1 Citing pvlib-python 3

2 Contents 5
   2.1 Package Overview ................................................................. 5
   2.2 What’s New .............................................................................. 12
   2.3 Installation ............................................................................. 18
   2.4 Contributing ........................................................................... 21
   2.5 Time and time zones ................................................................. 23
   2.6 Modules ................................................................................... 33
   2.7 Classes .................................................................................. 82
   2.8 Comparison with PVLIB MATLAB ............................................ 90
   2.9 Variables and Symbols ............................................................... 91

3 Indices and tables 93

Python Module Index 95
pvlid-python provides a set of documented functions for simulating the performance of photovoltaic energy systems. The toolbox was originally developed in MATLAB 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/.

The source code for pvlid-python is hosted on github.

Please see the Installation page for installation help.

For examples of how to use pvlid-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 pvlid-python GitHub wiki has a Projects and publications that use pvlid python page for inspiration and listing of your application.

There is a variable naming convention to ensure consistency throughout the library.
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 the most appropriate of:


Specific released versions of pvlib-python can be cited using their Zenodo DOI.
2.1 Package Overview

2.1.1 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.

2.1.2 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.

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

In [2]: import matplotlib.pyplot as plt

# seaborn makes the plots look nicer
In [3]: import seaborn as sns

In [4]: sns.set_color_codes()

In [5]: times = pd.DatetimeIndex(start='2015', end='2016', freq='1h')

# very approximate
# latitude, longitude, name, altitude
In [6]: coordinates = [(30, -110, 'Tucson', 700),
                    (35, -105, 'Albuquerque', 1500),
                    (40, -120, 'San Francisco', 10),
                    (50, 10, 'Berlin', 34)]
```
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 [14]: system = {'module': module, 'inverter': inverter,
            'surface_azimuth': 180}

In [15]: energies = {}

In [16]: for latitude, longitude, name, altitude in coordinates:
   ....:     system['surface_tilt'] = latitude
   ....:     cs = pvlib.clearsky.ineichen(times, latitude, longitude, altitude=altitude)
   ....:     solpos = pvlib.solarposition.get_solarposition(times, latitude, longitude)
   ....:     dni_extra = pvlib.irradiance.extraradiation(times)
   ....:     dni_extra = pd.Series(dni_extra, index=times)
   ....:     airmass = pvlib.atmosphere.relativeairmass(solpos['apparent_zenith'])
   ....:     pressure = pvlib.atmosphere.alt2pres(altitude)
   ....:     am_abs = pvlib.atmosphere.absoluteairmass(airmass, pressure)
   ....:     aoi = pvlib.irradiance.aoi(system['surface_tilt'], system['surface_azimuth'],
                                solpos['apparent_zenith'], solpos['azimuth'])
   ....:     total_irrad = pvlib.irradiance.total_irrad(system['surface_tilt'],
                                              system['surface_azimuth'],
                                              solpos['apparent_zenith'],
                                              solpos['azimuth'],
                                              cs['dni'], cs['ghi'], cs['dhi'],
                                              dni_extra=dni_extra,
                                              model='haydavies')
   ....:     temps = pvlib.pvsystem.sapm_celltemp(total_irrad['poa_global'],
                                              wind_speed, temp_air)
   ....:     dc = pvlib.pvsystem.sapm(module, total_irrad['poa_direct'],
                               total_irrad['poa_diffuse'], temps['temp_cell'],
                               am_abs, aoi)
   ....:     ac = pvlib.pvsystem.snlinverter(inverter, dc['v_mp'], dc['p_mp'])
   ....:     annual_energy = ac.sum()
   ....:     energies[name] = annual_energy

In [17]: energies = pd.Series(energies)
```
# based on the parameters specified above, these are in W*hrs
In [18]: print(energies.round(0))
Albuquerque  512617.0
Berlin        399745.0
San Francisco 458334.0
Tucson        477027.0
dtype: float64

In [19]: energies.plot(kind='bar', rot=0)
Out[19]: <matplotlib.axes._subplots.AxesSubplot at 0x7fb3b7c2c6d0>

In [20]: plt.ylabel('Yearly energy yield (W hr)')
Out[20]: <matplotlib.text.Text at 0x7fb3b7c31b10>

pvlib-python provides a basic_chain() function that implements much of the code above. Use this function with a full understanding of what it is doing internally!

In [21]: from pvlib.modelchain import basic_chain

In [22]: energies = {}

In [23]: for latitude, longitude, name, altitude in coordinates:
   ....:     dc, ac = basic_chain(times, latitude, longitude,
   ....:                             module, inverter,
   ....:                             altitude=altitude,
   ....:                             orientation_strategy='south_at_latitude_tilt')
   ....:     annual_energy = ac.sum()
   ....:     energies[name] = annual_energy
   ....:
In [24]: energies = pd.Series(energies)

# based on the parameters specified above, these are in W*hrs
In [25]: print(energies.round(0))

Albuquerque 512417.0
Berlin 399312.0
San Francisco 458110.0
Tucson 476849.0
dtype: float64

In [26]: energies.plot(kind='bar', rot=0)
Out[26]: <matplotlib.axes._subplots.AxesSubplot at 0x7fb3b4cedd90>

In [27]: plt.ylabel('Yearly energy yield (W hr)')
Out[27]: <matplotlib.text.Text at 0x7fb3b4d7a1d0>

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.

The following code demonstrates how to use `Location`, `PVSystem`, and `ModelChain` objects to accomplish our system modeling goal:

In [28]: from pvlib.pvsystem import PVSystem
```python
In [29]: from pvlib.location import Location
In [30]: from pvlib.modelchain import ModelChain
In [31]: system = PVSystem(module_parameters=module,
            inverter_parameters=inverter)
In [32]: energies = {}
In [33]: for latitude, longitude, name, altitude in coordinates:
            location = Location(latitude, longitude, name=name, altitude=altitude)
            mc = ModelChain(system, location,
            orientation_strategy='south_at_latitude_tilt')
            mc.run_model(times)
            annual_energy = mc.ac.sum()
            energies[name] = annual_energy

In [34]: energies = pd.Series(energies)
# based on the parameters specified above, these are in W*hrs
In [35]: print(energies.round(0))
Albuquerque   512417.0
Berlin         399312.0
San Francisco  458110.0
Tucson         476849.0
dtype: float64
In [36]: energies.plot(kind='bar', rot=0)
Out[36]: <matplotlib.axes._subplots.AxesSubplot at 0x7fb3b4d3c5d0>
In [37]: plt.ylabel('Yearly energy yield (W hr)')
Out[37]: <matplotlib.text.Text at 0x7fb3b4a43990>
```
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 following code demonstrates how to use a `LocalizedPVSystem` object to accomplish our modeling goal:

```python
In [38]: from pvlib.pvsystem import LocalizedPVSystem

In [39]: energies = {}

In [40]: for latitude, longitude, name, altitude in coordinates:
    localized_system = LocalizedPVSystem(module_parameters=module,
                                          inverter_parameters=inverter,
                                          surface_tilt=latitude,
                                          surface_azimuth=180,
                                          latitude=latitude,
                                          longitude=longitude,
                                          name=name,
                                          altitude=altitude)
    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'])
    temps = localized_system.sapm_celltemp(total_irrad['poa_global'],
                                             wind_speed, temp_air)
    aoi = localized_system.get_aoi(solar_position['apparent_zenith'],
                                    clearsky['apparent_zenith'],
                                    clearsky['apparent_elevation'],
                                    clearsky['apparent_azimuth'],
                                    clearsky['apparent_elevation'],
                                    clearsky['apparent_azimuth'])
```

---

Chapter 2. Contents
```python
solar_position['azimuth'])
airmass = localized_system.get_airmass(solar_position=solar_position)
dc = localized_system.sapm(total_irrad['poa_direct'],
total_irrad['poa_diffuse'],
temps['temp_cell'],
airmass['airmass_absolute'],
aoi)
ac = localized_system.snlinverter(dc['v_mp'], dc['p_mp'])
annual_energy = ac.sum()
energies[name] = annual_energy
```

```
In [41]: energies = pd.Series(energies)

# based on the parameters specified above, these are in W*hrs
In [42]: print(energies.round(0))
Albuquerque  512583.0
Berlin  399745.0
San Francisco  458334.0
Tucson  477012.0
dtype: float64
In [43]: energies.plot(kind='bar', rot=0)
Out[43]: <matplotlib.axes._subplots.AxesSubplot at 0x7fb3b498da50>
In [44]: plt.ylabel('Yearly energy yield (W hr)')
Out[44]: <matplotlib.text.Text at 0x7fb3b49f4410>
```
2.1.3 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.

2.1.4 Getting support

The best way to get support is to make an issue on our GitHub issues page.

2.1.5 How do I contribute?

We’re so glad you asked! Please see our wiki for information and instructions on how to contribute. We really appreciate it!

2.1.6 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 DOE EERE Postdoctoral Fellowship program for support. 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.

2.2 What’s New

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

2.2.1 v0.3.1 (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

2.2.2 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

2.2.3 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.
- 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)
• 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.

2.2.4 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

2.2.5 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
2.2.6 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

2.2.7 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 dateutil 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 validity remain. Garbage in, garbage or `nan` out.
- Removing unnecessary 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`.

2.2. What's New
• 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 \(K_t\) 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

2.3 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.

### 2.3.1 If you don’t have Python

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

Anaconda installs cleanly into a single directory, does not require Administrator or root privileges, does not affect other Python installs on your system, or interfere with OSX Frameworks. – The Anaconda Documentation

1. **Install** the full Anaconda Scientific Python distribution available at Continuum.io

   See the Anaconda FAQ for more information.

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

### 2.3.2 Install standard release

To install the most recent stable release of pvlib-python in a non-editable way, use conda (recommended if you use the Anaconda Python distribution) or pip (works with any Python distribution):

```
conda install -c pvlib pvlib
```

```
pip install 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.

### 2.3.3 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: source activate pvlibdev
3. **Install** additional packages into your development environment: 
   ```bash
   conda install jupyter ipython matplotlib seaborn nose flake8
   ```
   The [conda documentation](https://conda.io/docs/) has more information on how to use conda virtual environments. You can also add `--help` 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 
   ```bash
   pip install -e /path/to/your/pvlib-python
   ```
   You remember this path from the clone step, right? It’s probably something like `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 2, 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 `source activate pvlibdev` (or whatever you named your environment) when you start a new shell or terminal.

### 2.3.4 References

Here are a few recommended references for installing Python packages:

- The Pandas installation page
- [python4astronomers Modules, Packages, and all that](https://python4astronomers.com/)
- [Python Packaging User Guide](https://packaging.python.org/)
- [Conda User Guide](https://conda.io/docs/)

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](https://github.com/docs/github/forking)
- [Fork A Repo](https://github.com/docs/github/forking)
- [Cloning a repository](https://github.com/docs/github/clone)
- [Aha! Moments When Learning Git](https://github.com/docs/github/first-look)

### 2.4 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.
2.4.1 Easy ways to contribute

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

- Make GitHub issues and contribute to the conversation about how to resolve them.
- Read issues and pull requests that other people created and contribute to the conversation about how to resolve them.
- 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.

2.4.2 How to contribute new code

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.

Note that 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. This gives everybody an easy way to comment on the code and can make the process more efficient.

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.

The maintainers will follow same procedures, rather than making direct commits to the main repo. Exceptions may be made for extremely minor changes, such as fixing documentation typos.

2.4.3 This documentation

If this documentation is unclear, help us improve it! Consider looking at IPython, pandas, and Sandia-Labs/PVLIB_Python#33 for inspiration.
2.5 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.

2.5.1 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.

```python
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.

```python
In [4]: len(pytz.all_timezones)
Out[4]: 586
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/Port-au-Prince',
 'America/St_Barthlemy',
 'Antarctica/Davis',
 'Asia/Baghdad',
 'Asia/Gaza',
 'Asia/Kuala_Lumpur',
 'Asia/Riyadh',
 'Asia/Ust-Nera',
 'Australia/Brisbane',
 'Australia/Yancowinna',
 'EST',
 'Etc/GMT-10',
 'Europe/Andorra',
 'Europe/Kaliningrad',
 'Europe/San_Marino',
 'GB',
...]```
'Indian/Reunion',
'Pacific/Bougainville',
'Pacific/Midway',
'Pacific/Yap',
'US/Samoa']

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]:
[u'America/New_York',
 u'America/Detroit',
 u'America/Kentucky/Louisville',
 u'America/Kentucky/Monticello',
 u'America/Indiana/Indianapolis',
 u'America/Indiana/Vincennes',
 u'America/Indiana/Winamac',
 u'America/Indiana/Marengo',
 u'America/Indiana/Petersburg',
 u'America/Indiana/Vevay',
 u'America/Chicago',
 u'America/Indiana/Tell_City',
 u'America/Indiana/Knox',
 u'America/Menominee',
 u'America/North_Dakota/Center',
 u'America/North_Dakota/New_Salem',
 u'America/North_Dakota/Beulah',
 u'America/Denver',
 u'America/Boise',
 u'America/Phoenix',
 u'America/Los_Angeles',
 u'America/Anchorage',
 u'America/Juneau',
 u'America/Sitka',
 u'America/Metlakatla',
 u'America/Yakutat',
 u'America/Nome',
 u'America/Adak',
 u' 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',
```
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.

```python
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.

```python
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')
```

```python
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.

```python
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.

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

```python
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')
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...

```python
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.

```python
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.

```python
# 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`.

