vf_model Documentation

Release 0+untagged.50.gc8411bf

Marc Anoma

Jul 01, 2023

CONTENTS

pvfactors is an open-source Python library that makes it easy to calculate incident irradiance on various surfaces of a PV array, including back side PV surfaces. pvfactors was originally ported from the SunPower developed vf_model package which was first presented at the IEEE PV Specialist Conference 44 ([1], link to paper).

You can find explanations on how to install the package in the *Installation* section, and learn how to use it using both the *Tutorials* and *Developer API* sections, but preferably after reading the *Main concepts* section.

[1] Anoma, M., Jacob, D., Bourne, B.C., Scholl, J.A., Riley, D.M. and Hansen, C.W., 2017. View Factor Model and Validation for Bifacial PV and Diffuse Shade on Single-Axis Trackers. In 44th IEEE Photovoltaic Specialist Conference.

CHAPTER ONE

CITING PVFACTORS

We appreciate your use of `pvfactors`. If you use `pvfactors` in a published work, we kindly ask that you cite:

> Anoma, M., Jacob, D., Bourne, B.C., Scholl, J.A., Riley, D.M. and Hansen, C.W., 2017. View Factor Model and Validation for Bifacial PV and Diffuse Shade on Single-Axis Trackers. In 44th IEEE Photovoltaic Specialist Conference.

CHAPTER

TWO

CONTENTS

2.1 Installation

2.1.1 Install with pip

`pvfactors` currently supports python 3.6+.

The easiest way to install `pvfactors` is using pip:

```
$ pip install pvfactors
```

However, installing `shapely` from PyPI may not install all the necessary binary dependencies. If you run into an error like `OSError: [WinError 126] The specified module could not be found`, try installing conda from conda-forge with:

```
$ conda install -c conda-forge shapely
```

Windows users may also be able to resolve the issue by installing wheels from Christoph Gohlke.

2.1.2 pvlib implementation

A limited implementation of `pvfactors` is available in the bifacial module of pvlib-python: see here.

2.1.3 Contributing

Contributions are needed in order to improve this package. If you wish to contribute, you can start by forking and cloning the repository, and then installing `pvfactors` using pip in the root folder of the package:

```
$ pip install .
```

To install the package in editable mode, you can use:

```
$ pip install -e .
```

2.2 Main concepts

Understanding pvfactors is simple. The pvfactors package builds on top of 3 distinct blocks that allow a clear workflow in the calculation of irradiance, while keeping the complexity separated for the different aspects of modeling. The schematics below shows what these three blocks are.

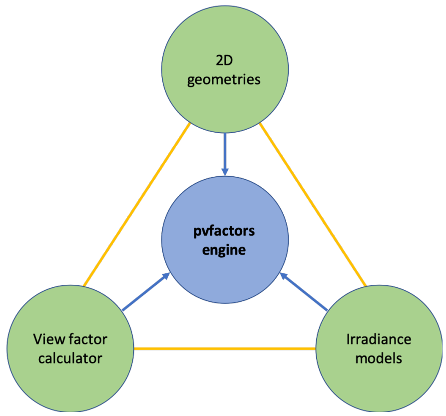


Fig. 1: Fig. 1: The 3 building blocks of pvfactors

In pvfactors, everything starts with the 2D geometry of the PV array, and everything flows from there.

- The user will use the *geometry API* to not only build the PV array geometry to be modeled, but also to get the results after the simulation.
- A selected (or custom made) irradiance model can then be used to define the sky irradiance components that are incident on the surfaces. For instance in Fig. 2, the front surfaces of the PV rows are receiving direct sunlight, while their back surfaces aren't receiving any. This is an example of what the irradiance model will define for all the surfaces.
- A calculator can then be used to calculate a matrix of view factors between all the different surfaces.

Finally, these 3 blocks will be assembled together inside the pvfactors engine (see `PVEngine`) to solve the irradiance mathematical system described in the paper and in the *theory section*.

2.2.1 2D geometries

The main interface for building the 2D geometry of a PV array is currently the `OrderedPVArray` class. It can be used for modeling both fixed tilt and single-axis tracker systems on a flat ground. Here are some details on the concepts behind the `OrderedPVArray` class.

Note: For more information on how the geometry sub-package is organized, the user can refer to the *detailed geometry API*.

Understanding PV array 2D geometries

Let's start with an example of a PV array 2D geometry plotted with `pvfactors`.

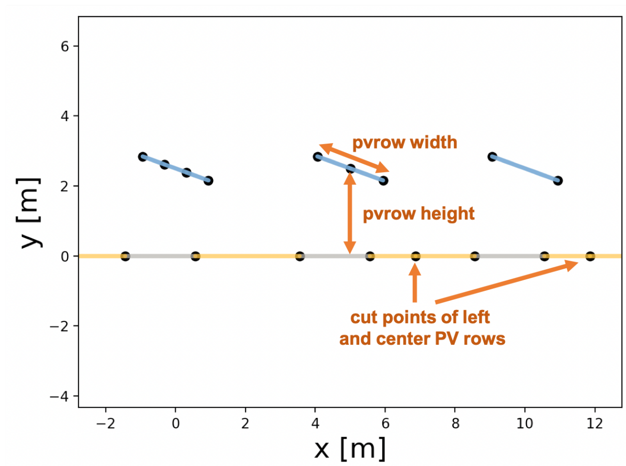


Fig. 2: Fig. 2: Example of PV array 2D geometry in `pvfactors`

As shown in the figure above, a `pvfactors` PV array is made out of a list of PV rows (the tilted blue lines), and a ground (the flat lines at y=0).

The PV rows:

- each PV row has 2 sides: a front and a back side
- each side of a PV row is made out of segments. The segments are fixed sections whose location on the PV row side is always constant throughout the simulations, which allows the users to consistently track and calculate irradiance for given sections of a PV row side
- each segment of each side of the PV rows is made out of collections of surfaces that are either shaded or illuminated, and these surfaces' size and length change during the simulation because they depend on the PV row rotation angles and the sun's position.

Note: In Fig. 2, the leftmost PV row's front side has 3 segments, while its back side has only 1. And the center PV row's back side has 2 segments, while its front side has only 1, etc.

The ground:

- it is made out of shaded surfaces (gray lines) and illuminated ones (yellow lines)
- the size and length of the ground surfaces will change with the PV row rotation and the sun angles. Physically, the shaded surfaces represent the shadows of the PV rows that are cast on the ground.
- the ground will also keep track of "cut points", which are defined by the PV rows (1 per PV row), and which indicate the extent of the ground that a PV row front side and back side can see.

Note: In Fig. 2, we can see 3 ground shadows, and the figure also shows 2 cut points (but there is a 3rd one located outside of the figure range on the right side).

PV array parameters

In pvfactors, a PV array has a number of fixed parameters that do not change with rotation and solar angles, and which can be passed as a dictionary with specific field names. Below is a sample of a PV array parameters dictionary, which was used to create the 2D geometry shown in Fig. 2.

```
pvarray_parameters = {
    'n_pvrows': 3,                          # number of pv rows
    'pvrow_height': 2.5,                    # height of pv rows (measured at center /␣
↪torque tube)
    'pvrow_width': 2,                       # width of pvrows
    'axis_azimuth': 0.,                     # azimuth angle of rotation axis
    'gcr': 0.4,                             # ground coverage ratio
    'cut': {0: {'front': 3}, 1: {'back': 2}} # discretization scheme of the pv rows
}
```

The *tutorial section* shows how such a dictionary can be used to create a PV array in pvfactors using the *OrderedPVArray* class. Here is a description of what each parameter means:

- n_pvrows: is the number of PV rows that the PV array will contain. In Fig. 2, we have 3 PV rows.
- pvrow_height: the PV row height (in meters) is the height of the PV row measured from the ground to the PV row center. In Fig. 2, the height of the PV rows is 2.5 m.
- pvrow_width: the PV row width (in meters) is the cross-section width of the entire PV row. In Fig. 2, it's the entire length of the blue lines, so 2 m in the example.
- axis_azimuth: the PV array axis azimuth (in degrees) is the direction of the rotation axis of the PV rows (physically, it could be seen as the torque tube direction for single-axis trackers). The azimuth convention used in pvfactors is that 0 deg is North, 90 deg is East, etc. In the 2D plane of the PV array geometry (as shown in Fig. 2), the axis of rotation is always the vector normal to that 2D plane and with the direction going into the 2D plane. So **positive rotation angles will lead to PV rows tilted to the left, and negative rotation angles will lead to PV rows tilted to the right**.
- gcr: it is the ground coverage ratio of the PV array. It is calculated as being equal to the ratio of the PV row width by the distance separating the PV row centers.
- cut: this optional parameter is used to discretize the PV row sides into equal-length segments. For instance here, the front side of the leftmost PV row (always with index 0) will have 3 segments, and the back side of the center PV row (with index 1) will have 2 segments.

2.2.2 Irradiance models

The irradiance models then assign irradiance sky values like direct, or circumsolar components to all the surfaces defined in the *OrderedPVArray*.

Description

As shown in the *full mode theory* and *fast mode theory* sections, we always need to calculate a sky term for the different surfaces of the PV array.

The sky term is the sum of all the irradiance components (for each surface) that are not directly related to the view factors or to the reflection process, but which still contribute to the incident irradiance on the surfaces. For instance, the direct component of the light incident on the front surface of a PV row is not directly dependent on the view factors, but we still need to account for it in the mathematical model, so this component will go into the sky term.

A lot of different assumptions can be made, which will lead to more or less accurate results. But `pvfactors` was designed to make the implementation of these assumptions modular: all of these assumptions can be implemented inside a single Python class which can be used by the other parts of the model. This was done to make it easy for users to create their own irradiance modeling assumptions (inside a new class), and to then plug it into the `pvfactors` *PVEngine*.

Available models

`pvfactors` currently provides two irradiance models that can be used interchangeably in the *PVEngine* and with the *OrderedPVArray*, and they are described in more details in the irradiance developer API.

- the isotropic model *IsotropicOrdered* assumes that all of the diffuse light from the sky dome is isotropic. It is a very intuitive assumption, but it generally leads to less accurate results.
- the (hybrid) perez model *HybridPerezOrdered* follows[1] and assumes that the diffuse light can be broken down into circumsolar, isotropic, and horizon components (see Fig. 3 below). Validation work shows that this model is more accurate for calculating back-side irradiance with `pvfactors`.

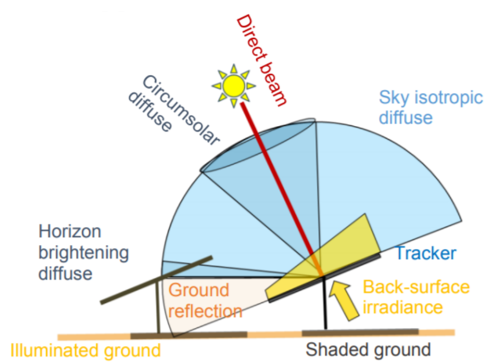


Fig. 3: Fig. 3: Schematic showing direct and diffuse irradiance components on a PV system and according to the Perez diffuse light model[1]

[1] Perez, R., Seals, R., Ineichen, P., Stewart, R. and Menicucci, D., 1987. A new simplified version of the Perez diffuse irradiance model for tilted surfaces. Solar energy, 39(3), pp.221-231.

2.2.3 View factor calculator

After creating a 2D geometry, the `VFCalculator` class can be used to calculate the view factors between all the surfaces of the array. A detailed description of what view factors are can be found in the *theory section*.

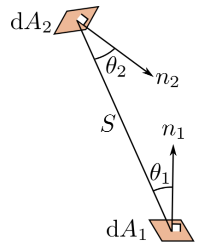


Fig. 4: Fig. 4: The view factor from a surface 1 to a surface 2 is the proportion of the space occupied by surface 2 in the hemisphere seen by surface 1.

2.2.4 Next steps

- *get started* using practical tutorials
- learn more about the *theory* behind `pvfactors`
- dive into the *developer API*

2.3 Tutorials

This section will cover some tutorials to help the users easily get started with `pvfactors`. The notebooks used for this section are all located in the tutorials folder of the Github repository.

Note: The users may find it useful to first read the theory and mathematical formulation for *Full simulations* and *Fast simulations* to better understand the differences between the two approaches.

2.3.1 Getting started: running simulations

Here is a quick overview on how to get started and run irradiance simulations with `pvfactors`.

Getting started

This is a quick overview of multiple capabilities of `pvfactors`:

- create a PV array
- use the engine to update the PV array
- plot the PV array 2D geometry for a given timestamp index
- run a timeseries bifacial simulation using the “full mode”
- run a timeseries bifacial simulation using the “fast mode”

Imports and settings

[1]:
```
# Import external libraries
import numpy as np
import matplotlib.pyplot as plt
from datetime import datetime
import pandas as pd
import warnings
warnings.filterwarnings("ignore", category=RuntimeWarning)

# Settings
%matplotlib inline
np.set_printoptions(precision=3, linewidth=300)
```

Get timeseries inputs

[2]:
```
df_inputs = pd.DataFrame(
    {'solar_zenith': [20., 50.],
     'solar_azimuth': [110., 250.],
     'surface_tilt': [10., 20.],
     'surface_azimuth': [90., 270.],
     'dni': [1000., 900.],
     'dhi': [50., 100.],
     'albedo': [0.2, 0.2]},
    index=[datetime(2017, 8, 31, 11), datetime(2017, 8, 31, 15)]
)
df_inputs
```

[2]:
```
                     solar_zenith  solar_azimuth  surface_tilt  \
2017-08-31 11:00:00          20.0          110.0          10.0
2017-08-31 15:00:00          50.0          250.0          20.0

                     surface_azimuth     dni    dhi  albedo
2017-08-31 11:00:00             90.0  1000.0   50.0     0.2
2017-08-31 15:00:00            270.0   900.0  100.0     0.2
```

Prepare some PV array parameters

[3]:
```
pvarray_parameters = {
    'n_pvrows': 3,            # number of pv rows
    'pvrow_height': 1,        # height of pvrows (measured at center / torque tube)
    'pvrow_width': 1,         # width of pvrows
    'axis_azimuth': 0.,       # azimuth angle of rotation axis
    'gcr': 0.4,               # ground coverage ratio
}
```

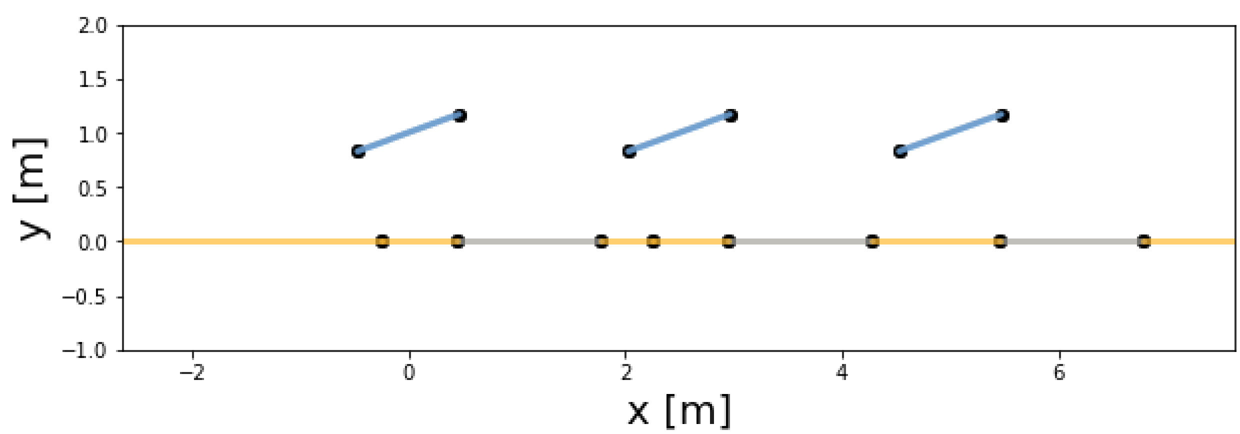
Create a PV array and update it with the engine

Use the PVEngine and the OrderedPVArray to run simulations

[4]:
```
from pvfactors.engine import PVEngine
from pvfactors.geometry import OrderedPVArray

# Create an ordered PV array
pvarray = OrderedPVArray.init_from_dict(pvarray_parameters)
# Create engine using the PV array
engine = PVEngine(pvarray)
# Fit engine to data: which will update the pvarray object as well
engine.fit(df_inputs.index, df_inputs.dni, df_inputs.dhi,
           df_inputs.solar_zenith, df_inputs.solar_azimuth,
           df_inputs.surface_tilt, df_inputs.surface_azimuth,
           df_inputs.albedo)
```

The user can then plot the PV array 2D geometry for any of the simulation timestamp

[5]:
```
# Plot pvarray shapely geometries
f, ax = plt.subplots(figsize=(10, 3))
pvarray.plot_at_idx(1, ax)
plt.show()
```



Run simulation using the full mode

The "full mode" allows the user to run the irradiance calculations by accounting for the equilibrium of reflections between all the surfaces in the system. So it is more precise than the "fast mode", and it happens to be almost as fast.

[6]:
```
# Create a function that will build a report from the simulation and return the
# incident irradiance on the back surface of the middle PV row
def fn_report(pvarray): return pd.DataFrame({'qinc_back': pvarray.ts_pvrows[1].back.get_
↪param_weighted('qinc')})

# Run full mode simulation
report = engine.run_full_mode(fn_build_report=fn_report)
```

[7]:
```python
# Print results (report is defined by report function passed by user)
df_report_full = report.assign(timestamps=df_inputs.index).set_index('timestamps')

print('Incident irradiance on back surface of middle PV row: \n')
df_report_full
```

```
Incident irradiance on back surface of middle PV row:
```

[7]:
```
                     qinc_back
timestamps
2017-08-31 11:00:00  106.627832
2017-08-31 15:00:00   79.668878
```

Run simulation using the fast mode

The "fast mode" allows the user to get slightly faster but less accurate results for the incident irradiance on the back surface of a single PV row. It assumes that the incident irradiance values on surfaces other than back surfaces are known (e.g. from the Perez transposition model).

[8]:
```python
# Run the fast mode calculation on the middle PV row: use the same report function as␣
↪previously
df_report_fast = engine.run_fast_mode(fn_build_report=fn_report, pvrow_index=1)

# Print the results
print('Incident irradiance on back surface of middle PV row: \n')
df_report_fast
```

```
Incident irradiance on back surface of middle PV row:
```

[8]:
```
                     qinc_back
2017-08-31 11:00:00  107.934226
2017-08-31 15:00:00   83.495861
```

We can observe here some differences between the fast and full modes for the back surface total irradiance, which are mainly due to the difference in how reflections are accounted for.

2.3.2 Details on the "full mode" simulations

In the "full mode", `pvfactors` calculates the equilibrium of reflections between all surfaces of the PV array for each timestamp. So the system to solve is implicit (matrix inversion required).

`pvfactors` relies on "timeseries geometries" of the PV array, which are the attributes named `ts_pvrows` and `ts_ground` in *OrderedPVArray*, and which contain vectors of coordinates for all timestamps and for all geometry elements. Please take a look at the tutorial sections below for more details on this.

PV Array geometry introduction

In this section, we will learn how to:

- create a 2D PV array geometry with PV rows at identical heights, tilt angles, and with identical widths
- plot that PV array
- calculate the inter-row direct shading, and get the length of the shadows on the PV rows
- understand what timeseries geometries are, including `ts_pvrows` and `ts_ground`

Imports and settings

[1]:
```
# Import external libraries
import matplotlib.pyplot as plt

# Settings
%matplotlib inline
```

Prepare PV array parameters

[2]:
```
pvarray_parameters = {
    'n_pvrows': 4,            # number of pv rows
    'pvrow_height': 1,        # height of pvrows (measured at center / torque tube)
    'pvrow_width': 1,         # width of pvrows
    'axis_azimuth': 0.,       # azimuth angle of rotation axis
    'surface_tilt': 20.,      # tilt of the pv rows
    'surface_azimuth': 90.,   # azimuth of the pv rows front surface
    'solar_zenith': 40.,      # solar zenith angle
    'solar_azimuth': 150.,    # solar azimuth angle
    'gcr': 0.5,               # ground coverage ratio
}
```

Create a PV array and its shadows

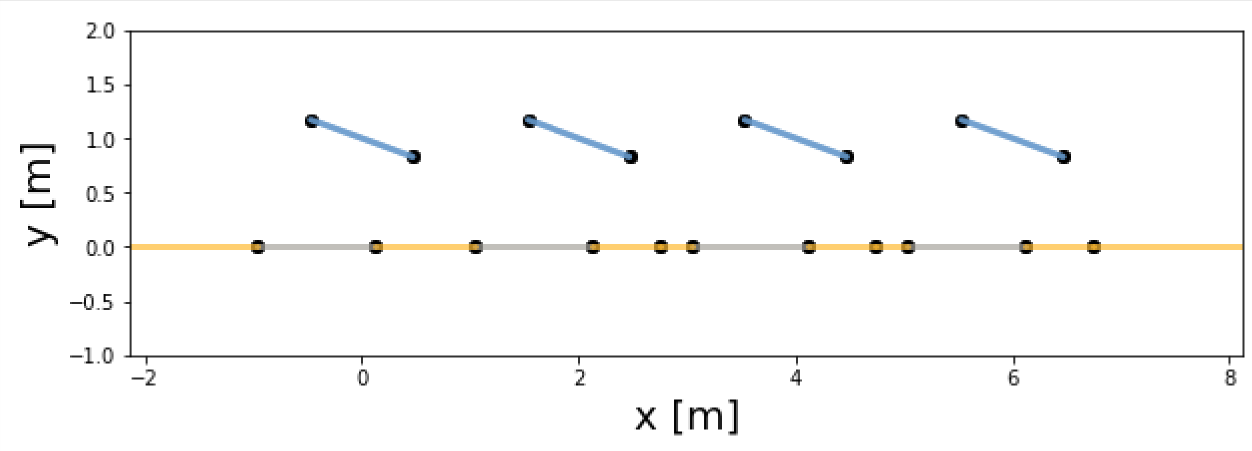
Import the `OrderedPVArray` class and create a transformed PV array object using the parameters above

[3]:
```
from pvfactors.geometry import OrderedPVArray

pvarray = OrderedPVArray.fit_from_dict_of_scalars(pvarray_parameters)
```

Plot the PV array.

Note: the index `0` is passed to the plotting method. We're explaining why a little later in this tutorial.

[4]:
```
# Plot pvarray shapely geometries
f, ax = plt.subplots(figsize=(10, 3))
pvarray.plot_at_idx(0, ax)
plt.show()
```



As we can see in the plot above: - the blue lines represent the PV rows - the gray lines represent the shadows cast by the PV rows on the ground from direct light - the yellow lines represent the ground areas that don't get any direct shading - there are additional points on the ground that may seem out of place: but they are called "cut points" and are necessary to calculate view factors. For instance, if you take the cut point located between the second and third shadows (counting from the left), it marks the point after which the leftmost PV row's back side is not able to see the ground anymore

Situation with direct shading

We can also create situations where direct shading happens either on the front or back surface of the PV rows.

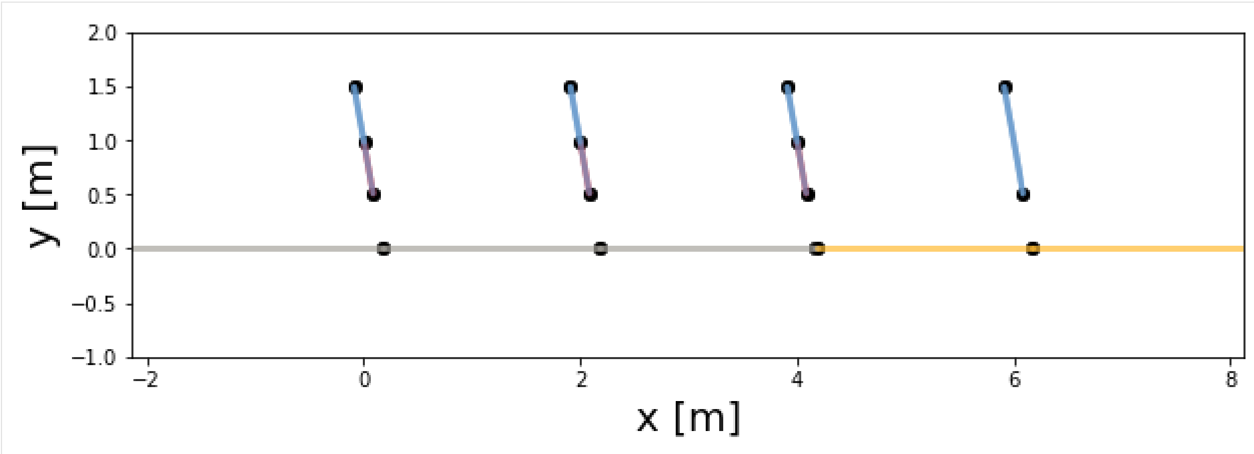
[5]:
```
# New configuration with direct shading
pvarray_parameters.update({'surface_tilt': 80., 'solar_zenith': 75., 'solar_azimuth': 90.
→})
```

[6]:
```
pvarray_parameters
```

[6]:
```
{'n_pvrows': 4,
 'pvrow_height': 1,
 'pvrow_width': 1,
 'axis_azimuth': 0.0,
 'surface_tilt': 80.0,
 'surface_azimuth': 90.0,
 'solar_zenith': 75.0,
 'solar_azimuth': 90.0,
 'gcr': 0.5}
```

[7]:
```
# Create new PV array
pvarray_w_direct_shading = OrderedPVArray.fit_from_dict_of_scalars(pvarray_parameters)
```

[8]:
```
# Plot pvarray shapely geometries
f, ax = plt.subplots(figsize=(10, 3))
pvarray_w_direct_shading.plot_at_idx(0, ax)
plt.show()
```



We can now see on the plot above that some inter-row shading is happening in the PV array.
It is also very easy to obtain the shadow length on the front surface of the shaded PV rows.

