PVAnalytics

pvlib

Feb 14, 2024

CONTENTS:

PVAnalytics is a python library that supports analytics for PV systems. It provides functions for quality control, filtering, and feature labeling and other tools supporting the analysis of PV system-level data. It can be used as a standalone analysis package and as a data cleaning "front end" for other PV analysis packages.

PVAnalytics is free and open source under a permissive license. The source code for PVAnalytics is hosted on github.

CHAPTER

ONE

LIBRARY OVERVIEW

The functions provided by PVAnalytics are organized in submodules based on their anticipated use. The list below provides a general overview; however, not all modules have functions at this time, see the API reference for current library status.

- `quality` contains submodules for different kinds of data quality checks.
  - `quality.data_shifts` contains quality checks for detecting and isolating data shifts in PV time series data.
  - `quality.irradiance` contains quality checks for irradiance measurements.
  - `quality.weather` contains quality checks for weather data (e.g. tests for physically plausible values of temperature, wind speed, humidity).
  - `quality.outliers` contains functions for identifying outliers.
  - `quality.gaps` contains functions for identifying gaps in the data (i.e. missing values, stuck values, and interpolation).
  - `quality.time` quality checks related to time (e.g. timestamp spacing, time shifts).
  - `quality.util` general purpose quality functions (e.g. simple range checks).
- `features` contains submodules with different methods for identifying and labeling salient features.
  - `features.clipping` functions for labeling inverter clipping.
  - `features.clearsky` functions for identifying periods of clear sky conditions.
  - `features.daytime` functions for identifying periods of day and night.
  - `features.orientation` functions for identifying orientation-related features in the data (e.g. days where the data looks like there is a functioning tracker). These functions are distinct from the functions in the `system` module in that we are identifying features of data rather than properties of the system that produced the data.
  - `features.shading` functions for identifying shadows.
- `system` identification of PV system characteristics from data (e.g. nameplate power, tilt, azimuth)
- `metrics` contains functions for computing PV system-level metrics (e.g. performance ratio)

CHAPTER

TWO

DEPENDENCIES

This project follows the guidelines laid out in NEP-29. It supports:

- All minor versions of Python released 42 months prior to the project, and at minimum the two latest minor versions.
- All minor versions of numpy released in the 24 months prior to the project, and at minimum the last three minor versions
- The latest release of pvlib.

Additionally, PVAnalytics relies on several other packages in the open source scientific python ecosystem. For details on dependencies and versions, see our setup.py.

2.1 API Reference

2.1.1 Quality

Data Shifts

Functions for identifying shifts in data values in time series and for identifying periods with data shifts. For functions that identify shifts in time, see `quality.time`

quality.data_shifts.detect_data_shifts(series)	Detect data shifts in a time series of daily values.
quality.data_shifts.get_longest_shift_segment_dates(series)	Return the start and end dates of the longest serially complete time series segment.

pvanalytics.quality.data_shifts.detect_data_shifts

pvanalytics.quality.data_shifts.**detect_data_shifts**(*series*, *filtering=True*, *use_default_models=True*, *method=None*, *cost=None*, *penalty=40*)

Detect data shifts in a time series of daily values.

> **Warning:** If the passed time series is less than 2 years in length, it will not be corrected for seasonality. Data shift detection will be run on the min-max normalized time series with no seasonality correction.

Parameters

- **series** (*Pandas series with datetime index.*) – Time series of daily PV data values, which can include irradiance and power data.
- **filtering** (*Boolean, default True.*) – Whether or not to filter out outliers and stale data from the time series. If True, then this data is filtered out before running the data shift detection sequence. If False, this data is not filtered out. Default set to True.
- **use_default_models** (*Boolean, default True*) – If True, then default change point detection search parameters are used. For time series shorter than 2 years in length, the search function is *rpt.Window* with *model='rbf'*, *width=50* and *penalty=30*. For time series 2 years or longer in length, the search function is *rpt.BottomUp* with *model='rbf'* and *penalty=40*.
- **method** (*ruptures search method instance or None, default None.*) – Ruptures search method instance. See https://centre-borelli.github.io/ruptures-docs/user-guide/.
- **cost** (*str or None, default None*) – Cost function passed to the ruptures changepoint search instance. See https://centre-borelli.github.io/ruptures-docs/user-guide/
- **penalty** (*int, default 40*) – Penalty value passed to the ruptures changepoint detection method. Default set to 40.

Returns Series of boolean values with the input Series' datetime index, where detected changepoints are labeled as True, and all other values are labeled as False.

Return type Pandas Series

References

Examples using `pvanalytics.quality.data_shifts.detect_data_shifts`

- *PV Fleets QA Process: Temperature*
- *PV Fleets QA Process: Irradiance*
- *PV Fleets QA Process: Power*
- *Data Shift Detection & Filtering*

pvanalytics.quality.data_shifts.get_longest_shift_segment_dates

pvanalytics.quality.data_shifts.**get_longest_shift_segment_dates**(*series*, *filtering=True*, *use_default_models=True*, *method=None*, *cost=None*, *penalty=40*, *buffer_day_length=7*)

Return the start and end dates of the longest serially complete time series segment.

During this process, data shift detection is performed, and the longest time series segment between changepoints is identified, and the start and end dates of that segment are returned, with a settable buffer period added to the start date and subtracted from the end date, to allow for the segment to stabilize (this helps if the changepoint is detected a few days early or a few days late, compared to the actual shift date).

Parameters

- **series** (*Pandas series with datetime index.*) – Daily time series of a PV data stream, which can include irradiance and power data streams. This series represents the summed daily values of the particular data stream.

- **filtering** (*Boolean, default True.*) – Whether or not to filter out outliers and stale data from the time series. If True, then this data is filtered out before running the data shift detection sequence. If False, this data is not filtered out. Default set to True.
- **use_default_models** (*Boolean, default True*) – If True, then default change point detection search parameters are used. For time series shorter than 2 years in length, the search function is *rpt.Window* with *model='rbf'*, *width=50* and *penalty=30*. For time series 2 years or longer in length, the search function is *rpt.BottomUp* with *model='rbf'* and *penalty=40*.
- **method** (*ruptures search method instance or None, default None.*) – Ruptures search method instance. See https://centre-borelli.github.io/ruptures-docs/user-guide/.
- **cost** (*str or None, default None*) – Cost function passed to the ruptures changepoint search instance. See https://centre-borelli.github.io/ruptures-docs/user-guide/
- **penalty** (*int, default 40*) – Penalty value passed to the ruptures changepoint detection method. Default set to 40.
- **buffer_day_length** (*int, default 7*) – Number of days to add to the start date and subtract from the end date of the longest detected data shift-free period. This buffer period helps to filter out any data that doesn't fit within the current data segment. This issue occurs when the changepoint is detected a few days early or late compared to the actual data shift date.

Returns

- **start_date** (*Pandas datetime*) – Start date of the longest continuous time series segment that is free of data shifts.
- **end_date** (*Pandas datetime*) – End date of the longest continuous time series segment that is free of data shifts.

References

Examples using `pvanalytics.quality.data_shifts.get_longest_shift_segment_dates`

- *PV Fleets QA Process: Temperature*
- *PV Fleets QA Process: Irradiance*
- *PV Fleets QA Process: Power*
- *Data Shift Detection & Filtering*

Irradiance

The `check_*_limits_qcrad` functions use the QCRad algorithm[1] to identify irradiance measurements that are beyond physical limits.

quality.irradiance.check_ghi_limits_qcrad(...)	Test for physical limits on GHI using the QCRad criteria.
quality.irradiance.check_dhi_limits_qcrad(...)	Test for physical limits on DHI using the QCRad criteria.
quality.irradiance.check_dni_limits_qcrad(...)	Test for physical limits on DNI using the QCRad criteria.

[1] C. N. Long and Y. Shi, An Automated Quality Assessment and Control Algorithm for Surface Radiation Measurements, The Open Atmospheric Science Journal 2, pp. 23-37, 2008.

pvanalytics.quality.irradiance.check_ghi_limits_qcrad

pvanalytics.quality.irradiance.**check_ghi_limits_qcrad**(*ghi*, *solar_zenith*, *dni_extra*, *limits=None*)

Test for physical limits on GHI using the QCRad criteria.

Test is applied to each GHI value. A GHI value passes if value > lower bound and value < upper bound. Lower bounds are constant for all tests. Upper bounds are calculated as

$$ub = min + mult * dni\_extra * cos(solar\_zenith)^{exp}$$

Parameters

- **ghi** (*Series*) – Global horizontal irradiance in W/m^2
- **solar_zenith** (*Series*) – Solar zenith angle in degrees
- **dni_extra** (*Series*) – Extraterrestrial normal irradiance in W/m^2
- **limits** (*dict, default QCRAD_LIMITS*) – Must have keys 'ghi_ub' and 'ghi_lb'. For 'ghi_ub' value is a dict with keys {'mult', 'exp', 'min'} and float values. For 'ghi_lb' value is a float.

Returns True where value passes limits test.

Return type Series

Notes

pvanalytics.quality.irradiance.check_dhi_limits_qcrad

pvanalytics.quality.irradiance.**check_dhi_limits_qcrad**(*dhi*, *solar_zenith*, *dni_extra*, *limits=None*)

Test for physical limits on DHI using the QCRad criteria.

Test is applied to each DHI value. A DHI value passes if value > lower bound and value < upper bound. Lower bounds are constant for all tests. Upper bounds are calculated as

$$ub = min + mult * dni\_extra * cos(solar\_zenith)^{exp}$$

Parameters

- **dhi** (*Series*) – Diffuse horizontal irradiance in W/m^2
- **solar_zenith** (*Series*) – Solar zenith angle in degrees
- **dni_extra** (*Series*) – Extraterrestrial normal irradiance in W/m^2
- **limits** (*dict, default QCRAD_LIMITS*) – Must have keys 'dhi_ub' and 'dhi_lb'. For 'dhi_ub' value is a dict with keys {'mult', 'exp', 'min'} and float values. For 'dhi_lb' value is a float.

Returns True where value passes limit test.

Return type Series

Notes

pvanalytics.quality.irradiance.check_dni_limits_qcrad

pvanalytics.quality.irradiance.**check_dni_limits_qcrad**(*dni*, *solar_zenith*, *dni_extra*, *limits=None*)

Test for physical limits on DNI using the QCRad criteria.

Test is applied to each DNI value. A DNI value passes if value > lower bound and value < upper bound. Lower bounds are constant for all tests. Upper bounds are calculated as

$$ub = min + mult * dni\_extra * cos(solar\_zenith)^{exp}$$

Parameters

- **dni** (*Series*) – Direct normal irradiance in W/m^2
- **solar_zenith** (*Series*) – Solar zenith angle in degrees
- **dni_extra** (*Series*) – Extraterrestrial normal irradiance in W/m^2
- **limits** (*dict, default QCRAD_LIMITS*) – Must have keys 'dni_ub' and 'dni_lb'. For 'dni_ub' value is a dict with keys {'mult', 'exp', 'min'} and float values. For 'dni_lb' value is a float.

Returns True where value passes limit test.

Return type Series

Notes

All three checks can be combined into a single function call.

quality.irradiance.check_irradiance_limits_qcrad(...)	Test for physical limits on GHI, DHI or DNI using the QCRad criteria.

pvanalytics.quality.irradiance.check_irradiance_limits_qcrad

pvanalytics.quality.irradiance.**check_irradiance_limits_qcrad**(*solar_zenith*, *dni_extra*, *ghi=None*, *dhi=None*, *dni=None*, *limits=None*)

Test for physical limits on GHI, DHI or DNI using the QCRad criteria.

Criteria from[1] are used to determine physically plausible lower and upper bounds. Each value is tested and a value passes if value > lower bound and value < upper bound. Lower bounds are constant for all tests. Upper bounds are calculated as

$$ub = min + mult * dni\_extra * cos(solar\_zenith)^{exp}$$

[1] C. N. Long and Y. Shi, An Automated Quality Assessment and Control Algorithm for Surface Radiation Measurements, The Open Atmospheric Science Journal 2, pp. 23-37, 2008.

Note: If any of *ghi*, *dhi*, or *dni* are None, the corresponding element of the returned tuple will also be None.

Parameters

- **solar_zenith** (*Series*) – Solar zenith angle in degrees
- **dni_extra** (*Series*) – Extraterrestrial normal irradiance in W/m^2
- **ghi** (*Series or None, default None*) – Global horizontal irradiance in W/m^2
- **dhi** (*Series or None, default None*) – Diffuse horizontal irradiance in W/m^2
- **dni** (*Series or None, default None*) – Direct normal irradiance in W/m^2
- **limits** (*dict, default QCRAD_LIMITS*) – for keys 'ghi_ub', 'dhi_ub', 'dni_ub', value is a dict with keys {'mult', 'exp', 'min'} and float values. For keys 'ghi_lb', 'dhi_lb', 'dni_lb', value is a float.

Returns

- **ghi_limit_flag** (*Series*) – True for each value that is physically possible. None if *ghi* is None.
- **dhi_limit_flag** (*Series*) – True for each value that is physically possible. None if *dni* is None.
- **dni_limit_flag** (*Series*) – True for each value that is physically possible. None if *dni* is None.

Notes

Copyright (c) 2019 SolarArbiter. See the file LICENSES/SOLARFORECASTARBITER_LICENSE at the top level directory of this distribution and at https://github.com/pvlib/pvanalytics/blob/master/LICENSES/SOLARFORECASTARBITER_LICENSE for more information.

References

Examples using pvanalytics.quality.irradiance.check_irradiance_limits_qcrad

- *QCrad Limits for Irradiance Data*

Irradiance measurements can also be checked for consistency.

quality.irradiance.check_irradiance_consistency_qcrad(...)	Check consistency of GHI, DHI and DNI using QCRad criteria.

pvanalytics.quality.irradiance.check_irradiance_consistency_qcrad

pvanalytics.quality.irradiance.**check_irradiance_consistency_qcrad**(*solar_zenith*, *ghi*, *dhi*, *dni*, *param=None*)

Check consistency of GHI, DHI and DNI using QCRad criteria.

Uses criteria given in[1] to validate the ratio of irradiance components.

[1] C. N. Long and Y. Shi, An Automated Quality Assessment and Control Algorithm for Surface Radiation Measurements, The Open Atmospheric Science Journal 2, pp. 23-37, 2008.

Warning: Not valid for night time. While you can pass data from night time to this function, be aware that the truth values returned for that data will not be valid.

Parameters

- **solar_zenith** (*Series*) – Solar zenith angle in degrees
- **ghi** (*Series*) – Global horizontal irradiance in W/m^2
- **dhi** (*Series*) – Diffuse horizontal irradiance in W/m^2
- **dni** (*Series*) – Direct normal irradiance in W/m^2
- **param** (*dict*) – keys are 'ghi_ratio' and 'dhi_ratio'. For each key, value is a dict with keys 'high_zenith' and 'low_zenith'; for each of these keys, value is a dict with keys 'zenith_bounds', 'ghi_bounds', and 'ratio_bounds' and value is an ordered pair [lower, upper] of float.

Returns

- **consistent_components** (*Series*) – True where *ghi*, *dhi* and *dni* components are consistent.
- **diffuse_ratio_limit** (*Series*) – True where diffuse to GHI ratio passes limit test.

Notes

References

Examples using pvanalytics.quality.irradiance.check_irradiance_consistency_qcrad

- *QCrad Consistency for Irradiance Data*

GHI and POA irradiance can be validated against clearsky values to eliminate data that is unrealistically high.

quality.irradiance.clearsky_limits(measured, ...)	Identify irradiance values which do not exceed clearsky values.

pvanalytics.quality.irradiance.clearsky_limits

pvanalytics.quality.irradiance.**clearsky_limits**(*measured*, *clearsky*, *csi_max=1.1*)

Identify irradiance values which do not exceed clearsky values.

Uses pvlib.irradiance.clearsky_index() to compute the clearsky index as the ratio of *measured* to *clearsky*. Compares the clearsky index to *csi_max* to identify values in *measured* that are less than or equal to *csi_max*.

Parameters

- **measured** (*Series*) – Measured irradiance in W/m^2.
- **clearsky** (*Series*) – Expected clearsky irradiance in W/m^2.

- **csi_max** (*float, default 1.1*) – Maximum ratio of *measured* to *clearsky* (clearsky index).

Returns True for each value where the clearsky index is less than or equal to *csi_max*.

Return type Series

Notes

Examples using `pvanalytics.quality.irradiance.clearsky_limits`

- *Clearsky Limits for Irradiance Data*

You may want to identify entire days that have unrealistically high or low insolation. The following function examines daily insolation, validating that it is within a reasonable range of the expected clearsky insolation for the same day.

quality.irradiance.daily_insolation_limits(...)	Check that daily insolation lies between minimum and maximum values.

pvanalytics.quality.irradiance.daily_insolation_limits

pvanalytics.quality.irradiance.**daily_insolation_limits**(*irrad*, *clearsky*, *daily_min=0.4*, *daily_max=1.25*)

Check that daily insolation lies between minimum and maximum values.

Irradiance measurements and clear-sky irradiance on each day are integrated with the trapezoid rule to calculate daily insolation.

Parameters

- **irrad** (*Series*) – Irradiance measurements (GHI or POA).
- **clearsky** (*Series*) – Clearsky irradiance.
- **daily_min** (*float, default 0.4*) – Minimum ratio of daily insolation to daily clearsky insolation.
- **daily_max** (*float, default 1.25*) – Maximum ratio of daily insolation to daily clearsky insolation.

Returns True for values on days where the ratio of daily insolation to daily clearsky insolation is between *daily_min* and *daily_max*.

Return type Series

Notes

The default limits (*daily_max* and *daily_min*) have been set for GHI and POA irradiance for systems with *fixed* azimuth and tilt. If you pass POA irradiance for a tracking system it is recommended that you increase *daily_max* to 1.35.

The default values for *daily_min* and *daily_max* were taken from the PVFleets QA Analysis project.

Examples using `pvanalytics.quality.irradiance.daily_insolation_limits`

- *Clearsky Limits for Daily Insolation*

There is function for calculating the component sum for GHI, DHI, and DNI, and correcting for nighttime periods. Using this function, we can estimate one irradiance field using the two other irradiance fields. This can be useful for comparison, as well as to calculate missing data fields.

quality.irradiance.calculate_component_sum_series(...)	Use the component sum equations to calculate the missing series, using the other available time series.

pvanalytics.quality.irradiance.calculate_component_sum_series

pvanalytics.quality.irradiance.**calculate_component_sum_series**(*solar_zenith*, *ghi=None*, *dhi=None*, *dni=None*, *dni_clear=None*, *zenith_limit=90*, *fill_night_value=None*)

Use the component sum equations to calculate the missing series, using the other available time series. One of the three parameters (ghi, dhi, dni) is passed as None, and the two series are used to calculate the missing series. After calculation, the series is run through a nighttime routine, where nighttime values are set based on the fill_night_value parameter.

The "component sum" or "closure" equation relates the three primary irradiance components as follows:

$$GHI = DHI + DNI * \cos(\theta_z)$$

Parameters

- **solar_zenith** (*Series*) – Zenith angles in decimal degrees, with datetime index. Angles must be >=0 and <=180. Must have the same datetime index as *dni*, *dhi*, and *dni*, when available.
- **ghi** (*Series, optional*) – Pandas series of GHI data, with datetime index. Must have the same datetime index as *dni*, *dhi*, and *solar_zenith*, when available.
- **dhi** (*Series, optional*) – Pandas series of DNI data, with datetime index. Must have the same datetime index as ghi, dni, and zenith series, when available.
- **dni** (*Series, optional*) – Pandas series of dni data, with datetime index. Must have the same datetime index as *ghi*, *dhi*, and *solar_zenith*, when available.
- **dni_clear** (*Series, optional*) – Pandas series of clearsky dni data. Must have the same datetime index as *ghi*, *dhi*, *dni*, and *solar_zenith*, when available. See pvlib-python's `pvlib.irradiance.dni()` for details.
- **zenith_limit** (*Float*) – Solar zenith boundary between night and day, in degrees. For calculation of the component sum, *solar_zenith* is set to 90 where *solar_zenith* > *zenith_limit*.

- **fill_night_value** (*String or float or int, default None*) – Options include 'equation', float or int values (np.nan, 0, etc.), or None. This is the fill value for nighttime periods. If a float or int value is passed (np.nan, 0 , -.5, etc.), then nighttime values are filled using the fill_night_value parameter. If 'equation' is used, nighttime periods are filled using the component sum equation with DNI=0: `GHI = 0 + DHI` If None, then the nighttime values are based on the component sum equation.

Returns Pandas series of the calculated values, based on the component sum equation and corrected for nighttime periods.

Return type Series

Examples using `pvanalytics.quality.irradiance.calculate_component_sum_series`

- *Component Sum Equations for Irradiance Data*

Gaps

Identify gaps in the data.

quality.gaps.interpolation_diff(x[, window, ...])	Identify sequences which appear to be linear.

pvanalytics.quality.gaps.interpolation_diff

pvanalytics.quality.gaps.**interpolation_diff**(*x*, *window=6*, *rtol=1e-05*, *atol=1e-08*, *mark='tail'*)

Identify sequences which appear to be linear.

Sequences are linear if the first difference appears to be constant. For a window of length N, the last value (index N-1) is flagged if all values in the window appear to be a line segment.

Parameters *rtol* and *atol* have the same meaning as in `numpy.allclose()`.

Parameters

- **x** (*Series*) – data to be processed
- **window** (*int, default 6*) – number of sequential values that, if the first difference is constant, are classified as a linear sequence
- **rtol** (*float, default 1e-5*) – tolerance relative to max(abs(x.diff()) for detecting a change
- **atol** (*float, default 1e-8*) – absolute tolerance for detecting a change in first difference
- **mark** (*str, default 'tail'*) – How much of the window to mark `True` when a sequence of interpolated values is detected. Can be one of 'tail', 'end', or 'all'.
  - If 'tail' (the default) then every point in the window *except* the first point is marked `True`.
  - If 'end' then the first *window - 1* values in an interpolated sequence are marked `False` and all subsequent values in the sequence are marked `True`.
  - If 'all' then every point in the window *including* the first point is marked `True`.

Returns True for each value that is part of a linear sequence

Return type Series

Raises `ValueError` – If *window* < *3* or *mark* is not one of 'tail', 'end', or 'all'.

Notes

Examples using `pvanalytics.quality.gaps.interpolation_diff`

- *Interpolated Data Periods*

Data sometimes contains sequences of values that are "stale" or "stuck." These are contiguous spans of data where the value does not change within the precision given. The functions below can be used to detect stale values.

Note: If the data has been altered in some way (i.e. temperature that has been rounded to an integer value) before being passed to these functions you may see unexpectedly large amounts of stale data.

quality.gaps.stale_values_diff(x[, window, ...])	Identify stale values in the data.
quality.gaps.stale_values_round(x[, window, ...])	Identify stale values by rounding.

pvanalytics.quality.gaps.stale_values_diff

pvanalytics.quality.gaps.**stale_values_diff**(*x*, *window=6*, *rtol=1e-05*, *atol=1e-08*, *mark='tail'*)

Identify stale values in the data.

For a window of length N, the last value (index N-1) is considered stale if all values in the window are close to the first value (index 0).

Parameters *rtol* and *atol* have the same meaning as in `numpy.allclose()`.

Parameters

- **x** (*Series*) – data to be processed
- **window** (*int, default 6*) – number of consecutive values which, if unchanged, indicates stale data
- **rtol** (*float, default 1e-5*) – relative tolerance for detecting a change in data values
- **atol** (*float, default 1e-8*) – absolute tolerance for detecting a change in data values
- **mark** (*str, default 'tail'*) – How much of the window to mark `True` when a sequence of stale values is detected. Can one be of 'tail', 'end', or 'all'.
  - If 'tail' (the default) then every point in the window *except* the first point is marked `True`.
  - If 'end' then the first *window - 1* values in a stale sequence sequence are marked `False` and all subsequent values in the sequence are marked `True`.
  - If 'all' then every point in the window *including* the first point is marked `True`.

Returns True for each value that is part of a stale sequence of data

Return type Series

Raises `ValueError` – If *window* < *2* or *mark* is not one of 'tail', 'end', or 'all'.

Notes

Examples using `pvanalytics.quality.gaps.stale_values_diff`

- *Stale Data Periods*

pvanalytics.quality.gaps.stale_values_round

`pvanalytics.quality.gaps.`**`stale_values_round`**(*x*, *window=6*, *decimals=3*, *mark='tail'*)

Identify stale values by rounding.

A value is considered stale if it is part of a sequence of length *window* of values that are identical when rounded to *decimals* decimal places.

Parameters

- **`x`** (*`Series`*) – Data to be processed.
- **`window`** (*`int, default 6`*) – Number of consecutive identical values for a data point to be considered stale.
- **`decimals`** (*`int, default 3`*) – Number of decimal places to round to.
- **`mark`** (*`str, default 'tail'`*) – How much of the window to mark `True` when a sequence of stale values is detected. Can be one of 'tail', 'end', or 'all'.
  - If 'tail' (the default) then every point in the window *except* the first point is marked `True`.
  - If 'end' then the first *window - 1* values in a stale sequence sequence are marked `False` and all subsequent values in the sequence are marked `True`.
  - If 'all' then every point in the window *including* the first point is marked `True`.

Returns True for each value that is part of a stale sequence of data.

Return type Series

Raises `ValueError` – If *mark* is not one of 'tail', 'end', or 'all'.

Notes

Examples using pvanalytics.quality.gaps.stale_values_round

- *Stale Data Periods*
- *PV Fleets QA Process: Temperature*
- *PV Fleets QA Process: Irradiance*
- *PV Fleets QA Process: Power*

The following functions identify days with incomplete data.

quality.gaps.completeness_score(series[, ...])	Calculate a data completeness score for each day.
quality.gaps.complete(series[, ...])	Select data points that are part of days with complete data.

pvanalytics.quality.gaps.completeness_score

pvanalytics.quality.gaps.**completeness_score**(*series*, *freq=None*, *keep_index=True*)

Calculate a data completeness score for each day.

The completeness score for a given day is the fraction of time in the day for which there is data (a value other than NaN). The time duration attributed to each value is equal to the timestamp spacing of *series*, or *freq* if it is specified. For example, a 24-hour time series with 30 minute timestamp spacing and 24 non-NaN values would have data for a total of 12 hours and therefore a completeness score of 0.5.

Parameters

- **series** (*Series*) – A DatetimeIndexed series.
- **freq** (*str, default None*) – Interval between samples in the series as a pandas frequency string. If None, the frequency is inferred using pandas.infer_freq().
- **keep_index** (*boolean, default True*) – Whether or not the returned series has the same index as *series*. If False the returned series will be indexed by day.

Returns A series of floats giving the completeness score for each day (fraction of the day for which *series* has data).

Return type Series

Raises ValueError – If *freq* is longer than the frequency inferred from *series*.

Examples using pvanalytics.quality.gaps.completeness_score

- *Missing Data Periods*
- *PV Fleets QA Process: Temperature*
- *PV Fleets QA Process: Irradiance*
- *PV Fleets QA Process: Power*

pvanalytics.quality.gaps.complete

pvanalytics.quality.gaps.**complete**(*series*, *minimum_completeness=0.333*, *freq=None*)

Select data points that are part of days with complete data.

A day has complete data if its completeness score is greater than or equal to *minimum_completeness*. The completeness score is calculated by *completeness_score()*.

Parameters

- **series** (*Series*) – The data to be checked for completeness.
- **minimum_completeness** (*float, default 0.333*) – Fraction of the day that must have data.
- **freq** (*str, default None*) – The expected frequency of the data in *series*. If none then the frequency is inferred from the data.

Returns A series of booleans with True for each value that is part of a day with completeness greater than *minimum_completeness*.

Return type Series

Raises **ValueError** – See *completeness_score()*.

See also:

completeness_score

Examples using pvanalytics.quality.gaps.complete

- *Missing Data Periods*

Many data sets may have leading and trailing periods of days with sporadic or no data. The following functions can be used to remove those periods.

quality.gaps.start_stop_dates(series[, days])	Get the start and end of data excluding leading and trailing gaps.
quality.gaps.trim(series[, days])	Mask the beginning and end of the data if not all True.
quality.gaps.trim_incomplete(series[, ...])	Trim the series based on the completeness score.

pvanalytics.quality.gaps.start_stop_dates

pvanalytics.quality.gaps.**start_stop_dates**(*series*, *days=10*)

Get the start and end of data excluding leading and trailing gaps.

Parameters

- **series** (*Series*) – A DatetimeIndexed series of booleans.
- **days** (*int, default 10*) – The minimum number of consecutive days where every value in *series* is True for data to start or stop.

Returns

- **start** (*Datetime or None*) – The first valid day. If there are no sufficiently long periods of valid days then None is returned.
- **stop** (*Datetime or None*) – The last valid day. None if start is None.

Examples using `pvanalytics.quality.gaps.start_stop_dates`

- *PV Fleets QA Process: Temperature*
- *PV Fleets QA Process: Irradiance*
- *PV Fleets QA Process: Power*

pvanalytics.quality.gaps.trim

pvanalytics.quality.gaps.**trim**(*series*, *days=10*)

Mask the beginning and end of the data if not all True.

Parameters

- **series** (*Series*) – A DatetimeIndexed series of booleans
- **days** (*int, default 10*) – Minimum number of consecutive days that are all True for 'good' data to start.

Returns A series of booleans with True for all data points between the first and last block of *days* consecutive days that are all True in *series*. If *series* does not contain such a block of consecutive True values, then the returned series will be entirely False.

Return type Series

See also:

start_stop_dates

pvanalytics.quality.gaps.trim_incomplete

pvanalytics.quality.gaps.**trim_incomplete**(*series*, *minimum_completeness=0.333333*, *days=10*, *freq=None*)

Trim the series based on the completeness score.

Combines *completeness_score()* and *trim()*.

Parameters

- **series** (*Series*) – A DatetimeIndexed series.
- **minimum_completeness** (*float, default 0.333333*) – The minimum completeness score for each day.
- **days** (*int, default 10*) – The number of consecutive days with completeness greater than *minumum_completeness* for the 'good' data to start or end. See *start_stop_dates()* for more information.
- **freq** (*str, default None*) – The expected frequency of the series. See *completeness_score()* fore more information.

Returns A series of booleans with the same index as *series* with False up to the first complete day, True between the first and the last complete days, and False following the last complete day.

Return type Series

See also:

trim, *completeness_score*

Examples using `pvanalytics.quality.gaps.trim_incomplete`

- *Missing Data Periods*
- *PV Fleets QA Process: Temperature*
- *PV Fleets QA Process: Irradiance*
- *PV Fleets QA Process: Power*

Outliers

Functions for detecting outliers.

`quality.outliers.tukey`(data[, k])	Identify outliers based on the interquartile range.
`quality.outliers.zscore`(data[, zmax, nan_policy])	Identify outliers using the z-score.
`quality.outliers.hampel`(data[, window, ...])	Identify outliers by the Hampel identifier.

pvanalytics.quality.outliers.tukey

pvanalytics.quality.outliers.**tukey**(*data*, *k=1.5*)

Identify outliers based on the interquartile range.

A value x is considered an outlier if it does *not* satisfy the following condition

$$Q_1 - k(Q_3 - Q_1) \leq x \leq Q_3 + k(Q_3 - Q_1)$$

where Q_1 is the value of the first quartile and Q_3 is the value of the third quartile.

Parameters

- **data** (*Series*) – The data in which to find outliers.
- **k** (*float, default 1.5*) – Multiplier of the interquartile range. A larger value will be more permissive of values that are far from the median.

Returns A series of booleans with True for each value that is an outlier.

Return type Series

Examples using `pvanalytics.quality.outliers.tukey`

- *Tukey Outlier Detection*

pvanalytics.quality.outliers.zscore

pvanalytics.quality.outliers.**zscore**(*data*, *zmax=1.5*, *nan_policy='raise'*)

Identify outliers using the z-score.

Points with z-score greater than *zmax* are considered as outliers.

Parameters

- **data** (*Series*) – A series of numeric values in which to find outliers.

- **zmax** (*float*) – Upper limit of the absolute values of the z-score.
- **nan_policy** (*{'raise', 'omit'}, default 'raise'*) – Define how to handle NaNs in the input series. If 'raise', a ValueError is raised when *data* contains NaNs. If 'omit', NaNs are ignored and False is returned at indices that contained NaN in *data*.

Returns A series of booleans with True for each value that is an outlier.

Return type Series

Examples using `pvanalytics.quality.outliers.zscore`

- *Z-Score Outlier Detection*
- *PV Fleets QA Process: Temperature*
- *PV Fleets QA Process: Irradiance*
- *PV Fleets QA Process: Power*

pvanalytics.quality.outliers.hampel

pvanalytics.quality.outliers.**hampel**(*data*, *window=5*, *max_deviation=3.0*, *scale=None*)

Identify outliers by the Hampel identifier.

The Hampel identifier is computed according to[1].

Parameters

- **data** (*Series*) – The data in which to find outliers.
- **window** (*int or offset, default 5*) – The size of the rolling window used to compute the Hampel identifier.
- **max_deviation** (*float, default 3.0*) – Any value with a Hampel identifier > *max_deviation* standard deviations from the median is considered an outlier.
- **scale** (*float, optional*) – Scale factor used to estimate the standard deviation as $MAD/scale$. If *scale=None* (default), then the scale factor is taken to be `scipy.stats.norm.ppf(3/4.)` (approx. 0.6745), and $MAD/scale$ approximates the standard deviation of Gaussian distributed data.

Returns True for each value that is an outlier according to its Hampel identifier.

Return type Series

References

Examples using `pvanalytics.quality.outliers.hampel`

- *Hampel Outlier Detection*

[1] Pearson, R.K., Neuvo, Y., Astola, J. et al. Generalized Hampel Filters. EURASIP J. Adv. Signal Process. 2016, 87 (2016). https://doi.org/10.1186/s13634-016-0383-6

Time

Quality control related to time. This includes things like time-stamp spacing, time-shifts, and time zone validation.

quality.time.spacing(times, freq)	Check that the spacing between *times* conforms to *freq*.

pvanalytics.quality.time.spacing

pvanalytics.quality.time.**spacing**(*times*, *freq*)

Check that the spacing between *times* conforms to *freq*.

Parameters

- **times** (*DatetimeIndex*) –
- **freq** (*string or Timedelta*) – Expected frequency of *times*.

Returns True when the difference between one time and the time before it conforms to *freq*.

Return type Series

Notes

Copyright (c) 2019 SolarArbiter. See the file LICENSES/SOLARFORECASTARBITER_LICENSE at the top level directory of this distribution and at https://github.com/pvlib/pvanalytics/blob/master/LICENSES/SOLARFORECASTARBITER_LICENSE for more information.

Timestamp shifts, such as daylight savings, can be identified with the following functions.

quality.time.shifts_ruptures(event_times, ...)	Identify time shifts using the ruptures library.
quality.time.has_dst(events, tz[, window, ...])	Return True if *events* appears to have daylight savings shifts at the dates on which *tz* transitions to or from daylight savings time.

pvanalytics.quality.time.shifts_ruptures

pvanalytics.quality.time.**shifts_ruptures**(*event_times*, *reference_times*, *period_min=15*, *shift_min=15*, *prediction_penalty=20*, *zscore_cutoff=2*, *bottom_quantile_threshold=0*, *top_quantile_threshold=0.5*)

Identify time shifts using the ruptures library.

Compares the event time in the expected time zone (*reference_times*) with the actual event time in *event_times*.

The Binary Segmentation changepoint detection method is applied to the difference between *event_times* and *reference_times*. For each period between change points the mode of the difference is rounded to a multiple of *shift_min* and returned as the time-shift for all days in that period.

Parameters

- **event_times** (*Series*) – Time of an event in minutes since midnight. Should be a time series of integers with a single value per day. Typically the time mid-way between sunrise and sunset.

- **reference_times** (*Series*) – Time of event in minutes since midnight for each day in the expected timezone. For example, passing solar transit time in a fixed offset time zone can be used to detect daylight savings shifts when it is unknown whether or not *event_times* is in a fixed offset time zone.
- **period_min** (*int, default 15*) – Minimum number of days between shifts. Must be less than or equal to the number of days in *event_times*. [days] Increasing this parameter will make the result less sensitive to transient shifts. For example if your intent is to find and correct daylight savings time shifts passing *period_min=60* can give good results while excluding shorter periods that appear shifted.
- **shift_min** (*int, default 15*) – Minimum shift amount in minutes. All shifts are rounded to a multiple of *shift_min*. [minutes]
- **prediction_penalty** (*int, default 13*) – Penalty used in assessing change points. See `ruptures.detection.Binseg.predict()` for more information.
- **zscore_cutoff** (*int, default 2*) – Z-score cutoff / maximum for filtering out outliers in each identified segment found via changepount detection
- **bottom_quantile_threshold** (*float, default 0*) – Bottom quantile threshold for each time series segment identified via changepoint detection. All data below this threshold is not considered when determining the mean value for the segment, which is later rounded to the nearest *period_min* value
- **top_quantile_threshold** (*float, default 0.5*) – Top quantile threshold for each time series segment identified via changepoint detection. All data above this threshold is not considered when determining the mean value for the segment, which is later rounded to the nearest *period_min* value

Returns

- **shifted** (*Series*) – Boolean series indicating whether there appears to be a time shift on that day.
- **shift_amount** (*Series*) – Time shift in minutes for each day in *event_times*. These times can be used to shift the data into the same time zone as *reference_times*.

Raises `ValueError` – If the number of days in *event_times* is less than *period_min*.

Notes

Timestamped data from monitored PV systems may not always be localized to a consistent timezone. In some cases, data is timestamped with local time that may or may not be adjusted for daylight savings time transitions. This function helps detect issues of this sort, by detecting points where the time of some daily event (e.g. solar noon) changes significantly with respect to a reference time for the event. If the data's timestamps have not been adjusted for daylight savings transitions, the time of day at solar noon will change by roughly 60 minutes in the days before and after the transition.

To use this changepoint detection method to determine if your data's timestamps involve daylight savings transitions, first reduce your PV system data (irradiance or power) to a daily time series, with each point being the observed midday time in minutes. For example, if sunrise and sunset are inferred from the PV system data, the midday time can be inferred as the average of each day's sunrise and sunset time of day. To establish the expected midday time, calculate solar transit time in time of day.

Derived from the PVFleets QA project.

References

Examples using pvanalytics.quality.time.shifts_ruptures

- *PV Fleets QA Process: Irradiance*
- *PV Fleets QA Process: Power*
- *Identifying and estimating time shifts*

pvanalytics.quality.time.has_dst

pvanalytics.quality.time.**has_dst**(*events*, *tz*, *window=7*, *min_difference=45*, *missing='raise'*)

Return True if *events* appears to have daylight savings shifts at the dates on which *tz* transitions to or from daylight savings time.

The mean event time in minutes since midnight is calculated over the *window* days before and after the date of each daylight savings transition in *tz*. For each date, the two mean event times (before and after) are compared, and if the difference is greater than *min_difference* then a shift has occurred on that date.

Parameters

- **events** (*Series*) – Series with one timestamp for each day. The timestamp should correspond to an event that occurs at roughly the same time on each day. For example, you may pass sunrise, sunset, or solar transit time. *events* need not be localized.
- **tz** (*str*) – Name of a timezone that observes daylight savings and has the same or similar UTC offset as the expected time zone for *events*.
- **window** (*int, default 7*) – Number of days before and after the shift date to consider. When passing rounded timestamps in *events* it may be necessary to use a smaller window. [days]
- **min_difference** (*int, default 45*) – Minimum difference between the mean event time before the shift date and the mean event time after the event time. If the difference is greater than *min_difference* a shift has occurred on that date. [minutes]
- **missing** (*{'raise', 'warn'}, default 'raise'*) – Whether to raise an exception or issue a warning when there is no data at a transition date. Can be 'raise' or 'warn'. If 'warn' and there is no data adjacent to a transition date, False is returned for that date.

Returns Boolean Series with the same index as *events* True for dates that appear to have daylight savings transitions.

Return type Series

Raises ValueError – If there is no data in the *window* days before or after a shift date in *events*.

Utilities

The `quality.util` module contains general-purpose/utility functions for building your own quality checks.

quality.util.check_limits(val[, ...])	Check whether a value falls withing the given limits.
quality.util.daily_min(series, minimum[, ...])	Return True for data on days when the day's minimum exceeds *minimum*.

pvanalytics.quality.util.check_limits

pvanalytics.quality.util.**check_limits**(*val*, *lower_bound=None*, *upper_bound=None*, *inclusive_lower=False*, *inclusive_upper=False*)

Check whether a value falls withing the given limits.

At least one of *lower_bound* or *upper_bound* must be provided.

Parameters

- **val** (*array_like*) – Values to test.
- **lower_bound** (*float, default None*) – Lower limit.
- **upper_bound** (*float, default None*) – Upper limit.
- **inclusive_lower** (*bool, default False*) – Whether the lower bound is inclusive (*val* >= *lower_bound*).
- **inclusive_upper** (*bool, default False*) – Whether the upper bound is inclusive (*val* <= *upper_bound*).

Returns True for every value in *val* that is between *lower_bound* and *upper_bound*.

Return type array_like

Raises **ValueError** – if *lower_bound* nor *upper_bound* is provided.

Notes

pvanalytics.quality.util.daily_min

pvanalytics.quality.util.**daily_min**(*series*, *minimum*, *inclusive=False*)

Return True for data on days when the day's minimum exceeds *minimum*.

