Perez coefficients. If model is not provided as an input, the default, '1990' will be used. All possible model selections are:
  - '1990'
  - 'allsitescomposite1990' (same as '1990')
  - 'allsitescomposite1988'
  - 'sandiacomposite1988'
  - 'usacomposite1988'
  - 'france1988'
  - 'phoenix1988'
  - 'elmonte1988'
  - 'osage1988'
  - 'albuquerque1988'
  - 'capecanaveral1988'
  - 'albany1988'
- **return_components** (*bool (optional, default=False)*) – Flag used to decide whether to return the calculated diffuse components or not.

Returns

- *numeric, OrderedDict, or DataFrame* – Return type controlled by *return_components* argument. If `return_components=False`, *sky_diffuse* is returned. If `return_components=True`, *diffuse_components* is returned.
- **sky_diffuse** (*numeric*) – The sky diffuse component of the solar radiation on a tilted surface.
- **diffuse_components** (*OrderedDict (array input) or DataFrame (Series input)*) –

  Keys/columns are:
  - sky_diffuse: Total sky diffuse
  - isotropic
  - circumsolar
  - horizon

References

pvlib.irradiance.haydavies

pvlib.irradiance.**haydavies**(*surface_tilt*, *surface_azimuth*, *dhi*, *dni*, *dni_extra*, *solar_zenith=None*, *solar_azimuth=None*, *projection_ratio=None*)

Determine diffuse irradiance from the sky on a tilted surface using Hay & Davies' 1980 model

$$I_d = DHI(AR_b + (1 - A)(\frac{1 + \cos\beta}{2}))$$

Hay and Davies' 1980 model determines the diffuse irradiance from the sky (ground reflected irradiance is not included in this algorithm) on a tilted surface using the surface tilt angle, surface azimuth angle, diffuse horizontal irradiance, direct normal irradiance, extraterrestrial irradiance, sun zenith angle, and sun azimuth angle.

Parameters

- **surface_tilt** (*numeric*) – Surface tilt angles in decimal degrees. The tilt angle is defined as degrees from horizontal (e.g. surface facing up = 0, surface facing horizon = 90)
- **surface_azimuth** (*numeric*) – Surface azimuth angles in decimal degrees. The azimuth convention is defined as degrees east of north (e.g. North=0, South=180, East=90, West=270).
- **dhi** (*numeric*) – Diffuse horizontal irradiance in W/m^2.
- **dni** (*numeric*) – Direct normal irradiance in W/m^2.
- **dni_extra** (*numeric*) – Extraterrestrial normal irradiance in W/m^2.
- **solar_zenith** (*None or numeric, default None*) – Solar apparent (refraction-corrected) zenith angles in decimal degrees. Must supply `solar_zenith` and `solar_azimuth` or supply `projection_ratio`.
- **solar_azimuth** (*None or numeric, default None*) – Solar azimuth angles in decimal degrees. Must supply `solar_zenith` and `solar_azimuth` or supply `projection_ratio`.
- **projection_ratio** (*None or numeric, default None*) – Ratio of angle of incidence projection to solar zenith angle projection. Must supply `solar_zenith` and `solar_azimuth` or supply `projection_ratio`.

Returns **sky_diffuse** (*numeric*) – The sky diffuse component of the solar radiation.

References

pvlib.irradiance.klucher

pvlib.irradiance.**klucher**(*surface_tilt*, *surface_azimuth*, *dhi*, *ghi*, *solar_zenith*, *solar_azimuth*)

Determine diffuse irradiance from the sky on a tilted surface using Klucher's 1979 model

$$I_d = DHI \frac{1 + \cos\beta}{2} (1 + F' \sin^3(\beta/2)) (1 + F' \cos^2\theta \sin^3\theta_z)$$

where

$$F' = 1 - (I_{d0}/GHI)^2$$

Klucher's 1979 model determines the diffuse irradiance from the sky (ground reflected irradiance is not included in this algorithm) on a tilted surface using the surface tilt angle, surface azimuth angle, diffuse horizontal irradiance, direct normal irradiance, global horizontal irradiance, extraterrestrial irradiance, sun zenith angle, and sun azimuth angle.

Parameters

- **surface_tilt** (*numeric*) – Surface tilt angles in decimal degrees. surface_tilt must be >=0 and <=180. The tilt angle is defined as degrees from horizontal (e.g. surface facing up = 0, surface facing horizon = 90)

- **surface_azimuth** (*numeric*) – Surface azimuth angles in decimal degrees. surface_azimuth must be >=0 and <=360. The Azimuth convention is defined as degrees east of north (e.g. North = 0, South=180 East = 90, West = 270).
- **dhi** (*numeric*) – Diffuse horizontal irradiance in W/m^2. DHI must be >=0.
- **ghi** (*numeric*) – Global irradiance in W/m^2. DNI must be >=0.
- **solar_zenith** (*numeric*) – Apparent (refraction-corrected) zenith angles in decimal degrees. solar_zenith must be >=0 and <=180.
- **solar_azimuth** (*numeric*) – Sun azimuth angles in decimal degrees. solar_azimuth must be >=0 and <=360. The Azimuth convention is defined as degrees east of north (e.g. North = 0, East = 90, West = 270).

Returns **diffuse** (*numeric*) – The sky diffuse component of the solar radiation.

References

pvlib.irradiance.reindl

pvlib.irradiance.**reindl**(*surface_tilt*, *surface_azimuth*, *dhi*, *dni*, *ghi*, *dni_extra*, *solar_zenith*, *solar_azimuth*)

Determine diffuse irradiance from the sky on a tilted surface using Reindl's 1990 model

$$I_d = DHI(AR_b + (1 - A)(\frac{1 + \cos\beta}{2})(1 + \sqrt{\frac{I_{hb}}{I_h}}\sin^3(\beta/2)))$$

Reindl's 1990 model determines the diffuse irradiance from the sky (ground reflected irradiance is not included in this algorithm) on a tilted surface using the surface tilt angle, surface azimuth angle, diffuse horizontal irradiance, direct normal irradiance, global horizontal irradiance, extraterrestrial irradiance, sun zenith angle, and sun azimuth angle.

Parameters

- **surface_tilt** (*numeric*) – Surface tilt angles in decimal degrees. The tilt angle is defined as degrees from horizontal (e.g. surface facing up = 0, surface facing horizon = 90)
- **surface_azimuth** (*numeric*) – Surface azimuth angles in decimal degrees. The azimuth convention is defined as degrees east of north (e.g. North = 0, South=180 East = 90, West = 270).
- **dhi** (*numeric*) – diffuse horizontal irradiance in W/m^2.
- **dni** (*numeric*) – direct normal irradiance in W/m^2.
- **ghi** (*numeric*) – Global irradiance in W/m^2.
- **dni_extra** (*numeric*) – Extraterrestrial normal irradiance in W/m^2.
- **solar_zenith** (*numeric*) – Apparent (refraction-corrected) zenith angles in decimal degrees.
- **solar_azimuth** (*numeric*) – Sun azimuth angles in decimal degrees. The azimuth convention is defined as degrees east of north (e.g. North = 0, East = 90, West = 270).

Returns **poa_sky_diffuse** (*numeric*) – The sky diffuse component of the solar radiation.

Notes

The poa_sky_diffuse calculation is generated from the Loutzenhiser et al. (2007) paper, equation 8. Note that I have removed the beam and ground reflectance portion of the equation and this generates ONLY the diffuse radiation from the sky and circumsolar, so the form of the equation varies slightly from equation 8.

References

pvlib.irradiance.king

pvlib.irradiance.**king**(*surface_tilt*, *dhi*, *ghi*, *solar_zenith*)

Determine diffuse irradiance from the sky on a tilted surface using the King model.

King's model determines the diffuse irradiance from the sky (ground reflected irradiance is not included in this algorithm) on a tilted surface using the surface tilt angle, diffuse horizontal irradiance, global horizontal irradiance, and sun zenith angle. Note that this model is not well documented and has not been published in any fashion (as of January 2012).

Parameters

- **surface_tilt** (*numeric*) – Surface tilt angles in decimal degrees. The tilt angle is defined as degrees from horizontal (e.g. surface facing up = 0, surface facing horizon = 90)
- **dhi** (*numeric*) – Diffuse horizontal irradiance in W/m^2.
- **ghi** (*numeric*) – Global horizontal irradiance in W/m^2.
- **solar_zenith** (*numeric*) – Apparent (refraction-corrected) zenith angles in decimal degrees.

Returns **poa_sky_diffuse** (*numeric*) – The diffuse component of the solar radiation.

DNI estimation models

irradiance.disc(ghi, solar_zenith, ... [, ...])	Estimate Direct Normal Irradiance from Global Horizontal Irradiance using the DISC model.
irradiance.dirint(ghi, solar_zenith, times)	Determine DNI from GHI using the DIRINT modification of the DISC model.
irradiance.dirindex(ghi, ghi_clearsky, ...)	Determine DNI from GHI using the DIRINDEX model.
irradiance.erbs(ghi, zenith, datetime_or_doy)	Estimate DNI and DHI from GHI using the Erbs model.
irradiance.liujordan(zenith, transmittance, ...)	Determine DNI, DHI, GHI from extraterrestrial flux, transmittance, and optical air mass number.
irradiance.gti_dirint(poa_global, aoi, ...)	Determine GHI, DNI, DHI from POA global using the GTI DIRINT model.

pvlib.irradiance.disc

pvlib.irradiance.**disc**(*ghi*, *solar_zenith*, *datetime_or_doy*, *pressure=101325*, *min_cos_zenith=0.065*, *max_zenith=87*, *max_airmass=12*)

Estimate Direct Normal Irradiance from Global Horizontal Irradiance using the DISC model.

The DISC algorithm converts global horizontal irradiance to direct normal irradiance through empirical relationships between the global and direct clearness indices.

The pvlib implementation limits the clearness index to 1.

The original report describing the DISC model[1] uses the relative airmass rather than the absolute (pressure-corrected) airmass. However, the NREL implementation of the DISC model[2] uses absolute airmass. PVLib Matlab also uses the absolute airmass. pvlib python defaults to absolute airmass, but the relative airmass can be used by supplying *pressure=None*.

Parameters

- **ghi** (*numeric*) – Global horizontal irradiance in W/m^2.
- **solar_zenith** (*numeric*) – True (not refraction-corrected) solar zenith angles in decimal degrees.
- **datetime_or_doy** (*int, float, array, pd.DatetimeIndex*) – Day of year or array of days of year e.g. pd.DatetimeIndex.dayofyear, or pd.DatetimeIndex.
- **pressure** (*None or numeric, default 101325*) – Site pressure in Pascal. If None, relative airmass is used instead of absolute (pressure-corrected) airmass.
- **min_cos_zenith** (*numeric, default 0.065*) – Minimum value of cos(zenith) to allow when calculating global clearness index *kt*. Equivalent to zenith = 86.273 degrees.
- **max_zenith** (*numeric, default 87*) – Maximum value of zenith to allow in DNI calculation. DNI will be set to 0 for times with zenith values greater than *max_zenith*.
- **max_airmass** (*numeric, default 12*) – Maximum value of the airmass to allow in Kn calculation. Default value (12) comes from range over which Kn was fit to airmass in the original paper.

Returns

output (*OrderedDict or DataFrame*) – Contains the following keys:

- `dni`: The modeled direct normal irradiance in W/m^2 provided by the Direct Insolation Simulation Code (DISC) model.
- `kt`: Ratio of global to extraterrestrial irradiance on a horizontal plane.
- `airmass`: Airmass

References

See also:

dirint()

pvlib.irradiance.dirint

pvlib.irradiance.**dirint**(*ghi*, *solar_zenith*, *times*, *pressure=101325.0*, *use_delta_kt_prime=True*, *temp_dew=None*, *min_cos_zenith=0.065*, *max_zenith=87*)

Determine DNI from GHI using the DIRINT modification of the DISC model.

Implements the modified DISC model known as "DIRINT" introduced in [1]. DIRINT predicts direct normal irradiance (DNI) from measured global horizontal irradiance (GHI). DIRINT improves upon the DISC model

[1] Maxwell, E. L., "A Quasi-Physical Model for Converting Hourly Global Horizontal to Direct Normal Insolation", Technical Report No. SERI/TR-215-3087, Golden, CO: Solar Energy Research Institute, 1987.

[2] Maxwell, E. "DISC Model", Excel Worksheet. https://www.nrel.gov/grid/solar-resource/disc.html

by using time-series GHI data and dew point temperature information. The effectiveness of the DIRINT model improves with each piece of information provided.

The pvlib implementation limits the clearness index to 1.

Parameters

- **ghi** (*array-like*) – Global horizontal irradiance in W/m^2.
- **solar_zenith** (*array-like*) – True (not refraction-corrected) solar_zenith angles in decimal degrees.
- **times** (*DatetimeIndex*) –
- **pressure** (*float or array-like, default 101325.0*) – The site pressure in Pascal. Pressure may be measured or an average pressure may be calculated from site altitude.
- **use_delta_kt_prime** (*bool, default True*) – If True, indicates that the stability index delta_kt_prime is included in the model. The stability index adjusts the estimated DNI in response to dynamics in the time series of GHI. It is recommended that delta_kt_prime is not used if the time between GHI points is 1.5 hours or greater. If use_delta_kt_prime=True, input data must be Series.
- **temp_dew** (*None, float, or array-like, default None*) – Surface dew point temperatures, in degrees C. Values of temp_dew may be numeric or NaN. Any single time period point with a temp_dew=NaN does not have dew point improvements applied. If temp_dew is not provided, then dew point improvements are not applied.
- **min_cos_zenith** (*numeric, default 0.065*) – Minimum value of cos(zenith) to allow when calculating global clearness index *kt*. Equivalent to zenith = 86.273 degrees.
- **max_zenith** (*numeric, default 87*) – Maximum value of zenith to allow in DNI calculation. DNI will be set to 0 for times with zenith values greater than *max_zenith*.

Returns **dni** (*array-like*) – The modeled direct normal irradiance in W/m^2 provided by the DIRINT model.

Notes

DIRINT model requires time series data (ie. one of the inputs must be a vector of length > 2).

References

pvlib.irradiance.dirindex

pvlib.irradiance.**dirindex**(*ghi*, *ghi_clearsky*, *dni_clearsky*, *zenith*, *times*, *pressure=101325.0*, *use_delta_kt_prime=True*, *temp_dew=None*, *min_cos_zenith=0.065*, *max_zenith=87*)

Determine DNI from GHI using the DIRINDEX model.

The DIRINDEX model [1] modifies the DIRINT model implemented in :py:func:`pvlib.irradiance.dirint` by taking into account information from a clear sky model. It is recommended that `ghi_clearsky` be calculated using the Ineichen clear sky model :py:func:`pvlib.clearsky.ineichen` with `perez_enhancement=True`.

The pvlib implementation limits the clearness index to 1.

Parameters

- **ghi** (*array-like*) – Global horizontal irradiance in W/m^2.
- **ghi_clearsky** (*array-like*) – Global horizontal irradiance from clear sky model, in W/m^2.
- **dni_clearsky** (*array-like*) – Direct normal irradiance from clear sky model, in W/m^2.
- **zenith** (*array-like*) – True (not refraction-corrected) zenith angles in decimal degrees. If Z is a vector it must be of the same size as all other vector inputs. Z must be >=0 and <=180.
- **times** (*DatetimeIndex*) –
- **pressure** (*float or array-like, default 101325.0*) – The site pressure in Pascal. Pressure may be measured or an average pressure may be calculated from site altitude.
- **use_delta_kt_prime** (*bool, default True*) – If True, indicates that the stability index delta_kt_prime is included in the model. The stability index adjusts the estimated DNI in response to dynamics in the time series of GHI. It is recommended that delta_kt_prime is not used if the time between GHI points is 1.5 hours or greater. If use_delta_kt_prime=True, input data must be Series.
- **temp_dew** (*None, float, or array-like, default None*) – Surface dew point temperatures, in degrees C. Values of temp_dew may be numeric or NaN. Any single time period point with a temp_dew=NaN does not have dew point improvements applied. If temp_dew is not provided, then dew point improvements are not applied.
- **min_cos_zenith** (*numeric, default 0.065*) – Minimum value of cos(zenith) to allow when calculating global clearness index *kt*. Equivalent to zenith = 86.273 degrees.
- **max_zenith** (*numeric, default 87*) – Maximum value of zenith to allow in DNI calculation. DNI will be set to 0 for times with zenith values greater than *max_zenith*.

Returns **dni** (*array-like*) – The modeled direct normal irradiance in W/m^2.

Notes

DIRINDEX model requires time series data (ie. one of the inputs must be a vector of length > 2).

References

pvlib.irradiance.erbs

pvlib.irradiance.**erbs**(*ghi*, *zenith*, *datetime_or_doy*, *min_cos_zenith=0.065*, *max_zenith=87*)

Estimate DNI and DHI from GHI using the Erbs model.

The Erbs model[1] estimates the diffuse fraction DF from global horizontal irradiance through an empirical relationship between DF and the ratio of GHI to extraterrestrial irradiance, Kt. The function uses the diffuse fraction

[1] D. G. Erbs, S. A. Klein and J. A. Duffie, Estimation of the diffuse radiation fraction for hourly, daily and monthly-average global radiation, Solar Energy 28(4), pp 293-302, 1982. Eq. 1

to compute DHI as

$$DHI = DF \times GHI$$

DNI is then estimated as

$$DNI = (GHI - DHI)/\cos(Z)$$

where Z is the zenith angle.

Parameters

- **ghi** (*numeric*) – Global horizontal irradiance in W/m^2.
- **zenith** (*numeric*) – True (not refraction-corrected) zenith angles in decimal degrees.
- **datetime_or_doy** (*int, float, array, pd.DatetimeIndex*) – Day of year or array of days of year e.g. pd.DatetimeIndex.dayofyear, or pd.DatetimeIndex.
- **min_cos_zenith** (*numeric, default 0.065*) – Minimum value of cos(zenith) to allow when calculating global clearness index *kt*. Equivalent to zenith = 86.273 degrees.
- **max_zenith** (*numeric, default 87*) – Maximum value of zenith to allow in DNI calculation. DNI will be set to 0 for times with zenith values greater than *max_zenith*.

Returns

data (*OrderedDict or DataFrame*) –

Contains the following keys/columns:

- `dni`: the modeled direct normal irradiance in W/m^2.
- `dhi`: the modeled diffuse horizontal irradiance in W/m^2.
- `kt`: Ratio of global to extraterrestrial irradiance on a horizontal plane.

References

See also:

dirint(), *disc()*

pvlib.irradiance.liujordan

`pvlib.irradiance.`**`liujordan`**(*zenith*, *transmittance*, *airmass*, *dni_extra=1367.0*)

Determine DNI, DHI, GHI from extraterrestrial flux, transmittance, and optical air mass number.

Liu and Jordan, 1960, developed a simplified direct radiation model. DHI is from an empirical equation for diffuse radiation from Liu and Jordan, 1960.

Parameters

- **zenith** (*pd.Series*) – True (not refraction-corrected) zenith angles in decimal degrees. If Z is a vector it must be of the same size as all other vector inputs. Z must be >=0 and <=180.
- **transmittance** (*float*) – Atmospheric transmittance between 0 and 1.
- **pressure** (*float, default 101325.0*) – Air pressure

- **dni_extra** (*float, default 1367.0*) – Direct irradiance incident at the top of the atmosphere.

Returns **irradiance** (*DataFrame*) – Modeled direct normal irradiance, direct horizontal irradiance, and global horizontal irradiance in W/m^2

References

pvlib.irradiance.gti_dirint

pvlib.irradiance.**gti_dirint**(*poa_global*, *aoi*, *solar_zenith*, *solar_azimuth*, *times*, *surface_tilt*, *surface_azimuth*, *pressure=101325.0*, *use_delta_kt_prime=True*, *temp_dew=None*, *albedo=0.25*, *model='perez'*, *model_perez='allsitescomposite1990'*, *calculate_gt_90=True*, *max_iterations=30*)

Determine GHI, DNI, DHI from POA global using the GTI DIRINT model.

The GTI DIRINT model is described in[1].

Warning: Model performance is poor for AOI greater than approximately 80 degrees *and* plane of array irradiance greater than approximately 200 W/m^2.

Parameters

- **poa_global** (*array-like*) – Plane of array global irradiance in W/m^2.
- **aoi** (*array-like*) – Angle of incidence of solar rays with respect to the module surface normal.
- **solar_zenith** (*array-like*) – True (not refraction-corrected) solar zenith angles in decimal degrees.
- **solar_azimuth** (*array-like*) – Solar azimuth angles in decimal degrees.
- **times** (*DatetimeIndex*) – Time indices for the input array-like data.
- **surface_tilt** (*numeric*) – Surface tilt angles in decimal degrees. Tilt must be >=0 and <=180. The tilt angle is defined as degrees from horizontal (e.g. surface facing up = 0, surface facing horizon = 90).
- **surface_azimuth** (*numeric*) – Surface azimuth angles in decimal degrees. surface_azimuth must be >=0 and <=360. The Azimuth convention is defined as degrees east of north (e.g. North = 0, South=180 East = 90, West = 270).
- **pressure** (*numeric, default 101325.0*) – The site pressure in Pascal. Pressure may be measured or an average pressure may be calculated from site altitude.
- **use_delta_kt_prime** (*bool, default True*) – If True, indicates that the stability index delta_kt_prime is included in the model. The stability index adjusts the estimated DNI in response to dynamics in the time series of GHI. It is recommended that delta_kt_prime is not used if the time between GHI points is 1.5 hours or greater. If use_delta_kt_prime=True, input data must be Series.
- **temp_dew** (*None, float, or array-like, default None*) – Surface dew point temperatures, in degrees C. Values of temp_dew may be numeric or NaN. Any

[1] B. Marion, A model for deriving the direct normal and diffuse horizontal irradiance from the global tilted irradiance, Solar Energy 122, 1037-1046. DOI: 10.1016/j.solener.2015.10.024

single time period point with a temp_dew=NaN does not have dew point improvements applied. If temp_dew is not provided, then dew point improvements are not applied.

- **albedo** (*numeric, default 0.25*) – Surface albedo
- **model** (*String, default 'isotropic'*) – Irradiance model.
- **model_perez** (*String, default 'allsitescomposite1990'*) – Used only if model='perez'. See *perez()*.
- **calculate_gt_90** (*bool, default True*) – Controls if the algorithm evaluates inputs with AOI >= 90 degrees. If False, returns nan for AOI >= 90 degrees. Significant speed ups can be achieved by setting this parameter to False.
- **max_iterations** (*int, default 30*) – Maximum number of iterations for the aoi < 90 deg algorithm.

Returns

data (*OrderedDict or DataFrame*) –

Contains the following keys/columns:

- `ghi`: the modeled global horizontal irradiance in W/m^2.
- `dni`: the modeled direct normal irradiance in W/m^2.
- `dhi`: the modeled diffuse horizontal irradiance in W/m^2.

References

Clearness index models

irradiance.clearness_index(ghi, ... [, ...])	Calculate the clearness index.
irradiance.clearness_index_zenith_independent(...)	Calculate the zenith angle independent clearness index.
irradiance.clearsky_index(ghi, clearsky_ghi)	Calculate the clearsky index.

pvlib.irradiance.clearness_index

pvlib.irradiance.**clearness_index**(*ghi*, *solar_zenith*, *extra_radiation*, *min_cos_zenith=0.065*, *max_clearness_index=2.0*)

Calculate the clearness index.

The clearness index is the ratio of global to extraterrestrial irradiance on a horizontal plane[1].

Parameters

- **ghi** (*numeric*) – Global horizontal irradiance in W/m^2.
- **solar_zenith** (*numeric*) – True (not refraction-corrected) solar zenith angle in decimal degrees.
- **extra_radiation** (*numeric*) – Irradiance incident at the top of the atmosphere
- **min_cos_zenith** (*numeric, default 0.065*) – Minimum value of cos(zenith) to allow when calculating global clearness index *kt*. Equivalent to zenith = 86.273 degrees.

[1] Maxwell, E. L., "A Quasi-Physical Model for Converting Hourly Global Horizontal to Direct Normal Insolation", Technical Report No. SERI/TR-215-3087, Golden, CO: Solar Energy Research Institute, 1987.

- **max_clearness_index** (*numeric, default 2.0*) – Maximum value of the clearness index. The default, 2.0, allows for over-irradiance events typically seen in sub-hourly data. NREL's SRRL Fortran code used 0.82 for hourly data.

Returns **kt** (*numeric*) – Clearness index

References

pvlib.irradiance.clearness_index_zenith_independent

pvlib.irradiance.**clearness_index_zenith_independent**(*clearness_index*, *airmass*, *max_clearness_index=2.0*)

Calculate the zenith angle independent clearness index.

See[1] for details.

Parameters

- **clearness_index** (*numeric*) – Ratio of global to extraterrestrial irradiance on a horizontal plane
- **airmass** (*numeric*) – Airmass
- **max_clearness_index** (*numeric, default 2.0*) – Maximum value of the clearness index. The default, 2.0, allows for over-irradiance events typically seen in sub-hourly data. NREL's SRRL Fortran code used 0.82 for hourly data.

Returns **kt_prime** (*numeric*) – Zenith independent clearness index

References

pvlib.irradiance.clearsky_index

pvlib.irradiance.**clearsky_index**(*ghi*, *clearsky_ghi*, *max_clearsky_index=2.0*)

Calculate the clearsky index.

The clearsky index is the ratio of global to clearsky global irradiance. Negative and non-finite clearsky index values will be truncated to zero.

Parameters

- **ghi** (*numeric*) – Global horizontal irradiance in W/m^2.
- **clearsky_ghi** (*numeric*) – Modeled clearsky GHI
- **max_clearsky_index** (*numeric, default 2.0*) – Maximum value of the clearsky index. The default, 2.0, allows for over-irradiance events typically seen in sub-hourly data.

Returns **clearsky_index** (*numeric*) – Clearsky index

3.12.6 PV Modeling

[1] Perez, R., P. Ineichen, E. Maxwell, R. Seals and A. Zelenka, (1992). "Dynamic Global-to-Direct Irradiance Conversion Models". ASHRAE Transactions-Research Series, pp. 354-369

Classes

The *PVSystem* class provides many methods that wrap the functions listed below. See its documentation for details.

pvsystem.PVSystem([surface_tilt, ...])	The PVSystem class defines a standard set of PV system attributes and modeling functions.
pvsystem.LocalizedPVSystem([pvsystem, location])	The LocalizedPVSystem class defines a standard set of installed PV system attributes and modeling functions.

Incident angle modifiers

iam.physical(aoi[, n, K, L])	Determine the incidence angle modifier using refractive index `n`, extinction coefficient `K`, and glazing thickness `L`.
iam.ashrae(aoi[, b])	Determine the incidence angle modifier using the ASHRAE transmission model.
iam.martin_ruiz(aoi[, a_r])	Determine the incidence angle modifier (IAM) using the Martin and Ruiz incident angle model.
iam.martin_ruiz_diffuse(surface_tilt[, a_r, ...])	Determine the incidence angle modifiers (iam) for diffuse sky and ground-reflected irradiance using the Martin and Ruiz incident angle model.
iam.sapm(aoi, module[, upper])	Determine the incidence angle modifier (IAM) using the SAPM model.
iam.interp(aoi, theta_ref, iam_ref[, ...])	Determine the incidence angle modifier (IAM) by interpolating a set of reference values, which are usually measured values.

pvlib.iam.physical

pvlib.iam.**physical**(*aoi*, *n=1.526*, *K=4.0*, *L=0.002*)

Determine the incidence angle modifier using refractive index `n`, extinction coefficient `K`, and glazing thickness `L`.

`iam.physical` calculates the incidence angle modifier as described in[1], Section 3. The calculation is based on a physical model of absorbtion and transmission through a transparent cover.

Parameters

- **aoi** (*numeric*) – The angle of incidence between the module normal vector and the sun-beam vector in degrees. Angles of 0 are replaced with 1e-06 to ensure non-nan results. Angles of nan will result in nan.
- **n** (*numeric, default 1.526*) – The effective index of refraction (unitless). Reference[1] indicates that a value of 1.526 is acceptable for glass.
- **K** (*numeric, default 4.0*) – The glazing extinction coefficient in units of 1/meters. Reference [1] indicates that a value of 4 is reasonable for "water white" glass.
- **L** (*numeric, default 0.002*) – The glazing thickness in units of meters. Reference[1] indicates that 0.002 meters (2 mm) is reasonable for most glass-covered PV panels.

Returns **iam** (*numeric*) – The incident angle modifier

[1] W. De Soto et al., "Improvement and validation of a model for photovoltaic array performance", Solar Energy, vol 80, pp. 78-88, 2006.

Notes

The pvlib python authors believe that Eqn. 14 in[1] is incorrect, which presents $\theta_r = \arcsin(n \sin(AOI))$. Here, $\theta_r = \arcsin(1/n \times \sin(AOI))$

References

See also:

pvlib.iam.martin_ruiz(), *pvlib.iam.ashrae()*, *pvlib.iam.interp()*, *pvlib.iam.sapm()*

pvlib.iam.ashrae

pvlib.iam.**ashrae**(*aoi*, *b=0.05*)

Determine the incidence angle modifier using the ASHRAE transmission model.