```python
# 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.

```python
# fail
In [37]: pytz.timezone('US/Mountain').localize(naive_python_date)
---------------------------------------------------------------------------
AttributeError: 'datetime.date' object has no attribute 'tzinfo'
```

### 2.5.2 pvlib-specific functionality

**Note:** This section applies to pvlib >= 0.3. Version 0.2 of pvlib used a `Location` object’s `tz` attribute to auto-magically 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.

```python
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.tmy.readtmy3(file_abspath)
```
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.

```python
In [45]: tmy3_data.index.tz
Out[45]: pytz.FixedOffset(-540)
In [46]: tmy3_data.ix[0:3, ['GHI', 'DNI', 'AOD']]
Out[46]:
   datetime       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.

```python
In [47]: solar_position = pvlib.solarposition.get_solarposition(tmy3_data.index, tmy3_metadata['latitude'], tmy3_metadata['longitude'])
In [48]: ax = solar_position.ix[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_xlim(-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)');
```

2.5. Time and time zones

---

**PVLIB_Python Documentation, Release 0.3.2+0.g1f2994b.dirty**
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.DatetimeIndex(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')
```

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

```python
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', '1997-01-01 00:00:00-09:00'],
```  

2.5. Time and time zones
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.
2.6 Modules

2.6.1 atmosphere

The atmosphere module contains methods to calculate relative and absolute airmass and to determine pressure from altitude or vice versa.

\texttt{pvlib.atmosphere.absolutearmass(airmass\_relative, pressure=101325.0)}

Determine absolute (pressure corrected) airmass from relative airmass and pressure

Gives the airmass for locations not at sea-level (i.e. not at standard pressure). The input argument “AMrelative” is the relative airmass. The input argument “pressure” is the pressure (in Pascals) at the location of interest and must be greater than 0. The calculation for absolute airmass is

\[ \text{absolute\_airmass} = \frac{\text{relative\_airmass} \times \text{pressure}}{101325} \]

**Parameters**
- **airmass\_relative**: scalar or Series
  - The airmass at sea-level.
- **pressure**: scalar or Series
  - The site pressure in Pascal.

**Returns**
- scalar or Series
  - Absolute (pressure corrected) airmass

**References**


\texttt{pvlib.atmosphere.alt2pres(altitude)}

Determine site pressure from altitude.

**Parameters**
- **Altitude**: scalar or Series
  - Altitude in meters above sea level

**Returns**
- **Pressure**: scalar or Series
  - Atmospheric pressure (Pascals)

**Notes**

The following assumptions are made

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base pressure</td>
<td>101325 Pa</td>
</tr>
<tr>
<td>Temperature at zero altitude</td>
<td>288.15 K</td>
</tr>
<tr>
<td>Gravitational acceleration</td>
<td>9.80665 m/s^2</td>
</tr>
<tr>
<td>Lapse rate</td>
<td>-6.5E-3 K/m</td>
</tr>
<tr>
<td>Gas constant for air</td>
<td>287.053 J/(kgK)</td>
</tr>
<tr>
<td>Relative Humidity</td>
<td>0%</td>
</tr>
</tbody>
</table>
References


`pvlib.atmosphere.pres2alt (pressure)`
Determine altitude from site pressure.

**Parameters**
- **pressure**: scalar or Series
  Atmospheric pressure (Pascals)

**Returns**
- **altitude**: scalar or Series
  Altitude in meters above sea level

Notes

The following assumptions are made

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base pressure</td>
<td>101325 Pa</td>
</tr>
<tr>
<td>Temperature at zero altitude</td>
<td>288.15 K</td>
</tr>
<tr>
<td>Gravitational acceleration</td>
<td>9.80665 m/s²</td>
</tr>
<tr>
<td>Lapse rate</td>
<td>-6.5E-3 K/m</td>
</tr>
<tr>
<td>Gas constant for air</td>
<td>287.053 J/(kgK)</td>
</tr>
<tr>
<td>Relative Humidity</td>
<td>0%</td>
</tr>
</tbody>
</table>

References


`pvlib.atmosphere.relativeairmass (zenith, model='kastenyoung1989')`
Gives the relative (not pressure-corrected) airmass.

Gives the airmass at sea-level when given a sun zenith angle (in degrees). The model variable allows selection of different airmass models (described below). If model is not included or is not valid, the default model is 'kastenyoung1989'.

**Parameters**
- **zenith**: float or Series
  Zenith angle of the sun in degrees. Note that some models use the apparent (refraction corrected) zenith angle, and some models use the true (not refraction-corrected) zenith angle. See model descriptions to determine which type of zenith angle is required. Apparent zenith angles must be calculated at sea level.

- **model**: String
  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

**Returns**

- `airmass_relative` : float or Series
  Relative airmass at sea level. Will return NaN values for any zenith angle greater than 90 degrees.

**References**


### 2.6.2 clearsky

The `clearsky` module contains several methods to calculate clear sky GHI, DNI, and DHI.

```python
cpplib.clearsky.haurwitz(apparent zenith)
```

Determine clear sky GHI from 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 of models which require only zenith angle [3]. Extreme care should be taken in the interpretation of this result!

**Parameters**

- `apparent zenith` : Series
  The apparent (refraction corrected) sun zenith angle in degrees.

**Returns**

- pd.Series
  The modeled global horizontal irradiance in W/m^2 provided by the Haurwitz clear-sky model.

Initial implementation of this algorithm by Matthew Reno.

**References**


`pvlib.clearsky.ineichen(time, latitude, longitude, altitude=0, linke_turbidity=None, solarposition_method='nrel_numpy', zenith_data=None, airmass_model='young1994', airmass_data=None, interp_turbidity=True)`

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**

- `time`: pandas.DatetimeIndex
  - `latitude`: float
  - `longitude`: float
  - `altitude`: float
  - `linke_turbidity`: None or float
    - If None, uses LinkeTurbidities.mat lookup table.
  - `solarposition_method`: string
    - Sets the solar position algorithm. See solarposition.get_solarposition()
  - `zenith_data`: None or Series
    - If None, ephemeris data will be calculated using solarposition_method.
  - `airmass_model`: string
    - See pvlib.airmass.relativeairmass().
  - `airmass_data`: None or Series
    - If None, absolute air mass data will be calculated using airmass_model and location.altitude.
  - `interp_turbidity`: bool
    - If True, interpolates the monthly Linke turbidity values found in LinkeTurbidities.mat to daily values.

**Returns**

DataFrame with the following columns: ghi, dni, dhi.

**Notes**

If you are using this function in a loop, it may be faster to load LinkeTurbidities.mat outside of the loop and feed it in as a keyword argument, rather than having the function open and process the file each time it is called.

**References**


```
pvlib.clearsky.lookup_linke_turbidity(time, latitude, longitude, filepath=None, interp_turbidity=True)
```
Look up the Linke Turbidity from the LinkeTurbidities.mat data file supplied with pvlib.

**Parameters**
- `time`: pandas.DatetimeIndex
- `latitude`: float
- `longitude`: float
- `filepath`: string
  - The path to the .mat file.
- `interp_turbidity`: bool
  - If True, interpolates the monthly Linke turbidity values found in LinkeTurbidities.mat to daily values.

**Returns**
- `turbidity`: Series

### 2.6.3 Irradiance

The irradiance module contains functions for modeling global horizontal irradiance, direct normal irradiance, diffuse horizontal irradiance, and total irradiance under various conditions.

```
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`: float or Series.
  - Panel tilt from horizontal.
- `surface_azimuth`: float or Series.
  - Panel azimuth from north.
- `solar_zenith`: float or Series.
  - Solar zenith angle.
- `solar_azimuth`: float or Series.
  - Solar azimuth angle.

**Returns**
- float or Series. Angle of incidence in degrees.

```
pvlib.irradiance.aoi_projection(surface_tilt, surface_azimuth, solar_zenith, solar_azimuth)
```
Calculates the dot product of the solar vector and the surface normal.

**Input all angles in degrees.**
**Parameters**

- **surface_tilt**: float or Series.
  Panel tilt from horizontal.
- **surface_azimuth**: float or Series.
  Panel azimuth from north.
- **solar_zenith**: float or Series.
  Solar zenith angle.
- **solar_azimuth**: float or Series.
  Solar azimuth angle.

**Returns**

float or Series. Dot product of panel normal and solar angle.

```
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**: float or Series.
  Panel tilt from horizontal.
- **surface_azimuth**: float or Series.
  Panel azimuth from north.
- **solar_zenith**: float or Series.
  Solar zenith angle.
- **solar_azimuth**: float or Series.
  Solar azimuth angle.
- **dni**: float or Series
  Direct Normal Irradiance

**Returns**

Series

```
pvlib.irradiance.dirint(ghi, zenith, times, pressure=101325, use_delta_kt_prime=True, temp_dew=None)
```
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 by using time-series GHI data and dew point temperature information. The effectiveness of the DIRINT model improves with each piece of information provided.

**Parameters**

- **ghi**: pd.Series
  Global horizontal irradiance in W/m^2.
- **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.
- **times**: DatetimeIndex
- **pressure**: float or pd.Series
  The site pressure in Pascal. Pressure may be measured or an average pressure may be calculated from site altitude.
use_delta_kt_prime : bool

 Indicates if the user would like to utilize the time-series nature of the GHI measurements. A value of False will not use the time-series improvements, any other numeric value will use time-series improvements. It is recommended that time-series data only be used if the time between measured data points is less than 1.5 hours. If none of the input arguments are vectors, then time-series improvements are not used (because it’s not a time-series).

temp_dew : None, float, or pd.Series

 Surface dew point temperatures, in degrees C. Values of temp_dew may be numeric or NaN. Any single time period point with a DewPtTemp=NaN does not have dew point improvements applied. If DewPtTemp is not provided, then dew point improvements are not applied.

Returns dni : pd.Series.

 The modeled direct normal irradiance in W/m^2 provided by the DIRINT model.

References


DIRINT model requires time series data (ie. one of the inputs must be a vector of length >2.

can use

disc (ghi, zenith, times, pressure=101325)

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.

Parameters ghi : Series

 Global horizontal irradiance in W/m^2.

solar_zenith : Series

 True (not refraction - corrected) solar zenith angles in decimal degrees.

times : DatetimeIndex

 Site pressure in Pascal.

Returns DataFrame with 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

See also:

atmosphere.alt2pres, dirint
References


```
import pvlib

pvlib.irradiance.extraradiation(datetime_or_doy, solar_constant=1366.1, method='spencer')
```

Determine extraterrestrial radiation from day of year.

**Parameters**

- `datetime_or_doy` : int, float, array, pd.DatetimeIndex
  
  Day of year, array of days of year e.g. pd.DatetimeIndex.dayofyear, or pd.DatetimeIndex.

- `solar_constant` : float
  
  The solar constant.

- `method` : string
  
  The method by which the ET radiation should be calculated. Options include 'pyephem', 'spencer', 'asce'.

**Returns**

float or Series

The extraterrestrial radiation present in watts per square meter on a surface which is normal to the sun. Eo is of the same size as the input doy.

'pyephem’ always returns a series.

**See also:**

pvlib.clearsky.disc

Notes

The Spencer method contains a minus sign discrepancy between equation 12 of [1]. It’s unclear what the correct formula is.

References


```
import pvlib

pvlib.irradiance.globalinplane(aoi, dni, poa_sky_diffuse, poa_ground_diffuse)
```

Determine the three components on in-plane irradiance

Combines in-plane irradiance components from the chosen diffuse translation, ground reflection and beam irradiance algorithms into the total in-plane irradiance.

**Parameters**

- `aoi` : float or Series

  Angle of incidence of solar rays with respect to the module surface, from `aoi()`.
**dni**: float or Series

Direct normal irradiance (W/m^2), as measured from a TMY file or calculated with a clearsky model.

**poa_sky_diffuse**: float or Series

Diffuse irradiance (W/m^2) in the plane of the modules, as calculated by a diffuse irradiance translation function

**poa_ground_diffuse**: float or Series

Ground reflected irradiance (W/m^2) in the plane of the modules, as calculated by an albedo model (e.g. `grounddiffuse()`) 

Returns DataFrame with 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)

Notes

Negative beam irradiation due to aoi > 90° or AOI < 0° is set to zero.

`pvlib.irradiance.grounddiffuse(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** : float or DataFrame

Surface tilt angles in decimal degrees. SurfTilt 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** : float or DataFrame

Global horizontal irradiance in W/m^2.

**albedo** : float or DataFrame

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 in


Returns float or DataFrame

Ground reflected irradiances in W/m^2.
References


The calculation is the last term of equations 3, 4, 7, 8, 10, 11, and 12.


and http://en.wikipedia.org/wiki/Albedo

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

\[ I_d = DHI(AR_b + (1 - A)(1 + \cos \beta/2)) \]

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**: 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 (e.g. North=0, South=180, East=90, West=270).

**dhi**: float or Series
- Diffuse horizontal irradiance in W/m².

**dni**: float or Series
- Direct normal irradiance in W/m².

**dni_extra**: float or Series
- Extraterrestrial normal irradiance in W/m².

**solar_zenith**: None, float or Series
- Solar apparent (refraction-corrected) zenith angles in decimal degrees. Must supply solar_zenith and solar_azimuth or supply projection_ratio.

**solar_azimuth**: None, float or Series
- Solar azimuth angles in decimal degrees. Must supply solar_zenith and solar_azimuth or supply projection_ratio.

**projection_ratio**: None, float or Series
- 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**: float or Series
- The diffuse component of the solar radiation on an arbitrarily tilted surface defined by the Perez model as given in reference [3]. Does not include the ground reflected irradiance or the irradiance due to the beam.
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**: float or Series
  
  Surface tilt angle in decimal degrees. surface_tilt must be \( \geq 0 \) and \( \leq 180 \). The tilt angle is defined as degrees from horizontal (e.g. surface facing up = 0, surface facing horizon = 90)

- **dhi**: float or Series
  
  Diffuse horizontal irradiance in W/m^2. DHI must be \( \geq 0 \).