Parameters

- **series** (*Series*) – A Datetimeindexed series of floats.
- **minimum** (*float*) – The smallest acceptable value for the daily minimum.
- **inclusive** (*boolean, default False*) – Use greater than or equal to when comparing daily minimums from *series* to *minimum*.

Returns True for values on days where the minimum value recorded on that day is greater than (or equal to) *minimum*.

Return type Series

Notes

This function is derived from code in the pvfleets_qa_analysis project under the terms of the 3-clause BSD license. Copyright (c) 2020 Alliance for Sustainable Energy, LLC.

Weather

Quality checks for weather data.

quality.weather.relative_humidity_limits(...)	Identify relative humidity values that are within limits.
quality.weather.temperature_limits(...[, limits])	Identify temperature values that are within limits.
quality.weather.wind_limits(wind_speed[, limits])	Identify wind speed values that are within limits.

pvanalytics.quality.weather.relative_humidity_limits

pvanalytics.quality.weather.**relative_humidity_limits**(*relative_humidity*, *limits=(0, 100)*)

Identify relative humidity values that are within limits.

Parameters

- **relative_humidity** (*Series*) – Relative humidity in %.
- **limits** (*tuple, default (0, 100)*) – (lower bound, upper bound) for relative humidity.

Returns True if *relative_humidity* >= lower bound and *relative_humidity* <= upper_bound.

Return type Series

Notes

Copyright (c) 2019 SolarArbiter. See the file LICENSES/SOLARFORECASTARBITER_LICENSE at the top level directory of this distribution and at https://github.com/pvlib/pvanalytics/blob/master/LICENSES/SOLARFORECASTARBITER_LICENSE for more information.

Examples using pvanalytics.quality.weather.relative_humidity_limits

- *Weather Limits*

pvanalytics.quality.weather.temperature_limits

pvanalytics.quality.weather.**temperature_limits**(*air_temperature*, *limits=(- 35.0, 50.0)*)

Identify temperature values that are within limits.

Parameters

- **air_temperature** (*Series*) – Air temperature [C].
- **limits** (*tuple, default (-35, 50)*) – (lower bound, upper bound) for temperature.

Returns True if *air_temperature* > lower bound and *air_temperature* < upper bound.

Return type Series

Notes

Examples using pvanalytics.quality.weather.temperature_limits

- *PV Fleets QA Process: Temperature*
- *Weather Limits*

pvanalytics.quality.weather.wind_limits

pvanalytics.quality.weather.**wind_limits**(*wind_speed*, *limits=(0.0, 50.0)*)

Identify wind speed values that are within limits.

Parameters

- **wind_speed** (*Series*) – Wind speed in m/s
- **limits** (*tuple, default (0, 50)*) – (lower bound, upper bound) for wind speed.

Returns True if *wind_speed* >= lower bound and *wind_speed* < upper bound.

Return type Series

Notes

Examples using pvanalytics.quality.weather.wind_limits

- *Weather Limits*

In addition to validating temperature by comparing with limits, module temperature should be positively correlated with irradiance. Poor correlation could indicate that the sensor has become detached from the module, for example. Unlike other functions in the `quality` module which return Boolean masks over the input series, this function returns a single Boolean value indicating whether the entire series has passed (`True`) or failed (`False`) the quality check.

quality.weather.module_temperature_check(...)	Test whether the module temperature is correlated with irradiance.

pvanalytics.quality.weather.module_temperature_check

pvanalytics.quality.weather.**module_temperature_check**(*module_temperature*, *irradiance*, *correlation_min=0.5*)

Test whether the module temperature is correlated with irradiance.

Parameters

- **module_temperature** (*Series*) – Time series of module temperature.
- **irradiance** (*Series*) – Time series of irradiance with the same index as *module_temperature*. This should be of relatively high quality (outliers and other problems removed).
- **correlation_min** (*float, default 0.5*) – Minimum correlation between *module_temperature* and *irradiance* for the module temperature sensor to 'pass'

Returns True if the correlation between *module_temperature* and *irradiance* exceeds *correlation_min*.

Return type bool

Examples using pvanalytics.quality.weather.module_temperature_check

- *Module Temperature Check*

References

2.1.2 Features

Functions for detecting features in the data.

Clipping

Functions for identifying inverter clipping

features.clipping.levels(ac_power[, window, ...])	Label clipping in AC power data based on levels in the data.
features.clipping.threshold(ac_power[, ...])	Detect clipping based on a maximum power threshold.
features.clipping.geometric(ac_power[, ...])	Identify clipping based on a the shape of the *ac_power* curve on each day.

pvanalytics.features.clipping.levels

pvanalytics.features.clipping.**levels**(*ac_power*, *window=4*, *fraction_in_window=0.75*, *rtol=0.005*, *levels=2*)

Label clipping in AC power data based on levels in the data.

Parameters

- **ac_power** (*Series*) – Time series of AC power measurements.
- **window** (*int, default 4*) – Number of data points in a window used to detect clipping.
- **fraction_in_window** (*float, default 0.75*) – Fraction of points which indicate clipping if AC power at each point is close to the plateau level.
- **rtol** (*float, default 5e-3*) – A point is close to a clipped level M if abs(ac_power - M) < rtol * max(ac_power)
- **levels** (*int, default 2*) – Number of clipped power levels to consider.

Returns True when clipping is indicated.

Return type Series

Notes

pvanalytics.features.clipping.threshold

pvanalytics.features.clipping.**threshold**(*ac_power*, *slope_max=0.0035*, *power_min=0.75*, *power_quantile=0.995*, *freq=None*)

Detect clipping based on a maximum power threshold.

This is a two-step process. First a clipping threshold is identified, then any values in *ac_power* greater than or equal to that threshold are flagged.

The clipping threshold is determined by computing a 'daily power curve' which is the *power_quantile* quantile of all values in *ac_power* at each minute of the day. This gives a rough estimate of the maximum power produced at each minute of the day.

The daily power curve is normalized by its maximum and the minutes of the day are identified where the normalized curve's slope is less than *slope_max*. If there is a continuous period of time spanning at least one hour where

the slope is less than *slope_max* and the value of the normalized daily power curve is greater than *power_min* times the median of the normalized daily power curve then the data has clipping in it. If no sufficiently long period with both a low slope and high power exists then there is no clipping in the data. The average of the daily power curve (not normalized) during the longest period that satisfies the criteria above is the clipping threshold.

Parameters

- **ac_power** (*Series*) – DatetimeIndexed series of AC power data.
- **slope_max** (*float, default 0.0035*) – Maximum absolute value of slope of AC power quantile for clipping to be indicated. The default value has been derived empirically to prevent false positives for tracking PV systems.
- **power_min** (*float, default 0.75*) – The power during periods with slope less than *slope_max* must be greater than *power_min* times the median normalized daytime power.
- **power_quantile** (*float, default 0.995*) – Quantile used to calculate the daily power curve.
- **freq** (*string, default None*) – A pandas string offset giving the frequency of data in *ac_power*. If None then the frequency is inferred from the series index.

Returns True when *ac_power* is greater than or equal to the clipping threshold.

Return type Series

Notes

This function is based on the pvfleets_qa_analysis project.

pvanalytics.features.clipping.geometric

pvanalytics.features.clipping.**geometric**(*ac_power*, *window=None*, *slope_max=0.2*, *freq=None*, *tracking=False*)

Identify clipping based on a the shape of the *ac_power* curve on each day.

Each day is checked for periods where the slope of *ac_power* is small. The power values in these periods are used to calculate a minimum and a maximum clipped power level for that day. Any power values that are within this range are flagged as clipped. The methodology for computing the thresholds varies depending on the frequency of *ac_power*. For high frequency data (less than 10 minute timestamp spacing) the minimum clipped power is the mean of the low-slope period(s) on that day minus 2 times the standard deviation in the same period(s). For lower frequency data the absolute minimum and maximum of the low slope period(s) on each day are used.

If the frequency of *ac_power* is less than ten minutes, then *ac_power* is down-sampled to 15 minutes and the mean value in each 15-minute period is used to reduce noise inherent in high frequency data.

Parameters

- **ac_power** (*Series*) – AC power data.
- **window** (*int, optional*) – Size of the rolling window used to identify low-slope periods. If not specified and *tracking* is False then *window=3* is used. If not specified and *tracking* is True then *window=5* is used.
- **slope_max** (*float, default 0.2*) – Maximum difference in maximum and minimum power for a window to be flagged as clipped. Units are percent of average power in the interval.

- **freq** (*str, optional*) – Frequency of *ac_power*. If not specified then `pandas.infer_freq()` is used.
- **tracking** (*bool, default False*) – If True then a larger default *window* is used. If *window* is specified then *tracking* has no effect.

Returns Boolean Series with True for values that appear to be clipped.

Return type Series

Raises `ValueError` – If the index of *ac_power* is not sorted.

Notes

Based on code from the PVFleets QA project.

Examples using `pvanalytics.features.clipping.geometric`

- *Clipping Detection*
- *PV Fleets QA Process: Power*
- *Detect if a System is Tracking*

Clearsky

features.clearsky.reno(ghi, ghi_clearsky)	Identify times when GHI is consistent with clearsky conditions.

pvanalytics.features.clearsky.reno

pvanalytics.features.clearsky.**reno**(*ghi*, *ghi_clearsky*)

Identify times when GHI is consistent with clearsky conditions.

Uses the function `pvlib.clearsky.detect_clearsky()`.

Note: Must be given GHI data with regular (constant) time intervals of 15 minutes or less.

Parameters

- **ghi** (*Series*) – Global horizontal irradiance in W/m^2. Must have an index with time intervals of at most 15 minutes.
- **ghi_clearsky** (*Series*) – Global horizontal irradiance in W/m^2 under clearsky conditions.

Returns True when clear sky conditions are indicated.

Return type Series

Raises `ValueError` – if the time intervals are greater than 15 minutes.

Notes

Clear-sky conditions are inferred when each of six criteria are met; see `pvlib.clearsky.detect_clearsky()` for references and details. Threshold values for each criterion were originally developed for ten minute windows containing one-minute data[1]. As indicated in[2], the algorithm also works for longer windows and data at different intervals if threshold criteria are roughly scaled to the window length. Here, the threshold values are based on [1] with the scaling indicated in [2].

References

Examples using `pvanalytics.features.clearsky.reno`

- *Clear-Sky Detection*

Orientation

System orientation refers to mounting type (fixed or tracker) and the azimuth and tilt of the mounting. A system's orientation can be determined by examining power or POA irradiance on days that are relatively sunny.

This module provides functions that operate on power or POA irradiance to identify system orientation on a daily basis. These functions can tell you whether a day's profile matches that of a fixed system or system with a single-axis tracker.

Care should be taken when interpreting function output since other factors such as malfunctioning trackers can interfere with identification.

features.orientation.fixed_nrel(...[, ...])	Flag days that match the profile of a fixed PV system on a sunny day.
features.orientation.tracking_nrel(...[, ...])	Flag days that match the profile of a single-axis tracking PV system on a sunny day.

pvanalytics.features.orientation.fixed_nrel

pvanalytics.features.orientation.**fixed_nrel**(*power_or_irradiance*, *daytime*, *r2_min=0.94*, *min_hours=5*, *peak_min=None*)

Flag days that match the profile of a fixed PV system on a sunny day.

This algorithm relies on the observation that the power profile of a fixed tilt PV system often resembles a quadratic polynomial on a sunny day, with a single peak when the sun is near the system azimuth.

A day is marked True when the r^2 for a quadratic fit to the power data is greater than *r2_min*.

Parameters

- **power_or_irradiance** (*Series*) – Timezone localized series of power or irradiance measurements.

[1] Reno, M.J. and C.W. Hansen, "Identification of periods of clear sky irradiance in time series of GHI measurements" Renewable Energy, v90, p. 520-531, 2016.

[2] B. H. Ellis, M. Deceglie and A. Jain, "Automatic Detection of Clear-Sky Periods From Irradiance Data," in IEEE Journal of Photovoltaics, vol. 9, no. 4, pp. 998-1005, July 2019. doi: 10.1109/JPHOTOV.2019.2914444

- **daytime** (*Series*) – Boolean series with True for times that are during the day. For best results this mask should exclude early morning and evening as well as night. Data at these times may have problems with shadows that interfere with curve fitting.
- **r2_min** (*float, default 0.94*) – Minimum r^2 of a quadratic fit for a day to be marked True.
- **min_hours** (*float, default 5.0*) – Minimum number of hours with data to attempt a fit on a day.
- **peak_min** (*float, default None*) – The maximum *power_or_irradiance* value for a day must be greater than *peak_min* for a fit to be attempted. If the maximum for a day is less than *peak_min* then the day will be marked False.

Returns True for values on days where *power_or_irradiance* matches the expected parabolic profile for a fixed PV system on a sunny day.

Return type Series

Notes

This algorithm is based on the PVFleets QA Analysis project. Copyright (c) 2020 Alliance for Sustainable Energy, LLC.

Examples using `pvanalytics.features.orientation.fixed_nrel`

- *Flag Sunny Days for a Fixed-Tilt System*

pvanalytics.features.orientation.tracking_nrel

pvanalytics.features.orientation.**tracking_nrel**(*power_or_irradiance*, *daytime*, *r2_min=0.915*, *r2_fixed_max=0.96*, *min_hours=5*, *peak_min=None*, *quadratic_mask=None*)

Flag days that match the profile of a single-axis tracking PV system on a sunny day.

This algorithm relies on the observation that the power profile of a single-axis tracking PV system tends to resemble a quartic polynomial on a sunny day, I.e., two peaks are observed, one before and one after the sun crosses the tracker azimuth. By contrast, the power profile for a fixed tilt PV system often resembles a quadratic polynomial on a sunny day, with a single peak when the sun is near the system azimuth.

The algorithm fits both a quartic and a quadratic polynomial to each day's data. A day is marked True if the quartic fit has a sufficiently high r^2 and the quadratic fit has a sufficiently low r^2. Specifically, a day is marked True when three conditions are met:

1. a restricted quartic[1] must fit the data with r^2 greater than *r2_min*
2. the r^2 for the restricted quartic fit must be greater than the r^2 for a quadratic fit
3. the r^2 for a quadratic fit must be less than *r2_fixed_max*

Values on days where any one of these conditions is not met are marked False.

Parameters

[1] The specific quartic used for this fit is centered within 70 minutes of 12:00, the y-value at the center must be within 15% of the median for the day, and it must open downwards.

- **power_or_irradiance** (*Series*) – Timezone localized series of power or irradiance measurements.
- **daytime** (*Series*) – Boolean series with True for times that are during the day. For best results this mask should exclude early morning and late afternoon as well as night. Data at these times may have problems with shadows that interfere with curve fitting.
- **r2_min** (*float, default 0.915*) – Minimum r^2 of a quartic fit for a day to be marked True.
- **r2_fixed_max** (*float, default 0.96*) – If the r^2 of a quadratic fit exceeds *r2_fixed_max*, then tracking/fixed cannot be distinguished and the day is marked False.
- **min_hours** (*float, default 5.0*) – Minimum number of hours with data to attempt a fit on a day.
- **peak_min** (*float, default None*) – The maximum *power_or_irradiance* value for a day must be greater than *peak_min* for a fit to be attempted. If the maximum for a day is less than *peak_min* then the day will be marked False.
- **quadratic_mask** (*Series, default None*) – If None then *daytime* is used. This Series is used to remove morning and afternoon times from the data before applying a quadratic fit. The mask should typically exclude more data than *daytime* in order to eliminate long tails in the morning or afternoon that can appear if a tracker is stuck in a West or East orientation.

Returns Boolean series with True for every value on a day that has a tracking profile (see criteria above).

Return type Series

Notes

This algorithm is based on the PVFleets QA Analysis project. Copyright (c) 2020 Alliance for Sustainable Energy, LLC.

Examples using `pvanalytics.features.orientation.tracking_nrel`

- *Flag Sunny Days for a Tracking System*

Daytime

Functions that relate to determining day/night periods in a time series, and getting sunrise and sunset times based on the day-night mask outputs.

features.daytime.power_or_irradiance(series)	Return True for values that are during the day.
features.daytime.get_sunrise(daytime_mask[, ...])	Using the outputs of `power_or_irradiance()`, derive sunrise values for each day in the associated time series.
features.daytime.get_sunset(daytime_mask[, ...])	Using the outputs of `power_or_irradiance()`, derive sunset values for each day in the associated time series.

pvanalytics.features.daytime.power_or_irradiance

pvanalytics.features.daytime.**power_or_irradiance**(*series*, *outliers=None*, *low_value_threshold=0.003*, *low_median_threshold=0.0015*, *low_diff_threshold=0.0005*, *median_days=7*, *clipping=None*, *freq=None*, *correction_window=31*, *hours_min=5*, *day_length_difference_max=30*, *day_length_window=14*)

Return True for values that are during the day.

After removing outliers and normalizing the data, a time is classified as night when two of the following three criteria are satisfied:

- near-zero value
- near-zero first-order derivative
- near-zero rolling median at the same time over the surrounding week (see *median_days*)

Mid-day times where power goes near zero or stops changing may be incorrectly classified as night. To correct these errors, night or day periods with duration that is too long or too short are identified, and times in these periods are re-classified to have the majority value at the same time on preceding and following days (as set by *correction_window*).

Finally any values that are True in *clipping* are marked as day.

Parameters

- **series** (*Series*) – Time series of power or irradiance.
- **outliers** (*Series, optional*) – Boolean time series with True for values in *series* that are outliers.
- **low_value_threshold** (*float, default 0.003*) – Maximum normalized power or irradiance value for a time to be considered night.
- **low_median_threshold** (*float, default 0.0015*) – Maximum rolling median of power or irradiance for a time to be considered night.
- **low_diff_threshold** (*float, default 0.0005*) – Maximum derivative of normalized power or irradiance for a time to be considered night.
- **median_days** (*int, default 7*) – Number of days to use to calculate the rolling median at each minute. [days]
- **clipping** (*Series, optional*) – True when clipping indicated. Any values where clipping is indicated are automatically considered 'daytime'.
- **freq** (*str, optional*) – A pandas freqstr specifying the expected timestamp spacing for the series. If None, the frequency will be inferred from the index.
- **correction_window** (*int, default 31*) – Number of adjacent days to examine when correcting day/night classification errors. [days]
- **hours_min** (*float, default 5*) – Minimum number of hours in a contiguous period of day or night. A day/night period shorter than *hours_min* is flagged for error correction. [hours]
- **day_length_difference_max** (*float, default 30*) – Days with length that is *day_length_difference_max* minutes less than the median length of surrounding days are flagged for corrections.

- **day_length_window** (*int, default 14*) – The length of the rolling window used for calculating the median length of the day when correcting errors in the morning or afternoon. [days]

Returns Boolean time series with True for times that are during the day.

Return type Series

Notes

NA values are treated like zeros.

References

Examples using pvanalytics.features.daytime.power_or_irradiance

- *Day-Night Masking*
- *Clearsky Limits for Irradiance Data*
- *Flag Sunny Days for a Fixed-Tilt System*
- *Flag Sunny Days for a Tracking System*
- *PV Fleets QA Process: Irradiance*
- *PV Fleets QA Process: Power*
- *Identifying and estimating time shifts*
- *Detect if a System is Tracking*
- *Module Temperature Check*

pvanalytics.features.daytime.get_sunrise

pvanalytics.features.daytime.**get_sunrise**(*daytime_mask*, *freq=None*, *data_alignment='L'*)

Using the outputs of *power_or_irradiance()*, derive sunrise values for each day in the associated time series.

This function assumes that each midnight-to-midnight period (according to the timezone of the input data) has one sunrise followed by one sunset. In cases where this is not satisfied (timezone of data is substantially different from the location's local time, locations near the poles, etc), or in the case of missing data, the returned sunrise and sunset times may be invalid.

Parameters

- **daytime_mask** (*Series*) – Boolean series delineating night periods from day periods, where day is True and night is False.
- **freq** (*str, optional*) – A pandas freqstr specifying the expected timestamp spacing for the series. If None, the frequency will be inferred from the index of daytime_mask.
- **data_alignment** (*str, default 'L'*) – The data alignment of the series (left-aligned or right-aligned). Data alignment affects the value selected as sunrise. Options are 'L' (left-aligned), 'R' (right-aligned), or 'C' (center-aligned)

Returns Series of daily sunrise times with the same index as daytime_mask.

Return type Series

References

Examples using pvanalytics.features.daytime.get_sunrise

- *PV Fleets QA Process: Irradiance*
- *PV Fleets QA Process: Power*
- *Identifying and estimating time shifts*

pvanalytics.features.daytime.get_sunset

pvanalytics.features.daytime.**get_sunset**(*daytime_mask*, *freq=None*, *data_alignment='L'*)

Using the outputs of *power_or_irradiance()*, derive sunset values for each day in the associated time series.

This function assumes that each midnight-to-midnight period (according to the timezone of the input data) has one sunrise followed by one sunset. In cases where this is not satisfied (timezone of data is substantially different from the location's

local time, locations near the poles, etc), or in the case of missing

data, the returned sunrise and sunset times may be invalid.

Parameters

- **daytime_mask** (*Series*) – Boolean series delineating night periods from day periods, where day is True and night is False.
- **freq** (*str, optional*) – A pandas freqstr specifying the expected timestamp spacing for the series. If None, the frequency will be inferred from the index of daytime_mask.
- **data_alignment** (*str, default 'L'*) – The data alignment of the series (left-aligned or right-aligned). Data alignment affects the value selected as sunrise. Options are 'L' (left-aligned), 'R' (right-aligned), or 'C' (center-aligned)

Returns Series of daily sunrise times with the same index as daytime_mask.

Return type Series

References

Examples using pvanalytics.features.daytime.get_sunset

- *PV Fleets QA Process: Irradiance*
- *PV Fleets QA Process: Power*
- *Identifying and estimating time shifts*

Shading

Functions for labeling shadows.

features.shading.fixed(ghi, daytime, clearsky)	Detects shadows from fixed structures such as wires and poles.

pvanalytics.features.shading.fixed

pvanalytics.features.shading.**fixed**(*ghi*, *daytime*, *clearsky*, *interval=None*, *min_gradient=2*)

Detects shadows from fixed structures such as wires and poles.

Uses morphological image processing methods to identify shadows from fixed local objects in GHI data. GHI data are assumed to be reasonably complete with relatively few missing values and at a fixed time interval nominally of 1 minute over the course of several months. Detection focuses on shadows with relatively short duration. The algorithm forms a 2D image of the GHI data by arranging time of day along the x-axis and day of year along the y-axis. Rapid change in GHI in the x-direction is used to identify edges of shadows; continuity in the y-direction is used to separate local object shading from cloud shadows.

Parameters

- **ghi** (*Series*) – Time series of GHI measurements. Data must be in local time at 1-minute frequency and should cover at least 60 days.
- **daytime** (*Series*) – Boolean series with True for times when the sun is up.
- **clearsky** (*Series*) – Clearsky GHI with same index as *ghi*.
- **interval** (*int, optional*) – Interval between data points in minutes. If not specified the interval is inferred from the frequency of the index of *ghi*.
- **min_gradient** (*float, default 2*) – Threshold value for the morphological gradient[3].

Returns

- *Series* – Boolean series with true for times that are impacted by shadows.
- *ndarray* – A boolean image (black and white) showing the shadows that were detected.

References

2.1.3 System

This module contains functions and classes relating to PV system parameters such as nameplate power, tilt, azimuth, or whether the system is equipped with tracker.

[3] https://docs.scipy.org/doc/scipy/reference/generated/scipy.ndimage.morphological_gradient.html

Tracking

system.Tracker(value)	Enum describing the orientation of a PV System.
system.is_tracking_envelope(series, daytime, ...)	Infer whether the system is equipped with a tracker.

pvanalytics.system.Tracker

class pvanalytics.system.**Tracker**(*value*)

Enum describing the orientation of a PV System.

Attributes

FIXED	A system with a fixed azimuth and tilt.
TRACKING	A system equipped with a tracker.
UNKNOWN	A system where the tracking cannot be determined.

Examples using pvanalytics.system.Tracker

- *PV Fleets QA Process: Power*
- *Detect if a System is Tracking*

pvanalytics.system.is_tracking_envelope

pvanalytics.system.**is_tracking_envelope**(*series*, *daytime*, *clipping*, *clip_max=0.1*, *envelope_quantile=0.995*, *envelope_min_fraction=0.05*, *fit_median=True*, *median_min_fraction=0.025*, *median_r2_min=0.9*, *fit_params=None*, *seasonal_split='north-america'*)

Infer whether the system is equipped with a tracker.

Data is grouped by season (optional) and within each season by the minute of the day. A maximum power or irradiance envelope (the *envelope_quantile* value at each minute) is calculated. Quadratic and quartic curves are fit to this daily envelope and the r^2 of the curve fits are used determine whether the system is tracking or fixed.

If the quadratic fit is a sufficiently good in both seasons, then `Tracker.FIXED` is returned.

If, in both seasons, the quartic fit is sufficiently good and the quadratic fit is sufficiently bad, then `Tracker.TRACKING` is returned.

If neither fit is sufficiently good, or the results from each season disagree, then `Tracker.UNKNOWN` is returned.

Optionally, an additional fit is made to the median of the data at each minute to confirm the determination of tracking or fixed. If performed, this result must be consistent with the fit to the upper envelope. If not, `Tracker.UNKNOWN` is returned.

Parameters

- **series** (*Series*) – Timezone localized Series of power or irradiance data.
- **daytime** (*Series*) – Boolean Series with True for times that are during the day.

- **clipping** (*Series*) – Boolean Series identifying where power or irradiance is being clipped.
- **clip_max** (*float, default 0.1*) – If the fraction of data flagged as clipped is greater than *clip_max* then it cannot be determined whether the system is tracking or fixed and `Tracker.UNKNOWN` is returned.
- **envelope_quantile** (*float, default 0.995*) – Quantile used to determine the upper power or irradiance envelope.
- **envelope_min_fraction** (*float, default 0.05*) – After calculating the power or irradiance envelope, data less than *envelope_min_fraction* times the maximum of the envelope is removed. This excludes data from morning and evening that may interfere with curve fitting.
- **fit_median** (*boolean, default True*) – Perform a secondary fit with the median power or irradiance to validate that the profile is consistent through the entire data set.
- **median_min_fraction** (*float, default 0.025*) – After calculating the median power or irradiance at each minute, data less than *median_min_fraction* times the maximum is removed. This excludes data from morning and evening that may interfere with curve fitting.
- **median_r2_min** (*float, default 0.9*) – Minimum r^2 for a curve fit to the median power or irradiance at each minute of the day (Applies only if *fit_median* is True).
- **fit_params** (*dict or None, default None*) – Minimum r-squared for curve fits according to the fraction of data with clipping. This should be a dictionary with tuple keys and dictionary values. The key must be a 2-tuple of `(clipping_min, clipping_max)` where the values specify the minimum and maximum fraction of data with clipping for which the associated fit parameters are applicable. The values of the dictionary are themselves dictionaries with keys `'fixed'` and `'tracking'`, which give the minimum r^2 for the curve fits, and `'fixed_max'` which gives the maximum r^2 for a quadratic fit if the system appears to have a tracker.

  If None `PVFLEETS_FIT_PARAMS` is used.
- **seasonal_split** (*dict or str or None, default 'north-america'*) – A dictionary with two keys, 'winter' and 'summer' with a list of integers specifying the winter months and summer months respectively. Seasonal grouping can be disabled by passing *seasonal_split=None*. Either season can be ignored by passing a dict that omits the key or sets its value to None. The default value, 'north-america' uses `{'winter': [11, 12, 1, 2], 'summer': [5, 6, 7, 8]}` which works well for PV systems located in North America.

Returns The tracking determined by curve fitting (FIXED, TRACKING, or UNKNOWN).

Return type *Tracker*

Notes

Derived from the PVFleets QA Analysis project.

See also:

pvanalytics.features.orientation.tracking_nrel, *pvanalytics.features.orientation.fixed_nrel*

Examples using pvanalytics.system.is_tracking_envelope

- *PV Fleets QA Process: Power*
- *Detect if a System is Tracking*

Orientation

The following function can be used to infer system orientation from power or plane of array irradiance measurements.

system.infer_orientation_daily_peak(...)	Determine system azimuth and tilt from power or POA using solar azimuth at the daily peak.
system.infer_orientation_fit_pvwatts(...[, ...])	Get the tilt and azimuth that give PVWatts v5 output that most closely fits the data in *power_ac*.

pvanalytics.system.infer_orientation_daily_peak

pvanalytics.system.**infer_orientation_daily_peak**(*power_or_poa*, *sunny*, *tilts*, *azimuths*, *solar_azimuth*, *solar_zenith*, *ghi*, *dhi*, *dni*)

Determine system azimuth and tilt from power or POA using solar azimuth at the daily peak.

The time of the daily peak is estimated by fitting a quadratic to to the data for each day in *power_or_poa* and finding the vertex of the fit. A brute force search is performed on clearsky POA irradiance for all pairs of candidate azimuths and tilts (*azimuths* and *tilts*) to find the pair that results in the closest azimuth to the azimuths calculated at the peak times from the curve fitting step. Closest is determined by minimizing the sum of squared difference between the solar azimuth at the peak time in *power_or_poa* and the solar azimuth at maximum clearsky POA irradiance.

The accuracy of the tilt and azimuth returned by this function will vary with the time-resolution of the clearsky and solar position data. For the best accuracy pass *solar_azimuth*, *solar_zenith*, and the clearsky data (*ghi*, *dhi*, and *dni*) with one-minute timestamp spacing. If *solar_azimuth* has timestamp spacing less than one minute it will be resampled and interpolated to estimate azimuth at each minute of the day. Regardless of the timestamp spacing these parameters must cover the same days as *power_or_poa*.

Parameters

- **power_or_poa** (*Series*) – Timezone localized series of power or POA irradiance measurements.
- **sunny** (*Series*) – Boolean series with True for values during clearsky conditions.
- **tilts** (*array-like*) – Candidate tilts in degrees.
- **azimuths** (*array-like*) – Candidate azimuths in degrees.
- **solar_azimuth** (*Series*) – Time series of solar azimuth.
- **solar_zenith** (*Series*) – Time series of solar zenith.
- **ghi** (*Series*) – Clear sky GHI.
- **dhi** (*Series*) – Clear sky DHI.
- **dni** (*Series*) – Clear sky DNI.

Returns

- **azimuth** (*float*)

- **tilt** (*float*)

Notes

Based on PVFleets QA project.

pvanalytics.system.infer_orientation_fit_pvwatts

pvanalytics.system.**infer_orientation_fit_pvwatts**(*power_ac*, *ghi*, *dhi*, *dni*, *solar_zenith*, *solar_azimuth*, *temperature=25*, *wind_speed=0*, *temperature_coefficient=- 0.0047*, *temperature_model_parameters=None*, *azimuth_min=0*, *azimuth_max=360*, *tilt_min=0*, *tilt_max=90*)

Get the tilt and azimuth that give PVWatts v5 output that most closely fits the data in *power_ac*.

Input data *power_ac*, *ghi*, *dhi*, *dni* should reflect clear-sky conditions.

Uses non-linear least squares to optimize over four free variables to find the values that result in the best fit between power modeled using PVWatts and *power_ac*. The four free variables are

- surface tilt
- surface azimuth
- the DC capacity of the system
- the DC input limit of the inverter.

Of these four parameters, only tilt and azimuth are returned. While, DC capacity and the DC input limit are calculated, their values may not be accurate. While its value is not returned, because the DC input limit is used as a free variable for the optimization process, this function can operate on *power_ac* data that includes inverter clipping.

All parameters passed as a Series must have the same index and must not contain any undefined values (i.e. NaNs). If any input contains NaNs a ValueError is raised.

Parameters

- **power_ac** (*Series*) – AC power from the system in clear sky conditions.
- **ghi** (*Series*) – Clear sky GHI with same index as *power_ac*. [W/m^2]
- **dhi** (*Series*) – Clear sky DHI with same index as *power_ac*. [W/m^2]
- **dni** (*Series*) – Clear sky DNI with same index as *power_ac*. [W/m^2]
- **solar_zenith** (*Series*) – Solar zenith. [degrees]
- **solar_azimuth** (*Series*) – Solar azimuth. [degrees]
- **temperature** (*float or Series, default 25*) – Air temperature at which to model the hypothetical system. If a float then a constant temperature is used. If a Series, must have the same index as *power_ac*. [C]
- **wind_speed** (*float or Series, default 0*) – Wind speed. If a float then a constant wind speed is used. If a Series, must have the same index as *power_ac*. [m/s]
- **temperature_coefficient** (*float, default -0.004*) – Temperature coefficient of DC power. [1/C]

- **temperature_model_parameters** (*dict, optional*) – Parameters fot the cell temperature model. If not specified `pvlib.temperature.TEMPERATURE_MODEL_PARAMETERS['sapm'][ 'open_rack_glass_glass']` is used. See `pvlib.temperature.sapm_cell()` for more information.
- **azimuth_min** (*Float, default 0*) – Minimum possible azimuth (bounds) for the least squares search problem. [degrees]
- **azimuth_max** (*Float, default 360*) – Maximum possible azimuth (bounds) for the least squares search problem. [degrees]
- **tilt_min** (*Float, default 0*) – Minimum possible tilt (bounds) for the least squares search problem. [degrees]
- **tilt_max** (*Float, default 90*) – Maximum possible tilt (bounds) for the least squares search problem. [degrees]

Returns

- **surface_tilt** (*float*) – Tilt of the array. [degrees]
- **surface_azimuth** (*float*) – Azimuth of the array. [degrees]
- **r_squared** (*float*) – r^2 value for the fit at the returned orientation.

Raises `ValueError` – If any input passed as a Series contains undefined values (i.e. NaNs).

Notes

To prevent significant slowdown, this function uses the SAPM thermal model (`sapm_cell()`) instead of the model specified in the PVWatts v5 reference[1] (`fuentes()`).

References

Examples using `pvanalytics.system.infer_orientation_fit_pvwatts`

- *PV Fleets QA Process: Power*
- *Infer Array Tilt/Azimuth - PVWatts Method*

2.1.4 Metrics

Performance Ratio

The following functions can be used to calculate system performance metrics.

metrics.performance_ratio_nrel(poa_global, ...)	Calculate NREL Performance Ratio.

[1] Aron Dobos, "PVWatts Version 5 Manual", NREL/TP-6A20-62641 (2014). DOI: 10.2172/1158421

pvanalytics.metrics.performance_ratio_nrel

pvanalytics.metrics.**performance_ratio_nrel**(*poa_global*, *temp_air*, *wind_speed*, *pac*, *pdc0*, *a=- 3.56*, *b=- 0.075*, *deltaT=3*, *gamma_pdc=- 0.00433*)

Calculate NREL Performance Ratio.

See equation [5] in Weather-Corrected Performance Ratio[1] for details on the weighted method for Tref.

Parameters

- **poa_global** (*numeric*) – Total incident irradiance [W/m^2].
- **temp_air** (*numeric*) – Ambient dry bulb temperature [C].
- **wind_speed** (*numeric*) – Wind speed at a height of 10 meters [m/s].
- **pac** (*numeric*) – AC power [kW].
- **pdc0** (*float*) – Power of the modules at 1000 W/m2 and cell reference temperature, otherwise referred to as the PV array STC nameplate rating [kW].
- **a** (*float*) – Parameter a in SAPM model [unitless].
- **b** (*float*) – Parameter b in in SAPM model [s/m].
- **deltaT** (*float*) – Parameter ΔT in SAPM model [C].
- **gamma_pdc** (*float*) – The temperature coefficient in units of 1/C. Typically -0.002 to -0.005 per degree C [1/C].

Returns **performance_ratio** – Performance Ratio of data.

Return type float

References

Examples using pvanalytics.metrics.performance_ratio_nrel

- *Calculate Performance Ratio (NREL)*

Variability

Functions to calculate variability statistics.

metrics.variability_index(measured, clearsky)	Calculate the variability index.

[1] Dierauf et al. "Weather-Corrected Performance Ratio". NREL, 2013. https://www.nrel.gov/docs/fy13osti/57991.pdf

pvanalytics.metrics.variability_index

pvanalytics.metrics.**variability_index**(*measured*, *clearsky*, *freq=None*)

Calculate the variability index.

Parameters

- **measured** (*Series*) – Time series of measured GHI. [W/m2]
- **clearsky** (*Series*) – Time series of the expected clearsky GHI. [W/m2]
- **freq** (*pandas datetime offset, optional*) – Aggregation period (e.g. 'D' for daily). If not specified, the variability index for the entire time series will be returned.

Returns **vi** – The calculated variability index

Return type Series or float

References

Examples using pvanalytics.metrics.variability_index

- *Calculate Variability Index*

2.2 Example Gallery

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

2.2.1 Clearsky Detection

This includes examples for identifying clearsky periods in time series data.

2.2.2 Clipping

This includes examples for identifying clipping in AC power time series.

2.2.3 Day-Night Masking

This includes examples for identifying day-night periods in time series data.

2.2.4 Gaps

This includes examples for identifying gaps and other related issues in time series data, including interpolated periods and stale data periods.

2.2.5 Irradiance-Quality

This includes examples for running irradiance quality checks on irradiance time series data.

2.2.6 Metrics

This includes examples for quantifying system time series metrics, including variability index (VI) and NREL performance ratio (PR).

2.2.7 Orientation

This includes examples related to the orientation of a system (fixed-tilt, tracking).

2.2.8 Outliers

This includes examples for identifying outliers in time series data.

2.2.9 PVFleets QA Examples

These examples highlight the QA processes for temperature, power and irradiance data streams that are used in the NREL PV Fleet Performance Data Initiative (https://www.nrel.gov/pv/fleet-performance-data-initiative.html).

2.2.10 Data/Time Shifts

This includes examples for identifying data/capacity/time shifts in time series data.

2.2.11 System

This includes examples for system parameter estimation, including azimuth and tilt estimation, and determination if the system is fixed tilt or tracking.

2.2.12 Weather

This includes examples for weather quality checks.

Clearsky Detection

This includes examples for identifying clearsky periods in time series data.

Clear-Sky Detection

Identifying periods of clear-sky conditions using measured irradiance.

Identifying and filtering for clear-sky conditions is a useful way to reduce noise when analyzing measured data. This example shows how to use *pvanalytics.features.clearsky.reno()* to identify clear-sky conditions using measured GHI data. For this example we'll use GHI measurements from NREL in Golden CO.

```
import pvanalytics
from pvanalytics.features.clearsky import reno
import pvlib
import matplotlib.pyplot as plt
import pandas as pd
import pathlib
```

First, read in the GHI measurements. For this example we'll use an example file included in pvanalytics covering a single day, but the same process applies to data of any length.

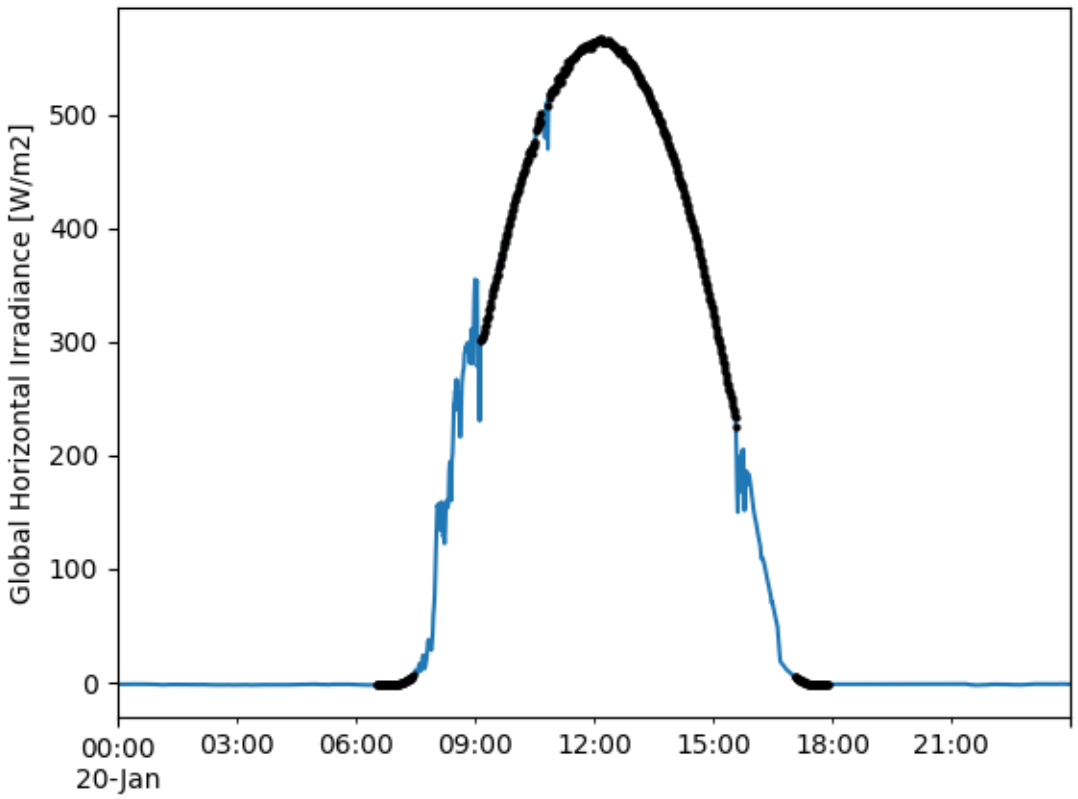
```
pvanalytics_dir = pathlib.Path(pvanalytics.__file__).parent
ghi_file = pvanalytics_dir / 'data' / 'midc_bms_ghi_20220120.csv'
data = pd.read_csv(ghi_file, index_col=0, parse_dates=True)

# or you can fetch the data straight from the source using pvlib:
# date = pd.to_datetime('2022-01-20')
# data = pvlib.iotools.read_midc_raw_data_from_nrel('BMS', date, date)

measured_ghi = data['Global CMP22 (vent/cor) [W/m^2]']
```

Now model clear-sky irradiance for the location and times of the measured data:

```
location = pvlib.location.Location(39.742, -105.18)
clearsky = location.get_clearsky(data.index)
clearsky_ghi = clearsky['ghi']
```

Finally, use *pvanalytics.features.clearsky.reno()* to identify measurements during clear-sky conditions:

```
is_clearsky = reno(measured_ghi, clearsky_ghi)

# clear-sky times indicated in black
measured_ghi.plot()
measured_ghi[is_clearsky].plot(ls='', marker='o', ms=2, c='k')
plt.ylabel('Global Horizontal Irradiance [W/m2]')
plt.show()
```



Total running time of the script: (0 minutes 0.329 seconds)

Clipping

This includes examples for identifying clipping in AC power time series.