The ASHRAE (American Society of Heating, Refrigeration, and Air Conditioning Engineers) transmission model is developed in[1], and in[2]. The model has been used in software such as PVSyst[3].

Parameters

- **aoi** (*numeric*) – The angle of incidence (AOI) between the module normal vector and the sun-beam vector in degrees. Angles of nan will result in nan.
- **b** (*float, default 0.05*) – A parameter to adjust the incidence angle modifier as a function of angle of incidence. Typical values are on the order of 0.05 [3].

Returns **iam** (*numeric*) – The incident angle modifier (IAM). Returns zero for all abs(aoi) >= 90 and for all `iam` values that would be less than 0.

Notes

The incidence angle modifier is calculated as

$$IAM = 1 - b(\sec(aoi) - 1)$$

As AOI approaches 90 degrees, the model yields negative values for IAM; negative IAM values are set to zero in this implementation.

References

See also:

pvlib.iam.physical(), *pvlib.iam.martin_ruiz()*, *pvlib.iam.interp()*

[1] Souka A.F., Safwat H.H., "Determination of the optimum orientations for the double exposure flat-plate collector and its reflections". Solar Energy vol .10, pp 170-174. 1966.

[2] ASHRAE standard 93-77

[3] PVsyst Contextual Help. https://files.pvsyst.com/help/index.html?iam_loss.htm retrieved on October 14, 2019

pvlib.iam.martin_ruiz

pvlib.iam.**martin_ruiz**(*aoi*, *a_r=0.16*)

Determine the incidence angle modifier (IAM) using the Martin and Ruiz incident angle model.

Parameters

- **aoi** (*numeric, degrees*) – The angle of incidence between the module normal vector and the sun-beam vector in degrees.
- **a_r** (*numeric*) – The angular losses coefficient described in equation 3 of[1]. This is an empirical dimensionless parameter. Values of a_r are generally on the order of 0.08 to 0.25 for flat-plate PV modules.

Returns **iam** (*numeric*) – The incident angle modifier(s)

Notes

martin_ruiz calculates the incidence angle modifier (IAM) as described in[1]. The information required is the incident angle (AOI) and the angular losses coefficient (a_r). Note that[1] has a corrigendum[2] which clarifies a mix-up of 'alpha's and 'a's in the former.

The incident angle modifier is defined as

$$IAM = \frac{1 - \exp(-\cos(\frac{aoi}{a_r}))}{1 - \exp(\frac{-1}{a_r})}$$

which is presented as $AL(\alpha) = 1 - IAM$ in equation 4 of[1], with α representing the angle of incidence AOI. Thus IAM = 1 at AOI = 0, and IAM = 0 at AOI = 90. This equation is only valid for -90 <= aoi <= 90, therefore *iam* is constrained to 0.0 outside this interval.

References

See also:

pvlib.iam.martin_ruiz_diffuse(), *pvlib.iam.physical()*, *pvlib.iam.ashrae()*, *pvlib.iam.interp()*, *pvlib.iam.sapm()*

pvlib.iam.martin_ruiz_diffuse

pvlib.iam.**martin_ruiz_diffuse**(*surface_tilt*, *a_r=0.16*, *c1=0.4244*, *c2=None*)

Determine the incidence angle modifiers (iam) for diffuse sky and ground-reflected irradiance using the Martin and Ruiz incident angle model.

Parameters

- **surface_tilt** (*float or array-like, default 0*) – Surface tilt angles in decimal degrees. The tilt angle is defined as degrees from horizontal (e.g. surface facing up = 0, surface facing horizon = 90) surface_tilt must be in the range [0, 180]

[1] N. Martin and J. M. Ruiz, "Calculation of the PV modules angular losses under field conditions by means of an analytical model", Solar Energy Materials & Solar Cells, vol. 70, pp. 25-38, 2001.

[2] N. Martin and J. M. Ruiz, "Corrigendum to 'Calculation of the PV modules angular losses under field conditions by means of an analytical model'", Solar Energy Materials & Solar Cells, vol. 110, pp. 154, 2013.

- **a_r** (*numeric*) – The angular losses coefficient described in equation 3 of[1]. This is an empirical dimensionless parameter. Values of a_r are generally on the order of 0.08 to 0.25 for flat-plate PV modules. a_r must be greater than zero.
- **c1** (*float*) – First fitting parameter for the expressions that approximate the integral of diffuse irradiance coming from different directions. c1 is given as the constant 4 / 3 / pi (0.4244) in[1].
- **c2** (*float*) – Second fitting parameter for the expressions that approximate the integral of diffuse irradiance coming from different directions. If c2 is None, it will be calculated according to the linear relationship given in[3].

Returns

- **iam_sky** (*numeric*) – The incident angle modifier for sky diffuse
- **iam_ground** (*numeric*) – The incident angle modifier for ground-reflected diffuse

Notes

Sky and ground modifiers are complementary: iam_sky for tilt = 30 is equal to iam_ground for tilt = 180 - 30. For vertical surfaces, tilt = 90, the two factors are equal.

References

See also:

pvlib.iam.martin_ruiz(), *pvlib.iam.physical()*, *pvlib.iam.ashrae()*, *pvlib.iam.interp()*, *pvlib.iam.sapm()*

pvlib.iam.sapm

pvlib.iam.**sapm**(*aoi*, *module*, *upper=None*)

Determine the incidence angle modifier (IAM) using the SAPM model.

Parameters

- **aoi** (*numeric*) – Angle of incidence in degrees. Negative input angles will return zeros.
- **module** (*dict-like*) – A dict or Series with the SAPM IAM model parameters. See the *sapm()* notes section for more details.
- **upper** (*None or float, default None*) – Upper limit on the results.

Returns **iam** (*numeric*) – The SAPM angle of incidence loss coefficient, termed F2 in[1].

Notes

The SAPM[1] traditionally does not define an upper limit on the AOI loss function and values slightly exceeding 1 may exist for moderate angles of incidence (15-40 degrees). However, users may consider imposing an upper limit of 1.

[1] N. Martin and J. M. Ruiz, "Calculation of the PV modules angular losses under field conditions by means of an analytical model", Solar Energy Materials & Solar Cells, vol. 70, pp. 25-38, 2001.

[3] "IEC 61853-3 Photovoltaic (PV) module performance testing and energy rating - Part 3: Energy rating of PV modules". IEC, Geneva, 2018.

[1] King, D. et al, 2004, "Sandia Photovoltaic Array Performance Model", SAND Report 3535, Sandia National Laboratories, Albuquerque, NM.

References

See also:

pvlib.iam.physical(), *pvlib.iam.ashrae()*, *pvlib.iam.martin_ruiz()*, *pvlib.iam.interp()*

pvlib.iam.interp

pvlib.iam.**interp**(*aoi*, *theta_ref*, *iam_ref*, *method='linear'*, *normalize=True*)

Determine the incidence angle modifier (IAM) by interpolating a set of reference values, which are usually measured values.

Parameters

- **aoi** (*numeric*) – The angle of incidence between the module normal vector and the sun-beam vector [degrees].
- **theta_ref** (*numeric*) – Vector of angles at which the IAM is known [degrees].
- **iam_ref** (*numeric*) – IAM values for each angle in `theta_ref` [unitless].
- **method** (*str, default 'linear'*) – Specifies the interpolation method. Useful options are: 'linear', 'quadratic', 'cubic'. See scipy.interpolate.interp1d for more options.
- **normalize** (*boolean, default True*) – When true, the interpolated values are divided by the interpolated value at zero degrees. This ensures that `iam=1.0` at normal incidence.

Returns **iam** (*numeric*) – The incident angle modifier(s) [unitless]

Notes

`theta_ref` must have two or more points and may span any range of angles. Typically there will be a dozen or more points in the range 0-90 degrees. Beyond the range of `theta_ref`, IAM values are extrapolated, but constrained to be non-negative.

The sign of `aoi` is ignored; only the magnitude is used.

See also:

pvlib.iam.physical(), *pvlib.iam.ashrae()*, *pvlib.iam.martin_ruiz()*, *pvlib.iam.sapm()*

PV temperature models

temperature.sapm_cell(poa_global, temp_air, ...)	Calculate cell temperature per the Sandia Array Performance Model.
temperature.sapm_module(poa_global, ...)	Calculate module back surface temperature per the Sandia Array Performance Model.
temperature.sapm_cell_from_module(..., [, ...])	Calculate cell temperature from module temperature using the Sandia Array Performance Model.
temperature.pvsyst_cell(poa_global, temp_air)	Calculate cell temperature using an empirical heat loss factor model as implemented in PVsyst.

Continued on next page

Table 24 – continued from previous page

temperature.faiman(poa_global, ...])	temp_air[,	Calculate cell or module temperature using the Faiman model.

pvlib.temperature.sapm_cell

pvlib.temperature.**sapm_cell**(*poa_global*, *temp_air*, *wind_speed*, *a*, *b*, *deltaT*, *irrad_ref=1000*)

Calculate cell temperature per the Sandia Array Performance Model.

See[1] for details on the Sandia Array Performance Model.

Parameters

- **poa_global** (*numeric*) – Total incident irradiance [W/m^2].
- **temp_air** (*numeric*) – Ambient dry bulb temperature [C].
- **wind_speed** (*numeric*) – Wind speed at a height of 10 meters [m/s].
- **a** (*float*) – Parameter a in (3.1).
- **b** (*float*) – Parameter b in (3.1).
- **deltaT** (*float*) – Parameter ΔT in (3.2) [C].
- **irrad_ref** (*float, default 1000*) – Reference irradiance, parameter E_0 in (3.2) [W/m^2].

Returns *numeric, values in degrees C.*

Notes

The model for cell temperature T_C is given by a pair of equations (Eq. 11 and 12 in[1]).

$$T_m = E \times \exp(a + b \times WS) + T_a \tag{3.1}$$

$$T_C = T_m + \frac{E}{E_0} \Delta T \tag{3.2}$$

The module back surface temperature T_m is implemented in *sapm_module()*.

Inputs to the model are plane-of-array irradiance E (W/m2) and ambient air temperature T_a (C). Model parameters depend both on the module construction and its mounting. Parameter sets are provided in[1] for representative modules and mounting, and are coded for convenience in pvlib.temperature.TEMPERATURE_MODEL_PARAMETERS.

Module	Mounting	a	b	$\Delta T[C]$
glass/glass	open rack	-3.47	-0.0594	3
glass/glass	close roof	-2.98	-0.0471	1
glass/polymer	open rack	-3.56	-0.075	3
glass/polymer	insulated back	-2.81	-0.0455	0

References

See also:

sapm_cell_from_module(), *sapm_module()*

[1] King, D. et al, 2004, "Sandia Photovoltaic Array Performance Model", SAND Report 3535, Sandia National Laboratories, Albuquerque, NM.

Examples

```
>>> from pvlib.temperature import sapm_cell, TEMPERATURE_MODEL_PARAMETERS
>>> params = TEMPERATURE_MODEL_PARAMETERS['sapm']['open_rack_glass_glass']
>>> sapm_cell(1000, 10, 0, **params)
44.11703066106086
```

pvlib.temperature.sapm_module

pvlib.temperature.**sapm_module**(*poa_global*, *temp_air*, *wind_speed*, *a*, *b*)

Calculate module back surface temperature per the Sandia Array Performance Model.

See[1] for details on the Sandia Array Performance Model.

Parameters

- **poa_global** (*numeric*) – Total incident irradiance [W/m^2].
- **temp_air** (*numeric*) – Ambient dry bulb temperature [C].
- **wind_speed** (*numeric*) – Wind speed at a height of 10 meters [m/s].
- **a** (*float*) – Parameter a in (3.3).
- **b** (*float*) – Parameter b in (3.3).

Returns *numeric, values in degrees C.*

Notes

The model for module temperature T_m is given by Eq. 11 in[1].

$$T_m = E \times \exp(a + b \times WS) + T_a \tag{3.3}$$

Inputs to the model are plane-of-array irradiance E (W/m2) and ambient air temperature T_a (C). Model outputs are surface temperature at the back of the module T_m and cell temperature T_C. Model parameters depend both on the module construction and its mounting. Parameter sets are provided in[1] for representative modules and mounting, and are coded for convenience in `temperature.TEMPERATURE_MODEL_PARAMETERS`.

Module	Mounting	a	b	$\Delta T[C]$
glass/glass	open rack	-3.47	-0.0594	3
glass/glass	close roof	-2.98	-0.0471	1
glass/polymer	open rack	-3.56	-0.075	3
glass/polymer	insulated back	-2.81	-0.0455	0

References

See also:

sapm_cell(), *sapm_cell_from_module()*

[1] King, D. et al, 2004, "Sandia Photovoltaic Array Performance Model", SAND Report 3535, Sandia National Laboratories, Albuquerque, NM.

pvlib.temperature.sapm_cell_from_module

pvlib.temperature.**sapm_cell_from_module**(*module_temperature*, *poa_global*, *deltaT*, *irrad_ref=1000*)

Calculate cell temperature from module temperature using the Sandia Array Performance Model.

See[1] for details on the Sandia Array Performance Model.

Parameters

- **module_temperature** (*numeric*) – Temperature of back of module surface [C].
- **poa_global** (*numeric*) – Total incident irradiance [W/m^2].
- **deltaT** (*float*) – Parameter ΔT in (3.2) [C].
- **irrad_ref** (*float, default 1000*) – Reference irradiance, parameter E_0 in (3.2) [W/m^2].

Returns *numeric, values in degrees C.*

Notes

The model for cell temperature T_C is given by Eq. 12 in[1].

$$T_C = T_m + \frac{E}{E_0}\Delta T \tag{3.4}$$

The module back surface temperature T_m is implemented in *sapm_module()*.

Model parameters depend both on the module construction and its mounting. Parameter sets are provided in[1] for representative modules and mounting, and are coded for convenience in pvlib.temperature.TEMPERATURE_MODEL_PARAMETERS.

Module	Mounting	a	b	$\Delta T[C]$
glass/glass	open rack	-3.47	-0.0594	3
glass/glass	close roof	-2.98	-0.0471	1
glass/polymer	open rack	-3.56	-0.075	3
glass/polymer	insulated back	-2.81	-0.0455	0

References

See also:

sapm_cell(), *sapm_module()*

pvlib.temperature.pvsyst_cell

pvlib.temperature.**pvsyst_cell**(*poa_global*, *temp_air*, *wind_speed=1.0*, *u_c=29.0*, *u_v=0.0*, *eta_m=0.1*, *alpha_absorption=0.9*)

Calculate cell temperature using an empirical heat loss factor model as implemented in PVsyst.

Parameters

- **poa_global** (*numeric*) – Total incident irradiance [W/m^2].

[1] King, D. et al, 2004, "Sandia Photovoltaic Array Performance Model", SAND Report 3535, Sandia National Laboratories, Albuquerque, NM.

- **temp_air** (*numeric*) – Ambient dry bulb temperature [C].
- **wind_speed** (*numeric, default 1.0*) – Wind speed in m/s measured at the same height for which the wind loss factor was determined. The default value 1.0 m/2 is the wind speed at module height used to determine NOCT. [m/s]
- **u_c** (*float, default 29.0*) – Combined heat loss factor coefficient. The default value is representative of freestanding modules with the rear surfaces exposed to open air (e.g., rack mounted). Parameter U_c in (3.5). $\left[\frac{\mathrm{W/m^2}}{\mathrm{C}}\right]$
- **u_v** (*float, default 0.0*) – Combined heat loss factor influenced by wind. Parameter U_v in (3.5). $\left[\frac{\mathrm{W/m^2}}{\mathrm{C\ (m/s)}}\right]$
- **eta_m** (*numeric, default 0.1*) – Module external efficiency as a fraction, i.e., DC power / poa_global. Parameter η_m in (3.5).
- **alpha_absorption** (*numeric, default 0.9*) – Absorption coefficient. Parameter α in (3.5).

Returns *numeric, values in degrees Celsius*

Notes

The Pvsyst model for cell temperature T_C is given by

$$T_C = T_a + \frac{\alpha E (1 - \eta_m)}{U_c + U_v \times WS} \tag{3.5}$$

Inputs to the model are plane-of-array irradiance E (W/m2), ambient air temperature T_a (C) and wind speed WS (m/s). Model output is cell temperature T_C. Model parameters depend both on the module construction and its mounting. Parameters are provided in[1] for open (freestanding) and close (insulated) mounting configurations, , and are coded for convenience in `temperature.TEMPERATURE_MODEL_PARAMETERS`. The heat loss factors provided represent the combined effect of convection, radiation and conduction, and their values are experimentally determined.

Mounting	U_c	U_v
freestanding	29.0	0.0
insulated	15.0	0.0

References

Examples

```
>>> from pvlib.temperature import pvsyst_cell, TEMPERATURE_MODEL_PARAMETERS
>>> params = TEMPERATURE_MODEL_PARAMETERS['pvsyst']['freestanding']
>>> pvsyst_cell(1000, 10, **params)
37.93103448275862
```

pvlib.temperature.faiman

pvlib.temperature.**faiman**(*poa_global*, *temp_air*, *wind_speed=1.0*, *u0=25.0*, *u1=6.84*)

Calculate cell or module temperature using the Faiman model. The Faiman model uses an empirical heat loss

[1] "PVsyst 6 Help", Files.pvsyst.com, 2018. [Online]. Available: http://files.pvsyst.com/help/index.html. [Accessed: 10- Dec- 2018].

factor model[1] and is adopted in the IEC 61853 standards[2] and[3].

Usage of this model in the IEC 61853 standard does not distinguish between cell and module temperature.

Parameters

- **poa_global** (*numeric*) – Total incident irradiance [W/m^2].
- **temp_air** (*numeric*) – Ambient dry bulb temperature [C].
- **wind_speed** (*numeric, default 1.0*) – Wind speed in m/s measured at the same height for which the wind loss factor was determined. The default value 1.0 m/s is the wind speed at module height used to determine NOCT. [m/s]
- **u0** (*numeric, default 25.0*) – Combined heat loss factor coefficient. The default value is one determined by Faiman for 7 silicon modules. $\left[\frac{\mathrm{W/m^2}}{\mathrm{C}}\right]$
- **u1** (*numeric, default 6.84*) – Combined heat loss factor influenced by wind. The default value is one determined by Faiman for 7 silicon modules. $\left[\frac{\mathrm{W/m^2}}{\mathrm{C\ (m/s)}}\right]$

Returns *numeric, values in degrees Celsius*

Notes

All arguments may be scalars or vectors. If multiple arguments are vectors they must be the same length.

References

Single diode models

Functions relevant for single diode models.

pvsystem.calcparams_cec(... [, EgRef, dEgdT, ...])	Calculates five parameter values for the single diode equation at effective irradiance and cell temperature using the CEC model.
pvsystem.calcparams_desoto(... [, EgRef, ...])	Calculates five parameter values for the single diode equation at effective irradiance and cell temperature using the De Soto et al.
pvsystem.calcparams_pvsyst(... [, R_sh_exp, ...])	Calculates five parameter values for the single diode equation at effective irradiance and cell temperature using the PVsyst v6 model.
pvsystem.i_from_v(resistance_shunt, ... [, ...])	Device current at the given device voltage for the single diode model.
pvsystem.singlediode(photocurrent, ... [, ...])	Solve the single-diode equation to obtain a photovoltaic IV curve.
pvsystem.v_from_i(resistance_shunt, ... [, ...])	Device voltage at the given device current for the single diode model.
pvsystem.max_power_point(photocurrent, ...)	Given the single diode equation coefficients, calculates the maximum power point (MPP).

[1] Faiman, D. (2008). "Assessing the outdoor operating temperature of photovoltaic modules." Progress in Photovoltaics 16(4): 307-315.

[2] "IEC 61853-2 Photovoltaic (PV) module performance testing and energy rating - Part 2: Spectral responsivity, incidence angle and module operating temperature measurements". IEC, Geneva, 2018.

[3] "IEC 61853-3 Photovoltaic (PV) module performance testing and energy rating - Part 3: Energy rating of PV modules". IEC, Geneva, 2018.

pvlib.pvsystem.calcparams_cec

pvlib.pvsystem.**calcparams_cec**(*effective_irradiance*, *temp_cell*, *alpha_sc*, *a_ref*, *I_L_ref*, *I_o_ref*, *R_sh_ref*, *R_s*, *Adjust*, *EgRef=1.121*, *dEgdT=-0.0002677*, *irrad_ref=1000*, *temp_ref=25*)

Calculates five parameter values for the single diode equation at effective irradiance and cell temperature using the CEC model. The CEC model[1] differs from the De soto et al. model[3] by the parameter Adjust. The five values returned by calcparams_cec can be used by singlediode to calculate an IV curve.

Parameters

- **effective_irradiance** (*numeric*) – The irradiance (W/m2) that is converted to photocurrent.
- **temp_cell** (*numeric*) – The average cell temperature of cells within a module in C.
- **alpha_sc** (*float*) – The short-circuit current temperature coefficient of the module in units of A/C.
- **a_ref** (*float*) – The product of the usual diode ideality factor (n, unitless), number of cells in series (Ns), and cell thermal voltage at reference conditions, in units of V.
- **I_L_ref** (*float*) – The light-generated current (or photocurrent) at reference conditions, in amperes.
- **I_o_ref** (*float*) – The dark or diode reverse saturation current at reference conditions, in amperes.
- **R_sh_ref** (*float*) – The shunt resistance at reference conditions, in ohms.
- **R_s** (*float*) – The series resistance at reference conditions, in ohms.
- **Adjust** (*float*) – The adjustment to the temperature coefficient for short circuit current, in percent
- **EgRef** (*float*) – The energy bandgap at reference temperature in units of eV. 1.121 eV for crystalline silicon. EgRef must be >0. For parameters from the SAM CEC module database, EgRef=1.121 is implicit for all cell types in the parameter estimation algorithm used by NREL.
- **dEgdT** (*float*) – The temperature dependence of the energy bandgap at reference conditions in units of 1/K. May be either a scalar value (e.g. -0.0002677 as in [3]) or a DataFrame (this may be useful if dEgdT is a modeled as a function of temperature). For parameters from the SAM CEC module database, dEgdT=-0.0002677 is implicit for all cell types in the parameter estimation algorithm used by NREL.
- **irrad_ref** (*float (optional, default=1000)*) – Reference irradiance in W/m^2.
- **temp_ref** (*float (optional, default=25)*) – Reference cell temperature in C.

Returns

- *Tuple of the following results*
- **photocurrent** (*numeric*) – Light-generated current in amperes
- **saturation_current** (*numeric*) – Diode saturation curent in amperes

[1] A. Dobos, "An Improved Coefficient Calculator for the California Energy Commission 6 Parameter Photovoltaic Module Model", Journal of Solar Energy Engineering, vol 134, 2012.

[3] W. De Soto et al., "Improvement and validation of a model for photovoltaic array performance", Solar Energy, vol 80, pp. 78-88, 2006.

- **resistance_series** (*float*) – Series resistance in ohms
- **resistance_shunt** (*numeric*) – Shunt resistance in ohms
- **nNsVth** (*numeric*) – The product of the usual diode ideality factor (n, unitless), number of cells in series (Ns), and cell thermal voltage at specified effective irradiance and cell temperature.

References

See also:

calcparams_desoto(), *singlediode()*, *retrieve_sam()*

pvlib.pvsystem.calcparams_desoto

pvlib.pvsystem.**calcparams_desoto**(*effective_irradiance*, *temp_cell*, *alpha_sc*, *a_ref*, *I_L_ref*, *I_o_ref*, *R_sh_ref*, *R_s*, *EgRef=1.121*, *dEgdT=-0.0002677*, *irrad_ref=1000*, *temp_ref=25*)

Calculates five parameter values for the single diode equation at effective irradiance and cell temperature using the De Soto et al. model described in[1]. The five values returned by calcparams_desoto can be used by singlediode to calculate an IV curve.

Parameters

- **effective_irradiance** (*numeric*) – The irradiance (W/m2) that is converted to photocurrent.
- **temp_cell** (*numeric*) – The average cell temperature of cells within a module in C.
- **alpha_sc** (*float*) – The short-circuit current temperature coefficient of the module in units of A/C.
- **a_ref** (*float*) – The product of the usual diode ideality factor (n, unitless), number of cells in series (Ns), and cell thermal voltage at reference conditions, in units of V.
- **I_L_ref** (*float*) – The light-generated current (or photocurrent) at reference conditions, in amperes.
- **I_o_ref** (*float*) – The dark or diode reverse saturation current at reference conditions, in amperes.
- **R_sh_ref** (*float*) – The shunt resistance at reference conditions, in ohms.
- **R_s** (*float*) – The series resistance at reference conditions, in ohms.
- **EgRef** (*float*) – The energy bandgap at reference temperature in units of eV. 1.121 eV for crystalline silicon. EgRef must be >0. For parameters from the SAM CEC module database, EgRef=1.121 is implicit for all cell types in the parameter estimation algorithm used by NREL.
- **dEgdT** (*float*) – The temperature dependence of the energy bandgap at reference conditions in units of 1/K. May be either a scalar value (e.g. -0.0002677 as in[1]) or a DataFrame (this may be useful if dEgdT is a modeled as a function of temperature). For parameters from the SAM CEC module database, dEgdT=-0.0002677 is implicit for all cell types in the parameter estimation algorithm used by NREL.

[1] W. De Soto et al., "Improvement and validation of a model for photovoltaic array performance", Solar Energy, vol 80, pp. 78-88, 2006.

- **irrad_ref** (*float (optional, default=1000)*) – Reference irradiance in W/m^2.
- **temp_ref** (*float (optional, default=25)*) – Reference cell temperature in C.

Returns

- *Tuple of the following results*
- **photocurrent** (*numeric*) – Light-generated current in amperes
- **saturation_current** (*numeric*) – Diode saturation curent in amperes
- **resistance_series** (*float*) – Series resistance in ohms
- **resistance_shunt** (*numeric*) – Shunt resistance in ohms
- **nNsVth** (*numeric*) – The product of the usual diode ideality factor (n, unitless), number of cells in series (Ns), and cell thermal voltage at specified effective irradiance and cell temperature.

References

See also:

singlediode(), *retrieve_sam()*

Notes

If the reference parameters in the ModuleParameters struct are read from a database or library of parameters (e.g. System Advisor Model), it is important to use the same EgRef and dEgdT values that were used to generate the reference parameters, regardless of the actual bandgap characteristics of the semiconductor. For example, in the case of the System Advisor Model library, created as described in [3], EgRef and dEgdT for all modules were 1.121 and -0.0002677, respectively.

This table of reference bandgap energies (EgRef), bandgap energy temperature dependence (dEgdT), and "typical" airmass response (M) is provided purely as reference to those who may generate their own reference module parameters (a_ref, IL_ref, I0_ref, etc.) based upon the various PV semiconductors. Again, we stress the importance of using identical EgRef and dEgdT when generation reference parameters and modifying the reference parameters (for irradiance, temperature, and airmass) per DeSoto's equations.

Crystalline Silicon (Si):

- EgRef = 1.121
- dEgdT = -0.0002677

```
>>> M = np.polyval([-1.26E-4, 2.816E-3, -0.024459, 0.086257, 0.9181],
...                AMa) # doctest: +SKIP
```

Source: [1]

Cadmium Telluride (CdTe):

- EgRef = 1.475
- dEgdT = -0.0003

```
>>> M = np.polyval([-2.46E-5, 9.607E-4, -0.0134, 0.0716, 0.9196],
...                AMa) # doctest: +SKIP
```

Source: [4]

Copper Indium diSelenide (CIS):

- EgRef = 1.010
- dEgdT = -0.00011

```
>>> M = np.polyval([-3.74E-5, 0.00125, -0.01462, 0.0718, 0.9210],
...                AMa) # doctest: +SKIP
```

Source: [4]

Copper Indium Gallium diSelenide (CIGS):

- EgRef = 1.15
- dEgdT = ????

```
>>> M = np.polyval([-9.07E-5, 0.0022, -0.0202, 0.0652, 0.9417],
...                AMa) # doctest: +SKIP
```

Source: Wikipedia

Gallium Arsenide (GaAs):

- EgRef = 1.424
- dEgdT = -0.000433
- M = unknown

Source: [4]

pvlib.pvsystem.calcparams_pvsyst

pvlib.pvsystem.**calcparams_pvsyst**(*effective_irradiance*, *temp_cell*, *alpha_sc*, *gamma_ref*, *mu_gamma*, *I_L_ref*, *I_o_ref*, *R_sh_ref*, *R_sh_0*, *R_s*, *cells_in_series*, *R_sh_exp=5.5*, *EgRef=1.121*, *irrad_ref=1000*, *temp_ref=25*)

Calculates five parameter values for the single diode equation at effective irradiance and cell temperature using the PVsyst v6 model. The PVsyst v6 model is described in[1],[2],[3]. The five values returned by calcparams_pvsyst can be used by singlediode to calculate an IV curve.

Parameters

- **effective_irradiance** (*numeric*) – The irradiance (W/m2) that is converted to photocurrent.
- **temp_cell** (*numeric*) – The average cell temperature of cells within a module in C.
- **alpha_sc** (*float*) – The short-circuit current temperature coefficient of the module in units of A/C.
- **gamma_ref** (*float*) – The diode ideality factor
- **mu_gamma** (*float*) – The temperature coefficient for the diode ideality factor, 1/K

[1] K. Sauer, T. Roessler, C. W. Hansen, Modeling the Irradiance and Temperature Dependence of Photovoltaic Modules in PVsyst, IEEE Journal of Photovoltaics v5(1), January 2015.