[9]:
```
# Shaded length on first pv row (leftmost)
l = pvarray_w_direct_shading.ts_pvrows[0].front.shaded_length
print("Shaded length on front surface of leftmost PV row: %.2f m" % l)
```

```
Shaded length on front surface of leftmost PV row: 0.48 m
```

[10]:
```
# Shaded length on last pv row (rightmost)
l = pvarray_w_direct_shading.ts_pvrows[-1].front.shaded_length
print("Shaded length on front surface of rightmost PV row: %.2f m" %l)
```

```
Shaded length on front surface of rightmost PV row: 0.00 m
```

As we can see, the rightmost PV row is not shaded at all.

What are timeseries geometries?

It is important to note that the two most important attributes of the PV array object are `ts_pvrows` and `ts_ground`. These contain what we call "timeseries geometries", which are objects that represent the geometry of the PV rows and the ground for all timestamps of the simulation.

For instance here, we can look at the coordinates of the front illuminated timeseries surface of the leftmost PV row.

[11]:
```
front_illum_ts_surface = pvarray_w_direct_shading.ts_pvrows[0].front.list_segments[0].
↪illum.list_ts_surfaces[0]
```

[12]:
```
coords = front_illum_ts_surface.coords
print("Coords: {}".format(coords))
```

```
Coords: [[[ 0.00340618]
  [ 0.98068262]]

 [[-0.08682409]
  [ 1.49240388]]]
```

These are the timeseries **line coordinates** of the surface, and it is made out of two timeseries **point coordinates**, `b1` and `b2` ("b" for boundary).

[13]:
```
b1 = coords.b1
b2 = coords.b2
print("b1 coords: {}".format(b1))
```

```
b1 coords: [[0.00340618]
 [0.98068262]]
```

Each timeseries point is also made of `x` and `y` timeseries coordinates, which are just numpy arrays.

[14]:
```
print("x coords of b1: {}".format(b1.x))
print("y coords of b1: {}".format(b1.y))
```

```
x coords of b1: [0.00340618]
y coords of b1: [0.98068262]
```

The `x` and `y` coordinates will be numpy arrays of all the values the coordinates take for all the simulation timestamps, as calculated at `fit()` time of the PV array object. This also explain why we needed to specify the index `0` when plotting the PV array: this was to select the coordinates for the first (and only) timestamp.

Discretize PV row sides and indexing

In this section, we will learn how to:

- create a PV array with discretized PV row sides
- understand the indices of the timeseries surfaces of a PV array
- plot a PV array with indices shown on plot

Imports and settings

[1]:
```
# Import external libraries
import matplotlib.pyplot as plt

# Settings
%matplotlib inline
```

Prepare PV array parameters

[2]:
```
pvarray_parameters = {
    'n_pvrows': 3,            # number of pv rows
    'pvrow_height': 1,        # height of pvrows (measured at center / torque tube)
    'pvrow_width': 1,         # width of pvrows
    'axis_azimuth': 0.,       # azimuth angle of rotation axis
    'surface_tilt': 20.,      # tilt of the pv rows
    'surface_azimuth': 270.,   # azimuth of the pv rows front surface
    'solar_zenith': 40.,      # solar zenith angle
    'solar_azimuth': 150.,    # solar azimuth angle
    'gcr': 0.5,               # ground coverage ratio
}
```

Create discretization scheme

[3]:
```
discretization = {'cut':{
    0: {'back': 5},  # discretize the back side of the leftmost PV row into 5 segments
    1: {'front': 3}  # discretize the front side of the center PV row into 3 segments
}}
pvarray_parameters.update(discretization)
```

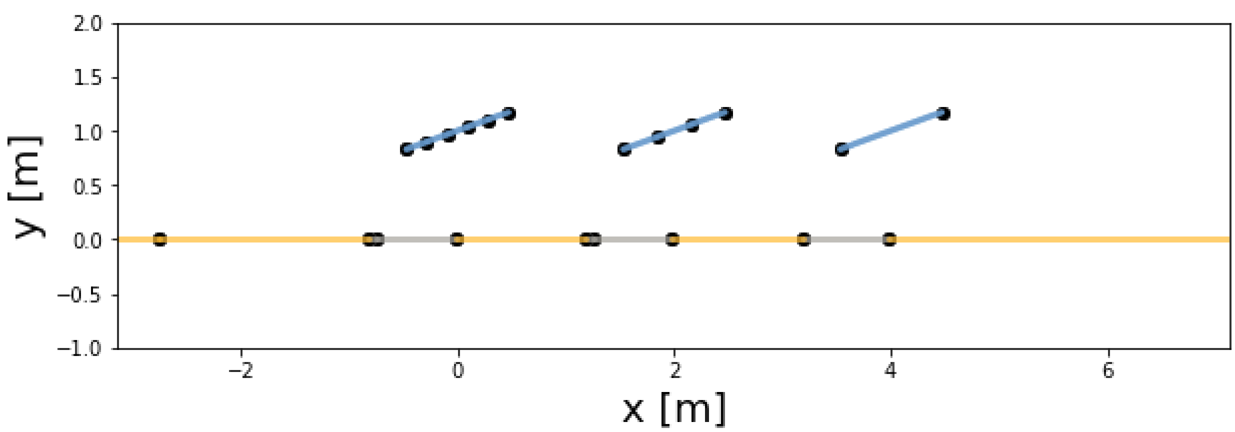
Create a PV array

Import the `OrderedPVArray` class and create a PV array object using the parameters above

[4]:
```
from pvfactors.geometry import OrderedPVArray
# Create pv array
pvarray = OrderedPVArray.fit_from_dict_of_scalars(pvarray_parameters)
```

Plot the PV array at index `0`

[5]:
```
# Plot pvarray shapely geometries
f, ax = plt.subplots(figsize=(10, 3))
pvarray.plot_at_idx(0, ax)
plt.show()
```



As we can see, there is some discretization on the leftmost and the center PV rows.
We can check that it was correctly done using the `pvarray` object.

[6]:
```
pvrow_left = pvarray.ts_pvrows[0]
n_segments = len(pvrow_left.back.list_segments)
print("Back side of leftmost PV row has {} segments".format(n_segments))
```

```
Back side of leftmost PV row has 5 segments
```

[7]:
```
pvrow_center = pvarray.ts_pvrows[1]
n_segments = len(pvrow_center.front.list_segments)
print("Front side of center PV row has {} segments".format(n_segments))
```

```
Front side of center PV row has 3 segments
```

Indexing the timeseries surfaces in a PV array

In order to perform some calculations on PV array surfaces, it is often important to index them. `pvfactors` takes care of this.

We can for instance check the index of the timeseries surfaces on the front side of the center PV row

[8]:
```
# List some indices
ts_surface_list = pvrow_center.front.all_ts_surfaces
print("Indices of surfaces on front side of center PV row")
for ts_surface in ts_surface_list:
    index = ts_surface.index
    print("... surface index: {}".format(index))
```

```
Indices of surfaces on front side of center PV row
... surface index: 40
... surface index: 41
... surface index: 42
... surface index: 43
... surface index: 44
... surface index: 45
```

Intuitively, one could have expected only 3 timeseries surfaces because that's what the previous plot at index `0` was showing. But it is important to understand that ALL timeseries surfaces are created at PV array fitting time, even the ones that don't exist for the given timestamps. So in this example: - we have 3 illuminated timeseries surfaces, which do exist at timestamp `0` - and 3 shaded timeseries surfaces, which do NOT exist at timestamp `0` (so they have zero length).

Let's check that.

[9]:
```
for ts_surface in ts_surface_list:
    index = ts_surface.index
    shaded = ts_surface.shaded
    length = ts_surface.length
    print("Surface with index: '{}' has shading status '{}' and length {} m".
→format(index, shaded, length))
```

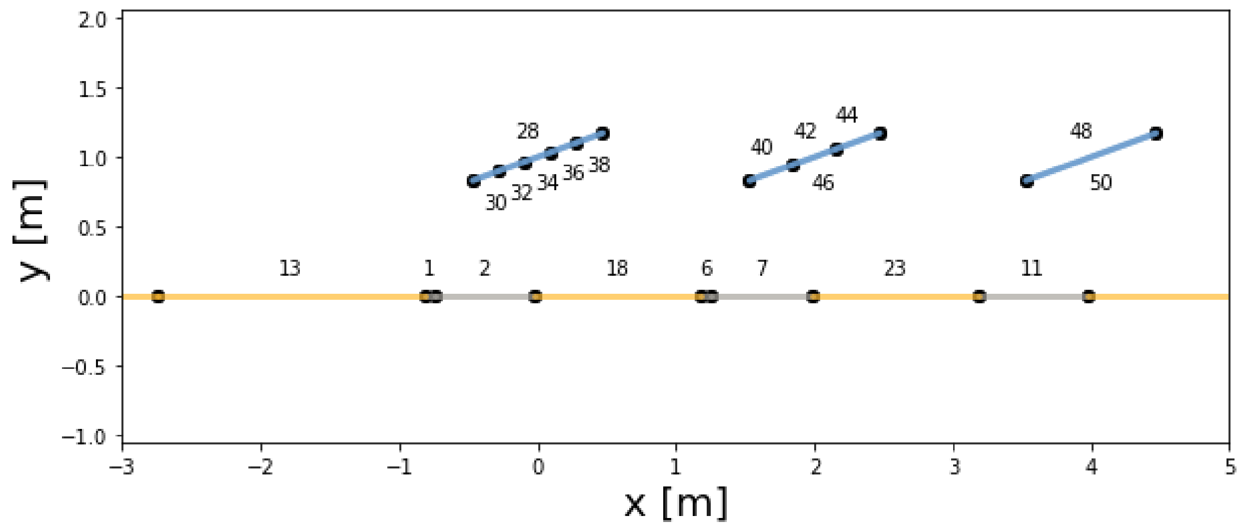
```
Surface with index: '40' has shading status 'False' and length [0.33333333] m
Surface with index: '41' has shading status 'True' and length [0.] m
Surface with index: '42' has shading status 'False' and length [0.33333333] m
Surface with index: '43' has shading status 'True' and length [0.] m
Surface with index: '44' has shading status 'False' and length [0.33333333] m
Surface with index: '45' has shading status 'True' and length [0.] m
```

As expected, all shaded timeseries surfaces on the front side of the PV row have length zero.

Plot PV array with indices

It is possible also to visualize the PV surface indices of all the non-zero surfaces when plotting a PV array, for a given timestamp (here at the first timestamp, so `0`).

[10]:
```
# Plot pvarray shapely geometries with surface indices
f, ax = plt.subplots(figsize=(10, 4))
pvarray.plot_at_idx(0, ax, with_surface_index=True)
ax.set_xlim(-3, 5)
plt.show()
```



As shown above, the surfaces on the front side of the center PV row have indices 40, 42, and 44.

Calculate view factors

In this section, we will learn how to:

- calculate the view factor matrix from a PV array object and understand its shape
- plot the pvarray with indices to visualize the meaning of the matrix

Note: the following calculation steps are already implemented in the simulation engine `PVEngine`, please refer to the next tutorials for running complete simulations.

Imports and settings

[1]:
```
# Import external libraries
import matplotlib.pyplot as plt
import numpy as np
import warnings
warnings.filterwarnings("ignore", category=RuntimeWarning)

# Settings
%matplotlib inline
np.set_printoptions(precision=3)
```

Prepare PV array parameters

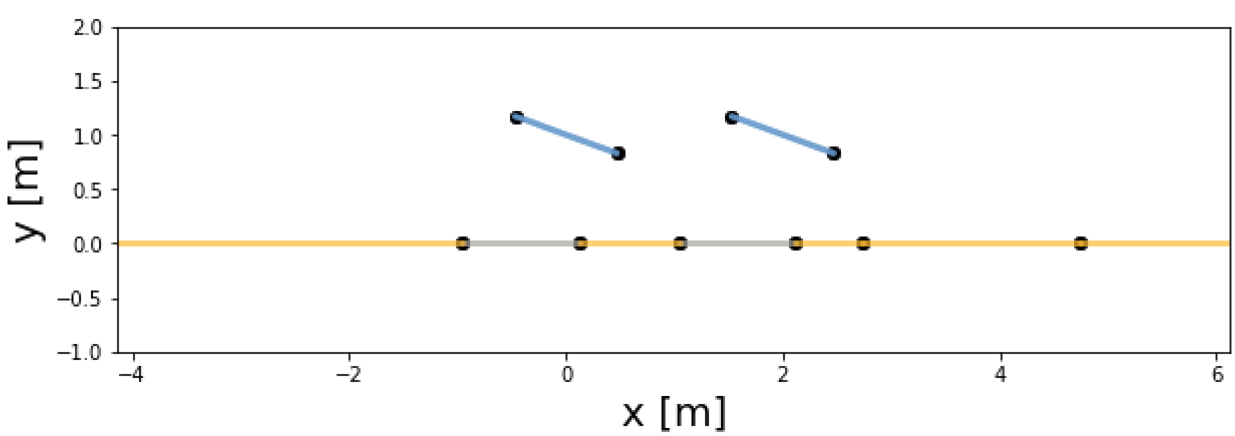
[2]:
```
pvarray_parameters = {
    'n_pvrows': 2,            # number of pv rows
    'pvrow_height': 1,        # height of pvrows (measured at center / torque tube)
    'pvrow_width': 1,         # width of pvrows
    'axis_azimuth': 0.,       # azimuth angle of rotation axis
    'surface_tilt': 20.,      # tilt of the pv rows
    'surface_azimuth': 90.,   # azimuth of the pv rows front surface
    'solar_zenith': 40.,      # solar zenith angle
    'solar_azimuth': 150.,    # solar azimuth angle
    'gcr': 0.5,               # ground coverage ratio
}
```

Create a PV array and required attributes

Import the `OrderedPVArray` class and create a PV array object using the parameters above

[3]:
```
from pvfactors.geometry import OrderedPVArray

pvarray = OrderedPVArray.fit_from_dict_of_scalars(pvarray_parameters)
```

[4]:
```
# Plot pvarray shapely geometries at timestep 0
f, ax = plt.subplots(figsize=(10, 3))
pvarray.plot_at_idx(0, ax)
plt.show()
```



As discussed in the "PV Array geometry introduction" tutorial, the ground also has "cut points" to indicate the limits of what the PV row front and back sides can see.

Calculating the view factor matrix

In order to calculate the view factor matrix, we need to pass the PV array object to view factor calculator method.

Create the view factor calculator.

[5]:
```
# import view factor calculator
from pvfactors.viewfactors import VFCalculator
# instantiate calculator
vf_calculator = VFCalculator()
```

[6]:
```
# calculate view factor matrix of the pv array
vf_matrix = vf_calculator.build_ts_vf_matrix(pvarray)
```

Important remarks:

- the view factor matrix has shape `[n_ts_surfaces + 1, n_ts_surfaces + 1, n_timestamps]`, where `n_ts_surfaces` is the number of timeseries surfaces in the PV array, and `n_timestamps` is the number of timestamps
- the first 2 dimensions have value `n_ts_surfaces` **+ 1** because the view factors to the sky are also calculated, so the sky is considered like another surface in the mathematical problem

[7]:
```
print("Number of dimensions: {}".format(vf_matrix.ndim))
print("Shape of vf matrix: {}".format(vf_matrix.shape))
```

```
Number of dimensions: 3
Shape of vf matrix: (24, 24, 1)
```

Here is a function to help make sense of this

[8]:
```
def select_view_factor(i, j, vf_matrix):
    "Function to print the view factors"
    n = vf_matrix.shape[0] - 1
    vf = vf_matrix[i, j, :]
    # print the view factor
    i = i if i < n else 'sky'
    j = j if j < n else 'sky'
    print('View factor from surface {} to surface {}: {}'.format(i, j, np.around(vf,␣
↪decimals=2)))
```

Let’s print some of the view factor values, and check their meaning on a PV array plot with surface indices

[9]:
```
# View factors from back of leftmost pv row
select_view_factor(17, 0, vf_matrix)
select_view_factor(17, 3, vf_matrix)
select_view_factor(17, 13, vf_matrix)
# View factors from back of rightmost pv row
select_view_factor(21, 3, vf_matrix)
# View factors from front of leftmost pv row
select_view_factor(15, 23, vf_matrix)
# View factors from front of rightmost pv row
select_view_factor(19, 23, vf_matrix)
```

```
View factor from surface 17 to surface 0: [0.4]
View factor from surface 17 to surface 3: [0.05]
```

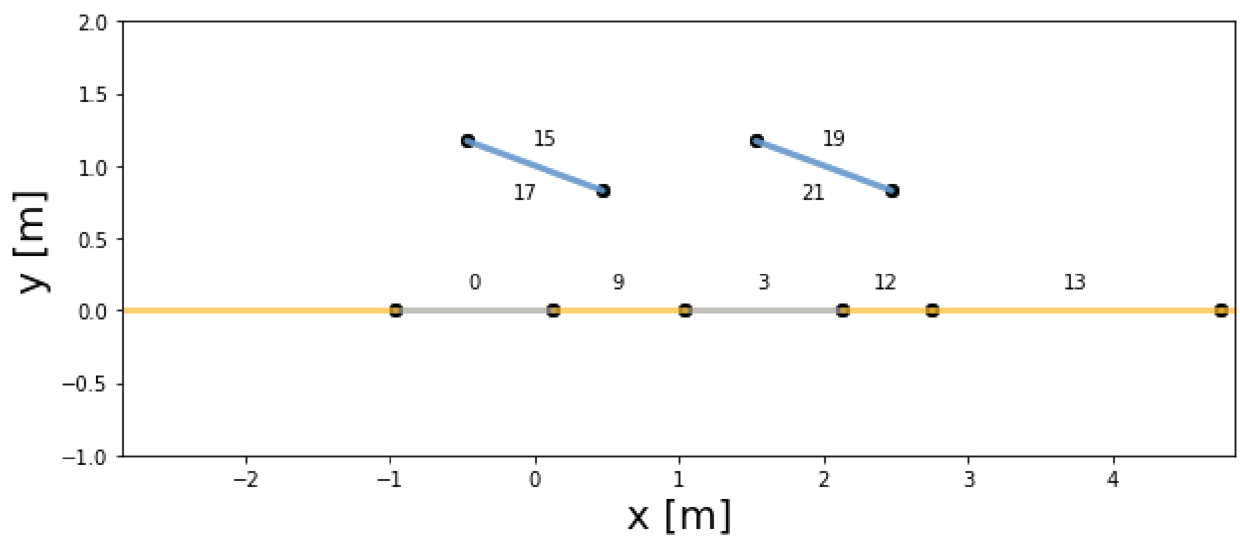
(continues on next page)

(continued from previous page)

```
View factor from surface 17 to surface 13: [0.]
View factor from surface 21 to surface 3: [0.4]
View factor from surface 15 to surface sky: [0.94]
View factor from surface 19 to surface sky: [0.97]
```

Let's plot the PV array with the surface indices to understand visually what these numbers mean:

[10]:
```
# Plot pvarray shapely geometries
f, ax = plt.subplots(figsize=(10, 4))
pvarray.plot_at_idx(0, ax, with_surface_index=True)
plt.show()
```



Run full timeseries simulations

In this section, we will learn how to:

- run full timeseries simulations using the `PVEngine` class, and visualize some of the results
- run full timeseries simulations using the `run_timeseries_engine()` function

Imports and settings

[1]:
```
# Import external libraries
import os
import numpy as np
import matplotlib.pyplot as plt
from datetime import datetime
import pandas as pd
import warnings

# Settings
%matplotlib inline
np.set_printoptions(precision=3, linewidth=300)
warnings.filterwarnings('ignore')
# Paths
```

(continues on next page)

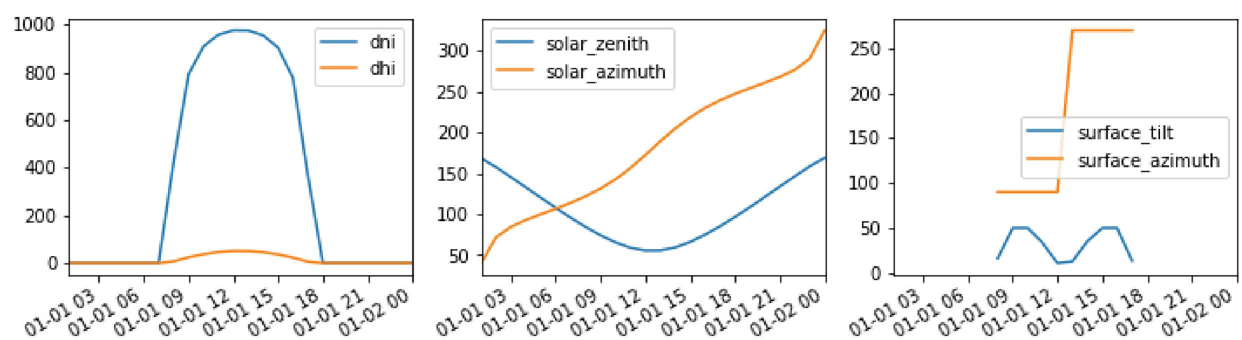
(continued from previous page)

```
LOCAL_DIR = os.getcwd()
DATA_DIR = os.path.join(LOCAL_DIR, 'data')
filepath = os.path.join(DATA_DIR, 'test_df_inputs_MET_clearsky_tucson.csv')
```

Get timeseries inputs

[2]:
```
def export_data(fp):
    tz = 'US/Arizona'
    df = pd.read_csv(fp, index_col=0)
    df.index = pd.DatetimeIndex(df.index).tz_convert(tz)
    return df

df = export_data(filepath)
df_inputs = df.iloc[:24, :]
```

[3]:
```
# Plot the data
f, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(12, 3))
df_inputs[['dni', 'dhi']].plot(ax=ax1)
df_inputs[['solar_zenith', 'solar_azimuth']].plot(ax=ax2)
df_inputs[['surface_tilt', 'surface_azimuth']].plot(ax=ax3)
plt.show()
```



[4]:
```
# Use a fixed albedo
albedo = 0.2
```

Prepare PV array parameters

[5]:
```
pvarray_parameters = {
    'n_pvrows': 3,            # number of pv rows
    'pvrow_height': 1,        # height of pvrows (measured at center / torque tube)
    'pvrow_width': 1,         # width of pvrows
    'axis_azimuth': 0.,       # azimuth angle of rotation axis
    'gcr': 0.4,               # ground coverage ratio
    'rho_front_pvrow': 0.01,  # pv row front surface reflectivity
    'rho_back_pvrow': 0.03,   # pv row back surface reflectivity
}
```

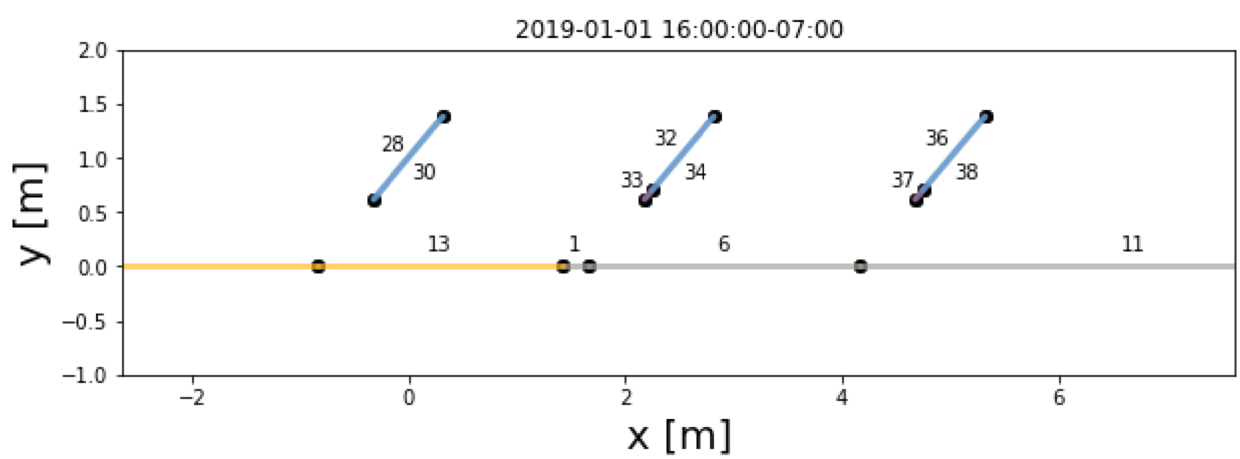
Run single timestep with PVEngine and inspect results