Clipping Detection

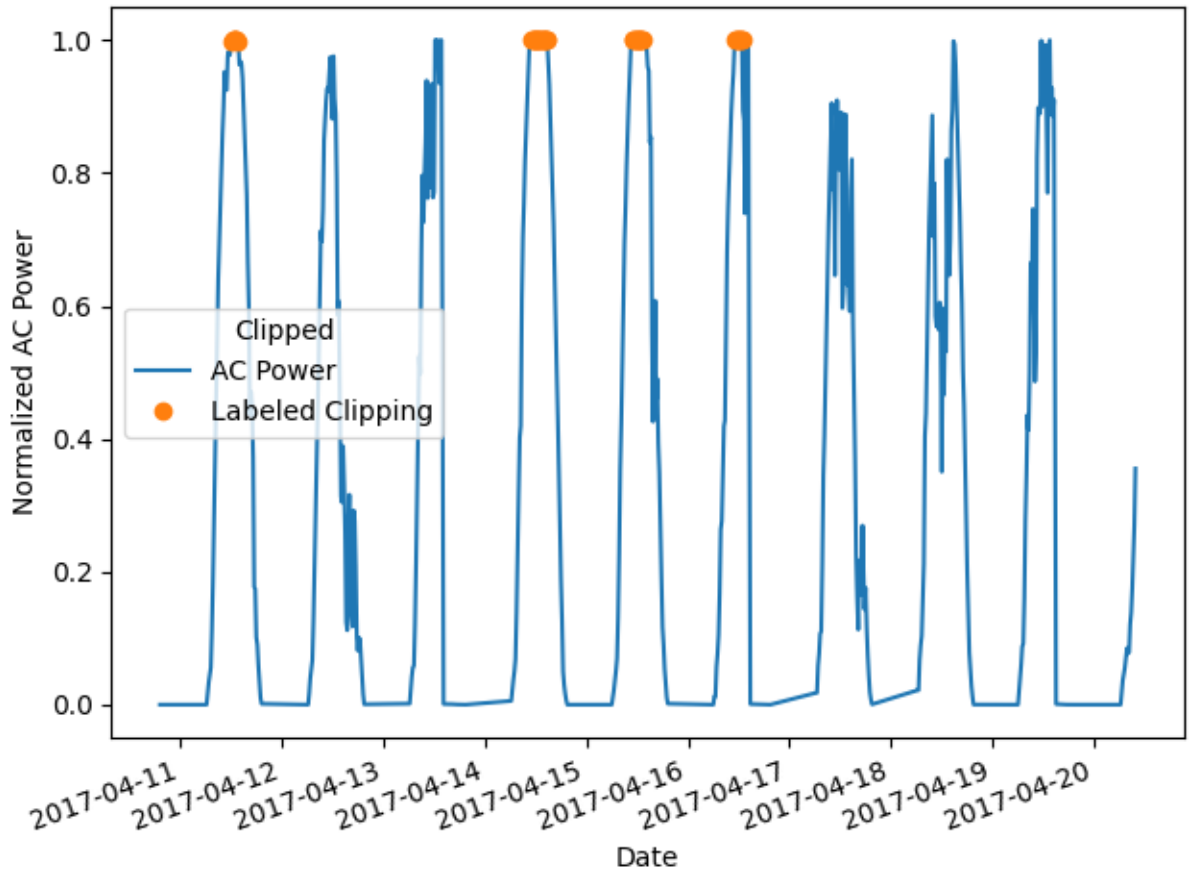
Identifying clipping periods using the PVAnalytics clipping module.

Identifying and removing clipping periods from AC power time series data aids in generating more accurate degradation analysis results, as using clipped data can lead to under-predicting degradation. In this example, we show how to use *pvanalytics.features.clipping.geometric()* to mask clipping periods in an AC power time series. We use a normalized time series example provided by the PV Fleets Initiative, where clipping periods are labeled as True, and non-clipping periods are labeled as False. This example is adapted from the DuraMAT DataHub clipping data set: https://datahub.duramat.org/dataset/inverter-clipping-ml-training-set-real-data

```
import pvanalytics
from pvanalytics.features.clipping import geometric
import matplotlib.pyplot as plt
import pandas as pd
import pathlib
import numpy as np
```

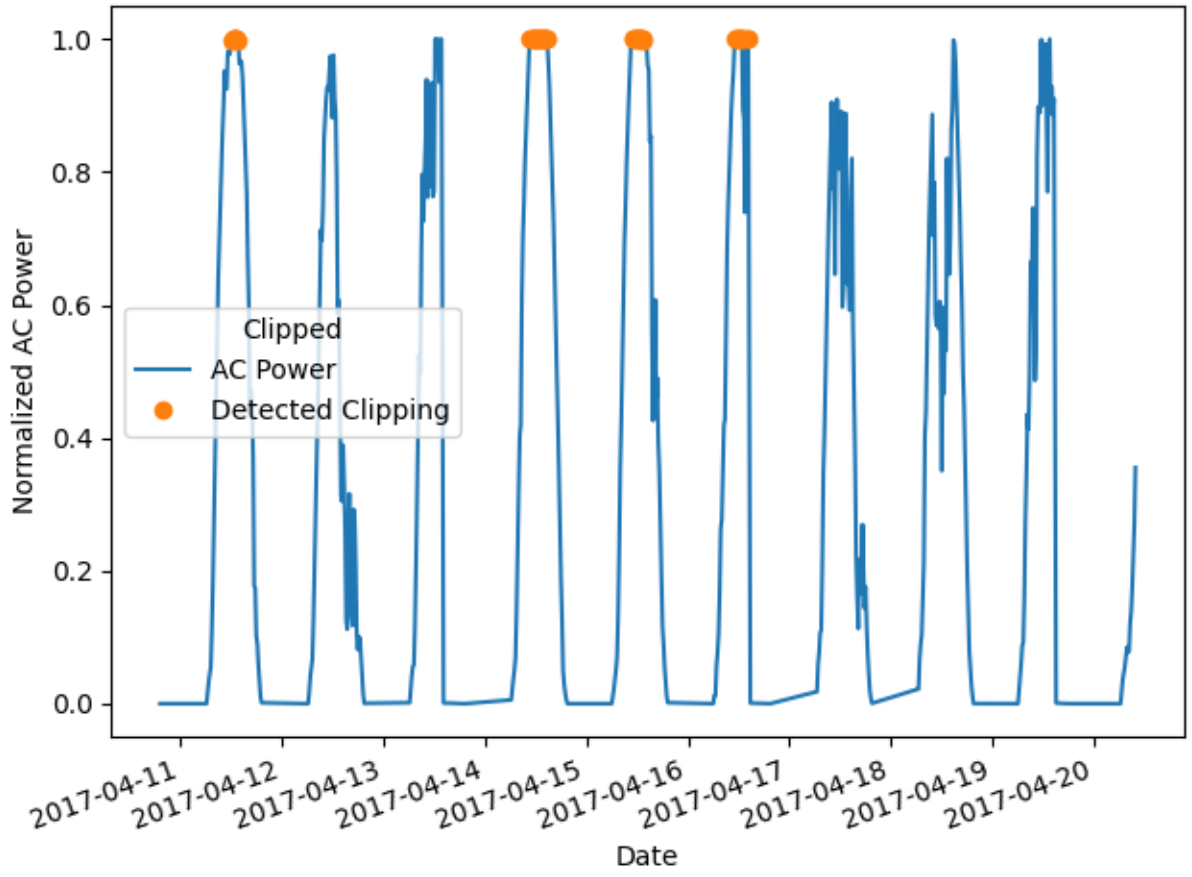
First, read in the ac_power_inv_7539 example, and visualize a subset of the clipping periods via the "label" mask column.

```
pvanalytics_dir = pathlib.Path(pvanalytics.__file__).parent
ac_power_file_1 = pvanalytics_dir / 'data' / 'ac_power_inv_7539.csv'
data = pd.read_csv(ac_power_file_1, index_col=0, parse_dates=True)
data['label'] = data['label'].astype(bool)
# This is the known frequency of the time series. You may need to infer
# the frequency or set the frequency with your AC power time series.
freq = "15min"

data['value_normalized'].plot()
data.loc[data['label'], 'value_normalized'].plot(ls='', marker='o')
plt.legend(labels=["AC Power", "Labeled Clipping"],
           title="Clipped")
plt.xticks(rotation=20)
plt.xlabel("Date")
plt.ylabel("Normalized AC Power")
plt.tight_layout()
plt.show()
```



Now, use *pvanalytics.features.clipping.geometric()* to identify clipping periods in the time series. Re-plot the data subset with this mask.

```
predicted_clipping_mask = geometric(ac_power=data['value_normalized'],
                                    freq=freq)
data['value_normalized'].plot()
data.loc[predicted_clipping_mask, 'value_normalized'].plot(ls='', marker='o')
plt.legend(labels=["AC Power", "Detected Clipping"],
           title="Clipped")
plt.xticks(rotation=20)
plt.xlabel("Date")
plt.ylabel("Normalized AC Power")
plt.tight_layout()
plt.show()
```



Compare the filter results to the ground-truth labeled data side-by-side, and generate an accuracy metric.

```
acc = 100 * np.sum(np.equal(data.label,
                            predicted_clipping_mask))/len(data.label)
print("Overall model prediction accuracy: " + str(round(acc, 2)) + "%")
```

```
Overall model prediction accuracy: 99.2%
```

Total running time of the script: (0 minutes 0.458 seconds)

Day-Night Masking

This includes examples for identifying day-night periods in time series data.

Day-Night Masking

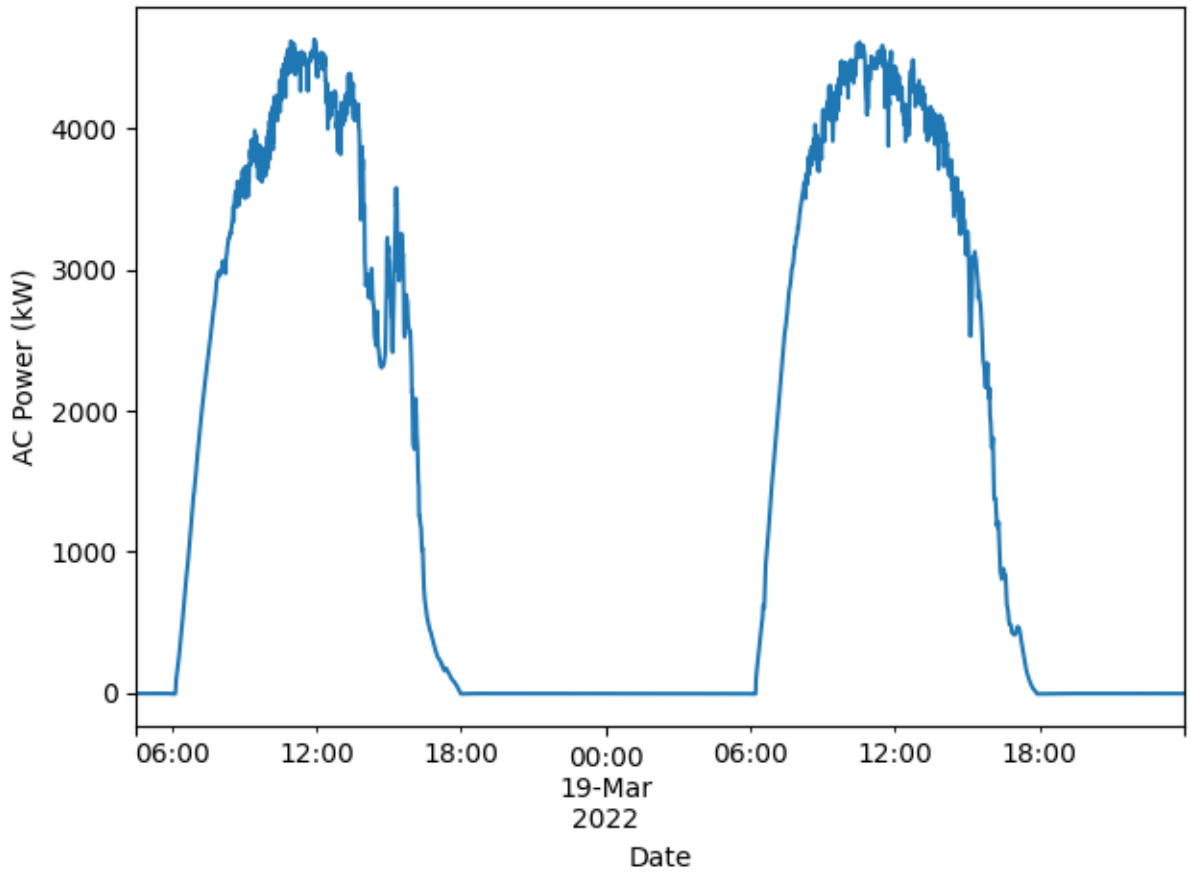
Masking day-night periods using the PVAnalytics daytime module.

Identifying and masking day-night periods in an AC power time series or irradiance time series can aid in future data analysis, such as detecting if a time series has daylight savings time or time shifts. Here, we use *pvanalytics.features.daytime.power_or_irradiance()* to mask day/night periods, as well as to estimate sunrise and sunset times in the data set. This function is particularly useful for cases where the time zone of a data stream is unknown or incorrect, as its outputs can be used to determine time zone.

```
import pvanalytics
from pvanalytics.features.daytime import power_or_irradiance
import matplotlib.pyplot as plt
import pandas as pd
import pathlib
import pvlib
import numpy as np
```

First, read in the 1-minute sampled AC power time series data, taken from the SERF East installation on the NREL campus. This sample is provided from the NREL PVDAQ database, and contains a column representing an AC power data stream.

```
pvanalytics_dir = pathlib.Path(pvanalytics.__file__).parent
ac_power_file = pvanalytics_dir / 'data' / 'serf_east_1min_ac_power.csv'
data = pd.read_csv(ac_power_file, index_col=0, parse_dates=True)
data = data.sort_index()

# This is the known frequency of the time series. You may need to infer
# the frequency or set the frequency with your AC power time series.
freq = "1min"
# These are the latitude-longitude coordinates associated with the
# SERF East system.
latitude = 39.742
longitude = -105.173
# Plot the time series.
data['ac_power__752'].plot()
plt.xlabel("Date")
plt.ylabel("AC Power (kW)")
plt.tight_layout()
plt.show()
```



It is critical to set all negative values in the AC power time series to 0 for *pvanalytics.features.daytime.power_or_irradiance()* to work properly. Negative erroneous data may affect daytime mask assignments.

```
data.loc[data['ac_power__752'] < 0, 'ac_power__752'] = 0
```

Now, use *pvanalytics.features.daytime.power_or_irradiance()* to mask day periods in the time series.

```
predicted_day_night_mask = power_or_irradiance(series=data['ac_power__752'],
                                               freq=freq)
```

Function pvlib.solarposition.sun_rise_set_transit_spa() is used to get ground-truth sunrise and sunset times for each day at the site location, and a SPA-daytime mask is calculated based on these times. Data associated with SPA daytime periods is labeled as True, and data associated with SPA nighttime periods is labeled as False. SPA sunrise and sunset times are used here as a point of comparison to the *pvanalytics.features.daytime.power_or_irradiance()* outputs. SPA-based sunrise and sunset values are not needed to run *pvanalytics.features.daytime.power_or_irradiance()*.

```
sunrise_sunset_df = pvlib.solarposition.sun_rise_set_transit_spa(data.index,
                                                                 latitude,
                                                                 longitude)
data['sunrise_time'] = sunrise_sunset_df['sunrise']
data['sunset_time'] = sunrise_sunset_df['sunset']

data['daytime_mask'] = True
```

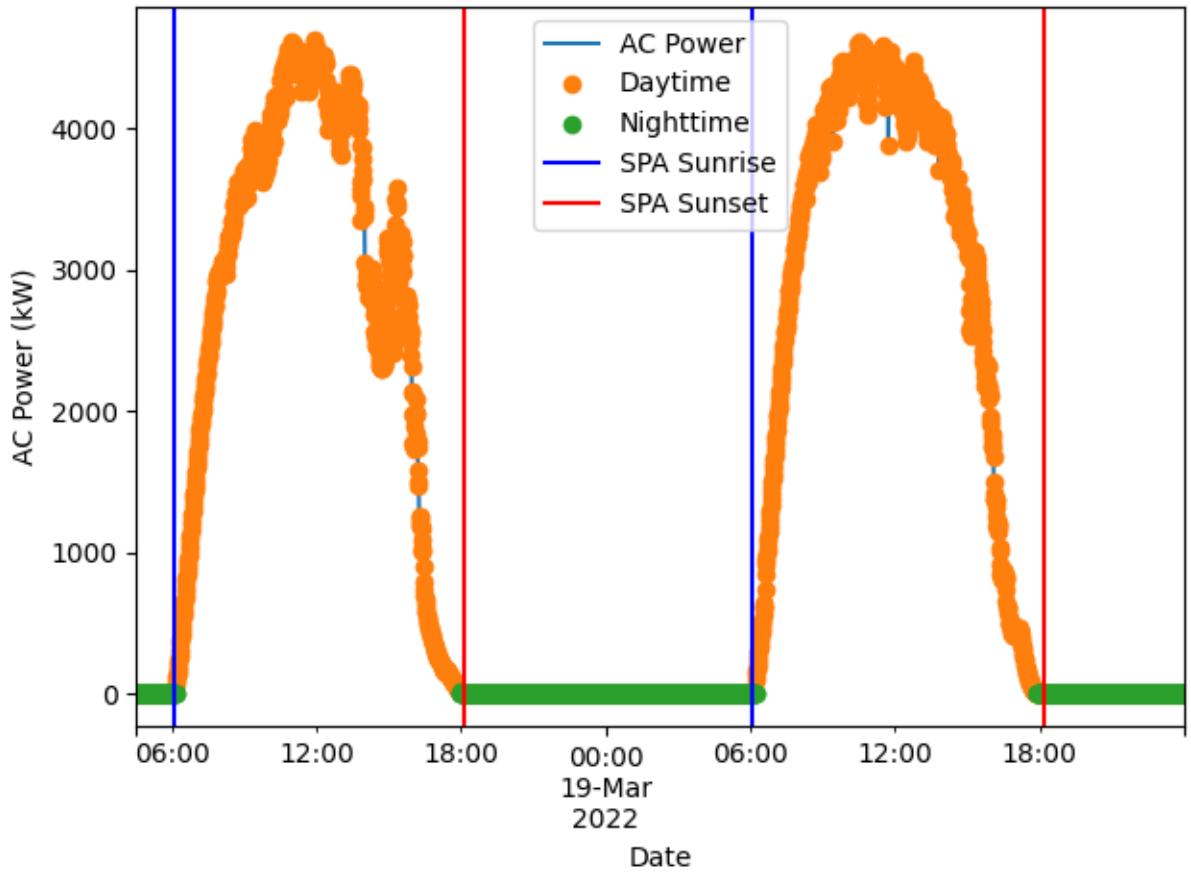
(continues on next page)

(continued from previous page)

```
data.loc[(data.index < data.sunrise_time) |
         (data.index > data.sunset_time), "daytime_mask"] = False
```

Plot the AC power data stream with the mask output from *pvanalytics.features.daytime.power_or_irradiance()*, as well as the SPA-calculated sunrise and sunset

```
data['ac_power__752'].plot()
data.loc[predicted_day_night_mask, 'ac_power__752'].plot(ls='', marker='o')
data.loc[~predicted_day_night_mask, 'ac_power__752'].plot(ls='', marker='o')
sunrise_sunset_times = sunrise_sunset_df[['sunrise',
                                          'sunset']].drop_duplicates()
for sunrise, sunset in sunrise_sunset_times.itertuples(index=False):
    plt.axvline(x=sunrise, c="blue")
    plt.axvline(x=sunset, c="red")
plt.legend(labels=["AC Power", "Daytime", "Nighttime",
                   "SPA Sunrise", "SPA Sunset"])
plt.xlabel("Date")
plt.ylabel("AC Power (kW)")
plt.tight_layout()
plt.show()
```



Compare the predicted mask to the ground-truth SPA mask, to get the model accuracy. Also, compare sunrise and sunset times for the predicted mask compared to the ground truth sunrise and sunset times.

```
acc = 100 * np.sum(np.equal(data.daytime_mask,
                            predicted_day_night_mask))/len(data.daytime_mask)
print("Overall model prediction accuracy: " + str(round(acc, 2)) + "%")

# Generate predicted + SPA sunrise times for each day
print("Sunrise Comparison:")
print(pd.DataFrame({'predicted_sunrise': predicted_day_night_mask
                    .index[predicted_day_night_mask]
                    .to_series().resample("d").first(),
                    'pvlib_spa_sunrise': sunrise_sunset_df["sunrise"]
                    .resample("d").first()}))
# Generate predicted + SPA sunset times for each day
print("Sunset Comparison:")
print(pd.DataFrame({'predicted_sunset': predicted_day_night_mask
                    .index[predicted_day_night_mask]
                    .to_series().resample("d").last(),
                    'pvlib_spa_sunset': sunrise_sunset_df["sunrise"]
                    .resample("d").last()}))
```

```
Overall model prediction accuracy: 98.39%
Sunrise Comparison:
                                  predicted_sunrise                   pvlib_spa_sunrise
measured_on
2022-03-18 00:00:00-07:00 2022-03-18 06:11:00-07:00    2022-03-18 06:07:09.226592-07:00
2022-03-19 00:00:00-07:00 2022-03-19 06:14:00-07:00 2022-03-19 06:05:32.867153920-07:00
Sunset Comparison:
                                   predicted_sunset                    pvlib_spa_sunset
measured_on
2022-03-18 00:00:00-07:00 2022-03-18 17:56:00-07:00    2022-03-18 06:07:09.226592-07:00
2022-03-19 00:00:00-07:00 2022-03-19 17:52:00-07:00 2022-03-19 06:05:32.867153920-07:00
```

Total running time of the script: (0 minutes 1.098 seconds)

Gaps

This includes examples for identifying gaps and other related issues in time series data, including interpolated periods and stale data periods.

Interpolated Data Periods

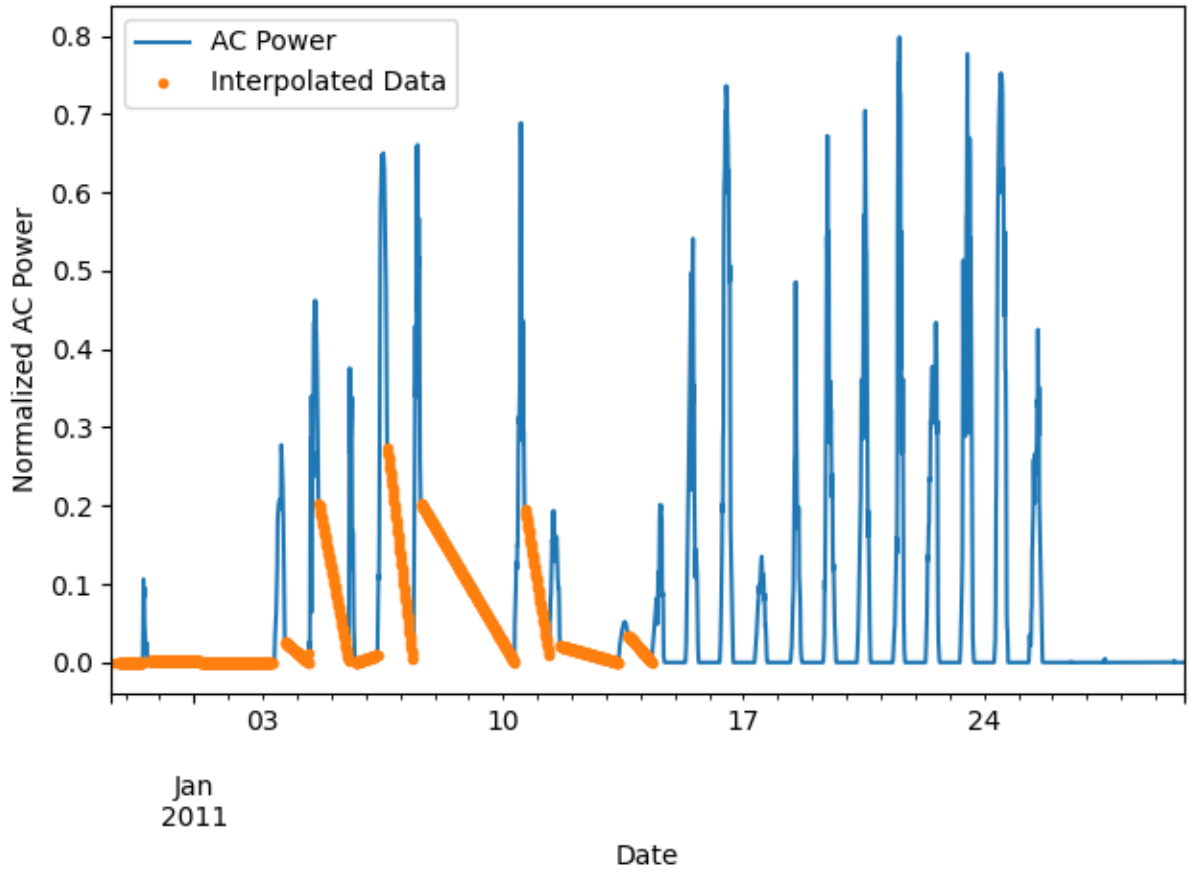
Identifying periods in a time series where the data has been linearly interpolated.

Identifying periods where time series data has been linearly interpolated and removing these periods may help to reduce noise when performing future data analysis. This example shows how to use *pvanalytics.quality.gaps.interpolation_diff()*, which identifies and masks linearly interpolated periods.

```
import pvanalytics
from pvanalytics.quality import gaps
import matplotlib.pyplot as plt
import pandas as pd
import pathlib
```

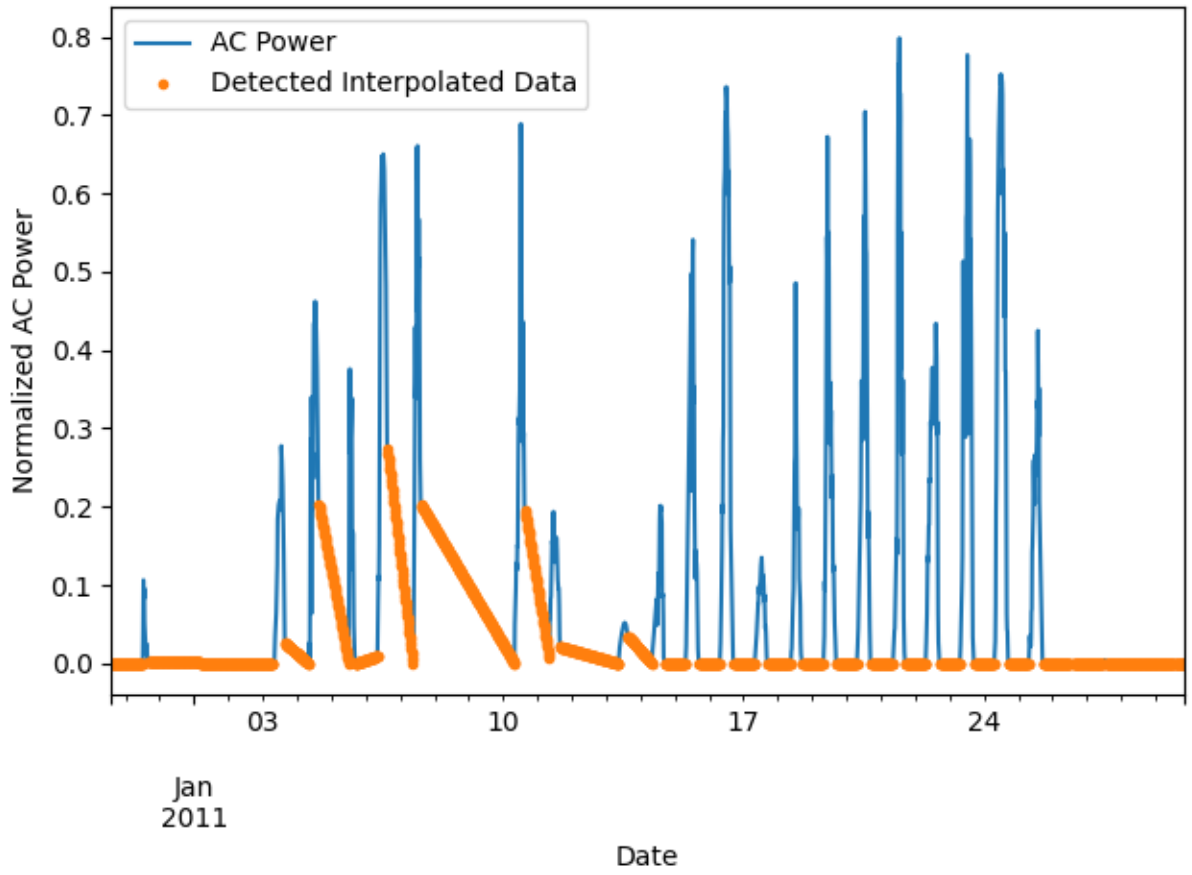
First, we import the AC power data stream that we are going to check for interpolated periods. The time series we download is a normalized AC power time series from the PV Fleets Initiative, and is available via the DuraMAT DataHub: https://datahub.duramat.org/dataset/inverter-clipping-ml-training-set-real-data. This data set has a Pandas DateTime index, with the min-max normalized AC power time series represented in the 'value_normalized' column. There is also an "interpolated_data_mask" column, where interpolated periods are labeled as True, and all other data is labeled as False. The data is sampled at 15-minute intervals.

```
pvanalytics_dir = pathlib.Path(pvanalytics.__file__).parent
file = pvanalytics_dir / 'data' / 'ac_power_inv_2173_interpolated_data.csv'
data = pd.read_csv(file, index_col=0, parse_dates=True)
data = data.asfreq("15T")
data['value_normalized'].plot()
data.loc[data["interpolated_data_mask"], "value_normalized"].plot(ls='',
                                                                  marker='.')
plt.legend(labels=["AC Power", "Interpolated Data"])
plt.xlabel("Date")
plt.ylabel("Normalized AC Power")
plt.tight_layout()
plt.show()
```



Now, we use *pvanalytics.quality.gaps.interpolation_diff()* to identify linearly interpolated periods in the time series. We re-plot the data with this mask. Please note that nighttime periods generally consist of repeating 0 values; this means that these periods can be linearly interpolated. Consequently, these periods are flagged by *pvanalytics.quality.gaps.interpolation_diff()*.

```
detected_interpolated_data_mask = gaps.interpolation_diff(
    data['value_normalized'])
data['value_normalized'].plot()
data.loc[detected_interpolated_data_mask,
         "value_normalized"].plot(ls='', marker='.')
plt.legend(labels=["AC Power", "Detected Interpolated Data"])
plt.xlabel("Date")
plt.ylabel("Normalized AC Power")
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.844 seconds)

Stale Data Periods

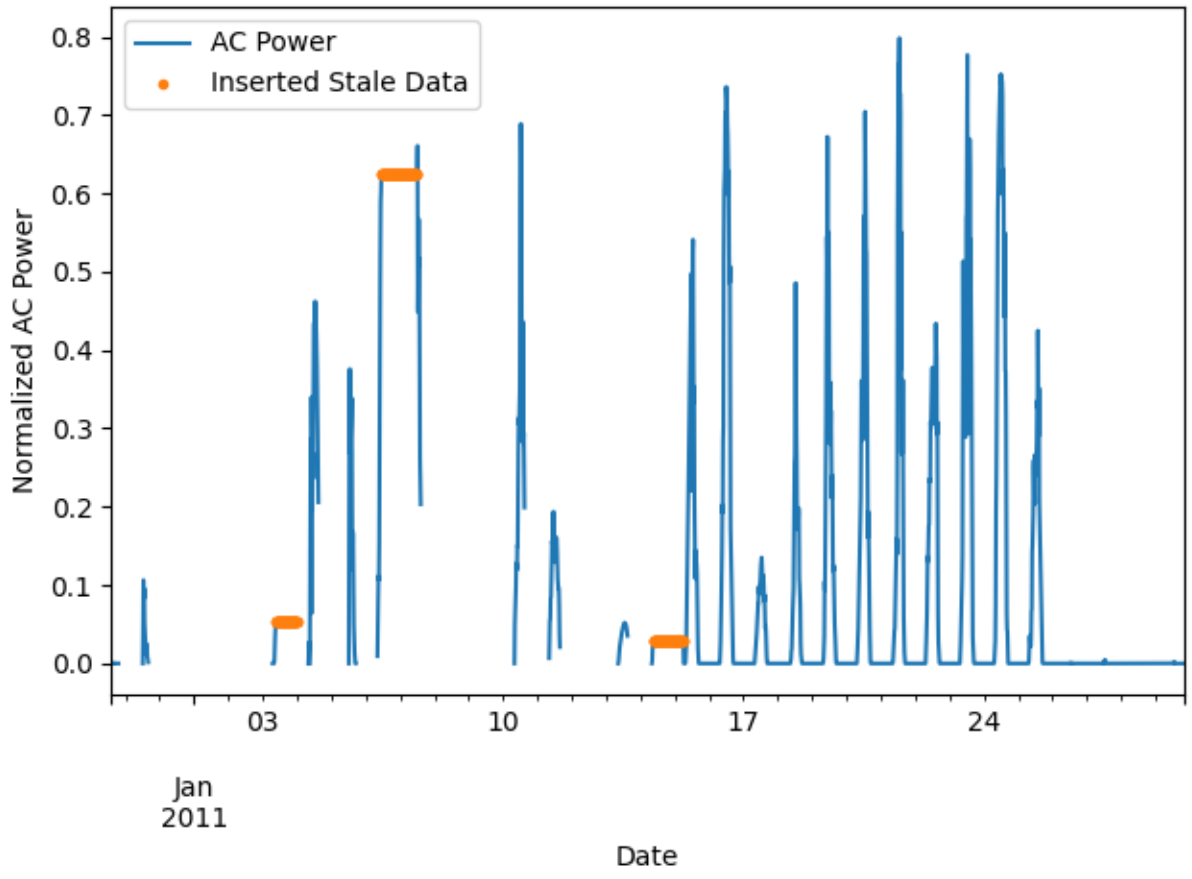
Identifying stale data periods in a time series.

Identifing and removing stale, or consecutive repeating, values in time series data reduces noise when performing data analysis. This example shows how to use two PVAnalytics functions, *pvanalytics.quality.gaps.stale_values_diff()* and *pvanalytics.quality.gaps.stale_values_round()*, to identify and mask stale data periods in time series data.

```
import pvanalytics
from pvanalytics.quality import gaps
import matplotlib.pyplot as plt
import pandas as pd
import pathlib
```

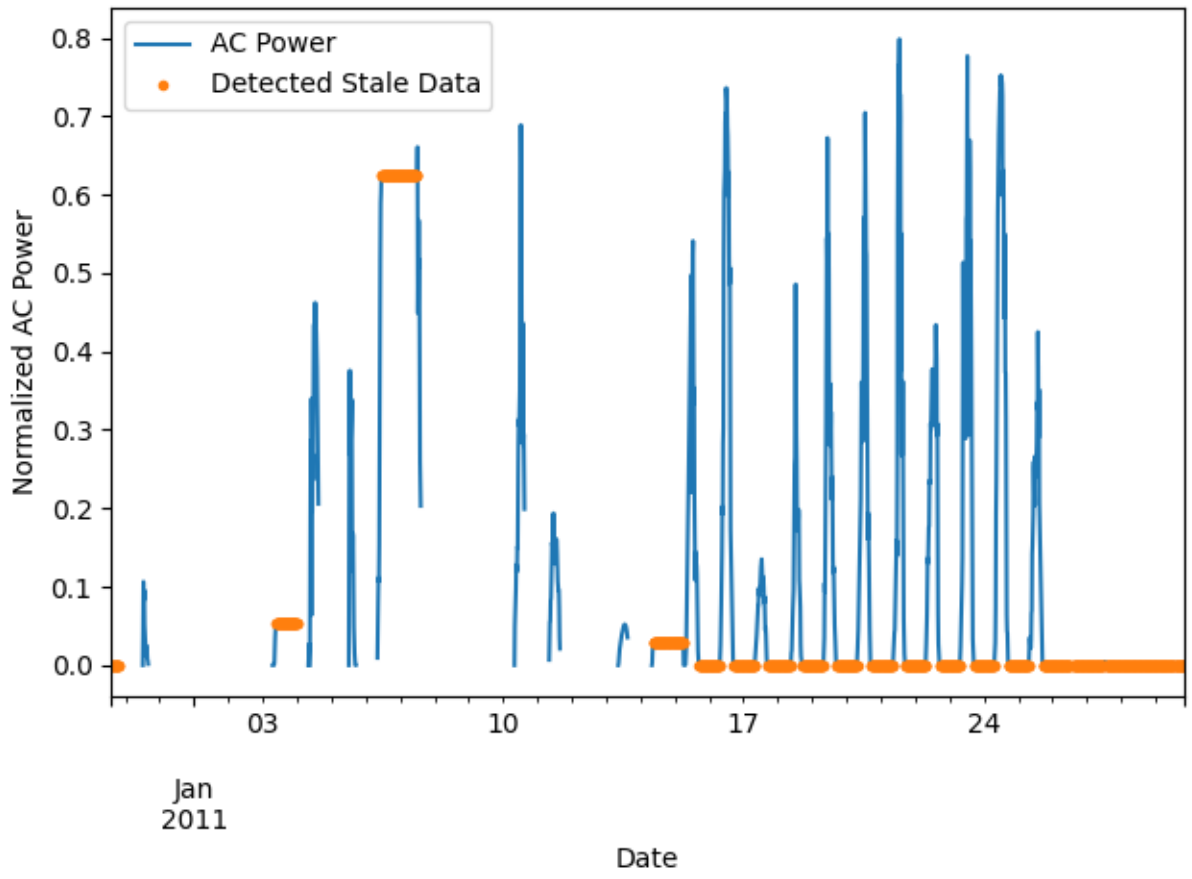
First, we import the AC power data stream that we are going to check for stale data periods. The time series we download is a normalized AC power time series from the PV Fleets Initiative, and is available via the DuraMAT DataHub: https://datahub.duramat.org/dataset/inverter-clipping-ml-training-set-real-data This data set has a Pandas DateTime index, with the min-max normalized AC power time series represented in the 'value_normalized' column. Additionally, there is a "stale_data_mask" column, where stale periods are labeled as True, and all other data is labeled as False. The data is sampled at 15-minute intervals.

```
pvanalytics_dir = pathlib.Path(pvanalytics.__file__).parent
file = pvanalytics_dir / 'data' / 'ac_power_inv_2173_stale_data.csv'
data = pd.read_csv(file, index_col=0, parse_dates=True)
data = data.asfreq("15min")
data['value_normalized'].plot()
data.loc[data["stale_data_mask"], "value_normalized"].plot(ls='', marker='.')
plt.legend(labels=["AC Power", "Inserted Stale Data"])
plt.xlabel("Date")
plt.ylabel("Normalized AC Power")
plt.tight_layout()
plt.show()
```



Now, we use *pvanalytics.quality.gaps.stale_values_diff()* to identify stale values in data. We visualize the detected stale periods graphically. Please note that nighttime periods generally contain consecutive repeating 0 values, which are flagged by *pvanalytics.quality.gaps.stale_values_diff()*.