[2] A. Mermoud, PV modules modelling, Presentation at the 2nd PV Performance Modeling Workshop, Santa Clara, CA, May 2013

[3] A. Mermoud, T. Lejeune, Performance Assessment of a Simulation Model for PV modules of any available technology, 25th European Photovoltaic Solar Energy Conference, Valencia, Spain, Sept. 2010

- **I_L_ref** (*float*) – The light-generated current (or photocurrent) at reference conditions, in amperes.
- **I_o_ref** (*float*) – The dark or diode reverse saturation current at reference conditions, in amperes.
- **R_sh_ref** (*float*) – The shunt resistance at reference conditions, in ohms.
- **R_sh_0** (*float*) – The shunt resistance at zero irradiance conditions, in ohms.
- **R_s** (*float*) – The series resistance at reference conditions, in ohms.
- **cells_in_series** (*integer*) – The number of cells connected in series.
- **R_sh_exp** (*float*) – The exponent in the equation for shunt resistance, unitless. Defaults to 5.5.
- **EgRef** (*float*) – The energy bandgap at reference temperature in units of eV. 1.121 eV for crystalline silicon. EgRef must be >0.
- **irrad_ref** (*float (optional, default=1000)*) – Reference irradiance in W/m^2.
- **temp_ref** (*float (optional, default=25)*) – Reference cell temperature in C.

Returns

- *Tuple of the following results*
- **photocurrent** (*numeric*) – Light-generated current in amperes
- **saturation_current** (*numeric*) – Diode saturation current in amperes
- **resistance_series** (*float*) – Series resistance in ohms
- **resistance_shunt** (*numeric*) – Shunt resistance in ohms
- **nNsVth** (*numeric*) – The product of the usual diode ideality factor (n, unitless), number of cells in series (Ns), and cell thermal voltage at specified effective irradiance and cell temperature.

References

See also:

calcparams_desoto(), *singlediode()*

pvlib.pvsystem.i_from_v

pvlib.pvsystem.**i_from_v**(*resistance_shunt*, *resistance_series*, *nNsVth*, *voltage*, *saturation_current*, *photocurrent*, *method='lambertw'*)

Device current at the given device voltage for the single diode model.

Uses the single diode model (SDM) as described in, e.g., Jain and Kapoor 2004[1].

The solution is per Eq 2 of [1] except when resistance_series=0, in which case the explict solution for current is used.

Ideal device parameters are specified by resistance_shunt=np.inf and resistance_series=0.

[1] A. Jain, A. Kapoor, "Exact analytical solutions of the parameters of real solar cells using Lambert W-function", Solar Energy Materials and Solar Cells, 81 (2004) 269-277.

Inputs to this function can include scalars and pandas.Series, but it is the caller's responsibility to ensure that the arguments are all float64 and within the proper ranges.

Parameters

- **resistance_shunt** (*numeric*) – Shunt resistance in ohms under desired IV curve conditions. Often abbreviated `Rsh`. 0 < resistance_shunt <= numpy.inf
- **resistance_series** (*numeric*) – Series resistance in ohms under desired IV curve conditions. Often abbreviated `Rs`. 0 <= resistance_series < numpy.inf
- **nNsVth** (*numeric*) – The product of three components. 1) The usual diode ideal factor (n), 2) the number of cells in series (Ns), and 3) the cell thermal voltage under the desired IV curve conditions (Vth). The thermal voltage of the cell (in volts) may be calculated as `k*temp_cell/q`, where k is Boltzmann's constant (J/K), temp_cell is the temperature of the p-n junction in Kelvin, and q is the charge of an electron (coulombs). 0 < nNsVth
- **voltage** (*numeric*) – The voltage in Volts under desired IV curve conditions.
- **saturation_current** (*numeric*) – Diode saturation current in amperes under desired IV curve conditions. Often abbreviated `I_0`. 0 < saturation_current
- **photocurrent** (*numeric*) – Light-generated current (photocurrent) in amperes under desired IV curve conditions. Often abbreviated `I_L`. 0 <= photocurrent
- **method** (*str*) – Method to use: `'lambertw'`, `'newton'`, or `'brentq'`. *Note*: `'brentq'` is limited to 1st quadrant only.

Returns **current** (*np.ndarray or scalar*)

References

pvlib.pvsystem.singlediode

pvlib.pvsystem.**singlediode**(*photocurrent*, *saturation_current*, *resistance_series*, *resistance_shunt*, *nNsVth*, *ivcurve_pnts=None*, *method='lambertw'*)

Solve the single-diode equation to obtain a photovoltaic IV curve.

Solves the single diode equation[1]

$$I = I_L - I_0 \left[\exp\left(\frac{V + IR_s}{nN_sV_{th}} \right) - 1 \right] - \frac{V + IR_s}{R_{sh}}$$

for I and V when given I_L, I_0, R_s, R_{sh}, and nN_sV_{th} which are described later. Returns a DataFrame which contains the 5 points on the I-V curve specified in[3]. If all I_L, I_0, R_s, R_{sh}, and nN_sV_{th} are scalar, a single curve is returned, if any are Series (of the same length), multiple IV curves are calculated.

The input parameters can be calculated from meteorological data using a function for a single diode model, e.g., *calcparams_desoto()*.

Parameters

- **photocurrent** (*numeric*) – Light-generated current I_L (photocurrent) `0 <= photocurrent`. [A]
- **saturation_current** (*numeric*) – Diode saturation I_0 current under desired IV curve conditions. `0 < saturation_current`. [A]

[1] S.R. Wenham, M.A. Green, M.E. Watt, "Applied Photovoltaics" ISBN 0 86758 909 4

[3] D. King et al, "Sandia Photovoltaic Array Performance Model", SAND2004-3535, Sandia National Laboratories, Albuquerque, NM

- **resistance_series** (*numeric*) – Series resistance R_s under desired IV curve conditions. `0 <= resistance_series < numpy.inf`. [ohm]
- **resistance_shunt** (*numeric*) – Shunt resistance R_{sh} under desired IV curve conditions. `0 < resistance_shunt <= numpy.inf`. [ohm]
- **nNsVth** (*numeric*) – The product of three components: 1) the usual diode ideality factor n, 2) the number of cells in series N_s, and 3) the cell thermal voltage V_{th}. The thermal voltage of the cell (in volts) may be calculated as $k_B T_c/q$, where k_B is Boltzmann's constant (J/K), T_c is the temperature of the p-n junction in Kelvin, and q is the charge of an electron (coulombs). `0 < nNsVth`. [V]
- **ivcurve_pnts** (*None or int, default None*) – Number of points in the desired IV curve. If None or 0, no points on the IV curves will be produced.
- **method** (*str, default 'lambertw'*) – Determines the method used to calculate points on the IV curve. The options are `'lambertw'`, `'newton'`, or `'brentq'`.

Returns

- *OrderedDict or DataFrame*
- *The returned dict-like object always contains the keys/columns –*
  - i_sc - short circuit current in amperes.
  - v_oc - open circuit voltage in volts.
  - i_mp - current at maximum power point in amperes.
  - v_mp - voltage at maximum power point in volts.
  - p_mp - power at maximum power point in watts.
  - i_x - current, in amperes, at `v = 0.5*v_oc`.
  - i_xx - current, in amperes, at `V = 0.5*(v_oc+v_mp)`.
- *If ivcurve_pnts is greater than 0, the output dictionary will also*
- *include the keys –*
  - i - IV curve current in amperes.
  - v - IV curve voltage in volts.
- *The output will be an OrderedDict if photocurrent is a scalar,*
- *array, or ivcurve_pnts is not None.*
- *The output will be a DataFrame if photocurrent is a Series and*
- *ivcurve_pnts is None.*

See also:

calcparams_desoto(), *calcparams_cec()*, *calcparams_pvsyst()*, *sapm()*, *pvlib.singlediode.bishop88()*

Notes

If the method is `'lambertw'` then the solution employed to solve the implicit diode equation utilizes the Lambert W function to obtain an explicit function of $V = f(I)$ and $I = f(V)$ as shown in[2].

[2] A. Jain, A. Kapoor, "Exact analytical solutions of the parameters of real solar cells using Lambert W-function", Solar Energy Materials and Solar Cells, 81 (2004) 269-277.

If the method is `'newton'` then the root-finding Newton-Raphson method is used. It should be safe for well behaved IV-curves, but the `'brentq'` method is recommended for reliability.

If the method is `'brentq'` then Brent's bisection search method is used that guarantees convergence by bounding the voltage between zero and open-circuit.

If the method is either `'newton'` or `'brentq'` and `ivcurve_pnts` are indicated, then *pvlib.singlediode.bishop88()*[4] is used to calculate the points on the IV curve points at diode voltages from zero to open-circuit voltage with a log spacing that gets closer as voltage increases. If the method is `'lambertw'` then the calculated points on the IV curve are linearly spaced.

References

pvlib.pvsystem.v_from_i

pvlib.pvsystem.**v_from_i**(*resistance_shunt*, *resistance_series*, *nNsVth*, *current*, *saturation_current*, *photocurrent*, *method='lambertw'*)

Device voltage at the given device current for the single diode model.

Uses the single diode model (SDM) as described in, e.g., Jain and Kapoor 2004[1]. The solution is per Eq 3 of[1] except when resistance_shunt=numpy.inf, in which case the explict solution for voltage is used. Ideal device parameters are specified by resistance_shunt=np.inf and resistance_series=0. Inputs to this function can include scalars and pandas.Series, but it is the caller's responsibility to ensure that the arguments are all float64 and within the proper ranges.

Parameters

- **resistance_shunt** (*numeric*) – Shunt resistance in ohms under desired IV curve conditions. Often abbreviated `Rsh`. 0 < resistance_shunt <= numpy.inf
- **resistance_series** (*numeric*) – Series resistance in ohms under desired IV curve conditions. Often abbreviated `Rs`. 0 <= resistance_series < numpy.inf
- **nNsVth** (*numeric*) – The product of three components. 1) The usual diode ideal factor (n), 2) the number of cells in series (Ns), and 3) the cell thermal voltage under the desired IV curve conditions (Vth). The thermal voltage of the cell (in volts) may be calculated as `k*temp_cell/q`, where k is Boltzmann's constant (J/K), temp_cell is the temperature of the p-n junction in Kelvin, and q is the charge of an electron (coulombs). 0 < nNsVth
- **current** (*numeric*) – The current in amperes under desired IV curve conditions.
- **saturation_current** (*numeric*) – Diode saturation current in amperes under desired IV curve conditions. Often abbreviated `I_0`. 0 < saturation_current
- **photocurrent** (*numeric*) – Light-generated current (photocurrent) in amperes under desired IV curve conditions. Often abbreviated `I_L`. 0 <= photocurrent
- **method** (*str*) – Method to use: `'lambertw'`, `'newton'`, or `'brentq'`. *Note*: `'brentq'` is limited to 1st quadrant only.

Returns **current** (*np.ndarray or scalar*)

[4] "Computer simulation of the effects of electrical mismatches in photovoltaic cell interconnection circuits" JW Bishop, Solar Cell (1988) https://doi.org/10.1016/0379-6787(88)90059-2

[1] A. Jain, A. Kapoor, "Exact analytical solutions of the parameters of real solar cells using Lambert W-function", Solar Energy Materials and Solar Cells, 81 (2004) 269-277.

References

pvlib.pvsystem.max_power_point

pvlib.pvsystem.**max_power_point**(*photocurrent*, *saturation_current*, *resistance_series*, *resistance_shunt*, *nNsVth*, *d2mutau=0*, *NsVbi=inf*, *method='brentq'*)

Given the single diode equation coefficients, calculates the maximum power point (MPP).

Parameters

- **photocurrent** (*numeric*) – photo-generated current [A]
- **saturation_current** (*numeric*) – diode reverse saturation current [A]
- **resistance_series** (*numeric*) – series resitance [ohms]
- **resistance_shunt** (*numeric*) – shunt resitance [ohms]
- **nNsVth** (*numeric*) – product of thermal voltage `Vth` [V], diode ideality factor `n`, and number of serices cells `Ns`
- **d2mutau** (*numeric, default 0*) – PVsyst parameter for cadmium-telluride (CdTe) and amorphous-silicon (a-Si) modules that accounts for recombination current in the intrinsic layer. The value is the ratio of intrinsic layer thickness squared d^2 to the diffusion length of charge carriers $\mu\tau$. [V]
- **NsVbi** (*numeric, default np.inf*) – PVsyst parameter for cadmium-telluride (CdTe) and amorphous-silicon (a-Si) modules that is the product of the PV module number of series cells `Ns` and the builtin voltage `Vbi` of the intrinsic layer. [V].
- **method** (*str*) – either `'newton'` or `'brentq'`

Returns *OrderedDict or pandas.Dataframe* – `(i_mp, v_mp, p_mp)`

Notes

Use this function when you only want to find the maximum power point. Use *singlediode()* when you need to find additional points on the IV curve. This function uses Brent's method by default because it is guaranteed to converge.

Low-level functions for solving the single diode equation.

singlediode.estimate_voc(photocurrent, ...)	Rough estimate of open circuit voltage useful for bounding searches for `i` of `v` when using *singlediode()*.
singlediode.bishop88(diode_voltage, ...[, ...])	Explicit calculation of points on the IV curve described by the single diode equation.
singlediode.bishop88_i_from_v(voltage, ...)	Find current given any voltage.
singlediode.bishop88_v_from_i(current, ...)	Find voltage given any current.
singlediode.bishop88_mpp(photocurrent, ...)	Find max power point.

pvlib.singlediode.estimate_voc

pvlib.singlediode.**estimate_voc**(*photocurrent*, *saturation_current*, *nNsVth*)

Rough estimate of open circuit voltage useful for bounding searches for `i` of `v` when using *singlediode()*.

Parameters

- **photocurrent** (*numeric*) – photo-generated current [A]
- **saturation_current** (*numeric*) – diode reverse saturation current [A]
- **nNsVth** (*numeric*) – product of thermal voltage `Vth` [V], diode ideality factor `n`, and number of series cells `Ns`

Returns *numeric* – rough estimate of open circuit voltage [V]

Notes

Calculating the open circuit voltage, V_{oc}, of an ideal device with infinite shunt resistance, $R_{sh} \to \infty$, and zero series resistance, $R_s = 0$, yields the following equation [1]. As an estimate of V_{oc} it is useful as an upper bound for the bisection method.

$$V_{oc,est} = nNsV_{th} \log\left(\frac{I_L}{I_0} + 1\right)$$

pvlib.singlediode.bishop88

pvlib.singlediode.**bishop88**(*diode_voltage*, *photocurrent*, *saturation_current*, *resistance_series*, *resistance_shunt*, *nNsVth*, *d2mutau=0*, *NsVbi=inf*, *breakdown_factor=0.0*, *breakdown_voltage=-5.5*, *breakdown_exp=3.28*, *gradients=False*)

Explicit calculation of points on the IV curve described by the single diode equation. Values are calculated as described in[1].

The single diode equation with recombination current and reverse bias breakdown is

$$I = I_L - I_0\left(\exp\frac{V_d}{nN_sV_{th}} - 1\right) - \frac{V_d}{R_{sh}} - \frac{I_L\frac{d^2}{\mu\tau}}{N_sV_{bi} - V_d} - a\frac{V_d}{R_{sh}}\left(1 - \frac{V_d}{V_{br}}\right)^{-m}$$

The input *diode_voltage* must be $V + IR_s$.

Warning:

- Usage of `d2mutau` is required with PVSyst coefficients for cadmium-telluride (CdTe) and amorphous-silicon (a:Si) PV modules only.
- Do not use `d2mutau` with CEC coefficients.

Parameters

- **diode_voltage** (*numeric*) – diode voltage V_d [V]
- **photocurrent** (*numeric*) – photo-generated current I_L [A]
- **saturation_current** (*numeric*) – diode reverse saturation current I_0 [A]
- **resistance_series** (*numeric*) – series resistance R_s [ohms]
- **resistance_shunt** (*numeric*) – shunt resistance R_{sh} [ohms]

[1] "Computer simulation of the effects of electrical mismatches in photovoltaic cell interconnection circuits" JW Bishop, Solar Cell (1988) DOI: 10.1016/0379-6787(88)90059-2

- **nNsVth** (*numeric*) – product of thermal voltage V_{th} [V], diode ideality factor n, and number of series cells N_s [V]
- **d2mutau** (*numeric, default 0*) – PVsyst parameter for cadmium-telluride (CdTe) and amorphous-silicon (a-Si) modules that accounts for recombination current in the intrinsic layer. The value is the ratio of intrinsic layer thickness squared d^2 to the diffusion length of charge carriers $\mu\tau$. [V]
- **NsVbi** (*numeric, default np.inf*) – PVsyst parameter for cadmium-telluride (CdTe) and amorphous-silicon (a-Si) modules that is the product of the PV module number of series cells N_s and the builtin voltage V_{bi} of the intrinsic layer. [V].
- **breakdown_factor** (*numeric, default 0*) – fraction of ohmic current involved in avalanche breakdown a. Default of 0 excludes the reverse bias term from the model. [unitless]
- **breakdown_voltage** (*numeric, default -5.5*) – reverse breakdown voltage of the photovoltaic junction V_{br} [V]
- **breakdown_exp** (*numeric, default 3.28*) – avalanche breakdown exponent m [unitless]
- **gradients** (*bool*) – False returns only I, V, and P. True also returns gradients

Returns *tuple* – currents [A], voltages [V], power [W], and optionally $\frac{dI}{dV_d}$, $\frac{dV}{dV_d}$, $\frac{dI}{dV}$, $\frac{dP}{dV}$, and $\frac{d^2P}{dVdV_d}$

Notes

The PVSyst thin-film recombination losses parameters `d2mutau` and `NsVbi` should only be applied to cadmium-telluride (CdTe) and amorphous- silicon (a-Si) PV modules,[2],[3]. The builtin voltage V_{bi} should account for all junctions. For example: tandem and triple junction cells would have builtin voltages of 1.8[V] and 2.7[V] respectively, based on the default of 0.9[V] for a single junction. The parameter `NsVbi` should only account for the number of series cells in a single parallel sub-string if the module has cells in parallel greater than 1.

References

pvlib.singlediode.bishop88_i_from_v

pvlib.singlediode.**bishop88_i_from_v**(*voltage*, *photocurrent*, *saturation_current*, *resistance_series*, *resistance_shunt*, *nNsVth*, *d2mutau=0*, *NsVbi=inf*, *breakdown_factor=0.0*, *breakdown_voltage=-5.5*, *breakdown_exp=3.28*, *method='newton'*)

Find current given any voltage.

Parameters

- **voltage** (*numeric*) – voltage (V) in volts [V]
- **photocurrent** (*numeric*) – photogenerated current (Iph or IL) [A]
- **saturation_current** (*numeric*) – diode dark or saturation current (Io or Isat) [A]

[2] "Improved equivalent circuit and Analytical Model for Amorphous Silicon Solar Cells and Modules." J. Mertens, et al., IEEE Transactions on Electron Devices, Vol 45, No 2, Feb 1998. DOI: 10.1109/16.658676

[3] "Performance assessment of a simulation model for PV modules of any available technology", André Mermoud and Thibault Lejeune, 25th EUPVSEC, 2010 DOI: 10.4229/25thEUPVSEC2010-4BV.1.114

- **resistance_series** (*numeric*) – series resistance (Rs) in [Ohm]
- **resistance_shunt** (*numeric*) – shunt resistance (Rsh) [Ohm]
- **nNsVth** (*numeric*) – product of diode ideality factor (n), number of series cells (Ns), and thermal voltage (Vth = k_b * T / q_e) in volts [V]
- **d2mutau** (*numeric, default 0*) – PVsyst parameter for cadmium-telluride (CdTe) and amorphous-silicon (a-Si) modules that accounts for recombination current in the intrinsic layer. The value is the ratio of intrinsic layer thickness squared d^2 to the diffusion length of charge carriers $\mu\tau$. [V]
- **NsVbi** (*numeric, default np.inf*) – PVsyst parameter for cadmium-telluride (CdTe) and amorphous-silicon (a-Si) modules that is the product of the PV module number of series cells `Ns` and the builtin voltage `Vbi` of the intrinsic layer. [V].
- **breakdown_factor** (*numeric, default 0*) – fraction of ohmic current involved in avalanche breakdown a. Default of 0 excludes the reverse bias term from the model. [unitless]
- **breakdown_voltage** (*numeric, default -5.5*) – reverse breakdown voltage of the photovoltaic junction V_{br} [V]
- **breakdown_exp** (*numeric, default 3.28*) – avalanche breakdown exponent m [unitless]
- **method** (*str, default 'newton'*) – Either `'newton'` or `'brentq'`. ''method" must be `'newton'` if `breakdown_factor` is not 0.

Returns **current** (*numeric*) – current (I) at the specified voltage (V). [A]

pvlib.singlediode.bishop88_v_from_i

pvlib.singlediode.**bishop88_v_from_i**(*current*, *photocurrent*, *saturation_current*, *resistance_series*, *resistance_shunt*, *nNsVth*, *d2mutau=0*, *NsVbi=inf*, *breakdown_factor=0.0*, *breakdown_voltage=-5.5*, *breakdown_exp=3.28*, *method='newton'*)

Find voltage given any current.

Parameters

- **current** (*numeric*) – current (I) in amperes [A]
- **photocurrent** (*numeric*) – photogenerated current (Iph or IL) [A]
- **saturation_current** (*numeric*) – diode dark or saturation current (Io or Isat) [A]
- **resistance_series** (*numeric*) – series resistance (Rs) in [Ohm]
- **resistance_shunt** (*numeric*) – shunt resistance (Rsh) [Ohm]
- **nNsVth** (*numeric*) – product of diode ideality factor (n), number of series cells (Ns), and thermal voltage (Vth = k_b * T / q_e) in volts [V]
- **d2mutau** (*numeric, default 0*) – PVsyst parameter for cadmium-telluride (CdTe) and amorphous-silicon (a-Si) modules that accounts for recombination current in the intrinsic layer. The value is the ratio of intrinsic layer thickness squared d^2 to the diffusion length of charge carriers $\mu\tau$. [V]

- **NsVbi** (*numeric, default np.inf*) – PVsyst parameter for cadmium-telluride (CdTe) and amorphous-silicon (a-Si) modules that is the product of the PV module number of series cells `Ns` and the builtin voltage `Vbi` of the intrinsic layer. [V].
- **breakdown_factor** (*numeric, default 0*) – fraction of ohmic current involved in avalanche breakdown a. Default of 0 excludes the reverse bias term from the model. [unitless]
- **breakdown_voltage** (*numeric, default -5.5*) – reverse breakdown voltage of the photovoltaic junction V_{br} [V]
- **breakdown_exp** (*numeric, default 3.28*) – avalanche breakdown exponent m [unitless]
- **method** (*str, default 'newton'*) – Either `'newton'` or `'brentq'`. ‘’method” must be `'newton'` if `breakdown_factor` is not 0.

Returns **voltage** (*numeric*) – voltage (V) at the specified current (I) in volts [V]

pvlib.singlediode.bishop88_mpp

pvlib.singlediode.**bishop88_mpp**(*photocurrent*, *saturation_current*, *resistance_series*, *resistance_shunt*, *nNsVth*, *d2mutau=0*, *NsVbi=inf*, *breakdown_factor=0.0*, *breakdown_voltage=-5.5*, *breakdown_exp=3.28*, *method='newton'*)

Find max power point.

Parameters

- **photocurrent** (*numeric*) – photogenerated current (Iph or IL) [A]
- **saturation_current** (*numeric*) – diode dark or saturation current (Io or Isat) [A]
- **resistance_series** (*numeric*) – series resistance (Rs) in [Ohm]
- **resistance_shunt** (*numeric*) – shunt resistance (Rsh) [Ohm]
- **nNsVth** (*numeric*) – product of diode ideality factor (n), number of series cells (Ns), and thermal voltage (Vth = k_b * T / q_e) in volts [V]
- **d2mutau** (*numeric, default 0*) – PVsyst parameter for cadmium-telluride (CdTe) and amorphous-silicon (a-Si) modules that accounts for recombination current in the intrinsic layer. The value is the ratio of intrinsic layer thickness squared d^2 to the diffusion length of charge carriers $\mu\tau$. [V]
- **NsVbi** (*numeric, default np.inf*) – PVsyst parameter for cadmium-telluride (CdTe) and amorphous-silicon (a-Si) modules that is the product of the PV module number of series cells `Ns` and the builtin voltage `Vbi` of the intrinsic layer. [V].
- **breakdown_factor** (*numeric, default 0*) – fraction of ohmic current involved in avalanche breakdown a. Default of 0 excludes the reverse bias term from the model. [unitless]
- **breakdown_voltage** (*numeric, default -5.5*) – reverse breakdown voltage of the photovoltaic junction V_{br} [V]
- **breakdown_exp** (*numeric, default 3.28*) – avalanche breakdown exponent m [unitless]
- **method** (*str, default 'newton'*) – Either `'newton'` or `'brentq'`. ‘’method” must be `'newton'` if `breakdown_factor` is not 0.

Returns *OrderedDict or pandas.DataFrame* – max power current `i_mp` [A], max power voltage `v_mp` [V], and max power `p_mp` [W]

SAPM model

Functions relevant for the SAPM model.

pvsystem.sapm(effective_irradiance, ...)	The Sandia PV Array Performance Model (SAPM) generates 5 points on a PV module's I-V curve (Voc, Isc, Ix, Ixx, Vmp/Imp) according to SAND2004-3535.
pvsystem.sapm_effective_irradiance(...)	Calculates the SAPM effective irradiance using the SAPM spectral loss and SAPM angle of incidence loss functions.
pvsystem.sapm_spectral_loss(...)	Calculates the SAPM spectral loss coefficient, F1.
pvsystem.sapm_aoi_loss(aoi, module[, upper])	Deprecated since version 0.7.
pvsystem.snlinverter(v_dc, p_dc, inverter)	Converts DC power and voltage to AC power using Sandia's Grid-Connected PV Inverter model.
pvsystem.adrinverter(v_dc, p_dc, inverter[, ...])	Converts DC power and voltage to AC power using Anton Driesse's Grid-Connected PV Inverter efficiency model
temperature.sapm_cell(poa_global, temp_air, ...)	Calculate cell temperature per the Sandia Array Performance Model.

pvlib.pvsystem.sapm

pvlib.pvsystem.**sapm**(*effective_irradiance*, *temp_cell*, *module*)

The Sandia PV Array Performance Model (SAPM) generates 5 points on a PV module's I-V curve (Voc, Isc, Ix, Ixx, Vmp/Imp) according to SAND2004-3535. Assumes a reference cell temperature of 25 C.

Parameters

- **effective_irradiance** (*numeric*) – Irradiance reaching the module's cells, after reflections and adjustment for spectrum. [W/m2]
- **temp_cell** (*numeric*) – Cell temperature [C].
- **module** (*dict-like*) – A dict or Series defining the SAPM parameters. See the notes section for more details.

Returns

A DataFrame with the columns –

- i_sc : Short-circuit current (A)
- i_mp : Current at the maximum-power point (A)
- v_oc : Open-circuit voltage (V)
- v_mp : Voltage at maximum-power point (V)
- p_mp : Power at maximum-power point (W)
- i_x : Current at module V = 0.5Voc, defines 4th point on I-V curve for modeling curve shape

- i_xx : Current at module V = 0.5(Voc+Vmp), defines 5th point on I-V curve for modeling curve shape

Notes

The SAPM parameters which are required in `module` are listed in the following table.

The Sandia module database contains parameter values for a limited set of modules. The CEC module database does not contain these parameters. Both databases can be accessed using *retrieve_sam()*.

Key	Description
A0-A4	The airmass coefficients used in calculating effective irradiance
B0-B5	The angle of incidence coefficients used in calculating effective irradiance
C0-C7	The empirically determined coefficients relating Imp, Vmp, Ix, and Ixx to effective irradiance
Isco	Short circuit current at reference condition (amps)
Impo	Maximum power current at reference condition (amps)
Voco	Open circuit voltage at reference condition (amps)
Vmpo	Maximum power voltage at reference condition (amps)
Aisc	Short circuit current temperature coefficient at reference condition (1/C)
Aimp	Maximum power current temperature coefficient at reference condition (1/C)
Bvoco	Open circuit voltage temperature coefficient at reference condition (V/C)
Mbvoc	Coefficient providing the irradiance dependence for the BetaVoc temperature coefficient at reference irradiance (V/C)
Bvmpo	Maximum power voltage temperature coefficient at reference condition
Mbvmp	Coefficient providing the irradiance dependence for the BetaVmp temperature coefficient at reference irradiance (V/C)
N	Empirically determined “diode factor” (dimensionless)
Cells_in_Series	Number of cells in series in a module’s cell string(s)
IXO	Ix at reference conditions
IXXO	Ixx at reference conditions
FD	Fraction of diffuse irradiance used by module

References

See also:

retrieve_sam(), *pvlib.temperature.sapm_cell()*, *pvlib.temperature.sapm_module()*

pvlib.pvsystem.sapm_effective_irradiance

pvlib.pvsystem.**sapm_effective_irradiance**(*poa_direct*, *poa_diffuse*, *airmass_absolute*, *aoi*, *module*)

Calculates the SAPM effective irradiance using the SAPM spectral loss and SAPM angle of incidence loss functions.