**Returns**

- float or Series
  
  The diffuse component of the solar radiation on an arbitrarily tilted surface defined by the isotropic sky model as given in Loutzenhiser et. al (2007) equation 3.

SkyDiffuse is the diffuse component ONLY and does not include the ground reflected irradiance or the irradiance due to the beam.

SkyDiffuse is a column vector vector with a number of elements equal to the input vector(s).

---

**References**

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**: float or Series

Diffuse horizontal irradiance in W/m^2.

**ghi**: float or Series

Global horizontal irradiance in W/m^2.

**solar_zenith**: float or Series

Apparent (refraction-corrected) zenith angles in decimal degrees.

**Returns poa_sky_diffuse**: float or Series

The diffuse component of the solar radiation on an arbitrarily tilted surface as given by a model developed by David L. King at Sandia National Laboratories.

**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

\[
I_d = \frac{DHI}{2} \left(1 + \cos \beta \right) \left(1 + F' \sin^3 (\beta/2) \right) \left(1 + F' \cos^2 \theta \sin^3 \theta_z \right)
\]

where

\[
F' = 1 - \left(\frac{I_{d0}}{GHI}\right)
\]

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**: float or Series

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**: float or Series

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**: float or Series

diffuse horizontal irradiance in W/m^2. DHI must be >=0.

**ghi**: float or Series

Global irradiance in W/m^2. DNI must be >=0.

**solar_zenith**: float or Series

Apparent (refraction-corrected) zenith angles in decimal degrees. solar_zenith must be >=0 and <=180.

**solar_azimuth**: float or Series

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 float or Series.

The diffuse component of the solar radiation on an
arbitrarily tilted surface defined by the Klucher model as given in
Loutzenhiser et. al (2007) equation 4. SkyDiffuse is the diffuse component ONLY and does not include the ground
reflected irradiance or the irradiance due to the beam.

SkyDiffuse is a column vector vector with a number of elements equal to
the input vector(s).

References


111-114.

pvlib.irradiance.perez(surface_tilt, surface_azimuth, dhi, dni, dni_extra, solar_zenith, solar_azimuth, airmass, modelt='allsitescomposite1990')

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, di-
rect 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 : float or Series

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 : float or Series

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 : float or Series

Diffuse horizontal irradiance in W/m^2. DHI must be >=0.
dni : float or Series

Direct normal irradiance in W/m^2. DNI must be >=0.
dni_extra : float or Series

Extraterrestrial normal irradiance in W/m^2.
solar_zenith : float or Series

apparent (refraction-corrected) zenith angles in decimal degrees. solar_zenith must be >=0 and <=180.
solar_azimuth : float or Series
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** : float or Series

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'

**Returns** float or Series

The diffuse component of the solar radiation on an arbitrarily tilted surface defined by the Perez model as given in reference [3]. SkyDiffuse is the diffuse component ONLY and does not include the ground reflected irradiance or the irradiance due to the beam.

**References**


**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` : float or Series.
   Panel tilt from horizontal.

`surface_azimuth` : float or Series.
   Panel azimuth from north.

`solar zenith` : float or Series.
   Solar zenith angle.

`solar azimuth` : float or Series.
   Solar azimuth angle.

Returns float or Series. Ratio of the plane of array irradiance to the horizontal plane irradiance

```
2.6. Modules
```

```
Parameters `surface_tilt` : float or Series.
   Surface tilt angles in decimal degrees. The tilt angle is defined as degrees from horizon-
   tal (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 de-
   grees east of north (e.g. North = 0, South=180 East = 90, West = 270).

dhi : float or Series.
   diffuse horizontal irradiance in W/m^2.

dni : float or Series.
   direct normal irradiance in W/m^2.

ghi: float or Series.
   Global irradiance in W/m^2.

dni_extra : float or Series.
   extraterrestrial normal irradiance in W/m^2.

`solar zenith` : float or Series.
   apparent (refraction-corrected) zenith angles in decimal degrees.

`solar azimuth` : float or Series.
   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` : float or Series.
The diffuse component of the solar radiation on an arbitrarily tilted surface defined by
the Reindl model as given in Loutzenhiser et. al (2007) equation 8. SkyDiffuse is the
diffuse component ONLY and does not include the ground reflected irradiance or the
irradiance due to the beam. SkyDiffuse is a column vector vector with a number of
elements equal to the input vector(s).

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

Solar Energy 45(1), 9-17.

pvlib.irradiance.total_irrad(surface_tilt, surface_azimuth, apparent_zenith, azimuth, dni, ghi, dhi, dni_extra=None, airmass=None, albedo=0.25, surface_type=None, model='isotropic', model_perez='allsitescomposite1990', **kwargs)

Determine diffuse irradiance from the sky on a tilted surface.

\[ I_{tot} = I_{beam} + I_{sky} + I_{ground} \]

Parameters surface_tilt : float or Series.
    Panel tilt from horizontal.
surface_azimuth : float or Series.
    Panel azimuth from north.
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 : float or Series
    Extraterrestrial direct normal irradiance
airmass : float or Series
    Airmass
albedo : float
    Surface albedo
surface_type : String
    Surface type. See grounddiffuse.
model : String
    Irradiance model.
model_perez : String
    See perez.


References

2.6.4 location

This module contains the Location class.

```python
class pvlib.location.Location
    (latitude, longitude, tz='UTC', altitude=0, name=None, **kwargs)

Bases: object
```

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.
    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.
    Altitude from sea level in meters.
name : None or string.
    Sets the name attribute of the Location object.

**kwargs
    Arbitrary keyword arguments. Included for compatibility, but not used.

See also:
	pvsystem.PVSystem

classmethod 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
        Optionally attach the TMY data to this object.

    Returns Location object (or the child class of Location that you
called this method from).

get_airmass (times=None, solar_position=None, model='kastenyoung1989')
    Calculate the relative and absolute airmass.
    Automatically chooses zenith or apparent zenith depending on the selected model.

    Parameters times : None or DatetimeIndex
        Only used if solar_position is not provided.
    solar_position : None or DataFrame
        DataFrame with with columns ‘apparent_zenith’, ‘zenith’.
    model : str
        Relative airmass model

    Returns airmass : DataFrame
        Columns are ‘airmass_relative’, ‘airmass_absolute’

get_clearsky (times, model='ineichen', **kwargs)
    Calculate the clear sky estimates of GHI, DNI, and/or DHI at this location.

    Parameters times : DatetimeIndex
    model : str
        The clear sky model to use.

    Returns clearsky : DataFrame
        Column names are: ghi, dni, dhi.

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 : DatetimeIndex
    pressure : None, float, or array-like
If None, pressure will be calculated using `atmosphere.alt2pres()` and `self.altitude`.

**temperature**: None, float, or array-like

**kwargs passed to :py:func:`solarposition.get_solarposition`**

**Returns** `solar_position` : DataFrame

Columns depend on the `method` kwarg, but always include `zenith` and `azimuth`.

### 2.6.5 modelchain

The `modelchain` module contains functions and classes that combine many of the PV power modeling steps. These tools make it easy to get started with `pvlib` and demonstrate standard ways to use the library. With great power comes great responsibility: users should take the time to read the source code for the module.

```python
class pvlib.modelchain.ModelChain(system, location, orientation_strategy='south_at_latitude_tilt',
    clearsky_model='nieichen', transposition_model='haydavies',
    solar_position_method='nrel_numpy',
    airmass_model='kastenyoung1989', **kwargs)
```

**Bases**: `object`

An experimental class that represents all of the modeling steps necessary for calculating power or energy for a PV system at a given location.

**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
  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’.

- `clearsky_model` : str
  Passed to `location.get_clearsky`.

- `transposition_model` : str
  Passed to `system.get_irradiance`.

- `solar_position_method` : str
  Passed to `location.get_solarposition`.

- `airmass_model` : str
  Passed to `location.get_airmass`.

- `**kwargs` : Arbitrary keyword arguments. Included for compatibility, but not used.

**run_model**(times, irradiance=None, weather=None)

Run the model.
**Parameters**

- **times** : DatetimeIndex
  
  Times at which to evaluate the model.

- **irradiance** : None or DataFrame
  
  If None, calculates clear sky data. Columns must be ‘dni’, ‘ghi’, ‘dhi’.

- **weather** : None or DataFrame
  
  If None, assumes air temperature is 20 C and wind speed is 0 m/s. Columns must be ‘wind_speed’, ‘temp_air’.

**Returns**

Assigns attributes: times, solar_position, airmass, irradiance, total_irrad, weather, temps, aoi, dc, ac

An experimental function that computes all of the modeling steps necessary for calculating power or energy for a PV system at a given location.

- **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, CEC, or other.

- **inverter_parameters** : None, dict or Series
  
  Inverter parameters as defined by the SAPM, CEC, or other.

- **irradiance** : None or DataFrame
  
  If None, calculates clear sky data. Columns must be ‘dni’, ‘ghi’, ‘dhi’.

- **weather** : None or DataFrame
  
  If None, assumes air temperature is 20 C and wind speed is 0 m/s. Columns must be ‘wind_speed’, ‘temp_air’.

- **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).

```python
pvlib.modelchain.basic_chain(times, latitude, longitude, module_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)
```

See the [PVLIB documentation](https://pvlib-python.readthedocs.io) for more information.
orientation_strategy : None or str

   The strategy for aligning the modules. If not None, sets the surface_azimuth and surface_tilt properties of the system.

transposition_model : str
   Passed to system.get_irradiance.

solar_position_method : str
   Passed to location.get_solarposition.

airmass_model : str
   Passed to location.get_airmass.

altitude : None or float
   If None, computed from pressure. Assumed to be 0 m if pressure is also None.

pressure : None or float
   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(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

2.6.6 pvsystem

The pvsystem module contains functions for modeling the output and performance of PV modules and inverters.

class pvlib.pvsystem.LocalizedPVSystem(pvsystem=None, location=None, **kwargs)
   Bases: pvlib.pvsystem.PVSystem, pvlib.location.Location

   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.

   See the PVSystem class for an object model that describes an unlocalized PV system.

class pvlib.pvsystem.PVSystem(surface_tilt=0, surface_azimuth=180, albedo=None, surface_type=None, module=None, module_parameters=None, series_modules=None, parallel_modules=None, inverter=None, inverter_parameters=None, rack_ing_model='open_rack_cell_glassback', **kwargs)
   Bases: object
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 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  
Tilt angle of the module surface. Up=0, horizon=90.

**surface_azimuth**: float or array-like  
Azimuth angle of the module surface. North=0, East=90, South=180, West=270.

**albedo**: None, float  
The ground albedo. If None, will attempt to use surface_type and irradiance.SURFACE_ALBEDOS to lookup albedo.

**surface_type**: None, string  
The ground surface type. See irradiance.SURFACE_ALBEDOS for valid values.

**module**: None, string  
The model name of the modules. May be used to look up the module_parameters dictionary via some other method.

**module_parameters**: None, dict or Series  
Module parameters as defined by the SAPM, CEC, or other.

**inverter**: None, string  
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  
Inverter parameters as defined by the SAPM, CEC, or other.

**racking_model**: None or string  
Used for cell and module temperature calculations.

**kwargs**  
Arbitrary keyword arguments. Included for compatibility, but not used.

See also:

* pvlib.location.Location  
* pvlib.tracking.SingleAxisTracker  
* pvlib.pvsystem.LocalizedPVSystem  

**ashraeiam**(aoi)  
Determine the incidence angle modifier using self.module_parameters[‘b’], aoi, and the ashraeiam() function.

**Parameters**  
**aoi**: numeric  
The angle of incidence in degrees.

**Returns**  
**modifier**: numeric
The AOI modifier.

**calcparams_desoto**(poa_global, temp_cell, **kwargs)

Use the `calcparams_desoto()` function, the input parameters and `self.module_parameters` to calculate the module currents and resistances.

**Parameters**

- **poa_global**: float or Series
  The irradiance (in W/m^2) absorbed by the module.

- **temp_cell**: float or Series
  The average cell temperature of cells within a module in C.

- **kwargs**
  See `pvsystem.calcparams_desoto` for details

**Returns**  See `pvsystem.calcparams_desoto` for details

**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

**get_irradiance**(solar_zenith, solar_azimuth, dni, ghi, dhi, dni_extra=None, airmass=None, model='haydavies', **kwargs)

Uses the `irradiance.total_irrad()` 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**: float or Series
  Extraterrestrial direct normal irradiance

- **airmass**: float or Series
  Airmass

- **model**: String
Irradiance model.

**kwargs

Passed to irradiance.total_irrad().

Returns poa_irradiance : DataFrame

Column names are: total, beam, sky, ground.

i_from_v(resistance_shunt, resistance_series, nNsVth, voltage, saturation_current, photocurrent)

Wrapper around the i_from_v() function.

Parameters See pvsystem.i_from_v for details

Returns See pvsystem.i_from_v for details

localize(location=None, latitude=None, longitude=None, **kwargs)

Creates a LocalizedPVSystem object using this object and location data. Must supply either location object or latitude, longitude, and any location kwargs

Parameters location : None or Location

latitude : None or float

longitude : None or float

**kwargs : see Location

Returns localized_system : LocalizedPVSystem

physicaliam(aoi)

Determine the incidence angle modifier using self.module_parameters['K'],
self.module_parameters['L'], self.module_parameters['n'], aoi, and the
physicaliam() function.

Parameters See pvsystem.physicaliam for details

Returns See pvsystem.physicaliam for details

sapm(poa_direct, poa_diffuse, temp_cell, airmass_absolute, aoi, **kwargs)

Use the sapm() function, the input parameters, and self.module_parameters to calculate Voc, Isc, Ix, Ixx, Vmp/Imp.

Parameters poa_direct : Series

The direct irradiance incident upon the module (W/m^2).

poa_diffuse : Series

The diffuse irradiance incident on module.

temp_cell : Series

The cell temperature (degrees C).

airmass_absolute : Series

Absolute airmass.

aoi : Series

Angle of incidence (degrees).

**kwargs

See pvsystem.sapm for details

Returns See pvsystem.sapm for details
**sapm_celltemp** *(irrad, wind, temp)*

Uses *sapm_celltemp()* to calculate module and cell temperatures based on `self.racking_model` and the input parameters.

**Parameters** See pvsystem.sapm_celltemp for details

**Returns** See pvsystem.sapm_celltemp for details

**singlediode** *(photocurrent, saturation_current, resistance_series, resistance_shunt, nNsVth)*

Wrapper around the `singlediode()` function.

**Parameters** See pvsystem.singlediode for details

**Returns** See pvsystem.singlediode for details

**snlinverter** *(v_dc, p_dc)*

Uses `snlinverter()` to calculate AC power based on `self.inverter_parameters` and the input parameters.

**Parameters** See pvsystem.snlinverter for details

**Returns** See pvsystem.snlinverter for details

**pvlib.pvsystem.ashraeiam** *(b, aoi)*

Determine the incidence angle modifier using the ASHRAE transmission model.

ashraeiam calculates the incidence angle modifier as developed in [1], and adopted by ASHRAE (American Society of Heating, Refrigeration, and Air Conditioning Engineers) [2]. The model has been used by model programs such as PVsyst [3].

Note: For incident angles near 90 degrees, this model has a discontinuity which has been addressed in this function.

**Parameters**

- **b** : float
  
  A parameter to adjust the modifier as a function of angle of incidence. Typical values are on the order of 0.05 [3].

- **aoi** : Series
  
  The angle of incidence between the module normal vector and the sun-beam vector in degrees.

**Returns**

- **IAM** : Series
  
  The incident angle modifier calculated as $1-b*(sec(aoi)-1)$ as described in [2,3].

  Returns nan for all abs(aoi) >= 90 and for all IAM values that would be less than 0.

**See also:**

irradiance.aoi, physicaliam

**References**


[2] ASHRAE standard 93-77

Applies the temperature and irradiance corrections to inputs for singlediode.

Applies the temperature and irradiance corrections to the IL, I0, Rs, Rsh, and a parameters at reference conditions (IL_ref, I0_ref, etc.) according to the De Soto et. al description given in [1]. The results of this correction procedure may be used in a single diode model to determine IV curves at irradiance = S, cell temperature = Tcell.

**Parameters**

**poa_global** : float or Series

The irradiance (in W/m^2) absorbed by the module.

**temp_cell** : float or Series

The average cell temperature of cells within a module in C.

**alpha_isc** : float

The short-circuit current temperature coefficient of the module in units of 1/C.

**module_parameters** : dict

Parameters describing PV module performance at reference conditions according to DeSoto’s paper. Parameters may be generated or found by lookup. For ease of use, retrieve_sam can automatically generate a dict based on the most recent SAM CEC module database. The module_parameters dict must contain the following 5 fields:

- a_ref - modified diode ideality factor parameter at reference conditions (units of eV), a_ref can be calculated from the usual diode ideality factor (n), number of cells in series (Ns), and cell temperature (Tcell) per equation (2) in [1].
- I_L_ref - Light-generated current (or photocurrent) in amperes at reference conditions. This value is referred to as Iph in some literature.
- I_o_ref - diode reverse saturation current in amperes, under reference conditions.
- R_sh_ref - shunt resistance under reference conditions (ohms).
- R_s - series resistance under reference conditions (ohms).

**EgRef** : float

The energy bandgap at reference temperature (in eV). 1.121 eV for silicon. EgRef must be >0.

**dEgdT** : float

The temperature dependence of the energy bandgap at SRC (in 1/C). May be either a scalar value (e.g. -0.0002677 as in [1]) or a DataFrame of dEgdT values corresponding to each input condition (this may be useful if dEgdT is a function of temperature).

**M** : float or Series (optional, default=1)

An optional airmass modifier, if omitted, M is given a value of 1, which assumes absolute (pressure corrected) airmass = 1.5. In this code, M is equal to M/Mref as described in [1] (i.e. Mref is assumed to be 1). Source [1] suggests that an appropriate value for M as a function absolute airmass (AMa) may be:

```python
>>> M = np.polyval([-0.000126, 0.002816, -0.024459, 0.086257, 0.918093],
                  AMa)
```

M may be a Series.

**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** : float or Series
Light-generated current in amperes at irradiance=S and cell temperature=Tcell.

**saturation_current** : float or Series
Diode saturation current in amperes at irradiance S and cell temperature Tcell.

**resistance_series** : float
Series resistance in ohms at irradiance S and cell temperature Tcell.

**resistance_shunt** : float or Series
Shunt resistance in ohms at irradiance S and cell temperature Tcell.

**nNsVth** : float or Series
Modified diode ideality factor at irradiance S and cell temperature Tcell. Note that in source [1] nNsVth = a (equation 2). nNsVth is the product of the usual diode ideality factor (n), the number of series-connected cells in the module (Ns), and the thermal voltage of a cell in the module (Vth) at a cell temperature of Tcell.

**See also:**

* sapm, sapm_celltemp, 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” air mass 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 air mass) per DeSoto’s equations.

**Silicon (Si):**

- EgRef = 1.121
- dEgdT = -0.0002677

```python
>>> M = np.polyval([-1.26E-4, 2.816E-3, -0.024459, 0.086257, 0.918093], AMa)
```

Source: [1]

**Cadmium Telluride (CdTe):**

- EgRef = 1.475
• $dE/gT = -0.0003$

```python
>>> M = np.polyval([-2.46E-5, 9.607E-4, -0.0134, 0.0716, 0.9196], AMa)
```

Source: [4]

**Copper Indium diSelenide (CIS):**

• $E_{g\text{Ref}} = 1.010$
• $dE/gT = -0.00011$

```python
>>> M = np.polyval([-3.74E-5, 0.00125, -0.01462, 0.0718, 0.9210], AMa)
```

Source: [4]

**Copper Indium Gallium diSelenide (CIGS):**

• $E_{g\text{Ref}} = 1.15$
• $dE/gT = ???$

```python
>>> M = np.polyval([-9.07E-5, 0.0022, -0.0202, 0.0652, 0.9417], AMa)
```


**Gallium Arsenide (GaAs):**

• $E_{g\text{Ref}} = 1.424$
• $dE/gT = -0.000433$
• $M = \text{unknown}$

Source: [4]

**References**


```
pvlib.pvsystem.i_from_v(resistance_shunt, resistance_series, nNsVth, voltage, saturation_current, photocurrent)
```

Calculates current from voltage per Eq 2 Jain and Kapoor 2004 [1].

**Parameters**

- `resistance_shunt`: float or Series
  
  Shunt resistance in ohms under desired IV curve conditions. Often abbreviated $R_{sh}$.

- `resistance_series`: float or Series
  
  Series resistance in ohms under desired IV curve conditions. Often abbreviated $R_s$.

- `nNsVth`: float or Series
The product of three components. 1) The usual diode ideal factor \( n \), 2) the number of cells in series \( (N_s) \), and 3) the cell thermal voltage under the desired IV curve conditions \( (V_{th}) \). The thermal voltage of the cell (in volts) may be calculated as \( k \cdot \text{temp}_\text{cell}/q \), where \( k \) is Boltzmann's constant \((\text{J/K})\), \( \text{temp}_\text{cell} \) is the temperature of the p-n junction in Kelvin, and \( q \) is the charge of an electron \((\text{coulombs})\).

**voltage**: float or Series

The voltage in Volts under desired IV curve conditions.

**saturation_current**: float or Series

Diode saturation current in amperes under desired IV curve conditions. Often abbreviated \( I_0 \).

**photocurrent**: float or Series

Light-generated current (photocurrent) in amperes under desired IV curve conditions. Often abbreviated \( I_L \).

**Returns current**: np.array

**References**


```python
def physicaliam(K, L, n, aoi):
    # Determine the incidence angle modifier using refractive index, glazing thickness, and extinction coefficient
    # physicaliam calculates the incidence angle modifier as described in De Soto et al. “Improvement and validation of a model for photovoltaic array performance”, section 3. The calculation is based upon a physical model of absorption and transmission through a cover. Required information includes, incident angle, cover extinction coefficient, cover thickness
    # Note: The authors of this function believe that eqn. 14 in [1] is incorrect. This function uses the following equation in its place: \( \theta_r = \arcsin(1/n \cdot \sin(\theta)) \)
```

**Parameters**

- **K**: float
  
  The glazing extinction coefficient in units of 1/meters. Reference [1] indicates that a value of 4 is reasonable for “water white” glass. \( K \) must be a numeric scalar or vector with all values \( \geq 0 \). If \( K \) is a vector, it must be the same size as all other input vectors.

- **L**: float
  
  The glazing thickness in units of meters. Reference [1] indicates that 0.002 meters (2 mm) is reasonable for most glass-covered PV panels. \( L \) must be a numeric scalar or vector with all values \( \geq 0 \). If \( L \) is a vector, it must be the same size as all other input vectors.

- **n**: float
  
  The effective index of refraction (unitless). Reference [1] indicates that a value of 1.526 is acceptable for glass. \( n \) must be a numeric scalar or vector with all values \( \geq 0 \). If \( n \) is a vector, it must be the same size as all other input vectors.

- **aoi**: Series

  The angle of incidence between the module normal vector and the sun-beam vector in degrees.
Returns IAM : float or Series

The incident angle modifier as specified in eqns. 14-16 of [1]. IAM is a column vector with the same number of elements as the largest input vector.

Theta must be a numeric scalar or vector. For any values of theta where abs(aoi)>90, IAM is set to 0. For any values of aoi where -90 < aoi < 0, theta is set to abs(aoi) and evaluated.

See also:
getaoi, ephemeris, spa, ashraeiam

References


pvlib.pvsystem.retrieve_sam(name=None, samfile=None)
Retrieve latest module and inverter info from SAM website.

This function will retrieve either:

- CEC module database
- Sandia Module database
- CEC Inverter database

and return it as a pandas dataframe.

Parameters name : String

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

samfile : String

Absolute path to the location of local versions of the SAM file. If file is specified, the latest versions of the SAM database will not be downloaded. The selected file must be in .csv format.

If set to ‘select’, a dialogue will open allowing the user to navigate to the appropriate page.

Returns 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
Examples