```
stale_data_mask = gaps.stale_values_diff(data['value_normalized'])
data['value_normalized'].plot()
data.loc[stale_data_mask, "value_normalized"].plot(ls='', marker='.')
plt.legend(labels=["AC Power", "Detected Stale Data"])
plt.xlabel("Date")
plt.ylabel("Normalized AC Power")
plt.tight_layout()
plt.show()
```



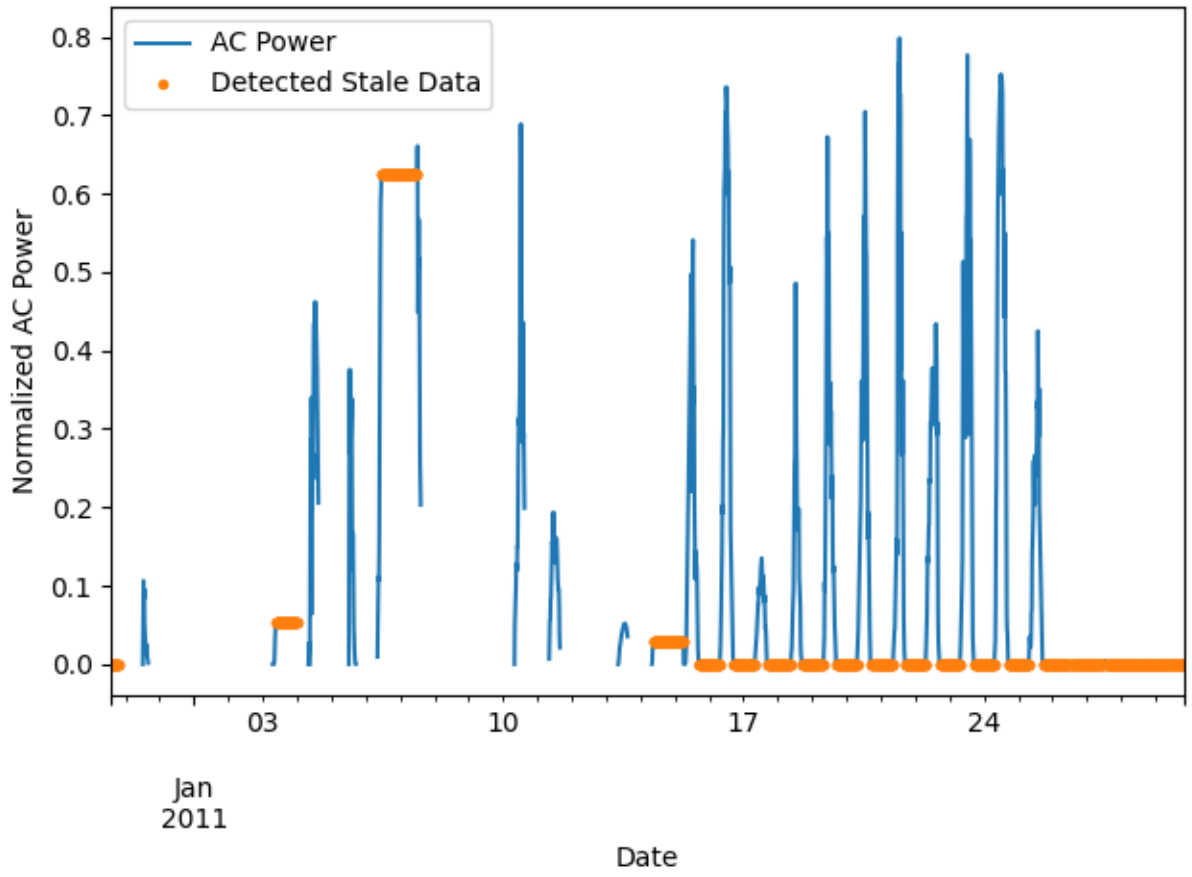
Now, we use *pvanalytics.quality.gaps.stale_values_round()* to identify stale values in data, using rounded data. This function yields similar results as *pvanalytics.quality.gaps.stale_values_diff()*, except it looks for consecutive repeating data that has been rounded to a settable decimals place. Please note that nighttime periods generally contain consecutive repeating 0 values, which are flagged by *pvanalytics.quality.gaps.stale_values_round()*.

```
stale_data_round_mask = gaps.stale_values_round(data['value_normalized'])
data['value_normalized'].plot()
data.loc[stale_data_round_mask, "value_normalized"].plot(ls='', marker='.')
plt.legend(labels=["AC Power", "Detected Stale Data"])
plt.xlabel("Date")
```

(continues on next page)

(continued from previous page)

```
plt.ylabel("Normalized AC Power")
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 1.136 seconds)

Missing Data Periods

Identifying days with missing data using a "completeness" score metric.

Identifying days with missing data and filtering these days out reduces noise when performing data analysis. This example shows how to use a daily data "completeness" score to identify and filter out days with missing data. This includes using *pvanalytics.quality.gaps.completeness_score()*, *pvanalytics.quality.gaps.complete()*, and *pvanalytics.quality.gaps.trim_incomplete()*.

```
import pvanalytics
from pvanalytics.quality import gaps
import matplotlib.pyplot as plt
import pandas as pd
import pathlib
```

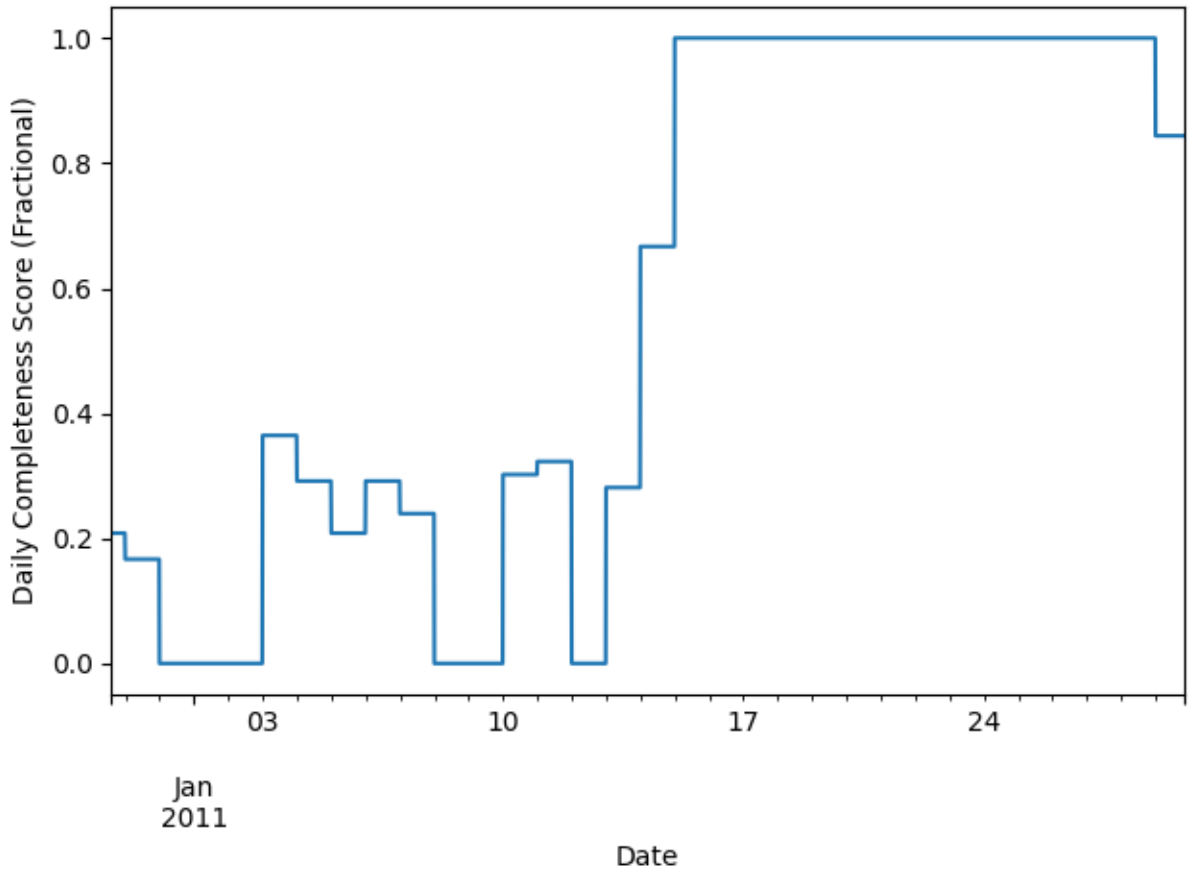
First, we import the AC power data stream that we are going to check for completeness. The time series we download is a normalized AC power time series from the PV Fleets Initiative, and is available via the DuraMAT DataHub:

https://datahub.duramat.org/dataset/inverter-clipping-ml-training-set-real-data. This data set has a Pandas DateTime index, with the min-max normalized AC power time series represented in the 'value_normalized' column. The data is sampled at 15-minute intervals. This data set does contain NaN values.

```
pvanalytics_dir = pathlib.Path(pvanalytics.__file__).parent
file = pvanalytics_dir / 'data' / 'ac_power_inv_2173.csv'
data = pd.read_csv(file, index_col=0, parse_dates=True)
data = data.asfreq("15min")
```

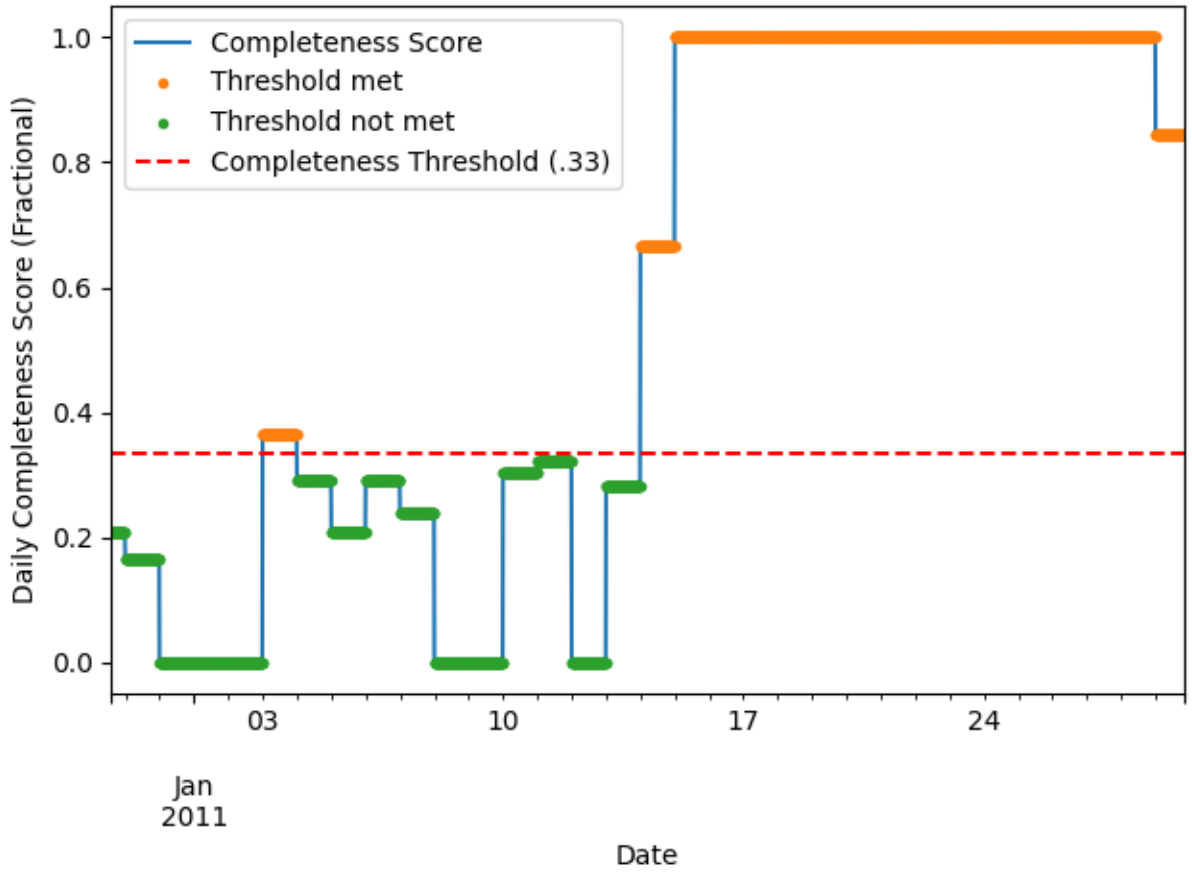
Now, we use *pvanalytics.quality.gaps.completeness_score()* to get the percentage of daily data that isn't NaN. This percentage score is calculated as the total number of non-NA values over a 24-hour period, meaning that nighttime values are expected.

```
data_completeness_score = gaps.completeness_score(data['value_normalized'])

# Visualize data completeness score as a time series.
data_completeness_score.plot()
plt.xlabel("Date")
plt.ylabel("Daily Completeness Score (Fractional)")
plt.tight_layout()
plt.show()
```



We mask complete days, based on daily completeness score, using *pvanalytics.quality.gaps.complete()*.

```
min_completeness = 0.333
daily_completeness_mask = gaps.complete(data['value_normalized'],
                                        minimum_completeness=min_completeness)

# Mask complete days, based on daily completeness score
data_completeness_score.plot()
data_completeness_score.loc[daily_completeness_mask].plot(ls='', marker='.')
data_completeness_score.loc[~daily_completeness_mask].plot(ls='', marker='.')
plt.axhline(y=min_completeness, color='r', linestyle='--')
plt.legend(labels=["Completeness Score", "Threshold met",
                   "Threshold not met", "Completeness Threshold (.33)"],
           loc="upper left")
plt.xlabel("Date")
plt.ylabel("Daily Completeness Score (Fractional)")
plt.tight_layout()
plt.show()
```



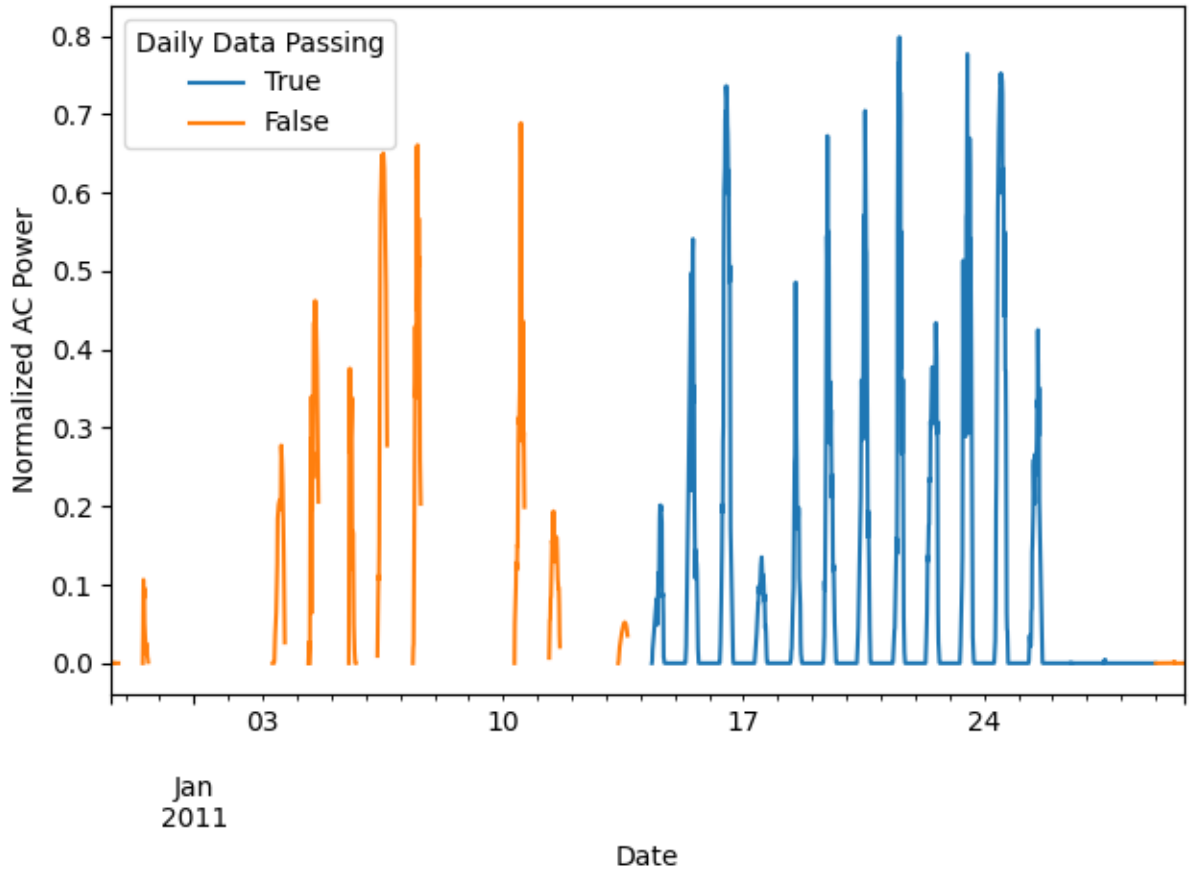
We trim the time series based on the completeness score, where the time series must have at least 10 consecutive days of data that meet the completeness threshold. This is done using *pvanalytics.quality.gaps.trim_incomplete()*.

```
number_consecutive_days = 10
completeness_trim_mask = gaps.trim_incomplete(data['value_normalized'],
                                              days=number_consecutive_days)
```

(continues on next page)

(continued from previous page)

```
# Re-visualize the time series with the data masked by the trim mask
data[completeness_trim_mask]['value_normalized'].plot()
data[~completeness_trim_mask]['value_normalized'].plot()
plt.legend(labels=[True, False],
           title="Daily Data Passing")
plt.xlabel("Date")
plt.ylabel("Normalized AC Power")
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 1.011 seconds)

Irradiance-Quality

This includes examples for running irradiance quality checks on irradiance time series data.

Clearsky Limits for Daily Insolation

Checking the clearsky limits for daily insolation data.

Identifying and filtering out invalid irradiance data is a useful way to reduce noise during analysis. In this example, we use *pvanalytics.quality.irradiance.daily_insolation_limits()* to determine when the daily insolation lies between a minimum and a maximum value. Irradiance measurements and clear-sky irradiance on each day are integrated with the trapezoid rule to calculate daily insolation. For this example we will use data from the RMIS weather system located on the NREL campus in Colorado, USA.

```
import pvanalytics
from pvanalytics.quality.irradiance import daily_insolation_limits
import pvlib
import matplotlib.pyplot as plt
import pandas as pd
import pathlib
```

First, read in data from the RMIS NREL system. This data set contains 5-minute right-aligned data. It includes POA, GHI, DNI, DHI, and GNI measurements.

```
pvanalytics_dir = pathlib.Path(pvanalytics.__file__).parent
rmis_file = pvanalytics_dir / 'data' / 'irradiance_RMIS_NREL.csv'
data = pd.read_csv(rmis_file, index_col=0, parse_dates=True)
# Make the datetime index tz-aware.
data.index = data.index.tz_localize("Etc/GMT+7")
```

Now model clear-sky irradiance for the location and times of the measured data:

```
location = pvlib.location.Location(39.7407, -105.1686)
clearsky = location.get_clearsky(data.index)
```

Use *pvanalytics.quality.irradiance.daily_insolation_limits()* to identify if the daily insolation lies between a minimum and a maximum value. Here, we check GHI irradiance field 'irradiance_ghi__7981'. *pvanalytics.quality.irradiance.daily_insolation_limits()* returns a mask that identifies data that falls between lower and upper limits. The defaults (used here) are upper bound of 125% of clear-sky daily insolation, and lower bound of 40% of clear-sky daily insolation.

```
daily_insolation_mask = daily_insolation_limits(data['irradiance_ghi__7981'],
                                                clearsky['ghi'])
```

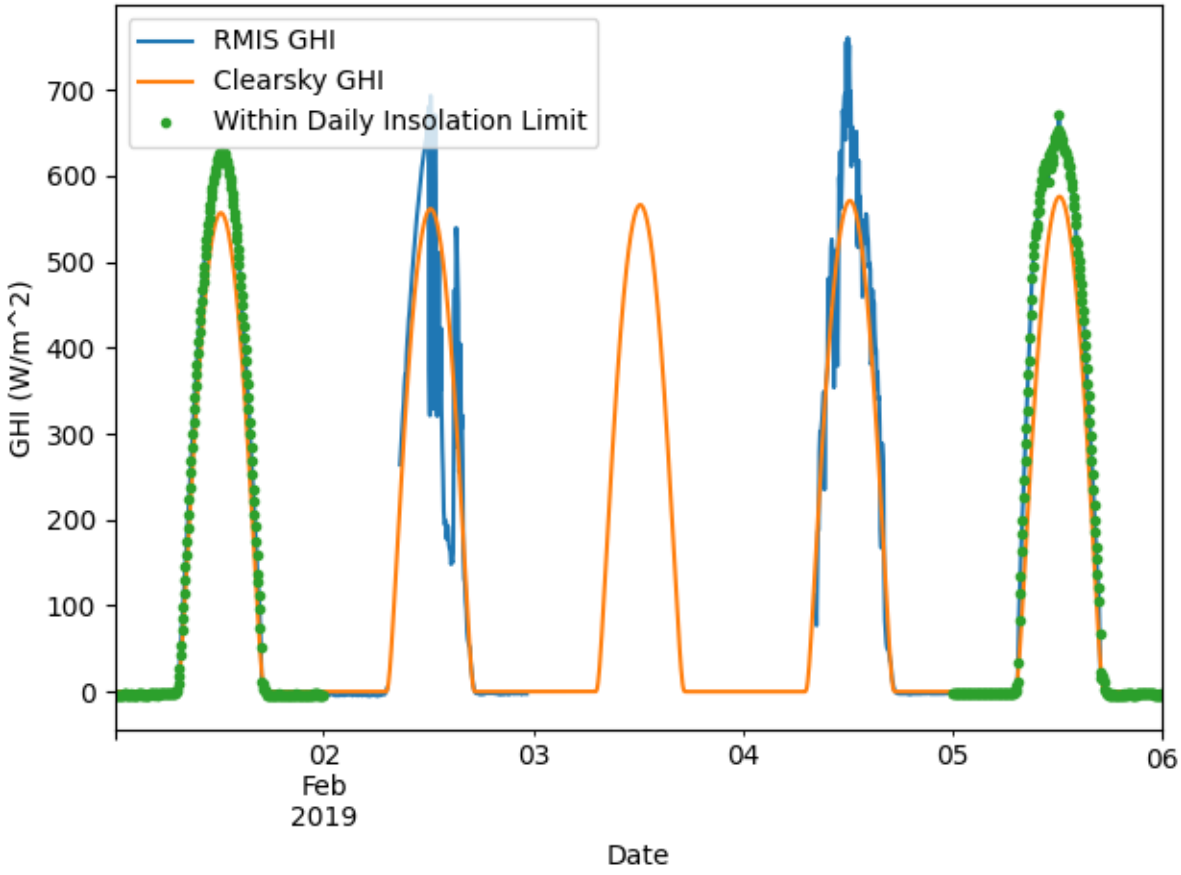
Plot the 'irradiance_ghi__7981' data stream and its associated clearsky GHI data stream. Mask the GHI time series by its daily_insolation_mask.

```
data['irradiance_ghi__7981'].plot()
clearsky['ghi'].plot()
data.loc[daily_insolation_mask, 'irradiance_ghi__7981'].plot(ls='', marker='.')
plt.legend(labels=["RMIS GHI", "Clearsky GHI",
                   "Within Daily Insolation Limit"],
           loc="upper left")
plt.xlabel("Date")
```

(continues on next page)

(continued from previous page)

```
plt.ylabel("GHI (W/m^2)")
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.400 seconds)

Clearsky Limits for Irradiance Data

Checking the clearsky limits of irradiance data.

Identifying and filtering out invalid irradiance data is a useful way to reduce noise during analysis. In this example, we use *pvanalytics.quality.irradiance.clearsky_limits()* to identify irradiance values that do not exceed a limit based on a clear-sky model. For this example we will use GHI data from the RMIS weather system located on the NREL campus in CO.

```
import pvanalytics
from pvanalytics.quality.irradiance import clearsky_limits
from pvanalytics.features.daytime import power_or_irradiance
import pvlib
import matplotlib.pyplot as plt
import pandas as pd
import pathlib
```

First, read in data from the RMIS NREL system. This data set contains 5-minute right-aligned POA, GHI, DNI, DHI, and GNI measurements, but only the GHI is relevant here.

```
pvanalytics_dir = pathlib.Path(pvanalytics.__file__).parent
rmis_file = pvanalytics_dir / 'data' / 'irradiance_RMIS_NREL.csv'
data = pd.read_csv(rmis_file, index_col=0, parse_dates=True)
freq = '5min'
# Make the datetime index tz-aware.
data.index = data.index.tz_localize("Etc/GMT+7")
```

Now model clear-sky irradiance for the location and times of the measured data. You can do this using `pvlib.location.Location.get_clearsky()`, using the lat-long coordinates associated the RMIS NREL system.

```
location = pvlib.location.Location(39.7407, -105.1686)
clearsky = location.get_clearsky(data.index)
```

Use *pvanalytics.quality.irradiance.clearsky_limits()*. Here, we check GHI data in field 'irradiance_ghi__7981'. *pvanalytics.quality.irradiance.clearsky_limits()* returns a mask that identifies data that falls between lower and upper limits. The defaults (used here) are upper bound of 110% of clear-sky GHI, and no lower bound.

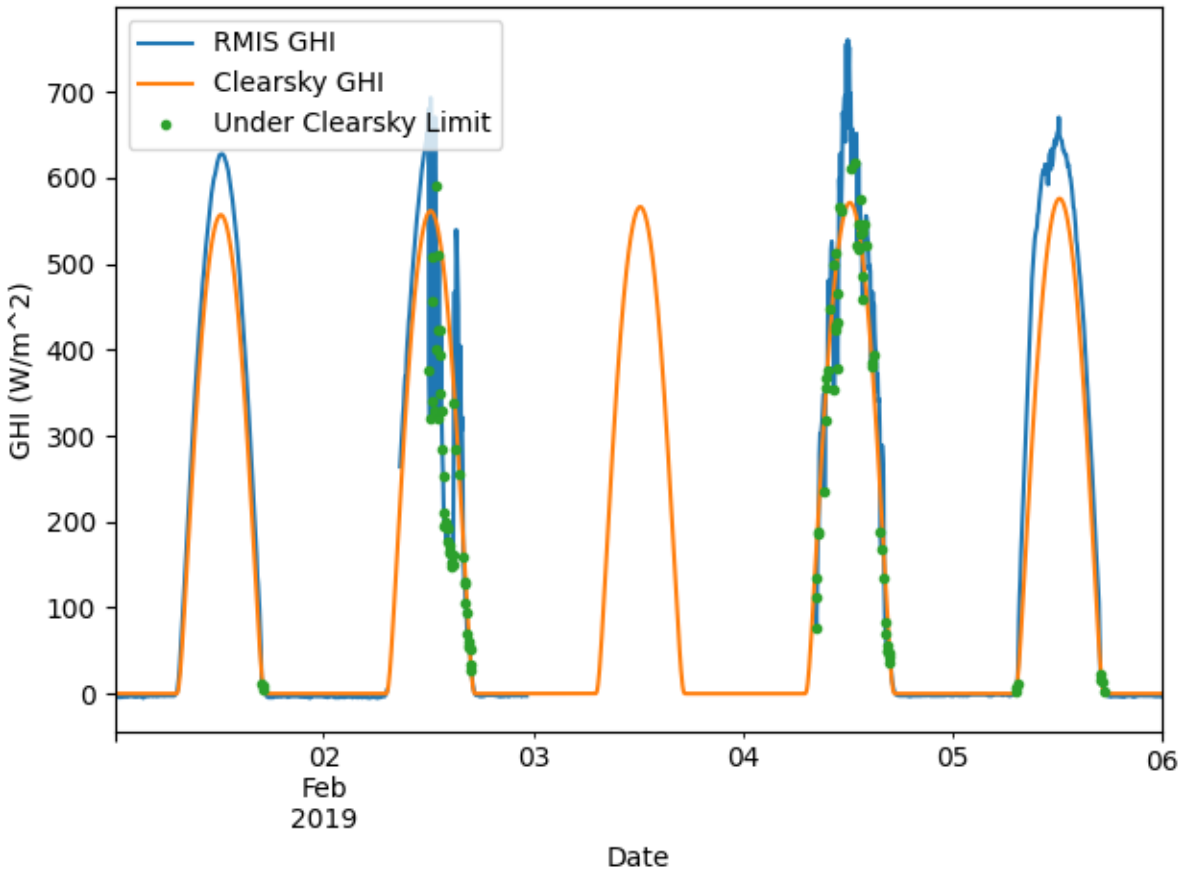
```
clearsky_limit_mask = clearsky_limits(data['irradiance_ghi__7981'],
                                      clearsky['ghi'])
```

Mask nighttime values in the GHI time series using the *pvanalytics.features.daytime.power_or_irradiance()* function. We will then remove nighttime values from the GHI time series.

```
day_night_mask = power_or_irradiance(series=data['irradiance_ghi__7981'],
                                     freq=freq)
```

Plot the 'irradiance_ghi__7981' data stream and its associated clearsky GHI data stream. Mask the GHI time series by its clearsky_limit_mask for daytime periods. Please note that a simple Ineichen model with static monthly turbidities isn't always accurate, as in this case. Other models that may provide better clear-sky estimates include McClear or PSM3.

```
data['irradiance_ghi__7981'].plot()
clearsky['ghi'].plot()
data.loc[clearsky_limit_mask & day_night_mask][
    'irradiance_ghi__7981'].plot(ls='', marker='.')
plt.legend(labels=["RMIS GHI", "Clearsky GHI",
                   "Under Clearsky Limit"],
           loc="upper left")
plt.xlabel("Date")
plt.ylabel("GHI (W/m^2)")
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.483 seconds)

QCrad Limits for Irradiance Data

Test for physical limits on GHI, DHI or DNI using the QCRad criteria.

Identifying and filtering out invalid irradiance data is a useful way to reduce noise during analysis. In this example, we use *pvanalytics.quality.irradiance.check_irradiance_limits_qcrad()* to test for physical limits on GHI, DHI or DNI using the QCRad criteria. For this example we will use data from the RMIS weather system located on the NREL campus in Colorado, USA.

```
import pvanalytics
from pvanalytics.quality.irradiance import check_irradiance_limits_qcrad
import pvlib
import matplotlib.pyplot as plt
import pandas as pd
import pathlib
```

First, read in data from the RMIS NREL system. This data set contains 5-minute right-aligned data. It includes POA, GHI, DNI, DHI, and GNI measurements.

```
pvanalytics_dir = pathlib.Path(pvanalytics.__file__).parent
rmis_file = pvanalytics_dir / 'data' / 'irradiance_RMIS_NREL.csv'
data = pd.read_csv(rmis_file, index_col=0, parse_dates=True)
```

Now generate solar zenith estimates for the location, based on the data's time zone and site latitude-longitude coordinates. This is done using the pvlib.solarposition.get_solarposition() function.

```
latitude = 39.742
longitude = -105.18
time_zone = "Etc/GMT+7"
data = data.tz_localize(time_zone)
solar_position = pvlib.solarposition.get_solarposition(data.index,
                                                       latitude,
                                                       longitude)
```

Generate the estimated extraterrestrial radiation for the time series, referred to as dni_extra. This is done using the pvlib.irradiance.get_extra_radiation() function.

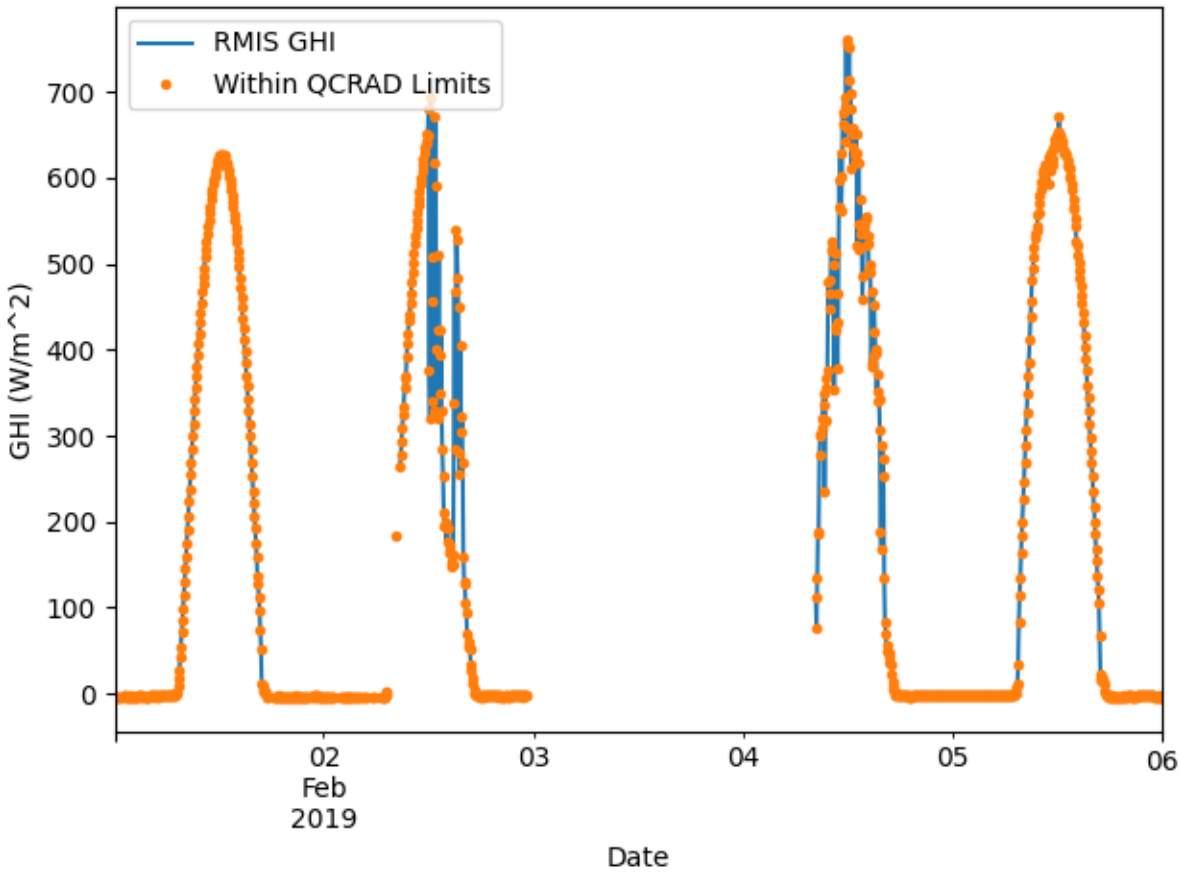
```
dni_extra = pvlib.irradiance.get_extra_radiation(data.index)
```

Use *pvanalytics.quality.irradiance.check_irradiance_limits_qcrad()* to generate the QCRAD irradiance limit mask

```
qcrad_limit_mask = check_irradiance_limits_qcrad(
    solar_zenith=solar_position['zenith'],
    dni_extra=dni_extra,
    ghi=data['irradiance_ghi__7981'],
    dhi=data['irradiance_dhi__7983'],
    dni=data['irradiance_dni__7982'])
```

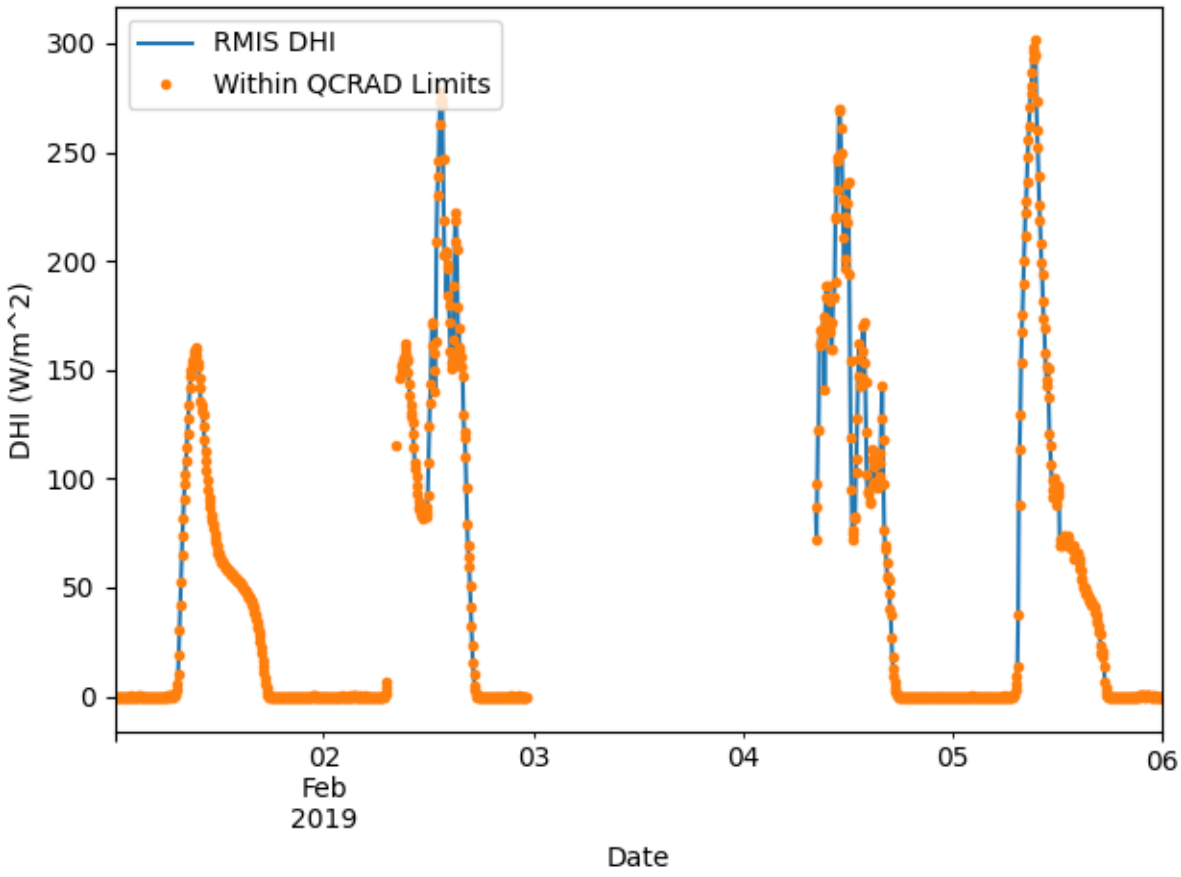
Plot the 'irradiance_ghi__7981' data stream with its associated QCRAD limit mask.

```
data['irradiance_ghi__7981'].plot()
data.loc[qcrad_limit_mask[0], 'irradiance_ghi__7981'].plot(ls='', marker='.')
plt.legend(labels=["RMIS GHI", "Within QCRAD Limits"],
           loc="upper left")
plt.xlabel("Date")
plt.ylabel("GHI (W/m^2)")
plt.tight_layout()
plt.show()
```



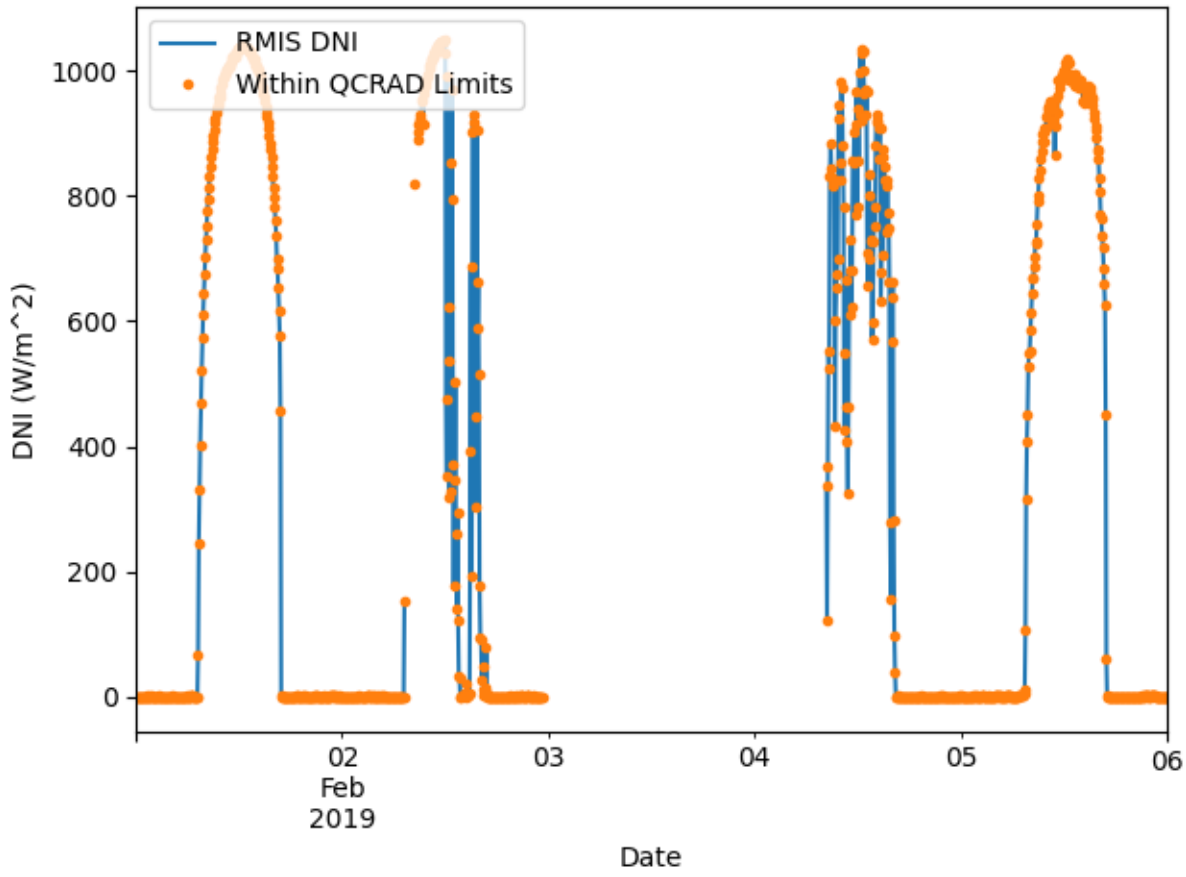
Plot the 'irradiance_dhi__7983 data stream with its associated QCRAD limit mask.