Parameters

- **poa_direct** (*numeric*) – The direct irradiance incident upon the module. [W/m2]
- **poa_diffuse** (*numeric*) – The diffuse irradiance incident on module. [W/m2]
- **airmass_absolute** (*numeric*) – Absolute airmass. [unitless]

- **aoi** (*numeric*) – Angle of incidence. [degrees]
- **module** (*dict-like*) – A dict, Series, or DataFrame defining the SAPM performance parameters. See the *sapm()* notes section for more details.

Returns **effective_irradiance** (*numeric*) – Effective irradiance accounting for reflections and spectral content. [W/m2]

Notes

The SAPM model for effective irradiance[1] translates broadband direct and diffuse irradiance on the plane of array to the irradiance absorbed by a module's cells.

The model is .. math:

```
`Ee = f_1(AM_a) (E_b f_2(AOI) + f_d E_d)`
```

where Ee is effective irradiance (W/m2), f_1 is a fourth degree polynomial in air mass AM_a, E_b is beam (direct) irradiance on the plane of array, E_d is diffuse irradiance on the plane of array, f_2 is a fifth degree polynomial in the angle of incidence AOI, and f_d is the fraction of diffuse irradiance on the plane of array that is not reflected away.

References

See also:

pvlib.iam.sapm(), *pvlib.pvsystem.sapm_spectral_loss()*, *pvlib.pvsystem.sapm()*

pvlib.pvsystem.sapm_spectral_loss

pvlib.pvsystem.**sapm_spectral_loss**(*airmass_absolute*, *module*)

Calculates the SAPM spectral loss coefficient, F1.

Parameters

- **airmass_absolute** (*numeric*) – Absolute airmass
- **module** (*dict-like*) – A dict, Series, or DataFrame defining the SAPM performance parameters. See the *sapm()* notes section for more details.

Returns **F1** (*numeric*) – The SAPM spectral loss coefficient.

Notes

nan airmass values will result in 0 output.

pvlib.pvsystem.sapm_aoi_loss

pvlib.pvsystem.**sapm_aoi_loss**(*aoi*, *module*, *upper=None*)

Deprecated since version 0.7: The sapm_aoi_loss function was deprecated in pvlib 0.7 and will be removed in 0.8. Use iam.sapm instead.

Determine the incidence angle modifier (IAM) using the SAPM model.

[1] D. King et al, "Sandia Photovoltaic Array Performance Model", SAND2004-3535, Sandia National Laboratories, Albuquerque, NM

Parameters

- **aoi** (*numeric*) – Angle of incidence in degrees. Negative input angles will return zeros.
- **module** (*dict-like*) – A dict or Series with the SAPM IAM model parameters. See the *sapm()* notes section for more details.
- **upper** (*None or float, default None*) – Upper limit on the results.

Returns **iam** (*numeric*) – The SAPM angle of incidence loss coefficient, termed F2 in[1].

Notes

The SAPM[1] traditionally does not define an upper limit on the AOI loss function and values slightly exceeding 1 may exist for moderate angles of incidence (15-40 degrees). However, users may consider imposing an upper limit of 1.

References

See also:

pvlib.iam.physical(), *pvlib.iam.ashrae()*, *pvlib.iam.martin_ruiz()*, *pvlib.iam.interp()*

pvlib.pvsystem.snlinverter

pvlib.pvsystem.**snlinverter**(*v_dc*, *p_dc*, *inverter*)

Converts DC power and voltage to AC power using Sandia's Grid-Connected PV Inverter model.

Determines the AC power output of an inverter given the DC voltage, DC power, and appropriate Sandia Grid-Connected Photovoltaic Inverter Model parameters. The output, ac_power, is clipped at the maximum power output, and gives a negative power during low-input power conditions, but does NOT account for maximum power point tracking voltage windows nor maximum current or voltage limits on the inverter.

Parameters

- **v_dc** (*numeric*) – DC voltages, in volts, which are provided as input to the inverter. Vdc must be >= 0.
- **p_dc** (*numeric*) – A scalar or DataFrame of DC powers, in watts, which are provided as input to the inverter. Pdc must be >= 0.
- **inverter** (*dict-like*) – A dict-like object defining the inverter to be used, giving the inverter performance parameters according to the Sandia Grid-Connected Photovoltaic Inverter Model (SAND 2007-5036)[1]. A set of inverter performance parameters are provided with pvlib, or may be generated from a System Advisor Model (SAM)[2] library using retrievesam. See Notes for required keys.

Returns **ac_power** (*numeric*) – Modeled AC power output given the input DC voltage, Vdc, and input DC power, Pdc. When ac_power would be greater than Pac0, it is set to Pac0 to represent inverter "clipping". When ac_power would be less than Ps0 (startup power required), then ac_power is set to -1*abs(Pnt) to represent nightly power losses. ac_power is not adjusted for

[1] King, D. et al, 2004, "Sandia Photovoltaic Array Performance Model", SAND Report 3535, Sandia National Laboratories, Albuquerque, NM.
[1] SAND2007-5036, "Performance Model for Grid-Connected Photovoltaic Inverters by D. King, S. Gonzalez, G. Galbraith, W. Boyson
[2] System Advisor Model web page. https://sam.nrel.gov.

maximum power point tracking (MPPT) voltage windows or maximum current limits of the inverter.

Notes

Required inverter keys are:

Column	Description
Pac0	AC-power output from inverter based on input power and voltage (W)
Pdc0	DC-power input to inverter, typically assumed to be equal to the PV array maximum power (W)
Vdc0	DC-voltage level at which the AC-power rating is achieved at the reference operating condition (V)
Ps0	DC-power required to start the inversion process, or self-consumption by inverter, strongly influences inverter efficiency at low power levels (W)
C0	Parameter defining the curvature (parabolic) of the relationship between ac-power and dc-power at the reference operating condition, default value of zero gives a linear relationship (1/W)
C1	Empirical coefficient allowing Pdco to vary linearly with dc-voltage input, default value is zero (1/V)
C2	Empirical coefficient allowing Pso to vary linearly with dc-voltage input, default value is zero (1/V)
C3	Empirical coefficient allowing Co to vary linearly with dc-voltage input, default value is zero (1/V)
Pnt	AC-power consumed by inverter at night (night tare) to maintain circuitry required to sense PV array voltage (W)

References

See also:

sapm(), *singlediode()*

pvlib.pvsystem.adrinverter

pvlib.pvsystem.**adrinverter**(*v_dc*, *p_dc*, *inverter*, *vtol=0.1*)

Converts DC power and voltage to AC power using Anton Driesse's Grid-Connected PV Inverter efficiency model

Parameters

- **v_dc** (*numeric*) – A scalar or pandas series of DC voltages, in volts, which are provided as input to the inverter. If Vdc and Pdc are vectors, they must be of the same size. v_dc must be >= 0. (V)
- **p_dc** (*numeric*) – A scalar or pandas series of DC powers, in watts, which are provided as input to the inverter. If Vdc and Pdc are vectors, they must be of the same size. p_dc must be >= 0. (W)
- **inverter** (*dict-like*) – A dict-like object defining the inverter to be used, giving the inverter performance parameters according to the model developed by Anton Driesse [1]. A set of inverter performance parameters may be loaded from the supplied data table using retrievesam. See Notes for required keys.
- **vtol** (*numeric, default 0.1*) – A unit-less fraction that determines how far the efficiency model is allowed to extrapolate beyond the inverter's normal input voltage operating range. 0.0 <= vtol <= 1.0

Returns **ac_power** (*numeric*) – A numpy array or pandas series of modeled AC power output given the input DC voltage, v_dc, and input DC power, p_dc. When ac_power would be greater than pac_max, it is set to p_max to represent inverter "clipping". When ac_power would be less than -p_nt (energy consumed rather than produced) then ac_power is set to -p_nt to represent nightly power losses. ac_power is not adjusted for maximum power point tracking (MPPT) voltage windows or maximum current limits of the inverter.

Notes

Required inverter keys are:

Column	Description
p_nom	The nominal power value used to normalize all power values, typically the DC power needed to produce maximum AC power output, (W).
v_nom	The nominal DC voltage value used to normalize DC voltage values, typically the level at which the highest efficiency is achieved, (V).
pac_max	The maximum AC output power value, used to clip the output if needed, (W).
ce_list	This is a list of 9 coefficients that capture the influence of input voltage and power on inverter losses, and thereby efficiency.
p_nt	ac-power consumed by inverter at night (night tare) to maintain circuitry required to sense PV array voltage, (W).

References

See also:

sapm(), *singlediode()*

Pvsyst model

Functions relevant for the Pvsyst model.

temperature.pvsyst_cell(poa_global, temp_air)	Calculate cell temperature using an empirical heat loss factor model as implemented in PVsyst.

PVWatts model

pvsystem.pvwatts_dc(g_poa_effective, ...[, ...])	Implements NREL's PVWatts DC power model.
pvsystem.pvwatts_ac(pdc, pdc0[, ...])	Implements NREL's PVWatts inverter model.
pvsystem.pvwatts_losses([soiling, shading, ...])	Implements NREL's PVWatts system loss model.

pvlib.pvsystem.pvwatts_dc

pvlib.pvsystem.**pvwatts_dc**(*g_poa_effective*, *temp_cell*, *pdc0*, *gamma_pdc*, *temp_ref=25.0*)

Implements NREL's PVWatts DC power model. The PVWatts DC model[1] is:

$$P_{dc} = \frac{G_{poaeff}}{1000} P_{dc0}(1 + \gamma_{pdc}(T_{cell} - T_{ref}))$$

Note that the pdc0 is also used as a symbol in *pvwatts_ac()*. pdc0 in this function refers to the DC power of the modules at reference conditions. pdc0 in *pvwatts_ac()* refers to the DC power input limit of the inverter.

Parameters

- **g_poa_effective** (*numeric*) – Irradiance transmitted to the PV cells in units of W/m**2. To be fully consistent with PVWatts, the user must have already applied angle of incidence losses, but not soiling, spectral, etc.
- **temp_cell** (*numeric*) – Cell temperature in degrees C.
- **pdc0** (*numeric*) – Power of the modules at 1000 W/m2 and cell reference temperature.
- **gamma_pdc** (*numeric*) – The temperature coefficient in units of 1/C. Typically -0.002 to -0.005 per degree C.
- **temp_ref** (*numeric, default 25.0*) – Cell reference temperature. PVWatts defines it to be 25 C and is included here for flexibility.

Returns **pdc** (*numeric*) – DC power.

References

pvlib.pvsystem.pvwatts_ac

pvlib.pvsystem.**pvwatts_ac**(*pdc*, *pdc0*, *eta_inv_nom=0.96*, *eta_inv_ref=0.9637*)

Implements NREL's PVWatts inverter model. The PVWatts inverter model[1] is:

$$\eta = \frac{\eta_{nom}}{\eta_{ref}}(-0.0162\zeta - \frac{0.0059}{\zeta} + 0.9858)$$

$$P_{ac} = \min(\eta P_{dc}, P_{ac0})$$

where $\zeta = P_{dc}/P_{dc0}$ and $P_{dc0} = P_{ac0}/\eta_{nom}$.

Note that the pdc0 is also used as a symbol in *pvwatts_dc()*. pdc0 in this function refers to the DC power input limit of the inverter. pdc0 in *pvwatts_dc()* refers to the DC power of the modules at reference conditions.

Parameters

- **pdc** (*numeric*) – DC power.
- **pdc0** (*numeric*) – DC input limit of the inverter.
- **eta_inv_nom** (*numeric, default 0.96*) – Nominal inverter efficiency.
- **eta_inv_ref** (*numeric, default 0.9637*) – Reference inverter efficiency. PVWatts defines it to be 0.9637 and is included here for flexibility.

Returns **pac** (*numeric*) – AC power.

[1] A. P. Dobos, "PVWatts Version 5 Manual" http://pvwatts.nrel.gov/downloads/pvwattsv5.pdf (2014).

[1] A. P. Dobos, "PVWatts Version 5 Manual," http://pvwatts.nrel.gov/downloads/pvwattsv5.pdf (2014).

References

pvlib.pvsystem.pvwatts_losses

pvlib.pvsystem.**pvwatts_losses**(*soiling=2*, *shading=3*, *snow=0*, *mismatch=2*, *wiring=2*, *connections=0.5*, *lid=1.5*, *nameplate_rating=1*, *age=0*, *availability=3*)

Implements NREL's PVWatts system loss model. The PVWatts loss model[1] is:

$$L_{total}(\%) = 100[1 - \Pi_i(1 - \frac{L_i}{100})]$$

All parameters must be in units of %. Parameters may be array-like, though all array sizes must match.

Parameters

- **soiling** (*numeric, default 2*) –
- **shading** (*numeric, default 3*) –
- **snow** (*numeric, default 0*) –
- **mismatch** (*numeric, default 2*) –
- **wiring** (*numeric, default 2*) –
- **connections** (*numeric, default 0.5*) –
- **lid** (*numeric, default 1.5*) – Light induced degradation
- **nameplate_rating** (*numeric, default 1*) –
- **age** (*numeric, default 0*) –
- **availability** (*numeric, default 3*) –

Returns **losses** (*numeric*) – System losses in units of %.

References

Functions for fitting diode models

ivtools.fit_sde_sandia(voltage, current[, ...])	Fits the single diode equation (SDE) to an IV curve.
ivtools.fit_sdm_cec_sam(celltype, v_mp, ...)	Estimates parameters for the CEC single diode model (SDM) using the SAM SDK.
ivtools.fit_sdm_desoto(v_mp, i_mp, v_oc, ...)	Calculates the parameters for the De Soto single diode model.

pvlib.ivtools.fit_sde_sandia

pvlib.ivtools.**fit_sde_sandia**(*voltage*, *current*, *v_oc=None*, *i_sc=None*, *v_mp_i_mp=None*, *vlim=0.2*, *ilim=0.1*)

Fits the single diode equation (SDE) to an IV curve.

Parameters

- **voltage** (*ndarray*) – 1D array of *float* type containing voltage at each point on the IV curve, increasing from 0 to v_oc inclusive [V]

[1] A. P. Dobos, "PVWatts Version 5 Manual" http://pvwatts.nrel.gov/downloads/pvwattsv5.pdf (2014).

- **current** (*ndarray*) – 1D array of *float* type containing current at each point on the IV curve, from i_sc to 0 inclusive [A]
- **v_oc** (*float, default None*) – Open circuit voltage [V]. If not provided, v_oc is taken as the last point in the voltage array.
- **i_sc** (*float, default None*) – Short circuit current [A]. If not provided, i_sc is taken as the first point in the current array.
- **v_mp_i_mp** (*tuple of float, default None*) – Voltage, current at maximum power point in units of [V], [A]. If not provided, the maximum power point is found at the maximum of voltage times current.
- **vlim** (*float, default 0.2*) – Defines portion of IV curve where the exponential term in the single diode equation can be neglected, i.e. voltage <= vlim x v_oc [V]
- **ilim** (*float, default 0.1*) – Defines portion of the IV curve where the exponential term in the single diode equation is signficant, approximately defined by current < (1 - ilim) x i_sc [A]

Returns

- **photocurrent** (*float*) – photocurrent [A]
- **saturation_current** (*float*) – dark (saturation) current [A]
- **resistance_shunt** (*float*) – shunt (parallel) resistance, in ohms
- **resistance_series** (*float*) – series resistance, in ohms
- **nNsVth** (*float*) – product of thermal voltage Vth [V], diode ideality factor n, and number of series cells Ns

Raises RuntimeError – If parameter extraction is not successful.

Notes

Inputs voltage, current, v_oc, i_sc and v_mp_i_mp are assumed to be from a single IV curve at constant irradiance and cell temperature.

fit_single_diode_sandia() obtains values for the five parameters for the single diode equation[1]:

$$I = I_L - I_0(\exp\frac{V + IR_s}{nNsVth} - 1) - \frac{V + IR_s}{R_{sh}}$$

See pvsystem.singlediode() for definition of the parameters.

The extraction method[2] proceeds in six steps.

1. In the single diode equation, replace $R_{sh} = 1/G_p$ and re-arrange

$$I = \frac{I_L}{1 + G_p R_s} - \frac{G_p V}{1 + G_p R_s} - \frac{I_0}{1 + G_p R_s}(\exp(\frac{V + IR_s}{nNsVth}) - 1)$$

2. The linear portion of the IV curve is defined as $V \leq vlim \times v_oc$. Over this portion of the IV curve,

$$\frac{I_0}{1 + G_p R_s}(\exp(\frac{V + IR_s}{nNsVth}) - 1) \approx 0$$

3. Fit the linear portion of the IV curve with a line.

[1] S.R. Wenham, M.A. Green, M.E. Watt, "Applied Photovoltaics" ISBN 0 86758 909 4

[2] C. B. Jones, C. W. Hansen, Single Diode Parameter Extraction from In-Field Photovoltaic I-V Curves on a Single Board Computer, 46th IEEE Photovoltaic Specialist Conference, Chicago, IL, 2019

$$I \approx \frac{I_L}{1 + G_p R_s} - \frac{G_p V}{1 + G_p R_s}$$
$$= \beta_0 + \beta_1 V$$

4. The exponential portion of the IV curve is defined by $\beta_0 + \beta_1 \times V - I > ilim \times i_s c$. Over this portion of the curve, $exp((V + IRs)/nNsVth) >> 1$ so that

$$\exp(\frac{V + IR_s}{nNsVth}) - 1 \approx \exp(\frac{V + IR_s}{nNsVth})$$

5. Fit the exponential portion of the IV curve.

$$\log(\beta_0 - \beta_1 V - I) \approx \log(\frac{I_0}{1 + G_p R_s} + \frac{V}{nNsVth} + \frac{IR_s}{nNsVth}$$
$$= \beta_2 + beta_3 V + \beta_4 I$$

6. Calculate values for `IL, I0, Rs, Rsh,` and `nNsVth` from the regression coefficents $\beta_0, \beta_1, \beta_3$ and β_4.

References

pvlib.ivtools.fit_sdm_cec_sam

pvlib.ivtools.**fit_sdm_cec_sam**(*celltype*, *v_mp*, *i_mp*, *v_oc*, *i_sc*, *alpha_sc*, *beta_voc*, *gamma_pmp*, *cells_in_series*, *temp_ref=25*)

Estimates parameters for the CEC single diode model (SDM) using the SAM SDK.

Parameters

- **celltype** (*str*) – Value is one of 'monoSi', 'multiSi', 'polySi', 'cis', 'cigs', 'cdte', 'amorphous'
- **v_mp** (*float*) – Voltage at maximum power point [V]
- **i_mp** (*float*) – Current at maximum power point [A]
- **v_oc** (*float*) – Open circuit voltage [V]
- **i_sc** (*float*) – Short circuit current [A]
- **alpha_sc** (*float*) – Temperature coefficient of short circuit current [A/C]
- **beta_voc** (*float*) – Temperature coefficient of open circuit voltage [V/C]
- **gamma_pmp** (*float*) – Temperature coefficient of power at maximum point point [%/C]
- **cells_in_series** (*int*) – Number of cells in series
- **temp_ref** (*float, default 25*) – Reference temperature condition [C]

Returns

- **I_L_ref** (*float*) – The light-generated current (or photocurrent) at reference conditions [A]
- **I_o_ref** (*float*) – The dark or diode reverse saturation current at reference conditions [A]
- **R_sh_ref** (*float*) – The shunt resistance at reference conditions, in ohms.
- **R_s** (*float*) – The series resistance at reference conditions, in ohms.
- **a_ref** (*float*) – The product of the usual diode ideality factor `n` (unitless), number of cells in series `Ns`, and cell thermal voltage at reference conditions [V]

- **Adjust** (*float*) – The adjustment to the temperature coefficient for short circuit current, in percent.

Raises

- `ImportError` – If NREL-PySAM is not installed.
- `RuntimeError` – If parameter extraction is not successful.

Notes

Inputs `v_mp`, `v_oc`, `i_mp` and `i_sc` are assumed to be from a single IV curve at constant irradiance and cell temperature. Irradiance is not explicitly used by the fitting procedure. The irradiance level at which the input IV curve is determined and the specified cell temperature `temp_ref` are the reference conditions for the output parameters `I_L_ref`, `I_o_ref`, `R_sh_ref`, `R_s`, `a_ref` and `Adjust`.

References

pvlib.ivtools.fit_sdm_desoto

pvlib.ivtools.**fit_sdm_desoto**(*v_mp*, *i_mp*, *v_oc*, *i_sc*, *alpha_sc*, *beta_voc*, *cells_in_series*, *EgRef=1.121*, *dEgdT=-0.0002677*, *temp_ref=25*, *irrad_ref=1000*, *root_kwargs={}*)

Calculates the parameters for the De Soto single diode model. This procedure (described in[1]) has the advantage of using common specifications given by manufacturers in the datasheets of PV modules.

The solution is found using the scipy.optimize.root() function, with the corresponding default solver method 'hybr'. No restriction is put on the fit variables, i.e. series or shunt resistance could go negative. Nevertheless, if it happens, check carefully the inputs and their units; alpha_sc and beta_voc are often given in %/K in manufacturers datasheets and should be given in A/K and V/K here.

The parameters returned by this function can be used by pvsystem.calcparams_desoto to calculate the values at different irradiance and cell temperature.

Parameters

- **v_mp** (*float*) – Module voltage at the maximum-power point at reference conditions [V].
- **i_mp** (*float*) – Module current at the maximum-power point at reference conditions [A].
- **v_oc** (*float*) – Open-circuit voltage at reference conditions [V].
- **i_sc** (*float*) – Short-circuit current at reference conditions [A].
- **alpha_sc** (*float*) – The short-circuit current (i_sc) temperature coefficient of the module [A/K].
- **beta_voc** (*float*) – The open-circuit voltage (v_oc) temperature coefficient of the module [V/K].
- **cells_in_series** (*integer*) – Number of cell in the module.
- **EgRef** (*float, default 1.121 eV - value for silicon*) – Energy of bandgap of semi-conductor used [eV]

[1] W. De Soto et al., "Improvement and validation of a model for photovoltaic array performance", Solar Energy, vol 80, pp. 78-88, 2006.

- **dEgdT** (*float, default -0.0002677 - value for silicon*) – Variation of bandgap according to temperature [eV/K]
- **temp_ref** (*float, default 25*) – Reference temperature condition [C]
- **irrad_ref** (*float, default 1000*) – Reference irradiance condition [W/m2]
- **root_kwargs** (*dictionary, default None*) – Dictionary of arguments to pass onto scipy.optimize.root()

Returns

- *Dictionary with the following elements –*
  - `I_L_ref` (float) – Light-generated current at reference conditions [A]
  - `I_o_ref` (float) – Diode saturation current at reference conditions [A]
  - `R_s` (float) – Series resistance [ohms]
  - `R_sh_ref` (float) – Shunt resistance at reference conditions [ohms].
  - `a_ref` (float) – Modified ideality factor at reference conditions. The product of the usual diode ideality factor (n, unitless), number of cells in series (Ns), and cell thermal voltage at specified effective irradiance and cell temperature.
  - `alpha_sc` (float) – The short-circuit current (i_sc) temperature coefficient of the module [A/K].
  - `EgRef` (float) – Energy of bandgap of semi-conductor used [eV]
  - `dEgdT` (float) – Variation of bandgap according to temperature [eV/K]
  - `irrad_ref` (float) – Reference irradiance condition [W/m2]
  - `temp_ref` (float) – Reference temperature condition [C]
- *scipy.optimize.OptimizeResult* – Optimization result of scipy.optimize.root(). See scipy.optimize.OptimizeResult for more details.

References

Other

pvsystem.retrieve_sam([name, path])	Retrieve latest module and inverter info from a local file or the SAM website.
pvsystem.systemdef(meta, surface_tilt, ...)	Generates a dict of system parameters used throughout a simulation.
pvsystem.scale_voltage_current_power(data, ...])	Scales the voltage, current, and power of the DataFrames returned by `singlediode()` and `sapm()`.

pvlib.pvsystem.retrieve_sam

pvlib.pvsystem.**retrieve_sam**(*name=None*, *path=None*)

Retrieve latest module and inverter info from a local file or the SAM website.

This function will retrieve either:

- CEC module database

- Sandia Module database
- CEC Inverter database
- Anton Driesse Inverter database

and return it as a pandas DataFrame.

Parameters

- **name** (*None or string, default None*) – Name can be one of:
  - 'CECMod' - returns the CEC module database
  - 'CECInverter' - returns the CEC Inverter database
  - 'SandiaInverter' - returns the CEC Inverter database (CEC is only current inverter db available; tag kept for backwards compatibility)
  - 'SandiaMod' - returns the Sandia Module database
  - 'ADRInverter' - returns the ADR Inverter database
- **path** (*None or string, default None*) – Path to the SAM file. May also be a URL.

Returns **samfile** (*DataFrame*) – A DataFrame containing all the elements of the desired database. Each column represents a module or inverter, and a specific dataset can be retrieved by the command

Raises `ValueError` – If no name or path is provided.

Notes

Files available at https://github.com/NREL/SAM/tree/develop/deploy/libraries

Documentation for module and inverter data sets: https://sam.nrel.gov/photovoltaic/pv-sub-page-2.html

Examples

```
>>> from pvlib import pvsystem
>>> invdb = pvsystem.retrieve_sam('CECInverter')
>>> inverter = invdb.AE_Solar_Energy__AE6_0__277V__277V__CEC_2012_
>>> inverter
Vac           277.000000
Paco         6000.000000
Pdco         6165.670000
Vdco          361.123000
Pso            36.792300
C0             -0.000002
C1             -0.000047
C2             -0.001861
C3              0.000721
Pnt             0.070000
Vdcmax        600.000000
Idcmax         32.000000
Mppt_low      200.000000
Mppt_high     500.000000
Name: AE_Solar_Energy__AE6_0__277V__277V__CEC_2012_, dtype: float64
```

pvlib.pvsystem.systemdef

pvlib.pvsystem.**systemdef**(*meta*, *surface_tilt*, *surface_azimuth*, *albedo*, *modules_per_string*, *strings_per_inverter*)

Generates a dict of system parameters used throughout a simulation.

Parameters

- **meta** (*dict*) – meta dict either generated from a TMY file using readtmy2 or readtmy3, or a dict containing at least the following fields:

meta field	format	description
meta.altitude	Float	site elevation
meta.latitude	Float	site latitude
meta.longitude	Float	site longitude
meta.Name	String	site name
meta.State	String	state
meta.TZ	Float	timezone

- **surface_tilt** (*float or Series*) – Surface tilt angles in decimal degrees. The tilt angle is defined as degrees from horizontal (e.g. surface facing up = 0, surface facing horizon = 90)
- **surface_azimuth** (*float or Series*) – Surface azimuth angles in decimal degrees. The azimuth convention is defined as degrees east of north (North=0, South=180, East=90, West=270).
- **albedo** (*float or Series*) – Ground reflectance, typically 0.1-0.4 for surfaces on Earth (land), may increase over snow, ice, etc. May also be known as the reflection coefficient. Must be >=0 and <=1.
- **modules_per_string** (*int*) – Number of modules connected in series in a string.
- **strings_per_inverter** (*int*) – Number of strings connected in parallel.

Returns

Result (*dict*) –

A dict with the following fields.

- 'surface_tilt'
- 'surface_azimuth'
- 'albedo'
- 'modules_per_string'
- 'strings_per_inverter'
- 'latitude'
- 'longitude'
- 'tz'
- 'name'
- 'altitude'

See also:

pvlib.iotools.read_tmy3(), *pvlib.iotools.read_tmy2()*

pvlib.pvsystem.scale_voltage_current_power

pvlib.pvsystem.**scale_voltage_current_power**(*data*, *voltage=1*, *current=1*)

Scales the voltage, current, and power of the DataFrames returned by *singlediode()* and *sapm()*.

Parameters

- **data** (*DataFrame*) – Must contain columns *'v_mp', 'v_oc', 'i_mp','i_x', 'i_xx', 'i_sc', 'p_mp'*.
- **voltage** (*numeric, default 1*) – The amount by which to multiply the voltages.
- **current** (*numeric, default 1*) – The amount by which to multiply the currents.

Returns **scaled_data** (*DataFrame*) – A scaled copy of the input data. *'p_mp'* is scaled by *voltage * current*.

3.12.7 Effects on PV System Output

snow.coverage_nrel(snowfall, poa_irradiance, ...)	Calculates the fraction of the slant height of a row of modules covered by snow at every time step.
snow.fully_covered_nrel(snowfall[, ...])	Calculates the timesteps when the row's slant height is fully covered by snow.
snow.dc_loss_nrel(snow_coverage, num_strings)	Calculates the fraction of DC capacity lost due to snow coverage.

pvlib.snow.coverage_nrel

pvlib.snow.**coverage_nrel**(*snowfall*, *poa_irradiance*, *temp_air*, *surface_tilt*, *initial_coverage=0*, *threshold_snowfall=1.0*, *can_slide_coefficient=-80.0*, *slide_amount_coefficient=0.197*)

Calculates the fraction of the slant height of a row of modules covered by snow at every time step.