```python
>>> from pvlib import pvsystem
>>> invdb = pvsystem.retrieve_sam(name='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
Mpppt_low 200.000000
Mpppt_high 500.000000
Name: AE_Solar_Energy__AE6_0__277V__277V__CEC_2012_, dtype: float64
```

`pvlib.pvsystem.sapm(module, poa_direct, poa_diffuse, temp_cell, airmass_absolute, aoi)`

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**

- **module**: Series or dict
  
  A DataFrame defining the SAPM performance parameters. See the notes section for more details.

- **poa_direct**: Series
  
  The direct irradiance incident upon the module (W/m^2).

- **poa_diffuse**: Series
  
  The diffuse irradiance incident on module.

- **temp_cell**: Series
  
  The cell temperature (degrees C).

- **airmass_absolute**: Series
  
  Absolute airmass.

- **aoi**: Series
  
  Angle of incidence (degrees).

**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

• effective_irradiance : Effective irradiance

See also:
retrieve_sam, sapm_celltemp

Notes

The coefficients from SAPM which are required in module are listed in the following table.

The modules in the Sandia module database contain these coefficients, but the modules in the CEC module database do not. Both databases can be accessed using retrieve_sam().

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0-A4</td>
<td>The airmass coefficients used in calculating effective irradiance</td>
</tr>
<tr>
<td>B0-B5</td>
<td>The angle of incidence coefficients used in calculating effective irradiance</td>
</tr>
<tr>
<td>C0-C7</td>
<td>The empirically determined coefficients relating Imp, Vmp, Ix, and Ixx to effective irradiance</td>
</tr>
<tr>
<td>Isco</td>
<td>Short circuit current at reference condition (amps)</td>
</tr>
<tr>
<td>Impo</td>
<td>Maximum power current at reference condition (amps)</td>
</tr>
<tr>
<td>Aisc</td>
<td>Short circuit current temperature coefficient at reference condition (I/C)</td>
</tr>
<tr>
<td>Aimp</td>
<td>Maximum power current temperature coefficient at reference condition (I/C)</td>
</tr>
<tr>
<td>Bvoco</td>
<td>Open circuit voltage temperature coefficient at reference condition (V/C)</td>
</tr>
<tr>
<td>Mbvoc</td>
<td>Coefficient providing the irradiance dependence for the BetaVoc temperature coefficient at reference irradiance (V/C)</td>
</tr>
<tr>
<td>Bvmpo</td>
<td>Maximum power voltage temperature coefficient at reference condition</td>
</tr>
<tr>
<td>Mbvmp</td>
<td>Coefficient providing the irradiance dependence for the BetaVmp temperature coefficient at reference irradiance (V/C)</td>
</tr>
<tr>
<td>N</td>
<td>Empirically determined “diode factor” (dimensionless)</td>
</tr>
<tr>
<td>Cells_in_Series</td>
<td>Number of cells in series in a module’s cell string(s)</td>
</tr>
<tr>
<td>IXO</td>
<td>Ix at reference conditions</td>
</tr>
<tr>
<td>IXXO</td>
<td>Ixx at reference conditions</td>
</tr>
<tr>
<td>FD</td>
<td>Fraction of diffuse irradiance used by module</td>
</tr>
</tbody>
</table>

References


pvlib.pvsystem.sapm_celltemp(poa_global, wind_speed, temp_air, model='open_rack_cell_glassback')

Estimate cell and module temperatures per the Sandia PV Array Performance Model (SAPM, SAND2004-3535), from the incident irradiance, wind speed, ambient temperature, and SAPM module parameters.

Parameters poa_global : float or Series
Total incident irradiance in W/m^2.

wind_speed : float or Series
Wind speed in m/s at a height of 10 meters.

temp_air : float or Series
Ambient dry bulb temperature in degrees C.
model: string, list, or dict

Model to be used.

If string, can be:
• 'open_rack_cell_glassback' (default)
• 'roof_mount_cell_glassback'
• 'open_rack_cell_polymerback'
• 'insulated_back_polymerback'
• 'open_rack_polymer_thinfilm_steel'
• '22x_concentrator_tracker'

If dict, supply the following parameters (if list, in the following order):
• a [float] SAPM module parameter for establishing the upper limit for module temperature at low wind speeds and high solar irradiance.
• b [float] SAPM module parameter for establishing the rate at which the module temperature drops as wind speed increases (see SAPM eqn. 11).
• deltaT [float] SAPM module parameter giving the temperature difference between the cell and module back surface at the reference irradiance, E0.

Returns: DataFrame with columns ‘temp_cell’ and ‘temp_module’.

Values in degrees C.

See also: sapm

References


pvlib.pvsystem.singlediode(module, photocurrent, saturation_current, resistance_series, resistance_shunt, nNsVth)

Solve the single-diode model to obtain a photovoltaic IV curve.

Singlediode solves the single diode equation [1]

\[ I = I_L - I_0 \left[ \exp \left( \frac{(V + I * R_s)/(n N_s V_{th})}{n N_s V_{th}} \right) - 1 \right] - \frac{(V + I * R_s)}{R_{sh}} \]

for I and V when given \( I_L \), \( I_0 \), \( R_s \), \( R_{sh} \), and \( n N_s V_{th} \) which are described later. Returns a DataFrame which contains the 5 points on the I-V curve specified in SAND2004-3535 [3]. If all \( I_L \), \( I_0 \), \( R_s \), \( R_{sh} \), and \( n N_s V_{th} \) are scalar, a single curve will be returned, if any are Series (of the same length), multiple IV curves will be calculated.

The input parameters can be calculated using calcparams_desoto from meteorological data.

Parameters:

module: DataFrame

A DataFrame defining the SAPM performance parameters.

photocurrent: float or Series

Light-generated current (photocurrent) in amperes under desired IV curve conditions. Often abbreviated \( I_L \).
saturation_current : float or Series

Diode saturation current in amperes under desired IV curve conditions. Often abbreviated I_0.

resistance_series : float or Series

Series resistance in ohms under desired IV curve conditions. Often abbreviated Rs.

resistance_shunt : float or Series

Shunt resistance in ohms under desired IV curve conditions. Often abbreviated Rsh.

nNsVth : float or Series

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).

Returns If photocurrent is a Series, a DataFrame with the following columns. All columns have the same number of rows as the largest input DataFrame.

If photocurrent is a scalar, a dict with the following keys.

- 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 \( V = 0.5*(v_{oc}+v_{mp}) \).

See also:
sapm, calcparams_desoto

Notes

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].

References

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**

- **inverter**: DataFrame

  A DataFrame 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.

  Required DataFrame columns are:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pac0</td>
<td>AC-power output from inverter based on input power and voltage (W)</td>
</tr>
<tr>
<td>Pdc0</td>
<td>DC-power input to inverter, typically assumed to be equal to the PV array maximum power (W)</td>
</tr>
<tr>
<td>Vdc0</td>
<td>DC-voltage level at which the AC-power rating is achieved at the reference operating condition (V)</td>
</tr>
<tr>
<td>Ps0</td>
<td>DC-power required to start the inversion process, or self-consumption by inverter, strongly influences inverter efficiency at low power levels (W)</td>
</tr>
<tr>
<td>C0</td>
<td>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)</td>
</tr>
<tr>
<td>C1</td>
<td>Empirical coefficient allowing Pdco to vary linearly with dc-voltage input, default value is zero (1/V)</td>
</tr>
<tr>
<td>C2</td>
<td>Empirical coefficient allowing Pso to vary linearly with dc-voltage input, default value is zero (1/V)</td>
</tr>
<tr>
<td>C3</td>
<td>Empirical coefficient allowing Co to vary linearly with dc-voltage input, default value is zero (1/V)</td>
</tr>
<tr>
<td>Pnt</td>
<td>AC-power consumed by inverter at night (night tare) to maintain circuitry required to sense PV array voltage (W)</td>
</tr>
</tbody>
</table>

- **v_dc**: float or Series

  DC voltages, in volts, which are provided as input to the inverter. Vdc must be >= 0.

- **p_dc**: float or Series

  A scalar or DataFrame of DC powers, in watts, which are provided as input to the inverter. Pdc must be >= 0.

**Returns**

- **ac_power**: float or Series

  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 maximum power point tracking (MPPT) voltage windows or maximum current limits of the inverter.

**See also:**

- `sapm`, `singlediode`
References


c12v.pvsystem.systemdef (meta, surface_tilt, surface_azimuth, albedo, series_modules, parallel_modules)
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:

<table>
<thead>
<tr>
<th>meta field</th>
<th>format</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>meta.altitude</td>
<td>Float</td>
<td>site elevation</td>
</tr>
<tr>
<td>meta.latitude</td>
<td>Float</td>
<td>site latitude</td>
</tr>
<tr>
<td>meta.longitude</td>
<td>Float</td>
<td>site longitude</td>
</tr>
<tr>
<td>meta.Name</td>
<td>String</td>
<td>site name</td>
</tr>
<tr>
<td>meta.State</td>
<td>String</td>
<td>state</td>
</tr>
<tr>
<td>meta.TZ</td>
<td>Float</td>
<td>timezone</td>
</tr>
</tbody>
</table>

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.

series_modules : int

Number of modules connected in series in a string.

parallel_modules : int

Number of strings connected in parallel.

Returns Result : dict

A dict with the following fields.
• ‘surface_tilt’
• ‘surface_azimuth’
• ‘albedo’
• ‘series_modules’
• ‘parallel_modules’
• ‘latitude’
• ‘longitude’
• ‘tz’
• ‘name’
• ‘altitude’

See also:

`pvlib.tmy.readtmy3`, `pvlib.tmy.readtmy2`

## 2.6.7 solarposition

Calculate the solar position using a variety of methods/packages.

```python
pvlib.solarposition.calc_time(lower_bound, upper_bound, latitude, longitude, attribute, value,
    altitude=0, pressure=101325, temperature=12, 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`
- `longitude` : `float`
- `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`
  
  Distance above sea level.
- `pressure` : `int` or `float`, optional
  
  Air pressure in Pascals. Set to 0 for no atmospheric correction.
- `temperature` : `int` or `float`, optional
  
  Air temperature in degrees C.
- `xtol` : `float`, optional
  
  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.

```python
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
- **latitude**: float
- **longitude**: float
- **pressure**: float or Series
  - Ambient pressure (Pascals)
- **temperature**: float or Series
  - Ambient temperature (C)

Returns  

A 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).

See also:

- `pyephem`, `spa_c`, `spa_python`

References


`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
- **latitude**: float
- **longitude**: float
- **altitude**: None or float
  - If None, computed from pressure. Assumed to be 0 m if pressure is also None.
- **pressure**: None or float
  - If None, computed from altitude. Assumed to be 101325 Pa if altitude is also None.
- **method**: string
  - ‘pyephem’ uses the PyEphem package: `pyephem()`
  - ‘nrel_c’ uses the NREL SPA C code [3]: `spa_c()`
  - ‘nrel_numpy’ uses an implementation of the NREL SPA algorithm described in [1] (default): `spa_python()`
‘nrel_numba’ uses an implementation of the NREL SPA algorithm described in [1], but also compiles the code first: `spa_python()`

‘ephemeris’ uses the pvlib ephemeris code: `ephemeris()`

**temperature**: float

Degrees C.

Other keywords are passed to the underlying solar position function.

### References


**pvlib.solarposition.get_sun_rise_set_transit** *(time, latitude, longitude, how='numpy', delta_t=None, numthreads=4)*

Calculate the sunrise, sunset, and sun transit times using the NREL SPA algorithm 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
  
  Only the date part is used

- **latitude**: float

- **longitude**: float

- **delta_t**: float, optional
  
  Difference between terrestrial time and UT1. By default, use USNO historical data and predictions

- **how**: str, optional
  
  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
  
  Number of threads to use if how == ‘numba’.

**Returns**

DataFrame

The DataFrame will have the following columns: sunrise, sunset, transit

### References


**pvlib.solarposition.pyephem** *(time, latitude, longitude, altitude=0, pressure=101325, temperature=12)*

Calculate the solar position using the PyEphem package.

**Parameters**

- **time**: pandas.DatetimeIndex

---

### 2.6. Modules

71
Localized or UTC.

latitude : float
longitude : float
altitude : float
distance above sea level.

pressure : int or float, optional
air pressure in Pascals.

temperature : int or float, optional
air temperature in degrees C.

Returns DataFrame

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.pyephem_earthsun_distance(time)
Calculates the distance from the earth to the sun using pyephem.

Parameters time : pd.DatetimeIndex

Returns pd.Series. Earth-sun distance in AU.

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 separately and used in accordance with its license.

Parameters time : pandas.DatetimeIndex

Localized or UTC.

latitude : float
longitude : float
pressure : float
  Pressure in Pascals
altitude : float
  Elevation above sea level.

temperature : float
  Temperature in C
delta_t : float
  Difference between terrestrial time and UT1. USNO has previous values and predictions.

raw_spa_output : bool
  If true, returns the raw SPA output.
Returns DataFrame

The DataFrame will have the following columns: elevation, azimuth, zenith, apparent_elevation, apparent_zenith.

See also:
pyephem, spa_python, ephemeris

References

NREL SPA code: http://rredc.nrel.gov/solar/codesandalgorithms/spa/

pvlib.solarposition.spa_python(time, latitude, longitude, altitude=0, pressure=101325, temperature=12, delta_t=None, atmos_refract=None, how='numpy', numthreads=4)

Calculate the solar position using a python implementation of the NREL SPA algorithm 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
    Localized or UTC.
latitude : float
longitude : float
altitude : float
pressure : int or float, optional
    avg. yearly air pressure in Pascals.
temperature : int or float, optional
    avg. yearly air temperature in degrees C.
delta_t : float, optional
    Difference between terrestrial time and UT1. The USNO has historical and forecasted delta_t [3].
atmos_refrac : float, optional
    The approximate atmospheric refraction (in degrees) at sunrise and sunset.
how : str, optional
    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
    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).
See also:

*pyephem*, *spa_c*, *ephemeris*

References


### 2.6.8 tmy

Import functions for TMY2 and TMY3 data files.

```python
tmy.readtmy2(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 conforming to modern conventions (e.g. N latitude is positive, E longitude is positive, the “24th” hour of any day is technically the “0th” hour of the next day). In the case of any discrepancies 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 data frame 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.