```
data['irradiance_dhi__7983'].plot()
data.loc[qcrad_limit_mask[1], 'irradiance_dhi__7983'].plot(ls='', marker='.')
plt.legend(labels=["RMIS DHI", "Within QCRAD Limits"],
           loc="upper left")
plt.xlabel("Date")
plt.ylabel("DHI (W/m^2)")
plt.tight_layout()
plt.show()
```



Plot the 'irradiance_dni__7982' data stream with its associated QCRAD limit mask.

```
data['irradiance_dni__7982'].plot()
data.loc[qcrad_limit_mask[2], 'irradiance_dni__7982'].plot(ls='', marker='.')
plt.legend(labels=["RMIS DNI", "Within QCRAD Limits"],
           loc="upper left")
plt.xlabel("Date")
plt.ylabel("DNI (W/m^2)")
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.741 seconds)

QCrad Consistency for Irradiance Data

Check consistency of GHI, DHI and DNI using QCRad criteria.

Identifying and filtering out invalid irradiance data is a useful way to reduce noise during analysis. In this example, we use *pvanalytics.quality.irradiance.check_irradiance_consistency_qcrad()* to check the consistency of GHI, DHI and DNI data using QCRad criteria. For this example we will use data from the RMIS weather system located on the NREL campus in Colorado, USA.

```
import pvanalytics
from pvanalytics.quality.irradiance import check_irradiance_consistency_qcrad
import pvlib
import matplotlib.pyplot as plt
import pandas as pd
import pathlib
```

First, read in data from the RMIS NREL system. This data set contains 5-minute right-aligned data. It includes POA, GHI, DNI, DHI, and GNI measurements.

```
pvanalytics_dir = pathlib.Path(pvanalytics.__file__).parent
rmis_file = pvanalytics_dir / 'data' / 'irradiance_RMIS_NREL.csv'
data = pd.read_csv(rmis_file, index_col=0, parse_dates=True)
```

Now generate solar zenith estimates for the location, based on the data's time zone and site latitude-longitude coordinates.

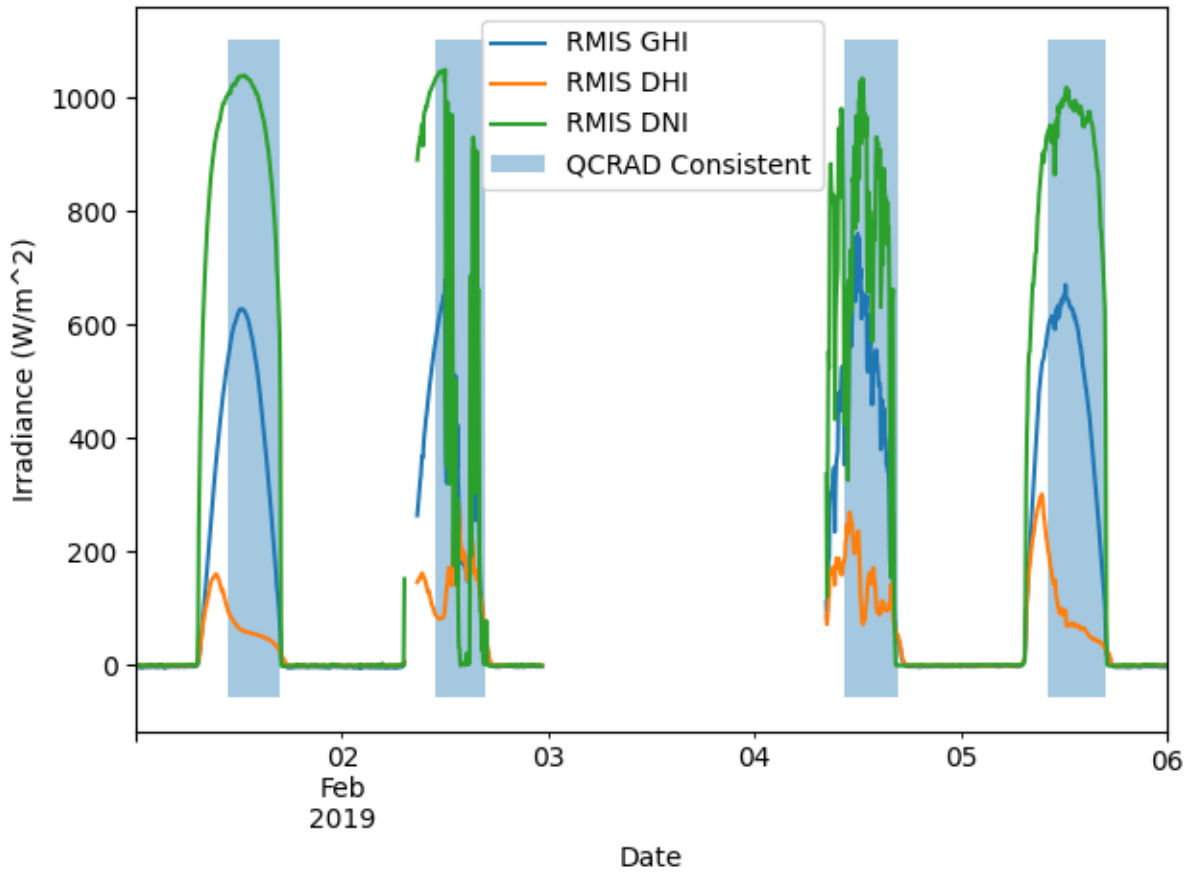
```
latitude = 39.742
longitude = -105.18
time_zone = "Etc/GMT+7"
data = data.tz_localize(time_zone)
solar_position = pvlib.solarposition.get_solarposition(data.index,
                                                       latitude,
                                                       longitude)
```

Use *pvanalytics.quality.irradiance.check_irradiance_consistency_qcrad()* to generate the QCRAD consistency mask.

```
qcrad_consistency_mask = check_irradiance_consistency_qcrad(
    solar_zenith=solar_position['zenith'],
    ghi=data['irradiance_ghi__7981'],
    dhi=data['irradiance_dhi__7983'],
    dni=data['irradiance_dni__7982'])
```

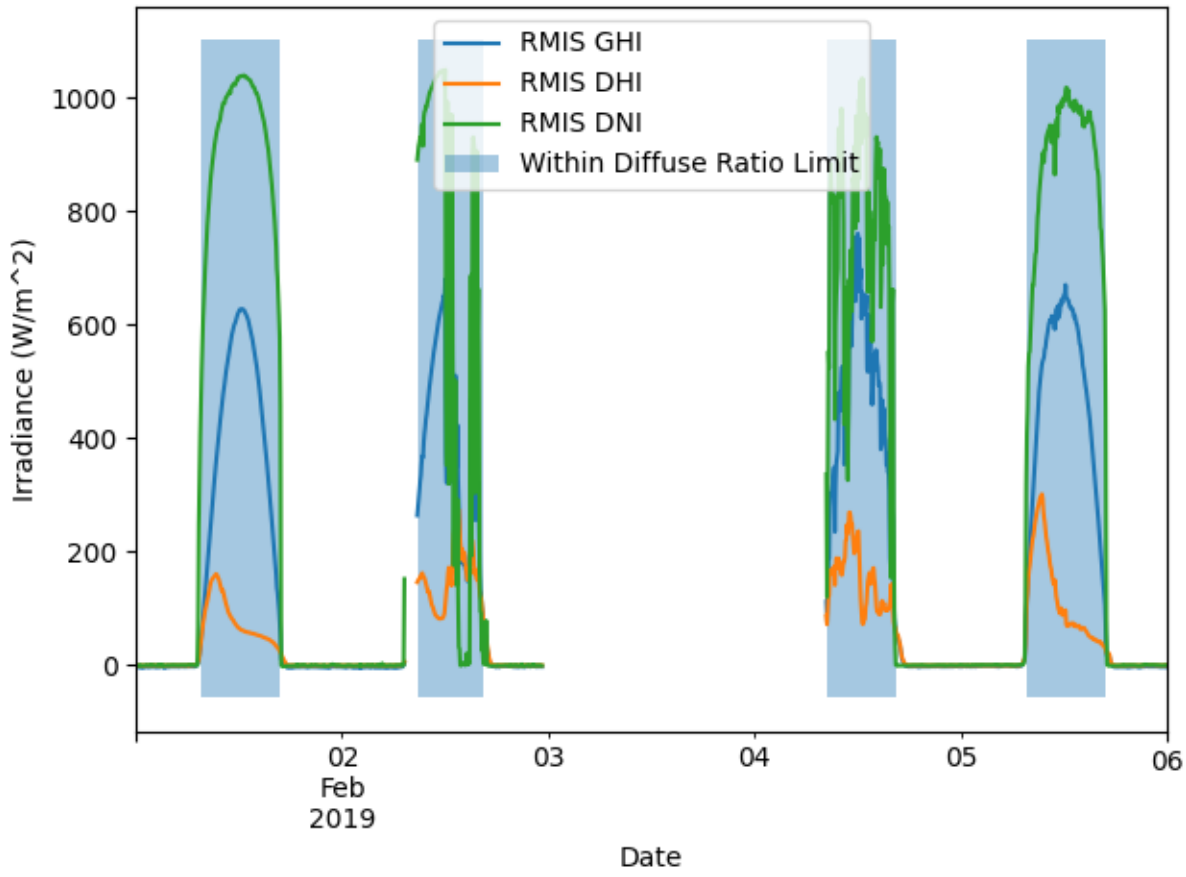
Plot the GHI, DHI, and DNI data streams with the QCRAD consistency mask overlay. This mask applies to all 3 data streams.

```
fig = data[['irradiance_ghi__7981', 'irradiance_dhi__7983',
            'irradiance_dni__7982']].plot()
# Highlight periods where the QCRAD consistency mask is True
fig.fill_between(data.index, fig.get_ylim()[0], fig.get_ylim()[1],
                 where=qcrad_consistency_mask[0], alpha=0.4)
fig.legend(labels=["RMIS GHI", "RMIS DHI", "RMIS DNI", "QCRAD Consistent"],
           loc="upper center")
plt.xlabel("Date")
plt.ylabel("Irradiance (W/m^2)")
plt.tight_layout()
plt.show()
```



Plot the GHI, DHI, and DNI data streams with the diffuse ratio limit mask overlay. This mask is true when the DHI / GHI ratio passes the limit test.

```
fig = data[['irradiance_ghi__7981', 'irradiance_dhi__7983',
            'irradiance_dni__7982']].plot()
# Highlight periods where the GHI ratio passes the limit test
fig.fill_between(data.index, fig.get_ylim()[0], fig.get_ylim()[1],
                 where=qcrad_consistency_mask[1], alpha=0.4)
fig.legend(labels=["RMIS GHI", "RMIS DHI", "RMIS DNI",
                   "Within Diffuse Ratio Limit"],
           loc="upper center")
plt.xlabel("Date")
plt.ylabel("Irradiance (W/m^2)")
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.778 seconds)

Component Sum Equations for Irradiance Data

Estimate GHI, DHI, and DNI using the component sum equations, with nighttime corrections.

Estimating GHI, DHI, and DNI using the component sum equations is useful if the associated field is missing, or as a comparison to an existing physical data stream.

```
import pvanalytics
from pvanalytics.quality.irradiance import calculate_component_sum_series
import pvlib
import matplotlib.pyplot as plt
import pandas as pd
import pathlib
```

First, read in data from the RMIS NREL system. This data set contains 5-minute right-aligned data. It includes POA, GHI, DNI, DHI, and GNI measurements.

```
pvanalytics_dir = pathlib.Path(pvanalytics.__file__).parent
rmis_file = pvanalytics_dir / 'data' / 'irradiance_RMIS_NREL.csv'
data = pd.read_csv(rmis_file, index_col=0, parse_dates=True)
```

Now generate solar zenith estimates for the location, based on the data's time zone and site latitude-longitude coordinates. This is done using the pvlib.solarposition.get_solarposition() function.

```
latitude = 39.742
longitude = -105.18
time_zone = "Etc/GMT+7"
data = data.tz_localize(time_zone)
solar_position = pvlib.solarposition.get_solarposition(data.index,
                                                       latitude,
                                                       longitude)
```

Get the clearsky DNI values associated with the current location, using the pvlib.solarposition.get_solarposition() function. These clearsky values are used to calculate DNI data.

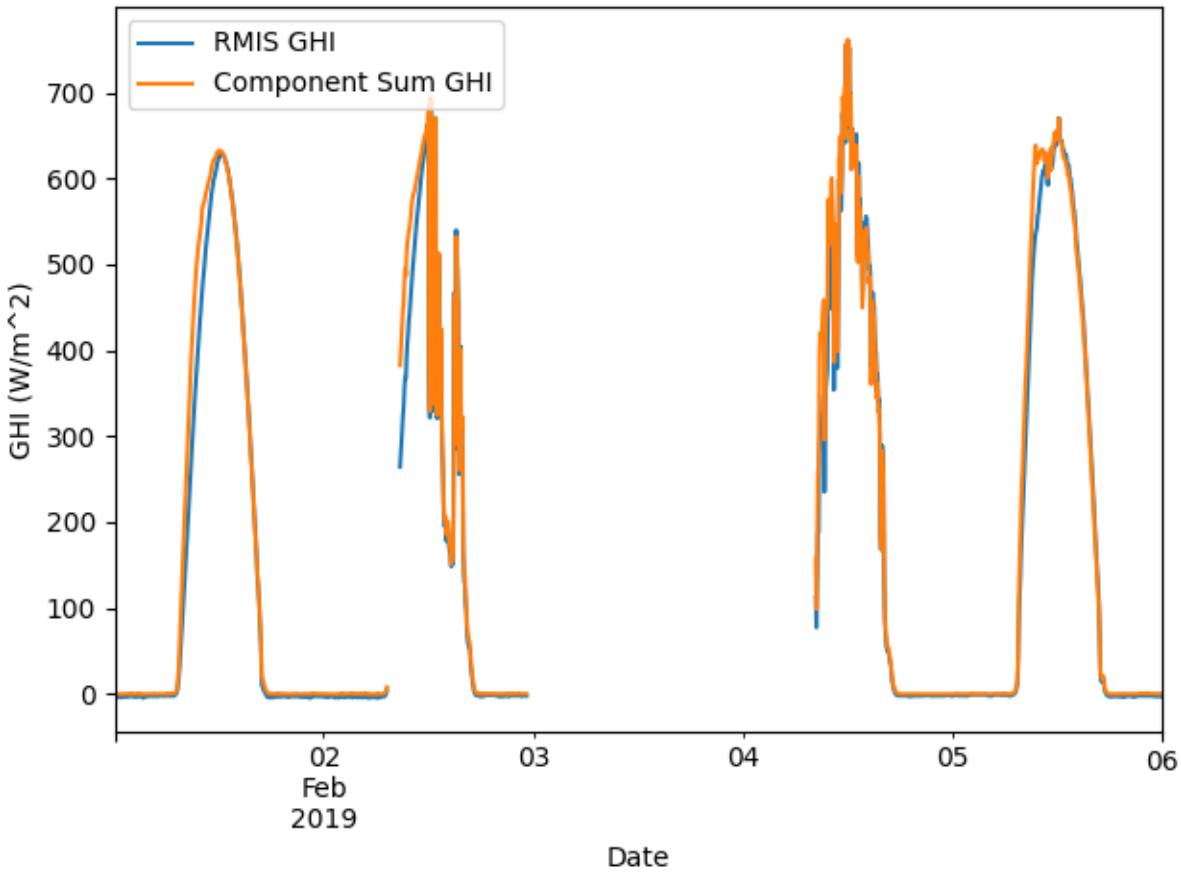
```
site = pvlib.location.Location(latitude, longitude, tz=time_zone)
clearsky = site.get_clearsky(data.index)
```

Use pvanalytics.quality.irradiance.calcuate_ghi_component() to estimate GHI measurements using DHI and DNI measurements

```
component_sum_ghi = calculate_component_sum_series(
    solar_zenith=solar_position['zenith'],
    dhi=data['irradiance_dhi__7983'],
    dni=data['irradiance_dni__7982'],
    zenith_limit=90,
    fill_night_value='equation')
```

Plot the 'irradiance_ghi__7981' data stream against the estimated component sum GHI, for comparison

```
data['irradiance_ghi__7981'].plot()
component_sum_ghi.plot()
plt.legend(labels=["RMIS GHI", "Component Sum GHI"],
           loc="upper left")
plt.xlabel("Date")
plt.ylabel("GHI (W/m^2)")
plt.tight_layout()
plt.show()
```

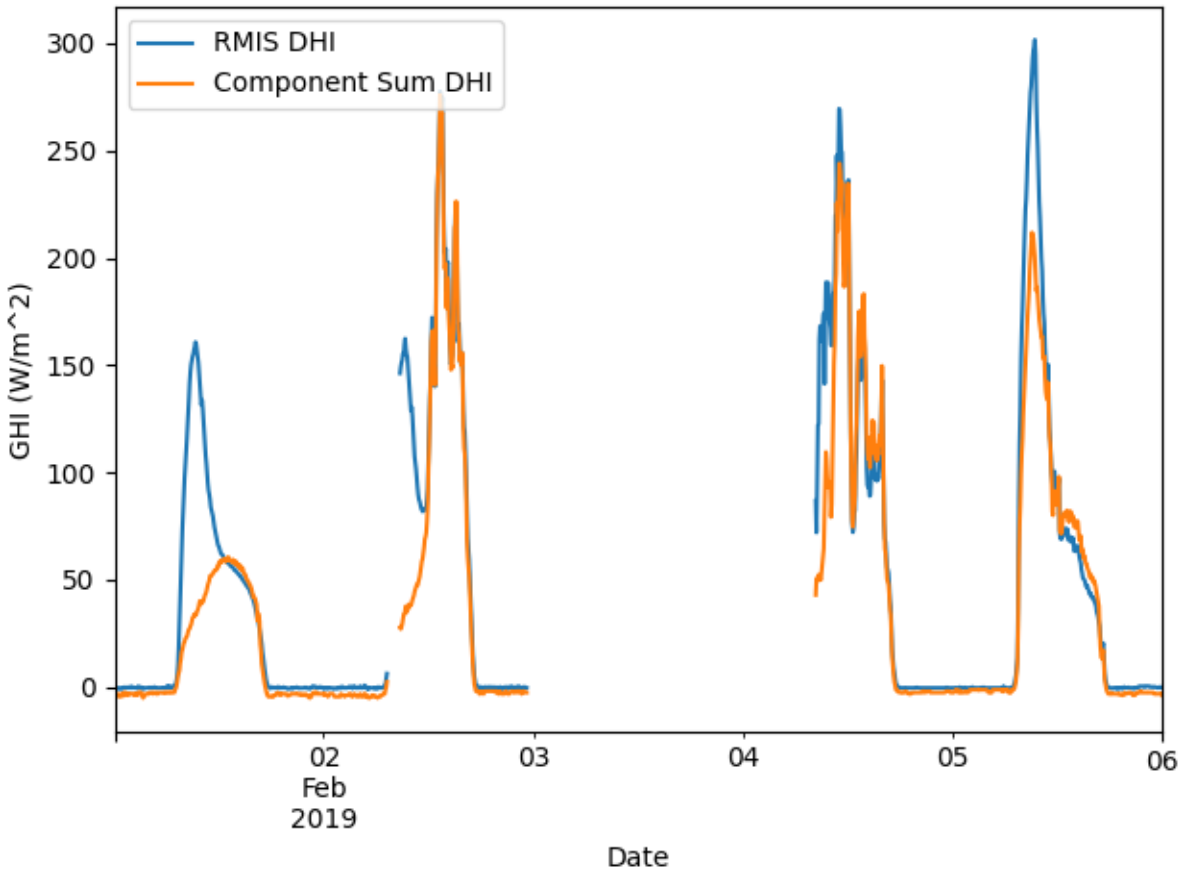


Use `pvanalytics.quality.irradiance.calcuate_dhi_component()` to estimate DHI measurements using GHI and DNI measurements

```
component_sum_dhi = calculate_component_sum_series(
    solar_zenith=solar_position['zenith'],
    dni=data['irradiance_dni__7982'],
    ghi=data['irradiance_ghi__7981'],
    zenith_limit=90,
    fill_night_value='equation')
```

Plot the 'irradiance_dhi__7983' data stream against the estimated component sum GHI, for comparison

```
data['irradiance_dhi__7983'].plot()
component_sum_dhi.plot()
plt.legend(labels=["RMIS DHI", "Component Sum DHI"],
           loc="upper left")
plt.xlabel("Date")
plt.ylabel("GHI (W/m^2)")
plt.tight_layout()
plt.show()
```

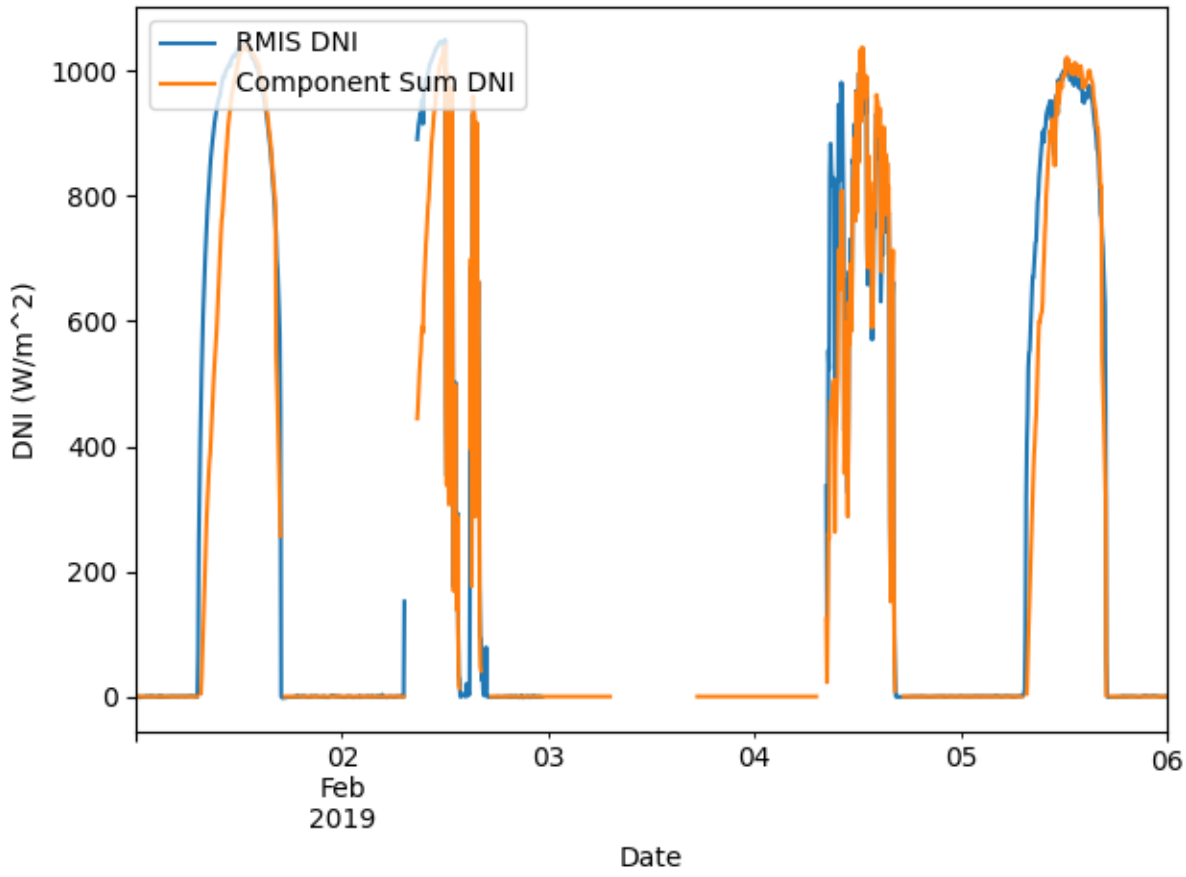


Use `pvanalytics.quality.irradiance.calcuate_dni_component()` to estimate DNI measurements using GHI and DHI measurements

```
component_sum_dni = calculate_component_sum_series(
    solar_zenith=solar_position['zenith'],
    dhi=data['irradiance_dhi__7983'],
    ghi=data['irradiance_ghi__7981'],
    dni_clear=clearsky['dni'],
    zenith_limit=90,
    fill_night_value='equation')
```

Plot the 'irradiance_dni__7982' data stream against the estimated component sum GHI, for comparison

```
data['irradiance_dni__7982'].plot()
component_sum_dni.plot()
plt.legend(labels=["RMIS DNI", "Component Sum DNI"],
           loc="upper left")
plt.xlabel("Date")
plt.ylabel("DNI (W/m^2)")
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.778 seconds)

Metrics

This includes examples for quantifying system time series metrics, including variability index (VI) and NREL performance ratio (PR).

Calculate Variability Index

Calculate the Variability Index for a GHI time series.

Highly variable irradiance can cause mismatch between irradiance and power measurements and result in noisy performance metrics. As such, identifying and removing highly variable conditions is useful in certain analyses. Identification and quantification of highly variable conditions are also of interest in grid integration and hourly modeling error contexts. The variability index (VI) is one way of quantifying the variability or jaggedness of an irradiance signal relative to a corresponding reference clear-sky irradiance profile. Note that quantifying variability is related to but distinct from clear-sky detection. For example, both clear and overcast skies have low VI. This example uses GHI data collected from the NREL RMIS system to calculate the variability index as a time series.

```
import pvanalytics
from pvanalytics.metrics import variability_index
import matplotlib.pyplot as plt
import pandas as pd
```

(continues on next page)

(continued from previous page)

```
import pathlib
import pvlib
```

First, read in data from the RMIS NREL system. This data set contains 5-minute right-aligned POA, GHI, DNI, DHI, and GNI measurements, but only the GHI is relevant here.

```
pvanalytics_dir = pathlib.Path(pvanalytics.__file__).parent
rmis_file = pvanalytics_dir / 'data' / 'irradiance_RMIS_NREL.csv'
data = pd.read_csv(rmis_file, index_col=0, parse_dates=True)
# Make the datetime index tz-aware.
data.index = data.index.tz_localize("Etc/GMT+7")
```

Now model clear-sky irradiance for the location and times of the measured data. You can do this using `pvlib.location.Location.get_clearsky()`, using the lat-long coordinates associated the RMIS NREL system.

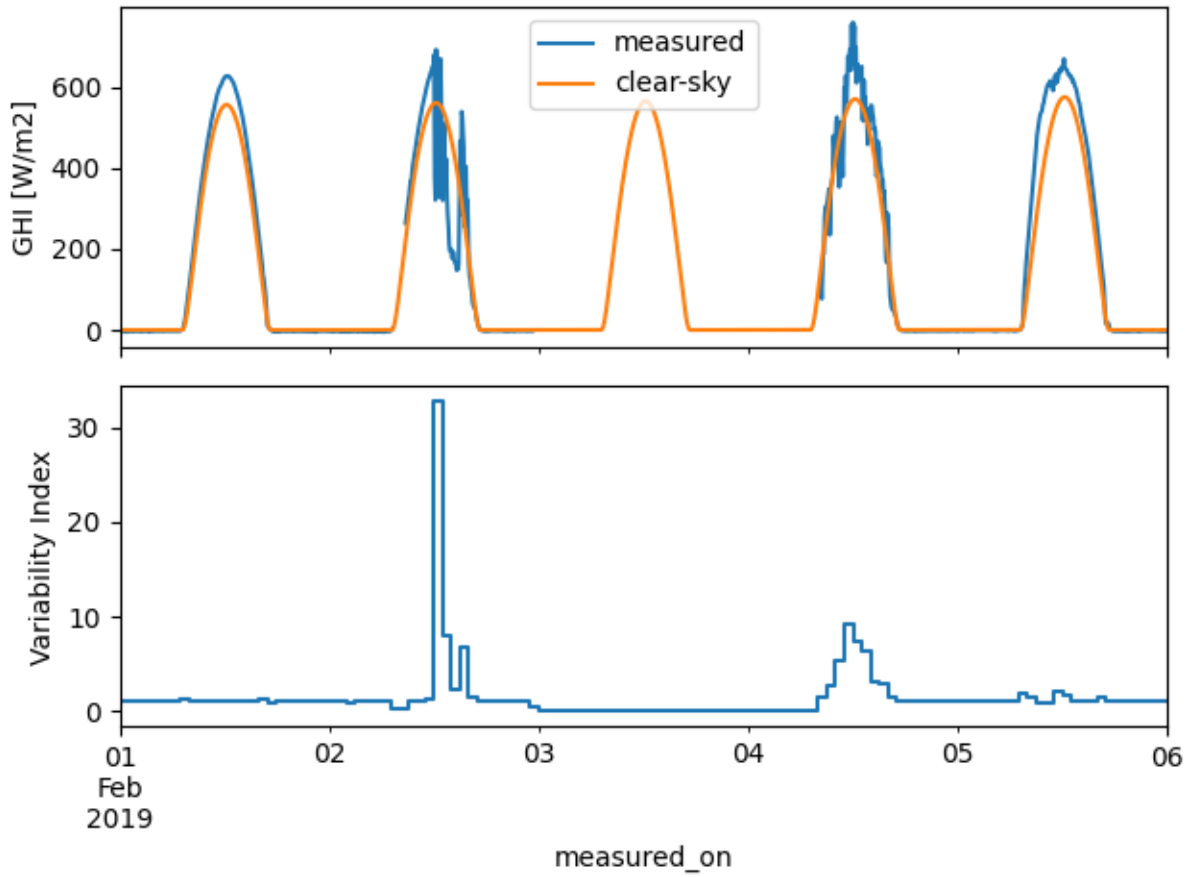
```
location = pvlib.location.Location(39.7407, -105.1686)
clearsky = location.get_clearsky(data.index)
```

Calculate the variability index for the system GHI data stream using the *pvanalytics.metrics.variability_index()* function, using an hourly frequency.

```
variability_index_series = variability_index(data['irradiance_ghi__7981'],
                                             clearsky['ghi'],
                                             freq='1h')
```

Plot the calculated VI against the underlying GHI measurements, for the purpose of comparison.

```
fig, axes = plt.subplots(2, 1, sharex=True)
data['irradiance_ghi__7981'].plot(ax=axes[0], label='measured')
clearsky['ghi'].plot(ax=axes[0], label='clear-sky')
variability_index_series.plot(ax=axes[1], drawstyle='steps-post')
axes[0].legend()
axes[0].set_ylabel("GHI [W/m2]")
axes[1].set_ylabel("Variability Index")
fig.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.442 seconds)

Calculate Performance Ratio (NREL)

Calculate the NREL Performance Ratio for a system.

When evaluating PV system performance it is often desirable to distinguish uncontrollable effects like weather variation from controllable effects like soiling and hardware issues. The NREL Performance Ratio (or "Weather-Corrected Performance Ratio") is a unitless metric that normalizes system output for variation in irradiance and temperature, making it insensitive to uncontrollable weather variation and more reflective of system health. In this example, we show how to calculate the NREL PR at two different frequencies: for a complete time series, and at daily intervals. We use the *pvanalytics.metrics.performance_ratio_nrel()* function.

```
import pvanalytics
from pvanalytics.metrics import performance_ratio_nrel
import pandas as pd
import pathlib
import matplotlib.pyplot as plt
```

First, we read in data from the NREL RSF II system. This data set contains 15-minute interval data for AC power, POA irradiance, ambient temperature, and wind speed, among others. The complete data set for the NREL RSF II installation is available in the PVDAQ database, under system ID 1283.

```
pvanalytics_dir = pathlib.Path(pvanalytics.__file__).parent
file = pvanalytics_dir / 'data' / 'nrel_RSF_II.csv'
data = pd.read_csv(file, index_col=0, parse_dates=True)
```

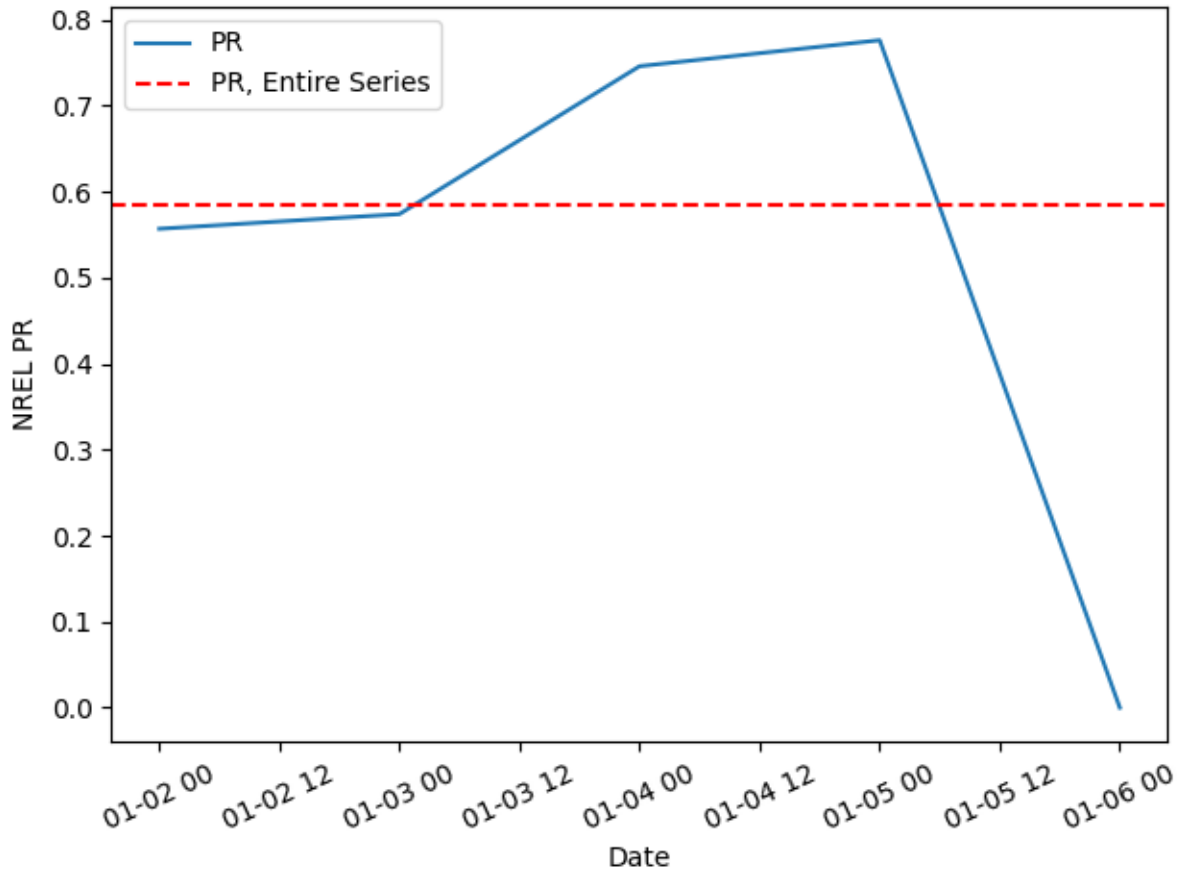
Now we calculate the PR for the entire time series, using the POA, ambient temperature, wind speed, and AC power fields. We use this data as parameters in the *pvanalytics.metrics.performance_ratio_nrel()* function. In this example we are calculating PR for a single inverter connected to a 204.12 kW PV array.

```
pr_whole_series = performance_ratio_nrel(data['poa_irradiance__1055'],
                                         data['ambient_temp__1053'],
                                         data['wind_speed__1051'],
                                         data['inv2_ac_power_w__1047']/1000,
                                         204.12)

print("RSF II, PR for the whole time series:")
print(pr_whole_series)
```

```
RSF II, PR for the whole time series:
0.5851958594021633
```

Next, we recalculate the PR on a daily basis. We separate the time series into daily intervals, and calculate the PR for each day. Note that this inverter was offline for the last day in this dataset, resulting in a PR value of zero for that day.

```
dates = list(pd.Series(data.index.date).drop_duplicates())

daily_pr_list = list()
for date in dates:
    data_subset = data[data.index.date == date]
    # Run the PR calculation for the specific day.
    pr = performance_ratio_nrel(data_subset['poa_irradiance__1055'],
                                data_subset['ambient_temp__1053'],
                                data_subset['wind_speed__1051'],
                                data_subset['inv2_ac_power_w__1047']/1000,
                                204.12)
    daily_pr_list.append({"date": date,
                          "PR": pr})

daily_pr_df = pd.DataFrame(daily_pr_list)

# Plot the PR time series to visualize it
daily_pr_df.set_index('date').plot()
plt.axhline(pr_whole_series, color='r', ls='--', label='PR, Entire Series')
plt.xticks(rotation=25)
plt.legend()
plt.ylabel('NREL PR')
plt.xlabel('Date')
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.275 seconds)

Orientation

This includes examples related to the orientation of a system (fixed-tilt, tracking).

Flag Sunny Days for a Fixed-Tilt System

Flag sunny days for a fixed-tilt PV system.

Identifying and masking sunny days for a fixed-tilt system is important when performing future analyses that require filtered sunny day data. For this example we will use data from the fixed-tilt NREL SERF East system located on the NREL campus in Colorado, USA, and generate a sunny day mask. This data set is publicly available via the PVDAQ database in the DOE Open Energy Data Initiative (OEDI) (https://data.openei.org/submissions/4568), as system ID 50. This data is timezone-localized.

```
import pvanalytics
from pvanalytics.features import daytime as day
from pvanalytics.features.orientation import fixed_nrel
import matplotlib.pyplot as plt
import pandas as pd
import pathlib
```

First, read in data from the NREL SERF East fixed-tilt system. This data set contains 15-minute interval AC power data.

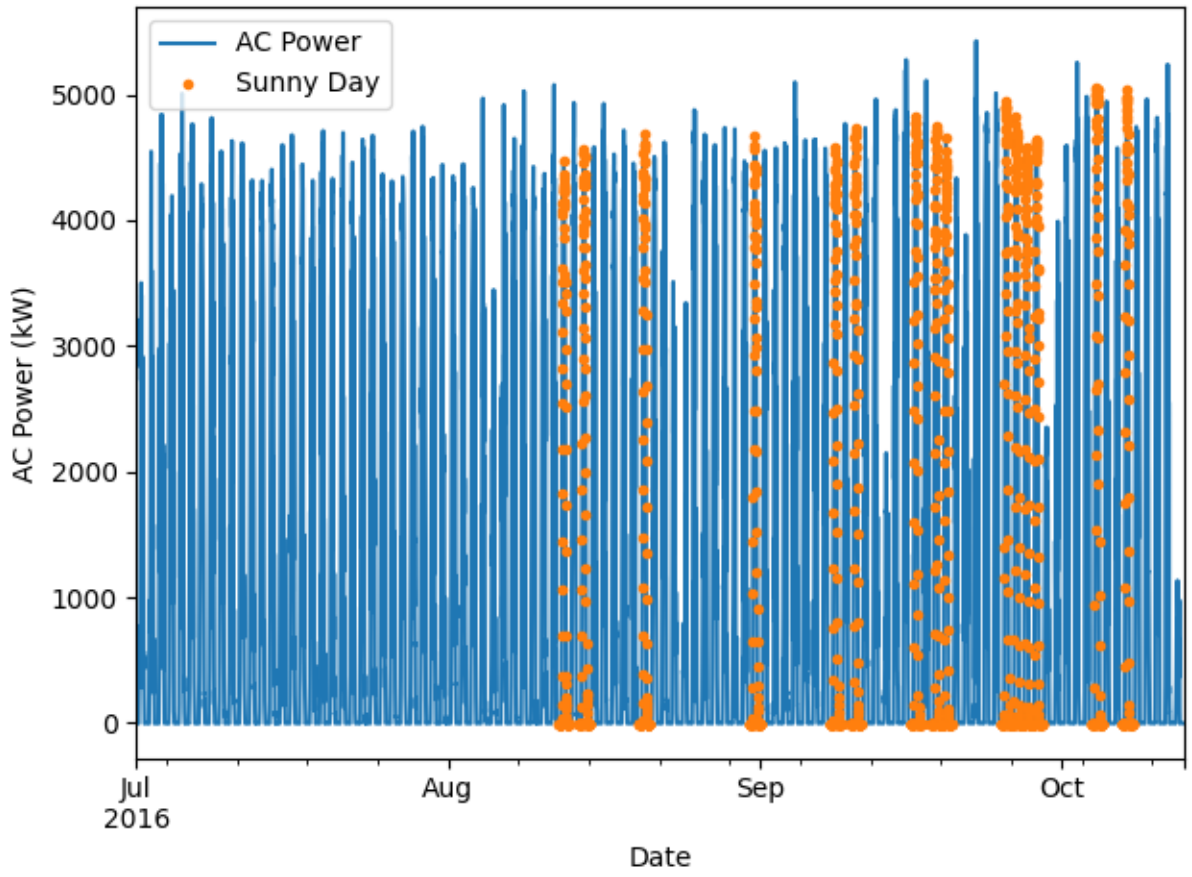
```
pvanalytics_dir = pathlib.Path(pvanalytics.__file__).parent
file = pvanalytics_dir / 'data' / 'serf_east_15min_ac_power.csv'
data = pd.read_csv(file, index_col=0, parse_dates=True)
```

Mask day-night periods using the *pvanalytics.features.daytime.power_or_irradiance()* function. Then apply *pvanalytics.features.orientation.fixed_nrel()* to the AC power stream and mask the sunny days in the time series.

```
daytime_mask = day.power_or_irradiance(data['ac_power'])

fixed_sunny_days = fixed_nrel(data['ac_power'],
                              daytime_mask)
```

Plot the AC power stream with the sunny day mask applied to it.

```
data['ac_power'].plot()
data.loc[fixed_sunny_days, 'ac_power'].plot(ls='', marker='.')
plt.legend(labels=["AC Power", "Sunny Day"],
           loc="upper left")
plt.xlabel("Date")
plt.ylabel("AC Power (kW)")
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.925 seconds)

Flag Sunny Days for a Tracking System

Flag sunny days for a single-axis tracking PV system.

Identifying and masking sunny days for a single-axis tracking system is important when performing future analyses that require filtered sunny day data. For this example we will use data from the single-axis tracking NREL Mesa system located on the NREL campus in Colorado, USA, and generate a sunny day mask. This data set is publicly available via the PVDAQ database in the DOE Open Energy Data Initiative (OEDI) (https://data.openei.org/submissions/4568), as system ID 50. This data is timezone-localized.