Instantiate the PVEngine class and fit it to the data

[6]:
```
from pvfactors.engine import PVEngine
from pvfactors.geometry import OrderedPVArray

# Create ordered PV array
pvarray = OrderedPVArray.init_from_dict(pvarray_parameters)
# Create engine
engine = PVEngine(pvarray)
# Fit engine to data
engine.fit(df_inputs.index, df_inputs.dni, df_inputs.dhi,
           df_inputs.solar_zenith, df_inputs.solar_azimuth,
           df_inputs.surface_tilt, df_inputs.surface_azimuth,
           albedo)
```

The user can run a simulation for a single timestep and plot the returned PV array

[7]:
```
# Get the PV array
pvarray = engine.run_full_mode(fn_build_report=lambda pvarray: pvarray)

# Plot pvarray shapely geometries
f, ax = plt.subplots(figsize=(10, 3))
pvarray.plot_at_idx(15, ax, with_surface_index=True)
ax.set_title(df.index[15])
plt.show()
```



The user can inspect the results very easily thanks to the simple geometry API

[8]:
```
# Get the calculated outputs from the pv array
center_row_front_incident_irradiance = pvarray.ts_pvrows[1].front.get_param_weighted(
↪'qinc')
left_row_back_reflected_incident_irradiance = pvarray.ts_pvrows[0].back.get_param_
↪weighted('reflection')
right_row_back_isotropic_incident_irradiance = pvarray.ts_pvrows[2].back.get_param_
↪weighted('isotropic')
```

(continues on next page)

(continued from previous page)

```
print("Incident irradiance on front surface of middle pv row: \n{} W/m2"
      .format(center_row_front_incident_irradiance))
print("Reflected irradiance on back surface of left pv row: \n{} W/m2"
      .format(left_row_back_reflected_incident_irradiance))
print("Isotropic irradiance on back surface of right pv row: \n{} W/m2"
      .format(right_row_back_isotropic_incident_irradiance))
```

```
Incident irradiance on front surface of middle pv row:
[     nan      nan      nan      nan      nan      nan      nan 117.633 587.344 685.115 652.526␣
↪616.77  618.875 656.024 685.556 550.172  87.66      nan      nan      nan      nan     ␣
↪nan      nan      nan] W/m2
Reflected irradiance on back surface of left pv row:
[   nan    nan    nan    nan    nan    nan    nan  8.375  6.597 39.275 58.563 68.346 64.
↪176 47.593 32.984 25.216  7.044    nan    nan    nan    nan    nan    nan    nan] W/m2
Isotropic irradiance on back surface of right pv row:
[  nan   nan   nan   nan   nan   nan   nan 0.076 2.15  3.116 1.697 0.199 0.414 2.627 4.
↪208 2.83  0.066   nan   nan   nan   nan   nan   nan   nan] W/m2
```

Run multiple timesteps with PVEngine

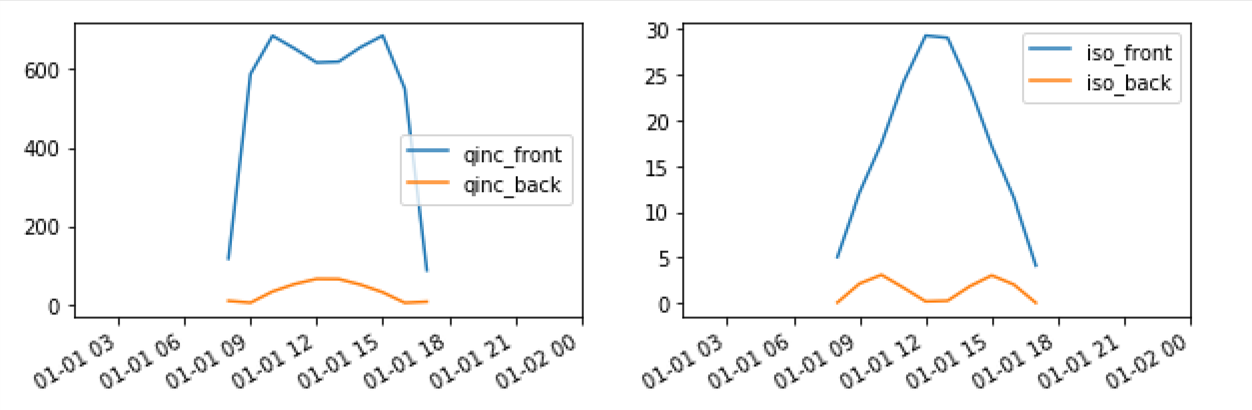
The users can also obtain a "report" that will look like whatever the users want, and which will rely on the simple geometry API shown above. Here is an example:

[9]:
```
# Create a function that will build a report
from pvfactors.report import example_fn_build_report

# Run full simulation
report = engine.run_full_mode(fn_build_report=example_fn_build_report)

# Print results (report is defined by report function passed by user)
df_report = pd.DataFrame(report, index=df_inputs.index)
df_report.iloc[6:11]
```

[9]:

	qinc_front	qinc_back	iso_front	iso_back
2019-01-01 07:00:00-07:00	NaN	NaN	NaN	NaN
2019-01-01 08:00:00-07:00	117.632919	9.703464	5.070103	0.076232
2019-01-01 09:00:00-07:00	587.344197	4.906038	12.087407	2.150237
2019-01-01 10:00:00-07:00	685.115436	33.478098	17.516188	3.115967
2019-01-01 11:00:00-07:00	652.526254	52.534503	24.250780	1.697046

[10]:
```
f, ax = plt.subplots(1, 2, figsize=(10, 3))
df_report[['qinc_front', 'qinc_back']].plot(ax=ax[0])
df_report[['iso_front', 'iso_back']].plot(ax=ax[1])
plt.show()
```



A function that builds a report needs to be specified, otherwise nothing will be returned by the simulation. Here is an example of a report function that will return the total incident irradiance (`'qinc'`) on the back surface of the rightmost PV row. A good way to get started building the reporting function is to use the example provided in the `report.py` module of the pvfactors package.

[11]:
```
def new_fn_build_report(pvarray): return {'total_inc_back': pvarray.ts_pvrows[1].back.
↪get_param_weighted('qinc')}
```

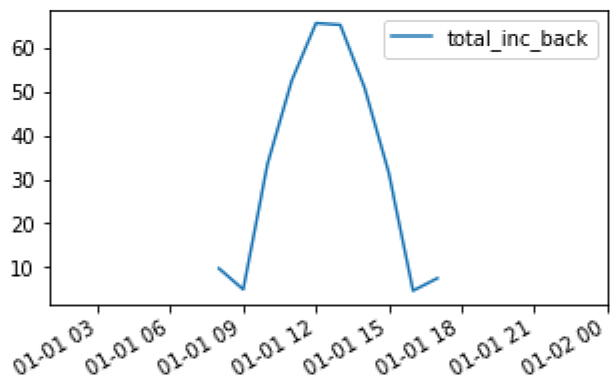
Now we can run the timeseries simulation again using the same engine but a different report function.

[12]:
```
# Run full simulation using new report function
new_report = engine.run_full_mode(fn_build_report=new_fn_build_report)

# Print results
df_new_report = pd.DataFrame(new_report, index=df_inputs.index)
df_new_report.iloc[6:11]
```

[12]:

	total_inc_back
2019-01-01 07:00:00-07:00	NaN
2019-01-01 08:00:00-07:00	9.703464
2019-01-01 09:00:00-07:00	4.906038
2019-01-01 10:00:00-07:00	33.478098
2019-01-01 11:00:00-07:00	52.534503

[13]:
```
f, ax = plt.subplots(figsize=(5, 3))
df_new_report.plot(ax=ax)
plt.show()
```



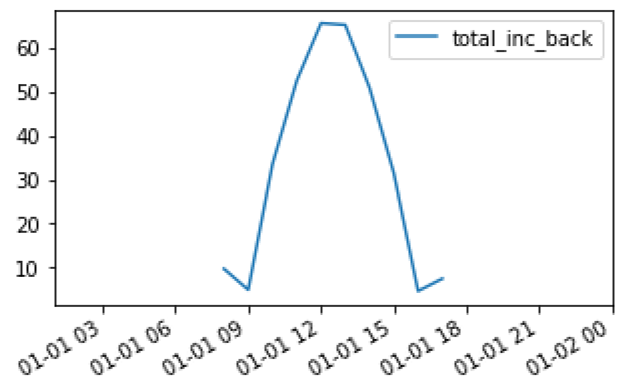
We can see in the printed output the new report generated by the simulation run.
For convenience, we've been using dictionaries as the data structure holding the reports, but it could be anything else, like numpy arrays, pandas dataframes, etc.

Run one or multiple timesteps with the `run_timeseries_engine()` function

The same thing can be accomplished using a function from the `run.py` module of the pvfactors package.
But only the report will be returned.

[14]:
```
# import function
from pvfactors.run import run_timeseries_engine

# run simulation using new_fn_build_report
report_from_fn = run_timeseries_engine(new_fn_build_report, pvarray_parameters, df_
↪inputs.index,
                                       df_inputs.dni, df_inputs.dhi,
                                       df_inputs.solar_zenith, df_inputs.solar_azimuth,
                                       df_inputs.surface_tilt, df_inputs.surface_azimuth,
                                       albedo)

# make a dataframe out of the report
df_report_from_fn = pd.DataFrame(report_from_fn, index=df_inputs.index)
```

[15]:
```
f, ax = plt.subplots(figsize=(5, 3))
df_report_from_fn.plot(ax=ax)
plt.show()
```



The plot above shows that we get the same results as previously.

Run full simulations in parallel

In this section, we will learn how to:

- run full timeseries simulations in parallel (with multiprocessing) using the `run_parallel_engine()` function

Note: for a better understanding, it might help to read the previous tutorial section on running full timeseries simulations sequentially before going through the following

Imports and settings

[1]:
```
# Import external libraries
import os
import numpy as np
import matplotlib.pyplot as plt
from datetime import datetime
import pandas as pd
import warnings

# Settings
%matplotlib inline
np.set_printoptions(precision=3, linewidth=300)
warnings.filterwarnings('ignore')
# Paths
LOCAL_DIR = os.getcwd()
DATA_DIR = os.path.join(LOCAL_DIR, 'data')
filepath = os.path.join(DATA_DIR, 'test_df_inputs_MET_clearsky_tucson.csv')
```

Get timeseries inputs

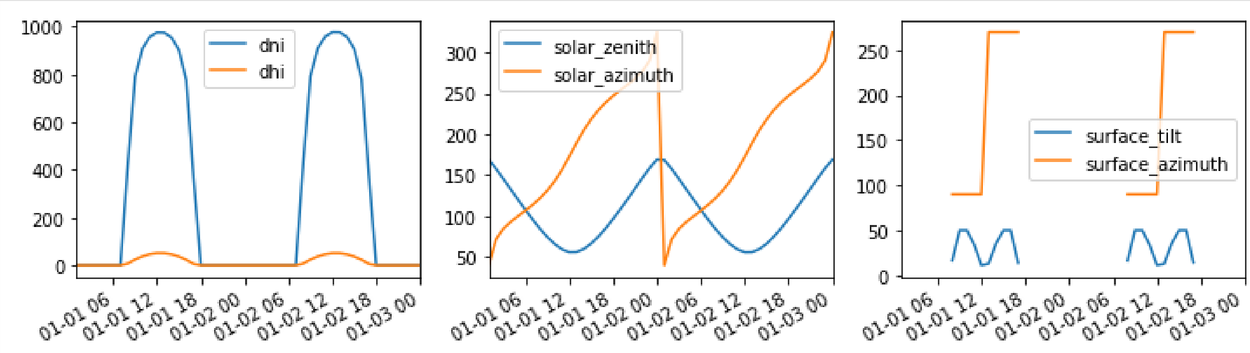
[2]:
```
def export_data(fp):
    tz = 'US/Arizona'
    df = pd.read_csv(fp, index_col=0)
    df.index = pd.DatetimeIndex(df.index).tz_convert(tz)
    return df
```

(continues on next page)

(continued from previous page)

```
df = export_data(filepath)
df_inputs = df.iloc[:48, :]
```

[3]:
```
# Plot the data
f, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(12, 3))
df_inputs[['dni', 'dhi']].plot(ax=ax1)
df_inputs[['solar_zenith', 'solar_azimuth']].plot(ax=ax2)
df_inputs[['surface_tilt', 'surface_azimuth']].plot(ax=ax3)
plt.show()
```



[4]:
```
# Use a fixed albedo
albedo = 0.2
```

Prepare PV array parameters

[5]:
```
pvarray_parameters = {
    'n_pvrows': 3,              # number of pv rows
    'pvrow_height': 1,          # height of pvrows (measured at center / torque tube)
    'pvrow_width': 1,           # width of pvrows
    'axis_azimuth': 0.,         # azimuth angle of rotation axis
    'gcr': 0.4,                 # ground coverage ratio
    'rho_front_pvrow': 0.01,    # pv row front surface reflectivity
    'rho_back_pvrow': 0.03      # pv row back surface reflectivity
}
```

Run simulations in parallel with `run_parallel_engine()`

Running full mode timeseries simulations in parallel is done using the `run_parallel_engine()`.
In the previous tutorial section on running timeseries simulations, we showed that a function needed to be passed in order to build a report out of the timeseries simulation.
For the parallel mode, it will not be very different but we will need to pass a class (or an object) instead. The reason is that python multiprocessing uses pickling to run different processes, but python functions cannot be pickled, so a class or an object with the necessary methods needs to be passed instead in order to build a report.

An example of a report building class is provided in the `report.py` module of the pvfactors package.

[6]:
```python
# Choose the number of workers
n_processes = 3
```

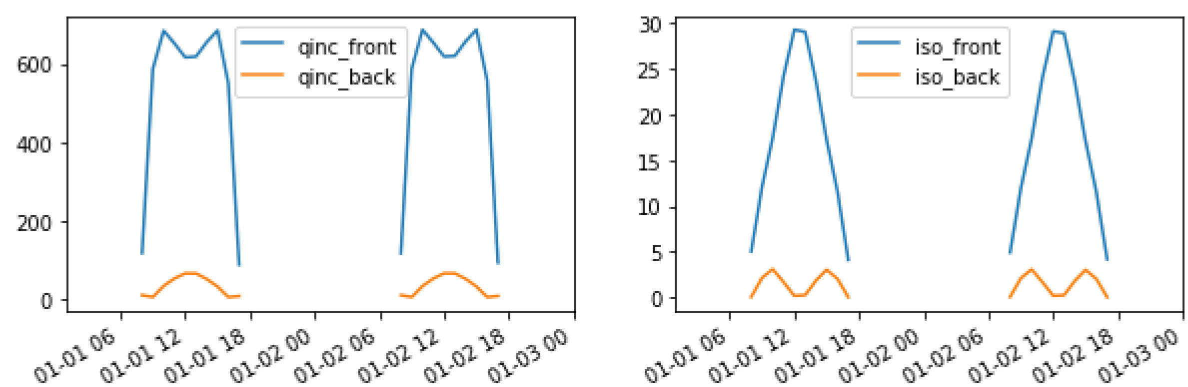
[7]:
```python
# import function to run simulations in parallel
from pvfactors.run import run_parallel_engine
# import the report building class for the simulation run
from pvfactors.report import ExampleReportBuilder

# run simulations in parallel mode
report = run_parallel_engine(ExampleReportBuilder, pvarray_parameters, df_inputs.index,
                             df_inputs.dni, df_inputs.dhi,
                             df_inputs.solar_zenith, df_inputs.solar_azimuth,
                             df_inputs.surface_tilt, df_inputs.surface_azimuth,
                             albedo, n_processes=n_processes)

# make a dataframe out of the report
df_report = pd.DataFrame(report, index=df_inputs.index)
df_report.iloc[6:11, :]
```

```
INFO:pvfactors.run:Parallel calculation elapsed time: 0.19188380241394043 sec
```

[7]:

	qinc_front	qinc_back	iso_front	iso_back
2019-01-01 07:00:00-07:00	NaN	NaN	NaN	NaN
2019-01-01 08:00:00-07:00	117.632919	9.703464	5.070103	0.076232
2019-01-01 09:00:00-07:00	587.344197	4.906038	12.087407	2.150237
2019-01-01 10:00:00-07:00	685.115436	33.478098	17.516188	3.115967
2019-01-01 11:00:00-07:00	652.526254	52.534503	24.250780	1.697046

[8]:
```python
f, ax = plt.subplots(1, 2, figsize=(10, 3))
df_report[['qinc_front', 'qinc_back']].plot(ax=ax[0])
df_report[['iso_front', 'iso_back']].plot(ax=ax[1])
plt.show()
```



The results above are consistent with running the simulations without parallel model (this is also tested in the package).

Building a report for parallel mode

For parallel simulations, a **class** (or object) that builds the report needs to be specified, otherwise nothing will be returned by the simulation.
Here is an example of a report building class that will return the total incident irradiance (`'qinc'`) on the back surface of the rightmost PV row. A good way to get started building the reporting class is to use the example provided in the `report.py` module of the pvfactors package.
Another important action of the class is to merge the different reports resulting from the parallel simulations: **since the users decide how the reports are built, the users are also responsible for specifying how to merge the reports after a parallel run**.

The static method that builds the reports needs to be named `build(report, pvarray)`.
And the static method that merges the reports needs to be named `merge(reports)`.

[9]:
```
class NewReportBuilder(object):
    """A class is required to build reports when running calculations with
    multiprocessing because of python constraints"""

    @staticmethod
    def build(pvarray):
        # Return back side qinc of rightmost PV row
        return {'total_inc_back': pvarray.ts_pvrows[1].back.get_param_weighted('qinc').
↪tolist()}

    @staticmethod
    def merge(reports):
        """Works for dictionary reports"""
        report = reports[0]
        # Merge other reports
        keys_report = list(reports[0].keys())
        for other_report in reports[1:]:
            for key in keys_report:
                report[key] += other_report[key]
        return report
```

[10]:
```
# run simulations in parallel mode using the new reporting class
new_report = run_parallel_engine(NewReportBuilder, pvarray_parameters, df_inputs.index,
                                 df_inputs.dni, df_inputs.dhi,
                                 df_inputs.solar_zenith, df_inputs.solar_azimuth,
                                 df_inputs.surface_tilt, df_inputs.surface_azimuth,
                                 albedo, n_processes=n_processes)

# make a dataframe out of the report
df_new_report = pd.DataFrame(new_report, index=df_inputs.index)
```

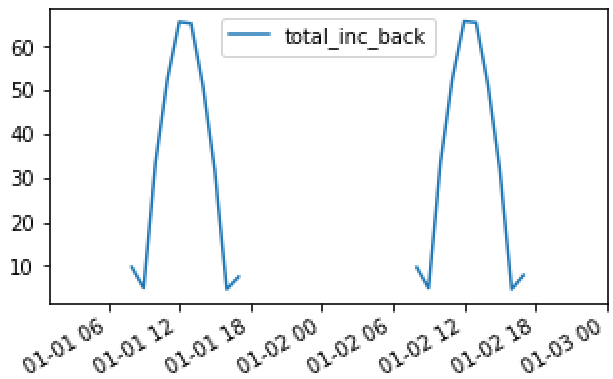
```
INFO:pvfactors.run:Parallel calculation elapsed time: 0.19736433029174805 sec
```

[11]:
```
f, ax = plt.subplots(figsize=(5, 3))
df_new_report.plot(ax=ax)
```

(continues on next page)

(continued from previous page)

```
plt.show()
```



The plot above shows that we're getting the same results we obtained in the previous tutorial section with the new report generating function.

Account for AOI reflection losses (in full mode only)

In this section, we will learn:

- how `pvfactors` accounts for AOI losses by default
- how to account for AOI-dependent reflection losses for direct, circumsolar, and horizon irradiance components
- how to account for AOI-dependent reflection losses for isotropic and reflection irradiance components
- how to run all of this using the `pvfactors` run functions

Imports and settings

[1]:
```
# Import external libraries
import os
import numpy as np
import matplotlib.pyplot as plt
from datetime import datetime
import pandas as pd
import warnings

# Settings
%matplotlib inline
np.set_printoptions(precision=3, linewidth=300)
warnings.filterwarnings('ignore')
plt.style.use('seaborn-whitegrid')
plt.rcParams.update({'font.size': 12})
# Paths
LOCAL_DIR = os.getcwd()
DATA_DIR = os.path.join(LOCAL_DIR, 'data')
filepath = os.path.join(DATA_DIR, 'test_df_inputs_MET_clearsky_tucson.csv')

RUN_FIXED_TILT = True
```

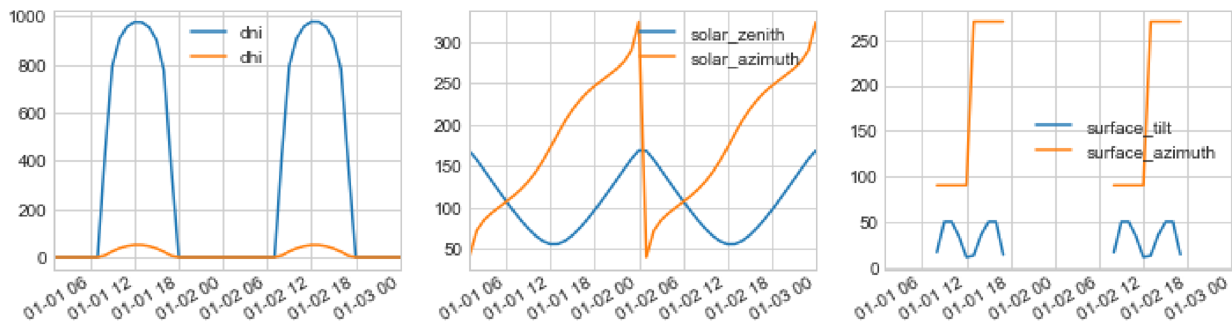
Let's define a few helper functions that will help clarify the notebook

[2]:
```
# Helper functions for plotting and simulation
def plot_irradiance(df_report):
    # Plot irradiance
    f, ax = plt.subplots(nrows=1, ncols=2, figsize=(12, 4))
    # Plot back surface irradiance
    df_report[['qinc_back', 'qabs_back']].plot(ax=ax[0])
    ax[0].set_title('Back surface irradiance')
    ax[0].set_ylabel('W/m2')
    # Plot front surface irradiance
    df_report[['qinc_front', 'qabs_front']].plot(ax=ax[1])
    ax[1].set_title('Front surface irradiance')
    ax[1].set_ylabel('W/m2')
    plt.show()

def plot_aoi_losses(df_report):
    # plotting AOI losses
    f, ax = plt.subplots(figsize=(5.5, 4))
    df_report[['aoi_losses_back_%']].plot(ax=ax)
    df_report[['aoi_losses_front_%']].plot(ax=ax)
    # Adjust axes
    ax.set_ylabel('%')
    ax.legend(['AOI losses back PV row', 'AOI losses front PV row'])
    ax.set_title('AOI losses')
    plt.show()

# Create a function that will build a simulation report
def fn_report(pvarray):
    # Get irradiance values
    report = {'qinc_back': pvarray.ts_pvrows[1].back.get_param_weighted('qinc'),
              'qabs_back': pvarray.ts_pvrows[1].back.get_param_weighted('qabs'),
              'qinc_front': pvarray.ts_pvrows[1].front.get_param_weighted('qinc'),
              'qabs_front': pvarray.ts_pvrows[1].front.get_param_weighted('qabs')}
    # Calculate AOI losses
    report['aoi_losses_back_%'] = (report['qinc_back'] - report['qabs_back']) / report[
↪'qinc_back'] * 100.
    report['aoi_losses_front_%'] = (report['qinc_front'] - report['qabs_front']) /␣
↪report['qinc_front'] * 100.
    # Return report
    return report
```

Get timeseries inputs

[3]:
```
def export_data(fp):
    tz = 'US/Arizona'
    df = pd.read_csv(fp, index_col=0)
    df.index = pd.DatetimeIndex(df.index).tz_convert(tz)
    return df

df = export_data(filepath)
df_inputs = df.iloc[:48, :]
```

[4]:
```
# Plot the data
f, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(12, 3))
df_inputs[['dni', 'dhi']].plot(ax=ax1)
df_inputs[['solar_zenith', 'solar_azimuth']].plot(ax=ax2)
df_inputs[['surface_tilt', 'surface_azimuth']].plot(ax=ax3)
plt.show()
```



[5]:
```
# Use a fixed albedo
albedo = 0.2
```

Prepare PV array parameters

[6]:
```
pvarray_parameters = {
    'n_pvrows': 3,            # number of pv rows
    'pvrow_height': 1,        # height of pvrows (measured at center / torque tube)
    'pvrow_width': 1,         # width of pvrows
    'axis_azimuth': 0.,       # azimuth angle of rotation axis
    'gcr': 0.4,               # ground coverage ratio
}
```

Default AOI loss behavior

In pvfactors:

- qinc is the total incident irradiance on a surface, and it does not account for reflection losses
- but qabs, which is the total absorbed irradiance by a surface, does accounts for it.