<table>
<thead>
<tr>
<th>key</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WBAN</td>
<td>Site identifier code (WBAN number)</td>
</tr>
<tr>
<td>City</td>
<td>Station name</td>
</tr>
<tr>
<td>State</td>
<td>Station state 2 letter designator</td>
</tr>
<tr>
<td>TZ</td>
<td>Hours from Greenwich</td>
</tr>
<tr>
<td>latitude</td>
<td>Latitude in decimal degrees</td>
</tr>
<tr>
<td>longitude</td>
<td>Longitude in decimal degrees</td>
</tr>
<tr>
<td>altitude</td>
<td>Site elevation in meters</td>
</tr>
<tr>
<td>TMYData field</td>
<td>description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>index</td>
<td>Pandas timeseries object containing timestamps</td>
</tr>
<tr>
<td>year</td>
<td>Extraterrestrial horizontal radiation recv’d during 60 minutes prior to timestamp, Wh/m^2</td>
</tr>
<tr>
<td>month</td>
<td>Extraterrestrial normal radiation recv’d during 60 minutes prior to timestamp, Wh/m^2</td>
</tr>
<tr>
<td>day</td>
<td>Direct and diffuse horizontal radiation recv’d during 60 minutes prior to timestamp, Wh/m^2</td>
</tr>
<tr>
<td>hour</td>
<td>See [1], Table 3-3</td>
</tr>
<tr>
<td>ETR</td>
<td>Amount of direct normal radiation (modeled) recv’d during 60 minutes prior to timestamp, Wh/m^2</td>
</tr>
<tr>
<td>ETRN</td>
<td>See [1], Table 3-3</td>
</tr>
<tr>
<td>GHI</td>
<td>See [1], Table 3-4</td>
</tr>
<tr>
<td>GHIUncertainty</td>
<td>See [1], Table 3-4</td>
</tr>
<tr>
<td>GHISource</td>
<td>See [1], Table 3-3</td>
</tr>
<tr>
<td>GHIUncertainty</td>
<td>See [1], Table 3-4</td>
</tr>
<tr>
<td>DNI</td>
<td>Amount of direct normal radiation recv’d during 60 minutes prior to timestamp, Wh/m^2</td>
</tr>
<tr>
<td>DNIsource</td>
<td>See [1], Table 3-3</td>
</tr>
<tr>
<td>DNIUncertainty</td>
<td>See [1], Table 3-4</td>
</tr>
<tr>
<td>DNIsource</td>
<td>See [1], Table 3-3</td>
</tr>
<tr>
<td>DNIUncertainty</td>
<td>See [1], Table 3-4</td>
</tr>
<tr>
<td>DHI</td>
<td>Avg. total horizontal illuminance recv’d during the 60 minutes prior to timestamp, units of 100 lux (e.g. value of 70 = 700 lux)</td>
</tr>
<tr>
<td>DHIsource</td>
<td>See [1], Table 3-3</td>
</tr>
<tr>
<td>DHIUncertainty</td>
<td>See [1], Table 3-4</td>
</tr>
<tr>
<td>DHIsource</td>
<td>See [1], Table 3-3</td>
</tr>
<tr>
<td>DHIUncertainty</td>
<td>See [1], Table 3-4</td>
</tr>
<tr>
<td>DNIllum</td>
<td>Avg. direct normal illuminance recv’d during the 60 minutes prior to timestamp, units of 100 lux</td>
</tr>
<tr>
<td>DNIllumSource</td>
<td>See [1], Table 3-3</td>
</tr>
<tr>
<td>DNIllumUncertainty</td>
<td>See [1], Table 3-4</td>
</tr>
<tr>
<td>DDIllum</td>
<td>Avg. horizontal diffuse illuminance recv’d during the 60 minutes prior to timestamp, units of 100 lux</td>
</tr>
<tr>
<td>DDIllumSource</td>
<td>See [1], Table 3-3</td>
</tr>
<tr>
<td>DDIllumUncertainty</td>
<td>See [1], Table 3-4</td>
</tr>
<tr>
<td>Zenithlum</td>
<td>Avg. luminance at the sky’s zenith during the 60 minutes prior to timestamp, units of 10 Cd/m^2 (e.g. value of 700 = 7000 Cd/m^2)</td>
</tr>
<tr>
<td>ZenithlumSource</td>
<td>See [1], Table 3-3</td>
</tr>
<tr>
<td>ZenithlumUncertainty</td>
<td>See [1], Table 3-4</td>
</tr>
<tr>
<td>TotCld</td>
<td>Amount of sky dome covered by clouds or obscuring phenomena at timestamp, tenths of sky</td>
</tr>
<tr>
<td>TotCldSource</td>
<td>See [1], Table 3-5, 8760x1 cell array of strings</td>
</tr>
<tr>
<td>TotCldUncertainty</td>
<td>See [1], Table 3-6</td>
</tr>
<tr>
<td>OpqCld</td>
<td>Amount of sky dome covered by clouds or obscuring phenomena that prevent observing the sky at timestamp, tenths of sky</td>
</tr>
<tr>
<td>OpqCldSource</td>
<td>See [1], Table 3-5, 8760x1 cell array of strings</td>
</tr>
<tr>
<td>OpqCldUncertainty</td>
<td>See [1], Table 3-6</td>
</tr>
<tr>
<td>DryBulb</td>
<td>Dry bulb temperature at the time indicated, in tenths of degree C (e.g. 352 = 35.2 C)</td>
</tr>
<tr>
<td>DryBulbSource</td>
<td>See [1], Table 3-5, 8760x1 cell array of strings</td>
</tr>
<tr>
<td>DryBulbUncertainty</td>
<td>See [1], Table 3-6</td>
</tr>
<tr>
<td>DewPoint</td>
<td>Dew-point temperature at the time indicated, in tenths of degree C (e.g. 76 = 7.6 C)</td>
</tr>
<tr>
<td>DewPointSource</td>
<td>See [1], Table 3-5, 8760x1 cell array of strings</td>
</tr>
<tr>
<td>DewPointUncertainty</td>
<td>See [1], Table 3-6</td>
</tr>
<tr>
<td>RHum</td>
<td>Relative humidity at the time indicated, percent</td>
</tr>
<tr>
<td>RHumSource</td>
<td>See [1], Table 3-5, 8760x1 cell array of strings</td>
</tr>
<tr>
<td>RHumUncertainty</td>
<td>See [1], Table 3-6</td>
</tr>
<tr>
<td>Pressure</td>
<td>Station pressure at the time indicated, 1 mbar</td>
</tr>
<tr>
<td>PressureSource</td>
<td>See [1], Table 3-5, 8760x1 cell array of strings</td>
</tr>
<tr>
<td>PressureUncertainty</td>
<td>See [1], Table 3-6</td>
</tr>
<tr>
<td>Wdir</td>
<td>Wind direction at the time indicated, degrees from east of north (360 = 0 = north; 90 = East; 0 = undefined, calm)</td>
</tr>
<tr>
<td>WdirSource</td>
<td>See [1], Table 3-5, 8760x1 cell array of strings</td>
</tr>
<tr>
<td>WdirUncertainty</td>
<td>See [1], Table 3-6</td>
</tr>
<tr>
<td>Wspd</td>
<td>Wind speed at the time indicated, in tenths of meters/second (e.g. 212 = 21.2 m/s)</td>
</tr>
</tbody>
</table>

2.6. Modules
## Table 2.1 – continued from previous page

<table>
<thead>
<tr>
<th>TMYData field</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WspdSource</td>
<td>See [1], Table 3-5, 8760x1 cell array of strings</td>
</tr>
<tr>
<td>WspdUncertainty</td>
<td>See [1], Table 3-6</td>
</tr>
<tr>
<td>Hvis</td>
<td>Distance to discernable remote objects at time indicated (7777=unlimited, 9999=missing data), in tenths of kilometers (e.g. 341 = 34.1 km).</td>
</tr>
<tr>
<td>HvisSource</td>
<td>See [1], Table 3-5, 8760x1 cell array of strings</td>
</tr>
<tr>
<td>HvisUncertainty</td>
<td>See [1], Table 3-6</td>
</tr>
<tr>
<td>CeilHgt</td>
<td>Height of cloud base above local terrain (7777=unlimited, 88888=cirroform, 99999=missing data), in meters</td>
</tr>
<tr>
<td>CeilHgtSource</td>
<td>See [1], Table 3-5, 8760x1 cell array of strings</td>
</tr>
<tr>
<td>CeilHgtUncertainty</td>
<td>See [1], Table 3-6</td>
</tr>
<tr>
<td>Pwat</td>
<td>Total precipitable water contained in a column of unit cross section from Earth to top of atmosphere, in millimeters</td>
</tr>
<tr>
<td>PwatSource</td>
<td>See [1], Table 3-5, 8760x1 cell array of strings</td>
</tr>
<tr>
<td>PwatUncertainty</td>
<td>See [1], Table 3-6</td>
</tr>
<tr>
<td>AOD</td>
<td>The broadband aerosol optical depth (broadband turbidity) in thousandths on the day indicated (e.g. 114 = 0.114)</td>
</tr>
<tr>
<td>AODSource</td>
<td>See [1], Table 3-5, 8760x1 cell array of strings</td>
</tr>
<tr>
<td>AODUncertainty</td>
<td>See [1], Table 3-6</td>
</tr>
<tr>
<td>SnowDepth</td>
<td>Snow depth in centimeters on the day indicated, (999 = missing data).</td>
</tr>
<tr>
<td>SnowDepthSource</td>
<td>See [1], Table 3-5, 8760x1 cell array of strings</td>
</tr>
<tr>
<td>SnowDepthUncertainty</td>
<td>Number of days since last snowfall (maximum value of 88, where 88 = 88 or greater days; 99 = missing data)</td>
</tr>
<tr>
<td>LastSnowfall</td>
<td>See [1], Table 3-5, 8760x1 cell array of strings</td>
</tr>
<tr>
<td>LastSnowfallSource</td>
<td>See [1], Table 3-5, 8760x1 cell array of strings</td>
</tr>
<tr>
<td>PresentWeather</td>
<td>See [1], Appendix B, an 8760x1 cell array of strings. Each string contains 10 numeric values. The string can be parsed to determine each of 10 observed weather metrics.</td>
</tr>
</tbody>
</table>

### References


```python
pVlib.tmy.readtmy3(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 discrepancies between this documentation and the TMY3 User’s Manual [1], the TMY3 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.

- `coerce_year` : None or int
  - If supplied, the year of the data will be set to this value.

- `recolumn` : bool
  - 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:

<table>
<thead>
<tr>
<th>key</th>
<th>format</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>altitude</td>
<td>Float</td>
<td>site elevation</td>
</tr>
<tr>
<td>latitude</td>
<td>Float</td>
<td>site latitude</td>
</tr>
<tr>
<td>longitude</td>
<td>Float</td>
<td>site longitude</td>
</tr>
<tr>
<td>Name</td>
<td>String</td>
<td>site name</td>
</tr>
<tr>
<td>State</td>
<td>String</td>
<td>state</td>
</tr>
<tr>
<td>TZ</td>
<td>Float</td>
<td>UTC offset</td>
</tr>
<tr>
<td>USAF</td>
<td>Int</td>
<td>USAF identifier</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TMYData field</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMYData.Index</td>
<td>A pandas datetime index. NOTE, the index is currently timezone unaware, and times are set to local standard time (daylight savings is not included)</td>
</tr>
<tr>
<td>TMYData.ETR</td>
<td>Extraterrestrial horizontal radiation received during 60 minutes prior to timestamp, Wh/m²</td>
</tr>
<tr>
<td>TMYData.ETRN</td>
<td>Extraterrestrial normal radiation received during 60 minutes prior to timestamp, Wh/m²</td>
</tr>
<tr>
<td>TMYData.GHI</td>
<td>Direct and diffuse horizontal radiation received during 60 minutes prior to timestamp, Wh/m²</td>
</tr>
<tr>
<td>TMYData.GHISource</td>
<td>See [1], Table 1-4</td>
</tr>
<tr>
<td>TMYData.GHIUncertainty</td>
<td>Uncertainty based on random and bias error estimates see [2]</td>
</tr>
<tr>
<td>TMYData.DNI</td>
<td>Amount of direct normal radiation (modeled) received during 60 minutes prior to timestamp, Wh/m²</td>
</tr>
<tr>
<td>TMYData.DNISource</td>
<td>See [1], Table 1-4</td>
</tr>
<tr>
<td>TMYData.DNIUncertainty</td>
<td>Uncertainty based on random and bias error estimates see [2]</td>
</tr>
<tr>
<td>TMYData.DHI</td>
<td>Amount of diffuse horizontal radiation received during 60 minutes prior to timestamp, Wh/m²</td>
</tr>
<tr>
<td>TMYData.DHISource</td>
<td>See [1], Table 1-4</td>
</tr>
<tr>
<td>TMYData.DHIUncertainty</td>
<td>Uncertainty based on random and bias error estimates see [2]</td>
</tr>
<tr>
<td>TMYData.GHillum</td>
<td>Avg. total horizontal illuminance received during the 60 minutes prior to timestamp, lx</td>
</tr>
<tr>
<td>TMYData.GHillumSource</td>
<td>See [1], Table 1-4</td>
</tr>
<tr>
<td>TMYData.GHillumUncertainty</td>
<td>Uncertainty based on random and bias error estimates see [2]</td>
</tr>
<tr>
<td>TMYData.DNillum</td>
<td>Avg. direct normal illuminance received during the 60 minutes prior to timestamp, lx</td>
</tr>
<tr>
<td>TMYData.DNillumSource</td>
<td>See [1], Table 1-4</td>
</tr>
<tr>
<td>TMYData.DNillumUncertainty</td>
<td>Uncertainty based on random and bias error estimates see [2]</td>
</tr>
<tr>
<td>TMYData.DHillum</td>
<td>Avg. horizontal diffuse illuminance received during the 60 minutes prior to timestamp, lx</td>
</tr>
<tr>
<td>TMYData.DHillumSource</td>
<td>See [1], Table 1-4</td>
</tr>
<tr>
<td>TMYData.DHillumUncertainty</td>
<td>Uncertainty based on random and bias error estimates see [2]</td>
</tr>
<tr>
<td>TMYData.Zenithlum</td>
<td>Avg. luminance at the sky’s zenith during the 60 minutes prior to timestamp, cd/m²</td>
</tr>
<tr>
<td>TMYData.ZenithlumSource</td>
<td>See [1], Table 1-4</td>
</tr>
<tr>
<td>TMYData.ZenithlumUncertainty</td>
<td>Uncertainty based on random and bias error estimates see [1] section 2.10</td>
</tr>
<tr>
<td>TMYData.TotCld</td>
<td>Amount of sky dome covered by clouds or obscuring phenomena at time stamp, tenths of sky</td>
</tr>
<tr>
<td>TMYData.TotCldSource</td>
<td>See [1], Table 1-5, 8760x1 cell array of strings</td>
</tr>
<tr>
<td>TMYData.TotCldUncertainty</td>
<td>See [1], Table 1-6</td>
</tr>
<tr>
<td>TMYData.OpqCld</td>
<td>Amount of sky dome covered by clouds or obscuring phenomena that prevent observing the sky at time stamp, tenths of sky</td>
</tr>
<tr>
<td>TMYData.OpqCldSource</td>
<td>See [1], Table 1-5, 8760x1 cell array of strings</td>
</tr>
<tr>
<td>TMYData.OpqCldUncertainty</td>
<td>See [1], Table 1-6</td>
</tr>
<tr>
<td>TMYData.DryBulb</td>
<td>Dry bulb temperature at the time indicated, deg C</td>
</tr>
<tr>
<td>TMYData.DryBulbSource</td>
<td>See [1], Table 1-5, 8760x1 cell array of strings</td>
</tr>
<tr>
<td>TMYData.DryBulbUncertainty</td>
<td>See [1], Table 1-6</td>
</tr>
<tr>
<td>TMYData.DewPoint</td>
<td>Dew-point temperature at the time indicated, deg C</td>
</tr>
<tr>
<td>TMYData.DewPointSource</td>
<td>See [1], Table 1-5, 8760x1 cell array of strings</td>
</tr>
</tbody>
</table>
Table 2.2 – continued from previous page

<table>
<thead>
<tr>
<th>TMYData field</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMYData.DewPointUncertainty</td>
<td>See [1], Table 1-6</td>
</tr>
<tr>
<td>TMYData.RHum</td>
<td>Relative humidity at the time indicated, percent</td>
</tr>
<tr>
<td>TMYData.RHumSource</td>
<td>See [1], Table 1-5, 8760x1 cell array of strings</td>
</tr>
<tr>
<td>TMYData.RHumUncertainty</td>
<td>See [1], Table 1-6</td>
</tr>
<tr>
<td>TMYData.Pressure</td>
<td>Station pressure at the time indicated, 1 mbar</td>
</tr>
<tr>
<td>TMYData.PressureSource</td>
<td>See [1], Table 1-5, 8760x1 cell array of strings</td>
</tr>
<tr>
<td>TMYData.PressureUncertainty</td>
<td>See [1], Table 1-6</td>
</tr>
<tr>
<td>TMYData.Wdir</td>
<td>Wind direction at time indicated, degrees from north (360 = north; 0 = undefined, calm)</td>
</tr>
<tr>
<td>TMYData.WdirSource</td>
<td>See [1], Table 1-5, 8760x1 cell array of strings</td>
</tr>
<tr>
<td>TMYData.WdirUncertainty</td>
<td>See [1], Table 1-6</td>
</tr>
<tr>
<td>TMYData.Wspd</td>
<td>Wind speed at the time indicated, meter/second</td>
</tr>
<tr>
<td>TMYData.WspdSource</td>
<td>See [1], Table 1-5, 8760x1 cell array of strings</td>
</tr>
<tr>
<td>TMYData.WspdUncertainty</td>
<td>See [1], Table 1-6</td>
</tr>
<tr>
<td>TMYData.Hvis</td>
<td>Distance to discernable remote objects at time indicated (7777=unlimited), meter</td>
</tr>
<tr>
<td>TMYData.HvisSource</td>
<td>See [1], Table 1-5, 8760x1 cell array of strings</td>
</tr>
<tr>
<td>TMYData.HvisUncertainty</td>
<td>See [1], Table 1-6</td>
</tr>
<tr>
<td>TMYData.CeilHgt</td>
<td>Height of cloud base above local terrain (7777=unlimited), meter</td>
</tr>
<tr>
<td>TMYData.CeilHgtSource</td>
<td>See [1], Table 1-5, 8760x1 cell array of strings</td>
</tr>
<tr>
<td>TMYData.CeilHgtUncertainty</td>
<td>See [1], Table 1-6</td>
</tr>
<tr>
<td>TMYData.Pwat</td>
<td>Total precipitable water contained in a column of unit cross section from earth to top of atmosphere, cm</td>
</tr>
<tr>
<td>TMYData.PwatSource</td>
<td>See [1], Table 1-5, 8760x1 cell array of strings</td>
</tr>
<tr>
<td>TMYData.PwatUncertainty</td>
<td>See [1], Table 1-6</td>
</tr>
<tr>
<td>TMYData.AOD</td>
<td>The broadband aerosol optical depth per unit of air mass due to extinction by aerosol component of atmosphere, unitless</td>
</tr>
<tr>
<td>TMYData.AODSource</td>
<td>See [1], Table 1-5, 8760x1 cell array of strings</td>
</tr>
<tr>
<td>TMYData.AODUncertainty</td>
<td>See [1], Table 1-6</td>
</tr>
<tr>
<td>TMYData.Alb</td>
<td>The ratio of reflected solar irradiance to global horizontal irradiance, unitless</td>
</tr>
<tr>
<td>TMYData.AlbSource</td>
<td>See [1], Table 1-5, 8760x1 cell array of strings</td>
</tr>
<tr>
<td>TMYData.AlbUncertainty</td>
<td>See [1], Table 1-6</td>
</tr>
<tr>
<td>TMYData.Lprecipdepth</td>
<td>The amount of liquid precipitation observed at indicated time for the period indicated in the liquid precipitation depth field, millimeter</td>
</tr>
<tr>
<td>TMYData.Lprecipquantity</td>
<td>The period of accumulation for the liquid precipitation depth field, hour</td>
</tr>
<tr>
<td>TMYData.LprecipSource</td>
<td>See [1], Table 1-5, 8760x1 cell array of strings</td>
</tr>
<tr>
<td>TMYData.LprecipUncertainty</td>
<td>See [1], Table 1-6</td>
</tr>
</tbody>
</table>

References


2.6.9 tracking

class pvlib.tracking.LocalizedSingleAxisTracker (pvsystem=None, location=None, **kwargs)

Bases: pvlib.tracking.SingleAxisTracker, pvlib.location.Location

Highly experimental.

class pvlib.tracking.SingleAxisTracker (axis_tilt=0, axis_azimuth=0, max_angle=90, back=track=True, gcr=0.2857142857142857, **kwargs)

Bases: pvlib.pvsystem.PVSystem
Inherits all of the PV modeling methods from PVSystem.

**get_irradiance**