Implements the model described in[1] with minor improvements in[2], with the change that the output is in fraction of the row's slant height rather than in tenths of the row slant height. As described in[1], model validation focused on fixed tilt systems.

Parameters

- **snowfall** (*Series*) – Accumulated snowfall within each time period. [cm]
- **poa_irradiance** (*Series*) – Total in-plane irradiance [W/m^2]
- **temp_air** (*Series*) – Ambient air temperature [C]
- **surface_tilt** (*numeric*) – Tilt of module's from horizontal, e.g. surface facing up = 0, surface facing horizon = 90. [degrees]
- **initial_coverage** (*float, default 0*) – Fraction of row's slant height that is covered with snow at the beginning of the simulation. [unitless]
- **threshold_snowfall** (*float, default 1.0*) – Hourly snowfall above which snow coverage is set to the row's slant height. [cm/hr]

[1] Marion, B.; Schaefer, R.; Caine, H.; Sanchez, G. (2013). "Measured and modeled photovoltaic system energy losses from snow for Colorado and Wisconsin locations." Solar Energy 97; pp.112-121.

[2] Ryberg, D; Freeman, J. (2017). "Integration, Validation, and Application of a PV Snow Coverage Model in SAM" NREL Technical Report NREL/TP-6A20-68705

- **can_slide_coefficient** (*float, default -80.*) – Coefficient to determine if snow can slide given irradiance and air temperature. [W/(m^2 C)]
- **slide_amount_coefficient** (*float, default 0.197*) – Coefficient to determine fraction of snow that slides off in one hour. [unitless]

Returns **snow_coverage** (*Series*) – The fraction of the slant height of a row of modules that is covered by snow at each time step.

Notes

In[1], *can_slide_coefficient* is termed *m*, and the value of *slide_amount_coefficient* is given in tenths of a module's slant height.

References

pvlib.snow.fully_covered_nrel

pvlib.snow.**fully_covered_nrel**(*snowfall*, *threshold_snowfall=1.0*)

Calculates the timesteps when the row's slant height is fully covered by snow.

Parameters

- **snowfall** (*Series*) – Accumulated snowfall in each time period [cm]
- **threshold_snowfall** (*float, default 1.0*) – Hourly snowfall above which snow coverage is set to the row's slant height. [cm/hr]

Returns **boolean** (*Series*) – True where the snowfall exceeds the defined threshold to fully cover the panel.

Notes

Implements the model described in[1] with minor improvements in[2].

References

pvlib.snow.dc_loss_nrel

pvlib.snow.**dc_loss_nrel**(*snow_coverage*, *num_strings*)

Calculates the fraction of DC capacity lost due to snow coverage.

DC capacity loss assumes that if a string is partially covered by snow, the string's capacity is lost; see[1], Eq. 11.8.

Module orientation is accounted for by specifying the number of cell strings in parallel along the slant height. For example, a typical 60-cell module has 3 parallel strings, each comprising 20 cells in series, with the cells arranged in 6 columns of 10 cells each. For a row consisting of single modules, if the module is mounted in portrait orientation, i.e., the row slant height is along a column of 10 cells, there is 1 string in parallel along the row slant height, so *num_strings=1*. In contrast, if the module is mounted in landscape orientation with the row slant height comprising 6 cells, there are 3 parallel strings along the row slant height, so *num_strings=3*.

[1] Marion, B.; Schaefer, R.; Caine, H.; Sanchez, G. (2013). "Measured and modeled photovoltaic system energy losses from snow for Colorado and Wisconsin locations." Solar Energy 97; pp.112-121.

[2] Ryberg, D; Freeman, J. "Integration, Validation, and Application of a PV Snow Coverage Model in SAM" (2017) NREL Technical Report NREL/TP-6A20-68705

[1] Gilman, P. et al., (2018). "SAM Photovoltaic Model Technical Reference Update", NREL Technical Report NREL/TP-6A20-67399. Available at https://www.nrel.gov/docs/fy18osti/67399.pdf

Parameters

- **snow_coverage** (*numeric*) – The fraction of row slant height covered by snow at each time step.
- **num_strings** (*int*) – The number of parallel-connected strings along a row slant height.

Returns **loss** (*numeric*) – fraction of DC capacity loss due to snow coverage at each time step.

References

soiling.hsu(rainfall, cleaning_threshold, ...)	Calculates soiling ratio given particulate and rain data using the model from Humboldt State University (HSU).
soiling.kimber(rainfall[, ...])	Calculates fraction of energy lost due to soiling given rainfall data and daily loss rate using the Kimber model.

pvlib.soiling.hsu

pvlib.soiling.**hsu**(*rainfall*, *cleaning_threshold*, *tilt*, *pm2_5*, *pm10*, *depo_veloc=None*, *rain_accum_period=Timedelta('0 days 01:00:00')*)

Calculates soiling ratio given particulate and rain data using the model from Humboldt State University (HSU).

The HSU soiling model[1] returns the soiling ratio, a value between zero and one which is equivalent to (1 - transmission loss). Therefore a soiling ratio of 1.0 is equivalent to zero transmission loss.

Parameters

- **rainfall** (*Series*) – Rain accumulated in each time period. [mm]
- **cleaning_threshold** (*float*) – Amount of rain in an accumulation period needed to clean the PV modules. [mm]
- **tilt** (*float*) – Tilt of the PV panels from horizontal. [degree]
- **pm2_5** (*numeric*) – Concentration of airborne particulate matter (PM) with aerodynamic diameter less than 2.5 microns. [g/m^3]
- **pm10** (*numeric*) – Concentration of airborne particulate matter (PM) with aerodynamicdiameter less than 10 microns. [g/m^3]
- **depo_veloc** (*dict, default {'2_5': 0.0009, '10': 0.004}*) – Deposition or settling velocity of particulates. [m/s]
- **rain_accum_period** (*Timedelta, default 1 hour*) – Period for accumulating rainfall to check against *cleaning_threshold* It is recommended that *rain_accum_period* be between 1 hour and 24 hours.

Returns **soiling_ratio** (*Series*) – Values between 0 and 1. Equal to 1 - transmission loss.

References

[1] M. Coello and L. Boyle, "Simple Model For Predicting Time Series Soiling of Photovoltaic Panels," in IEEE Journal of Photovoltaics. doi: 10.1109/JPHOTOV.2019.2919628

pvlib.soiling.kimber

pvlib.soiling.**kimber**(*rainfall*, *cleaning_threshold=6*, *soiling_loss_rate=0.0015*, *grace_period=14*, *max_soiling=0.3*, *manual_wash_dates=None*, *initial_soiling=0*, *rain_accum_period=24*)

Calculates fraction of energy lost due to soiling given rainfall data and daily loss rate using the Kimber model.

Kimber soiling model[1] assumes soiling builds up at a daily rate unless the daily rainfall is greater than a threshold. The model also assumes that if daily rainfall has exceeded the threshold within a grace period, then the ground is too damp to cause soiling build-up. The model also assumes there is a maximum soiling build-up. Scheduled manual washes and rain events are assumed to reset soiling to zero.

Parameters

- **rainfall** (*pandas.Series*) – Accumulated rainfall at the end of each time period. [mm]
- **cleaning_threshold** (*float, default 6*) – Amount of daily rainfall required to clean the panels. [mm]
- **soiling_loss_rate** (*float, default 0.0015*) – Fraction of energy lost due to one day of soiling. [unitless]
- **grace_period** (*int, default 14*) – Number of days after a rainfall event when it's assumed the ground is damp, and so it's assumed there is no soiling. [days]
- **max_soiling** (*float, default 0.3*) – Maximum fraction of energy lost due to soiling. Soiling will build up until this value. [unitless]
- **manual_wash_dates** (*sequence or None, default None*) – List or tuple of dates as Python `datetime.date` when the panels were washed manually. Note there is no grace period after a manual wash, so soiling begins to build up immediately.
- **initial_soiling** (*float, default 0*) – Initial fraction of energy lost due to soiling at time zero in the *rainfall* series input. [unitless]
- **rain_accum_period** (*int, default 24*) – Period for accumulating rainfall to check against *cleaning_threshold*. The Kimber model defines this period as one day. [hours]

Returns *pandas.Series* – fraction of energy lost due to soiling, has same intervals as input

Notes

The soiling loss rate depends on both the geographical region and the soiling environment type. Rates measured by Kimber[1] are summarized in the following table:

Region/Environment	Rural	Suburban	Urban/Highway/Airport
Central Valley	0.0011	0.0019	0.0020
Northern CA	0.0011	0.0010	0.0016
Southern CA	0	0.0016	0.0019
Desert	0.0030	0.0030	0.0030

Rainfall thresholds and grace periods may also vary by region. Please consult[1] for more information.

[1] "The Effect of Soiling on Large Grid-Connected Photovoltaic Systems in California and the Southwest Region of the United States," Adrianne Kimber, et al., IEEE 4th World Conference on Photovoltaic Energy Conference, 2006, DOI: 10.1109/WCPEC.2006.279690

References

3.12.8 Tracking

SingleAxisTracker

The *SingleAxisTracker* inherits from *PVSystem*.

tracking.SingleAxisTracker([axis_tilt, ...])	Inherits the PV modeling methods from *PVSystem*.
tracking.SingleAxisTracker.singleaxis(...)	Get tracking data.
tracking.SingleAxisTracker.get_irradiance(...)	Uses the *irradiance.get_total_irradiance()* function to calculate the plane of array irradiance components on a tilted surface defined by the input data and `self.albedo`.
tracking.SingleAxisTracker.localize([...])	Creates a `LocalizedSingleAxisTracker` object using this object and location data.
tracking.LocalizedSingleAxisTracker([...])	The LocalizedSingleAxisTracker class defines a standard set of installed PV system attributes and modeling functions.

pvlib.tracking.SingleAxisTracker.singleaxis

`SingleAxisTracker.`**`singleaxis`**(*apparent_zenith*, *apparent_azimuth*)

Get tracking data. See *pvlib.tracking.singleaxis()* more detail.

Parameters

- **apparent_zenith** (*float, 1d array, or Series*) – Solar apparent zenith angles in decimal degrees.
- **apparent_azimuth** (*float, 1d array, or Series*) – Solar apparent azimuth angles in decimal degrees.

Returns *tracking data*

pvlib.tracking.SingleAxisTracker.localize

`SingleAxisTracker.`**`localize`**(*location=None*, *latitude=None*, *longitude=None*, ***kwargs*)

Creates a *LocalizedSingleAxisTracker* object using this object and location data. Must supply either location object or latitude, longitude, and any location kwargs

Parameters

- **location** (*None or Location, default None*) –
- **latitude** (*None or float, default None*) –
- **longitude** (*None or float, default None*) –
- ****kwargs** (*see Location*) –

Returns **localized_system** (*LocalizedSingleAxisTracker*)

Functions

tracking.singleaxis(apparent_zenith, ...[, ...])	Determine the rotation angle of a single axis tracker when given a particular sun zenith and azimuth angle.

pvlib.tracking.singleaxis

pvlib.tracking.**singleaxis**(*apparent_zenith*, *apparent_azimuth*, *axis_tilt=0*, *axis_azimuth=0*, *max_angle=90*, *backtrack=True*, *gcr=0.2857142857142857*)

Determine the rotation angle of a single axis tracker when given a particular sun zenith and azimuth angle. See[1] for details about the equations. Backtracking may be specified, and if so, a ground coverage ratio is required.

Rotation angle is determined in a panel-oriented coordinate system. The tracker azimuth axis_azimuth defines the positive y-axis; the positive x-axis is 90 degress clockwise from the y-axis and parallel to the earth surface, and the positive z-axis is normal and oriented towards the sun. Rotation angle tracker_theta indicates tracker position relative to horizontal: tracker_theta = 0 is horizontal, and positive tracker_theta is a clockwise rotation around the y axis in the x, y, z coordinate system. For example, if tracker azimuth axis_azimuth is 180 (oriented south), tracker_theta = 30 is a rotation of 30 degrees towards the west, and tracker_theta = -90 is a rotation to the vertical plane facing east.

Parameters

- **apparent_zenith** (*float, 1d array, or Series*) – Solar apparent zenith angles in decimal degrees.
- **apparent_azimuth** (*float, 1d array, or Series*) – Solar apparent azimuth angles in decimal degrees.
- **axis_tilt** (*float, default 0*) – The tilt of the axis of rotation (i.e, the y-axis defined by axis_azimuth) with respect to horizontal, in decimal degrees.
- **axis_azimuth** (*float, default 0*) – A value denoting the compass direction along which the axis of rotation lies. Measured in decimal degrees East of North.
- **max_angle** (*float, default 90*) – A value denoting the maximum rotation angle, in decimal degrees, of the one-axis tracker from its horizontal position (horizontal if axis_tilt = 0). A max_angle of 90 degrees allows the tracker to rotate to a vertical position to point the panel towards a horizon. max_angle of 180 degrees allows for full rotation.
- **backtrack** (*bool, default True*) – Controls whether the tracker has the capability to "backtrack" to avoid row-to-row shading. False denotes no backtrack capability. True denotes backtrack capability.
- **gcr** (*float, default 2.0/7.0*) – A value denoting the ground coverage ratio of a tracker system which utilizes backtracking; i.e. the ratio between the PV array surface area to total ground area. A tracker system with modules 2 meters wide, centered on the tracking axis, with 6 meters between the tracking axes has a gcr of 2/6=0.333. If gcr is not provided, a gcr of 2/7 is default. gcr must be <=1.

Returns

dict or DataFrame with the following columns –

- *tracker_theta*: The rotation angle of the tracker. tracker_theta = 0 is horizontal, and positive rotation angles are clockwise.
- *aoi*: The angle-of-incidence of direct irradiance onto the rotated panel surface.

[1] Lorenzo, E et al., 2011, "Tracking and back-tracking", Prog. in Photovoltaics: Research and Applications, v. 19, pp. 747-753.

- *surface_tilt*: The angle between the panel surface and the earth surface, accounting for panel rotation.
- *surface_azimuth*: The azimuth of the rotated panel, determined by projecting the vector normal to the panel's surface to the earth's surface.

References

3.12.9 IO Tools

Functions for reading and writing data from a variety of file formats relevant to solar energy modeling.

iotools.read_tmy2(filename)	Read a TMY2 file in to a DataFrame.
iotools.read_tmy3([filename, coerce_year, ...])	Read a TMY3 file in to a pandas dataframe.
iotools.read_epw(filename[, coerce_year])	Read an EPW file in to a pandas dataframe.
iotools.parse_epw(csvdata[, coerce_year])	Given a file-like buffer with data in Energy Plus Weather (EPW) format, parse the data into a dataframe.
iotools.read_srml(filename)	Read University of Oregon SRML 1min .tsv file into pandas dataframe.
iotools.read_srml_month_from_solardat(...)	Request a month of SRML data from solardat and read it into a Dataframe.
iotools.read_surfrad(filename[, map_variables])	Read in a daily NOAA SURFRAD file.
iotools.read_midc(filename[, variable_map, ...])	Read in National Renewable Energy Laboratory Measurement and Instrumentation Data Center weather data.
iotools.read_midc_raw_data_from_nrel(site, ...)	Request and read MIDC data directly from the raw data api.
iotools.read_ecmwf_macc(filename, latitude, ...)	Read data from ECMWF MACC reanalysis netCDF4 file.
iotools.get_ecmwf_macc(filename, params, ...)	Download data from ECMWF MACC Reanalysis API.
iotools.read_crn(filename)	Read a NOAA USCRN fixed-width file into pandas dataframe.
iotools.read_solrad(filename)	Read NOAA SOLRAD fixed-width file into pandas dataframe.
iotools.get_psm3(latitude, longitude, ...[, ...])	Retrieve NSRDB PSM3 timeseries weather data from the PSM3 API.
iotools.read_psm3(filename)	Read an NSRDB PSM3 weather file (formatted as SAM CSV).
iotools.parse_psm3(fbuf)	Parse an NSRDB PSM3 weather file (formatted as SAM CSV).
iotools.get_pvgis_tmy(lat, lon[, ...])	Get TMY data from PVGIS.
iotools.read_pvgis_tmy(filename[, pvgis_format])	Read a file downloaded from PVGIS.

pvlib.iotools.read_tmy2

pvlib.iotools.**read_tmy2**(*filename*)

Read a TMY2 file in to a DataFrame.

Note that values contained in the DataFrame are unchanged from the TMY2 file (i.e. units are retained). Time/Date and location data imported from the TMY2 file have been modified to a "friendlier" form con-

forming to modern conventions (e.g. N latitude is postive, E longitude is positive, the "24th" hour of any day is technically the "0th" hour of the next day). In the case of any discrepencies between this documentation and the TMY2 User's Manual[1], the TMY2 User's Manual takes precedence.

Parameters **filename** (*None or string*) – If None, attempts to use a Tkinter file browser. A string can be a relative file path, absolute file path, or url.

Returns

- *Tuple of the form (data, metadata).*
- **data** (*DataFrame*) – A dataframe with the columns described in the table below. For a more detailed descriptions of each component, please consult the TMY2 User's Manual ([1]), especially tables 3-1 through 3-6, and Appendix B.
- **metadata** (*dict*) – The site metadata available in the file.

Notes

The returned structures have the following fields.

key	description
WBAN	Site identifier code (WBAN number)
City	Station name
State	Station state 2 letter designator
TZ	Hours from Greenwich
latitude	Latitude in decimal degrees
longitude	Longitude in decimal degrees
altitude	Site elevation in meters

TMYData field	description
index	Pandas timeseries object containing timestamps
year	
month	
day	
hour	
ETR	Extraterrestrial horizontal radiation recv'd during 60 minutes prior to timestamp, Wh/m^2
ETRN	Extraterrestrial normal radiation recv'd during 60 minutes prior to timestamp, Wh/m^2
GHI	Direct and diffuse horizontal radiation recv'd during 60 minutes prior to timestamp, Wh/m^2
GHISource	See[1], Table 3-3
GHIUncertainty	See[1], Table 3-4
DNI	Amount of direct normal radiation (modeled) recv'd during 60 mintues prior to timestamp, Wh/m^2
DNISource	See[1], Table 3-3
DNIUncertainty	See[1], Table 3-4
DHI	Amount of diffuse horizontal radiation recv'd during 60 minutes prior to timestamp, Wh/m^2
DHISource	See[1], Table 3-3
DHIUncertainty	See[1], Table 3-4
GHillum	Avg. total horizontal illuminance recv'd during the 60 minutes prior to timestamp, units of 100 lux (e.g. v
GHillumSource	See[1], Table 3-3
GHillumUncertainty	See[1], Table 3-4
DNillum	Avg. direct normal illuminance recv'd during the 60 minutes prior to timestamp, units of 100 lux

[1] Marion, W and Urban, K. "Wilcox, S and Marion, W. "User's Manual for TMY2s". NREL 1995.

Table 37 – continued from previous page

TMYData field	description
DNillumSource	See[1], Table 3-3
DNillumUncertainty	See[1], Table 3-4
DHillum	Avg. horizontal diffuse illuminance recv'd during the 60 minutes prior to timestamp, units of 100 lux
DHillumSource	See[1], Table 3-3
DHillumUncertainty	See[1], Table 3-4
Zenithlum	Avg. luminance at the sky's zenith during the 60 minutes prior to timestamp, units of 10 Cd/m^2 (e.g. val
ZenithlumSource	See[1], Table 3-3
ZenithlumUncertainty	See[1], Table 3-4
TotCld	Amount of sky dome covered by clouds or obscuring phenonema at time stamp, tenths of sky
TotCldSource	See[1], Table 3-5, 8760x1 cell array of strings
TotCldUncertainty	See[1], Table 3-6
OpqCld	Amount of sky dome covered by clouds or obscuring phenonema that prevent observing the sky at time st
OpqCldSource	See[1], Table 3-5, 8760x1 cell array of strings
OpqCldUncertainty	See[1], Table 3-6
DryBulb	Dry bulb temperature at the time indicated, in tenths of degree C (e.g. 352 = 35.2 C).
DryBulbSource	See[1], Table 3-5, 8760x1 cell array of strings
DryBulbUncertainty	See[1], Table 3-6
DewPoint	Dew-point temperature at the time indicated, in tenths of degree C (e.g. 76 = 7.6 C).
DewPointSource	See[1], Table 3-5, 8760x1 cell array of strings
DewPointUncertainty	See[1], Table 3-6
RHum	Relative humidity at the time indicated, percent
RHumSource	See[1], Table 3-5, 8760x1 cell array of strings
RHumUncertainty	See[1], Table 3-6
Pressure	Station pressure at the time indicated, 1 mbar
PressureSource	See[1], Table 3-5, 8760x1 cell array of strings
PressureUncertainty	See[1], Table 3-6
Wdir	Wind direction at time indicated, degrees from east of north (360 = 0 = north; 90 = East; 0 = undefined,cal
WdirSource	See[1], Table 3-5, 8760x1 cell array of strings
WdirUncertainty	See[1], Table 3-6
Wspd	Wind speed at the time indicated, in tenths of meters/second (e.g. 212 = 21.2 m/s)
WspdSource	See[1], Table 3-5, 8760x1 cell array of strings
WspdUncertainty	See[1], Table 3-6
Hvis	Distance to discernable remote objects at time indicated (7777=unlimited, 9999=missing data), in tenths o
HvisSource	See[1], Table 3-5, 8760x1 cell array of strings
HvisUncertainty	See[1], Table 3-6
CeilHgt	Height of cloud base above local terrain (7777=unlimited, 88888=cirroform, 99999=missing data), in met
CeilHgtSource	See[1], Table 3-5, 8760x1 cell array of strings
CeilHgtUncertainty	See[1], Table 3-6
Pwat	Total precipitable water contained in a column of unit cross section from Earth to top of atmosphere, in m
PwatSource	See[1], Table 3-5, 8760x1 cell array of strings
PwatUncertainty	See[1], Table 3-6
AOD	The broadband aerosol optical depth (broadband turbidity) in thousandths on the day indicated (e.g. 114 =
AODSource	See[1], Table 3-5, 8760x1 cell array of strings
AODUncertainty	See[1], Table 3-6
SnowDepth	Snow depth in centimeters on the day indicated, (999 = missing data).
SnowDepthSource	See[1], Table 3-5, 8760x1 cell array of strings
SnowDepthUncertainty	See[1], Table 3-6
LastSnowfall	Number of days since last snowfall (maximum value of 88, where 88 = 88 or greater days; 99 = missing d
LastSnowfallSource	See[1], Table 3-5, 8760x1 cell array of strings

Table 37 – continued from previous page

TMYData field	description
LastSnowfallUncertainty	See[1], Table 3-6
PresentWeather	See[1], Appendix B, an 8760x1 cell array of strings. Each string contains 10 numeric values. The string can

References

pvlib.iotools.read_tmy3

pvlib.iotools.**read_tmy3**(*filename=None*, *coerce_year=None*, *recolumn=True*)

Read a TMY3 file in to a pandas dataframe.

Note that values contained in the metadata dictionary are unchanged from the TMY3 file (i.e. units are retained). In the case of any discrepencies between this documentation and the TMY3 User's Manual[1], the TMY3 User's Manual takes precedence.

The TMY3 files were updated in Jan. 2015. This function requires the use of the updated files.

Parameters

- **filename** (*None or string, default None*) – If None, attempts to use a Tkinter file browser. A string can be a relative file path, absolute file path, or url.
- **coerce_year** (*None or int, default None*) – If supplied, the year of the index will be set to *coerce_year*, except for the last index value which will be set to the *next* year so that the index increases monotonically.
- **recolumn** (*bool, default True*) – If `True`, apply standard names to TMY3 columns. Typically this results in stripping the units from the column name.

Returns

- *Tuple of the form (data, metadata).*
- **data** (*DataFrame*) – A pandas dataframe with the columns described in the table below. For more detailed descriptions of each component, please consult the TMY3 User's Manual ([1]), especially tables 1-1 through 1-6.
- **metadata** (*dict*) – The site metadata available in the file.

Notes

The returned structures have the following fields.

key	format	description
altitude	Float	site elevation
latitude	Float	site latitudeitude
longitude	Float	site longitudeitude
Name	String	site name
State	String	state
TZ	Float	UTC offset
USAF	Int	USAF identifier

[1] Wilcox, S and Marion, W. "Users Manual for TMY3 Data Sets". NREL/TP-581-43156, Revised May 2008.

TMYData field	description
TMYData.Index	A pandas datetime index. NOTE, the index is currently timezone unaware, and times are set to local standard time (daylight savings is not included)
TMYData.ETR	Extraterrestrial horizontal radiation recv'd during 60 minutes prior to timestamp, Wh/m^2
TMYData.ETRN	Extraterrestrial normal radiation recv'd during 60 minutes prior to timestamp, Wh/m^2
TMYData.GHI	Direct and diffuse horizontal radiation recv'd during 60 minutes prior to timestamp, Wh/m^2
TMYData.GHISource	See[1], Table 1-4
TMYData.GHIUncertainty	Uncertainty based on random and bias error estimates see[2]
TMYData.DNI	Amount of direct normal radiation (modeled) recv'd during 60 mintues prior to timestamp, Wh/m^2
TMYData.DNISource	See[1], Table 1-4
TMYData.DNIUncertainty	Uncertainty based on random and bias error estimates see[2]
TMYData.DHI	Amount of diffuse horizontal radiation recv'd during 60 minutes prior to timestamp, Wh/m^2
TMYData.DHISource	See[1], Table 1-4
TMYData.DHIUncertainty	Uncertainty based on random and bias error estimates see[2]
TMYData.GHillum	Avg. total horizontal illuminance recv'd during the 60 minutes prior to timestamp, lx
TMYData.GHillumSource	See[1], Table 1-4
TMYData.GHillumUncertainty	Uncertainty based on random and bias error estimates see[2]
TMYData.DNillum	Avg. direct normal illuminance recv'd during the 60 minutes prior to timestamp, lx
TMYData.DNillumSource	See[1], Table 1-4
TMYData.DNillumUncertainty	Uncertainty based on random and bias error estimates see[2]
TMYData.DHillum	Avg. horizontal diffuse illuminance recv'd during the 60 minutes prior to timestamp, lx
TMYData.DHillumSource	See[1], Table 1-4
TMYData.DHillumUncertainty	Uncertainty based on random and bias error estimates see[2]
TMYData.Zenithlum	Avg. luminance at the sky's zenith during the 60 minutes prior to timestamp, cd/m^2
TMYData.ZenithlumSource	See[1], Table 1-4
TMYData.ZenithlumUncertainty	Uncertainty based on random and bias error estimates see[1] section 2.10
TMYData.TotCld	Amount of sky dome covered by clouds or obscuring phenonema at time stamp, tenths of sky
TMYData.TotCldSource	See[1], Table 1-5, 8760x1 cell array of strings
TMYData.TotCldUncertainty	See[1], Table 1-6
TMYData.OpqCld	Amount of sky dome covered by clouds or obscuring phenonema that prevent observing the sky at time stamp, tenths of sky
TMYData.OpqCldSource	See[1], Table 1-5, 8760x1 cell array of strings
TMYData.OpqCldUncertainty	See[1], Table 1-6

Continued on next page

Table 38 – continued from previous page

TMYData field	description
TMYData.DryBulb	Dry bulb temperature at the time indicated, deg C
TMYData.DryBulbSource	See[1], Table 1-5, 8760x1 cell array of strings
TMYData.DryBulbUncertainty	See[1], Table 1-6
TMYData.DewPoint	Dew-point temperature at the time indicated, deg C
TMYData.DewPointSource	See[1], Table 1-5, 8760x1 cell array of strings
TMYData.DewPointUncertainty	See[1], Table 1-6
TMYData.RHum	Relatitudeive humidity at the time indicated, percent
TMYData.RHumSource	See[1], Table 1-5, 8760x1 cell array of strings
TMYData.RHumUncertainty	See[1], Table 1-6
TMYData.Pressure	Station pressure at the time indicated, 1 mbar
TMYData.PressureSource	See[1], Table 1-5, 8760x1 cell array of strings
TMYData.PressureUncertainty	See[1], Table 1-6
TMYData.Wdir	Wind direction at time indicated, degrees from north (360 = north; 0 = undefined,calm)
TMYData.WdirSource	See[1], Table 1-5, 8760x1 cell array of strings
TMYData.WdirUncertainty	See[1], Table 1-6
TMYData.Wspd	Wind speed at the time indicated, meter/second
TMYData.WspdSource	See[1], Table 1-5, 8760x1 cell array of strings
TMYData.WspdUncertainty	See[1], Table 1-6
TMYData.Hvis	Distance to discernable remote objects at time indicated (7777=unlimited), meter
TMYData.HvisSource	See[1], Table 1-5, 8760x1 cell array of strings
TMYData.HvisUncertainty	See[1], Table 1-6
TMYData.CeilHgt	Height of cloud base above local terrain (7777=unlimited), meter
TMYData.CeilHgtSource	See[1], Table 1-5, 8760x1 cell array of strings
TMYData.CeilHgtUncertainty	See[1], Table 1-6
TMYData.Pwat	Total precipitable water contained in a column of unit cross section from earth to top of atmosphere, cm
TMYData.PwatSource	See[1], Table 1-5, 8760x1 cell array of strings
TMYData.PwatUncertainty	See[1], Table 1-6
TMYData.AOD	The broadband aerosol optical depth per unit of air mass due to extinction by aerosol component of atmosphere, unitless
TMYData.AODSource	See[1], Table 1-5, 8760x1 cell array of strings
TMYData.AODUncertainty	See[1], Table 1-6
TMYData.Alb	The ratio of reflected solar irradiance to global horizontal irradiance, unitless
TMYData.AlbSource	See[1], Table 1-5, 8760x1 cell array of strings
TMYData.AlbUncertainty	See[1], Table 1-6
TMYData.Lprecipdepth	The amount of liquid precipitation observed at indicated time for the period indicated in the liquid precipitation quantity field, millimeter
TMYData.Lprecipquantity	The period of accumulatitudeion for the liquid precipitation depth field, hour
TMYData.LprecipSource	See[1], Table 1-5, 8760x1 cell array of strings
TMYData.LprecipUncertainty	See[1], Table 1-6
TMYData.PresWth	Present weather code, see[2].
TMYData.PresWthSource	Present weather code source, see[2].