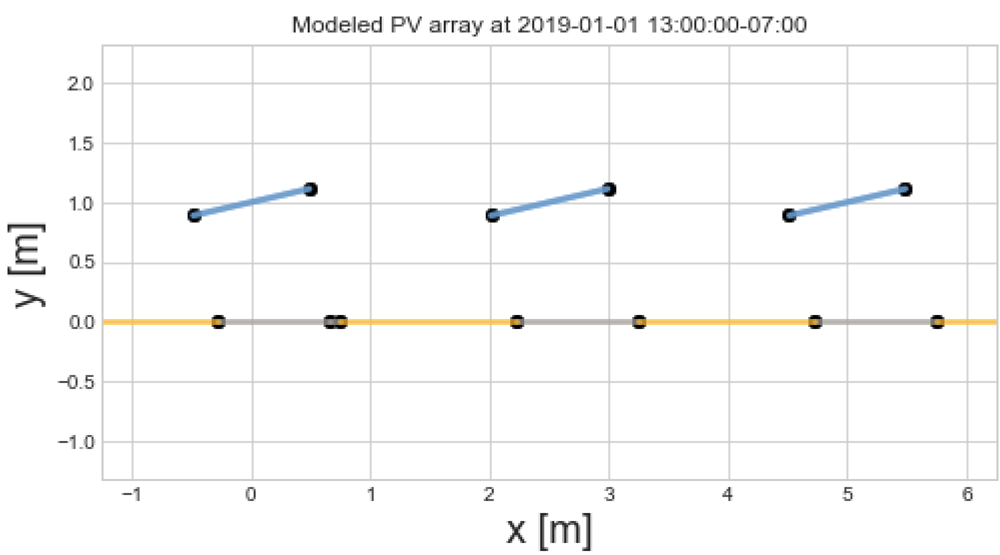
By default, pvfactors assumes that all reflection losses (or AOI losses) are diffuse; i.e. they do not depend on angle of incidence (AOI). Here is an example.

Let's run a full mode simulation (reflection equilibrium) and compare the calculated incident and absorbed irradiance on both sides of a PV row in a modeled PV array. We'll use 3% reflection for PV row front surfaces, and 5% for the back surfaces.

[7]:
```
from pvfactors.geometry import OrderedPVArray
# Create PV array
pvarray = OrderedPVArray.init_from_dict(pvarray_parameters)
```

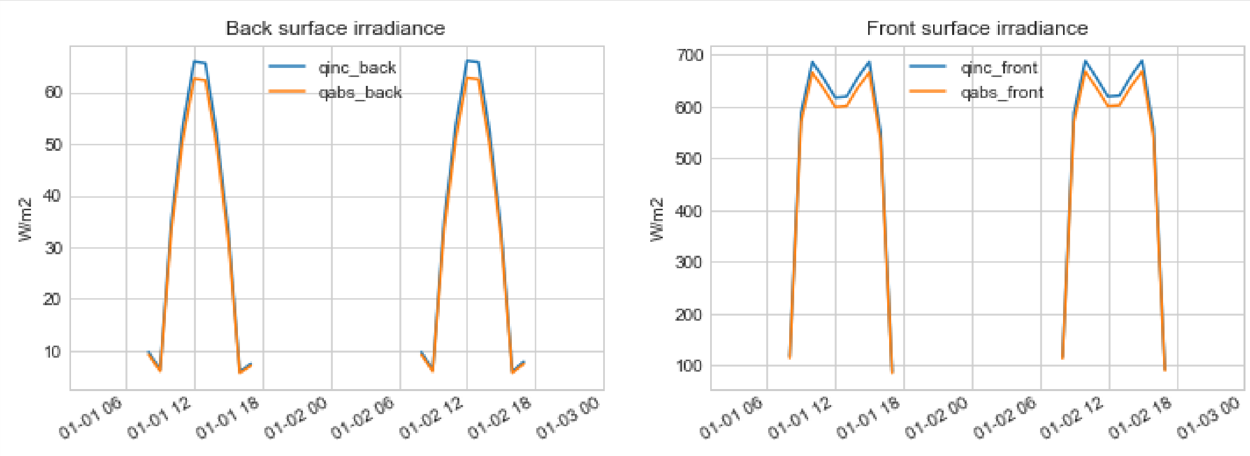
[8]:
```
from pvfactors.engine import PVEngine
from pvfactors.irradiance import HybridPerezOrdered
# Create irradiance model
irradiance_model = HybridPerezOrdered(rho_front=0.03, rho_back=0.05)
# Create engine
engine = PVEngine(pvarray, irradiance_model=irradiance_model)
# Fit engine to data
engine.fit(df_inputs.index, df_inputs.dni, df_inputs.dhi,
           df_inputs.solar_zenith, df_inputs.solar_azimuth,
           df_inputs.surface_tilt, df_inputs.surface_azimuth,
           albedo)
```

[9]:
```
# Plot pvarray shapely geometries
f, ax = plt.subplots(figsize=(8, 4))
pvarray.plot_at_idx(12, ax)
plt.title('Modeled PV array at {}'.format(df_inputs.index[12]))
plt.show()
```



[10]:
```
# Run full mode simulation
report = engine.run_full_mode(fn_build_report=fn_report)
# Turn report into dataframe
df_report = pd.DataFrame(report, index=df_inputs.index)
```

[11]:
```
plot_irradiance(df_report)
```



Let's plot the back AOI losses

[12]:
```
plot_aoi_losses(df_report)
```

As shown above, by default `pvfactors` apply constant values of AOI losses for all the surfaces in the system, and for all the incident irradiance components:

- 3% loss for the irradiance incident on front of PV rows, which corresponds to the chosen `rho_front` in the irradiance model
- 5% loss for the irradiance incident on back of PV rows, which corresponds to the chosen `rho_back` in the irradiance model

Use an fAOI function in the irradiance model

The next step that can improve the AOI loss calculation, especially for the PV row front surface that receives a lot of direct light, would be to use reflection losses that would be dependent on the AOI, and that would be applied to all the irradiance model components: direct, circumsolar, and horizon light components.

What is an fAOI function?

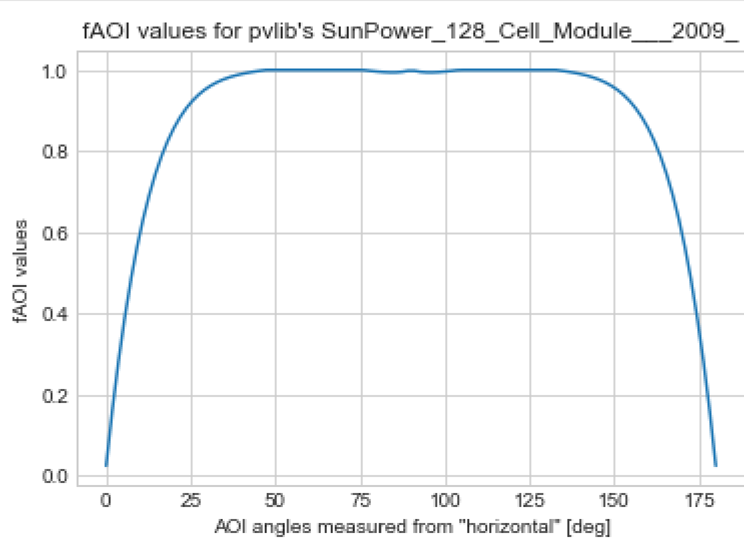
The `fAOI` function that the users need to provide takes an angle of incidence as input (AOI measured in degrees and against the surface horizontal - from 0 to 180 deg, **not** against the surface normal vector - which would have been from 0 to 90 deg), and it returns a transmission value for the incident light. So it's effectively a factor that removes reflection losses.

Let's see what this looks like. First, let's create such a function using a `pvfactors` utility function, and then we'll plot it.

Given a pvlib module database name, you can create an `fAOI` function as follows using pvfactors.

[13]:
```
# import utility function
from pvfactors.viewfactors.aoimethods import faoi_fn_from_pvlib_sandia
# Choose a module name
module_name = 'SunPower_128_Cell_Module___2009_'
# Create an faoi function
faoi_function = faoi_fn_from_pvlib_sandia(module_name)
```

[14]:
```
# Plot faoi function values
aoi_values = np.linspace(0, 180, 100)
faoi_values = faoi_function(aoi_values)

f, ax = plt.subplots()
ax.plot(aoi_values, faoi_values)
ax.set_title('fAOI values for pvlib\'s {}'.format(module_name))
ax.set_ylabel('fAOI values')
ax.set_xlabel('AOI angles measured from "horizontal" [deg]')
plt.show()
```



As expected, there are less reflection losses for incident light rays normal to the surface than everywhere else.

Use the fAOI function

It's then easy to use the created fAOI function in the irradiance models. It just has to be passed to the model at initialization.

For this example, we will use the same fAOI function for the front and back surfaces of the PV rows.

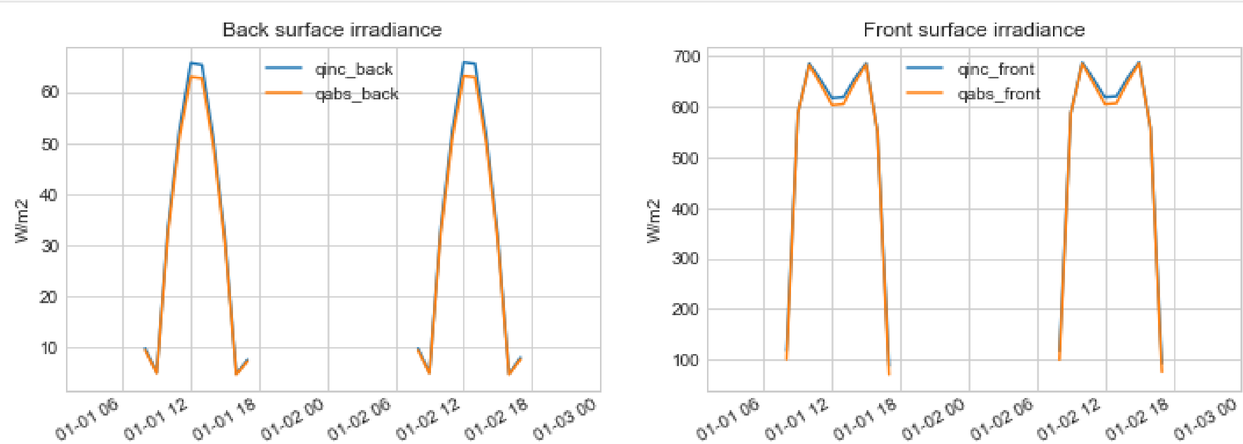
[15]:
```
# Create irradiance model with fAOI function
irradiance_model = HybridPerezOrdered(faoi_fn_front=faoi_function, faoi_fn_back=faoi_
↪function)
```

Then pass the model to the PVEngine and run the simulation as usual.

[16]:
```
# Create engine
engine = PVEngine(pvarray, irradiance_model=irradiance_model)
# Fit engine to data
engine.fit(df_inputs.index, df_inputs.dni, df_inputs.dhi,
           df_inputs.solar_zenith, df_inputs.solar_azimuth,
           df_inputs.surface_tilt, df_inputs.surface_azimuth,
           albedo)
# Run full mode simulation
report = engine.run_full_mode(fn_build_report=fn_report)
# Turn report into dataframe
df_report = pd.DataFrame(report, index=df_inputs.index)
```

Let's now see what the irradiance and AOI losses look like.

[17]:
```
plot_irradiance(df_report)
```



[18]:
```
plot_aoi_losses(df_report)
```

We can now see the changes in AOI losses, which now use the `fAOI` function for the direct, circumsolar, and horizon light components. But it still uses the constant `rho_front` and `rho_back` values for the reflection and isotropic components of the incident light on the surfaces.

Advanced: use an `fAOI` function for the (ground and array) reflection and isotropic components

The more advanced use is to apply the `fAOI` losses to the reflection and isotropic component of the light incident on the PV row surfaces.

In order to do so you simply need to pass the `fAOI` function to the view factor calculator before initializing the `PVEngine`.

In this case, the simulation workflow will be as follows:

- the `PVEngine` will still calculate the equilibrium of reflections assuming diffuse surfaces and constant reflection losses
- it will then use the calculated radiosity values and apply the `fAOI` using an integral combining the AOI losses and the view factor integrands, as described in the theory section, and similarly to Marion, B., et al (2017)

A word of caution

The users should be careful when using `fAOI` losses with the view factor calculator for the following reasons:

- in order to be fully consistent in the `PVEngine` calculations, it is wiser to re-calculate a global hemispherical reflectivity value using the `fAOI` function, which will be used in the reflection equilibrium calculation
- the method used for accounting `fAOI` losses in reflections is physically valid only if the surfaces are "infinitesimal" because it uses view factor formulas only valid in this case (see http://www.thermalradiation.net/sectionb/B-71.html). So in order to make it work in `pvfactors`, you'll need to discretize the PV row sides into smaller segments
- the method relies on the numerical calculation of an integral, and that calculation will converge only given a sufficient number of integral points (which can be provided to the `pvfactors` view factor calculator). Marion, B., et al (2017) seems to be using 180 points, but in `pvfactors`' implementation it doesn't look like it's enough for the integral to converge, so we'll use 1000 integral points in this example

- the two points above slow down the computation time by an order of magnitude. 8760 simulations that normally take a couple of seconds to run with `pvfactors`'s full mode can then take up to a minute

Apply `fAOI` losses to reflection terms

Discretize the PV row sides of the PV array:

[19]:
```
# first let's discretize the PV row sides
pvarray_parameters.update({
    'cut': {1: {'front': 5, 'back': 5}}
})
# Create a new pv array
pvarray = OrderedPVArray.init_from_dict(pvarray_parameters)
```

Add `fAOI` losses to the view factor calculator, and use 1000 integration points

[20]:
```
from pvfactors.viewfactors import VFCalculator

vf_calculator = VFCalculator(faoi_fn_front=faoi_function, faoi_fn_back=faoi_function,
                             n_aoi_integral_sections=1000)
```

Re-calculate global hemispherical reflectivity values based on `fAOI` function

[21]:
```
# For back PV row surface
is_back = True
rho_back = vf_calculator.vf_aoi_methods.rho_from_faoi_fn(is_back)
# For front PV row surface
is_back = False
rho_front = vf_calculator.vf_aoi_methods.rho_from_faoi_fn(is_back)

# Print results
print('Reflectivity values for front side: {}, and back side: {}'.format(rho_front, rho_
↪back))
```

```
Reflectivity values for front side: 0.029002539185428944, and back side: 0.
↪029002539185428944
```

Since we're using the same `fAOI` function for front and back sides, we now get the same global hemispherical reflectivity values.

We can now create the irradiance model.

[22]:
```
irradiance_model = HybridPerezOrdered(rho_front=rho_front, rho_back=rho_back,
                                      faoi_fn_front=faoi_function, faoi_fn_back=faoi_
↪function)
```

Simulations can then be run the usual way:

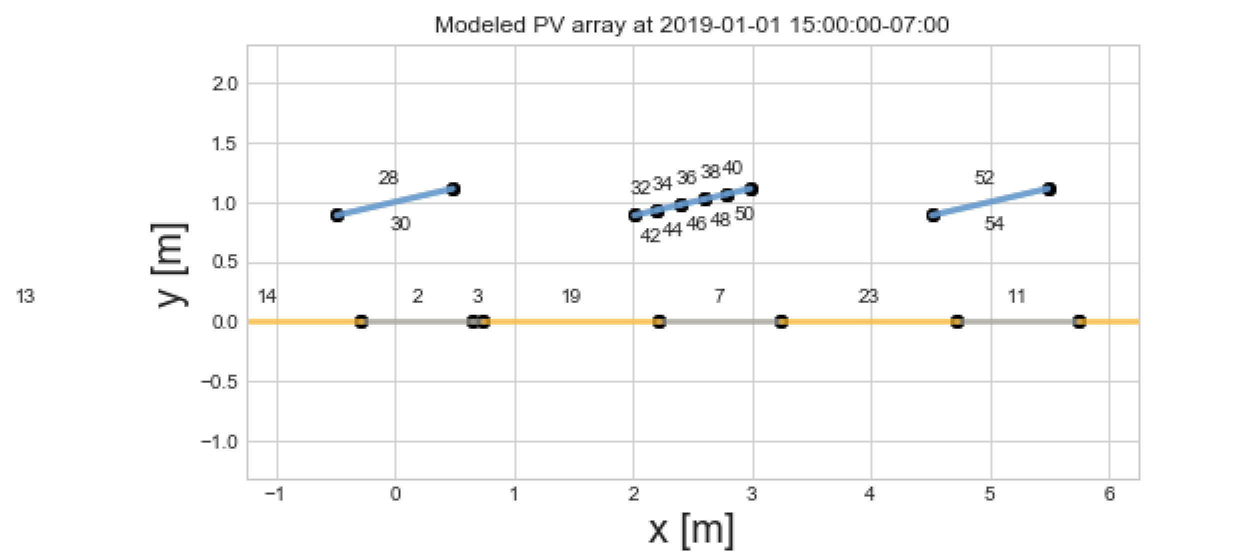
[23]:
```
# Create engine
engine = PVEngine(pvarray, vf_calculator=vf_calculator,
                  irradiance_model=irradiance_model)
# Fit engine to data
engine.fit(df_inputs.index, df_inputs.dni, df_inputs.dhi,
           df_inputs.solar_zenith, df_inputs.solar_azimuth,
```

(continues on next page)

(continued from previous page)

```
            df_inputs.surface_tilt, df_inputs.surface_azimuth,
            albedo)
```

[24]:
```
# Plot pvarray shapely geometries
f, ax = plt.subplots(figsize=(8, 4))
ax = pvarray.plot_at_idx(12, ax, with_surface_index=True)
plt.title('Modeled PV array at {}'.format(df_inputs.index[14]))
plt.show()
```

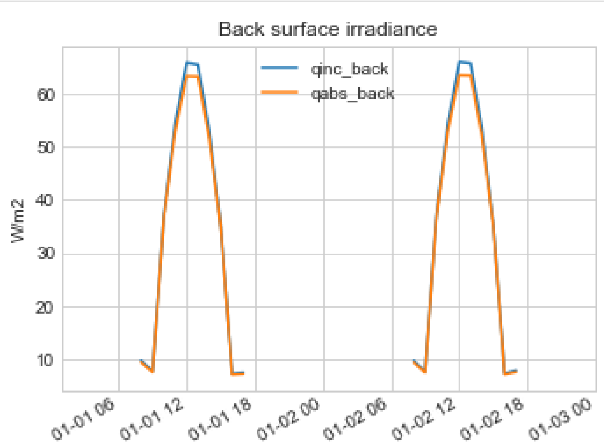


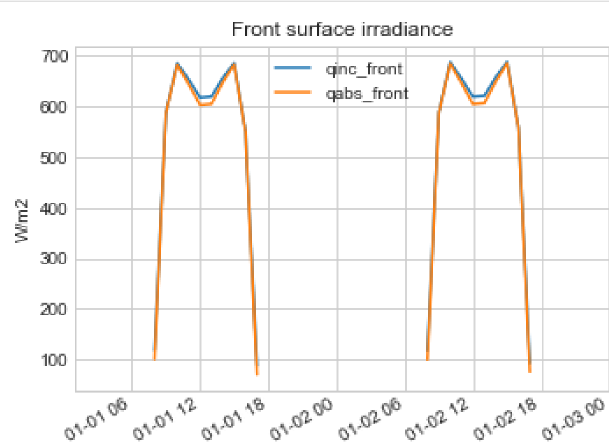
Run the simulation:

[25]:
```
# Run full mode simulation
report = engine.run_full_mode(fn_build_report=fn_report)
# Turn report into dataframe
df_report = pd.DataFrame(report, index=df_inputs.index)
```

Let's now see what the irradiance and AOI losses look like.

[26]:
```
plot_irradiance(df_report)
```





[27]:
```
plot_aoi_losses(df_report)
```

This is the way to apply `fAOI` losses to all the irradiance components in a `pvfactors` simulation.

Doing all of the above using the "run functions"

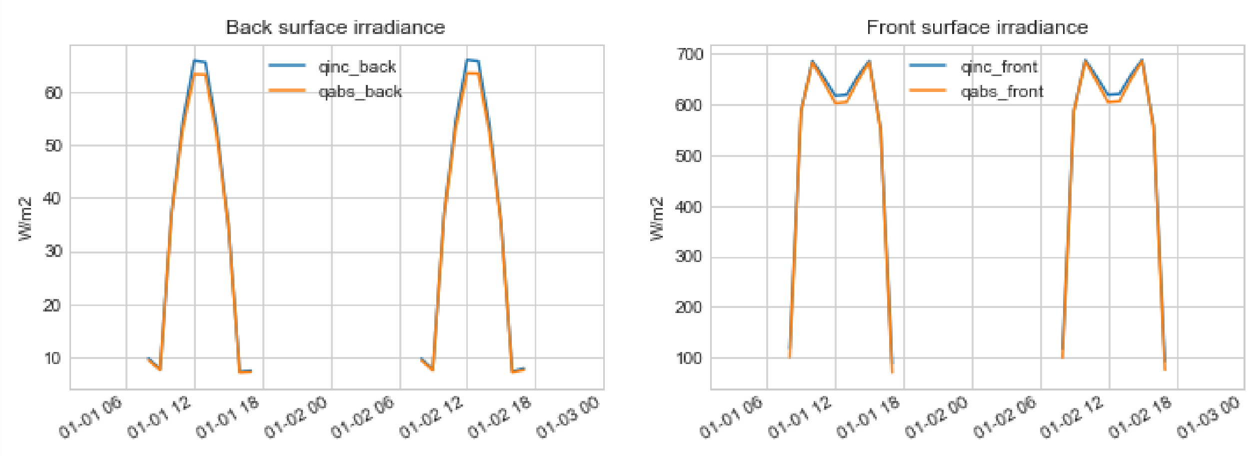
When using the "run functions", you'll just need to define the parameters in advance and then pass it to the functions.

[28]:
```
# Define the parameters for the irradiance model and the view factor calculator
irradiance_params = {'rho_front': rho_front, 'rho_back': rho_back,
                     'faoi_fn_front': faoi_function, 'faoi_fn_back': faoi_function}

vf_calculator_params = {'faoi_fn_front': faoi_function, 'faoi_fn_back': faoi_function,
                        'n_aoi_integral_sections': 1000}
```

Using `run_timeseries_engine()`

[29]:
```
from pvfactors.run import run_timeseries_engine

# run simulations in parallel mode
report_from_fn = run_timeseries_engine(fn_report, pvarray_parameters, df_inputs.index,
                                       df_inputs.dni, df_inputs.dhi,
                                       df_inputs.solar_zenith, df_inputs.solar_azimuth,
                                       df_inputs.surface_tilt, df_inputs.surface_azimuth,
                                       albedo,
                                       irradiance_model_params=irradiance_params,
                                       vf_calculator_params=vf_calculator_params)

# Turn report into dataframe
df_report_from_fn = pd.DataFrame(report_from_fn, index=df_inputs.index)
```

[30]:
```
plot_irradiance(df_report_from_fn)
```



[31]:
```
plot_aoi_losses(df_report_from_fn)
```

Using run_parallel_engine()

Because of Python's multiprocessing, and because functions cannot be pickled in Python, the functions need to be wrapped up into classes.