```
import pvanalytics
from pvanalytics.features import daytime as day
from pvanalytics.features.orientation import tracking_nrel
import matplotlib.pyplot as plt
import pandas as pd
import pathlib
```

First, read in data from the NREL Mesa 1-axis tracking system. This data set contains 15-minute interval AC power data.

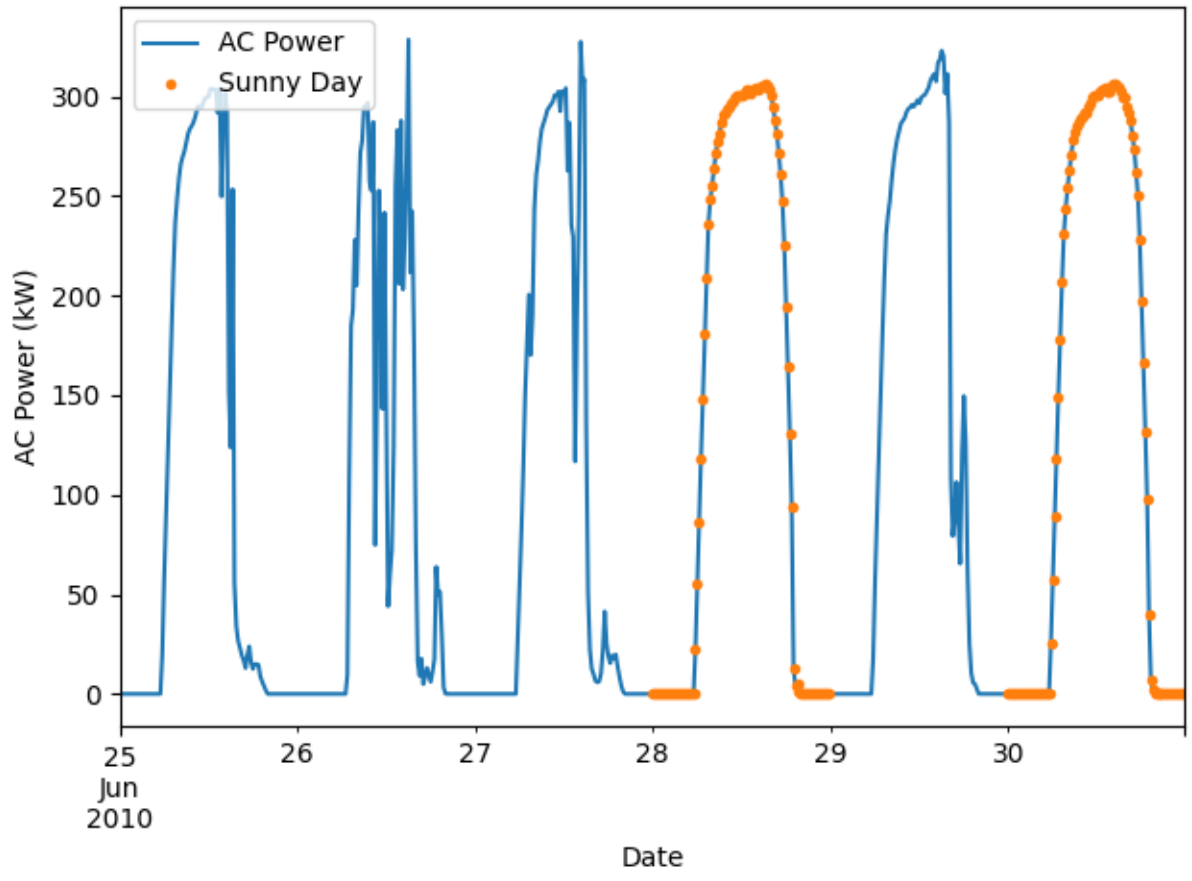
```
pvanalytics_dir = pathlib.Path(pvanalytics.__file__).parent
file = pvanalytics_dir / 'data' / 'nrel_1axis_tracker_mesa_ac_power.csv'
data = pd.read_csv(file, index_col=0, parse_dates=True)
```

Mask day-night periods using the *pvanalytics.features.daytime.power_or_irradiance()* function. Then apply *pvanalytics.features.orientation.tracking_nrel()* to the AC power stream and mask the sunny days in the time series.

```
daytime_mask = day.power_or_irradiance(data['ac_power'])

tracking_sunny_days = tracking_nrel(data['ac_power'],
                                    daytime_mask)
```

Plot the AC power stream with the sunny day mask applied to it.

```
data['ac_power'].plot()
data.loc[tracking_sunny_days, 'ac_power'].plot(ls='', marker='.')
plt.legend(labels=["AC Power", "Sunny Day"],
           loc="upper left")
plt.xlabel("Date")
plt.ylabel("AC Power (kW)")
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.976 seconds)

Outliers

This includes examples for identifying outliers in time series data.

Z-Score Outlier Detection

Identifying outliers in time series using z-score outlier detection.

Identifying and removing outliers from PV sensor time series data allows for more accurate data analysis. In this example, we demonstrate how to use *pvanalytics.quality.outliers.zscore()* to identify and filter out outliers in a time series.

```
import pvanalytics
from pvanalytics.quality.outliers import zscore
import matplotlib.pyplot as plt
import pandas as pd
import pathlib
```

First, we read in the ac_power_inv_7539_outliers example. Min-max normalized AC power is represented by the "value_normalized" column. There is a boolean column "outlier" where inserted outliers are labeled as True, and all other values are labeled as False. These outlier values were inserted manually into the data set to illustrate outlier detection by each of the functions. We use a normalized time series example provided by the PV Fleets Initiative. This example is adapted from the DuraMAT DataHub clipping data set: https://datahub.duramat.org/dataset/inverter-clipping-ml-training-set-real-data

```
pvanalytics_dir = pathlib.Path(pvanalytics.__file__).parent
ac_power_file = pvanalytics_dir / 'data' / 'ac_power_inv_7539_outliers.csv'
data = pd.read_csv(ac_power_file, index_col=0, parse_dates=True)
print(data.head(10))
```

```
                          value_normalized  outlier
timestamp
2017-04-10 19:15:00+00:00          0.000002    False
2017-04-10 19:30:00+00:00          0.000000    False
2017-04-11 06:15:00+00:00          0.000000    False
2017-04-11 06:45:00+00:00          0.033103    False
2017-04-11 07:00:00+00:00          0.043992    False
2017-04-11 07:15:00+00:00          0.055615    False
2017-04-11 07:30:00+00:00          0.110986    False
2017-04-11 07:45:00+00:00          0.184948    False
2017-04-11 08:00:00+00:00          0.276810    False
2017-04-11 08:15:00+00:00          0.358061    False
```

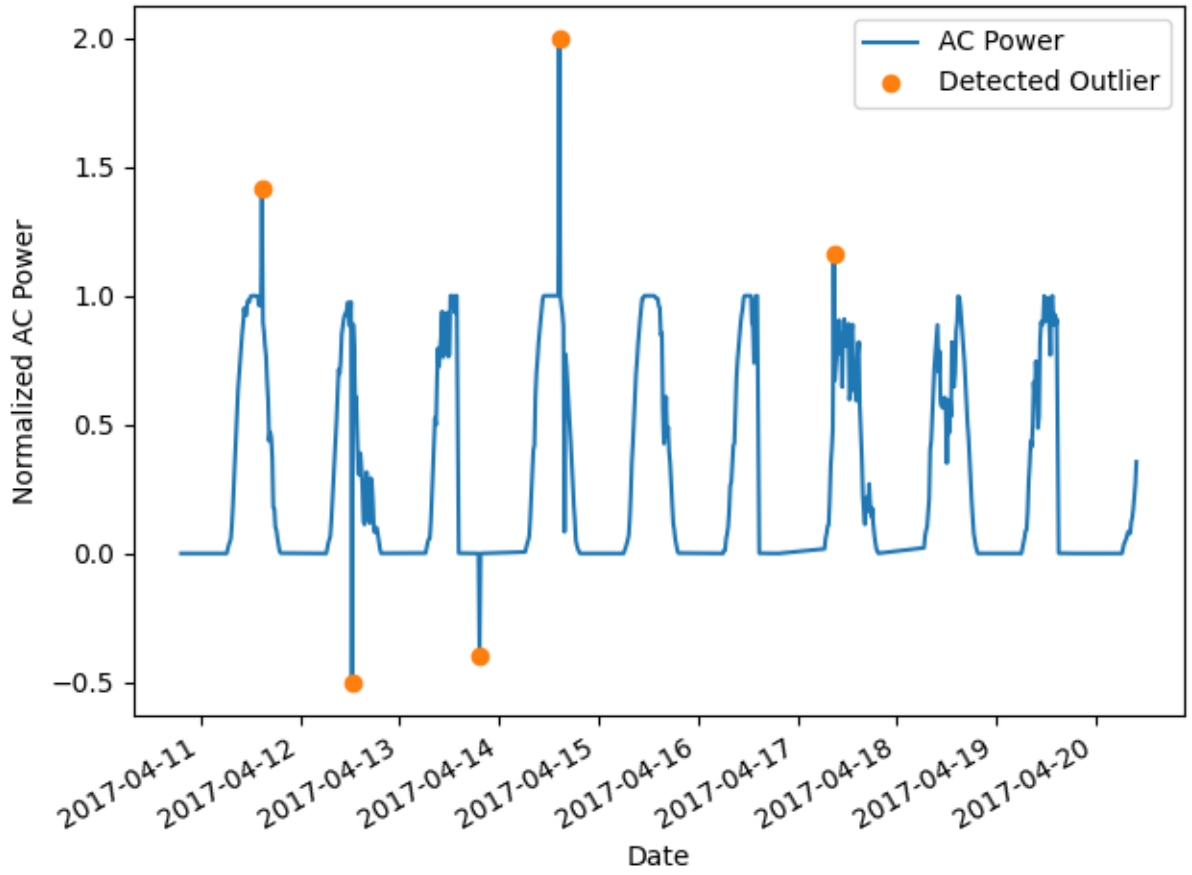
We then use *pvanalytics.quality.outliers.zscore()* to identify outliers in the time series, and plot the data with the z-score outlier mask.

```
zscore_outlier_mask = zscore(data=data['value_normalized'])
data['value_normalized'].plot()
data.loc[zscore_outlier_mask, 'value_normalized'].plot(ls='', marker='o')
plt.legend(labels=["AC Power", "Detected Outlier"])
```

(continues on next page)

(continued from previous page)

```
plt.xlabel("Date")
plt.ylabel("Normalized AC Power")
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.219 seconds)

Tukey Outlier Detection

Identifying outliers in time series using Tukey outlier detection.

Identifying and removing outliers from PV sensor time series data allows for more accurate data analysis. In this example, we demonstrate how to use *pvanalytics.quality.outliers.tukey()* to identify and filter out outliers in a time series.

```
import pvanalytics
from pvanalytics.quality.outliers import tukey
import matplotlib.pyplot as plt
import pandas as pd
import pathlib
```

First, we read in the ac_power_inv_7539_outliers example. Min-max normalized AC power is represented by the "value_normalized" column. There is a boolean column "outlier" where inserted outliers are labeled as True, and

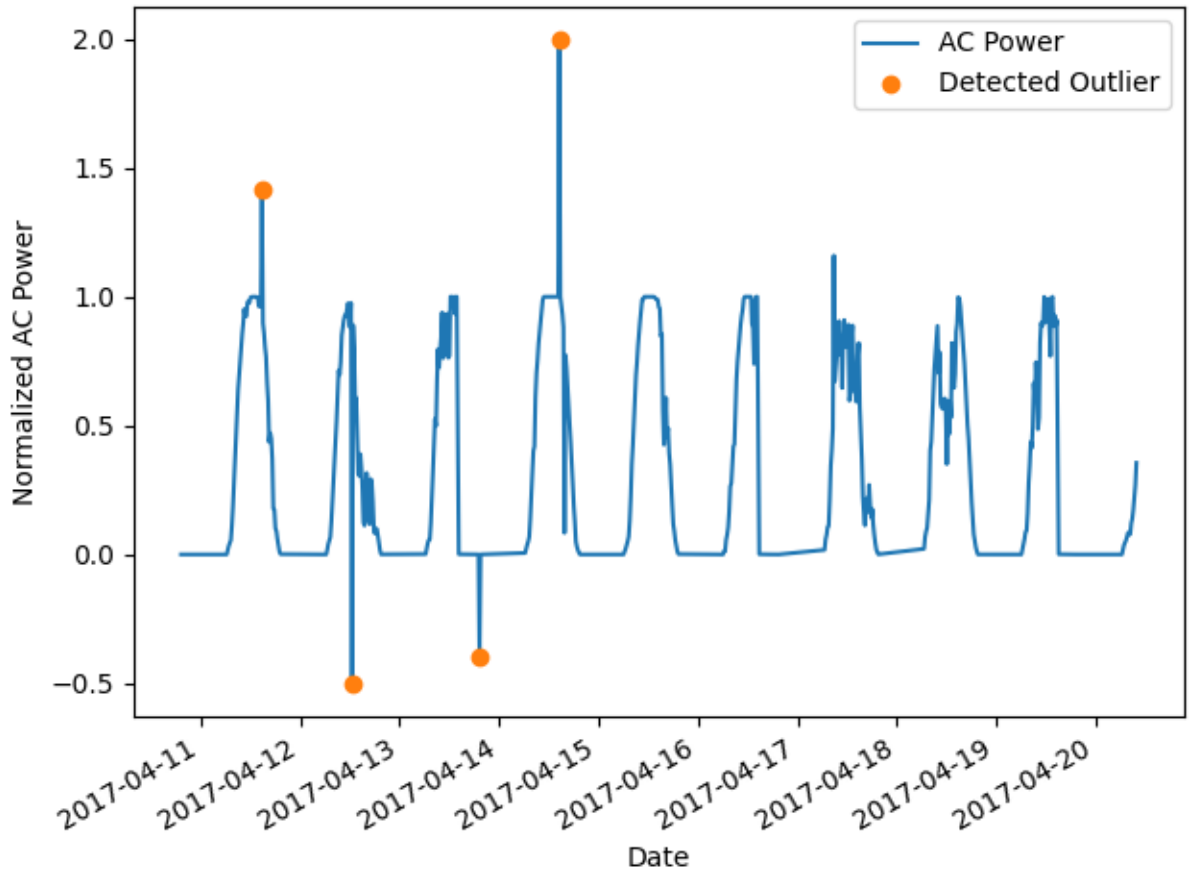
all other values are labeled as False. These outlier values were inserted manually into the data set to illustrate outlier detection by each of the functions. We use a normalized time series example provided by the PV Fleets Initiative. This example is adapted from the DuraMAT DataHub clipping data set: https://datahub.duramat.org/dataset/inverter-clipping-ml-training-set-real-data

```
pvanalytics_dir = pathlib.Path(pvanalytics.__file__).parent
ac_power_file_1 = pvanalytics_dir / 'data' / 'ac_power_inv_7539_outliers.csv'
data = pd.read_csv(ac_power_file_1, index_col=0, parse_dates=True)
print(data.head(10))
```

```
                           value_normalized  outlier
timestamp
2017-04-10 19:15:00+00:00          0.000002    False
2017-04-10 19:30:00+00:00          0.000000    False
2017-04-11 06:15:00+00:00          0.000000    False
2017-04-11 06:45:00+00:00          0.033103    False
2017-04-11 07:00:00+00:00          0.043992    False
2017-04-11 07:15:00+00:00          0.055615    False
2017-04-11 07:30:00+00:00          0.110986    False
2017-04-11 07:45:00+00:00          0.184948    False
2017-04-11 08:00:00+00:00          0.276810    False
2017-04-11 08:15:00+00:00          0.358061    False
```

We then use *pvanalytics.quality.outliers.tukey()* to identify outliers in the time series, and plot the data with the tukey outlier mask.

```
tukey_outlier_mask = tukey(data=data['value_normalized'],
                           k=0.5)
data['value_normalized'].plot()
data.loc[tukey_outlier_mask, 'value_normalized'].plot(ls='', marker='o')
plt.legend(labels=["AC Power", "Detected Outlier"])
plt.xlabel("Date")
plt.ylabel("Normalized AC Power")
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.220 seconds)

Hampel Outlier Detection

Identifying outliers in time series using Hampel outlier detection.

Identifying and removing outliers from PV sensor time series data allows for more accurate data analysis. In this example, we demonstrate how to use *pvanalytics.quality.outliers.hampel()* to identify and filter out outliers in a time series.

```
import pvanalytics
from pvanalytics.quality.outliers import hampel
import matplotlib.pyplot as plt
import pandas as pd
import pathlib
```

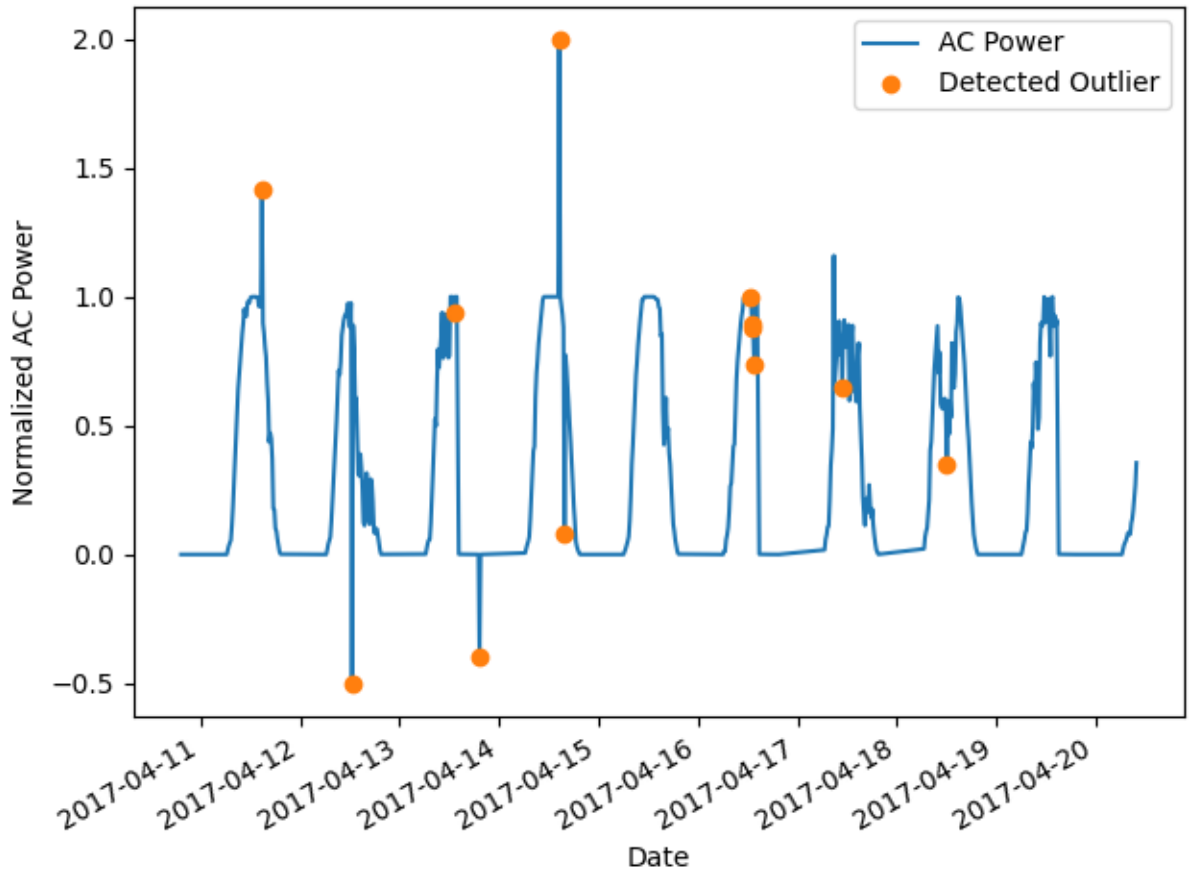
First, we read in the ac_power_inv_7539_outliers example. Min-max normalized AC power is represented by the "value_normalized" column. There is a boolean column "outlier" where inserted outliers are labeled as True, and all other values are labeled as False. These outlier values were inserted manually into the data set to illustrate outlier detection by each of the functions. We use a normalized time series example provided by the PV Fleets Initiative. This example is adapted from the DuraMAT DataHub clipping data set: https://datahub.duramat.org/dataset/inverter-clipping-ml-training-set-real-data

```
pvanalytics_dir = pathlib.Path(pvanalytics.__file__).parent
ac_power_file_1 = pvanalytics_dir / 'data' / 'ac_power_inv_7539_outliers.csv'
data = pd.read_csv(ac_power_file_1, index_col=0, parse_dates=True)
print(data.head(10))
```

```
                           value_normalized  outlier
timestamp
2017-04-10 19:15:00+00:00          0.000002    False
2017-04-10 19:30:00+00:00          0.000000    False
2017-04-11 06:15:00+00:00          0.000000    False
2017-04-11 06:45:00+00:00          0.033103    False
2017-04-11 07:00:00+00:00          0.043992    False
2017-04-11 07:15:00+00:00          0.055615    False
2017-04-11 07:30:00+00:00          0.110986    False
2017-04-11 07:45:00+00:00          0.184948    False
2017-04-11 08:00:00+00:00          0.276810    False
2017-04-11 08:15:00+00:00          0.358061    False
```

We then use *pvanalytics.quality.outliers.hampel()* to identify outliers in the time series, and plot the data with the hampel outlier mask.

```
hampel_outlier_mask = hampel(data=data['value_normalized'],
                             window=10)
data['value_normalized'].plot()
data.loc[hampel_outlier_mask, 'value_normalized'].plot(ls='', marker='o')
plt.legend(labels=["AC Power", "Detected Outlier"])
plt.xlabel("Date")
plt.ylabel("Normalized AC Power")
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.346 seconds)

PVFleets QA Examples

These examples highlight the QA processes for temperature, power and irradiance data streams that are used in the NREL PV Fleet Performance Data Initiative (https://www.nrel.gov/pv/fleet-performance-data-initiative.html).

PV Fleets QA Process: Temperature

PV Fleets Temperature QA Pipeline

The NREL PV Fleets Data Initiative uses PVAnalytics routines to assess the quality of systems' PV data. In this example, the PV Fleets process for assessing the data quality of a temperature data stream is shown. This example pipeline illustrates how several PVAnalytics functions can be used in sequence to assess the quality of a temperature data stream.

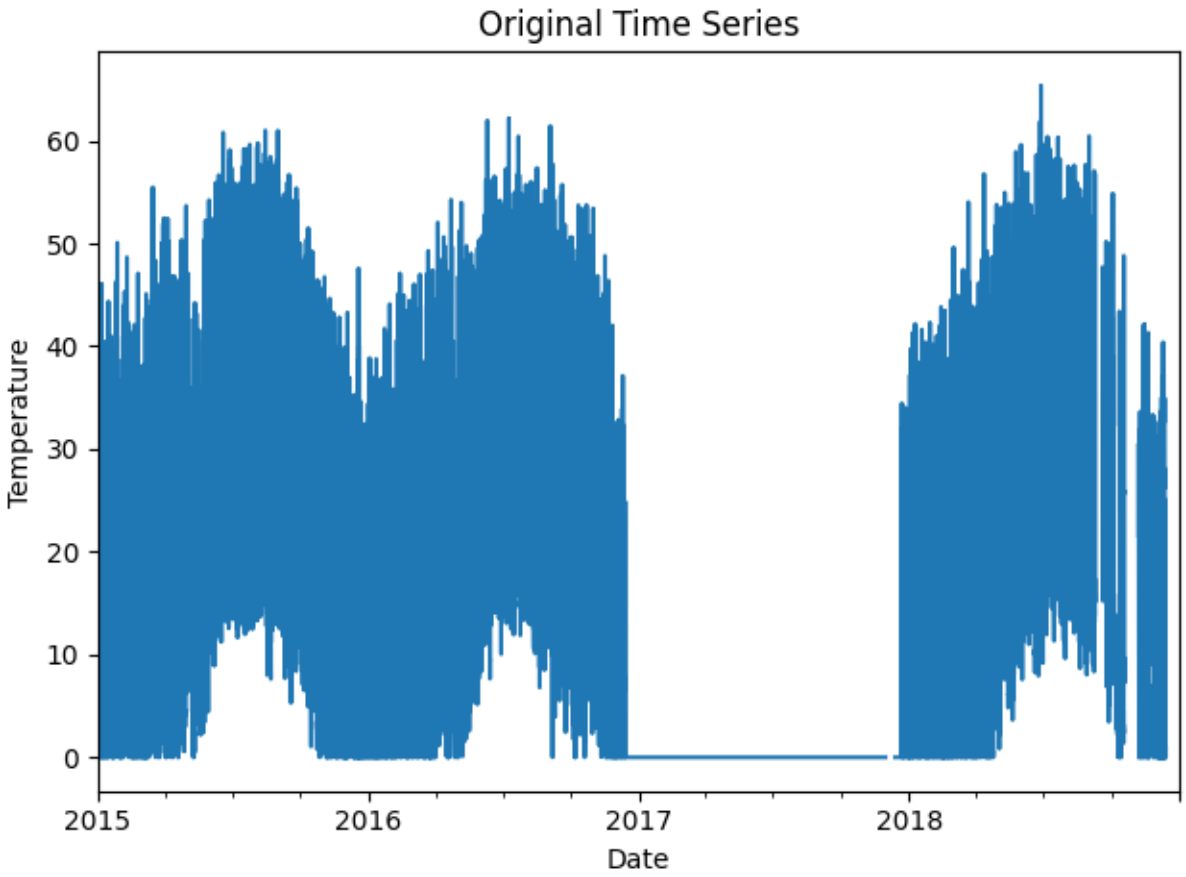
```
import pandas as pd
import pathlib
from matplotlib import pyplot as plt
import pvanalytics
from pvanalytics.quality import data_shifts as ds
from pvanalytics.quality import gaps
from pvanalytics.quality.outliers import zscore
```

First, we import a module temperature data stream from a PV installation at NREL. This data set is publicly available via the PVDAQ database in the DOE Open Energy Data Initiative (OEDI) (https://data.openei.org/submissions/4568), under system ID 4. This data is timezone-localized.

```
pvanalytics_dir = pathlib.Path(pvanalytics.__file__).parent
file = pvanalytics_dir / 'data' / 'system_4_module_temperature.parquet'
time_series = pd.read_parquet(file)
time_series.set_index('index', inplace=True)
time_series.index = pd.to_datetime(time_series.index)
time_series = time_series['module_temp_1']
latitude = 39.7406
longitude = -105.1774
# Identify the temperature data stream type (this affects the type of
# checks we do)
data_stream_type = "module"
data_freq = '15min'
time_series = time_series.asfreq(data_freq)
```

First, let's visualize the original time series as reference.

```
time_series.plot(title="Original Time Series")
plt.xlabel("Date")
plt.ylabel("Temperature")
plt.tight_layout()
plt.show()
```



Now, let's run basic data checks to identify stale and abnormal/outlier data in the time series. Basic data checks include the following steps:

1) Flatlined/stale data periods (*pvanalytics.quality.gaps.stale_values_round()*)
2) "Abnormal" data periods, which are out of the temperature limits of -40 to 185 deg C. Additional checks based on thresholds are applied depending on the type of temperature sensor (ambient or module) (*pvanalytics.quality.weather.temperature_limits()*)
3) Outliers, which are defined as more than one 4 standard deviations away from the mean (*pvanalytics.quality.outliers.zscore()*)

Additionally, we identify the units of the temperature stream as either Celsius or Fahrenheit.

```
# REMOVE STALE DATA
stale_data_mask = gaps.stale_values_round(time_series,
                                          window=3,
                                          decimals=2)

# FIND ABNORMAL PERIODS
temperature_limit_mask = pvanalytics.quality.weather.temperature_limits(
    time_series, limits=(-40, 185))
temperature_limit_mask = temperature_limit_mask.reindex(
    index=time_series.index,
    method='ffill',
    fill_value=False)
```

(continues on next page)

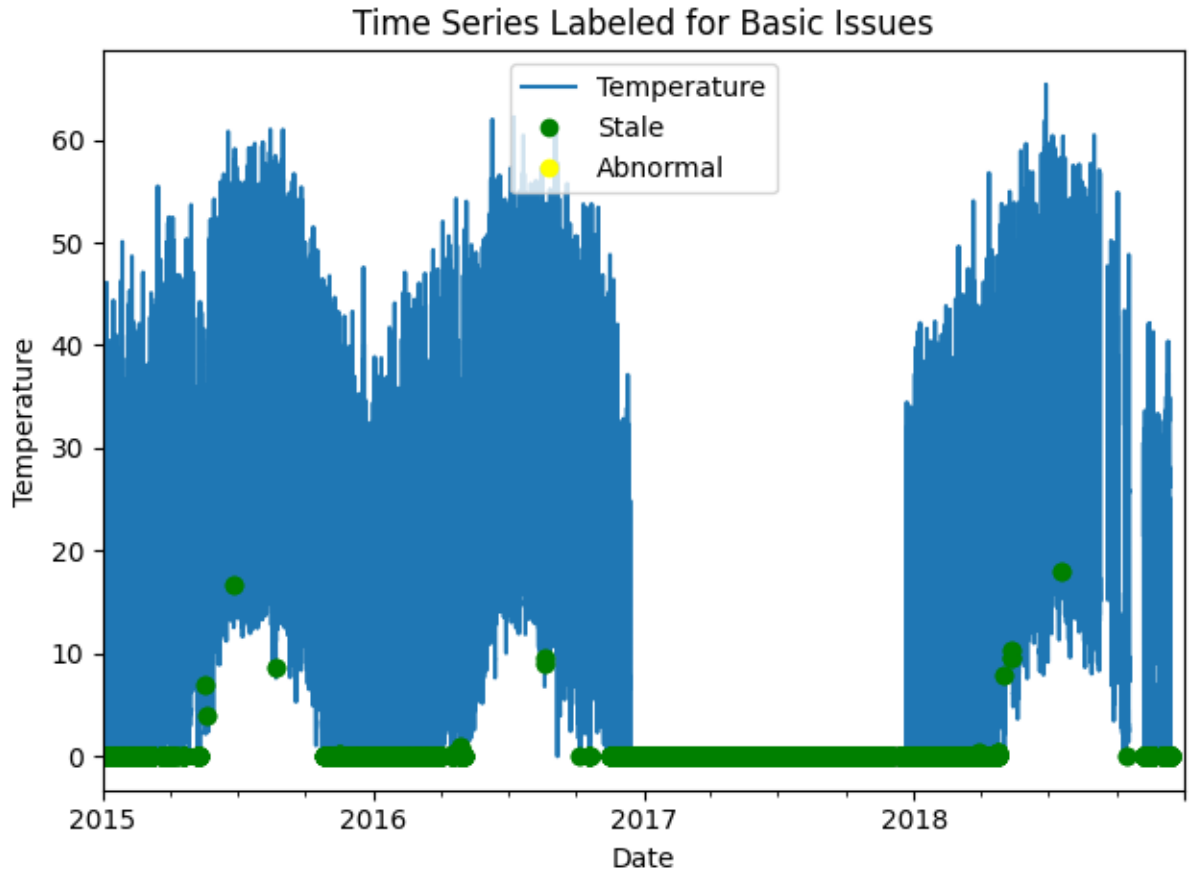
(continued from previous page)

```
# FIND OUTLIERS (Z-SCORE FILTER)
zscore_outlier_mask = zscore(time_series,
                             zmax=4,
                             nan_policy='omit')

# PERFORM ADDITIONAL CHECKS, INCLUDING CHECKING UNITS (CELSIUS OR FAHRENHEIT)
temperature_mean = time_series.mean()
if temperature_mean > 35:
    temp_units = 'F'
else:
    temp_units = 'C'

print("Estimated Temperature units: " + str(temp_units))

# Run additional checks based on temperature sensor type.
if data_stream_type == 'module':
    if temp_units == 'C':
        module_limit_mask = (time_series <= 85)
        temperature_limit_mask = (temperature_limit_mask & module_limit_mask)
if data_stream_type == 'ambient':
    ambient_limit_mask = pvanalytics.quality.weather.temperature_limits(
        time_series, limits=(-40, 120))
    temperature_limit_mask = (temperature_limit_mask & ambient_limit_mask)
    if temp_units == 'C':
        ambient_limit_mask_2 = (time_series <= 50)
        temperature_limit_mask = (temperature_limit_mask &
                                  ambient_limit_mask_2)
# Get the percentage of data flagged for each issue, so it can later be logged
pct_stale = round((len(time_series[
    stale_data_mask].dropna())/len(time_series.dropna())*100), 1)
pct_erroneous = round((len(time_series[
    ~temperature_limit_mask].dropna())/len(time_series.dropna())*100), 1)
pct_outlier = round((len(time_series[
    zscore_outlier_mask].dropna())/len(time_series.dropna())*100), 1)

# Visualize all of the time series issues (stale, abnormal, outlier)
time_series.plot()
labels = ["Temperature"]
if any(stale_data_mask):
    time_series.loc[stale_data_mask].plot(ls='',
                                          marker='o',
                                          color="green")
    labels.append("Stale")
if any(~temperature_limit_mask):
    time_series.loc[~temperature_limit_mask].plot(ls='',
                                                  marker='o',
                                                  color="yellow")
    labels.append("Abnormal")
if any(zscore_outlier_mask):
    time_series.loc[zscore_outlier_mask].plot(ls='',
                                              marker='o',
```

(continues on next page)

(continued from previous page)

```
                                                color="purple")
    labels.append("Outlier")
plt.legend(labels=labels)
plt.title("Time Series Labeled for Basic Issues")
plt.xlabel("Date")
plt.ylabel("Temperature")
plt.tight_layout()
plt.show()
```




```
Estimated Temperature units: C
```

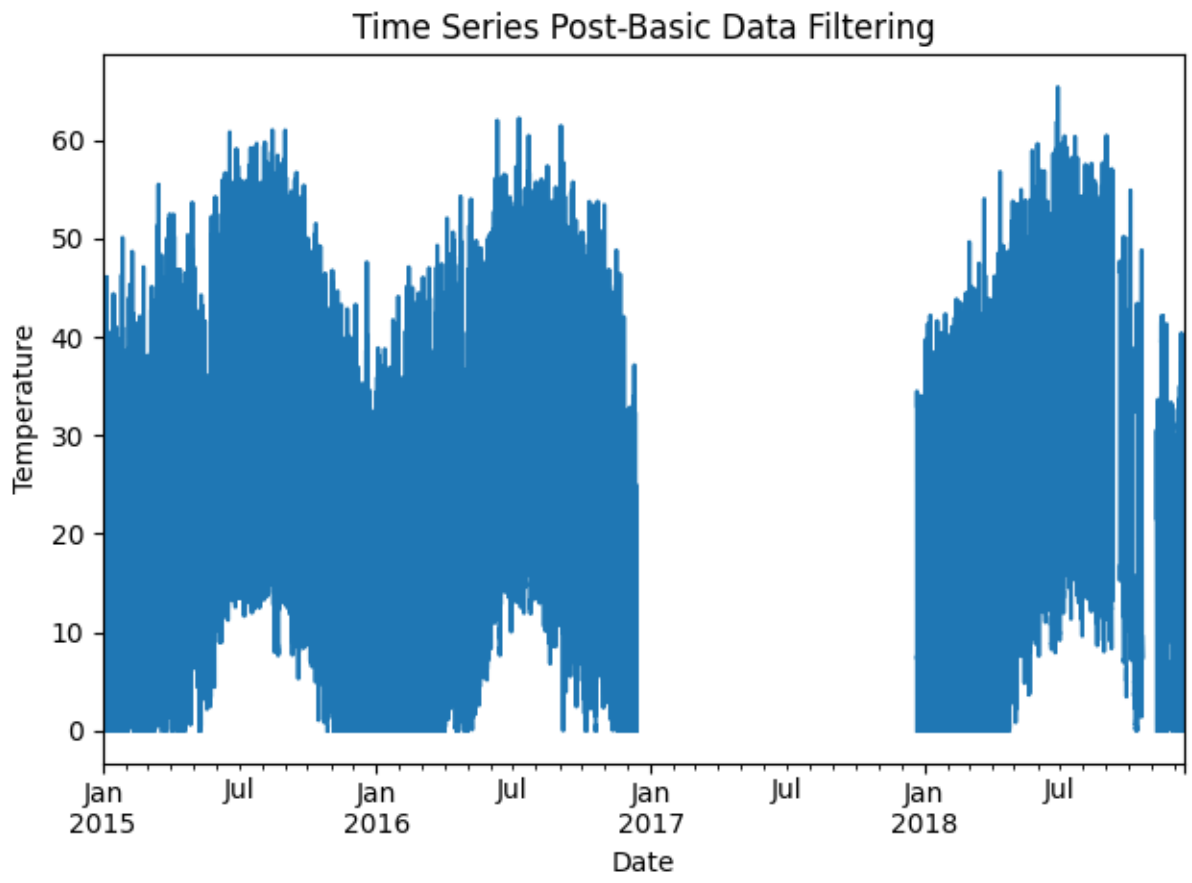
Now, let's filter out any of the flagged data from the basic temperature checks (stale or abnormal data). Then we can re-visualize the data post-filtering.

```
# Filter the time series, taking out all of the issues
issue_mask = ((~stale_data_mask) & (temperature_limit_mask) &
              (~zscore_outlier_mask))
time_series = time_series[issue_mask]
time_series = time_series.asfreq(data_freq)

# Visualize the time series post-filtering
time_series.plot(title="Time Series Post-Basic Data Filtering")
```

(continues on next page)

(continued from previous page)

```
plt.xlabel("Date")
plt.ylabel("Temperature")
plt.tight_layout()
plt.show()
```



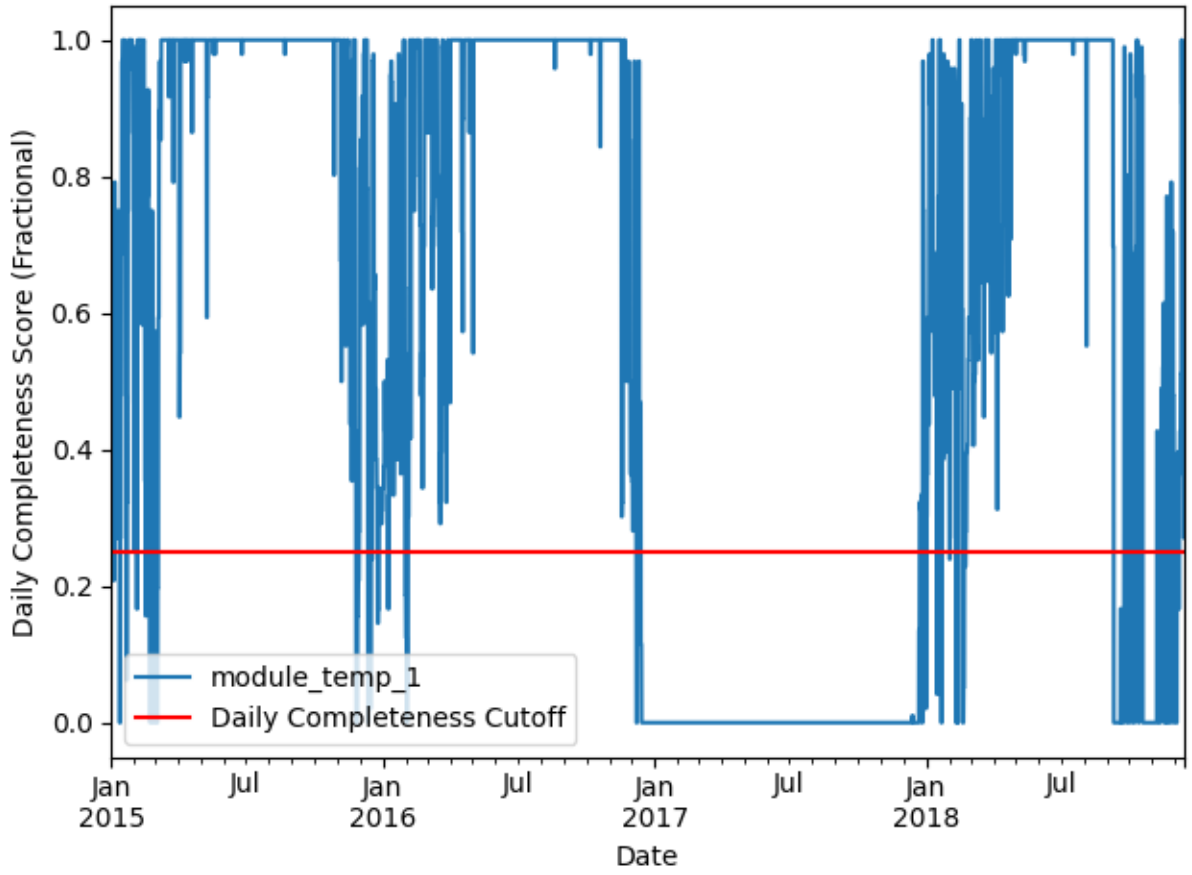
We filter the time series based on its daily completeness score. This filtering scheme requires at least 25% of data to be present for each day to be included. We further require at least 10 consecutive days meeting this 25% threshold to be included.