Continued on next page

Table 38 – continued from previous page

TMYData field	description
TMYData.PresWthUncertainty	Present weather code uncertainty, see[2].

Warning: TMY3 irradiance data corresponds to the *previous* hour, so the first index is 1AM, corresponding to the irradiance from midnight to 1AM, and the last index is midnight of the *next* year. For example, if the last index in the TMY3 file was 1988-12-31 24:00:00 this becomes 1989-01-01 00:00:00 after calling *read_tmy3()*.

Warning: When coercing the year, the last index in the dataframe will become midnight of the *next* year. For example, if the last index in the TMY3 was 1988-12-31 24:00:00, and year is coerced to 1990 then this becomes 1991-01-01 00:00:00.

References

pvlib.iotools.read_epw

pvlib.iotools.**read_epw**(*filename*, *coerce_year=None*)

Read an EPW file in to a pandas dataframe.

Note that values contained in the metadata dictionary are unchanged from the EPW file.

EPW files are commonly used by building simulation professionals and are widely available on the web. For example via: https://energyplus.net/weather , http://climate.onebuilding.org or http://www.ladybug.tools/epwmap/

Parameters

- **filename** (*String*) – Can be a relative file path, absolute file path, or url.
- **coerce_year** (*None or int, default None*) – If supplied, the year of the data will be set to this value. This can be a useful feature because EPW data is composed of data from different years. Warning: EPW files always have 365*24 = 8760 data rows; be careful with the use of leap years.

Returns

- **data** (*DataFrame*) – A pandas dataframe with the columns described in the table below. For more detailed descriptions of each component, please consult the EnergyPlus Auxiliary Programs documentation[1]
- **metadata** (*dict*) – The site metadata available in the file.

See also:

pvlib.iotools.parse_epw()

Notes

The returned structures have the following fields.

[2] Wilcox, S. (2007). National Solar Radiation Database 1991 2005 Update: Users Manual. 472 pp.; NREL Report No. TP-581-41364.
[1] EnergyPlus documentation, Auxiliary Programs

key	format	description
loc	String	default identifier, not used
city	String	site loccation
state-prov	String	state, province or region (if available)
country	String	site country code
data_type	String	type of original data source
WMO_code	String	WMO identifier
latitude	Float	site latitude
longitude	Float	site longitude
TZ	Float	UTC offset
altitude	Float	site elevation

EPWData field	description
index	A pandas datetime index. NOTE, times are set to local standard time (daylight savings is not included
year	Year, from original EPW file. Can be overwritten using coerce function.
month	Month, from original EPW file.
day	Day of the month, from original EPW file.
hour	Hour of the day from original EPW file. Note that EPW's convention of 1-24h is not taken over in the
minute	Minute, from original EPW file. Not used.
data_source_unct	Data source and uncertainty flags. See[1], chapter 2.13
temp_air	Dry bulb temperature at the time indicated, deg C
temp_dew	Dew-point temperature at the time indicated, deg C
relative_humidity	Relative humidity at the time indicated, percent
atmospheric_pressure	Station pressure at the time indicated, Pa
etr	Extraterrestrial horizontal radiation recv'd during 60 minutes prior to timestamp, Wh/m^2
etrn	Extraterrestrial normal radiation recv'd during 60 minutes prior to timestamp, Wh/m^2
ghi_infrared	Horizontal infrared radiation recv'd during 60 minutes prior to timestamp, Wh/m^2
ghi	Direct and diffuse horizontal radiation recv'd during 60 minutes prior to timestamp, Wh/m^2
dni	Amount of direct normal radiation (modeled) recv'd during 60 minutes prior to timestamp, Wh/m^2
dhi	Amount of diffuse horizontal radiation recv'd during 60 minutes prior to timestamp, Wh/m^2
global_hor_illum	Avg. total horizontal illuminance recv'd during the 60 minutes prior to timestamp, lx
direct_normal_illum	Avg. direct normal illuminance recv'd during the 60 minutes prior to timestamp, lx
diffuse_horizontal_illum	Avg. horizontal diffuse illuminance recv'd during the 60 minutes prior to timestamp, lx
zenith_luminance	Avg. luminance at the sky's zenith during the 60 minutes prior to timestamp, cd/m^2
wind_direction	Wind direction at time indicated, degrees from north (360 = north; 0 = undefined,calm)
wind_speed	Wind speed at the time indicated, m/s
total_sky_cover	Amount of sky dome covered by clouds or obscuring phenomena at time stamp, tenths of sky
opaque_sky_cover	Amount of sky dome covered by clouds or obscuring phenomena that prevent observing the sky at tim
visibility	Horizontal visibility at the time indicated, km
ceiling_height	Height of cloud base above local terrain (7777=unlimited), meter
present_weather_observation	Indicator for remaining fields: If 0, then the observed weather codes are taken from the following field
present_weather_codes	Present weather code, see [1], chapter 2.9.1.28
precipitable_water	Total precipitable water contained in a column of unit cross section from earth to top of atmosphere, c
aerosol_optical_depth	The broadband aerosol optical depth per unit of air mass due to extinction by aerosol component of at
snow_depth	Snow depth in centimeters on the day indicated, (999 = missing data)
days_since_last_snowfall	Number of days since last snowfall (maximum value of 88, where 88 = 88 or greater days; 99 = missi
albedo	The ratio of reflected solar irradiance to global horizontal irradiance, unitless
liquid_precipitation_depth	The amount of liquid precipitation observed at indicated time for the period indicated in the liquid pre
liquid_precipitation_quantity	The period of accumulation for the liquid precipitation depth field, hour

References

pvlib.iotools.parse_epw

pvlib.iotools.**parse_epw**(*csvdata*, *coerce_year=None*)

Given a file-like buffer with data in Energy Plus Weather (EPW) format, parse the data into a dataframe.

Parameters

- **csvdata** (*file-like buffer*) – a file-like buffer containing data in the EPW format
- **coerce_year** (*None or int, default None*) – If supplied, the year of the data will be set to this value. This can be a useful feature because EPW data is composed of data from different years. Warning: EPW files always have 365*24 = 8760 data rows; be careful with the use of leap years.

Returns

- **data** (*DataFrame*) – A pandas dataframe with the columns described in the table below. For more detailed descriptions of each component, please consult the EnergyPlus Auxiliary Programs documentation available at: https://energyplus.net/documentation.
- **metadata** (*dict*) – The site metadata available in the file.

See also:

pvlib.iotools.read_epw()

pvlib.iotools.read_srml

pvlib.iotools.**read_srml**(*filename*)

Read University of Oregon SRML 1min .tsv file into pandas dataframe. The SRML is described in[1].

Parameters **filename** (*str*) – filepath or url to read for the tsv file.

Returns **data** (*Dataframe*) – A dataframe with datetime index and all of the variables listed in the *VARIABLE_MAP* dict inside of the map_columns function, along with their associated quality control flags.

Notes

The time index is shifted back by one interval to account for the daily endtime of 2400, and to avoid time parsing errors on leap years. The returned data values are labeled by the left endpoint of interval, and should be understood to occur during the interval from the time of the row until the time of the next row. This is consistent with pandas' default labeling behavior.

See SRML's Archival Files page for more information.

References

pvlib.iotools.read_srml_month_from_solardat

pvlib.iotools.**read_srml_month_from_solardat**(*station*, *year*, *month*, *filetype='PO'*)

Request a month of SRML data from solardat and read it into a Dataframe. The SRML is described in[1].

[1] University of Oregon Solar Radiation Monitoring Laboratory http://solardat.uoregon.edu/

[1] University of Oregon Solar Radiation Measurement Laboratory http://solardat.uoregon.edu/

Parameters

- **station** (*str*) – The name of the SRML station to request.
- **year** (*int*) – Year to request data for
- **month** (*int*) – Month to request data for.
- **filetype** (*string*) – SRML file type to gather. See notes for explanation.

Returns **data** (*pd.DataFrame*) – One month of data from SRML.

Notes

File types designate the time interval of a file and if it contains raw or processed data. For instance, *RO* designates raw, one minute data and *PO* designates processed one minute data. The availability of file types varies between sites. Below is a table of file types and their time intervals. See [1] for site information.

time interval	raw filetype	processed filetype
1 minute	RO	PO
5 minute	RF	PF
15 minute	RQ	PQ
hourly	RH	PH

References

pvlib.iotools.read_surfrad

pvlib.iotools.**read_surfrad**(*filename*, *map_variables=True*)

Read in a daily NOAA SURFRAD file. The SURFRAD network is described in[1].

Parameters

- **filename** (*str*) – Filepath or url.
- **map_variables** (*bool*) – When true, renames columns of the Dataframe to pvlib variable names where applicable. See variable SURFRAD_COLUMNS.

Returns

- *Tuple of the form (data, metadata).*
- **data** (*Dataframe*) – Dataframe with the fields found below.
- **metadata** (*dict*) – Site metadata included in the file.

Notes

Metadata dictionary includes the following fields:

[1] NOAA Earth System Research Laboratory Surface Radiation Budget Network SURFRAD Homepage

Key	Format	Description
station	String	site name
latitude	Float	site latitude
longitude	Float	site longitude
elevation	Int	site elevation
surfrad_version	Int	surfrad version
tz	String	Timezone (UTC)

Dataframe includes the following fields:

raw, mapped	Format	Description
Mapped field names are returned when the map_variables argument is True		
year	int	year as 4 digit int
jday	int	day of year 1-365(or 366)
month	int	month (1-12)
day	int	day of month(1-31)
hour	int	hour (0-23)
minute	int	minute (0-59)
dt	float	decimal time i.e. 23.5 = 2330
zen, solar_zenith	float	solar zenith angle (deg)
Fields below have associated qc flags labeled <field>_flag.		
dw_solar, ghi	float	downwelling global solar(W/m^2)
uw_solar	float	updownwelling global solar(W/m^2)
direct_n, dni	float	direct normal solar (W/m^2)
diffuse, dhi	float	downwelling diffuse solar (W/m^2)
dw_ir	float	downwelling thermal infrared (W/m^2)
dw_casetemp	float	downwelling IR case temp (K)
dw_dometemp	float	downwelling IR dome temp (K)
uw_ir	float	upwelling thermal infrared (W/m^2)
uw_casetemp	float	upwelling IR case temp (K)
uw_dometemp	float	upwelling IR case temp (K)
uvb	float	global uvb (miliWatts/m^2)
par	float	photosynthetically active radiation(W/m^2)
netsolar	float	net solar (dw_solar - uw_solar) (W/m^2)
netir	float	net infrared (dw_ir - uw_ir) (W/m^2)
totalnet	float	net radiation (netsolar+netir) (W/m^2)
temp, temp_air	float	10-meter air temperature (?C)
rh, relative_humidity	float	relative humidity (%)
windspd, wind_speed	float	wind speed (m/s)
winddir, wind_direction	float	wind direction (deg, clockwise from north)
pressure	float	station pressure (mb)

See README files located in the station directories in the SURFRAD data archives[2]_ for details on SURFRAD daily data files.

References

pvlib.iotools.read_midc

pvlib.iotools.**read_midc**(*filename*, *variable_map={}*, *raw_data=False*, ***kwargs*)

Read in National Renewable Energy Laboratory Measurement and Instrumentation Data Center weather data. The MIDC is described in[1].

Parameters

- **filename** (*string or file-like object*) – Filename, url, or file-like object of data to read.
- **variable_map** (*dictionary*) – Dictionary for mapping MIDC field names to pvlib names. Used to rename the columns of the resulting DataFrame. Does not map names by default. See Notes for an example.
- **raw_data** (*boolean*) – Set to true to use format_index_raw to correctly format the date/time columns of MIDC raw data files.
- **kwargs** (*dict*) – Additional keyword arguments to pass to *pandas.read_csv*

Returns **data** (*Dataframe*) – A dataframe with DatetimeIndex localized to the provided timezone.

Notes

The *variable_map* argument should map fields from MIDC data to pvlib names.

E.g. if a MIDC file contains the variable 'Global Horizontal [W/m^2]', passing the dictionary below will rename the column to 'ghi' in the returned Dataframe.

{'Global Horizontal [W/m^2]': 'ghi'}

See the MIDC_VARIABLE_MAP for collection of mappings by site. For a full list of pvlib variable names see the Variable Style Rules.

Be sure to check the units for the variables you will use on the MIDC site.

References

pvlib.iotools.read_midc_raw_data_from_nrel

pvlib.iotools.**read_midc_raw_data_from_nrel**(*site*, *start*, *end*, *variable_map={}*, *timeout=30*)

Request and read MIDC data directly from the raw data api.

Parameters

- **site** (*string*) – The MIDC station id.
- **start** (*datetime*) – Start date for requested data.
- **end** (*datetime*) – End date for requested data.
- **variable_map** (*dict*) – A dictionary mapping MIDC field names to pvlib names. Used to rename columns of the resulting DataFrame. See Notes of *pvlib.iotools.read_midc()* for example.
- **timeout** (*float, default 30*) – Number of seconds to wait to connect/read from the API before failing.

Returns *data* – Dataframe with DatetimeIndex localized to the station location.

[1] NREL: Measurement and Instrumentation Data Center https://midcdmz.nrel.gov/

Raises

- `requests.HTTPError` – For any error in retrieving the CSV file from the MIDC API
- `requests.Timeout` – If data is not received in within `timeout` seconds

Notes

Requests spanning an instrumentation change will yield an error. See the MIDC raw data api page here for more details and considerations.

pvlib.iotools.read_ecmwf_macc

`pvlib.iotools.`**`read_ecmwf_macc`**(*filename*, *latitude*, *longitude*, *utc_time_range=None*)

Read data from ECMWF MACC reanalysis netCDF4 file.

Parameters

- **`filename`** (*`string`*) – full path to netCDF4 data file.
- **`latitude`** (*`float`*) – latitude in degrees
- **`longitude`** (*`float`*) – longitude in degrees
- **`utc_time_range`** (*`sequence of datetime.datetime`*) – pair of start and stop naive or UTC date-times

Returns **data** (*pandas.DataFrame*) – dataframe for specified range of UTC date-times

pvlib.iotools.get_ecmwf_macc

`pvlib.iotools.`**`get_ecmwf_macc`**(*filename*, *params*, *startdate*, *stopdate*, *lookup_params=True*, *server=None*, *target=<function _ecmwf>*)

Download data from ECMWF MACC Reanalysis API.

Parameters

- **`filename`** (*`str`*) – full path of file where to save data, `.nc` appended if not given
- **`params`** (*`str or sequence of str`*) – keynames of parameter[s] to download
- **`startdate`** (*`datetime.datetime or datetime.date`*) – UTC date
- **`stopdate`** (*`datetime.datetime or datetime.date`*) – UTC date
- **`lookup_params`** (*`bool, default True`*) – optional flag, if `False`, then codes are already formatted
- **`server`** (*`ecmwfapi.api.ECMWFDataServer`*) – optionally provide a server object, default is `None`
- **`target`** (*`callable`*) – optional function that calls `server.retrieve` to pass to thread

Returns **t** (*thread*) – a thread object, use it to check status by calling *t.is_alive()*

Notes

To download data from ECMWF requires the API client and a registration key. Please read the documentation in Access ECMWF Public Datasets. Follow the instructions in step 4 and save the ECMWF registration key as *$HOME/.ecmwfapirc* or set *ECMWF_API_KEY* as the path to the key.

This function returns a daemon thread that runs in the background. Exiting Python will kill this thread, however this thread will not block the main thread or other threads. This thread will terminate when the file is downloaded or if the thread raises an unhandled exception. You may submit multiple requests simultaneously to break up large downloads. You can also check the status and retrieve downloads online at http://apps.ecmwf.int/webmars/joblist/. This is useful if you kill the thread. Downloads expire after 24 hours.

Warning: Your request may be queued online for an hour or more before it begins to download

Precipitable water P_{wat} is equivalent to the total column of water vapor (TCWV), but the units given by ECMWF MACC Reanalysis are kg/m^2 at STP (1-atm, 25-C). Divide by ten to convert to centimeters of precipitable water:

$$P_{wat}\,(\text{cm}) = TCWV\left(\frac{\text{kg}}{\text{m}^2}\right)\frac{100\frac{\text{cm}}{\text{m}}}{1000\frac{\text{kg}}{\text{m}^3}}$$

The keynames available for the `params` argument are given by `pvlib.iotools.ecmwf_macc.PARAMS` which maps the keys to codes used in the API. The following keynames are available:

keyname	description
tcwv	total column water vapor in kg/m^2 at STP
aod550	aerosol optical depth measured at 550-nm
aod469	aerosol optical depth measured at 469-nm
aod670	aerosol optical depth measured at 670-nm
aod865	aerosol optical depth measured at 865-nm
aod1240	aerosol optical depth measured at 1240-nm

If `lookup_params` is `False` then `params` must contain the codes preformatted according to the ECMWF MACC Reanalysis API. This is useful if you want to retrieve codes that are not mapped in `pvlib.iotools.ecmwf_macc.PARAMS`.

Specify a custom `target` function to modify how the ECMWF API function `server.retrieve` is called. The `target` function must have the following signature in which the parameter definitions are similar to *`pvlib.iotools.get_ecmwf_macc()`*.

```
target(server, startdate, stopdate, params, filename) -> None
```

Examples

Retrieve the AOD measured at 550-nm and the total column of water vapor for November 1, 2012.

```
>>> from datetime import date
>>> from pvlib.iotools import get_ecmwf_macc
>>> filename = 'aod_tcwv_20121101.nc'  # .nc extension added if missing
>>> params = ('aod550', 'tcwv')
>>> start = end = date(2012, 11, 1)
>>> t = get_ecmwf_macc(filename, params, start, end)
>>> t.is_alive()
True
```

pvlib.iotools.read_crn

pvlib.iotools.**read_crn**(*filename*)

Read a NOAA USCRN fixed-width file into pandas dataframe. The CRN is described in[1] and[2].

Parameters **filename** (*str, path object, or file-like*) – filepath or url to read for the fixed-width file.

Returns **data** (*Dataframe*) – A dataframe with DatetimeIndex and all of the variables in the file.

Notes

CRN files contain 5 minute averages labeled by the interval ending time. Here, missing data is flagged as NaN, rather than the lowest possible integer for a field (e.g. -999 or -99). Air temperature in deg C. Wind speed in m/s at a height of 1.5 m above ground level.

Variables corresponding to standard pvlib variables are renamed, e.g. *SOLAR_RADIATION* becomes *ghi*. See the *pvlib.iotools.crn.VARIABLE_MAP* dict for the complete mapping.

References

pvlib.iotools.read_solrad

pvlib.iotools.**read_solrad**(*filename*)

Read NOAA SOLRAD fixed-width file into pandas dataframe. The SOLRAD network is described in[1] and[2].

Parameters **filename** (*str*) – filepath or url to read for the fixed-width file.

Returns **data** (*Dataframe*) – A dataframe with DatetimeIndex and all of the variables in the file.

Notes

SOLRAD data resolution is described by the README_SOLRAD.txt: "Before 1-jan. 2015 the data were reported as 3-min averages; on and after 1-Jan. 2015, SOLRAD data are reported as 1-min. averages of 1-sec. samples." Here, missing data is flagged as NaN, rather than -9999.9.

References

pvlib.iotools.get_psm3

pvlib.iotools.**get_psm3**(*latitude*, *longitude*, *api_key*, *email*, *names='tmy'*, *interval=60*, *leap_day=False*, *full_name='pvlib python'*, *affiliation='pvlib python'*, *timeout=30*)

Retrieve NSRDB PSM3 timeseries weather data from the PSM3 API. The NSRDB is described in[1] and the PSM3 API is described in[2] and[3].

[1] U.S. Climate Reference Network https://www.ncdc.noaa.gov/crn/qcdatasets.html

[2] Diamond, H. J. et. al., 2013: U.S. Climate Reference Network after one decade of operations: status and assessment. Bull. Amer. Meteor. Soc., 94, 489-498. DOI: 10.1175/BAMS-D-12-00170.1

[1] NOAA SOLRAD Network https://www.esrl.noaa.gov/gmd/grad/solrad/index.html

[2] B. B. Hicks et. al., (1996), The NOAA Integrated Surface Irradiance Study (ISIS). A New Surface Radiation Monitoring Program. Bull. Amer. Meteor. Soc., 77, 2857-2864. DOI: 10.1175/1520-0477(1996)077<2857:TNISIS>2.0.CO;2

[1] NREL National Solar Radiation Database (NSRDB)

[2] NREL Developer Network - Physical Solar Model (PSM) v3

[3] NREL Developer Network - Physical Solar Model (PSM) v3 TMY

Parameters

- **latitude** (*float or int*) – in decimal degrees, between -90 and 90, north is positive
- **longitude** (*float or int*) – in decimal degrees, between -180 and 180, east is positive
- **api_key** (*str*) – NREL Developer Network API key
- **email** (*str*) – NREL API uses this to automatically communicate messages back to the user only if necessary
- **names** (*str, default 'tmy'*) – PSM3 API parameter specifing year or TMY variant to download, see notes below for options
- **interval** (*int, default 60*) – interval size in minutes, can only be either 30 or 60. Only used for single-year requests (i.e., it is ignored for tmy/tgy/tdy requests).
- **leap_day** (*boolean, default False*) – include leap day in the results. Only used for single-year requests (i.e., it is ignored for tmy/tgy/tdy requests).
- **full_name** (*str, default 'pvlib python'*) – optional
- **affiliation** (*str, default 'pvlib python'*) – optional
- **timeout** (*int, default 30*) – time in seconds to wait for server response before timeout

Returns

- **headers** (*dict*) – metadata from NREL PSM3 about the record, see *pvlib.iotools.parse_psm3()* for fields
- **data** (*pandas.DataFrame*) – timeseries data from NREL PSM3

Raises `requests.HTTPError` – if the request response status is not ok, then the `'errors'` field from the JSON response or any error message in the content will be raised as an exception, for example if the *api_key* was rejected or if the coordinates were not found in the NSRDB

Notes

The required NREL developer key, *api_key*, is available for free by registering at the NREL Developer Network.

> **Warning:** The "DEMO_KEY" *api_key* is severely rate limited and may result in rejected requests.

The PSM3 API *names* parameter must be a single value from the following list:

```
['1998', '1999', '2000', '2001', '2002', '2003', '2004', '2005',
 '2006', '2007', '2008', '2009', '2010', '2011', '2012', '2013',
 '2014', '2015', '2016', '2017', '2018', 'tmy', 'tmy-2016', 'tmy-2017',
 'tdy-2017', 'tgy-2017', 'tmy-2018', 'tdy-2018', 'tgy-2018']
```

> **Warning:** PSM3 is limited to data found in the NSRDB, please consult the references below for locations with available data

See also:

pvlib.iotools.read_psm3(), *pvlib.iotools.parse_psm3()*

References

pvlib.iotools.read_psm3

pvlib.iotools.**read_psm3**(*filename*)

Read an NSRDB PSM3 weather file (formatted as SAM CSV). The NSRDB is described in[1] and the SAM CSV format is described in[2].

Parameters **filename** (*str*) – Filename of a file containing data to read.

Returns

- **headers** (*dict*) – metadata from NREL PSM3 about the record, see *pvlib.iotools.parse_psm3()* for fields
- **data** (*pandas.DataFrame*) – timeseries data from NREL PSM3

See also:

pvlib.iotools.parse_psm3(), *pvlib.iotools.get_psm3()*

References

pvlib.iotools.parse_psm3

pvlib.iotools.**parse_psm3**(*fbuf*)

Parse an NSRDB PSM3 weather file (formatted as SAM CSV). The NSRDB is described in[1] and the SAM CSV format is described in[2].

Parameters **fbuf** (*file-like object*) – File-like object containing data to read.

Returns

- **headers** (*dict*) – metadata from NREL PSM3 about the record, see notes for fields
- **data** (*pandas.DataFrame*) – timeseries data from NREL PSM3

Notes

The return is a tuple with two items. The first item is a header with metadata from NREL PSM3 about the record containing the following fields:

- Source
- Location ID
- City
- State
- Country

[1] NREL National Solar Radiation Database (NSRDB)
[2] Standard Time Series Data File Format
[1] NREL National Solar Radiation Database (NSRDB)
[2] Standard Time Series Data File Format

- Latitude
- Longitude
- Time Zone
- Elevation
- Local Time Zone
- Clearsky DHI Units
- Clearsky DNI Units
- Clearsky GHI Units
- Dew Point Units
- DHI Units
- DNI Units
- GHI Units
- Solar Zenith Angle Units
- Temperature Units
- Pressure Units
- Relative Humidity Units
- Precipitable Water Units
- Wind Direction Units
- Wind Speed
- Cloud Type -15
- Cloud Type 0
- Cloud Type 1
- Cloud Type 2
- Cloud Type 3
- Cloud Type 4
- Cloud Type 5
- Cloud Type 6
- Cloud Type 7
- Cloud Type 8
- Cloud Type 9
- Cloud Type 10
- Cloud Type 11
- Cloud Type 12
- Fill Flag 0
- Fill Flag 1
- Fill Flag 2

- Fill Flag 3
- Fill Flag 4
- Fill Flag 5
- Surface Albedo Units
- Version

The second item is a dataframe with the PSM3 timeseries data.

Examples

```
>>> # Read a local PSM3 file:
>>> with open(filename, 'r') as f:  # doctest: +SKIP
...     metadata, df = iotools.parse_psm3(f)  # doctest: +SKIP
```

See also:

pvlib.iotools.read_psm3(), *pvlib.iotools.get_psm3()*

References

pvlib.iotools.get_pvgis_tmy

pvlib.iotools.**get_pvgis_tmy**(*lat*, *lon*, *outputformat='json'*, *usehorizon=True*, *userhorizon=None*, *startyear=None*, *endyear=None*, *url='https://re.jrc.ec.europa.eu/api/'*, *timeout=30*)

Get TMY data from PVGIS. For more information see the PVGIS[1] TMY tool documentation[2].

Parameters

- **lat** (*float*) – Latitude in degrees north
- **lon** (*float*) – Longitude in dgrees east
- **outputformat** (*str, default 'json'*) – Must be in `['csv', 'basic', 'epw', 'json']`. See PVGIS TMY tool documentation[2] for more info.
- **usehorizon** (*bool, default True*) – include effects of horizon
- **userhorizon** (*list of float, default None*) – optional user specified elevation of horizon in degrees, at equally spaced azimuth clockwise from north, only valid if *usehorizon* is true, if *usehorizon* is true but *userhorizon* is *None* then PVGIS will calculate the horizon[3]
- **startyear** (*int, default None*) – first year to calculate TMY
- **endyear** (*int, default None*) – last year to calculate TMY, must be at least 10 years from first year
- **url** (str, default `pvlib.iotools.pvgis.URL`) – base url of PVGIS API, append `tmy` to get TMY endpoint
- **timeout** (*int, default 30*) – time in seconds to wait for server response before timeout

[1] PVGIS
[2] PVGIS TMY tool
[3] PVGIS horizon profile tool

Returns

- **data** (*pandas.DataFrame*) – the weather data
- **months_selected** (*list*) – TMY year for each month, `None` for basic and EPW
- **inputs** (*dict*) – the inputs, `None` for basic and EPW
- **meta** (*list or dict*) – meta data, `None` for basic

Raises `requests.HTTPError` – if the request response status is `HTTP/1.1 400 BAD REQUEST`, then the error message in the response will be raised as an exception, otherwise raise whatever `HTTP/1.1` error occurred

See also:

read_pvgis_tmy()

References

pvlib.iotools.read_pvgis_tmy

pvlib.iotools.**read_pvgis_tmy**(*filename*, *pvgis_format=None*)

Read a file downloaded from PVGIS.

Parameters

- **filename** (*str, pathlib.Path, or file-like buffer*) – Name, path, or buffer of file downloaded from PVGIS.
- **pvgis_format** (*str, default None*) – Format of PVGIS file or buffer. Equivalent to the `outputformat` parameter in the PVGIS TMY API. If *filename* is a file and *pvgis_format* is `None` then the file extension will be used to determine the PVGIS format to parse. For PVGIS files from the API with `outputformat='basic'`, please set *pvgis_format* to `'basic'`. If *filename* is a buffer, then *pvgis_format* is required and must be in `['csv', 'epw', 'json', 'basic']`.