[32]:
```
class ReportBuilder(object):
    """Class for building the reports with multiprocessing"""
    @staticmethod
    def build(pvarray):
        pvrow = pvarray.ts_pvrows[1]
        report = {'qinc_front': pvrow.front.get_param_weighted('qinc'),
                  'qabs_front': pvrow.front.get_param_weighted('qabs'),
                  'qinc_back': pvrow.back.get_param_weighted('qinc'),
                  'qabs_back': pvrow.back.get_param_weighted('qabs')}

        # Calculate AOI losses
        report['aoi_losses_back_%'] = (report['qinc_back'] - report['qabs_back']) /
↪report['qinc_back'] * 100.
```

(continues on next page)

(continued from previous page)

```
        report['aoi_losses_front_%'] = (report['qinc_front'] - report['qabs_front']) /␣
↪report['qinc_front'] * 100.
        # Return report
        return report

    @staticmethod
    def merge(reports):
        report = reports[0]
        keys = report.keys()
        for other_report in reports[1:]:
            for key in keys:
                report[key] = list(report[key])
                report[key] += list(other_report[key])
        return report


class FaoiClass(object):
    """Class for passing the faoi function to engine"""
    @staticmethod
    def faoi(*args, **kwargs):
        fn = faoi_fn_from_pvlib_sandia(module_name)
        return fn(*args, **kwargs)
```

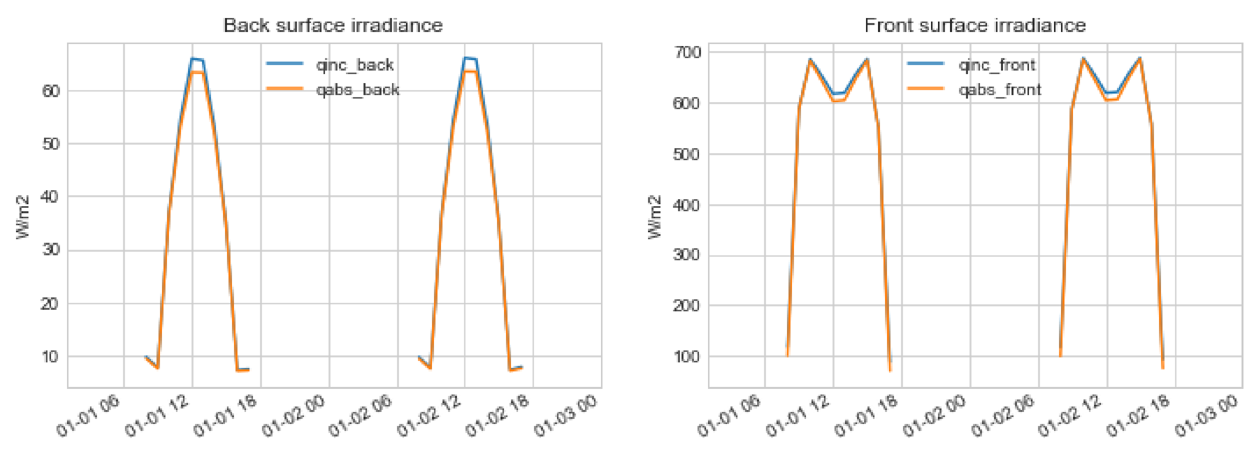
Pass the objects through the dictionaries and run the simulation

[33]:
```
# Define the parameters for the irradiance model and the view factor calculator
irradiance_params = {'rho_front': rho_front, 'rho_back': rho_back,
                     'faoi_fn_front': FaoiClass, 'faoi_fn_back': FaoiClass}

vf_calculator_params = {'faoi_fn_front': FaoiClass, 'faoi_fn_back': FaoiClass,
                        'n_aoi_integral_sections': 1000}
```

[34]:
```
from pvfactors.run import run_parallel_engine

# run simulations in parallel mode
report_from_fn = run_parallel_engine(ReportBuilder, pvarray_parameters, df_inputs.index,
                                     df_inputs.dni, df_inputs.dhi,
                                     df_inputs.solar_zenith, df_inputs.solar_azimuth,
                                     df_inputs.surface_tilt, df_inputs.surface_azimuth,
                                     albedo,
                                     irradiance_model_params=irradiance_params,
                                     vf_calculator_params=vf_calculator_params)

# Turn report into dataframe
df_report_from_fn = pd.DataFrame(report_from_fn, index=df_inputs.index)
```

```
INFO:pvfactors.run:Parallel calculation elapsed time: 0.731104850769043 sec
```

[35]:
```
plot_irradiance(df_report_from_fn)
```



[36]:
```
plot_aoi_losses(df_report_from_fn)
```

It's that easy!

2.3.3 Details on the "fast mode" simulations

In the "fast mode", `pvfactors` assumes that all incident irradiance values for the system are known except for the PV row back surfaces. So since the system to solve is now explicit (no matrix inversion needed), it runs a little bit faster than the full mode, but it is less accurate.

Note: Some tests show that for 8760 hourly simulations, the run time is less than 1 second for the fast mode vs. less than 2 seconds for the full mode.

Run fast simulations

In this section, we will learn how to:

- run timeseries simulations with "fast" mode and using the PVEngine
- run timeseries simulations with "fast" mode and using the run_timeseries_engine() function

Note: we recommend using the "full" mode instead, because it is more accurate and it's about the same run time. See previous tutorials on full mode simulations.

Imports and settings

[1]:
```
# Import external libraries
import os
import numpy as np
import matplotlib.pyplot as plt
from datetime import datetime
import pandas as pd
import warnings

# Settings
%matplotlib inline
np.set_printoptions(precision=3, linewidth=300)
warnings.filterwarnings('ignore')
# Paths
LOCAL_DIR = os.getcwd()
DATA_DIR = os.path.join(LOCAL_DIR, 'data')
filepath = os.path.join(DATA_DIR, 'test_df_inputs_MET_clearsky_tucson.csv')
```

Overview of "fast" mode

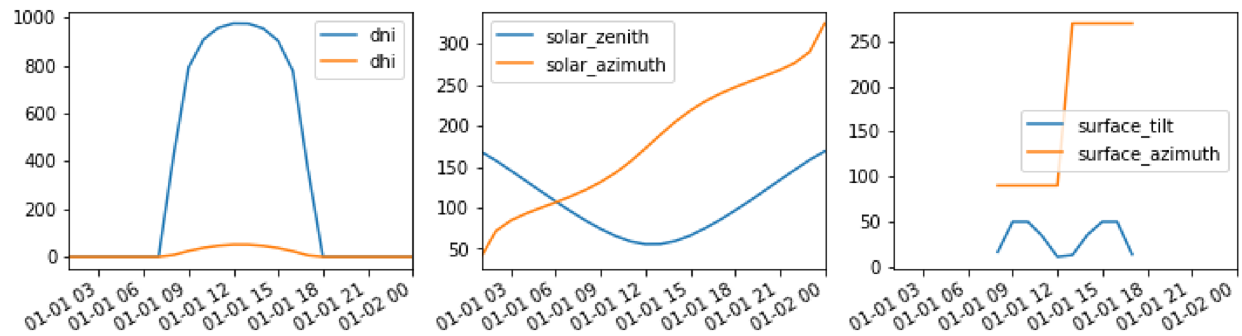
The fast mode simulation was first introduced in pvfactors v1.0.2. It relies on a mathematical simplification (see Theory section of the documentation) of the problem that assumes that we already know the irradiance incident on all front PV row surfaces and ground surfaces (for instance using the Perez model). In this mode, we therefore only calculate view factors from PV row back surfaces to the other ones assuming that back surfaces don't see each other. This way we do not need to solve a linear system of equations anymore for "ordered" PV arrays.

This is an approximation compared to the "full" mode, since we're not calculating the impact of the multiple reflections on the PV array surfaces. But the initial results show that it still provides a very reasonable estimate of back surface incident irradiance values.

Get timeseries inputs

[2]:
```
def import_data(fp):
    """Import 8760 data to run pvfactors simulation"""
    tz = 'US/Arizona'
    df = pd.read_csv(fp, index_col=0)
    df.index = pd.DatetimeIndex(df.index).tz_convert(tz)
    return df

df = import_data(filepath)
df_inputs = df.iloc[:24, :]
```

[3]:
```
# Plot the data
f, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(12, 3))
df_inputs[['dni', 'dhi']].plot(ax=ax1)
df_inputs[['solar_zenith', 'solar_azimuth']].plot(ax=ax2)
df_inputs[['surface_tilt', 'surface_azimuth']].plot(ax=ax3)
plt.show()
```



[4]:
```
# Use a fixed albedo
albedo = 0.2
```

Prepare PV array parameters

[5]:
```
pvarray_parameters = {
    'n_pvrows': 3,            # number of pv rows
    'pvrow_height': 1,        # height of pvrows (measured at center / torque tube)
    'pvrow_width': 1,         # width of pvrows
    'axis_azimuth': 0.,       # azimuth angle of rotation axis
    'gcr': 0.4,               # ground coverage ratio
}
```

Run "fast" simulations with the PVEngine

The `PVEngine` can be used to easily run fast mode simulations, using its `run_fast_mode()` method.
In order to run the fast mode, the users need to specify which PV row to look at for calculating back surface incident irradiance. The way this is done is by specifying the index of the PV row either at initialiatization, or in the `run_fast_mode()` method.
Optionally, a specific segment index can also be passed to the PV Engine to calculate the irradiance only for a segment of a PV row's back surface.

[6]:
```
# Import PVEngine and OrderedPVArray
from pvfactors.engine import PVEngine
from pvfactors.geometry import OrderedPVArray

# Instantiate PV array
pvarray = OrderedPVArray.init_from_dict(pvarray_parameters)
# Create PV engine, and specify the index of the PV row for fast mode
```

(continues on next page)

(continued from previous page)

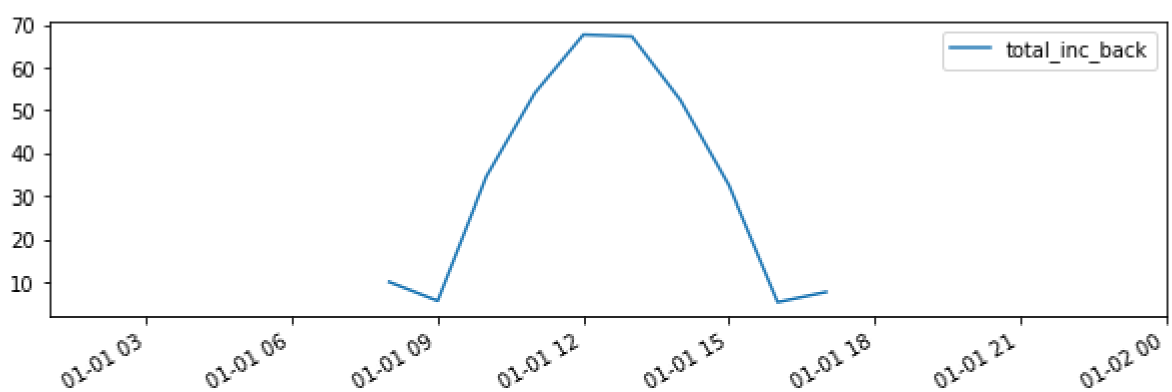
```
fast_mode_pvrow_index = 1  # look at the middle PV row
eng = PVEngine(pvarray, fast_mode_pvrow_index=fast_mode_pvrow_index)

# Fit PV engine to the timeseries data
eng.fit(df_inputs.index, df_inputs.dni, df_inputs.dhi,
        df_inputs.solar_zenith, df_inputs.solar_azimuth,
        df_inputs.surface_tilt, df_inputs.surface_azimuth,
        albedo)
```

A report function needs to be passed to the `run_fast_mode()` method in order to return calculated values. The report function will need to rely on the pvarray's `ts_pvrows` attribute in order to get the calculated outputs.

[7]:
```
# Create a function to build the report: the function will get the total incident␣
↪irradiance on the back
# of the middle PV row
def fn_report(pvarray): return {'total_inc_back': (pvarray.ts_pvrows[fast_mode_pvrow_
↪index]
                                                   .back.list_segments[0].get_param_
↪weighted('qinc'))}
```

[8]:
```
# Run timeseries simulations
report = eng.run_fast_mode(fn_build_report=fn_report)
```

[9]:
```
# make a dataframe out of the report
df_report = pd.DataFrame(report, index=df_inputs.index)

# and plot the results
f, ax = plt.subplots(figsize=(10, 3))
df_report.plot(ax=ax)
plt.show()
```



Run "fast" simulations using `run_timeseries_engine()`

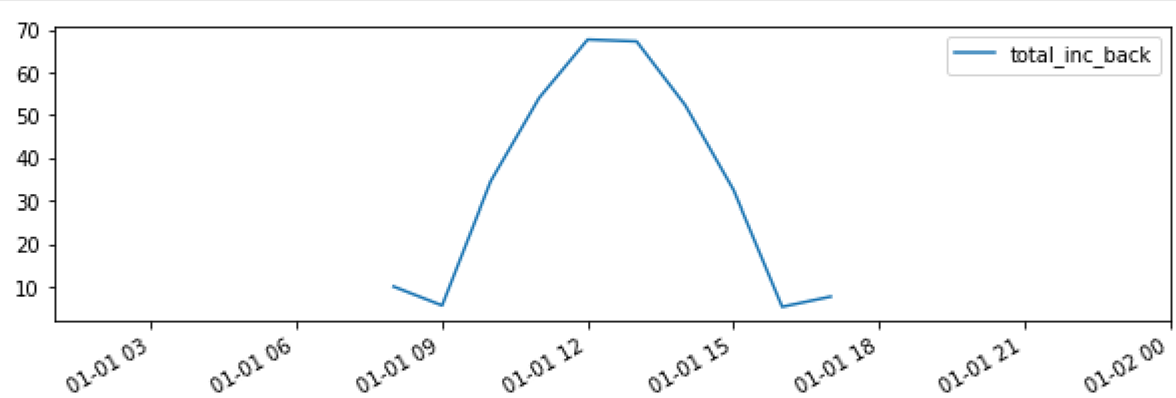
The same thing can be done more rapidly using the `run_timeseries_engine()` function.

[10]:
```
# Choose center row (index 1) for the fast simulation
fast_mode_pvrow_index = 1
```

[11]:
```
# Create a function to build the report: the function will get the total incident␣
↪irradiance on the back
# of the middle PV row
def fn_report(pvarray): return {'total_inc_back': (pvarray.ts_pvrows[fast_mode_pvrow_
↪index]
                                                    .back.list_segments[0].get_param_
↪weighted('qinc'))}
```

[12]:
```
# import function to run simulations in parallel
from pvfactors.run import run_timeseries_engine

# run simulations
report = run_timeseries_engine(
    fn_report, pvarray_parameters, df_inputs.index,
    df_inputs.dni, df_inputs.dhi,
    df_inputs.solar_zenith, df_inputs.solar_azimuth,
    df_inputs.surface_tilt, df_inputs.surface_azimuth, albedo,
    fast_mode_pvrow_index=fast_mode_pvrow_index)  # this will trigger fast mode␣
↪calculation

# make a dataframe out of the report
df_report = pd.DataFrame(report, index=df_inputs.index)
```

[13]:
```
f, ax = plt.subplots(figsize=(10, 3))
df_report.plot(ax=ax)
plt.show()
```



The results obtained are strictly identical to when the `PVEngine` was used, but it takes a little less code to run a simulation.

2.4 Theory

The theory of the model is explained here. For more details, please refer to[1] .

Contents:

2.4.1 Introduction

Due to new bifacial technologies and larger utility-scale photovoltaic (PV) arrays, there is a growing need for models that can more accurately account for the multiple diffuse light components and reflections incident on the front and back surfaces of a PV array.

Fig. 5: Fig. 1: Example of bifacial modules on single-axis tracker

Ray tracing models are often chosen for their high level of accuracy, but in order to reach such precision they often become computationally intensive and slower to run.
The view factor model presented here uses a simplified method for the calculation of bifacial irradiance. It is an application of view factors on 2D geometry representations of PV arrays (for both single-axis trackers and fixed tilt systems), invariant by translation along the tracker axis. It can be used for energy production calculation of large PV arrays thanks to its high computational speed (less than 2 seconds for annual hourly simulations), and also because edge effects occurring in large PV arrays are negligible.

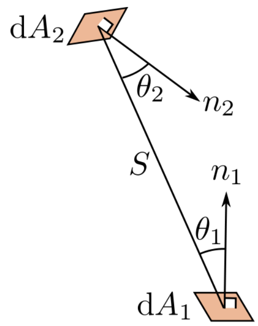
The goal of this view factor model is to allow fast and accurate irradiance calculations to provide quantitative answers to diffuse shading and bifacial PV questions.

[1] Anoma, M., Jacob, D., Bourne, B.C., Scholl, J.A., Riley, D.M. and Hansen, C.W., 2017. View Factor Model and Validation for Bifacial PV and Diffuse Shade on Single-Axis Trackers. In 44th IEEE Photovoltaic Specialist Conference.

2.4.2 View Factors

Theory

The view factors, also called configuration factors, come from the definition of the directional spectral radiative power of a differential area.



Let's take the example of black body surfaces, and then extrapolate the results to more general ones.
For a black body differential area dA_1, we can write that the radiative power emitted to another black body differential area dA_2 is:

$$d^2Q_{\lambda,d1-d2}\ d\lambda = i_{\lambda,b,1}\ d\lambda\ d\omega_1\ dA_{1,projected}$$

where:
* $d^2Q_{\lambda,d1-d2}$ is the spectral radiative power from dA_1 to dA_2
* $i_{\lambda,b,1}$ is the blackbody spectral intensity from dA_1
* λ is the wavelength
* $d\omega_1$ is the solid angle from dA_1 to dA_2
* $dA_{1,projected}$ is the projected area dA_1 onto the S direction

We can then integrate over λ for a black body and rearrange:

$$d^2Q_{d1-d2}\ = \frac{\sigma\ T_1^4}{\pi}\ d\omega_1\ cos\theta_1\ dA_1$$

Then:

$$d^2Q_{d1-d2}\ = \frac{\sigma\ T_1^4}{\pi}\ \frac{cos\theta_2\ dA_2}{S^2}\ cos\theta_1\ dA_1$$

And finally:

$$\frac{d^2Q_{d1-d2}}{dA_1} = \sigma\ T_1^4 \frac{cos\theta_2\ cos\theta_1}{\pi\ S^2}\ dA_2$$

The view factor from the differential area dA_1 to the differential area dA_2 is then defined as:

$$d^2F_{d1-d2}\ = \frac{cos\theta_2\ cos\theta_1}{\pi\ S^2}\ dA_2$$

And for two finite areas A_1 and A_2:

$$F_{1-2}\ = \frac{1}{A_1}\int_{A_1}\int_{A_2} d^2F_{d1-d2}\ dA_1\ = \frac{1}{A_1}\int_{A_1}\int_{A_2} \frac{cos\theta_2\ cos\theta_1}{\pi\ S^2}\ dA_2\ dA_1$$

We can also note that by reciprocity:

$$A_1 \; F_{1-2} = A_2 \; F_{2-1}$$

This approach also holds for diffuse surfaces, whose optical properties don't depend on the direction of the rays.
We can understand the view factor from a surface A_1 to a surface A_2 as the fraction of the hemisphere around A_1 that is occupied by A_2.

Application

We will be using configuration factors in the case of 2D geometries, which simplifies the calculations. The 2D assumption is made because the tracker rows considered will be fairly long, and the edge effects will therefore have less impact.
Also, instead of doing the numerical integration of the double integral representing the view factor, we will systematically try to use analytical solutions of those integrals from tables.
Here are links describing some view factors relevant to PV array geometries.

- View factor of a wedge: http://www.thermalradiation.net/sectionc/C-5.html
- View factor of parallel planes: http://www.thermalradiation.net/sectionc/C-2a.htm
- View factor of angled planes: http://www.thermalradiation.net/sectionc/C-5a.html
- The Hottel method is also widely used in the model

Adding non-diffuse reflection losses

For the derivation shown above, we assumed that the surfaces were diffuse. But as shown in[1], it is possible to add an approximation of non-diffuse effects by calculating absorption losses that are function of the angle-of-incidence (AOI) of the light.

If we're interested in calculating the **absorbed** irradiance coming from an infinite strip to an infinitesimal surface, we can calculate a view factor derated by AOI losses by starting with the formula derived in http://www.thermalradiation.net/sectionb/B-71.html.

The view factor from the infinitesimal surface dA_1 to the infinite strip $A_{2,1}$ is equal to:

$$dF_{dA_1-A_{2,1}} \; = \frac{1}{2} \, (cos\theta_2 \; - \; cos\theta_1)$$

For this small view of the strip, we can assume that a given AOI modifier function ($f(AOI)$), which represents reflection losses, is constant. Such that:

$$dF_{dA_1-A_{2,1},AOI} \; = \frac{1}{2} \, f(AOI) \, (cos\theta_2 \; - \; cos\theta_1)$$

We can then calculate the view factor derated by AOI losses from the infinitesimal surface dA_1 to the whole surface A_2 by summing up the values for all the small strips constituting that surface. Such that:

$$dF_{dA_1-A_2,AOI} \; = \sum_{j=1}^{3} dF_{dA_1-A_{2,j},AOI}$$

Note: Since this formula was derived for "infinitesimal" surfaces, in practice we can cut up the PV row sides into "small" segments to make this approximation more valid.

[1] Marion, B., MacAlpine, S., Deline, C., Asgharzadeh, A., Toor, F., Riley, D., Stein, J. and Hansen, C., 2017, June. A practical irradiance model for bifacial PV modules. In 2017 IEEE 44th Photovoltaic Specialist Conference (PVSC) (pp. 1537-1542). IEEE.

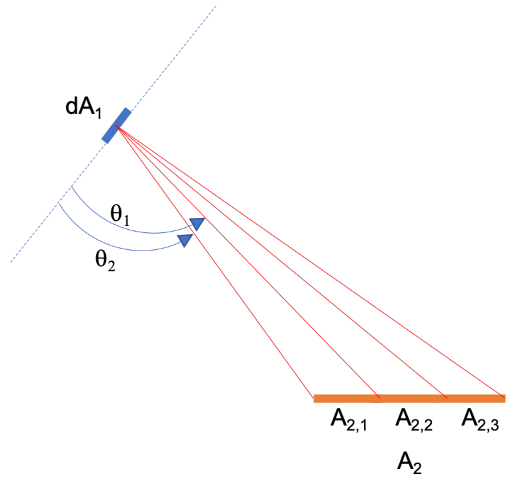
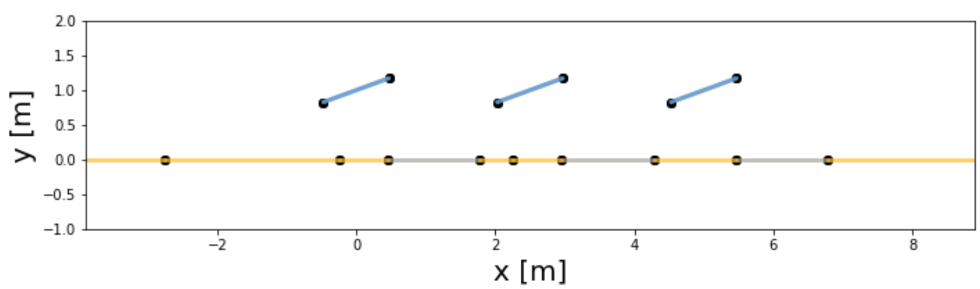


Fig. 6: Fig. 1: Schematics illustrating view factor formula from dA1 to infinite strips

2.4.3 Mathematical Model

In order to use the view factors as follows, we need to assume that the surfaces considered are diffuse (lambertian). Which means that their optical properties are independent of the angle of the rays (incident, reflected, or emitted).

The current version of the view factor model only addresses PV rows that are made out of straight lines (no "dual-tilt" for instance), with a flat ground. But the PV array can have any azimuth or tilt angle for the simulations. Below is the 2D representation of such a PV array, plotted with `pvfactors`.



The mathematical model used in `pvfactors` simulations is different depending on the simulation type that is run.

- in "full simulations", all of the reflections between the modeled surfaces are taken into account in the calculations, which leads to results that account for the equilibrium of reflections between surfaces.
- in "fast simulations", assumptions are made on the reflected irradiance from the environment surrounding the surfaces of interest.

Full simulations

When making some assumptions, it is possible to represent the calculation of irradiance terms on each surface with a linear system. The dimension of this system changes depending on the number of surfaces considered. But we can formulate it for the general case of `n` surfaces.

For a surface `i` we can write that:

$$q_{o,i} = q_{emitted,i} + q_{reflected,i}$$

Unit: W/m^2.

* $q_{o,i}$ is the radiosity of surface `i`, and it represents the outgoing radiative flux from it.
* $q_{emitted,i}$ is the emitted radiative flux from that surface. For instance the total emitted radiative flux of a blackbody is known to be σT^4 (with T the surface temperature and σ the Stefan–Boltzmann constant).
* $q_{reflected,i}$ is the reflected flux from that surface.

Finding values of interest like back side irradiance can only be done after finding the radiosity $q_{o,i}$ of each surface `i`. This can become a very complex system of equations where one would need to solve the energy balance on the considered systems .

But if we decide to make the assumption that $q_{emitted,i}$ is negligible, we can simplify the problem in a way that would enable us to find more easily some approximations of the values of interest. For now, this assumption makes some sense because the temperatures of the PV systems and the surroundings are generally not very high (< 330K). Besides the surfaces are not real black bodies, which means that their total (or broadband) emissions and absorptions will be even lower.
Under this assumption, we end up with:

$$q_{o,i} \approx q_{reflected,i}$$

where:

$$q_{reflected,i} = \rho_i * q_{incident,i}$$

with:
* $q_{incident,i}$ is the incident radiative flux on surface `i`.
* ρ_i is the total reflectivity of surface `i`.

We can further develop this expression and involve configuration factors as well as irradiance terms as follows:

$$q_{reflected,i} = \rho_i * (\sum_j q_{o,j} * F_{i,j} + Sky_i)$$

where:
* $\sum_j q_{o,j} * F_{i,j}$ is the contribution of all the surfaces `j` surrounding `i` to the incident radiative flux onto surface `i`.
* $F_{i,j}$ is the configuration factor (or view factor) of surface `i` to surface `j`.
* Sky_i is a sky irradiance term specific to surface `i` which contributes to the incident radiative flux $q_{incident,i}$, and associated with irradiance terms not represented in the geometrical model. For instance, it will be equal to $DNI_{POA} + circumsolar_{POA} + horizon_{POA}$ for the front (illuminated) side of the modules, when using the `HybridPerezOrdered` model.