```
# Visualize daily data completeness
data_completeness_score = gaps.completeness_score(time_series)

# Visualize data completeness score as a time series.
data_completeness_score.plot()
plt.xlabel("Date")
plt.ylabel("Daily Completeness Score (Fractional)")
plt.axhline(y=0.25, color='r', linestyle='-',
            label='Daily Completeness Cutoff')
plt.legend()
plt.tight_layout()
plt.show()

# Trim the series based on daily completeness score
```

(continues on next page)

(continued from previous page)

```
trim_series = pvanalytics.quality.gaps.trim_incomplete(
    time_series,
    minimum_completeness=.25,
    freq=data_freq)
first_valid_date, last_valid_date = \
    pvanalytics.quality.gaps.start_stop_dates(trim_series)
time_series = time_series[first_valid_date.tz_convert(time_series.index.tz):
                          last_valid_date.tz_convert(time_series.index.tz)]
time_series = time_series.asfreq(data_freq)
```



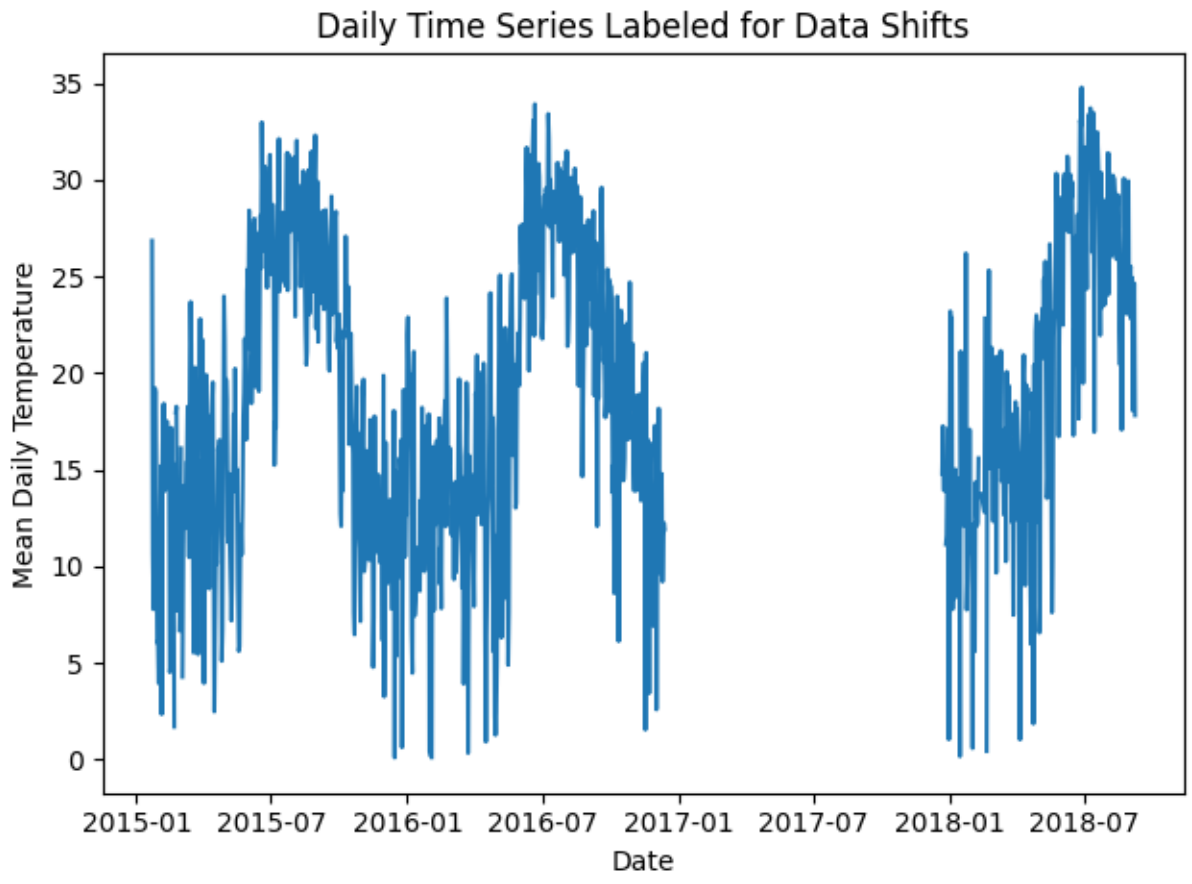
Next, we check the time series for any abrupt data shifts. We take the longest continuous part of the time series that is free of data shifts. We use *pvanalytics.quality.data_shifts.detect_data_shifts()* to detect data shifts in the time series.

```
# Resample the time series to daily mean
time_series_daily = time_series.resample('D').mean()
data_shift_start_date, data_shift_end_date = \
    ds.get_longest_shift_segment_dates(time_series_daily)
data_shift_period_length = (data_shift_end_date -
                            data_shift_start_date).days

# Get the number of shift dates
data_shift_mask = ds.detect_data_shifts(time_series_daily)
```

(continues on next page)

(continued from previous page)

```
# Get the shift dates
shift_dates = list(time_series_daily[data_shift_mask].index)
if len(shift_dates) > 0:
    shift_found = True
else:
    shift_found = False

# Visualize the time shifts for the daily time series
print("Shift Found: ", shift_found)
edges = ([time_series_daily.index[0]] + shift_dates +
         [time_series_daily.index[-1]])
fig, ax = plt.subplots()
for (st, ed) in zip(edges[:-1], edges[1:]):
    ax.plot(time_series_daily.loc[st:ed])
plt.title("Daily Time Series Labeled for Data Shifts")
plt.xlabel("Date")
plt.ylabel("Mean Daily Temperature")
plt.tight_layout()
plt.show()
```

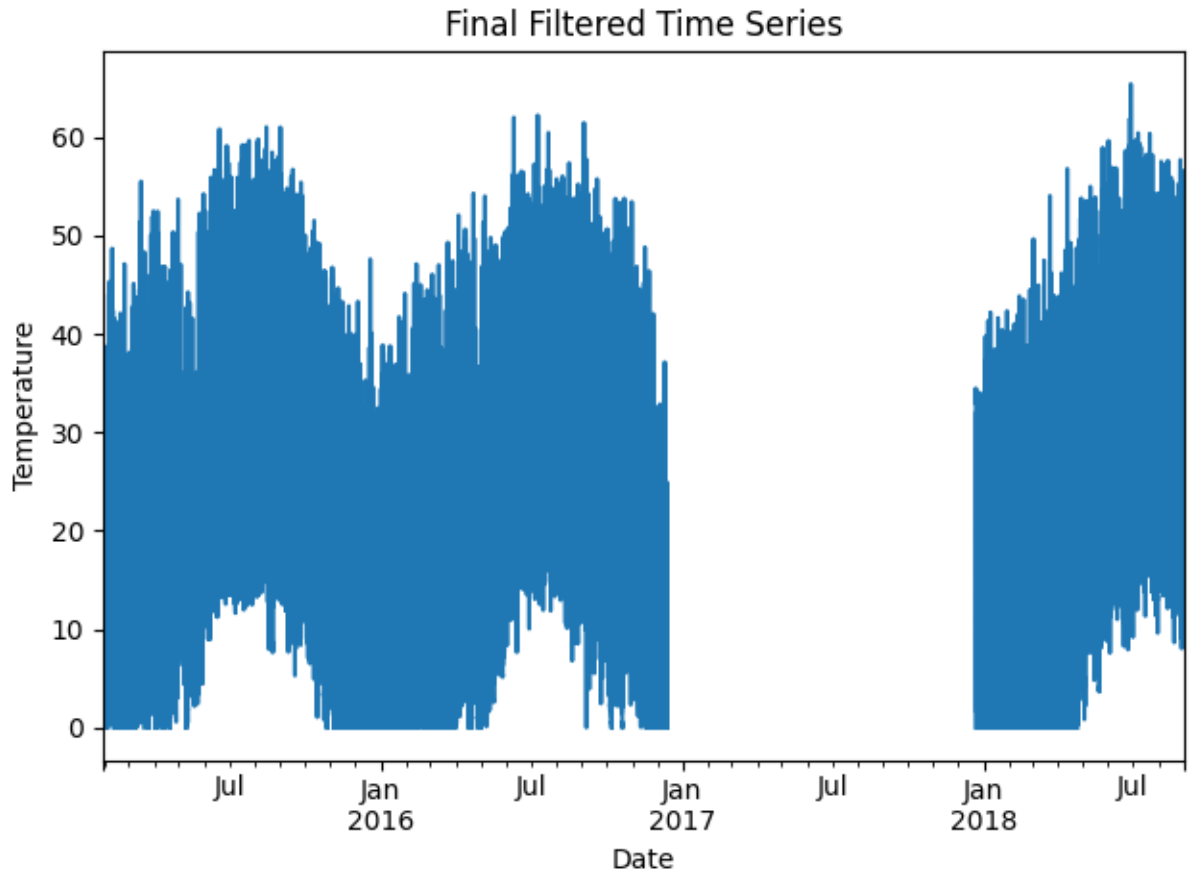



```
Shift Found:  False
```

Finally, we filter the time series to only include the longest shift-free period. We then visualize the final time series

post-QA filtering.

```
time_series = time_series[
    (time_series.index >=
     data_shift_start_date.tz_convert(time_series.index.tz)) &
    (time_series.index <=
     data_shift_end_date.tz_convert(time_series.index.tz))]

time_series = time_series.asfreq(data_freq)

# Plot the final filtered time series.
time_series.plot(title="Final Filtered Time Series")
plt.xlabel("Date")
plt.ylabel("Temperature")
plt.tight_layout()
plt.show()
```



Generate a dictionary output for the QA assessment of this data stream, including the percent stale and erroneous data detected, any shift dates, and the detected temperature units for the data stream.

```
qa_check_dict = {"temperature_units": temp_units,
                 "pct_stale": pct_stale,
                 "pct_erroneous": pct_erroneous,
                 "pct_outlier": pct_outlier,
```

(continues on next page)

(continued from previous page)

```
                "data_shifts": shift_found,
                "shift_dates": shift_dates}

print("QA Results:")
print(qa_check_dict)
```

```
QA Results:
{'temperature_units': 'C', 'pct_stale': 36.6, 'pct_erroneous': 0.0, 'pct_outlier': 0.0,
↪'data_shifts': False, 'shift_dates': []}
```

Total running time of the script: (0 minutes 13.256 seconds)

PV Fleets QA Process: Irradiance

PV Fleets Irradiance QA Pipeline

The NREL PV Fleets Data Initiative uses PVAnalytics routines to assess the quality of systems' PV data. In this example, the PV Fleets process for assessing the data quality of an irradiance data stream is shown. This example pipeline illustrates how several PVAnalytics functions can be used in sequence to assess the quality of an irradiance data stream.

```
import pandas as pd
import pathlib
from matplotlib import pyplot as plt
import pvanalytics
import pvlib
from pvanalytics.quality import data_shifts as ds
from pvanalytics.quality import gaps
from pvanalytics.quality.outliers import zscore
from pvanalytics.features.daytime import power_or_irradiance
from pvanalytics.quality.time import shifts_ruptures
from pvanalytics.features import daytime
```

First, we import a POA irradiance data stream from a PV installation at NREL. This data set is publicly available via the PVDAQ database in the DOE Open Energy Data Initiative (OEDI) (https://data.openei.org/submissions/4568), under system ID 15. This data is timezone-localized.

```
pvanalytics_dir = pathlib.Path(pvanalytics.__file__).parent
file = pvanalytics_dir / 'data' / 'system_15_poa_irradiance.parquet'
time_series = pd.read_parquet(file)
time_series.set_index('measured_on', inplace=True)
time_series.index = pd.to_datetime(time_series.index)
time_series = time_series['poa_irradiance__484']
latitude = 39.7406
longitude = -105.1775
data_freq = '15min'
time_series = time_series.asfreq(data_freq)
```

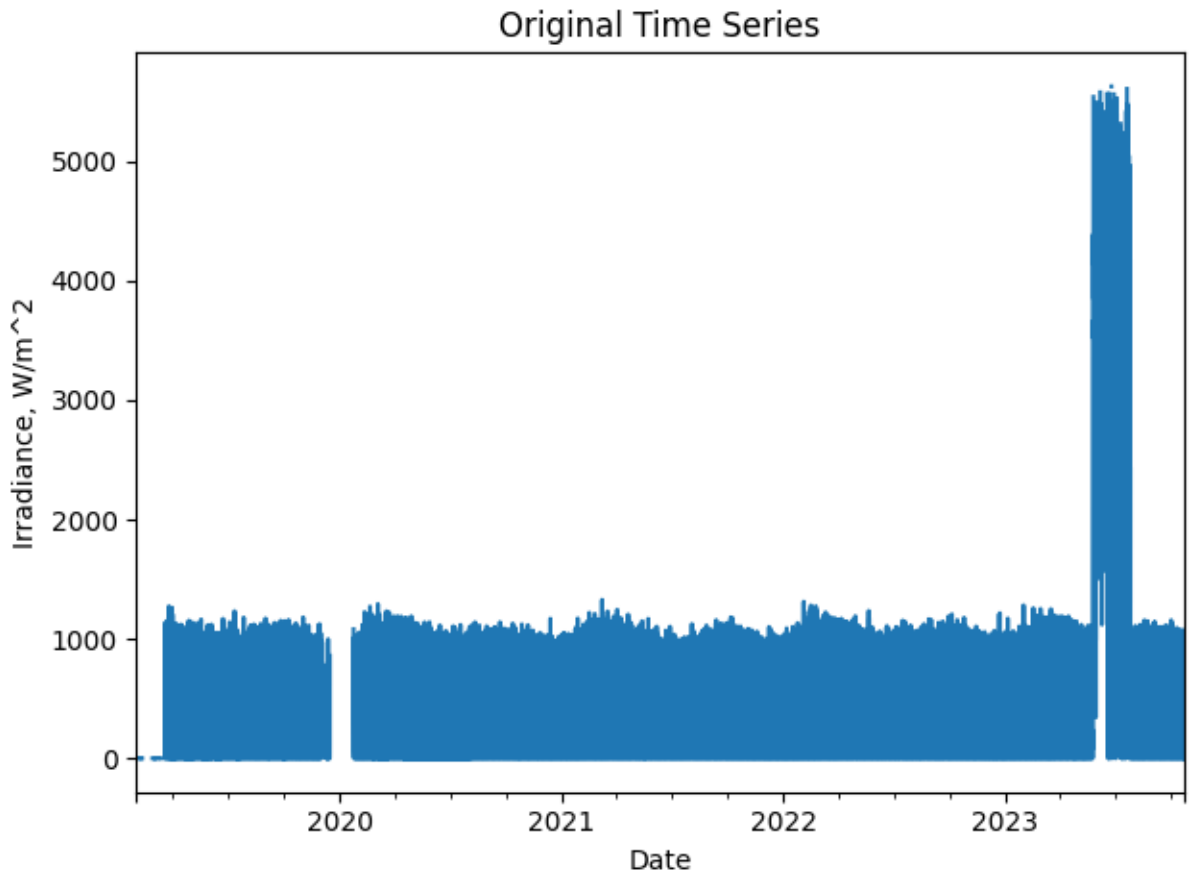
First, let's visualize the original time series as reference.

```
time_series.plot(title="Original Time Series")
plt.xlabel("Date")
```

(continues on next page)

(continued from previous page)

```
plt.ylabel("Irradiance, W/m^2")
plt.tight_layout()
plt.show()
```



Now, let's run basic data checks to identify stale and abnormal/outlier data in the time series. Basic data checks include the following steps:

1) Flatlined/stale data periods (*pvanalytics.quality.gaps.stale_values_round()*)
2) Negative irradiance data
3) "Abnormal" data periods, which are defined as days with a daily minimum greater than 50 OR any data greater than 1300
4) Outliers, which are defined as more than one 4 standard deviations away from the mean (*pvanalytics.quality.outliers.zscore()*)

```
# REMOVE STALE DATA (that isn't during nighttime periods)
# Day/night mask
daytime_mask = power_or_irradiance(time_series)
# Stale data mask
stale_data_mask = gaps.stale_values_round(time_series,
                                          window=3,
                                          decimals=2)
stale_data_mask = stale_data_mask & daytime_mask
```

(continues on next page)

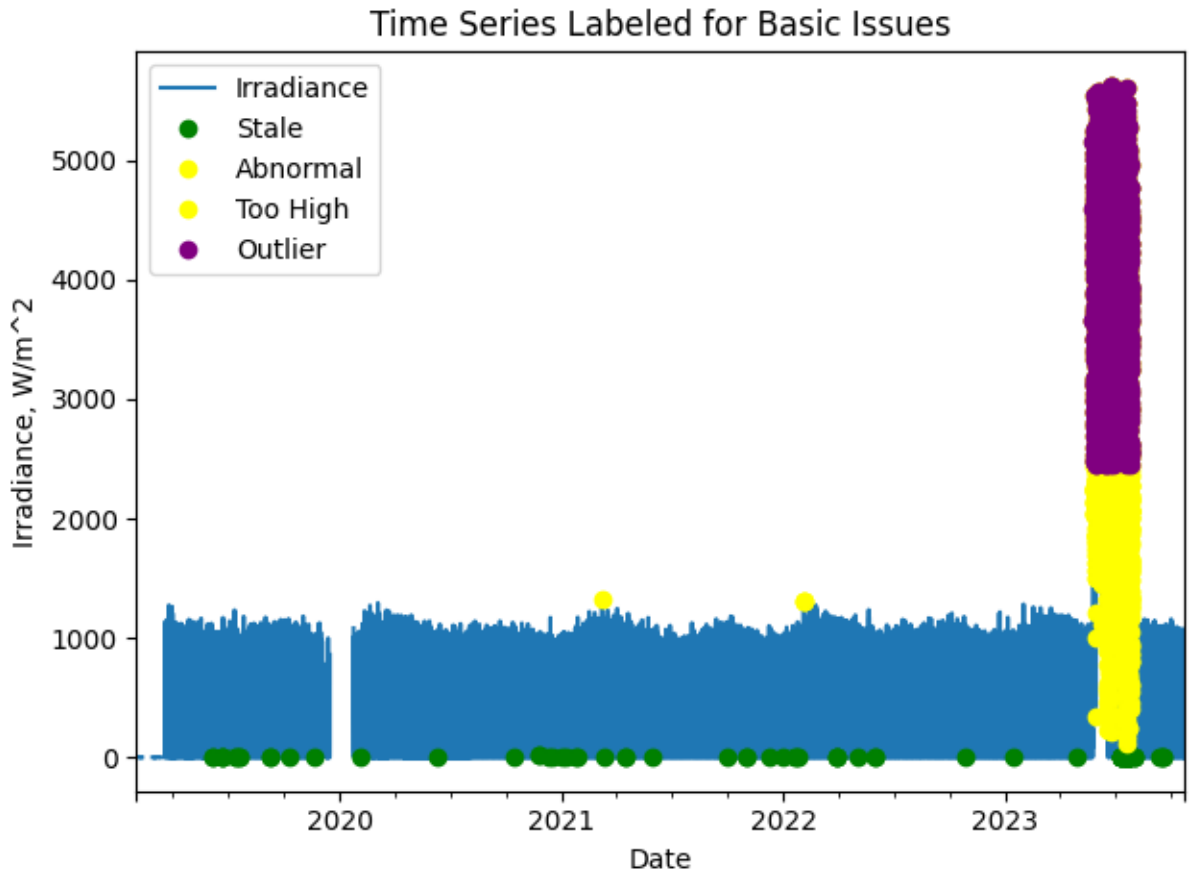
(continued from previous page)

```
# REMOVE NEGATIVE DATA
negative_mask = (time_series < 0)

# FIND ABNORMAL PERIODS
daily_min = time_series.resample('D').min()
erroneous_mask = (daily_min > 50)
erroneous_mask = erroneous_mask.reindex(index=time_series.index,
                                        method='ffill',
                                        fill_value=False)

# Remove values greater than or equal to 1300
out_of_bounds_mask = (time_series >= 1300)

# FIND OUTLIERS (Z-SCORE FILTER)
zscore_outlier_mask = zscore(time_series,
                             zmax=4,
                             nan_policy='omit')

# Get the percentage of data flagged for each issue, so it can later be logged
pct_stale = round((len(time_series[
    stale_data_mask].dropna())/len(time_series.dropna())*100), 1)
pct_negative = round((len(time_series[
    negative_mask].dropna())/len(time_series.dropna())*100), 1)
pct_erroneous = round((len(time_series[
    erroneous_mask].dropna())/len(time_series.dropna())*100), 1)
pct_outlier = round((len(time_series[
    zscore_outlier_mask].dropna())/len(time_series.dropna())*100), 1)

# Visualize all of the time series issues (stale, abnormal, outlier, etc)
time_series.plot()
labels = ["Irradiance"]
if any(stale_data_mask):
    time_series.loc[stale_data_mask].plot(ls='', marker='o', color="green")
    labels.append("Stale")
if any(negative_mask):
    time_series.loc[negative_mask].plot(ls='', marker='o', color="orange")
    labels.append("Negative")
if any(erroneous_mask):
    time_series.loc[erroneous_mask].plot(ls='', marker='o', color="yellow")
    labels.append("Abnormal")
if any(out_of_bounds_mask):
    time_series.loc[out_of_bounds_mask].plot(ls='', marker='o', color="yellow")
    labels.append("Too High")
if any(zscore_outlier_mask):
    time_series.loc[zscore_outlier_mask].plot(
        ls='', marker='o', color="purple")
    labels.append("Outlier")
plt.legend(labels=labels)
plt.title("Time Series Labeled for Basic Issues")
plt.xlabel("Date")
plt.ylabel("Irradiance, W/m^2")
```

(continues on next page)

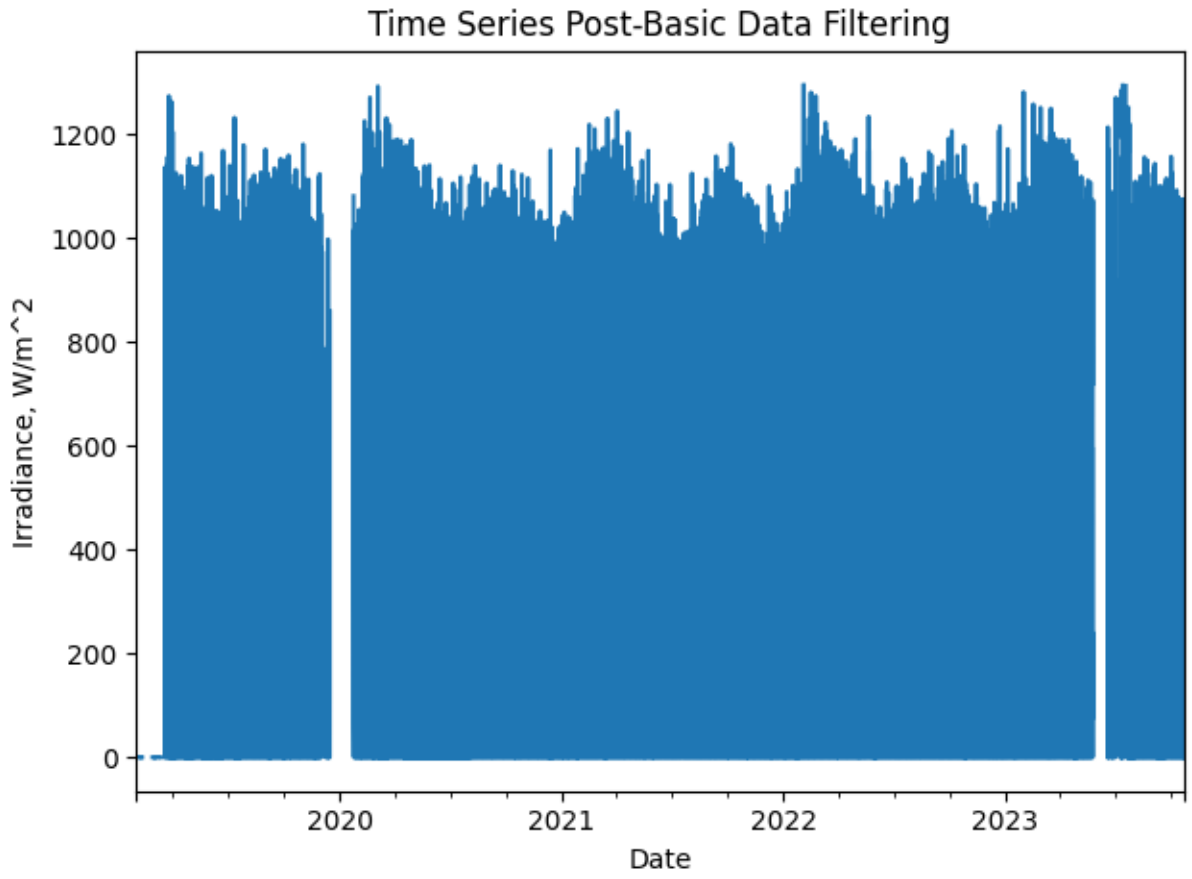
(continued from previous page)

```
plt.tight_layout()
plt.show()
```



Now, let's filter out any of the flagged data from the basic irradiance checks (stale or abnormal data). Then we can re-visualize the data post-filtering.

```
# Filter the time series, taking out all of the issues
issue_mask = ((~stale_data_mask) & (~negative_mask) & (~erroneous_mask) &
              (~out_of_bounds_mask) & (~zscore_outlier_mask))
time_series = time_series[issue_mask]
time_series = time_series.asfreq(data_freq)

# Visualize the time series post-filtering
time_series.plot(title="Time Series Post-Basic Data Filtering")
plt.xlabel("Date")
plt.ylabel("Irradiance, W/m^2")
plt.tight_layout()
plt.show()
```



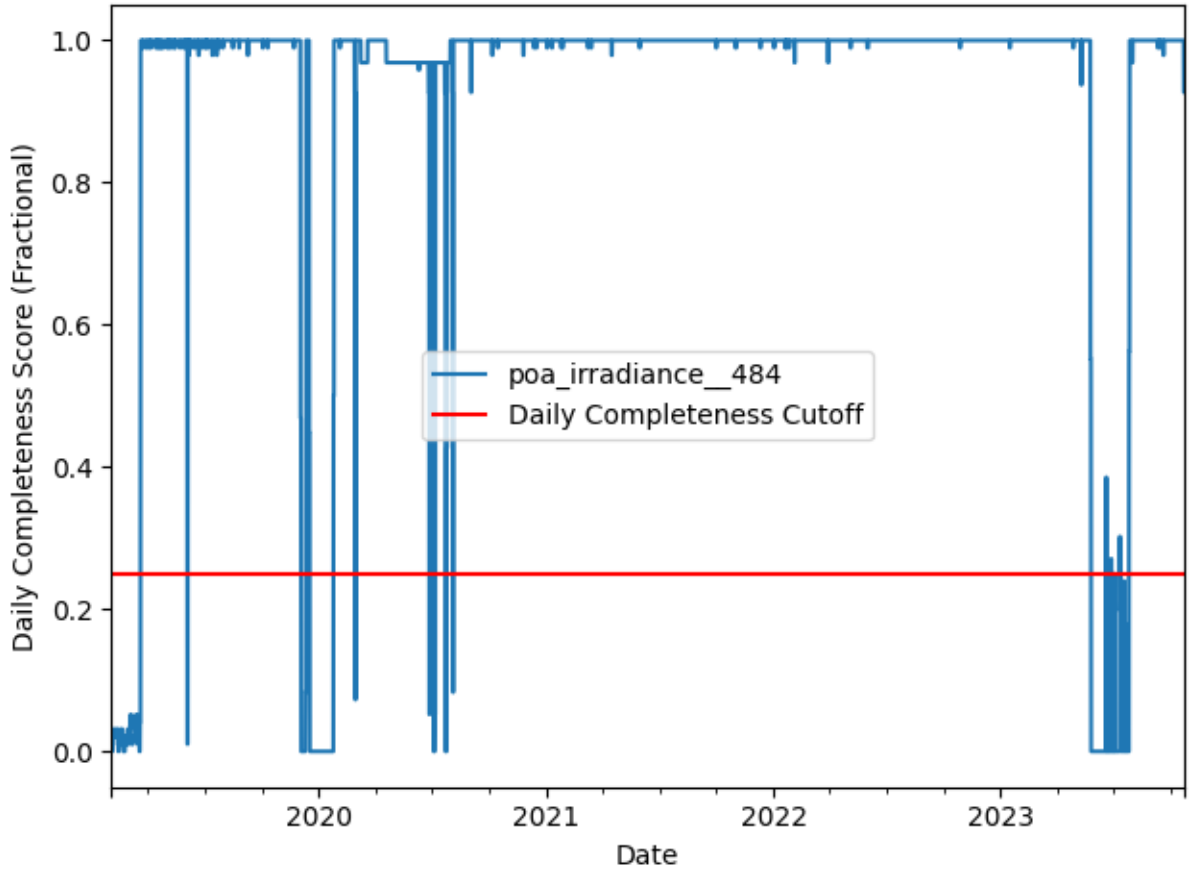
We filter the time series based on its daily completeness score. This filtering scheme requires at least 25% of data to be present for each day to be included. We further require at least 10 consecutive days meeting this 25% threshold to be included.

```
# Visualize daily data completeness
data_completeness_score = gaps.completeness_score(time_series)

# Visualize data completeness score as a time series.
data_completeness_score.plot()
plt.xlabel("Date")
plt.ylabel("Daily Completeness Score (Fractional)")
plt.axhline(y=0.25, color='r', linestyle='-',
            label='Daily Completeness Cutoff')
plt.legend()
plt.tight_layout()
plt.show()

# Trim the series based on daily completeness score
trim_series = pvanalytics.quality.gaps.trim_incomplete(
    time_series,
    minimum_completeness=.25,
    freq=data_freq)
first_valid_date, last_valid_date = \
    pvanalytics.quality.gaps.start_stop_dates(trim_series)
```

(continues on next page)

(continued from previous page)

```
time_series = time_series[first_valid_date.tz_convert(time_series.index.tz):
                          last_valid_date.tz_convert(time_series.index.tz)]
time_series = time_series.asfreq(data_freq)
```



Next, we check the time series for any time shifts, which may be caused by time drift or by incorrect time zone assignment. To do this, we compare the modelled midday time for the particular system location to its measured midday time. We use *pvanalytics.quality.gaps.stale_values_round()*) to determine the presence of time shifts in the series.

```
# Get the modeled sunrise and sunset time series based on the system's
# latitude-longitude coordinates
modeled_sunrise_sunset_df = pvlib.solarposition.sun_rise_set_transit_spa(
    time_series.index, latitude, longitude)

# Calculate the midday point between sunrise and sunset for each day
# in the modeled irradiance series
modeled_midday_series = modeled_sunrise_sunset_df['sunrise'] + \
    (modeled_sunrise_sunset_df['sunset'] -
     modeled_sunrise_sunset_df['sunrise']) / 2

# Run day-night mask on the irradiance time series
daytime_mask = power_or_irradiance(time_series,
                                   freq=data_freq,
```

(continues on next page)

(continued from previous page)

```
                                    low_value_threshold=.005)

# Generate the sunrise, sunset, and halfway points for the data stream
sunrise_series = daytime.get_sunrise(daytime_mask)
sunset_series = daytime.get_sunset(daytime_mask)
midday_series = sunrise_series + ((sunset_series - sunrise_series)/2)

# Convert the midday and modeled midday series to daily values
midday_series_daily, modeled_midday_series_daily = (
    midday_series.resample('D').mean(),
    modeled_midday_series.resample('D').mean())

# Set midday value series as minutes since midnight, from midday datetime
# values
midday_series_daily = (midday_series_daily.dt.hour * 60 +
                       midday_series_daily.dt.minute +
                       midday_series_daily.dt.second / 60)
modeled_midday_series_daily = \
    (modeled_midday_series_daily.dt.hour * 60 +
     modeled_midday_series_daily.dt.minute +
     modeled_midday_series_daily.dt.second / 60)

# Estimate the time shifts by comparing the modelled midday point to the
# measured midday point.
is_shifted, time_shift_series = shifts_ruptures(modeled_midday_series_daily,
                                                midday_series_daily,
                                                period_min=15,
                                                shift_min=15,
                                                zscore_cutoff=1.5)

# Create a midday difference series between modeled and measured midday, to
# visualize time shifts. First, resample each time series to daily frequency,
# and compare the data stream's daily halfway point to the modeled halfway
# point
midday_diff_series = (modeled_midday_series.resample('D').mean() -
                      midday_series.resample('D').mean()
                      ).dt.total_seconds() / 60

# Generate boolean for detected time shifts
if any(time_shift_series != 0):
    time_shifts_detected = True
else:
    time_shifts_detected = False

# Build a list of time shifts for re-indexing. We choose to use dicts.
time_shift_series.index = pd.to_datetime(
    time_shift_series.index)
changepoints = (time_shift_series != time_shift_series.shift(1))
changepoints = changepoints[changepoints].index
changepoint_amts = pd.Series(time_shift_series.loc[changepoints])
time_shift_list = list()
for idx in range(len(changepoint_amts)):
```

(continues on next page)

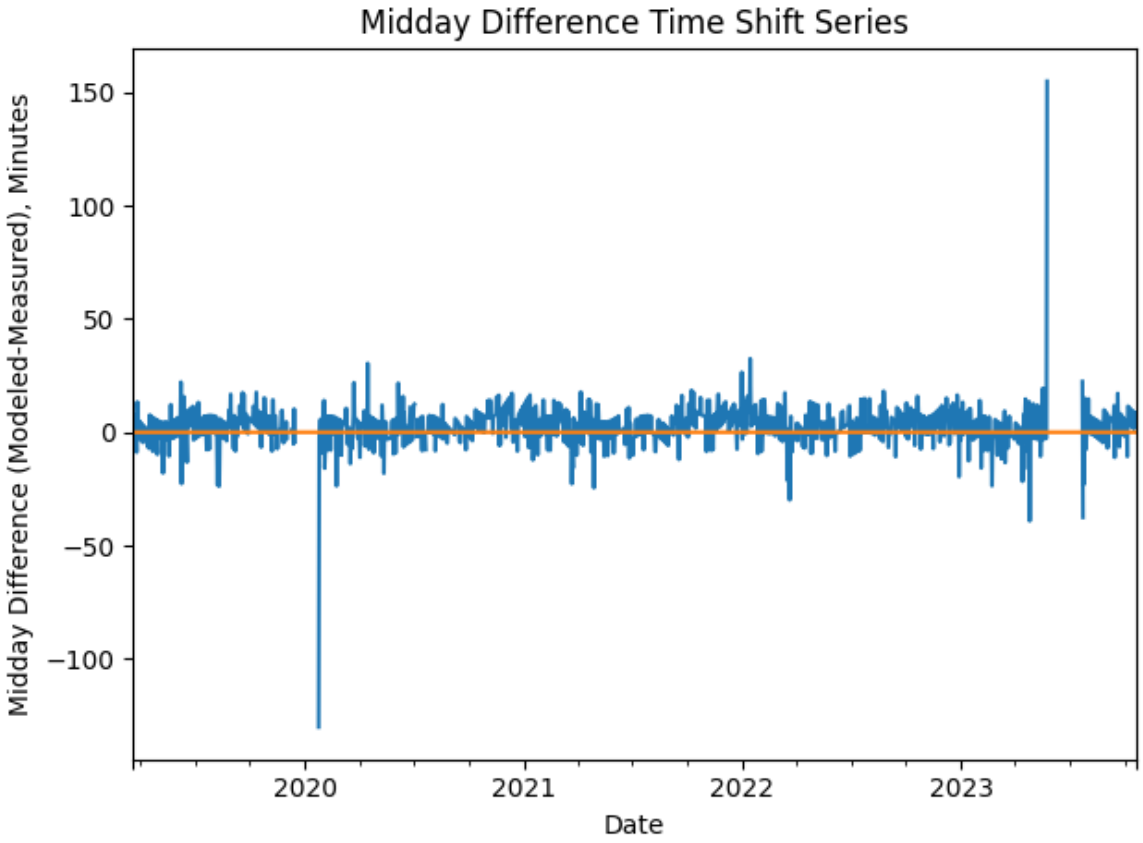
(continued from previous page)

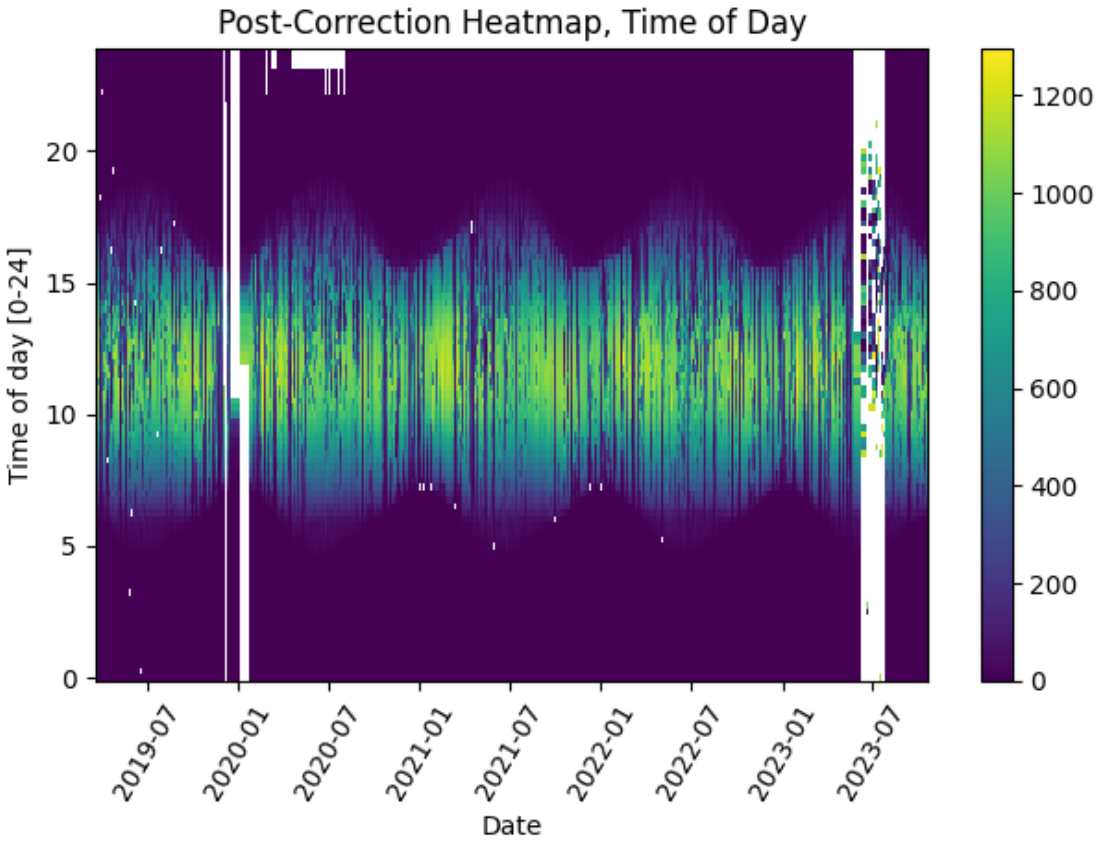
```
    if idx < (len(changepoint_amts) - 1):
        time_shift_list.append({"datetime_start":
                                str(changepoint_amts.index[idx]),
                                "datetime_end":
                                    str(changepoint_amts.index[idx + 1]),
                                "time_shift": changepoint_amts[idx]})
    else:
        time_shift_list.append({"datetime_start":
                                str(changepoint_amts.index[idx]),
                                "datetime_end":
                                    str(time_shift_series.index.max()),
                                "time_shift": changepoint_amts[idx]})

# Correct any time shifts in the time series
new_index = pd.Series(time_series.index, index=time_series.index)
for i in time_shift_list:
    new_index[(time_series.index >= pd.to_datetime(i['datetime_start'])) &
              (time_series.index < pd.to_datetime(i['datetime_end']))] = \
        time_series.index + pd.Timedelta(minutes=i['time_shift'])
time_series.index = new_index

# Remove duplicated indices and sort the time series (just in case)
time_series = time_series[~time_series.index.duplicated(
    keep='first')].sort_index()

# Plot the difference between measured and modeled midday, as well as the
# CPD-estimated time shift series.
midday_diff_series.plot()
time_shift_series.plot()
plt.title("Midday Difference Time Shift Series")
plt.xlabel("Date")
plt.ylabel("Midday Difference (Modeled-Measured), Minutes")
plt.tight_layout()
plt.show()

# Plot the heatmap of the irradiance time series
plt.figure()
# Get time of day from the associated datetime column
time_of_day = pd.Series(time_series.index.hour +
                        time_series.index.minute/60,
                        index=time_series.index)
# Pivot the dataframe
dataframe = pd.DataFrame(pd.concat([time_series, time_of_day], axis=1))
dataframe.columns = ["values", 'time_of_day']
dataframe = dataframe.dropna()
dataframe_pivoted = dataframe.pivot_table(index='time_of_day',
                                          columns=dataframe.index.date,
                                          values="values")
plt.pcolormesh(dataframe_pivoted.columns,
               dataframe_pivoted.index,
               dataframe_pivoted,
               shading='auto')
```

(continues on next page)

(continued from previous page)

```
plt.ylabel('Time of day [0-24]')
plt.xlabel('Date')
plt.xticks(rotation=60)
plt.title('Post-Correction Heatmap, Time of Day')
plt.colorbar()
plt.tight_layout()
plt.show()
```





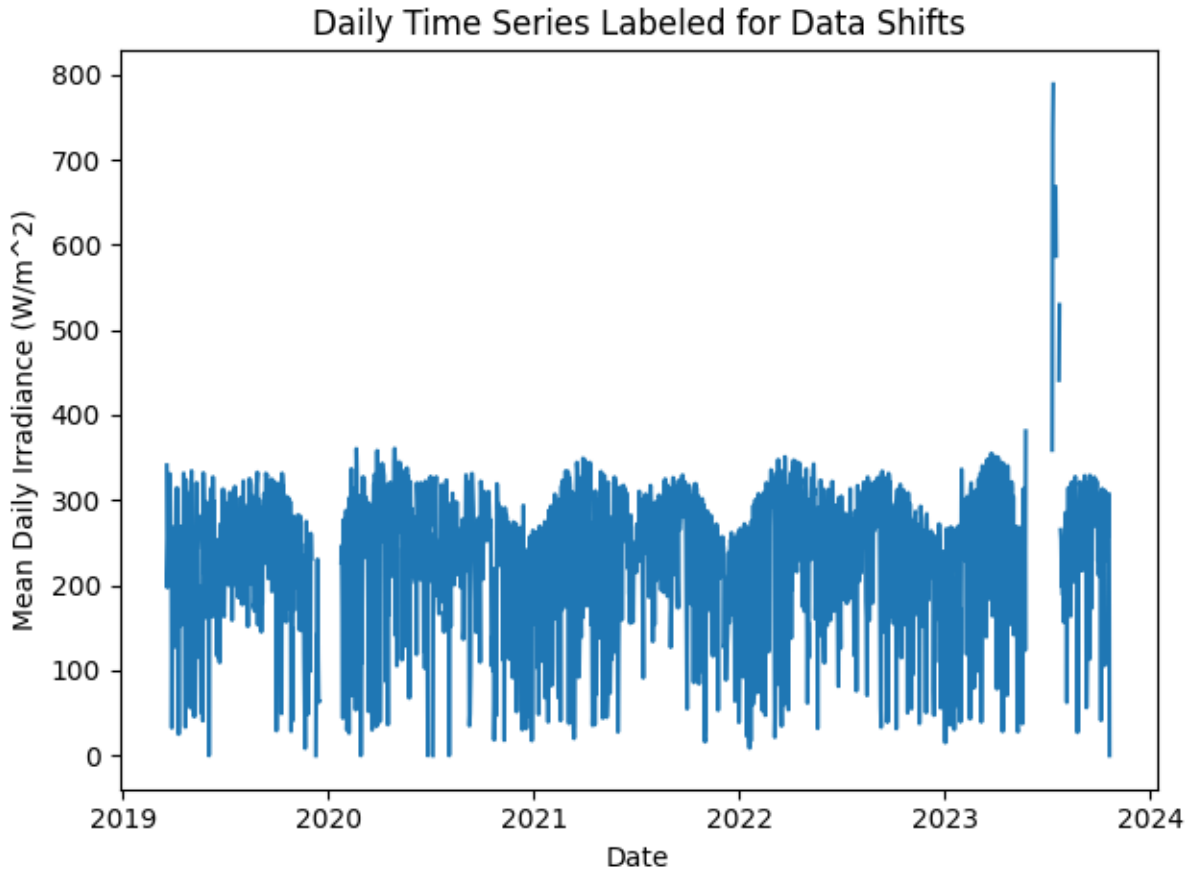
Next, we check the time series for any abrupt data shifts. We take the longest continuous part of the time series that is free of data shifts. We use *pvanalytics.quality.data_shifts.detect_data_shifts()* to detect data shifts in the time series.