Returns

- **data** (*pandas.DataFrame*) – the weather data
- **months_selected** (*list*) – TMY year for each month, `None` for basic and EPW
- **inputs** (*dict*) – the inputs, `None` for basic and EPW
- **meta** (*list or dict*) – meta data, `None` for basic

Raises

- `ValueError` – if *pvgis_format* is `None` and the file extension is neither `.csv`, `.json`, nor `.epw`, or if *pvgis_format* is provided as input but isn't in `['csv', 'epw', 'json', 'basic']`
- `TypeError` – if *pvgis_format* is `None` and *filename* is a buffer

See also:

get_pvgis_tmy()

A *Location* object may be created from metadata in some files.

location.Location.from_tmy(tmy_metadata[, ...])	Create an object based on a metadata dictionary from tmy2 or tmy3 data readers.
location.Location.from_epw(metadata[, data])	Create a Location object based on a metadata dictionary from epw data readers.

pvlib.location.Location.from_tmy

classmethod Location.**from_tmy**(*tmy_metadata*, *tmy_data=None*, ***kwargs*)

Create an object based on a metadata dictionary from tmy2 or tmy3 data readers.

Parameters

- **tmy_metadata** (*dict*) – Returned from tmy.readtmy2 or tmy.readtmy3
- **tmy_data** (*None or DataFrame, default None*) – Optionally attach the TMY data to this object.

Returns *Location*

pvlib.location.Location.from_epw

classmethod Location.**from_epw**(*metadata*, *data=None*, ***kwargs*)

Create a Location object based on a metadata dictionary from epw data readers.

Parameters

- **metadata** (*dict*) – Returned from epw.read_epw
- **data** (*None or DataFrame, default None*) – Optionally attach the epw data to this object.

Returns

- *Location object (or the child class of Location that you*
- *called this method from).*

3.12.10 Forecasting

Forecast models

forecast.GFS([resolution, set_type])	Subclass of the ForecastModel class representing GFS forecast model.
forecast.NAM([set_type])	Subclass of the ForecastModel class representing NAM forecast model.
forecast.RAP([resolution, set_type])	Subclass of the ForecastModel class representing RAP forecast model.
forecast.HRRR([set_type])	Subclass of the ForecastModel class representing HRRR forecast model.
forecast.HRRR_ESRL([set_type])	Subclass of the ForecastModel class representing NOAA/GSD/ESRL's HRRR forecast model.
forecast.NDFD([set_type])	Subclass of the ForecastModel class representing NDFD forecast model.

pvlib.forecast.GFS

class pvlib.forecast.**GFS**(*resolution='half'*, *set_type='best'*)

Subclass of the ForecastModel class representing GFS forecast model.

Model data corresponds to 0.25 degree resolution forecasts.

Parameters

- **resolution** (*string, default 'half'*) – Resolution of the model, either 'half' or 'quarter' degree.
- **set_type** (*string, default 'best'*) – Type of model to pull data from.

dataframe_variables

Common variables present in the final set of data.

Type list

model

Name of the UNIDATA forecast model.

Type string

model_type

UNIDATA category in which the model is located.

Type string

variables

Defines the variables to obtain from the weather model and how they should be renamed to common variable names.

Type dict

units

Dictionary containing the units of the standard variables and the model specific variables.

Type dict

__init__(*resolution='half'*, *set_type='best'*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([resolution, set_type])	Initialize self.
cloud_cover_to_ghi_linear(cloud_cover, ghi_clear)	Convert cloud cover to GHI using a linear relationship.
cloud_cover_to_irradiance(cloud_cover[, how])	Convert cloud cover to irradiance.
cloud_cover_to_irradiance_clearsky_scaling(...)	Estimates irradiance from cloud cover in the following steps:
cloud_cover_to_irradiance_liujordan(...)	Estimates irradiance from cloud cover in the following steps:
cloud_cover_to_transmittance_linear(cloud_cover)	Convert cloud cover to atmospheric transmittance using a linear model.
connect_to_catalog()	

Continued on next page

Table 43 – continued from previous page

get_data(latitude, longitude, start, end[, ...])	Submits a query to the UNIDATA servers using Siphon NCSS and converts the netcdf data to a pandas DataFrame.
get_processed_data(*args, **kwargs)	Get and process forecast data.
gust_to_speed(data[, scaling])	Computes standard wind speed from gust.
isobaric_to_ambient_temperature(data)	Calculates temperature from isobaric temperature.
kelvin_to_celsius(temperature)	Converts Kelvin to celsius.
process_data(data[, cloud_cover])	Defines the steps needed to convert raw forecast data into processed forecast data.
rename(data[, variables])	Renames the columns according the variable mapping.
set_dataset()	Retrieves the designated dataset, creates NCSS object, and creates a NCSS query object.
set_location(tz, latitude, longitude)	Sets the location for the query.
set_query_latlon()	Sets the NCSS query location latitude and longitude.
set_query_time_range(start, end)	**param start** Must be tz-localized.
set_time(time)	Converts time data into a pandas date object.
uv_to_speed(data)	Computes wind speed from wind components.

Attributes

access_url_key
base_tds_url
catalog_url
data_format
units

pvlib.forecast.NAM

class pvlib.forecast.**NAM**(*set_type='best'*)

Subclass of the ForecastModel class representing NAM forecast model.

Model data corresponds to NAM CONUS 12km resolution forecasts from CONDUIT.

Parameters **set_type** (*string, default 'best'*) – Type of model to pull data from.

dataframe_variables

Common variables present in the final set of data.

Type list

model

Name of the UNIDATA forecast model.

Type string

model_type

UNIDATA category in which the model is located.

Type string

variables

Defines the variables to obtain from the weather model and how they should be renamed to common

variable names.

Type dict

units
Dictionary containing the units of the standard variables and the model specific variables.

Type dict

__init__(*set_type='best'*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([set_type])	Initialize self.
cloud_cover_to_ghi_linear(cloud_cover, ghi_clear)	Convert cloud cover to GHI using a linear relationship.
cloud_cover_to_irradiance(cloud_cover[, how])	Convert cloud cover to irradiance.
cloud_cover_to_irradiance_clearsky_scaling(...)	Estimates irradiance from cloud cover in the following steps:
cloud_cover_to_irradiance_liujordan(...)	Estimates irradiance from cloud cover in the following steps:
cloud_cover_to_transmittance_linear(cloud_cover)	Convert cloud cover to atmospheric transmittance using a linear model.
connect_to_catalog()	
get_data(latitude, longitude, start, end[, ...])	Submits a query to the UNIDATA servers using Siphon NCSS and converts the netcdf data to a pandas DataFrame.
get_processed_data(*args, **kwargs)	Get and process forecast data.
gust_to_speed(data[, scaling])	Computes standard wind speed from gust.
isobaric_to_ambient_temperature(data)	Calculates temperature from isobaric temperature.
kelvin_to_celsius(temperature)	Converts Kelvin to celsius.
process_data(data[, cloud_cover])	Defines the steps needed to convert raw forecast data into processed forecast data.
rename(data[, variables])	Renames the columns according the variable mapping.
set_dataset()	Retrieves the designated dataset, creates NCSS object, and creates a NCSS query object.
set_location(tz, latitude, longitude)	Sets the location for the query.
set_query_latlon()	Sets the NCSS query location latitude and longitude.
set_query_time_range(start, end)	**param start** Must be tz-localized.
set_time(time)	Converts time data into a pandas date object.
uv_to_speed(data)	Computes wind speed from wind components.

Attributes

access_url_key
base_tds_url
catalog_url

Continued on next page

Table 46 – continued from previous page

data_format
units

pvlib.forecast.RAP

class pvlib.forecast.**RAP**(*resolution='20'*, *set_type='best'*)

Subclass of the ForecastModel class representing RAP forecast model.

Model data corresponds to Rapid Refresh CONUS 20km resolution forecasts.

Parameters

- **resolution** (*string or int, default '20'*) – The model resolution, either '20' or '40' (km)
- **set_type** (*string, default 'best'*) – Type of model to pull data from.

dataframe_variables
Common variables present in the final set of data.

Type list

model
Name of the UNIDATA forecast model.

Type string

model_type
UNIDATA category in which the model is located.

Type string

variables
Defines the variables to obtain from the weather model and how they should be renamed to common variable names.

Type dict

units
Dictionary containing the units of the standard variables and the model specific variables.

Type dict

__init__(*resolution='20'*, *set_type='best'*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([resolution, set_type])	Initialize self.
cloud_cover_to_ghi_linear(cloud_cover, ghi_clear)	Convert cloud cover to GHI using a linear relationship.
cloud_cover_to_irradiance(cloud_cover[, how])	Convert cloud cover to irradiance.
cloud_cover_to_irradiance_clearsky_scaling(...)	Estimates irradiance from cloud cover in the following steps:
cloud_cover_to_irradiance_liujordan(...)	Estimates irradiance from cloud cover in the following steps:

Continued on next page

Table 47 – continued from previous page

cloud_cover_to_transmittance_linear(cloud_cover)	Convert cloud cover to atmospheric transmittance using a linear model.
connect_to_catalog()	
get_data(latitude, longitude, start, end[, ...])	Submits a query to the UNIDATA servers using Siphon NCSS and converts the netcdf data to a pandas DataFrame.
get_processed_data(*args, **kwargs)	Get and process forecast data.
gust_to_speed(data[, scaling])	Computes standard wind speed from gust.
isobaric_to_ambient_temperature(data)	Calculates temperature from isobaric temperature.
kelvin_to_celsius(temperature)	Converts Kelvin to celsius.
process_data(data[, cloud_cover])	Defines the steps needed to convert raw forecast data into processed forecast data.
rename(data[, variables])	Renames the columns according the variable mapping.
set_dataset()	Retrieves the designated dataset, creates NCSS object, and creates a NCSS query object.
set_location(tz, latitude, longitude)	Sets the location for the query.
set_query_latlon()	Sets the NCSS query location latitude and longitude.
set_query_time_range(start, end)	**param start** Must be tz-localized.
set_time(time)	Converts time data into a pandas date object.
uv_to_speed(data)	Computes wind speed from wind components.

Attributes

access_url_key
base_tds_url
catalog_url
data_format
units

pvlib.forecast.HRRR

class pvlib.forecast.**HRRR**(*set_type='best'*)

Subclass of the ForecastModel class representing HRRR forecast model.

Model data corresponds to NCEP HRRR CONUS 2.5km resolution forecasts.

Parameters **set_type** (*string, default 'best'*) – Type of model to pull data from.

dataframe_variables

Common variables present in the final set of data.

Type list

model

Name of the UNIDATA forecast model.

Type string

model_type

UNIDATA category in which the model is located.

Type string

variables
Defines the variables to obtain from the weather model and how they should be renamed to common variable names.

Type dict

units
Dictionary containing the units of the standard variables and the model specific variables.

Type dict

__init__(*set_type='best'*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([set_type])	Initialize self.
cloud_cover_to_ghi_linear(cloud_cover, ghi_clear)	Convert cloud cover to GHI using a linear relationship.
cloud_cover_to_irradiance(cloud_cover[, how])	Convert cloud cover to irradiance.
cloud_cover_to_irradiance_clearsky_scaling(...)	Estimates irradiance from cloud cover in the following steps:
cloud_cover_to_irradiance_liujordan(...)	Estimates irradiance from cloud cover in the following steps:
cloud_cover_to_transmittance_linear(cloud_cover)	Convert cloud cover to atmospheric transmittance using a linear model.
connect_to_catalog()	
get_data(latitude, longitude, start, end[, ...])	Submits a query to the UNIDATA servers using Siphon NCSS and converts the netcdf data to a pandas DataFrame.
get_processed_data(*args, **kwargs)	Get and process forecast data.
gust_to_speed(data[, scaling])	Computes standard wind speed from gust.
isobaric_to_ambient_temperature(data)	Calculates temperature from isobaric temperature.
kelvin_to_celsius(temperature)	Converts Kelvin to celsius.
process_data(data[, cloud_cover])	Defines the steps needed to convert raw forecast data into processed forecast data.
rename(data[, variables])	Renames the columns according the variable mapping.
set_dataset()	Retrieves the designated dataset, creates NCSS object, and creates a NCSS query object.
set_location(tz, latitude, longitude)	Sets the location for the query.
set_query_latlon()	Sets the NCSS query location latitude and longitude.
set_query_time_range(start, end)	**param start** Must be tz-localized.
set_time(time)	Converts time data into a pandas date object.
uv_to_speed(data)	Computes wind speed from wind components.

Attributes

access_url_key
base_tds_url
catalog_url
data_format
units

pvlib.forecast.HRRR_ESRL

class pvlib.forecast.**HRRR_ESRL**(*set_type='best'*)

Subclass of the ForecastModel class representing NOAA/GSD/ESRL's HRRR forecast model. This is not an operational product.

Model data corresponds to NOAA/GSD/ESRL HRRR CONUS 3km resolution surface forecasts.

Parameters **set_type** (*string, default 'best'*) – Type of model to pull data from.

dataframe_variables
Common variables present in the final set of data.

Type list

model
Name of the UNIDATA forecast model.

Type string

model_type
UNIDATA category in which the model is located.

Type string

variables
Defines the variables to obtain from the weather model and how they should be renamed to common variable names.

Type dict

units
Dictionary containing the units of the standard variables and the model specific variables.

Type dict

__init__(*set_type='best'*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([set_type])	Initialize self.
cloud_cover_to_ghi_linear(cloud_cover, ghi_clear)	Convert cloud cover to GHI using a linear relationship.
cloud_cover_to_irradiance(cloud_cover[, how])	Convert cloud cover to irradiance.
cloud_cover_to_irradiance_clearsky_scaling(...)	Estimates irradiance from cloud cover in the following steps:
cloud_cover_to_irradiance_liujordan(...)	Estimates irradiance from cloud cover in the following steps:

Continued on next page

Table 51 – continued from previous page

cloud_cover_to_transmittance_linear(cloud_cover)	Convert cloud cover to atmospheric transmittance using a linear model.
connect_to_catalog()	
get_data(latitude, longitude, start, end[, ...])	Submits a query to the UNIDATA servers using Siphon NCSS and converts the netcdf data to a pandas DataFrame.
get_processed_data(*args, **kwargs)	Get and process forecast data.
gust_to_speed(data[, scaling])	Computes standard wind speed from gust.
isobaric_to_ambient_temperature(data)	Calculates temperature from isobaric temperature.
kelvin_to_celsius(temperature)	Converts Kelvin to celsius.
process_data(data[, cloud_cover])	Defines the steps needed to convert raw forecast data into processed forecast data.
rename(data[, variables])	Renames the columns according the variable mapping.
set_dataset()	Retrieves the designated dataset, creates NCSS object, and creates a NCSS query object.
set_location(tz, latitude, longitude)	Sets the location for the query.
set_query_latlon()	Sets the NCSS query location latitude and longitude.
set_query_time_range(start, end)	**param start** Must be tz-localized.
set_time(time)	Converts time data into a pandas date object.
uv_to_speed(data)	Computes wind speed from wind components.

Attributes

access_url_key
base_tds_url
catalog_url
data_format
units

pvlib.forecast.NDFD

class pvlib.forecast.**NDFD**(*set_type='best'*)

Subclass of the ForecastModel class representing NDFD forecast model.

Model data corresponds to NWS CONUS CONDUIT forecasts.

Parameters **set_type** (*string, default 'best'*) – Type of model to pull data from.

dataframe_variables

Common variables present in the final set of data.

Type list

model

Name of the UNIDATA forecast model.

Type string

model_type

UNIDATA category in which the model is located.

Type string

variables
Defines the variables to obtain from the weather model and how they should be renamed to common variable names.

Type dict

units
Dictionary containing the units of the standard variables and the model specific variables.

Type dict

__init__(*set_type='best'*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([set_type])	Initialize self.
cloud_cover_to_ghi_linear(cloud_cover, ghi_clear)	Convert cloud cover to GHI using a linear relationship.
cloud_cover_to_irradiance(cloud_cover[, how])	Convert cloud cover to irradiance.
cloud_cover_to_irradiance_clearsky_scaling(...)	Estimates irradiance from cloud cover in the following steps:
cloud_cover_to_irradiance_liujordan(...)	Estimates irradiance from cloud cover in the following steps:
cloud_cover_to_transmittance_linear(cloud_cover)	Convert cloud cover to atmospheric transmittance using a linear model.
connect_to_catalog()	
get_data(latitude, longitude, start, end[, ...])	Submits a query to the UNIDATA servers using Siphon NCSS and converts the netcdf data to a pandas DataFrame.
get_processed_data(*args, **kwargs)	Get and process forecast data.
gust_to_speed(data[, scaling])	Computes standard wind speed from gust.
isobaric_to_ambient_temperature(data)	Calculates temperature from isobaric temperature.
kelvin_to_celsius(temperature)	Converts Kelvin to celsius.
process_data(data, **kwargs)	Defines the steps needed to convert raw forecast data into processed forecast data.
rename(data[, variables])	Renames the columns according the variable mapping.
set_dataset()	Retrieves the designated dataset, creates NCSS object, and creates a NCSS query object.
set_location(tz, latitude, longitude)	Sets the location for the query.
set_query_latlon()	Sets the NCSS query location latitude and longitude.
set_query_time_range(start, end)	**param start** Must be tz-localized.
set_time(time)	Converts time data into a pandas date object.
uv_to_speed(data)	Computes wind speed from wind components.

Attributes

access_url_key
base_tds_url
catalog_url
data_format
units

Getting data

forecast.ForecastModel.get_data(latitude, ...)	Submits a query to the UNIDATA servers using Siphon NCSS and converts the netcdf data to a pandas DataFrame.
forecast.ForecastModel.get_processed_data(...)	Get and process forecast data.

pvlib.forecast.ForecastModel.get_data

ForecastModel.**get_data**(*latitude*, *longitude*, *start*, *end*, *vert_level=None*, *query_variables=None*, *close_netcdf_data=True*, ***kwargs*)

Submits a query to the UNIDATA servers using Siphon NCSS and converts the netcdf data to a pandas DataFrame.

Parameters

- **latitude** (*float*) – The latitude value.
- **longitude** (*float*) – The longitude value.
- **start** (*datetime or timestamp*) – The start time.
- **end** (*datetime or timestamp*) – The end time.
- **vert_level** (*None, float or integer, default None*) – Vertical altitude of interest.
- **query_variables** (*None or list, default None*) – If None, uses self.variables.
- **close_netcdf_data** (*bool, default True*) – Controls if the temporary netcdf data file should be closed. Set to False to access the raw data.
- ****kwargs** – Additional keyword arguments are silently ignored.

Returns **forecast_data** (*DataFrame*) – column names are the weather model's variable names.

pvlib.forecast.ForecastModel.get_processed_data

ForecastModel.**get_processed_data**(**args*, ***kwargs*)

Get and process forecast data.

Parameters

- ***args** (*positional arguments*) – Passed to get_data
- ****kwargs** (*keyword arguments*) – Passed to get_data and process_data

Returns **data** (*DataFrame*) – Processed forecast data

Processing data

forecast.ForecastModel.process_data(data, ...)	Defines the steps needed to convert raw forecast data into processed forecast data.
forecast.ForecastModel.rename(data[, variables])	Renames the columns according the variable mapping.
forecast.ForecastModel.cloud_cover_to_ghi_linear(...)	Convert cloud cover to GHI using a linear relationship.
forecast.ForecastModel.cloud_cover_to_irradiance_clearsky_scaling(...)	Estimates irradiance from cloud cover in the following steps:
forecast.ForecastModel.cloud_cover_to_transmittance_linear(...)	Convert cloud cover to atmospheric transmittance using a linear model.
forecast.ForecastModel.cloud_cover_to_irradiance_liujordan(...)	Estimates irradiance from cloud cover in the following steps:
forecast.ForecastModel.cloud_cover_to_irradiance(...)	Convert cloud cover to irradiance.
forecast.ForecastModel.kelvin_to_celsius(...)	Converts Kelvin to celsius.
forecast.ForecastModel.isobaric_to_ambient_temperature(data)	Calculates temperature from isobaric temperature.
forecast.ForecastModel.uv_to_speed(data)	Computes wind speed from wind components.
forecast.ForecastModel.gust_to_speed(data[, ...])	Computes standard wind speed from gust.

pvlib.forecast.ForecastModel.process_data

ForecastModel.**process_data**(*data*, ***kwargs*)

Defines the steps needed to convert raw forecast data into processed forecast data. Most forecast models implement their own version of this method which also call this one.

Parameters **data** (*DataFrame*) – Raw forecast data

Returns **data** (*DataFrame*) – Processed forecast data.

pvlib.forecast.ForecastModel.rename

ForecastModel.**rename**(*data*, *variables=None*)

Renames the columns according the variable mapping.

Parameters

- **data** (*DataFrame*) –
- **variables** (*None or dict, default None*) – If None, uses self.variables

Returns **data** (*DataFrame*) – Renamed data.

pvlib.forecast.ForecastModel.cloud_cover_to_ghi_linear

ForecastModel.**cloud_cover_to_ghi_linear**(*cloud_cover*, *ghi_clear*, *offset=35*, ***kwargs*)

Convert cloud cover to GHI using a linear relationship.

0% cloud cover returns ghi_clear.

100% cloud cover returns offset*ghi_clear.

Parameters

- **cloud_cover** (*numeric*) – Cloud cover in %.
- **ghi_clear** (*numeric*) – GHI under clear sky conditions.
- **offset** (*numeric, default 35*) – Determines the minimum GHI.
- **kwargs** – Not used.

Returns **ghi** (*numeric*) – Estimated GHI.

References

Larson et. al. "Day-ahead forecasting of solar power output from photovoltaic plants in the American Southwest" Renewable Energy 91, 11-20 (2016).

pvlib.forecast.ForecastModel.cloud_cover_to_irradiance_clearsky_scaling

ForecastModel.**cloud_cover_to_irradiance_clearsky_scaling**(*cloud_cover*, *method='linear'*, ***kwargs*)

Estimates irradiance from cloud cover in the following steps:

1. Determine clear sky GHI using Ineichen model and climatological turbidity.
2. Estimate cloudy sky GHI using a function of cloud_cover e.g. *cloud_cover_to_ghi_linear()*
3. Estimate cloudy sky DNI using the DISC model.
4. Calculate DHI from DNI and GHI.

Parameters

- **cloud_cover** (*Series*) – Cloud cover in %.
- **method** (*str, default 'linear'*) – Method for converting cloud cover to GHI. 'linear' is currently the only option.
- ****kwargs** – Passed to the method that does the conversion

Returns **irrads** (*DataFrame*) – Estimated GHI, DNI, and DHI.

pvlib.forecast.ForecastModel.cloud_cover_to_transmittance_linear

ForecastModel.**cloud_cover_to_transmittance_linear**(*cloud_cover*, *offset=0.75*, ***kwargs*)

Convert cloud cover to atmospheric transmittance using a linear model.

0% cloud cover returns offset.

100% cloud cover returns 0.

Parameters

- **cloud_cover** (*numeric*) – Cloud cover in %.
- **offset** (*numeric, default 0.75*) – Determines the maximum transmittance.
- **kwargs** – Not used.

Returns **ghi** (*numeric*) – Estimated GHI.

pvlib.forecast.ForecastModel.cloud_cover_to_irradiance_liujordan

ForecastModel.**cloud_cover_to_irradiance_liujordan**(*cloud_cover*, ***kwargs*)

Estimates irradiance from cloud cover in the following steps:

1. Determine transmittance using a function of cloud cover e.g. *cloud_cover_to_transmittance_linear()*
2. Calculate GHI, DNI, DHI using the *pvlib.irradiance.liujordan()* model

Parameters **cloud_cover** (*Series*) –

Returns **irradiance** (*DataFrame*) – Columns include ghi, dni, dhi

pvlib.forecast.ForecastModel.cloud_cover_to_irradiance

ForecastModel.**cloud_cover_to_irradiance**(*cloud_cover*, *how='clearsky_scaling'*, ***kwargs*)

Convert cloud cover to irradiance. A wrapper method.

Parameters

- **cloud_cover** (*Series*) –
- **how** (*str, default 'clearsky_scaling'*) – Selects the method for conversion. Can be one of clearsky_scaling or liujordan.
- ****kwargs** – Passed to the selected method.

Returns **irradiance** (*DataFrame*) – Columns include ghi, dni, dhi

pvlib.forecast.ForecastModel.kelvin_to_celsius

ForecastModel.**kelvin_to_celsius**(*temperature*)

Converts Kelvin to celsius.

Parameters **temperature** (*numeric*) –

Returns **temperature** (*numeric*)

pvlib.forecast.ForecastModel.isobaric_to_ambient_temperature

ForecastModel.**isobaric_to_ambient_temperature**(*data*)

Calculates temperature from isobaric temperature.

Parameters **data** (*DataFrame*) – Must contain columns pressure, temperature_iso, temperature_dew_iso. Input temperature in K.

Returns **temperature** (*Series*) – Temperature in K

pvlib.forecast.ForecastModel.uv_to_speed

ForecastModel.**uv_to_speed**(*data*)
Computes wind speed from wind components.

Parameters **data** (*DataFrame*) – Must contain the columns 'wind_speed_u' and 'wind_speed_v'.

Returns **wind_speed** (*Series*)

pvlib.forecast.ForecastModel.gust_to_speed

ForecastModel.**gust_to_speed**(*data*, *scaling=0.7142857142857143*)
Computes standard wind speed from gust. Very approximate and location dependent.

Parameters **data** (*DataFrame*) – Must contain the column 'wind_speed_gust'.

Returns **wind_speed** (*Series*)

IO support

These are public for now, but use at your own risk.

forecast.ForecastModel.set_dataset()	Retrieves the designated dataset, creates NCSS object, and creates a NCSS query object.
forecast.ForecastModel.set_query_latlon()	Sets the NCSS query location latitude and longitude.
forecast.ForecastModel.set_location(tz, ...)	Sets the location for the query.
forecast.ForecastModel.set_time(time)	Converts time data into a pandas date object.

pvlib.forecast.ForecastModel.set_dataset

ForecastModel.**set_dataset**()
Retrieves the designated dataset, creates NCSS object, and creates a NCSS query object.

pvlib.forecast.ForecastModel.set_query_latlon

ForecastModel.**set_query_latlon**()
Sets the NCSS query location latitude and longitude.

pvlib.forecast.ForecastModel.set_location

ForecastModel.**set_location**(*tz*, *latitude*, *longitude*)
Sets the location for the query.

Parameters

- **tz** (*tzinfo*) – Timezone of the query
- **latitude** (*float*) – Latitude of the query

- **longitude** (*float*) – Longitude of the query

Notes

Assigns `self.location`.

pvlib.forecast.ForecastModel.set_time

`ForecastModel.`**`set_time`**(*time*)

Converts time data into a pandas date object.

Parameters **time** (*netcdf*) – Contains time information.

Returns *pandas.DatetimeIndex*

3.12.11 ModelChain

Creating a ModelChain object.

modelchain.ModelChain(system, location[, ...])	The ModelChain class to provides a standardized, high-level interface for all of the modeling steps necessary for calculating PV power from a time series of weather inputs.

Running

Running a ModelChain.

modelchain.ModelChain.run_model(weather[, times])	Run the model.
modelchain.ModelChain.complete_irradiance(weather)	Determine the missing irradiation columns.
modelchain.ModelChain.prepare_inputs(weather)	Prepare the solar position, irradiance, and weather inputs to the model.

pvlib.modelchain.ModelChain.run_model

`ModelChain.`**`run_model`**(*weather*, *times=None*)

Run the model.

Parameters

- **weather** (*DataFrame*) – Column names must be `'dni'`, `'ghi'`, `'dhi'`, `'wind_speed'`, `'temp_air'`. All irradiance components are required. Air temperature of 20 C and wind speed of 0 m/s will be added to the DataFrame if not provided.
- **times** (*None, deprecated*) – Deprecated argument included for API compatibility, but not used internally. The index of the weather DataFrame is used for times.

Returns

- *self*

- **Assigns attributes** (`solar_position`, `airmass`, `irradiance`,)
- `total_irrad`, `effective_irradiance`, `weather`,
- `cell_temperature`, `aoi`, `aoi_modifier`, `spectral_modifier`,
- `dc`, `ac`, `losses`,
- `diode_params` (if dc_model is a single diode model)

pvlib.modelchain.ModelChain.complete_irradiance

`ModelChain.`**`complete_irradiance`**(*weather*, *times=None*)

Determine the missing irradiation columns. Only two of the following data columns (dni, ghi, dhi) are needed to calculate the missing data.

This function is not safe at the moment. Results can be too high or negative. Please contribute and help to improve this function on https://github.com/pvlib/pvlib-python

Parameters

- **`weather`** (*DataFrame*) – Column names must be `'dni'`, `'ghi'`, `'dhi'`, `'wind_speed'`, `'temp_air'`. All irradiance components are required. Air temperature of 20 C and wind speed of 0 m/s will be added to the DataFrame if not provided.
- **`times`** (*None, deprecated*) – Deprecated argument included for API compatibility, but not used internally. The index of the weather DataFrame is used for times.