This results into a linear system that can be written as follows:

$$\mathbf{q_o} = \mathbf{R}.(\mathbf{F}.\mathbf{q_o} + \mathbf{Sky})$$
$$(\mathbf{R}^{-1} - \mathbf{F}).\mathbf{q_o} = \mathbf{Sky}$$

Or, for a system of `n` surfaces:

$$(\begin{pmatrix} \rho_1 & 0 & 0 & \cdots & 0 \\ 0 & \rho_2 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \rho_n \end{pmatrix}^{-1} - \begin{pmatrix} F_{1,1} & F_{1,2} & F_{1,3} & \cdots & F_{1,n} \\ F_{2,1} & F_{2,2} & F_{2,3} & \cdots & F_{2,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ F_{n,1} & F_{n,2} & F_{n,3} & \cdots & F_{n,n} \end{pmatrix}).\begin{pmatrix} q_{o,1} \\ q_{o,2} \\ \vdots \\ q_{o,n} \end{pmatrix} = \begin{pmatrix} Sky_1 \\ Sky_2 \\ \vdots \\ Sky_n \end{pmatrix}$$

After solving this system and finding all of the radiosities, it is very easy to deduce values of interest like back side or front side incident irradiance.

Fast simulations

In the case of fast simulations and when interested in back side surfaces only, we can make additional assumptions that allow us to calculate the incident irradiance on back side surfaces without solving a linear system of equations.

In the full simulation case, we defined a vector of incident irradiance on all surfaces as follows:

$$\mathbf{q_{inc}} = \mathbf{F}.\mathbf{q_o} + \mathbf{Sky}$$

And we realized that we needed to solve for $\mathbf{q_o}$ in order to find $\mathbf{q_{inc}}$. But with the following assumptions, we can find an approximation of $\mathbf{q_{inc}}$ for back side surfaces without having to solve a linear system of equations:

1) we can assume that the radiosity of the surfaces is equal to their reflectivity multiplied by the incident irradiance on the surfaces as calculated by the Perez transposition model[1], which only works for front side surfaces. I.e.

$$\mathbf{q_o}\mathbf{R}.\mathbf{q_{perez}}$$

Here, $\mathbf{q_{perez}}$ can have values equal to zero for back side surfaces, which will lead to a good assumption if the back side surfaces don't see each other, which is the case in *OrderedPVArray*.

2) we can then also reduce the calculation of view factors to the view factors of the back side surfaces of interest, leading to the following:

$$\mathbf{q_{inc-back}}\mathbf{F_{back}}.\mathbf{R}.\mathbf{q_{perez}} + \mathbf{Sky_{back}}$$

Example

For instance, if we are interested in back side surfaces with indices 3 and 7, this will look like this:

$$\begin{pmatrix} q_{inc,3} \\ q_{inc,7} \end{pmatrix} = \begin{pmatrix} F_{3,1} & F_{3,2} & F_{3,3} & \cdots & F_{3,n} \\ F_{7,1} & F_{7,2} & F_{7,3} & \cdots & F_{7,n} \end{pmatrix} . \begin{pmatrix} \rho_1 & 0 & 0 & \cdots & 0 \\ 0 & \rho_2 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \rho_n \end{pmatrix} . \begin{pmatrix} q_{perez,1} \\ q_{perez,2} \\ \vdots \\ q_{perez,n} \end{pmatrix} + \begin{pmatrix} Sky_3 \\ Sky_7 \end{pmatrix}$$

[1] Perez, R., Seals, R., Ineichen, P., Stewart, R. and Menicucci, D., 1987. A new simplified version of the Perez diffuse irradiance model for tilted surfaces. Solar energy, 39(3), pp.221-231.

Grouping terms

For each back surface element, we can then group reflection terms that have identical reflectivity and $\mathbf{q_{perez}}$ terms into something more intuitive:

$$
\begin{aligned}
q_{inc-back} F_{to\ shaded\ ground} . albedo . q_{perez\ shaded\ ground} \\
+ F_{to\ illuminated\ ground} . albedo . q_{perez\ illuminated\ ground} \\
+ F_{to\ shaded\ front\ pv\ row} . \rho_{front\ pv\ row} . q_{perez\ front\ shaded\ pv\ row} \\
+ F_{to\ illuminated\ front\ pv\ row} . \rho_{front\ pv\ row} . q_{perez\ front\ shaded\ pv\ row} \\
+ F_{to\ sky\ dome} . luminance_{sky\ dome} \\
+ Sky_{inc-back}
\end{aligned}
$$

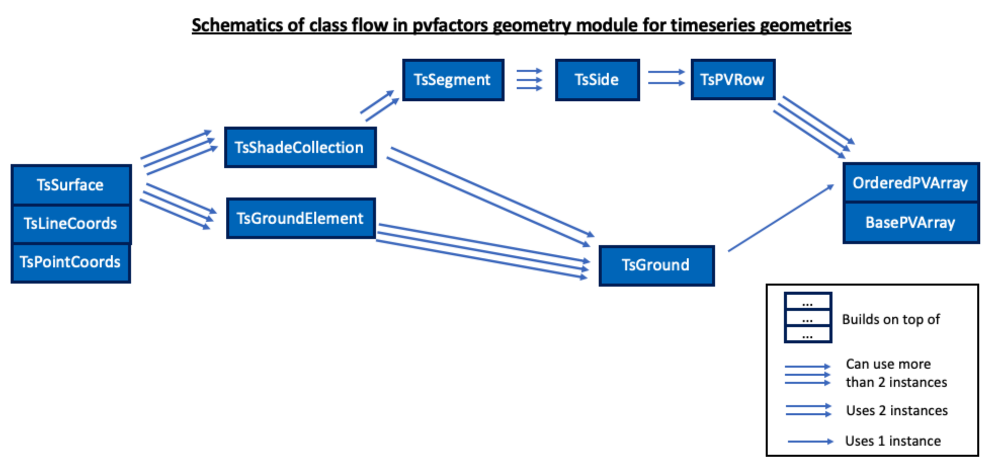
This form is quite useful because we can then rely on vectorization to calculate back surface incident irradiance quite rapidly.

2.5 Developer API

This is the class and function reference of pvfactors. For clarity and simplicity, all inherited methods and attributes have been removed from the class descriptions as there were often too many irrelevant ones coming from base packages like shapely.

2.5.1 geometry

The geometry sub-package of pvfactors implements multiple classes that make the construction of a 2D geometry for a PV array intuitive and scalable. It is meant to be decoupled from irradiance and view factor calculations so that it can be used independently for other purposes, like visualization for instance. The following schematics summarizes the organization of the classes in this sub-package.



base

Base classes for pvfactors geometry subpackage.

BaseSurface	Base surfaces will be extensions of `LineString` classes, but adding an orientation to it (normal vector).
PVSurface	PV surfaces inherit from *BaseSurface*.
ShadeCollection	A group of *PVSurface* objects that all have the same shading status.
PVSegment	A PV segment will be a collection of 2 collinear and contiguous shade collections, a shaded one and an illuminated one.
BaseSide	A side represents a fixed collection of PV segments objects that should all be collinear, with the same normal vector
BasePVArray	Base class for PV arrays in pvfactors.

pvfactors.geometry.base.BaseSurface

class pvfactors.geometry.base.**BaseSurface**(*coords*, *normal_vector=None*, *index=None*, *param_names=None*, *params=None*)

Base surfaces will be extensions of `LineString` classes, but adding an orientation to it (normal vector). So two surfaces could use the same linestring, but have opposite orientations.

__init__(*coords*, *normal_vector=None*, *index=None*, *param_names=None*, *params=None*)

Create a surface using linestring coordinates. Normal vector can have two directions for a given LineString, so the user can provide it in order to be specific, otherwise it will be automatically calculated, but then the surface won't know if it was supposed to be pointing "up" or "down". If the surface is empty, the normal vector will take the default value.

Parameters

- **coords** (*list*) – List of linestring coordinates for the surface
- **normal_vector** (*list, optional*) – Normal vector for the surface (Default = None, so will be calculated)
- **index** (*int, optional*) – Surface index (Default = None)
- **param_names** (*list of str, optional*) – Names of the surface parameters, eg reflectivity, total incident irradiance, temperature, etc. (Default = None)
- **params** (*dict, optional*) – Surface float parameters (Default = None)

Methods

__init__(coords[, normal_vector, index, ...])	Create a surface using linestring coordinates.
`difference`(linestring)	Calculate remaining surface after removing part belonging from provided linestring,
`get_param`(param)	Get parameter value from surface.
`plot`(ax[, color, with_index])	Plot the surface on the given axes.
`update_params`(new_dict)	Update surface parameters.

Attributes

pvfactors.geometry.base.PVSurface

class pvfactors.geometry.base.**PVSurface**(*coords=None*, *normal_vector=None*, *shaded=False*, *index=None*, *param_names=None*, *params=None*)

PV surfaces inherit from `BaseSurface`. The only difference is that PV surfaces have a `shaded` attribute.

__init__(*coords=None*, *normal_vector=None*, *shaded=False*, *index=None*, *param_names=None*, *params=None*)

Initialize PV surface.

Parameters

- **coords** (*list, optional*) – List of linestring coordinates for the surface
- **normal_vector** (*list, optional*) – Normal vector for the surface (Default = None, so will be calculated)
- **shaded** (*bool, optional*) – Flag telling if surface is shaded or not (Default = False)
- **index** (*int, optional*) – Surface index (Default = None)
- **param_names** (*list of str, optional*) – Names of the surface parameters, eg reflectivity, total incident irradiance, temperature, etc. (Default = None)
- **params** (*dict, optional*) – Surface float parameters (Default = None)

Methods

`__init__`([coords, normal_vector, shaded, ...])	Initialize PV surface.

Attributes

pvfactors.geometry.base.ShadeCollection

class pvfactors.geometry.base.**ShadeCollection**(*list_surfaces=None*, *shaded=None*, *param_names=None*)

A group of `PVSurface` objects that all have the same shading status. The PV surfaces are not necessarily contiguous or collinear.

__init__(*list_surfaces=None*, *shaded=None*, *param_names=None*)

Initialize shade collection.

Parameters

- **list_surfaces** (*list, optional*) – List of `PVSurface` object (Default = None)
- **shaded** (*bool, optional*) – Shading status of the collection. If not specified, will be derived from list of surfaces (Default = None)

- **param_names** (*list of str, optional*) – Names of the surface parameters, eg reflectivity, total incident irradiance, temperature, etc. (Default = None)

Methods

`__init__`([list_surfaces, shaded, param_names])	Initialize shade collection.
`add_linestring`(linestring[, normal_vector])	Add PV surface to the collection using a linestring
`add_pvsurface`(pvsurface)	Add PV surface to the collection.
`cut_at_point`(point)	Cut collection at point if the collection contains it.
`from_linestring_coords`(coords, shaded[, ...])	Create a shade collection with a single PV surface.
`get_param_weighted`(param)	Get the parameter from the collection's surfaces, after weighting by surface length.
`get_param_ww`(param)	Get the parameter from the collection's surfaces with weight, i.e. after multiplying by the surface lengths.
`merge_surfaces`()	Merge all surfaces in the shade collection into one contiguous surface, even if they're not contiguous, by using bounds.
`plot`(ax[, color, with_index])	Plot the surfaces in the shade collection.
`remove_linestring`(linestring)	Remove linestring from shade collection.
`update_geom_collection`(list_surfaces)	Force update of geometry collection, even if list is empty https://github.com/Toblerity/Shapely/blob/master/shapely/geometry/collection.py#L42
`update_params`(new_dict)	Update surface parameters in the collection.

Attributes

`n_surfaces`	Number of surfaces in collection.
`n_vector`	Unique normal vector of the shade collection, if it exists.
`surface_indices`	Indices of the surfaces in the collection.

pvfactors.geometry.base.PVSegment

class `pvfactors.geometry.base.PVSegment`(*illum_collection=<pvfactors.geometry.base.ShadeCollection object>, shaded_collection=<pvfactors.geometry.base.ShadeCollection object>, index=None*)

A PV segment will be a collection of 2 collinear and contiguous shade collections, a shaded one and an illuminated one. It inherits from `shapely.geometry.GeometryCollection` so that users can still call basic geometrical methods and properties on it, eg call length, etc.

`__init__`(*illum_collection=<pvfactors.geometry.base.ShadeCollection object>, shaded_collection=<pvfactors.geometry.base.ShadeCollection object>, index=None*)

Initialize PV segment.

Parameters

- **illum_collection** (*ShadeCollection*, optional) – Illuminated collection of the PV segment (Default = empty shade collection with no shading)

- **shaded_collection** (*ShadeCollection*, optional) – Shaded collection of the PV segment (Default = empty shade collection with shading)
- **index** (*int, optional*) – Index of the PV segment (Default = None)

Methods

`__init__`([illum_collection, ...])	Initialize PV segment.
`cast_shadow`(linestring)	Cast shadow on PV segment using linestring: will rearrange the PV surfaces between the shaded and illuminated collections of the segment
`cut_at_point`(point)	Cut PV segment at point if the segment contains it.
`from_linestring_coords`(coords[, shaded, ...])	Create a PV segment with a single PV surface.
`get_param_weighted`(param)	Get the parameter from the segment's surfaces, after weighting by surface length.
`get_param_ww`(param)	Get the parameter from the segment's surfaces with weight, i.e. after multiplying by the surface lengths.
`plot`(ax[, color_shaded, color_illum, with_index])	Plot the surfaces in the PV Segment.
`update_params`(new_dict)	Update surface parameters in the collection.

Attributes

`all_surfaces`	List of all the `pvfactors.geometry.base.PVSurface`
`illum_collection`	Illuminated collection of the PV segment.
`n_surfaces`	Number of surfaces in collection.
`n_vector`	Since shaded and illum surfaces are supposed to be collinear, this should return either surfaces' normal vector.
`shaded_collection`	Shaded collection of the PV segment
`shaded_length`	Length of the shaded collection of the PV segment.
`surface_indices`	Indices of the surfaces in the PV segment.

pvfactors.geometry.base.BaseSide

class `pvfactors.geometry.base.`**BaseSide**(*list_segments=None*)

A side represents a fixed collection of PV segments objects that should all be collinear, with the same normal vector

`__init__`(*list_segments=None*)

Create a side geometry.

Parameters **list_segments** (list of *PVSegment*, optional) – List of PV segments for side (Default = None)

Methods

__init__([list_segments])	Create a side geometry.
cast_shadow(linestring)	Cast shadow on Side using linestring: will rearrange the PV surfaces between the shaded and illuminated collections of the segments.
cut_at_point(point)	Cut Side at point if the side contains it.
from_linestring_coords(coords[, shaded, ...])	Create a Side with a single PV surface, or multiple discretized identical ones.
get_param_weighted(param)	Get the parameter from the side's surfaces, after weighting by surface length.
get_param_ww(param)	Get the parameter from the side's surfaces with weight, i.e. after multiplying by the surface lengths.
merge_shaded_areas()	Merge shaded areas of all PV segments
plot(ax[, color_shaded, color_illum, with_index])	Plot the surfaces in the Side object.
update_params(new_dict)	Update surface parameters in the Side.

Attributes

all_surfaces	List of all surfaces in the Side object.
n_surfaces	Number of surfaces in the Side object.
n_vector	Normal vector of the Side.
shaded_length	Shaded length of the Side.
surface_indices	List of all surface indices in the Side object.

pvfactors.geometry.base.BasePVArray

class pvfactors.geometry.base.**BasePVArray**(*axis_azimuth=None*)

Base class for PV arrays in pvfactors. Will provide basic capabilities.

__init__(*axis_azimuth=None*)

Initialize Base of PV array.

Parameters **axis_azimuth** (*float, optional*) – Azimuth angle of rotation axis [deg] (Default = None)

Methods

__init__([axis_azimuth])	Initialize Base of PV array.
fit(*args, **kwargs)	Not implemented.
plot_at_idx(idx, ax[, ...])	Plot all the PV rows and the ground in the PV array at a desired step index.
update_params(new_dict)	Update timeseries surface parameters in the collection.

Attributes

all_ts_surfaces	List of all timeseries surfaces in PV array
n_ts_surfaces	Number of timeseries surfaces in the PV array.
registry_cols	
ts_surface_indices	List of indices of all the timeseries surfaces

pvrow

Module will classes related to PV row geometries

TsPVRow	Timeseries PV row class: this class is a vectorized version of the PV row geometries.
TsSide	Timeseries side class: this class is a vectorized version of the BaseSide geometries.
TsSegment	A TsSegment is a timeseries segment that has a timeseries shaded collection and a timeseries illuminated collection.
PVRowSide	A PV row side represents the whole surface of one side of a PV row.
PVRow	A PV row is made of two PV row sides, a front and a back one.

pvfactors.geometry.pvrow.TsPVRow

class pvfactors.geometry.pvrow.**TsPVRow**(*ts_front_side*, *ts_back_side*, *xy_center*, *index=None*, *full_pvrow_coords=None*)

Timeseries PV row class: this class is a vectorized version of the PV row geometries. The coordinates and attributes (front and back sides) are all vectorized.

__init__(*ts_front_side*, *ts_back_side*, *xy_center*, *index=None*, *full_pvrow_coords=None*)

Initialize timeseries PV row with its front and back sides.

Parameters

- **ts_front_side** (*TsSide*) – Timeseries front side of the PV row
- **ts_back_side** (*TsSide*) – Timeseries back side of the PV row
- **xy_center** (*tuple of float*) – x and y coordinates of the PV row center point (invariant)
- **index** (*int, optional*) – index of the PV row (Default = None)
- **full_pvrow_coords** (*TsLineCoords*, optional) – Timeseries coordinates of the full PV row, end to end (Default = None)

Methods

`__init__`(ts_front_side, ts_back_side, xy_center)	Initialize timeseries PV row with its front and back sides.
`at`(idx)	Generate a PV row geometry for the desired index.
`from_raw_inputs`(xy_center, width, ...[, ...])	Create timeseries PV row using raw inputs.
`plot_at_idx`(idx, ax[, color_shaded, ...])	Plot timeseries PV row at a certain index.
`surfaces_at_idx`(idx)	Get all PV surface geometries in timeseries PV row for a certain index.
`update_params`(new_dict)	Update timeseries surface parameters of the PV row.

Attributes

`all_ts_surfaces`	List of all timeseries surfaces
`centroid`	Centroid point of the timeseries pv row
`highest_point`	Timeseries point coordinates of highest point of PV row
`length`	Length of both sides of the timeseries PV row
`n_ts_surfaces`	Number of timeseries surfaces in the ts PV row

pvfactors.geometry.pvrow.TsSide

class `pvfactors.geometry.pvrow.`**`TsSide`**(*segments*, *n_vector=None*)

Timeseries side class: this class is a vectorized version of the BaseSide geometries. The coordinates and attributes (list of segments, normal vector) are all vectorized.

`__init__`(*segments*, *n_vector=None*)

Initialize timeseries side using list of timeseries segments.

Parameters

- **`segments`** (list of *`TsSegment`*) – List of timeseries segments of the side
- **`n_vector`** (*`np.ndarray, optional`*) – Timeseries normal vectors of the side (Default = None)

Methods

`__init__`(segments[, n_vector])	Initialize timeseries side using list of timeseries segments.
`at`(idx)	Generate a side geometry for the desired index.
`from_raw_inputs`(xy_center, width, ...[, ...])	Create timeseries side using raw PV row inputs.
`get_param_weighted`(param)	Get timeseries parameter for the side, after weighting by surface length.
`get_param_ww`(param)	Get timeseries parameter from the side's surfaces with weight, i.e. after multiplying by the surface lengths.
`plot_at_idx`(idx, ax[, color_shaded, color_illum])	Plot timeseries side at a certain index.
`surfaces_at_idx`(idx)	Get all PV surface geometries in timeseries side for a certain index.
`update_params`(new_dict)	Update timeseries surface parameters of the side.

Attributes

`all_ts_surfaces`	List of all timeseries surfaces
`length`	Timeseries length of side.
`n_ts_surfaces`	Number of timeseries surfaces in the ts side
`shaded_length`	Timeseries shaded length of the side.

pvfactors.geometry.pvrow.TsSegment

class `pvfactors.geometry.pvrow.`**`TsSegment`**(*coords*, *illum_collection*, *shaded_collection*, *index=None*, *n_vector=None*)

A TsSegment is a timeseries segment that has a timeseries shaded collection and a timeseries illuminated collection.

`__init__`(*coords*, *illum_collection*, *shaded_collection*, *index=None*, *n_vector=None*)

Initialize timeseries segment using segment coordinates and timeseries illuminated and shaded surfaces.

Parameters

- **coords** (*TsLineCoords*) – Timeseries coordinates of full segment
- **illum_collection** (*TsShadeCollection*) – Timeseries collection for illuminated part of segment
- **shaded_collection** (*TsShadeCollection*) – Timeseries collection for shaded part of segment
- **index** (*int, optional*) – Index of segment (Default = None)
- **n_vector** (*np.ndarray, optional*) – Timeseries normal vectors of the side (Default = None)

Methods

`__init__`(coords, illum_collection, ...[, ...])	Initialize timeseries segment using segment coordinates and timeseries illuminated and shaded surfaces.
`at`(idx)	Generate a PV segment geometry for the desired index.
`get_param_weighted`(param)	Get timeseries parameter for the segment, after weighting by surface length.
`get_param_ww`(param)	Get timeseries parameter from the segment's surfaces with weight, i.e. after multiplying by the surface lengths.
`plot_at_idx`(idx, ax[, color_shaded, color_illum])	Plot timeseries segment at a certain index.
`surfaces_at_idx`(idx)	Get all PV surface geometries in timeseries segment for a certain index.
`update_params`(new_dict)	Update timeseries surface parameters of the segment.

Attributes

`all_ts_surfaces`	List of all timeseries surfaces in segment
`centroid`	Timeseries point coordinates of the segment's centroid
`highest_point`	Timeseries point coordinates of highest point of segment
`length`	Timeseries length of segment.
`lowest_point`	Timeseries point coordinates of lowest point of segment
`n_ts_surfaces`	Number of timeseries surfaces in the segment
`shaded_length`	Timeseries length of shaded part of segment.

pvfactors.geometry.pvrow.PVRowSide

class `pvfactors.geometry.pvrow.`**`PVRowSide`**(*list_segments=[]*)

A PV row side represents the whole surface of one side of a PV row. At its core it will contain a fixed number of *`PVSegment`* objects that will together constitue one side of a PV row: a PV row side can also be "discretized" into multiple segments

`__init__`(*list_segments=[]*)

Initialize PVRowSide using its base class *`pvfactors.geometry.base.BaseSide`*

Parameters **`list_segments`** (list of *`PVSegment`*) – List of PV segments for PV row side.

Methods

__init__([list_segments])	Initialize PVRowSide using its base class *pvfactors.geometry.base.BaseSide*

Attributes

pvfactors.geometry.pvrow.PVRow

class pvfactors.geometry.pvrow.**PVRow**(*front_side=<pvfactors.geometry.pvrow.PVRowSide object>*, *back_side=<pvfactors.geometry.pvrow.PVRowSide object>*, *index=None*, *original_linestring=None*)

A PV row is made of two PV row sides, a front and a back one.

__init__(*front_side=<pvfactors.geometry.pvrow.PVRowSide object>*, *back_side=<pvfactors.geometry.pvrow.PVRowSide object>*, *index=None*, *original_linestring=None*)

Initialize PV row.

Parameters

- **front_side** (*PVRowSide*, optional) – Front side of the PV Row (Default = Empty PVRowSide)
- **back_side** (*PVRowSide*, optional) – Back side of the PV Row (Default = Empty PVRowSide)
- **index** (*int, optional*) – Index of PV row (Default = None)
- **original_linestring** (shapely.geometry.LineString, optional) – Full continuous linestring that the PV row will be made of (Default = None)

Methods

__init__([front_side, back_side, index, ...])	Initialize PV row.
from_center_tilt_width(xy_center, tilt, ...)	Create a PV row using mainly the coordinates of the line center, a tilt angle, and its length.
from_linestring_coords(coords[, shaded, ...])	Create a PV row with a single PV surface and using linestring coordinates.
plot(ax[, color_shaded, color_illum, with_index])	Plot the surfaces of the PV Row.
update_params(new_dict)	Update surface parameters for both front and back sides.

Attributes

`all_surfaces`	List of all the surfaces in the PV row.
`boundary`	Boundaries of the PV Row's orginal linestring.
`highest_point`	Highest point of the PV Row.
`lowest_point`	Lowest point of the PV Row.
`surface_indices`	List of all surface indices in the PV Row.

pvground

Classes for implementation of ground geometry

`TsGround`	Timeseries ground class: this class is a vectorized version of the PV ground geometry class, and it will store timeseries shaded ground and illuminated ground elements, as well as pv row cut points.
`TsGroundElement`	Special class for timeseries ground elements: a ground element has known timeseries coordinate boundaries, but it will also have a break down of its area into n+1 timeseries surfaces located in the n+1 ground zones defined by the n ground cutting points.
`PVGround`	Class that defines the ground geometry in PV arrays.

pvfactors.geometry.pvground.TsGround

class `pvfactors.geometry.pvground.`**`TsGround`**(*shadow_elements*, *illum_elements*, *param_names=None*, *flag_overlap=None*, *cut_point_coords=None*, *y_ground=None*)

Timeseries ground class: this class is a vectorized version of the PV ground geometry class, and it will store timeseries shaded ground and illuminated ground elements, as well as pv row cut points.