```
def get_irradiance(dni, ghi, dhi, dni_extra=None, airmass=None, model='haydavies', **kwargs)
```

Uses the irradiance.total_irrad() 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**: float or Series. Extraterrestrial direct normal irradiance.
- **airmass**: float or Series. Airmass.
- **model**: String. Irradiance model.
- **kwargs**: Passed to irradiance.total_irrad().

**Returns** **poa_irradiance**: DataFrame. Column names are: total, beam, sky, ground.

**localize**

```
def 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.
- **latitude**: None or float.
- **longitude**: None or float.
- **kwargs**: see Location.

**Returns** **localized_system**: LocalizedSingleAxisTracker.

**singleaxis**

```
import pvlib

def singleaxis(apparent_zenith, apparent_azimuth)

def 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 using the equations in [1] when given a particular sun zenith and azimuth angle. 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 degrees 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 `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**: Series
  Solar apparent zenith angles in decimal degrees.

- **apparent_azimuth**: Series
  Solar apparent azimuth angles in decimal degrees.

- **axis_tilt**: float
  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
  A value denoting the compass direction along which the axis of rotation lies. Measured in decimal degrees East of North.

- **max_angle**: float
  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
  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
  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 $\leq 1$.

**Returns**

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.

- **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


2.6.10 tools

Collection of functions used in pvlib_python

\texttt{pvlib.tools.asind(number)}

Inverse Sine returning an angle in degrees

\textbf{Parameters} number : float

\textbf{Input number}

\textbf{Returns} result : float

\textbf{arcsin result}

\texttt{pvlib.tools.cosd(angle)}

Cosine with angle input in degrees

\textbf{Parameters} angle : float

\textbf{Angle in degrees}

\textbf{Returns} result : float

\textbf{Cosine of the angle}

\texttt{pvlib.tools.datetime_to_djd(time)}

Converts a datetime to the Dublin Julian Day

\textbf{Parameters} time : datetime.datetime

\textbf{time to convert}

\textbf{Returns} float

\textbf{fractional days since 12/31/1899+0000}

\texttt{pvlib.tools.djd_to_datetime(djd, tz='UTC')}

Converts a Dublin Julian Day float to a datetime.datetime object

\textbf{Parameters} djd : float

\textbf{fractional days since 12/31/1899+0000}

\texttt{tz} : str

\textbf{timezone to localize the result to}

\textbf{Returns} datetime.datetime

\textbf{The resultant datetime localized to tz}

\texttt{pvlib.tools.localize_to_utc(time, location)}

Converts or localizes a time series to UTC.

\textbf{Parameters} time : datetime.datetime, pandas.DatetimeIndex, or pandas.Series/DataFrame with a DatetimeIndex.

\textbf{location} : pvlib.Location object
Returns pandas object localized to UTC.

```
pvlib.tools.sind(angle)
Sine with angle input in degrees

Parameters angle : float
Angle in degrees

Returns result : float
Sin of the angle
```

```
pvlib.tools.tand(angle)
Tan with angle input in degrees

Parameters angle : float
Angle in degrees

Returns result : float
Tan of the angle
```

2.7 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.

2.7.1 Location

```
class pvlib.location.Location(latitude, longitude, tz='UTC', altitude=0, name=None, **kwargs)
Bases: object

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.
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.
```
Altitude from sea level in meters.

**name**: None or string.

Sets the name attribute of the Location object.

**kwargs**

Arbitrary keyword arguments. Included for compatibility, but not used.

See also:

classmethod 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

Optionally attach the TMY data to this object.

**Returns**

Location object (or the child class of Location that you called this method from).

get_airmass(times=None, solar_position=None, model='kastenyoung1989')

Calculate the relative and absolute airmass.

Automatically chooses zenith or apparent zenith depending on the selected model.

**Parameters**

times : None or DatetimeIndex

Only used if solar_position is not provided.

solar_position : None or DataFrame

DataFrame with with columns ‘apparent_zenith’, ‘zenith’.

model : str

Relative airmass model

**Returns**

airmass : DataFrame

Columns are ‘airmass_relative’, ‘airmass_absolute’

get_clearsky(times, model='ineichen', **kwargs)

Calculate the clear sky estimates of GHI, DNI, and/or DHI at this location.

**Parameters**

times : DatetimeIndex

model : str

The clear sky model to use.

**kwargs** passed to the relevant function(s).

**Returns**

clearsky : DataFrame

Column names are: ghi, dni, dhi.

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 : DatetimeIndex

pressure : None, float, or array-like

If None, pressure will be calculated using atmosphere.alt2pres() and self.altitude.

temperature : None, float, or array-like

kwarg s passed to :py:func:`solarposition.get_solarposition`

Returns solar_position : DataFrame

Columns depend on the method kwarg, but always include zenith and azimuth.

### 2.7.2 PVSystem