Returns *self*

Notes

Assigns attributes: `weather`

Examples

This example does not work until the parameters *my_system*, *my_location*, *my_datetime* and *my_weather* are not defined properly but shows the basic idea how this method can be used.

```
>>> from pvlib.modelchain import ModelChain
```

```
>>> # my_weather containing 'dhi' and 'ghi'.
>>> mc = ModelChain(my_system, my_location)  # doctest: +SKIP
>>> mc.complete_irradiance(my_weather)  # doctest: +SKIP
>>> mc.run_model(mc.weather)  # doctest: +SKIP
```

```
>>> # my_weather containing 'dhi', 'ghi' and 'dni'.
>>> mc = ModelChain(my_system, my_location)  # doctest: +SKIP
>>> mc.run_model(my_weather)  # doctest: +SKIP
```

pvlib.modelchain.ModelChain.prepare_inputs

`ModelChain.`**`prepare_inputs`**(*weather*, *times=None*)

Prepare the solar position, irradiance, and weather inputs to the model.

Parameters

- **weather** (*DataFrame*) – Column names must be `'dni'`, `'ghi'`, `'dhi'`, `'wind_speed'`, `'temp_air'`. All irradiance components are required. Air temperature of 20 C and wind speed of 0 m/s will be added to the DataFrame if not provided.
- **times** (*None, deprecated*) – Deprecated argument included for API compatibility, but not used internally. The index of the weather DataFrame is used for times.

Notes

Assigns attributes: `solar_position`, `airmass`, `total_irrad`, `aoi`

See also:

ModelChain.complete_irradiance()

Attributes

Simple ModelChain attributes:

`system`, `location`, `clearsky_model`, `transposition_model`, `solar_position_method`, `airmass_model`

Properties

ModelChain properties that are aliases for your specific modeling functions.

modelchain.ModelChain.orientation_strategy
modelchain.ModelChain.dc_model
modelchain.ModelChain.ac_model
modelchain.ModelChain.aoi_model
modelchain.ModelChain.spectral_model
modelchain.ModelChain.temperature_model
modelchain.ModelChain.losses_model
modelchain.ModelChain.effective_irradiance_model()

pvlib.modelchain.ModelChain.orientation_strategy

ModelChain.**orientation_strategy**

pvlib.modelchain.ModelChain.dc_model

ModelChain.**dc_model**

pvlib.modelchain.ModelChain.ac_model

ModelChain.**ac_model**

pvlib.modelchain.ModelChain.aoi_model

ModelChain.**aoi_model**

pvlib.modelchain.ModelChain.spectral_model

ModelChain.**spectral_model**

pvlib.modelchain.ModelChain.temperature_model

ModelChain.**temperature_model**

pvlib.modelchain.ModelChain.losses_model

ModelChain.**losses_model**

pvlib.modelchain.ModelChain.effective_irradiance_model

ModelChain.**effective_irradiance_model**()

Model definitions

ModelChain model definitions.

modelchain.ModelChain.sapm()
modelchain.ModelChain.cec()
modelchain.ModelChain.desoto()
modelchain.ModelChain.pvsyst()
modelchain.ModelChain.pvwatts_dc()
modelchain.ModelChain.snlinverter()
modelchain.ModelChain.adrinverter()
modelchain.ModelChain.pvwatts_inverter()
modelchain.ModelChain.ashrae_aoi_loss()
modelchain.ModelChain.physical_aoi_loss()
modelchain.ModelChain.sapm_aoi_loss()
modelchain.ModelChain.no_aoi_loss()
modelchain.ModelChain.first_solar_spectral_loss()
modelchain.ModelChain.sapm_spectral_loss()
modelchain.ModelChain.no_spectral_loss()
modelchain.ModelChain.sapm_temp()
modelchain.ModelChain.pvsyst_temp()

Continued on next page

Table 61 – continued from previous page

modelchain.ModelChain.faiman_temp()
modelchain.ModelChain.pvwatts_losses()
modelchain.ModelChain.no_extra_losses()

pvlib.modelchain.ModelChain.sapm

ModelChain.**sapm**()

pvlib.modelchain.ModelChain.cec

ModelChain.**cec**()

pvlib.modelchain.ModelChain.desoto

ModelChain.**desoto**()

pvlib.modelchain.ModelChain.pvsyst

ModelChain.**pvsyst**()

pvlib.modelchain.ModelChain.pvwatts_dc

ModelChain.**pvwatts_dc**()

pvlib.modelchain.ModelChain.snlinverter

ModelChain.**snlinverter**()

pvlib.modelchain.ModelChain.adrinverter

ModelChain.**adrinverter**()

pvlib.modelchain.ModelChain.pvwatts_inverter

ModelChain.**pvwatts_inverter**()

pvlib.modelchain.ModelChain.ashrae_aoi_loss

ModelChain.**ashrae_aoi_loss**()

pvlib.modelchain.ModelChain.physical_aoi_loss

ModelChain.**physical_aoi_loss**()

pvlib.modelchain.ModelChain.sapm_aoi_loss

ModelChain.**sapm_aoi_loss**()

pvlib.modelchain.ModelChain.no_aoi_loss

ModelChain.**no_aoi_loss**()

pvlib.modelchain.ModelChain.first_solar_spectral_loss

ModelChain.**first_solar_spectral_loss**()

pvlib.modelchain.ModelChain.sapm_spectral_loss

ModelChain.**sapm_spectral_loss**()

pvlib.modelchain.ModelChain.no_spectral_loss

ModelChain.**no_spectral_loss**()

pvlib.modelchain.ModelChain.sapm_temp

ModelChain.**sapm_temp**()

pvlib.modelchain.ModelChain.pvsyst_temp

ModelChain.**pvsyst_temp**()

pvlib.modelchain.ModelChain.faiman_temp

ModelChain.**faiman_temp**()

pvlib.modelchain.ModelChain.pvwatts_losses

ModelChain.**pvwatts_losses**()

pvlib.modelchain.ModelChain.no_extra_losses

ModelChain.**no_extra_losses**()

Inference methods

Methods that automatically determine which models should be used based on the information in the associated *PVSystem* object.

modelchain.ModelChain.infer_dc_model()
modelchain.ModelChain.infer_ac_model()
modelchain.ModelChain.infer_aoi_model()
modelchain.ModelChain.infer_spectral_model()
modelchain.ModelChain.infer_temperature_model()
modelchain.ModelChain.infer_losses_model()

pvlib.modelchain.ModelChain.infer_dc_model

ModelChain.**infer_dc_model**()

pvlib.modelchain.ModelChain.infer_ac_model

ModelChain.**infer_ac_model**()

pvlib.modelchain.ModelChain.infer_aoi_model

ModelChain.**infer_aoi_model**()

pvlib.modelchain.ModelChain.infer_spectral_model

ModelChain.**infer_spectral_model**()

pvlib.modelchain.ModelChain.infer_temperature_model

ModelChain.**infer_temperature_model**()

pvlib.modelchain.ModelChain.infer_losses_model

ModelChain.**infer_losses_model**()

Functions

Functions for power modeling.

modelchain.basic_chain(times, latitude, ...)	An experimental function that computes all of the modeling steps necessary for calculating power or energy for a PV system at a given location.
modelchain.get_orientation(strategy, **kwargs)	Determine a PV system's surface tilt and surface azimuth using a named strategy.

pvlib.modelchain.basic_chain

pvlib.modelchain.**basic_chain**(*times*, *latitude*, *longitude*, *module_parameters*, *temperature_model_parameters*, *inverter_parameters*, *irradiance=None*, *weather=None*, *surface_tilt=None*, *surface_azimuth=None*, *orientation_strategy=None*, *transposition_model='haydavies'*, *solar_position_method='nrel_numpy'*, *airmass_model='kastenyoung1989'*, *altitude=None*, *pressure=None*, ***kwargs*)

An experimental function that computes all of the modeling steps necessary for calculating power or energy for a PV system at a given location.

Parameters

- **times** (*DatetimeIndex*) – Times at which to evaluate the model.
- **latitude** (*float.*) – Positive is north of the equator. Use decimal degrees notation.
- **longitude** (*float.*) – Positive is east of the prime meridian. Use decimal degrees notation.
- **module_parameters** (*None, dict or Series*) – Module parameters as defined by the SAPM. See pvsystem.sapm for details.
- **temperature_model_parameters** (*None, dict or Series.*) – Temperature model parameters as defined by the SAPM. See temperature.sapm_cell for details.
- **inverter_parameters** (*None, dict or Series*) – Inverter parameters as defined by the CEC. See pvsystem.snlinverter for details.
- **irradiance** (*None or DataFrame, default None*) – If None, calculates clear sky data. Columns must be 'dni', 'ghi', 'dhi'.
- **weather** (*None or DataFrame, default None*) – If None, assumes air temperature is 20 C and wind speed is 0 m/s. Columns must be 'wind_speed', 'temp_air'.
- **surface_tilt** (*None, float or Series, default None*) – Surface tilt angles in decimal degrees. The tilt angle is defined as degrees from horizontal (e.g. surface facing up = 0, surface facing horizon = 90)
- **surface_azimuth** (*None, float or Series, default None*) – Surface azimuth angles in decimal degrees. The azimuth convention is defined as degrees east of north (North=0, South=180, East=90, West=270).
- **orientation_strategy** (*None or str, default None*) – The strategy for aligning the modules. If not None, sets the `surface_azimuth` and `surface_tilt` properties of the `system`. Allowed strategies include 'flat', 'south_at_latitude_tilt'. Ignored for SingleAxisTracker systems.
- **transposition_model** (*str, default 'haydavies'*) – Passed to system.get_irradiance.

- **solar_position_method** (*str, default 'nrel_numpy'*) – Passed to solarposition.get_solarposition.
- **airmass_model** (*str, default 'kastenyoung1989'*) – Passed to atmosphere.relativeairmass.
- **altitude** (*None or float, default None*) – If None, computed from pressure. Assumed to be 0 m if pressure is also None.
- **pressure** (*None or float, default None*) – If None, computed from altitude. Assumed to be 101325 Pa if altitude is also None.
- ****kwargs** – Arbitrary keyword arguments. See code for details.

Returns **output** (*(dc, ac)*) – Tuple of DC power (with SAPM parameters) (DataFrame) and AC power (Series).

pvlib.modelchain.get_orientation

pvlib.modelchain.**get_orientation**(*strategy*, ***kwargs*)

Determine a PV system's surface tilt and surface azimuth using a named strategy.

Parameters

- **strategy** (*str*) – The orientation strategy. Allowed strategies include 'flat', 'south_at_latitude_tilt'.
- ****kwargs** – Strategy-dependent keyword arguments. See code for details.

Returns *surface_tilt, surface_azimuth*

3.12.12 Bifacial

Methods for calculating back surface irradiance

bifacial.pvfactors_timeseries(solar_azimuth, ...)	Calculate front and back surface plane-of-array irradiance on a fixed tilt or single-axis tracker PV array configuration, and using the open-source "pvfactors" package.

pvlib.bifacial.pvfactors_timeseries

pvlib.bifacial.**pvfactors_timeseries**(*solar_azimuth*, *solar_zenith*, *surface_azimuth*, *surface_tilt*, *axis_azimuth*, *timestamps*, *dni*, *dhi*, *gcr*, *pvrow_height*, *pvrow_width*, *albedo*, *n_pvrows=3*, *index_observed_pvrow=1*, *rho_front_pvrow=0.03*, *rho_back_pvrow=0.05*, *horizon_band_angle=15.0*, *run_parallel_calculations=True*, *n_workers_for_parallel_calcs=2*)

Calculate front and back surface plane-of-array irradiance on a fixed tilt or single-axis tracker PV array configuration, and using the open-source "pvfactors" package. pvfactors implements the model described in[1]. Please refer to pvfactors online documentation for more details: https://sunpower.github.io/pvfactors/

Parameters

[1] Anoma, Marc Abou, et al. "View Factor Model and Validation for Bifacial PV and Diffuse Shade on Single-Axis Trackers." 44th IEEE Photovoltaic Specialist Conference. 2017.

- **solar_azimuth** (*numeric*) – Sun's azimuth angles using pvlib's azimuth convention (deg)
- **solar_zenith** (*numeric*) – Sun's zenith angles (deg)
- **surface_azimuth** (*numeric*) – Azimuth angle of the front surface of the PV modules, using pvlib's convention (deg)
- **surface_tilt** (*numeric*) – Tilt angle of the PV modules, going from 0 to 180 (deg)
- **axis_azimuth** (*float*) – Azimuth angle of the rotation axis of the PV modules, using pvlib's convention (deg). This is supposed to be fixed for all timestamps.
- **timestamps** (*datetime or DatetimeIndex*) – List of simulation timestamps
- **dni** (*numeric*) – Direct normal irradiance (W/m2)
- **dhi** (*numeric*) – Diffuse horizontal irradiance (W/m2)
- **gcr** (*float*) – Ground coverage ratio of the pv array
- **pvrow_height** (*float*) – Height of the pv rows, measured at their center (m)
- **pvrow_width** (*float*) – Width of the pv rows in the considered 2D plane (m)
- **albedo** (*float*) – Ground albedo
- **n_pvrows** (*int, default 3*) – Number of PV rows to consider in the PV array
- **index_observed_pvrow** (*int, default 1*) – Index of the PV row whose incident irradiance will be returned. Indices of PV rows go from 0 to n_pvrows-1.
- **rho_front_pvrow** (*float, default 0.03*) – Front surface reflectivity of PV rows
- **rho_back_pvrow** (*float, default 0.05*) – Back surface reflectivity of PV rows
- **horizon_band_angle** (*float, default 15*) – Elevation angle of the sky dome's diffuse horizon band (deg)
- **run_parallel_calculations** (*bool, default True*) – pvfactors is capable of using multiprocessing. Use this flag to decide to run calculations in parallel (recommended) or not.
- **n_workers_for_parallel_calcs** (*int, default 2*) – Number of workers to use in the case of parallel calculations. The '-1' value will lead to using a value equal to the number of CPU's on the machine running the model.

Returns

- **front_poa_irradiance** (*numeric*) – Calculated incident irradiance on the front surface of the PV modules (W/m2)
- **back_poa_irradiance** (*numeric*) – Calculated incident irradiance on the back surface of the PV modules (W/m2)
- **df_registries** (*pandas DataFrame*) – DataFrame containing detailed outputs of the simulation; for instance the shapely geometries, the irradiance components incident on all surfaces of the PV array (for all timestamps), etc. In the pvfactors documentation, this is refered to as the "surface registry".

References

3.12.13 Scaling

Methods for manipulating irradiance for temporal or spatial considerations

scaling.wvm(clearsky_index, positions, ...)	Compute spatial aggregation time series smoothing on clear sky index based on the Wavelet Variability model of Lave et al [1-2].

pvlib.scaling.wvm

pvlib.scaling.**wvm**(*clearsky_index*, *positions*, *cloud_speed*, *dt=None*)

Compute spatial aggregation time series smoothing on clear sky index based on the Wavelet Variability model of Lave et al [1-2]. Implementation is basically a port of the Matlab version of the code [3].

Parameters

- **clearsky_index** (*numeric or pandas.Series*) – Clear Sky Index time series that will be smoothed.
- **positions** (*numeric*) – Array of coordinate distances as (x,y) pairs representing the easting, northing of the site positions in meters [m]. Distributed plants could be simulated by gridded points throughout the plant footprint.
- **cloud_speed** (*numeric*) – Speed of cloud movement in meters per second [m/s].
- **dt** (*float, default None*) – The time series time delta. By default, is inferred from the clearsky_index. Must be specified for a time series that doesn't include an index. Units of seconds [s].

Returns

- **smoothed** (*numeric or pandas.Series*) – The Clear Sky Index time series smoothed for the described plant.
- **wavelet** (*numeric*) – The individual wavelets for the time series before smoothing.
- **tmscales** (*numeric*) – The timescales associated with the wavelets in seconds [s].

References

[1] M. Lave, J. Kleissl and J.S. Stein. A Wavelet-Based Variability Model (WVM) for Solar PV Power Plants. IEEE Transactions on Sustainable Energy, vol. 4, no. 2, pp. 501-509, 2013.

[2] M. Lave and J. Kleissl. Cloud speed impact on solar variability scaling - Application to the wavelet variability model. Solar Energy, vol. 91, pp. 11-21, 2013.

[3] Wavelet Variability Model - Matlab Code: https://pvpmc.sandia.gov/applications/wavelet-variability-model/

3.13 Comparison with PVLIB MATLAB

PVLIB was originally developed as a library for MATLAB at Sandia National Lab, and Sandia remains the official maintainer of the MATLAB library. Sandia supported the initial Python port and then released further project maintenance and development to the pvlib-python maintainers.

The pvlib-python maintainers collaborate with the PVLIB MATLAB maintainers but operate independently. We'd all like to keep the core functionality of the Python and MATLAB projects synchronized, but this will require the efforts of the larger pvlib-python community, not just the maintainers. Therefore, do not expect feature parity between the libaries, only similarity.

The PV_LIB Matlab help webpage is a good reference for this comparison.

3.13.1 Missing functions

See pvlib-python GitHub issue #2 for a list of functions missing from the Python version of the library.

3.13.2 Major differences

- pvlib-python uses git version control to track all changes to the code. A summary of changes is included in the whatsnew file for each release. PVLIB MATLAB documents changes in Changelog.docx
- pvlib-python has a comprehensive test suite, whereas PVLIB MATLAB does not have a test suite at all. Specifically, pvlib-python
  - Uses TravisCI for automated testing on Linux.
  - Uses Appveyor for automated testing on Windows.
  - Uses Coveralls to measure test coverage.
- Using readthedocs for automated documentation building and hosting.
- Removed `pvl_` from module/function names.
- Consolidated similar functions into topical modules. For example, functions from `pvl_clearsky_ineichen.m` and `pvl_clearsky_haurwitz.m` have been consolidated into `clearsky.py`.
- PVLIB MATLAB uses `location` structs as the input to some functions. pvlib-python just uses the lat, lon, etc. as inputs to the functions. Furthermore, pvlib-python replaces the structs with classes, and these classes have methods, such as *`get_solarposition()`*, that automatically reference the appropriate data. See *Modeling paradigms* for more information.
- pvlib-python implements a handful of class designed to simplify the PV modeling process. These include *`Location`*, *`PVSystem`*, *`LocalizedPVSystem`*, *`SingleAxisTracker`*, and *`ModelChain`*.

3.13.3 Other differences

- Very few tests of input validitity exist in the Python code. We believe that the vast majority of these tests were not necessary. We also make use of Python's robust support for raising and catching exceptions.
- Removed unnecessary and sometimes undesired behavior such as setting maximum zenith=90 or airmass=0. Instead, we make extensive use of `nan` values in returned arrays.
- Implemented the NREL solar position calculation algorithm. Also added a PyEphem option to solar position calculations.
- Specify time zones using a string from the standard IANA Time Zone Database naming conventions or using a pytz.timezone instead of an integer GMT offset.
- `clearsky.ineichen` supports interpolating monthly Linke Turbidities to daily resolution.
- Instead of requiring effective irradiance as an input, `pvsystem.sapm` calculates and returns it based on input POA irradiance, AM, and AOI.

- pvlib-python does not come with as much example data.
- pvlib-python does not currently implement as many algorithms as PVLIB MATLAB.

3.13.4 Documentation

- Using Sphinx to build the documentation, including dynamically created inline examples.
- Additional Jupyter tutorials in `/docs/tutorials`.

3.14 Variables and Symbols

There is a convention on consistent variable names throughout the library:

Table 66: List of used Variables and Parameters

variable	description
tz	timezone
latitude	latitude
longitude	longitude
dni	direct normal irradiance
dni_extra	direct normal irradiance at top of atmosphere (extraterrestrial)
dhi	diffuse horizontal irradiance
ghi	global horizontal irradiance
aoi	angle of incidence between $90\,\mathrm{deg}$ and $90\,\mathrm{deg}$
aoi_projection	cos(aoi)
airmass	airmass
airmass_relative	relative airmass
airmass_absolute	absolute airmass
poa_ground_diffuse	in plane ground reflected irradiation
poa_direct	direct/beam irradiation in plane
poa_diffuse	total diffuse irradiation in plane. sum of ground and sky diffuse.
poa_global	global irradiation in plane. sum of diffuse and beam projection.
poa_sky_diffuse	diffuse irradiation in plane from scattered light in the atmosphere (without ground reflected irradiation)
g_poa_effective	broadband plane of array effective irradiance.
surface_tilt	tilt angle of the surface
surface_azimuth	azimuth angle of the surface
solar_zenith	zenith angle of the sun in degrees
apparent_zenith	refraction-corrected solar zenith angle in degrees
solar_azimuth	azimuth angle of the sun in degrees East of North
temp_cell	temperature of the cell
temp_module	temperature of the module
temp_air	temperature of the air
temp_dew	dewpoint temperature
relative_humidity	relative humidity
v_mp, i_mp, p_mp	module voltage, current, power at the maximum power point

Continued on next page

Table 66 – continued from previous page

variable	description
v_oc	open circuit module voltage
i_sc	short circuit module current
i_x, i_xx	Sandia Array Performance Model IV curve parameters
effective_irradiance	effective irradiance
photocurrent	photocurrent
saturation_current	diode saturation current
resistance_series	series resistance
resistance_shunt	shunt resistance
transposition_factor	the gain ratio of the radiation on inclined plane to global horizontal irradiation: $\frac{poa\_global}{ghi}$
pdc0	nameplate DC rating
pdc, dc	dc power
gamma_pdc	module temperature coefficient. Typically in units of 1/C.
pac, ac	ac powe.
eta_inv	inverter efficiency
eta_inv_ref	reference inverter efficiency
eta_inv_nom	nominal inverter efficiency

For a definition and further explanation on the variables, common symbols and units refer to the following sources:

- Reference Variable List by PVPMC
- IEC 61724-1:2017 – Photovoltaic system performance - Part 1: Monitoring section: 3 – Terms and definitions; the Indian Standard referencing the withdrawn earlier global IEC standard IEC 61724:1998 is available online: IS/IEC 61724 (1998) and can provide relevant contents.
- Explanation of Solar irradiation and solar geometry by SoDa Service
  - Acronyms, Terminology and Units
  - Plane orientations and radiation components
  - Time references
  - Units and conversion tool
  - Terminology: definitions of the main quantities.
  - Acronyms in solar radiation (more extensive list)

Note: These further references might not use the same terminology as *pvlib*. But the physical process referred to is the same.

3.15 Single Diode Equation

This section reviews the solutions to the single diode equation used in pvlib-python to generate an IV curve of a PV module.

pvlib-python supports two ways to solve the single diode equation:

1. Lambert W-Function
2. Bishop's Algorithm

The *pvlib.pvsystem.singlediode()* function allows the user to choose the method using the `method` keyword.

3.15.1 Lambert W-Function

When `method='lambertw'`, the Lambert W-function is used as previously shown by Jain, Kapoor [1, 2] and Hansen [3]. The following algorithm can be found on Wikipedia: Theory of Solar Cells, given the basic single diode model equation.

$$I = I_L - I_0 \left(\exp\left(\frac{V + IR_s}{nNsV_{th}} \right) - 1 \right) - \frac{V + IR_s}{R_{sh}}$$

Lambert W-function is the inverse of the function $f(w) = w \exp(w)$ or $w = f^{-1}(w \exp(w))$ also given as $w = W(w \exp(w))$. Defining the following parameter, z, is necessary to transform the single diode equation into a form that can be expressed as a Lambert W-function.

$$z = \frac{R_s I_0}{nNsV_{th}\left(1 + \frac{R_s}{R_{sh}}\right)} \exp\left(\frac{R_s(I_L + I_0) + V}{nNsV_{th}\left(1 + \frac{R_s}{R_{sh}}\right)} \right)$$

Then the module current can be solved using the Lambert W-function, $W(z)$.

$$I = \frac{I_L + I_0 - \frac{V}{R_{sh}}}{1 + \frac{R_s}{R_{sh}}} - \frac{nNsV_{th}}{R_s} W(z)$$

3.15.2 Bishop's Algorithm

The function *pvlib.singlediode.bishop88()* uses an explicit solution [4] that finds points on the IV curve by first solving for pairs (V_d, I) where V_d is the diode voltage $V_d = V + I * Rs$. Then the voltage is backed out from V_d. Points with specific voltage, such as open circuit, are located using the bisection search method, `brentq`, bounded by a zero diode voltage and an estimate of open circuit voltage given by

$$V_{oc,est} = nNsV_{th} \log\left(\frac{I_L}{I_0} + 1 \right)$$

We know that $V_d = 0$ corresponds to a voltage less than zero, and we can also show that when $V_d = V_{oc,est}$, the resulting current is also negative, meaning that the corresponding voltage must be in the 4th quadrant and therefore greater than the open circuit voltage (see proof below). Therefore the entire forward-bias 1st quadrant IV-curve is bounded because $V_{oc} < V_{oc,est}$, and so a bisection search between 0 and $V_{oc,est}$ will always find any desired condition

in the 1st quadrant including V_{oc}.

$$I = I_L - I_0 \left(\exp\left(\frac{V_{oc,est}}{nNsV_{th}} \right) - 1 \right) - \frac{V_{oc,est}}{R_{sh}}$$

$$I = I_L - I_0 \left(\exp\left(\frac{nNsV_{th} \log\left(\frac{I_L}{I_0} + 1\right)}{nNsV_{th}} \right) - 1 \right) - \frac{nNsV_{th} \log\left(\frac{I_L}{I_0} + 1\right)}{R_{sh}}$$

$$I = I_L - I_0 \left(\exp\left(\log\left(\frac{I_L}{I_0} + 1 \right) \right) - 1 \right) - \frac{nNsV_{th} \log\left(\frac{I_L}{I_0} + 1\right)}{R_{sh}}$$

$$I = I_L - I_0 \left(\frac{I_L}{I_0} + 1 - 1 \right) - \frac{nNsV_{th} \log\left(\frac{I_L}{I_0} + 1\right)}{R_{sh}}$$

$$I = I_L - I_0 \left(\frac{I_L}{I_0} \right) - \frac{nNsV_{th} \log\left(\frac{I_L}{I_0} + 1\right)}{R_{sh}}$$

$$I = I_L - I_L - \frac{nNsV_{th} \log\left(\frac{I_L}{I_0} + 1\right)}{R_{sh}}$$

$$I = -\frac{nNsV_{th} \log\left(\frac{I_L}{I_0} + 1\right)}{R_{sh}}$$

3.15.3 References

[1] "Exact analytical solutions of the parameters of real solar cells using Lambert W-function," A. Jain, A. Kapoor, Solar Energy Materials and Solar Cells, 81, (2004) pp 269-277. DOI: 10.1016/j.solmat.2003.11.018

[2] "A new method to determine the diode ideality factor of real solar cell using Lambert W-function," A. Jain, A. Kapoor, Solar Energy Materials and Solar Cells, 85, (2005) 391-396. DOI: 10.1016/j.solmat.2004.05.022

[3] "Parameter Estimation for Single Diode Models of Photovoltaic Modules," Clifford W. Hansen, Sandia Report SAND2015-2065, 2015 DOI: 10.13140/RG.2.1.4336.7842

[4] "Computer simulation of the effects of electrical mismatches in photovoltaic cell interconnection circuits" JW Bishop, Solar Cell (1988) DOI: 10.1016/0379-6787(88)90059-2

CHAPTER 4

Indices and tables

- genindex
- search

Bibliography

[Ine02] P. Ineichen and R. Perez, "A New airmass independent formulation for the Linke turbidity coefficient", Solar Energy, 73, pp. 151-157, 2002.

[Ine08ss] P. Ineichen, "A broadband simplified version of the Solis clear sky model," Solar Energy, 82, 758-762 (2008).

[Ine16] P. Ineichen, "Validation of models that estimate the clear sky global and beam solar irradiance," Solar Energy, 132, 332-344 (2016).

[Ine08con] P. Ineichen, "Conversion function between the Linke turbidity and the atmospheric water vapor and aerosol content", Solar Energy, 82, 1095 (2008).

[Ren12] M. Reno, C. Hansen, and J. Stein, "Global Horizontal Irradiance Clear Sky Models: Implementation and Analysis", Sandia National Laboratories, SAND2012-2389, 2012.

[Ren16] Reno, M.J. and C.W. Hansen, "Identification of periods of clear sky irradiance in time series of GHI measurements" Renewable Energy, v90, p. 520-531, 2016.

[Mol98] B. Molineaux, P. Ineichen, and N. O'Neill, "Equivalence of pyrheliometric and monochromatic aerosol optical depths at a single key wavelength.," Appl. Opt., vol. 37, no. 30, pp. 7008–18, Oct. 1998.

[Kas96] F. Kasten, "The linke turbidity factor based on improved values of the integral Rayleigh optical thickness," Sol. Energy, vol. 56, no. 3, pp. 239–244, Mar. 1996.

[Bir80] R. E. Bird and R. L. Hulstrom, "Direct Insolation Models," 1980.

[Ang61] A. ÅNGSTRÖM, "Techniques of Determinig the Turbidity of the Atmosphere," Tellus A, vol. 13, no. 2, pp. 214–223, 1961.

[Lar16] Larson et. al. "Day-ahead forecasting of solar power output from photovoltaic plants in the American Southwest" Renewable Energy 91, 11-20 (2016).

[Liu60] B. Y. Liu and R. C. Jordan, The interrelationship and characteristic distribution of direct, diffuse, and total solar radiation, *Solar Energy* **4**, 1 (1960).

Python Module Index

p

Index

Symbols

A

B

C

D

E

F

G

H

R

S

T

U

V

W