`__init__`(*shadow_elements*, *illum_elements*, *param_names=None*, *flag_overlap=None*, *cut_point_coords=None*, *y_ground=None*)

Initialize timeseries ground using list of timeseries surfaces for the ground shadows

Parameters

- **shadow_elements** (list of *`TsGroundElement`*) – Timeseries shaded ground elements
- **illum_elements** (list of *`TsGroundElement`*) – Timeseries illuminated ground elements
- **param_names** (*list of str, optional*) – List of names of surface parameters to use when creating geometries (Default = None)
- **flag_overlap** (*list of bool, optional*) – Flags indicating if the ground shadows are overlapping, for all time steps (Default=None). I.e. is there direct shading on pv rows?
- **cut_point_coords** (list of *`TsPointCoords`*, optional) – List of cut point coordinates, as calculated for timeseries PV rows (Default = None)
- **y_ground** (*float, optional*) – Y coordinate of flat ground [m] (Default=None)

Methods

`__init__`(shadow_elements, illum_elements[, ...])	Initialize timeseries ground using list of timeseries surfaces for the ground shadows
`at`(idx[, x_min_max, merge_if_flag_overlap, ...])	Generate a PV ground geometry for the desired index.
`from_ordered_shadows_coords`(shadow_coords[, ...])	Create timeseries ground from list of ground shadow coordinates.
`from_ts_pvrows_and_angles`(list_ts_pvrows, ...)	Create timeseries ground from list of timeseries PV rows, and PV array and solar angles.
`get_param_weighted`(param)	Get timeseries parameter for the ts ground, after weighting by surface length.
`get_param_ww`(param)	Get timeseries parameter from the ground's surfaces with weight, i.e. after multiplying by the surface lengths.
`n_non_point_surfaces_at`(idx)	Return the number of `PVSurface` that are not points at given index
`non_point_illum_surfaces_at`(idx)	Return a list of illuminated surfaces, that are not points at given index
`non_point_shaded_surfaces_at`(idx)	Return a list of shaded surfaces, that are not points at given index
`non_point_surfaces_at`(idx)	Return a list of all surfaces that are not points at given index
`plot_at_idx`(idx, ax[, color_shaded, ...])	Plot timeseries ground at a certain index.
`shadow_coords_left_of_cut_point`(idx_cut_pt)	Get coordinates of shadows located on the left side of the cut point with given index.
`shadow_coords_right_of_cut_point`(idx_cut_pt)	Get coordinates of shadows located on the right side of the cut point with given index.
`ts_surfaces_side_of_cut_point`(side, idx_cut_pt)	Get a list of all the ts ground surfaces an a request side of a cut point
`update_illum_params`(new_dict)	Update the illuminated parameters with new ones, not only for the timeseries ground, but also for its ground elements and the timeseries surfaces of the ground elements, so that they are all synced.
`update_params`(new_dict)	Update the illuminated parameters with new ones, not only for the timeseries ground, but also for its ground elements and the timeseries surfaces of the ground elements, so that they are all synced.
`update_shaded_params`(new_dict)	Update the shaded parameters with new ones, not only for the timeseries ground, but also for its ground elements and the timeseries surfaces of the ground elements, so that they are all synced.

Attributes

all_ts_surfaces	Number of timeseries surfaces in the ts ground
length	Length of the timeseries ground
n_ts_illum_surfaces	Number of illuminated timeseries surfaces in the ts ground
n_ts_shaded_surfaces	Number of shaded timeseries surfaces in the ts ground
n_ts_surfaces	Number of timeseries surfaces in the ts ground
shaded_length	Length of the timeseries ground
x_max	
x_min	

pvfactors.geometry.pvground.TsGroundElement

class pvfactors.geometry.pvground.**TsGroundElement**(*coords*, *list_ordered_cut_pts_coords=None*, *param_names=None*, *shaded=False*)

Special class for timeseries ground elements: a ground element has known timeseries coordinate boundaries, but it will also have a break down of its area into n+1 timeseries surfaces located in the n+1 ground zones defined by the n ground cutting points. This is crucial to calculate view factors in a vectorized way.

__init__(*coords*, *list_ordered_cut_pts_coords=None*, *param_names=None*, *shaded=False*)

Initialize the timeseries ground element using its timeseries line coordinates, and build the timeseries surfaces for all the cut point zones.

Parameters

- **coords** (*TsLineCoords*) – Timeseries line coordinates of the ground element
- **list_ordered_cut_pts_coords** (*list, optional*) – List of all the cut point timeseries coordinates (Default = [])
- **param_names** (*list of str, optional*) – List of names of surface parameters to use when creating geometries (Default = None)
- **shaded** (*bool, optional*) – Flag specifying is element is a shadow or not (Default = False)

Methods

`__init__`(coords[, ...])	Initialize the timeseries ground element using its timeseries line coordinates, and build the timeseries surfaces for all the cut point zones.
`get_param_weighted`(param)	Get timeseries parameter for the ground element, after weighting by surface length.
`get_param_ww`(param)	Get timeseries parameter from the ground element with weight, i.e. after multiplying by the surface lengths.
`non_point_surfaces_at`(idx)	Return list of non-point surfaces (from left to right) at given index that make up the ground element.
`surfaces_at`(idx)	Return list of surfaces (from left to right) at given index that make up the ground element.

Attributes

`all_ts_surfaces`	List of all ts surfaces making up the ts ground element
`b1`	Timeseries coordinates of first boundary point
`b2`	Timeseries coordinates of second boundary point
`centroid`	Timeseries point coordinates of the element's centroid
`length`	Timeseries length of the ground

pvfactors.geometry.pvground.PVGround

class `pvfactors.geometry.pvground.PVGround`(*list_segments=None*, *original_linestring=None*)

Class that defines the ground geometry in PV arrays.

`__init__`(*list_segments=None*, *original_linestring=None*)

Initialize PV ground geometry.

Parameters

- **list_segments** (list of *PVSegment*, optional) – List of PV segments that will constitute the ground (Default = [])
- **original_linestring** (`shapely.geometry.LineString`, optional) – Full continuous linestring that the ground will be made of (Default = None)

Methods

`__init__`([list_segments, original_linestring])	Initialize PV ground geometry.
`as_flat`([x_min_max, shaded, y_ground, ...])	Build a horizontal flat ground surface, made of 1 PV segment.
`from_lists_surfaces`(list_shaded_surfaces, ...)	Create ground from lists of shaded and illuminated PV surfaces.

Attributes

boundary	Boundaries of the ground's original linestring.

pvarray

Module containing PV array classes, which will use PV rows and ground geometries.

OrderedPVArray	An ordered PV array has a flat horizontal ground, and pv rows which are all at the same height, with the same surface tilt and azimuth angles, and also all equally spaced.

pvfactors.geometry.pvarray.OrderedPVArray

class pvfactors.geometry.pvarray.**OrderedPVArray**(*axis_azimuth=None*, *gcr=None*, *pvrow_height=None*, *n_pvrows=None*, *pvrow_width=None*, *param_names=None*, *cut=None*)

An ordered PV array has a flat horizontal ground, and pv rows which are all at the same height, with the same surface tilt and azimuth angles, and also all equally spaced. These simplifications allow faster and easier calculations. In the ordered PV array, the list of PV rows must be ordered from left to right (along the x-axis) in the 2D geometry.

__init__(*axis_azimuth=None*, *gcr=None*, *pvrow_height=None*, *n_pvrows=None*, *pvrow_width=None*, *param_names=None*, *cut=None*)

Initialize ordered PV array. List of PV rows will be ordered from left to right.

Parameters

- **axis_azimuth** (*float, optional*) – Azimuth angle of rotation axis [deg] (Default = None)
- **gcr** (*float, optional*) – Ground coverage ratio (Default = None)
- **pvrow_height** (*float, optional*) – Unique height of all PV rows in [m] (Default = None)
- **n_pvrows** (*int, optional*) – Number of PV rows in the PV array (Default = None)
- **pvrow_width** (*float, optional*) – Width of the PV rows in the 2D plane in [m] (Default = None)
- **param_names** (*list of str, optional*) – List of surface parameter names for the PV surfaces (Default = None)
- **cut** (*dict, optional*) – Nested dictionary that tells if some PV row sides need to be discretized, and how (Default = None). Example: {1: {'front': 5}}, will create 5 segments on the front side of the PV row with index 1

Methods

__init__([axis_azimuth, gcr, pvrow_height, ...])	Initialize ordered PV array.
fit(solar_zenith, solar_azimuth, ...)	Fit the ordered PV array to the list of solar and surface angles.
fit_from_dict_of_scalars(pvarray_params[, ...])	Instantiate, and fit ordered PV array using dictionary of scalar inputs.
init_from_dict(pvarray_params[, param_names])	Instantiate ordered PV array from dictionary of parameters

Attributes

y_ground

timeseries

Timeseries geometry tools. They allow the vectorization of geometry calculations.

TsShadeCollection	Collection of timeseries surfaces that are all either shaded or illuminated.
TsSurface	Timeseries surface class: vectorized representation of PV surface geometries.
TsLineCoords	Timeseries line coordinates class: will provide a helpful shapely-like API to invoke timeseries coordinates.
TsPointCoords	Timeseries point coordinates: provides a shapely-like API for timeseries point coordinates.

pvfactors.geometry.timeseries.TsShadeCollection

class pvfactors.geometry.timeseries.**TsShadeCollection**(*list_ts_surfaces*, *shaded*)

Collection of timeseries surfaces that are all either shaded or illuminated. This will be used by both ground and PV row geometries.

__init__(*list_ts_surfaces*, *shaded*)

Initialize using list of surfaces and shading status

Parameters

- **list_ts_surfaces** (list of *TsSurface*) – List of timeseries surfaces in collection
- **shaded** (*bool*) – Shading status of the collection

Methods

`__init__`(list_ts_surfaces, shaded)	Initialize using list of surfaces and shading status
`at`(idx)	Generate a ponctual shade collection for the desired index.
`get_param_weighted`(param)	Get timeseries parameter for the collection, after weighting by surface length.
`get_param_ww`(param)	Get timeseries parameter from the collection with weight, i.e. after multiplying by the surface lengths.
`update_params`(new_dict)	Update timeseries surface parameters of the segment.

Attributes

`length`	Total length of the collection
`list_ts_surfaces`	List of timeseries surfaces in collection
`n_ts_surfaces`	Number of timeseries surfaces in the collection

pvfactors.geometry.timeseries.TsSurface

class `pvfactors.geometry.timeseries.`**`TsSurface`**(*coords*, *n_vector=None*, *param_names=None*, *index=None*, *shaded=False*)

Timeseries surface class: vectorized representation of PV surface geometries.

`__init__`(*coords*, *n_vector=None*, *param_names=None*, *index=None*, *shaded=False*)

Initialize timeseries surface using timeseries coordinates.

Parameters

- **coords** (*TsLineCoords*) – Timeseries coordinates of full segment
- **index** (*int, optional*) – Index of segment (Default = None)
- **n_vector** (*np.ndarray, optional*) – Timeseries normal vectors of the side (Default = None)
- **index** – Index of the timeseries surfaces (Default = None)
- **shaded** (*bool, optional*) – Is the surface shaded or not (Default = False)

Methods

`__init__`(coords[, n_vector, param_names, ...])	Initialize timeseries surface using timeseries coordinates.
`at`(idx)	Generate a PV segment geometry for the desired index.
`get_param`(param)	Get timeseries parameter values of surface
`plot_at_idx`(idx, ax, color)	Plot timeseries PV row at a certain index, only if it's not too small.
`update_params`(new_dict)	Update timeseries surface parameters.

Attributes

`b1`	Timeseries coordinates of first boundary point
`b2`	Timeseries coordinates of second boundary point
`centroid`	Timeseries point coordinates of the surface's centroid
`highest_point`	Timeseries point coordinates of highest point of surface
`is_empty`	Check if surface is "empty" by checking if its length is always zero
`length`	Timeseries length of the surface
`lowest_point`	Timeseries point coordinates of lowest point of surface
`u_vector`	Vector orthogonal to the surface's normal vector

pvfactors.geometry.timeseries.TsLineCoords

class `pvfactors.geometry.timeseries.`**`TsLineCoords`**(*b1_ts_coords*, *b2_ts_coords*, *coords=None*)

Timeseries line coordinates class: will provide a helpful shapely-like API to invoke timeseries coordinates.

`__init__`(*b1_ts_coords*, *b2_ts_coords*, *coords=None*)

Initialize timeseries line coordinates using the timeseries coordinates of its boundaries.

Parameters

- **b1_ts_coords** (*TsPointCoords*) – Timeseries coordinates of first boundary point
- **b2_ts_coords** (*TsPointCoords*) – Timeseries coordinates of second boundary point
- **coords** (*np.ndarray, optional*) – Timeseries coordinates as numpy array

Methods

`__init__`(b1_ts_coords, b2_ts_coords[, coords])	Initialize timeseries line coordinates using the timeseries coordinates of its boundaries.
`at`(idx)	Get coordinates at a given index
`from_array`(coords_array)	Create timeseries line coordinates from numpy array of coordinates.

Attributes

`as_array`	Timeseries line coordinates as numpy array
`centroid`	Timeseries point coordinates of the line coordinates
`highest_point`	Timeseries point coordinates of highest point of timeseries line coords
`length`	Timeseries length of the line.
`lowest_point`	Timeseries point coordinates of lowest point of timeseries line coords

pvfactors.geometry.timeseries.TsPointCoords

`class pvfactors.geometry.timeseries.TsPointCoords(x, y)`

Timeseries point coordinates: provides a shapely-like API for timeseries point coordinates.

`__init__(x, y)`

Initialize timeseries point coordinates using numpy array of coords.

Parameters

- **x** (*np.ndarray*) – Timeseries x coordinates
- **y** (*np.ndarray*) – Timeseries y coordinates

Methods

__init__(x, y)	Initialize timeseries point coordinates using numpy array of coords.
at(idx)	Get coordinates at a given index
from_array(coords_array)	Create timeseries point coords from numpy array of coordinates.

Attributes

as_array	Timeseries point coordinates as numpy array

2.5.2 viewfactors

The viewfactors sub-package of pvfactors implements the methods used to calculate view factors from pvfactors PV array objects.

calculator

Module with classes and functions to calculate views and view factors

VFCalculator	This calculator class will be used for the calculation of view factors for *OrderedPVArray*, and it will rely on both *VFTsMethods* and *AOIMethods*

pvfactors.viewfactors.calculator.VFCalculator

`class pvfactors.viewfactors.calculator.VFCalculator(faoi_fn_front=None, faoi_fn_back=None, n_aoi_integral_sections=300)`

This calculator class will be used for the calculation of view factors for *OrderedPVArray*, and it will rely on both *VFTsMethods* and *AOIMethods*

__init__(*faoi_fn_front=None*, *faoi_fn_back=None*, *n_aoi_integral_sections=300*)

Initialize the view factor calculator with the calculation methods that will be used. The AOI methods will not be instantiated if an fAOI function is missing.

Parameters

- **faoi_fn_front** (*function or object, optional*) – Function (or object containing `faoi` method) which takes a list (or numpy array) of incidence angles measured from the surface horizontal (with values from 0 to 180 deg) and returns the fAOI values for the front side of PV rows (default = None)
- **faoi_fn_back** (*function or object, optional*) – Function (or object containing `faoi` method) which takes a list (or numpy array) of incidence angles measured from the surface horizontal (with values from 0 to 180 deg) and returns the fAOI values for the back side of PV rows (default = None)
- **n_integral_sections** (*int, optional*) – Number of integral divisions of the 0 to 180 deg interval to use for the fAOI loss integral (default = 300)

Methods

`__init__`([faoi_fn_front, faoi_fn_back, ...])	Initialize the view factor calculator with the calculation methods that will be used.
`build_ts_vf_aoi_matrix`(pvarray, rho_mat)	Calculate the view factor aoi matrix elements from all PV row surfaces to all other surfaces, only.
`build_ts_vf_matrix`(pvarray)	Calculate timeseries view factor matrix for the given ordered pv array
`fit`(n_timestamps)	Fit the view factor calculator to the timeseries inputs.
`get_vf_ts_pvrow_element`(pvrow_idx, ...)	Calculate timeseries view factors of timeseries pvrow element (segment or surface) to all other elements of the PV array.

timeseries view factor methods

Module with view factor calculation tools

`VFTsMethods`	This class contains all the methods used to calculate timeseries view factors for all the surfaces in `OrderedPVArray`

pvfactors.viewfactors.vfmethods.VFTsMethods

class pvfactors.viewfactors.vfmethods.**VFTsMethods**

This class contains all the methods used to calculate timeseries view factors for all the surfaces in `OrderedPVArray`

__init__()

Methods

`__init__`()	
`calculate_vf_to_gnd`(pvrow_element_coords, ...)	Calculate view factors from timeseries pvrow_element to the entire ground.
`calculate_vf_to_pvrow`(pvrow_element_coords, ...)	Calculate view factors from timeseries pvrow element to timeseries PV rows around it.
`calculate_vf_to_shadow_obstruction_hottel`(...)	Calculate view factors from timeseries pvrow_element to the shadow of a specific timeseries PV row which is casted on the ground.
`vf_pvrow_gnd_surf`(ts_pvrows, ts_ground, ...)	Calculate the view factors between timeseries PV row and ground surfaces, and assign it to the passed view factor matrix using the surface indices.
`vf_pvrow_surf_to_gnd_surf_obstruction_hottel`(...)	Calculate view factors from timeseries PV row surface to a timeseries ground surface.
`vf_pvrow_to_pvrow`(ts_pvrows, tilted_to_left, ...)	Calculate the view factors between timeseries PV row surfaces, and assign values to the passed view factor matrix using the surface indices.

view factor aoi methods

Module containing AOI loss calculation methods

AOIMethods	Class containing methods related to calculating AOI losses for *OrderedPVArray* objects.

pvfactors.viewfactors.aoimethods.AOIMethods

class `pvfactors.viewfactors.aoimethods.`**`AOIMethods`**(*faoi_fn_front*, *faoi_fn_back*, *n_integral_sections=300*)

Class containing methods related to calculating AOI losses for *OrderedPVArray* objects.

`__init__`(*faoi_fn_front*, *faoi_fn_back*, *n_integral_sections=300*)

Instantiate class with faoi function and number of sections to use to calculate integrals of view factors with faoi losses

Parameters

- **faoi_fn_front** (*function*) – Function which takes a list (or numpy array) of incidence angles measured from the surface horizontal (with values from 0 to 180 deg) and returns the fAOI values for the front side of PV rows
- **faoi_fn_back** (*function*) – Function which takes a list (or numpy array) of incidence angles measured from the surface horizontal (with values from 0 to 180 deg) and returns the fAOI values for the back side of PV rows
- **n_integral_sections** (*int, optional*) – Number of integral divisions of the 0 to 180 deg interval to use for the fAOI loss integral (default = 300)

Methods

`__init__`(faoi_fn_front, faoi_fn_back[, ...])	Instantiate class with faoi function and number of sections to use to calculate integrals of view factors with faoi losses
`fit`(n_timestamps)	Fit the AOI methods to timeseries inputs: create all the necessary integration attributes.
`rho_from_faoi_fn`(is_back)	Calculate global average reflectivity from faoi function for either side of the PV row (requires calculating view factors)
`vf_aoi_pvrow_to_gnd`(ts_pvrows, ts_ground, ...)	Calculate the view factors between timeseries PV row and ground surfaces while accounting for non-diffuse AOI losses, and assign it to the passed view factor aoi matrix using the surface indices.
`vf_aoi_pvrow_to_pvrow`(ts_pvrows, ...)	Calculate the view factors between timeseries PV row surfaces while accounting for AOI losses, and assign values to the passed view factor matrix using the surface indices.
`vf_aoi_pvrow_to_sky`(ts_pvrows, ts_ground, ...)	Calculate the view factors between timeseries PV row surface and sky while accounting for AOI losses, and assign values to the passed view factor matrix using the surface indices.

2.5.3 irradiance

The irradiance sub-package of pvfactors implements all irradiance related models and methods that can be applied to pvfactors PV array objects.

base

Module with Base classes for irradiance models

`BaseModel`	Base class for irradiance models

pvfactors.irradiance.base.BaseModel

class `pvfactors.irradiance.base.BaseModel`

Base class for irradiance models

`__init__()`

Methods

`__init__()`	
`fit(*args, **kwargs)`	Not implemented
`get_full_modeling_vectors(*args, **kwargs)`	Not implemented
`get_summed_components(pvarray[, absorbed])`	Get sum of irradiance components for irradiance model, either absorbed or only incident.
`get_ts_modeling_vectors(pvarray)`	Get matrices of summed up irradiance values from a PV array, as well as the inverse reflectivity values (the latter need to be named "inv_rho"), and the total perez irradiance values.
`initialize_rho(rho_scalar, rho_calculated, ...)`	Initialize reflectivity value: - if a scalar value is passed, use it - otherwise try to use calculated value - else use default value
`transform(*args, **kwargs)`	Not implemented
`update_ts_surface_sky_term(ts_surface[, ...])`	Update the 'sky_term' parameter of a timeseries surface.

Attributes

`cats`	
`gnd_illum`	Not implemented
`gnd_shaded`	Not implemented
`irradiance_comp`	
`params`	
`pvrow_illum`	Not implemented
`pvrow_shaded`	Not implemented
`sky_luminance`	Not implemented

models

Module containing irradiance models used with pv array geometries

IsotropicOrdered	Diffuse isotropic sky model for *OrderedPVArray*.
HybridPerezOrdered	Model is based off Perez diffuse light model, and applied to pvfactors *OrderedPVArray* objects.

pvfactors.irradiance.models.IsotropicOrdered

class pvfactors.irradiance.models.**IsotropicOrdered**(*rho_front=0.01*, *rho_back=0.03*, *module_transparency=0.0*, *module_spacing_ratio=0.0*, *faoi_fn_front=None*, *faoi_fn_back=None*)

Diffuse isotropic sky model for `OrderedPVArray`. It will calculate the appropriate values for an isotropic sky dome and apply it to the PV array.

__init__(*rho_front=0.01*, *rho_back=0.03*, *module_transparency=0.0*, *module_spacing_ratio=0.0*, *faoi_fn_front=None*, *faoi_fn_back=None*)

Initialize irradiance model values that will be saved later on.

Parameters

- **rho_front** (*float, optional*) – Reflectivity of the front side of the PV rows (default = 0.01)
- **rho_back** (*float, optional*) – Reflectivity of the back side of the PV rows (default = 0.03)
- **module_transparency** (*float, optional*) – Module transparency (from 0 to 1), which will let some direct light pass through the PV modules in the PV rows and reach the shaded ground (Default = 0., fully opaque)
- **module_spacing_ratio** (*float, optional*) – Module spacing ratio (from 0 to 1), which is the ratio of the area covered by the space between PV modules over the total area of the PV rows, and which determines how much direct light will reach the shaded ground through the PV rows (Default = 0., no spacing at all)
- **faoi_fn_front** (*function or object, optional*) – Function (or object containing `faoi` method) which takes a list (or numpy array) of incidence angles measured from the surface horizontal (with values from 0 to 180 deg) and returns the fAOI values for the front side of PV rows (default = None)
- **faoi_fn_back** (*function or object, optional*) – Function (or object containing `faoi` method) which takes a list (or numpy array) of incidence angles measured from the surface horizontal (with values from 0 to 180 deg) and returns the fAOI values for the back side of PV rows (default = None)

Methods

__init__([rho_front, rho_back, ...])	Initialize irradiance model values that will be saved later on.
`fit`(timestamps, DNI, DHI, solar_zenith, ...)	Use vectorization to calculate values used for the isotropic irradiance model.
`get_full_modeling_vectors`(pvarray, idx)	Get the modeling vectors used in matrix calculations of mathematical model.
`get_full_ts_modeling_vectors`(pvarray)	Get the modeling vectors used in matrix calculations of the mathematical model, including the sky values.
`get_summed_components`(pvarray[, absorbed])	Get sum of irradiance components for irradiance model, either absorbed or only incident.
`get_ts_modeling_vectors`(pvarray)	Get matrices of summed up irradiance values from a PV array, as well as the inverse reflectivity values (the latter need to be named "inv_rho"), and the total perez irradiance values.
`initialize_rho`(rho_scalar, rho_calculated, ...)	Initialize reflectivity value: - if a scalar value is passed, use it - otherwise try to use calculated value - else use default value
`transform`(pvarray)	Apply calculated irradiance values to PV array time-series geometries: assign values as parameters to timeseries surfaces.
`update_ts_surface_sky_term`(ts_surface[, ...])	Update the 'sky_term' parameter of a timeseries surface.