```python
class pvlib.pvsystem.PVSystem(surface_tilt=0, surface_azimuth=180, albedo=None, surface_type=None, module=None, module_parameters=None, series_modules=None, parallel_modules=None, inverter=None, inverter_parameters=None, racking_model='open_rack_cell_glassback', **kwargs)
```

Bases: object

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 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

Tilt angle of the module surface. Up=0, horizon=90.

surface_azimuth: float or array-like

Azimuth angle of the module surface. North=0, East=90, South=180, West=270.

albedo : None, float

The ground albedo. If None, will attempt to use surface_type and irradiance.SURFACE_ALBEDOS to lookup albedo.

surface_type : None, string

The ground surface type. See irradiance.SURFACE_ALBEDOS for valid values.

module : None, string

The model name of the modules. May be used to look up the module_parameters dictionary via some other method.

module_parameters : None, dict or Series

Module parameters as defined by the SAPM, CEC, or other.

inverter : None, string

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

Inverter parameters as defined by the SAPM, CEC, or other.

racking_model : None or string

Used for cell and module temperature calculations.

**kwargs

Arbitrary keyword arguments. Included for compatibility, but not used.

See also:

pvlbl.location.Location, pvlib.tracking.SingleAxisTracker, pvlib.pvsystem.LocalizedPVSystem

ashraeiam (aoi)

Determine the incidence angle modifier using self.module_parameters[‘b’], aoi, and the ashraeiam() function.

Parameters aoi : numeric

The angle of incidence in degrees.

Returns modifier : numeric

The AOI modifier.

calcparams_desoto (poa_global, temp_cell, **kwargs)

Use the calcparams_desoto() function, the input parameters and self.module_parameters to calculate the module currents and resistances.

Parameters poa_global : float or Series

The irradiance (in W/m^2) absorbed by the module.

temp_cell : float or Series

The average cell temperature of cells within a module in C.

**kwargs

See pvsystem.calcparams_desoto for details

Returns See pvsystem.calcparams_desoto for details

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

get_irradiance (solar_zenith, solar_azimuth, dni, ghi, dhi, dni_extra=None, airmass=None, model='haydavies', **kwargs)

Uses the irradiance.total_irrad() 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** : float or Series
Extraterrestrial direct normal irradiance

**airmass** : float or Series
Airmass

**model** : String
Irradiance model.

**kwargs**
Passed to `irradiance.total_irrad()`.

**Returns**
`poa_irradiance` : DataFrame
Column names are: total, beam, sky, ground.

**i_from_v** *(resistance_shunt, resistance_series, nNsVth, voltage, saturation_current, photocurrent)*
Wrapper around the `i_from_v()` function.

**Parameters**
See `pvsystem.i_from_v` for details

**Returns**
See `pvsystem.i_from_v` for details

**localize** *(location=None, latitude=None, longitude=None, **kwargs)*
Creates a LocalizedPVSystem object using this object and location data. Must supply either location object or latitude, longitude, and any location kwargs

**Parameters**
location : None or Location
latitude : None or float
longitude : None or float

**kwargs** : see Location

**Returns**
localized_system : LocalizedPVSystem

**physicaliam** *(aoi)*
Determine the incidence angle modifier using `self.module_parameters['K'], self.module_parameters['L'], self.module_parameters['n'], aoi, and the physicaliam()` function.

**Parameters**
See `pvsystem.physicaliam` for details

**Returns**
See `pvsystem.physicaliam` for details
sapm(\texttt{poa\_direct}, \texttt{poa\_diffuse}, \texttt{temp\_cell}, \texttt{airmass\_absolute}, \texttt{aoi}, **\texttt{kwargs})

Use the \texttt{sapm()} function, the input parameters, and \texttt{self.module\_parameters} to calculate \texttt{Voc}, \texttt{Isc}, \texttt{Ixx}, \texttt{Vmp/Imp}.

\textbf{Parameters} \texttt{poa\_direct} : Series

The direct irradiance incident upon the module (W/m^2).

\texttt{poa\_diffuse} : Series

The diffuse irradiance incident on module.

\texttt{temp\_cell} : Series

The cell temperature (degrees C).

\texttt{airmass\_absolute} : Series

Absolute airmass.

\texttt{aoi} : Series

Angle of incidence (degrees).

**\texttt{kwargs}

See \texttt{pvsystem.sapm} for details

\textbf{Returns} See \texttt{pvsystem.sapm} for details

\texttt{sapm\_celltemp(\texttt{irrad}, \texttt{wind}, \texttt{temp})}

Uses \texttt{sapm\_celltemp()} to calculate module and cell temperatures based on \texttt{self.racking\_model} and the input parameters.

\textbf{Parameters} See \texttt{pvsystem.sapm\_celltemp} for details

\textbf{Returns} See \texttt{pvsystem.sapm\_celltemp} for details

\texttt{singlediode(\texttt{photocurrent}, \texttt{saturation\_current}, \texttt{resistance\_series}, \texttt{resistance\_shunt}, \texttt{nNsVth})}

Wrapper around the \texttt{singlediode()} function.

\textbf{Parameters} See \texttt{pvsystem.singlediode} for details

\textbf{Returns} See \texttt{pvsystem.singlediode} for details

\texttt{snlinverter(\texttt{v\_dc}, \texttt{p\_dc})}

Uses \texttt{snlinverter()} to calculate AC power based on \texttt{self.inverter\_parameters} and the input parameters.

\textbf{Parameters} See \texttt{pvsystem.snlinverter} for details

\textbf{Returns} See \texttt{pvsystem.snlinverter} for details

2.7.3 ModelChain

\texttt{class} \texttt{pvlib.modelchain.ModelChain(\texttt{system}, \texttt{location}, \texttt{orientation\_strategy='south\_at\_latitude\_tilt'}, \texttt{clearsky\_model='ineichen'}, \texttt{transposition\_model='haydavies'}, \texttt{solar\_position\_method='nrel\_numpy'}, \texttt{airmass\_model='kastenyoung1989'}, **\texttt{kwargs})}

\textbf{Bases:} \texttt{object}

An experimental class that represents all of the modeling steps necessary for calculating power or energy for a PV system at a given location.

\textbf{Parameters} \texttt{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

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’.

**clearsky_model** : str

Passed to location.get_clearsky.

**transposition_model** : str

Passed to system.get_irradiance.

**solar_position_method** : str

Passed to location.get_solarposition.

**airmass_model** : str

Passed to location.get_airmass.

**kwargs

Arbitrary keyword arguments. Included for compatibility, but not used.

**orientation_strategy**

**run_model**(times, irradiance=None, weather=None)

Run the model.

**Parameters**

- **times** : DatetimeIndex
  - Times at which to evaluate the model.

- **irradiance** : None or DataFrame
  - If None, calculates clear sky data. Columns must be ‘dni’, ‘ghi’, ‘dhi’.

- **weather** : None or DataFrame
  - If None, assumes air temperature is 20 C and wind speed is 0 m/s. Columns must be ‘wind_speed’, ‘temp_air’.

**Returns**

- **self**
  - Assigns attributes: times, solar_position, airmass, irradiance, total_irrad, weather, temps, aoi, dc, ac

---

### 2.7.4 LocalizedPVSystem

**class** `pvlib.pvsystem.LocalizedPVSystem` *(pvsystem=None, location=None, **kwargs)*

**Bases:** `pvlib.pvsystem.PVSystem`, `pvlib.location.Location`

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.

See the **PVSystem** class for an object model that describes an unlocalized PV system.
2.7.5 SingleAxisTracker

class pvlib.tracking.SingleAxisTracker (axis_tilt=0, axis_azimuth=0, max_angle=90, back-
track=True, gcr=0.2857142857142857, **kwargs)

Bases: pvlib.pvsystem.PVSystem

Inherits all of the PV modeling methods from PVSystem.

get_irradiance (dni, ghi, dhi, dni_extra=None, airmass=None, model='haydavies', **kwargs)

Uses the irradiance.total_irrad() function to calculate the plane of array irradiance com-
ponents 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 : float or Series
   Extraterrestrial direct normal irradiance
 airmass : float or Series
   Airmass
 model : String
   Irradiance model.
 **kwargs
 Passed to irradiance.total_irrad().

Returns poa_irradiance : DataFrame
 Column names are: total, beam, sky, ground.

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
 latitude : None or float
 longitude : None or float
 **kwargs : see Location

Returns localized_system : LocalizedSingleAxisTracker

singleaxis (apparent_zenith, apparent_azimuth)
2.7.6 LocalizedSingleAxisTracker

class pvlib.tracking.LocalizedSingleAxisTracker (pvsystem=None, location=None, **kwargs)

    Bases: pvlib.tracking.SingleAxisTracker, pvlib.location.Location

    Highly experimental.

2.8 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 libraries, only similarity.

The PV_LIB Matlab help webpage is a good reference for this comparison.

2.8.1 Missing functions

See pvlib-python GitHub issue #2 for a list of functions missing from the Python version of the library.

2.8.2 Major differences

- pvlib-python uses git version control to track all changes to the code. A summary of changes is included in the whatstnew 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, PVSysstem, LocalizedPVSysstem, SingleAxisTracker, and ModelChain.
2.8.3 Other differences

- Very few tests of input validity 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 \texttt{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 \texttt{pytz.timezone} instead of an integer GMT offset.

- \texttt{clearsky.ineichen} supports interpolating monthly Linke Turbidities to daily resolution.

- Instead of requiring effective irradiance as an input, \texttt{pvsystem.sapm} calculates and returns it based on input POA irradiance, AM, and AOI.

- \texttt{pvlib-python} does not come with as much example data.

- \texttt{pvlib-python} does not currently implement as many algorithms as PVLIB MATLAB.

2.8.4 Documentation

- Using Sphinx to build the documentation, including dynamically created inline examples.

- Additional Jupyter tutorials in \texttt{/docs/tutorials}.

2.9 Variables and Symbols

There is a convention on consistent variable names throughout the library:

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
\textbf{variable} & \textbf{description} \\
\hline
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 \\
aoi\_projection & \text{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) \\
surface\_tilt & tilt angle of the surface \\
\hline
\end{tabular}
\caption{List of used Variables and Parameters}
\end{table}

Continued on next page

2.9. Variables and Symbols
### Table 2.3 – continued from previous page

<table>
<thead>
<tr>
<th>variable</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>surface_azimuth</td>
<td>azimuth angle of the surface</td>
</tr>
<tr>
<td>solar_zenith</td>
<td>zenith angle of the sun in degrees</td>
</tr>
<tr>
<td>apparent_zenith</td>
<td>refraction-corrected solar zenith angle in degrees</td>
</tr>
<tr>
<td>solar_azimuth</td>
<td>azimuth angle of the sun in degrees East of North</td>
</tr>
<tr>
<td>temp_cell</td>
<td>temperature of the cell</td>
</tr>
<tr>
<td>temp_module</td>
<td>temperature of the module</td>
</tr>
<tr>
<td>temp_air</td>
<td>temperature of the air</td>
</tr>
<tr>
<td>temp_dew</td>
<td>dewpoint temperature</td>
</tr>
<tr>
<td>relative_humidity</td>
<td>relative humidity</td>
</tr>
<tr>
<td>v_mp, i_mp, p_mp</td>
<td>module voltage, current, power at the maximum power point</td>
</tr>
<tr>
<td>v_oc</td>
<td>open circuit module voltage</td>
</tr>
<tr>
<td>i_sc</td>
<td>short circuit module current</td>
</tr>
<tr>
<td>i_x, i_xx</td>
<td>Sandia Array Performance Model IV curve parameters</td>
</tr>
<tr>
<td>effective_irradiance</td>
<td>effective irradiance</td>
</tr>
<tr>
<td>photocurrent</td>
<td>photocurrent</td>
</tr>
<tr>
<td>saturation_current</td>
<td>diode saturation current</td>
</tr>
<tr>
<td>resistance_series</td>
<td>series resistance</td>
</tr>
<tr>
<td>resistance_shunt</td>
<td>shunt resistance</td>
</tr>
<tr>
<td>transposition_factor</td>
<td>the gain ratio of the radiation on inclined plane to global horizontal irradiation: $\frac{p_{oa}}{ghi}$</td>
</tr>
</tbody>
</table>

For a definition and further explanation on the variables, common symbols and units refer to the following sources:

- Reference Variable List by PVPMC
- 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.
CHAPTER 3

Indices and tables

• genindex
• search
Python Module Index

p

pvlib.atmosphere, 33
pvlib.clearsky, 35
pvlib.irradiance, 37
pvlib.location, 49
pvlib.modelchain, 51
pvlib.pvsystem, 53
pvlib.solarposition, 69
pvlib.tmy, 74
pvlib.tools, 81
pvlib.tracking, 78
<table>
<thead>
<tr>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
</tbody>
</table>
absoluteairmass() (in module pvlib.atmosphere), 33  
alt2pres() (in module pvlib.atmosphere), 33  
aoi() (in module pvlib.irradiance), 37  
aoi_projection() (in module pvlib.irradiance), 37  
ashraeiam() (in module pvlib.pvsystem), 57  
ashraeiam() (pvlib.pvsystem.PVSystem method), 54, 85  
asind() (in module pvlib.tools), 81  
B     |
basic_chain() (in module pvlib.modelchain), 52  
beam_component() (in module pvlib.irradiance), 38  
C     |
calc_time() (in module pvlib.solarposition), 69  
calcparams_desoto() (in module pvlib.pvsystem), 57  
calcparams_desoto() (pvlib.pvsystem.PVSystem method), 55, 85  
cosd() (in module pvlib.tools), 81  
D     |
datetime_to_djd() (in module pvlib.tools), 81  
dirint() (in module pvlib.irradiance), 38  
disc() (in module pvlib.irradiance), 39  
djd_to_datetime() (in module pvlib.tools), 81  
E     |
ephemeris() (in module pvlib.solarposition), 69  
extraradiation() (in module pvlib.irradiance), 40  
F     |
from_tmy() (pvlib.location.Location class method), 50, 83  
G     |
get_airmass() (pvlib.location.Location method), 50, 83  
get_aoi() (pvlib.pvsystem.PVSystem method), 55, 85  
get_clearsky() (pvlib.location.Location method), 50, 83  
get_irradiance() (pvlib.pvsystem.PVSystem method), 55, 85  
get_irradiance() (pvlib.tracking.SingleAxisTracker method), 79, 89  
get_orientation() (in module pvlib.modelchain), 53  
get_solarposition() (in module pvlib.solarposition), 70  
get_solarposition() (pvlib.location.Location method), 50, 83  
get_sunrise_set_transit() (in module pvlib.solarposition), 71  
globalinplane() (in module pvlib.irradiance), 40  
grounddiffuse() (in module pvlib.irradiance), 41  
H     |
haurwitz() (in module pvlib.clearsky), 35  
haydavies() (in module pvlib.irradiance), 42  
I     |
i_from_v() (in module pvlib.pvsystem), 60  
i_from_v() (pvlib.pvsystem.PVSystem method), 56, 86  
ieichen() (in module pvlib.clearsky), 36  
isotropic() (in module pvlib.irradiance), 43  
K     |
king() (in module pvlib.irradiance), 43  
klucher() (in module pvlib.irradiance), 44  
L     |
localize() (pvlib.pvsystem.PVSystem method), 56, 86  
localize() (pvlib.tracking.SingleAxisTracker method), 79, 89  
localize_to_utc() (in module pvlib.tools), 81  
LocalizedPVSystem (class in pvlib.pvsystem), 53, 88  
LocalizedSingleAxisTracker (class in pvlib.tracking), 78, 90  
Location (class in pvlib.location), 49, 82  
lookup_linke_turbidity() (in module pvlib.clearsky), 37  
M     |
ModelChain (class in pvlib.modelchain), 51, 87  

97
O
orientation_strategy (pvlib.modelchain.ModelChain attribute), 51, 88

P
perez() (in module pvlib.irradiance), 45
physicaliam() (in module pvlib.pvsystem), 61
physicaliam() (pvlib.pvsystem.PVSystem method), 56, 86
poa_horizontal_ratio() (in module pvlib.irradiance), 46
pres2alt() (in module pvlib.atmosphere), 34
pvlib.atmosphere (module), 33
pvlib.clearsky (module), 35
pvlib.irradiance (module), 37
pvlib.location (module), 49
pvlib.modelchain (module), 51
pvlib.pvsystem (module), 53
pvlib.solarposition (module), 69
pvlib.tmy (module), 74
pvlib.tools (module), 81
pvlib.tracking (module), 78
PVSystem (class in pvlib.pvsystem), 53, 84
pyephem() (in module pvlib.solarposition), 71
pyephem_earthsun_distance() (in module pvlib.solarposition), 72

R
readtmy2() (in module pvlib.tmy), 74
readtmy3() (in module pvlib.tmy), 76
reindl() (in module pvlib.irradiance), 47
relativeairmass() (in module pvlib.atmosphere), 34
retrieve_sam() (in module pvlib.pvsystem), 62
run_model() (pvlib.modelchain.ModelChain method), 51, 88

S
sapm() (in module pvlib.pvsystem), 63
sapm() (pvlib.pvsystem.PVSystem method), 56, 86
sapm_celltemp() (in module pvlib.pvsystem), 64
sapm_celltemp() (pvlib.pvsystem.PVSystem method), 56, 87
sind() (in module pvlib.tools), 82
singleaxis() (in module pvlib.tracking), 79
singleaxis() (pvlib.tracking.SingleAxisTracker method), 79, 89
SingleAxisTracker (class in pvlib.tracking), 78, 89
singlydiode() (in module pvlib.pvsystem), 65
singlydiode() (pvlib.pvsystem.PVSystem method), 57, 87
snlinverter() (in module pvlib.pvsystem), 66
snlinverter() (pvlib.pvsystem.PVSystem method), 57, 87
spa_c() (in module pvlib.solarposition), 72
spa_python() (in module pvlib.solarposition), 73
systemdef() (in module pvlib.pvsystem), 68

tand() (in module pvlib.tools), 82
total_irrad() (in module pvlib.irradiance), 48