Attributes

`cats`	
`gnd_illum`	Total timeseries irradiance incident on ground illuminated areas
`gnd_shaded`	Total timeseries irradiance incident on ground shaded areas
`irradiance_comp`	
`irradiance_comp_absorbed`	
`params`	
`pvrow_illum`	Total timeseries irradiance incident on PV row's front shaded areas and calculated by Perez transposition
`pvrow_shaded`	Total timeseries irradiance incident on PV row's front illuminated areas and calculated by Perez transposition
`sky_luminance`	Total timeseries isotropic luminance of sky

pvfactors.irradiance.models.HybridPerezOrdered

class pvfactors.irradiance.models.**HybridPerezOrdered**(*horizon_band_angle=6.5*, *circumsolar_angle=30.0*, *circumsolar_model='uniform_disk'*, *rho_front=0.01*, *rho_back=0.03*, *module_transparency=0.0*, *module_spacing_ratio=0.0*, *faoi_fn_front=None*, *faoi_fn_back=None*)

Model is based off Perez diffuse light model, and applied to pvfactors `OrderedPVArray` objects. The model applies direct, circumsolar, and horizon irradiance to the PV array surfaces.

__init__(*horizon_band_angle=6.5*, *circumsolar_angle=30.0*, *circumsolar_model='uniform_disk'*, *rho_front=0.01*, *rho_back=0.03*, *module_transparency=0.0*, *module_spacing_ratio=0.0*, *faoi_fn_front=None*, *faoi_fn_back=None*)

Initialize irradiance model values that will be saved later on.

Parameters

- **horizon_band_angle** (*float, optional*) – Width of the horizon band in [deg] (Default = DEFAULT_HORIZON_BAND_ANGLE)
- **circumsolar_angle** (*float, optional*) – Diameter of the circumsolar area in [deg] (Default = DEFAULT_CIRCUMSOLAR_ANGLE)
- **circumsolar_model** (*str*) – Circumsolar shading model to use (Default = 'uniform_disk')
- **rho_front** (*float, optional*) – Reflectivity of the front side of the PV rows (default = 0.01)
- **rho_back** (*float, optional*) – Reflectivity of the back side of the PV rows (default = 0.03)
- **module_transparency** (*float, optional*) – Module transparency (from 0 to 1), which will let some direct light pass through the PV modules in the PV rows and reach the shaded ground (Default = 0., fully opaque)
- **module_spacing_ratio** (*float, optional*) – Module spacing ratio (from 0 to 1), which is the ratio of the area covered by the space between PV modules over the total area of the PV rows, and which determines how much direct light will reach the shaded ground through the PV rows (Default = 0., no spacing at all)
- **faoi_fn_front** (*function or object, optional*) – Function (or object containing `faoi` method) which takes a list (or numpy array) of incidence angles measured from the surface horizontal (with values from 0 to 180 deg) and returns the fAOI values for the front side of PV rows (default = None)
- **faoi_fn_back** (*function or object, optional*) – Function (or object containing `faoi` method) which takes a list (or numpy array) of incidence angles measured from the surface horizontal (with values from 0 to 180 deg) and returns the fAOI values for the back side of PV rows (default = None)

Methods

__init__([horizon_band_angle, ...])	Initialize irradiance model values that will be saved later on.
fit(timestamps, DNI, DHI, solar_zenith, ...)	Use vectorization to calculate values used for the hybrid Perez irradiance model.
get_full_modeling_vectors(pvarray, idx)	Get the modeling vectors used in matrix calculations of mathematical model.
get_full_ts_modeling_vectors(pvarray)	Get the modeling vectors used in matrix calculations of the mathematical model, including the sky values.
get_summed_components(pvarray[, absorbed])	Get sum of irradiance components for irradiance model, either absorbed or only incident.
get_ts_modeling_vectors(pvarray)	Get matrices of summed up irradiance values from a PV array, as well as the inverse reflectivity values (the latter need to be named "inv_rho"), and the total perez irradiance values.
initialize_rho(rho_scalar, rho_calculated, ...)	Initialize reflectivity value: - if a scalar value is passed, use it - otherwise try to use calculated value - else use default value
transform(pvarray)	Apply calculated irradiance values to PV array timeseries geometries: assign values as parameters to timeseries surfaces.
update_ts_surface_sky_term(ts_surface[, ...])	Update the 'sky_term' parameter of a timeseries surface.

Attributes

cats	
gnd_illum	Total timeseries irradiance incident on ground illuminated areas
gnd_shaded	Total timeseries irradiance incident on ground shaded areas
irradiance_comp	
irradiance_comp_absorbed	
params	
pvrow_illum	Total timeseries irradiance incident on PV row's front shaded areas and calculated by Perez transposition
pvrow_shaded	Total timeseries irradiance incident on PV row's front illuminated areas and calculated by Perez transposition
sky_luminance	Total timeseries isotropic luminance of sky

2.5.4 engine

This module contains the engine class that will run the complete timeseries simulations.

PVEngine	Class putting all of the calculations together into simple workflows.

pvfactors.engine.PVEngine

class pvfactors.engine.**PVEngine**(*pvarray*, *vf_calculator=None*, *irradiance_model=None*, *fast_mode_pvrow_index=None*, *fast_mode_segment_index=None*)

Class putting all of the calculations together into simple workflows.

__init__(*pvarray*, *vf_calculator=None*, *irradiance_model=None*, *fast_mode_pvrow_index=None*, *fast_mode_segment_index=None*)

Create pv engine class, and initialize timeseries parameters.

Parameters

- **pvarray** (BasePVArray *(or child) object*) – The initialized PV array object that will be used for calculations
- **vf_calculator** (*vf calculator object, optional*) – Calculator that will be used to calculate the view factor matrices, will use *VFCalculator* if None (Default = None)
- **irradiance_model** (*irradiance model object, optional*) – The irradiance model that will be applied to the PV array, will use *HybridPerezOrdered* if None (Default = None)
- **fast_mode_pvrow_index** (*int, optional*) – If a pvrow index is passed, then the PVEngine fast mode will be activated and the engine calculation will be done only for the back surface of the pvrow with the corresponding index (Default = None)
- **fast_mode_segment_index** (*int, optional*) – If a segment index is passed, then the PVEngine fast mode will calculate back surface irradiance only for the selected segment of the selected back surface (Default = None)

Methods

__init__(pvarray[, vf_calculator, ...])	Create pv engine class, and initialize timeseries parameters.
fit(timestamps, DNI, DHI, solar_zenith, ...)	Fit the timeseries data to the engine.
run_fast_mode([fn_build_report, ...])	Run all simulation timesteps using the fast mode for the back surface of a PV row, and assuming that the incident irradiance on all other surfaces is known (all but back surfaces).
run_full_mode([fn_build_report])	Run all simulation timesteps using the full mode, which calculates the equilibrium of reflections in the system, and returns a report that will be built by the function passed by the user.
with_rho_initialization(pvarray, ...[, ...])	Before creating the PV engine object, update the front and back reflectivity scalars using the faoi functions, if those values weren't passed originally

2.5.5 run

Module containing the functions to run engine calculations in normal or parallel mode.

run_timeseries_engine	Run timeseries simulation without multiprocessing.
run_parallel_engine	Run timeseries simulation using multiprocessing.

pvfactors.run.run_timeseries_engine

pvfactors.run.**run_timeseries_engine**(*fn_build_report*, *pvarray_parameters*, *timestamps*, *dni*, *dhi*, *solar_zenith*, *solar_azimuth*, *surface_tilt*, *surface_azimuth*, *albedo*, *cls_pvarray=<class 'pvfactors.geometry.pvarray.OrderedPVArray'>*, *cls_engine=<class 'pvfactors.engine.PVEngine'>*, *cls_irradiance=<class 'pvfactors.irradiance.models.HybridPerezOrdered'>*, *cls_vf=<class 'pvfactors.viewfactors.calculator.VFCalculator'>*, *fast_mode_pvrow_index=None*, *fast_mode_segment_index=None*, *irradiance_model_params=None*, *vf_calculator_params=None*, *ghi=None*)

Run timeseries simulation without multiprocessing. This is the functional approach to the *PVEngine* class.

Parameters

- **fn_build_report** (*function*) – Function that will build the report of the simulation
- **pvarray_parameters** (*dict*) – The parameters defining the PV array
- **timestamps** (*array-like*) – List of timestamps of the simulation.
- **dni** (*array-like*) – Direct normal irradiance values [W/m2]
- **dhi** (*array-like*) – Diffuse horizontal irradiance values [W/m2]
- **solar_zenith** (*array-like*) – Solar zenith angles [deg]
- **solar_azimuth** (*array-like*) – Solar azimuth angles [deg]
- **surface_tilt** (*array-like*) – Surface tilt angles, from 0 to 180 [deg]
- **surface_azimuth** (*array-like*) – Surface azimuth angles [deg]
- **albedo** (*array-like*) – Albedo values (or ground reflectivity)
- **cls_pvarray** (*class of PV array, optional*) – Class that will be used to build the PV array (Default = *OrderedPVArray* class)
- **cls_engine** (*class of PV engine, optional*) – Class of the engine to use to run the simulations (Default = *PVEngine* class)
- **cls_irradiance** (*class of irradiance model, optional*) – The irradiance model that will be applied to the PV array (Default = *HybridPerezOrdered* class)
- **cls_vf** (*class of VF calculator, optional*) – Calculator that will be used to calculate the view factor matrices (Default = *VFCalculator* class)
- **fast_mode_pvrow_index** (*int, optional*) – If a valid pvrow index is passed, then the PVEngine fast mode will be activated and the engine calculation will be done only for the back surface of the selected pvrow (Default = None)

- **fast_mode_segment_index** (*int, optional*) – If a segment index is passed, then the PVEngine fast mode will calculate back surface irradiance only for the selected segment of the selected back surface (Default = None)
- **irradiance_model_params** (*dict, optional*) – Dictionary of parameters that will be passed to the irradiance model class as kwargs at instantiation (Default = None)
- **vf_calculator_params** (*dict, optional*) – Dictionary of parameters that will be passed to the VF calculator class as kwargs at instantiation (Default = None)
- **ghi** (*array-like, optional*) – Global horizontal irradiance values [W/m2] (Default = None)

Returns Saved results from the simulation, as specified by user's report function

Return type report

pvfactors.run.run_parallel_engine

pvfactors.run.**run_parallel_engine**(*report_builder*, *pvarray_parameters*, *timestamps*, *dni*, *dhi*, *solar_zenith*, *solar_azimuth*, *surface_tilt*, *surface_azimuth*, *albedo*, *cls_pvarray=<class 'pvfactors.geometry.pvarray.OrderedPVArray'>*, *cls_engine=<class 'pvfactors.engine.PVEngine'>*, *cls_irradiance=<class 'pvfactors.irradiance.models.HybridPerezOrdered'>*, *cls_vf=<class 'pvfactors.viewfactors.calculator.VFCalculator'>*, *fast_mode_pvrow_index=None*, *fast_mode_segment_index=None*, *irradiance_model_params=None*, *vf_calculator_params=None*, *n_processes=2*, *ghi=None*)

Run timeseries simulation using multiprocessing. Here, instead of a function that will build the report, the users will need to pass a class (or an object).

Parameters

- **report_builder** (*class or object*) – Class or object that will build and merge the reports. It **must** have a `build()` and a `merge()` method that perform the tasks
- **pvarray_parameters** (*dict*) – The parameters defining the PV array
- **timestamps** (*array-like*) – List of timestamps of the simulation.
- **dni** (*array-like*) – Direct normal irradiance values [W/m2]
- **dhi** (*array-like*) – Diffuse horizontal irradiance values [W/m2]
- **solar_zenith** (*array-like*) – Solar zenith angles [deg]
- **solar_azimuth** (*array-like*) – Solar azimuth angles [deg]
- **surface_tilt** (*array-like*) – Surface tilt angles, from 0 to 180 [deg]
- **surface_azimuth** (*array-like*) – Surface azimuth angles [deg]
- **albedo** (*array-like*) – Albedo values (or ground reflectivity)
- **cls_pvarray** (*class of PV array, optional*) – Class that will be used to build the PV array (Default = *OrderedPVArray* class)
- **cls_engine** (*class of PV engine, optional*) – Class of the engine to use to run the simulations (Default = *PVEngine* class)

- **cls_irradiance** (*class of irradiance model, optional*) – The irradiance model that will be applied to the PV array (Default = *HybridPerezOrdered* class)
- **cls_vf** (*class of VF calculator, optional*) – Calculator that will be used to calculate the view factor matrices (Default = *VFCalculator* class)
- **fast_mode_pvrow_index** (*int, optional*) – If a valid pvrow index is passed, then the PVEngine fast mode will be activated and the engine calculation will be done only for the back surface of the selected pvrow (Default = None)
- **fast_mode_segment_index** (*int, optional*) – If a segment index is passed, then the PVEngine fast mode will calculate back surface irradiance only for the selected segment of the selected back surface (Default = None)
- **irradiance_model_params** (*dict, optional*) – Dictionary of parameters that will be passed to the irradiance model class as kwargs at instantiation (Default = None)
- **vf_calculator_params** (*dict, optional*) – Dictionary of parameters that will be passed to the VF calculator class as kwargs at instantiation (Default = None)
- **n_processes** (*int, optional*) – Number of parallel processes to run for the calculation (Default = 2)
- **ghi** (*array-like, optional*) – Global horizontal irradiance values [W/m2] (Default = None)

Returns Saved results from the simulation, as specified by user's report class (or object)

Return type report

2.5.6 report

Module containing examples of report builder functions and classes.

example_fn_build_report	Example function that builds a report when used in the *PVEngine* with full or fast mode simulations.
ExampleReportBuilder	A class is required to build reports when running calculations with multiprocessing because of python constraints

pvfactors.report.example_fn_build_report

pvfactors.report.**example_fn_build_report**(*pvarray*)

Example function that builds a report when used in the *PVEngine* with full or fast mode simulations. Here it will be a dictionary with lists of calculated values.

Parameters **pvarray** (*PV array object*) – PV array with updated calculation values

Returns **report** – Report updated with newly calculated values

Return type dict

pvfactors.report.ExampleReportBuilder

`class pvfactors.report.ExampleReportBuilder`

A class is required to build reports when running calculations with multiprocessing because of python constraints

`__init__()`

Methods

`__init__()`	
`build(pvarray)`	Method that will build the simulation report, using `example_fn_build_report()`.
`merge(reports)`	Method used to merge multiple reports together.

2.6 What's New

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

2.6.1 v1.5.3 (June 30, 2023)

This is the first release of the `solarfactors` fork.

Installation

- The `docs` and `testing` extras in `setup.py` are now called `doc` and `test` (GH1)

Requirements

- Removed the upper version limit on `pvlib` (GH5)

Testing

- Migrated CI infrastructure to GitHub Actions (GH1)
- Add python3.10 to test configuration (PR #129)
- Add python3.11 to test configuration (GH1)

Contributors

- Kevin Anderson (@kandersolar)

2.6.2 v1.5.2 (February 22, 2022)

Requirements

- Add python 3.9 to test configuration (PR #122)
- Set the upper bound on shapely to version 2.0 (not yet released). The shapely dependency may be dropped altogether in a future pvfactors release. (PR #130)

Fixes

- A small bug in the pvlib-python implementation of the Perez transposition model was discovered and fixed in pvlib v0.9.0. To ensure the error does not affect pvfactors output moving forward, the pvlib dependency is updated from `pvlib>=0.7.0,<0.9.0` to `pvlib>=0.9.0,<0.10.0`. This will likely change the results of irradiance simulations. According to the pvlib release notes, the differences are "expected to be small and primarily occur at low irradiance conditions". (PR #121)
- Fixed a bug that affected some irradiance simulations when *surface_tilt* is exactly zero. See GH #125 for details. (PR #128)

Maintenance

- Update CI including sphinx for documentation (PR #124)
- Add documentation for making new releases (PR #133)

Contributors

- Kevin Anderson (@kanderso-nrel)
- Marc Anoma (@anomam)
- Mark Campanelli (@campanelli-sunpower)

2.6.3 v1.5.1 (March 27, 2021)

Enhancements

- Update pvlib dependency from `pvlib>=0.6.0,<0.8.0` to `pvlib>=0.7.0,<0.9.0` (PR #116)

Contributors

- Marc Anoma (@anomam)
- Kevin Anderson (@kanderso-nrel)

2.6.4 v1.5.0 (February 7, 2021)

Enhancements

- Add import check for shapely/geos (#110)
- Drop Python 2.7, 3.5, add Python 3.8 (#112)

Fix

- TsSegement was missing proper indexing (#102)
- Fix CI: restrict pvlib to <0.8.0 because of API break, reduce test length because of hanging CI (#112)

Contributors

- Thomas Capelle (@tcapelle)
- Kevin Anderson (@kanderso-nrel)
- Marc Anoma (@anomam)

2.6.5 v1.4.1 (November 29, 2019)

Fix

The vectorization of the calculations (from v1.3.0) in the PVEngine had removed the ability to account for timeseries albedo values (it was only using the first albedo value). This fix repairs that issue by building the full 3D matrices for the reflectivity values (and the inverse reflectivity values as well).

- PVEngine needs to use timeseries albedo values (#98)

Contributors

- Marc Anoma (@anomam)

2.6.6 v1.4.0 (November 21, 2019)

Enhancements

pvfactors can now account for AOI losses by either using constant diffuse losses, are by using an fAOI function that will provide the corresponding loss for each value of angle of incidence.

- Test for continuity of results with direct shading (#91)
- Implement non-diffuse AOI loss methods (#92)

- Implement fAOI modifiers for irradiance models (#93)
- Merge new AOI methods into full mode workflow (#94)
- Include fAOI losses from irradiance models in tests (#95)
- Update docs for AOI methods (#96)

Contributors

- Marc Anoma (@anomam)

2.6.7 v1.3.0 (November 6, 2019)

Enhancements

pvfactors is now only using timeseries geometries and vectorization for the view factor matrix calculation, even with the full reflection equilibrium mode. This resulted in an incredible speed boost, in which 8760 simulations now run in less than 2 seconds when using the full mode (it previously took a couple minutes). So there's not much reason anymore to use the "fast" mode, which is less accurate and not that faster anymore. Lots of package clean up and documentation updates in addition to this.

- Create timeseries ground elements (#80)
- Index all timeseries surfaces (#82)
- Vectorize calculation of vf matrix (#83)
- Implement vectorized full mode (#84)
- Clean up package now that full mode is vectorized (#86)
- Reorganize geometry sub-package (#87)
- Add docs section on main concepts (#88)
- Update docs tutorials (#89)

Contributors

- Marc Anoma (@anomam)

2.6.8 v1.2.2 (October 8, 2019)

Enhancements

Passing GHI to the irradiance models when using the fast mode should provide more accuracy.

- Add GHI to run functions inputs (#78)

Fixes

The OrderedPVArray didn't handle it well when the fit function was called multiple times. A fix was implemented for this.

- Fix accumulation of pvrows when fitting multiple times (#77)

Contributors

- Marc Anoma (@anomam)

2.6.9 v1.2.1 (September 13, 2019)

Enhancements

Added module spacing and transparency inputs to irradiance models, and updated README file to make it clearer.

- Add module transparency and spacing to irradiance models (#72)
- Use reStructuredText for README and add TOC (#74)

Fixes

Fix small issue in irradiance models for fast mode: made sure that shaded surfaces are not getting any Perez circumsolar irradiance, except via module spacing and transparency.

- Fix irradiance models for fast mode shaded surfaces (#73)

Contributors

- Marc Anoma (@anomam)

2.6.10 v1.2.0 (September 9, 2019)

Huge speed improvements and enhancements: implementation of a fully vectorized fast mode which now runs 8760 simulations in less than 2 seconds (and calculates same or better results than previous version of fast mode). The improvements done for fast mode also benefit the full simulation mode as some speed improvements have been observed as well.

- Vectorize shading (#64)
- Create timeseries PV row geometries (#65)
- Create timeseries ground (#66)
- Timeseries view factors (#67)
- Update irradiance models (#68)
- Update engine and run functions for timeseries fast mode (#69)
- Update docs for vectorized fast mode (#70)

Contributors

- Marc Anoma (@anomam)

2.6.11 v1.1.0 (August 2, 2019)

Some clean ups and enhancements: the PV Array geometry class `OrderedPVArray` now uses vectorization to calculate the geometry coordinates, which makes the simulations around 30% faster.

- Vectorize geometry calculations (#60)
- Add common project folders to .gitignore (#61)
- Tutorial for fast mode (#62)

Contributors

- Cedric Leroy (@cedricleroy)
- Marc Anoma (@anomam)

2.6.12 v1.0.3 (July 12, 2019)

Enhancement: users can now pass irradiance model arguments to run functions. This was only possible when using the PV engine directly until now.

- Pass irradiance model params to run functions (#57)

Contributors

- Marc Anoma (@anomam)

2.6.13 v1.0.2 (July 5, 2019)

Some bug fixes and enhancements. Now the PVEngine can run simulations using a "fast-mode" with observed speed gain of around 30% and accuracy drop of around 4% compared to the full mode.

- Update python dependencies and test requirements (#50)
- Added a Tolerance for direct shading detection to cast_shadow function (#51)
- Fix broken tests from #51 & check circleci (#52)
- Implement a fast simulation mode in PVEngine (#53)
- Build sphinx docs into CircleCI artifacts (#54)
- Make engine more robust to bad weather data (#55)

Contributors

- Marc Anoma (@anomam)
- Thomas Capelle (@tcapelle)

2.6.14 v1.0.1 (May 14, 2019)

A number of small fixes. And also newer and correct build for this version.

- Fix small negative vf between pvrows (#45)
- Passing calculated view factor matrix to pv array for use in reports (#46)
- Small fixes (#44)
- Fix "difference" calculation method for linestrings (#47)

Contributors

- Marc Anoma (@anomam)

2.6.15 v1.0.0 (April 19, 2019)

Major release for pvfactors. The whole code base was revamped, which led to a 5x speed increase in computational speed. The package API has now also been completely upgraded, with a seperation and uncoupling between geometry, irradiance, and view factor modeling. All of these items are now unified into an engine and also some run functions to run full or partial simulations, and inspect the results. The documentation was completely revamped as well, with a new tutorial section containing lots of examples to get familiar with pvfactors, and also a developer API section that documents all of the classes and functions of the package.

- Fix pvlib version in order to create conda build (#26)
- Update docs: reorganize, clean up, and add API (#27)
- Fix img url and update circleci look (#28)
- New Geometry API (#29)
- API refactoring for view factor calculation (#30)
- New irradiance API (#31)
- Implement perez model with new irradiance API (#33)
- Implement engine to run simulations using new APIs (#32)
- Implement functional run and parallel computation (#37)
- Migrate last elements to new API (#38)
- Remove old API files (#39)
- Update docs for new pvfactors API (#40)
- Update docstrings (#41)

Contributors

- Marc Anoma (@anomam)

2.6.16 v0.1.5 (December 14, 2018)

Updates so that pvfactors is not broken by pvlib-python updates in upcoming version 6.1

- Updates for upcoming pvlib version 6.1 (#24)
- Small fixes to display long description on PyPI, and docs

Contributors

- Marc Anoma

2.6.17 v0.1.4 (November 22, 2018)

Major simplification of simulation input types, and update of docs for PyPI. Now the only PV array angles needed for simulations are 'surface_tilt' and 'surface_azimuth', and they also follow the pvlib-python convention.

- Small updates for PyPI upload (#21)
- Use 'surface_azimuth' and 'surface_tilt' only, with pvlib convention (#22)

Contributors

- Marc Anoma

2.6.18 v0.1.3 (September 13, 2018)

Backwards compatibility fix for timeseries simulation.

- Make sure that all timestamps are returned in outputs (#17)

Contributors

- Marc Anoma

2.6.19 v0.1.2 (September 12, 2018)

Major updates of simulation API and package organization, as well as documentation.

- Refactor tools.py: return 1 output df in timeseries Perez (#13)
- Simplify timeseries calculation API (#14)
- Update docs because of simulation API changes (#15)

Contributors

- Marc Anoma

2.6.20 v0.1.1 (September 6, 2018)

Implementation of important package and model improvements.

- Migration to CircleCI 2.0 (#5)
- Removed dependency on geopandas (#3)
- Implementation of back horizon band shading (#7)
- Clean up and add Github features (#9)
- Output all the surface registries calculated for each timestamp in a Perez timeseries simulation (#10)

Contributors

- Marc Anoma

2.6.21 v0.1.0 (May 14, 2018)

This is the first release of pvfactors. We hope this package will help answer some important questions on irradiance calculation for the PV industry.

- Use shapely and geodataframes to create 2D PV array geometries and record them
- Add ability to discretize ‘’pvrow” surfaces in order to calculate irradiance distributions (eg diffuse shading)
- Use Perez diffuse light model
- Add multiprocessing and improve computational speed
- Create extensive documentation including a Jupyter notebook tutorial
- Implement circumsolar and horizon band shading to improve diffuse shading calculations
- Created tools functions for running timeseries simulations
- Make package compatible with Python3
- Add continuous integration with CircleCI
- Add versioneer for “auto-versioning” of package

Contributors

- Marc Anoma

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

A

B

E

H

I

O

P

R

S

T

V