```
# Resample the time series to daily mean
time_series_daily = time_series.resample('D').mean()
data_shift_start_date, data_shift_end_date = \
    ds.get_longest_shift_segment_dates(time_series_daily)
data_shift_period_length = (data_shift_end_date - data_shift_start_date).days

# Get the number of shift dates
data_shift_mask = ds.detect_data_shifts(time_series_daily)
# Get the shift dates
shift_dates = list(time_series_daily[data_shift_mask].index)
if len(shift_dates) > 0:
    shift_found = True
else:
    shift_found = False

# Visualize the time shifts for the daily time series
print("Shift Found:", shift_found)
edges = [time_series_daily.index[0]] + \
    shift_dates + [time_series_daily.index[-1]]
fig, ax = plt.subplots()
for (st, ed) in zip(edges[:-1], edges[1:]):
```

(continues on next page)

(continued from previous page)

```
    ax.plot(time_series_daily.loc[st:ed])
plt.title("Daily Time Series Labeled for Data Shifts")
plt.xlabel("Date")
plt.ylabel("Mean Daily Irradiance (W/m^2)")
plt.tight_layout()
plt.show()
```



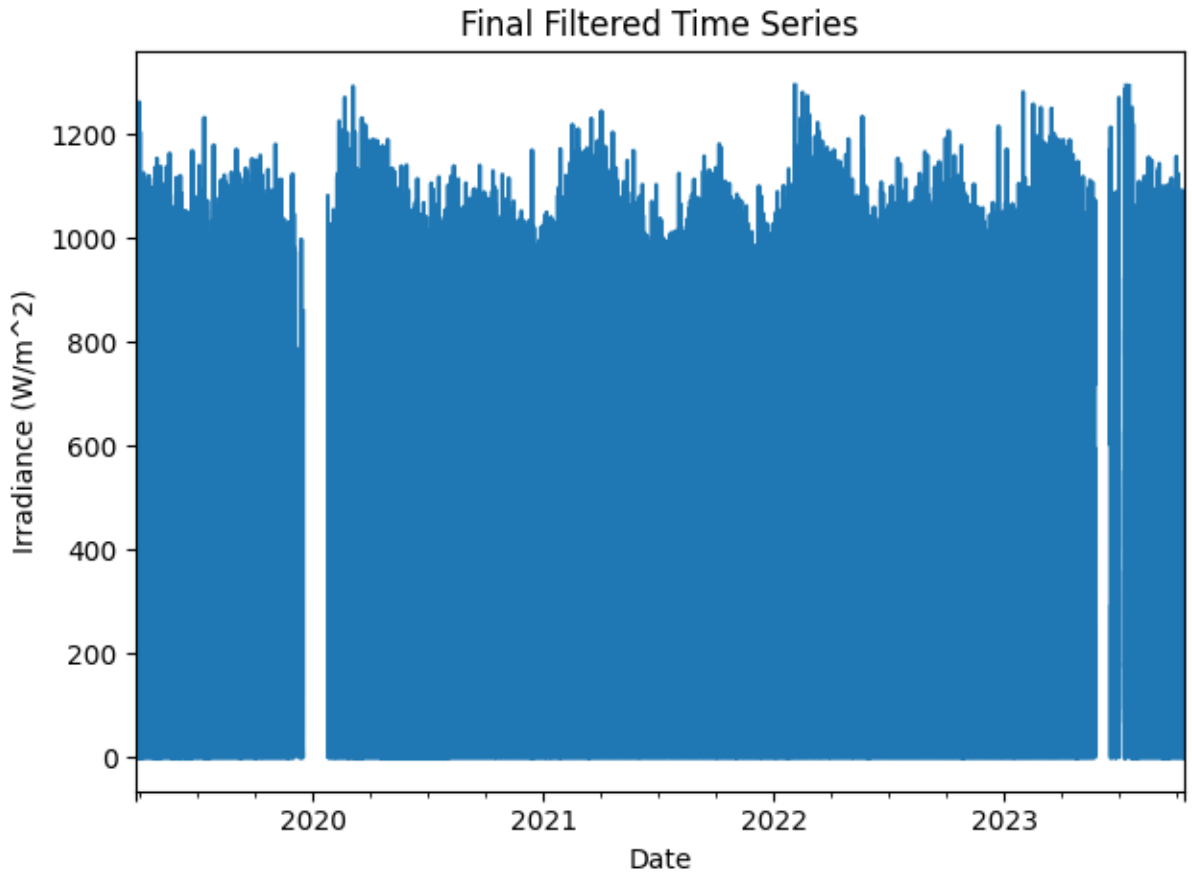

```
Shift Found: False
```

We filter the time series to only include the longest shift-free period.

```
# Filter the time series to only include the longest shift-free period
time_series = time_series[
    (time_series.index >= data_shift_start_date.tz_convert(
        time_series.index.tz)) &
    (time_series.index <= data_shift_end_date.tz_convert(
        time_series.index.tz))]

time_series = time_series.asfreq(data_freq)
```

Display the final irradiance time series, post-QA filtering.

```
time_series.plot(title="Final Filtered Time Series")
plt.xlabel("Date")
plt.ylabel("Irradiance (W/m^2)")
plt.tight_layout()
plt.show()
```



Generate a dictionary output for the QA assessment of this data stream, including the percent stale and erroneous data detected, any shift dates, and any detected time shifts.

```
qa_check_dict = {"original_time_zone_offset": time_series.index.tz,
                 "pct_stale": pct_stale,
                 "pct_negative": pct_negative,
                 "pct_erroneous": pct_erroneous,
                 "pct_outlier": pct_outlier,
                 "time_shifts_detected": time_shifts_detected,
                 "time_shift_list": time_shift_list,
                 "data_shifts": shift_found,
                 "shift_dates": shift_dates}

print("QA Results:")
print(qa_check_dict)
```

```
QA Results:
```

(continues on next page)

(continued from previous page)

```
{'original_time_zone_offset': pytz.FixedOffset(-420), 'pct_stale': 0.1, 'pct_negative':␣
↪0.0, 'pct_erroneous': 1.3, 'pct_outlier': 1.2, 'time_shifts_detected': False, 'time_
↪shift_list': [{'datetime_start': '2019-03-20 00:00:00-07:00', 'datetime_end': '2023-10-
↪21 00:00:00-07:00', 'time_shift': 0.0}], 'data_shifts': False, 'shift_dates': []}
```

Total running time of the script: (0 minutes 33.230 seconds)

PV Fleets QA Process: Power

PV Fleets Power QA Pipeline

The NREL PV Fleets Data Initiative uses PVAnalytics routines to assess the quality of systems' PV data. In this example, the PV Fleets process for assessing the data quality of an AC power data stream is shown. This example pipeline illustrates how several PVAnalytics functions can be used in sequence to assess the quality of a power or energy data stream.

```
import pandas as pd
import pathlib
from matplotlib import pyplot as plt
import pvanalytics
from pvanalytics.quality import data_shifts as ds
from pvanalytics.quality import gaps
from pvanalytics.quality.outliers import zscore
from pvanalytics.system import (is_tracking_envelope,
                                infer_orientation_fit_pvwatts)
from pvanalytics.features.daytime import power_or_irradiance
from pvanalytics.quality.time import shifts_ruptures
from pvanalytics.features import daytime
import pvlib
from pvanalytics.features.clipping import geometric
```

First, we import an AC power data stream from a PV installation at NREL. This data set is publicly available via the PVDAQ database in the DOE Open Energy Data Initiative (OEDI) (https://data.openei.org/submissions/4568), under system ID 50. This data is timezone-localized.

```
pvanalytics_dir = pathlib.Path(pvanalytics.__file__).parent
file = pvanalytics_dir / 'data' / 'system_50_ac_power_2_full_DST.parquet'
time_series = pd.read_parquet(file)
time_series.set_index('measured_on', inplace=True)
time_series.index = pd.to_datetime(time_series.index)
time_series = time_series['ac_power_2']
latitude = 39.7406
longitude = -105.1775
data_freq = '15min'
time_series = time_series.asfreq(data_freq)
```

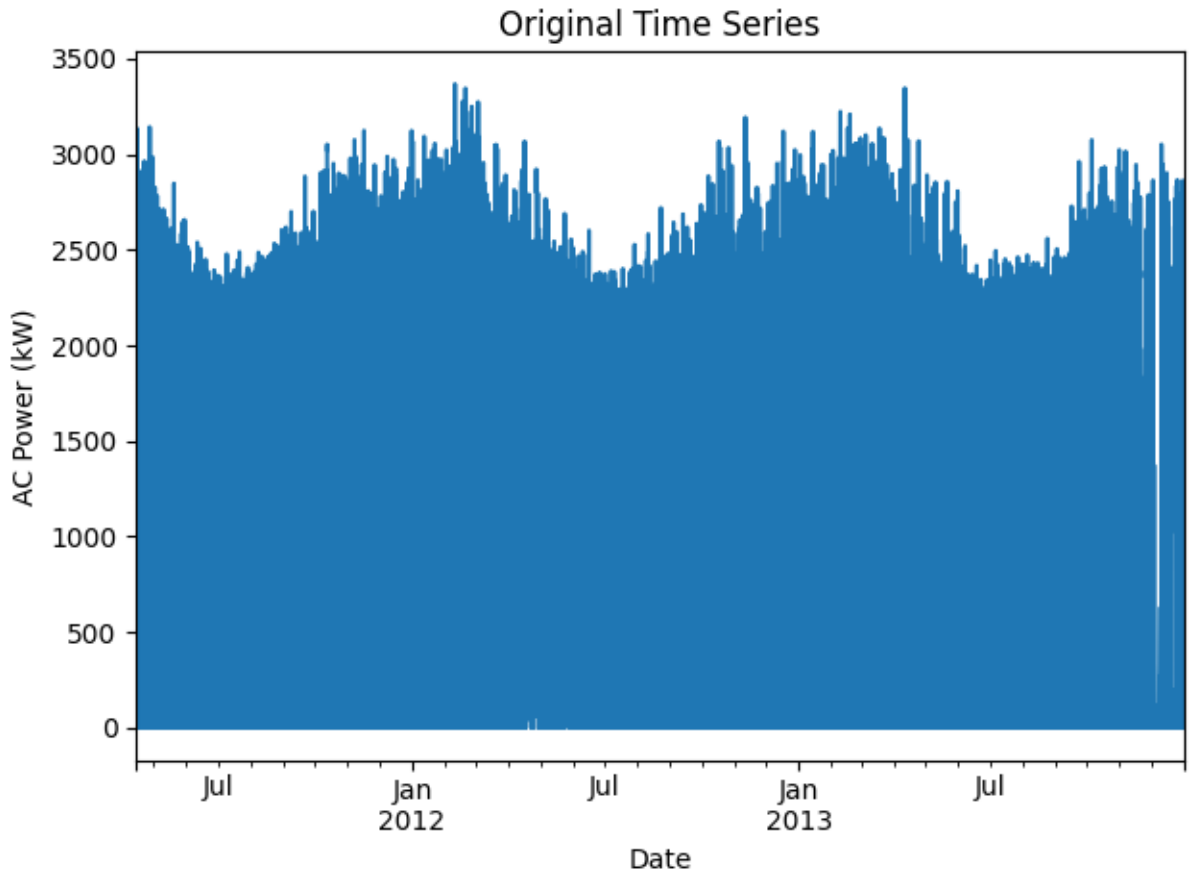
First, let's visualize the original time series as reference.

```
time_series.plot(title="Original Time Series")
plt.xlabel("Date")
plt.ylabel("AC Power (kW)")
```

(continues on next page)

(continued from previous page)

```
plt.tight_layout()
plt.show()
```



Now, let's run basic data checks to identify stale and abnormal/outlier data in the time series. Basic data checks include the following steps:

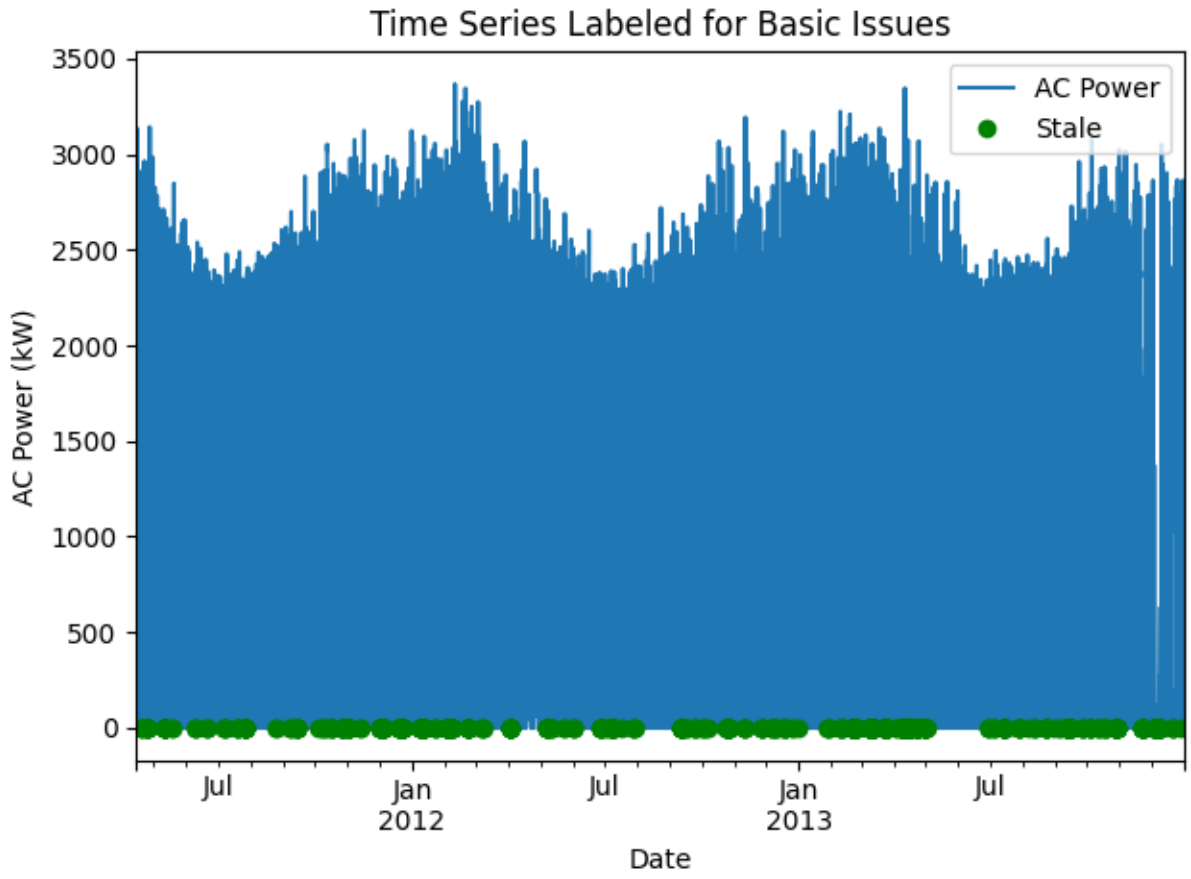
1) Flatlined/stale data periods (*pvanalytics.quality.gaps.stale_values_round()*)
2) Negative data
3) "Abnormal" data periods, which are defined as less than 10% of the daily time series mean
3) Outliers, which are defined as more than one 4 standard deviations away from the mean (*pvanalytics.quality.outliers.zscore()*)

```
# REMOVE STALE DATA (that isn't during nighttime periods)
# Day/night mask
daytime_mask = power_or_irradiance(time_series)
# Stale data mask
stale_data_mask = gaps.stale_values_round(time_series,
                                          window=3,
                                          decimals=2)
stale_data_mask = stale_data_mask & daytime_mask

# REMOVE NEGATIVE DATA
```

(continues on next page)

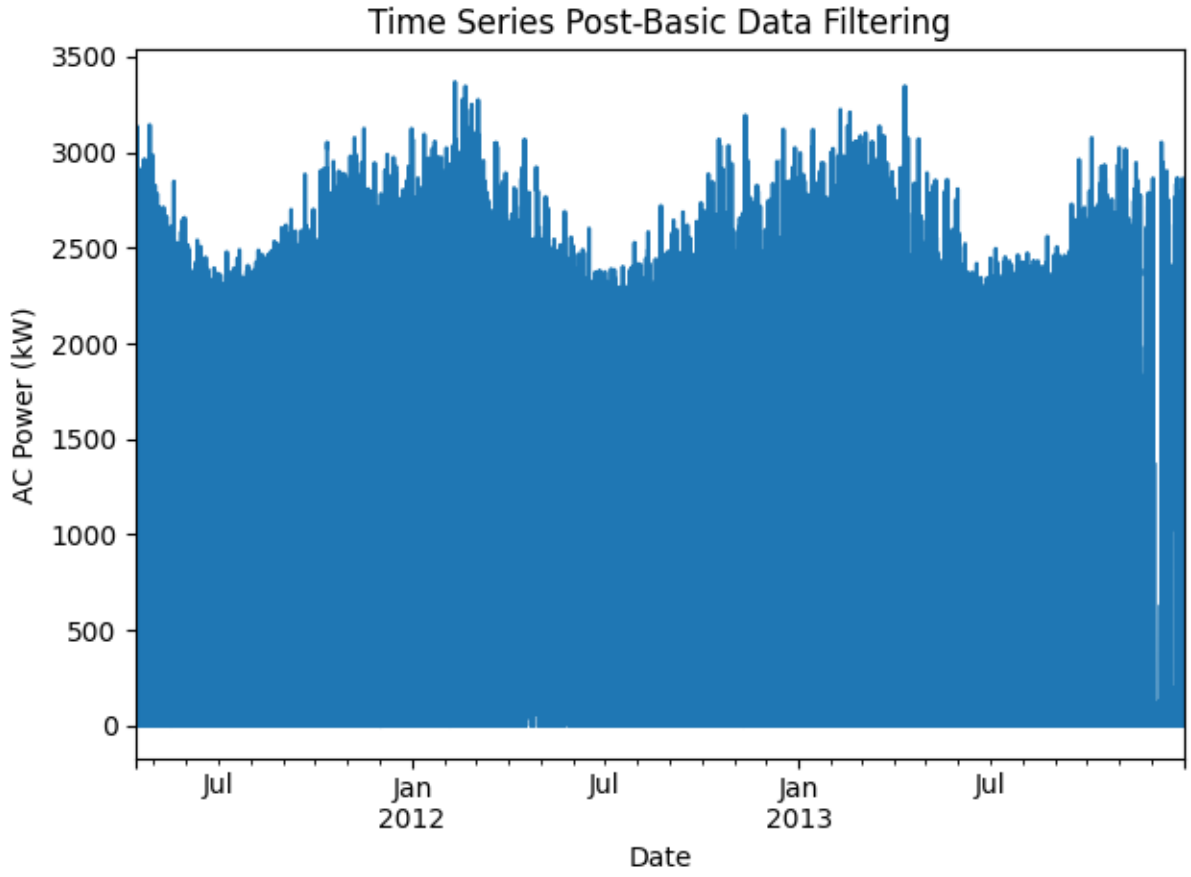
(continued from previous page)

```
negative_mask = (time_series < 0)

# FIND ABNORMAL PERIODS
daily_min = time_series.resample('D').min()
series_min = 0.1 * time_series.mean()
erroneous_mask = (daily_min >= series_min)
erroneous_mask = erroneous_mask.reindex(index=time_series.index,
                                        method='ffill',
                                        fill_value=False)

# FIND OUTLIERS (Z-SCORE FILTER)
zscore_outlier_mask = zscore(time_series, zmax=4,
                             nan_policy='omit')

# Get the percentage of data flagged for each issue, so it can later be logged
pct_stale = round((len(time_series[
    stale_data_mask].dropna())/len(time_series.dropna())*100), 1)
pct_negative = round((len(time_series[
    negative_mask].dropna())/len(time_series.dropna())*100), 1)
pct_erroneous = round((len(time_series[
    erroneous_mask].dropna())/len(time_series.dropna())*100), 1)
pct_outlier = round((len(time_series[
    zscore_outlier_mask].dropna())/len(time_series.dropna())*100), 1)

# Visualize all of the time series issues (stale, abnormal, outlier, etc)
time_series.plot()
labels = ["AC Power"]
if any(stale_data_mask):
    time_series.loc[stale_data_mask].plot(ls='', marker='o', color="green")
    labels.append("Stale")
if any(negative_mask):
    time_series.loc[negative_mask].plot(ls='', marker='o', color="orange")
    labels.append("Negative")
if any(erroneous_mask):
    time_series.loc[erroneous_mask].plot(ls='', marker='o', color="yellow")
    labels.append("Abnormal")
if any(zscore_outlier_mask):
    time_series.loc[zscore_outlier_mask].plot(
        ls='', marker='o', color="purple")
    labels.append("Outlier")
plt.legend(labels=labels)
plt.title("Time Series Labeled for Basic Issues")
plt.xlabel("Date")
plt.ylabel("AC Power (kW)")
plt.tight_layout()
plt.show()
```



Now, let's filter out any of the flagged data from the basic power checks (stale or abnormal data). Then we can re-visualize the data post-filtering.

```
# Filter the time series, taking out all of the issues
issue_mask = ((~stale_data_mask) & (~negative_mask) &
              (~erroneous_mask) & (~zscore_outlier_mask))

time_series = time_series[issue_mask]
time_series = time_series.asfreq(data_freq)

# Visualize the time series post-filtering
time_series.plot(title="Time Series Post-Basic Data Filtering")
plt.xlabel("Date")
plt.ylabel("AC Power (kW)")
plt.tight_layout()
plt.show()
```



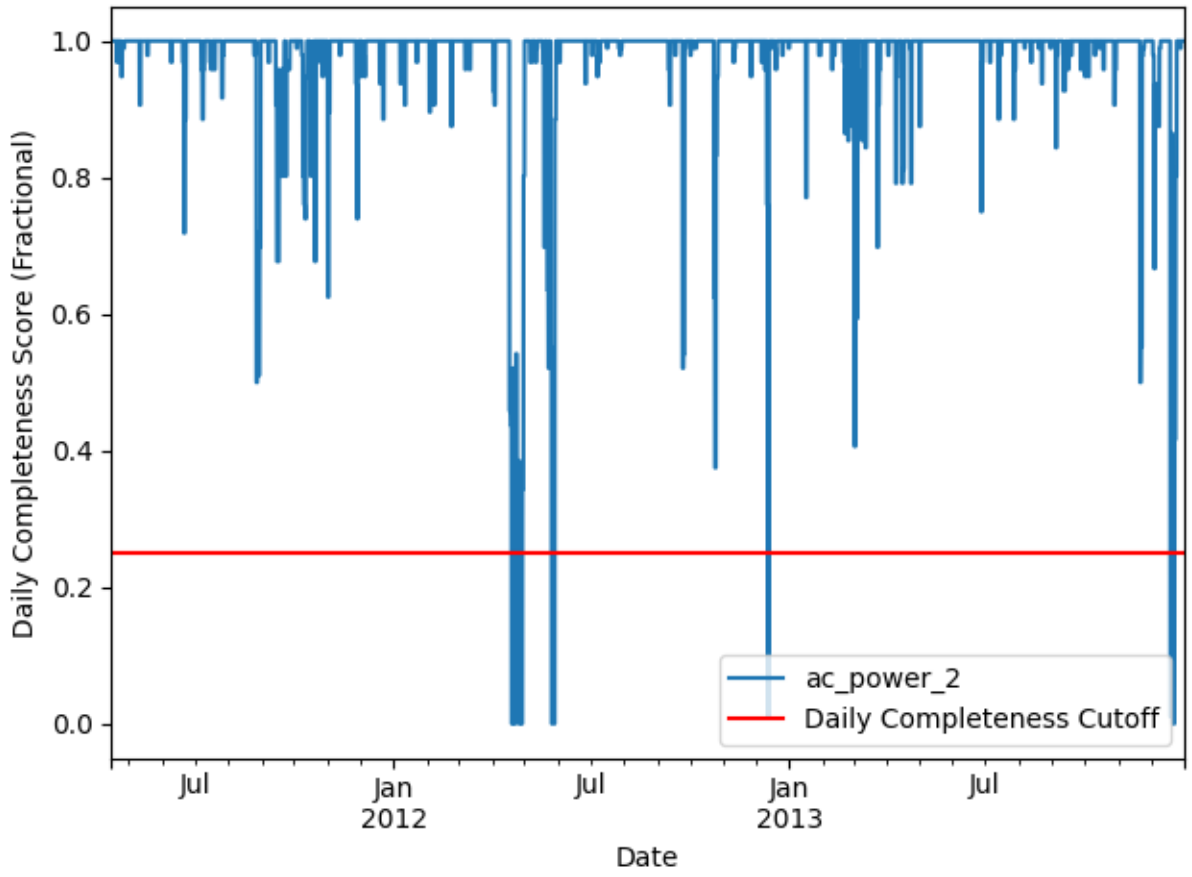
We filter the time series based on its daily completeness score. This filtering scheme requires at least 25% of data to be present for each day to be included. We further require at least 10 consecutive days meeting this 25% threshold to be included.

```
# Visualize daily data completeness
data_completeness_score = gaps.completeness_score(time_series)

# Visualize data completeness score as a time series.
data_completeness_score.plot()
plt.xlabel("Date")
plt.ylabel("Daily Completeness Score (Fractional)")
plt.axhline(y=0.25, color='r', linestyle='-',
            label='Daily Completeness Cutoff')
plt.legend()
plt.tight_layout()
plt.show()

# Trim the series based on daily completeness score
trim_series = pvanalytics.quality.gaps.trim_incomplete(
    time_series, minimum_completeness=.25, freq=data_freq)
first_valid_date, last_valid_date = \
    pvanalytics.quality.gaps.start_stop_dates(trim_series)
time_series = time_series[first_valid_date.tz_convert(time_series.index.tz):
                          last_valid_date.tz_convert(time_series.index.tz)]
```

(continues on next page)

(continued from previous page)

```
time_series = time_series.asfreq(data_freq)
```



Next, we check the time series for any time shifts, which may be caused by time drift or by incorrect time zone assignment. To do this, we compare the modelled midday time for the particular system location to its measured midday time. We use *pvanalytics.quality.gaps.stale_values_round()*) to determine the presence of time shifts in the series.

```
# Plot the heatmap of the AC power time series before time shift correction.
plt.figure()
# Get time of day from the associated datetime column
time_of_day = pd.Series(time_series.index.hour +
                        time_series.index.minute/60,
                        index=time_series.index)
# Pivot the dataframe
dataframe = pd.DataFrame(pd.concat([time_series, time_of_day], axis=1))
dataframe.columns = ["values", 'time_of_day']
dataframe = dataframe.dropna()
dataframe_pivoted = dataframe.pivot_table(index='time_of_day',
                                          columns=dataframe.index.date,
                                          values="values")
plt.pcolormesh(dataframe_pivoted.columns,
               dataframe_pivoted.index,
               dataframe_pivoted,
```

(continues on next page)

(continued from previous page)

```
               shading='auto')
plt.ylabel('Time of day [0-24]')
plt.xlabel('Date')
plt.xticks(rotation=60)
plt.title('Pre-Correction Heatmap, Time of Day')
plt.colorbar()
plt.tight_layout()
plt.show()

# Get the modeled sunrise and sunset time series based on the system's
# latitude-longitude coordinates
modeled_sunrise_sunset_df = pvlib.solarposition.sun_rise_set_transit_spa(
    time_series.index, latitude, longitude)

# Calculate the midday point between sunrise and sunset for each day
# in the modeled irradiance series
modeled_midday_series = modeled_sunrise_sunset_df['sunrise'] + \
    (modeled_sunrise_sunset_df['sunset'] -
     modeled_sunrise_sunset_df['sunrise']) / 2

# Run day-night mask on the irradiance time series
daytime_mask = power_or_irradiance(time_series,
                                   freq=data_freq,
                                   low_value_threshold=.005)

# Generate the sunrise, sunset, and halfway points for the data stream
sunrise_series = daytime.get_sunrise(daytime_mask)
sunset_series = daytime.get_sunset(daytime_mask)
midday_series = sunrise_series + ((sunset_series - sunrise_series)/2)

# Convert the midday and modeled midday series to daily values
midday_series_daily, modeled_midday_series_daily = (
    midday_series.resample('D').mean(),
    modeled_midday_series.resample('D').mean())

# Set midday value series as minutes since midnight, from midday datetime
# values
midday_series_daily = (midday_series_daily.dt.hour * 60 +
                       midday_series_daily.dt.minute +
                       midday_series_daily.dt.second / 60)
modeled_midday_series_daily = \
    (modeled_midday_series_daily.dt.hour * 60 +
     modeled_midday_series_daily.dt.minute +
     modeled_midday_series_daily.dt.second / 60)

# Estimate the time shifts by comparing the modelled midday point to the
# measured midday point.
is_shifted, time_shift_series = shifts_ruptures(modeled_midday_series_daily,
                                                midday_series_daily,
                                                period_min=15,
                                                shift_min=15,
                                                zscore_cutoff=1.5)
```

(continues on next page)

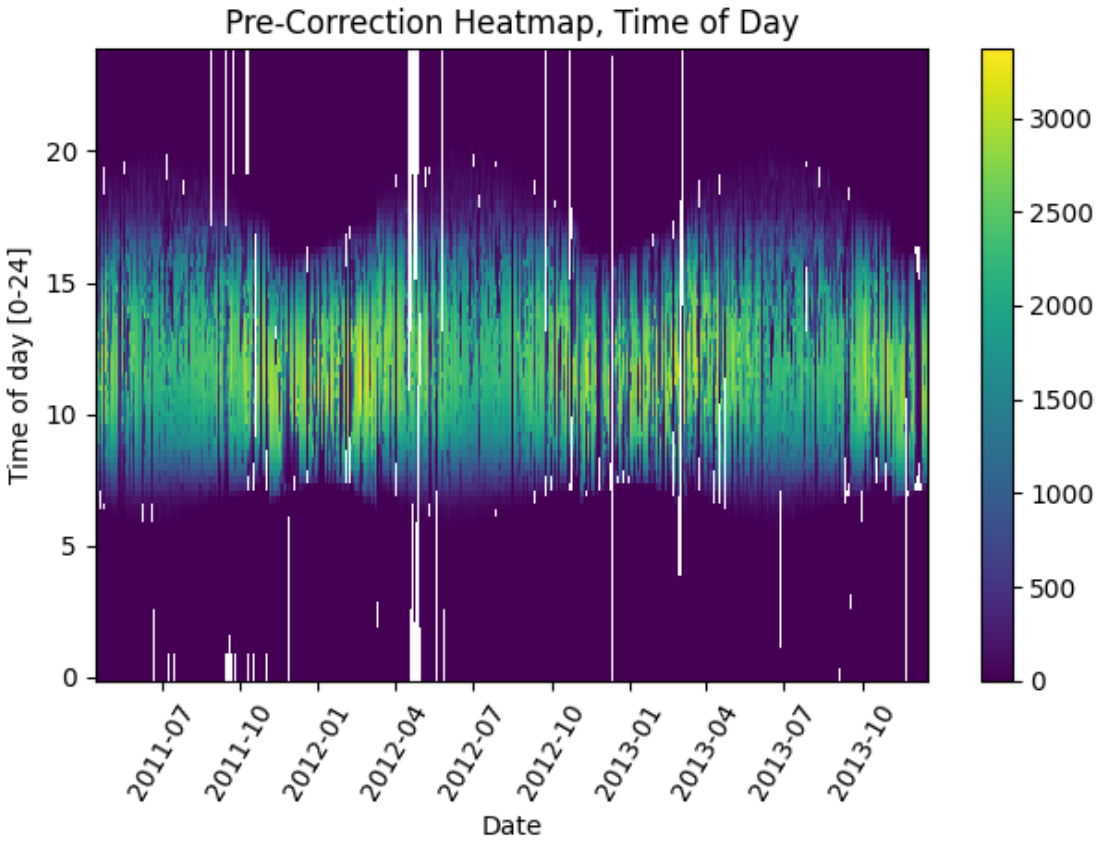
(continued from previous page)

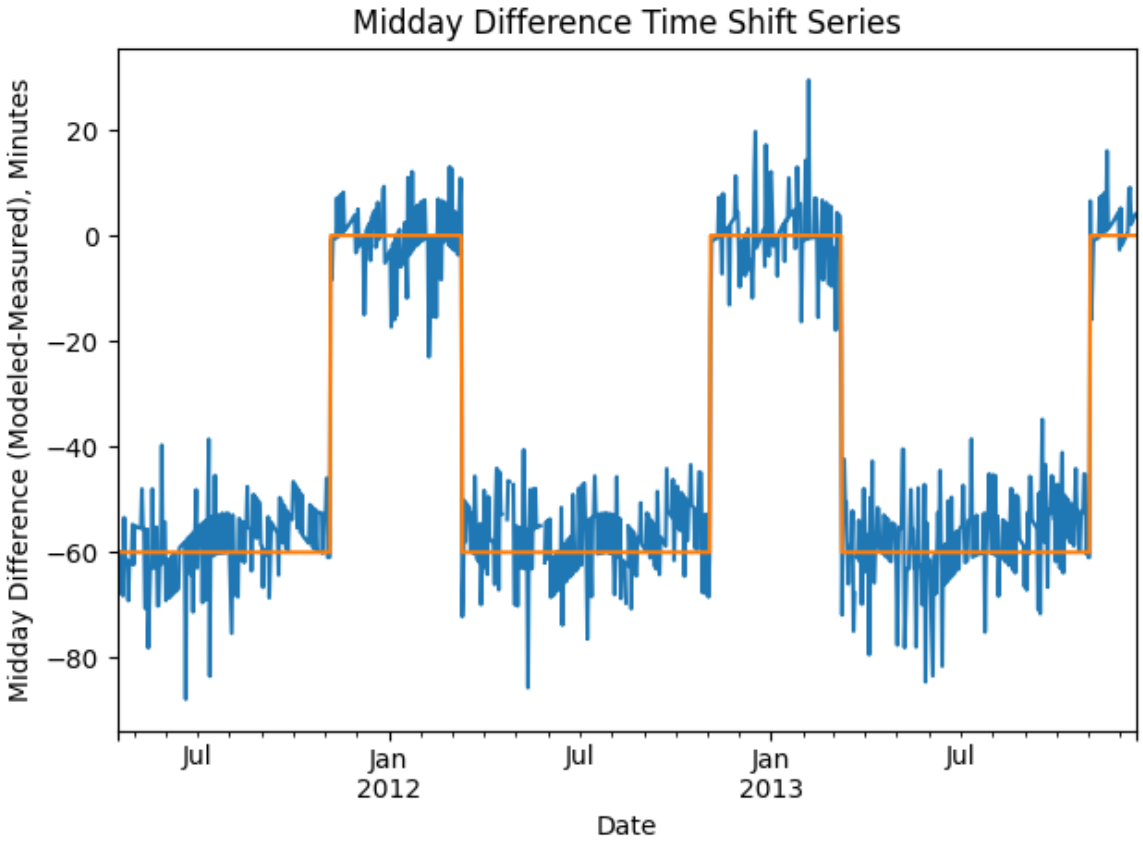
```
# Create a midday difference series between modeled and measured midday, to
# visualize time shifts. First, resample each time series to daily frequency,
# and compare the data stream's daily halfway point to the modeled halfway
# point
midday_diff_series = (modeled_midday_series.resample('D').mean() -
                      midday_series.resample('D').mean()
                      ).dt.total_seconds() / 60

# Generate boolean for detected time shifts
if any(time_shift_series != 0):
    time_shifts_detected = True
else:
    time_shifts_detected = False

# Build a list of time shifts for re-indexing. We choose to use dicts.
time_shift_series.index = pd.to_datetime(
    time_shift_series.index)
changepoints = (time_shift_series != time_shift_series.shift(1))
changepoints = changepoints[changepoints].index
changepoint_amts = pd.Series(time_shift_series.loc[changepoints])
time_shift_list = list()
for idx in range(len(changepoint_amts)):
    if idx < (len(changepoint_amts) - 1):
        time_shift_list.append({"datetime_start":
                                str(changepoint_amts.index[idx]),
                                "datetime_end":
                                    str(changepoint_amts.index[idx + 1]),
                                "time_shift": changepoint_amts[idx]})
    else:
        time_shift_list.append({"datetime_start":
                                str(changepoint_amts.index[idx]),
                                "datetime_end":
                                    str(time_shift_series.index.max()),
                                "time_shift": changepoint_amts[idx]})

# Correct any time shifts in the time series
new_index = pd.Series(time_series.index, index=time_series.index)
for i in time_shift_list:
    new_index[(time_series.index >= pd.to_datetime(i['datetime_start'])) &
              (time_series.index < pd.to_datetime(i['datetime_end']))] = \
        time_series.index + pd.Timedelta(minutes=i['time_shift'])
time_series.index = new_index

# Remove duplicated indices and sort the time series (just in case)
time_series = time_series[~time_series.index.duplicated(
    keep='first')].sort_index()

# Plot the difference between measured and modeled midday, as well as the
# CPD-estimated time shift series.
plt.figure()
midday_diff_series.plot()
```

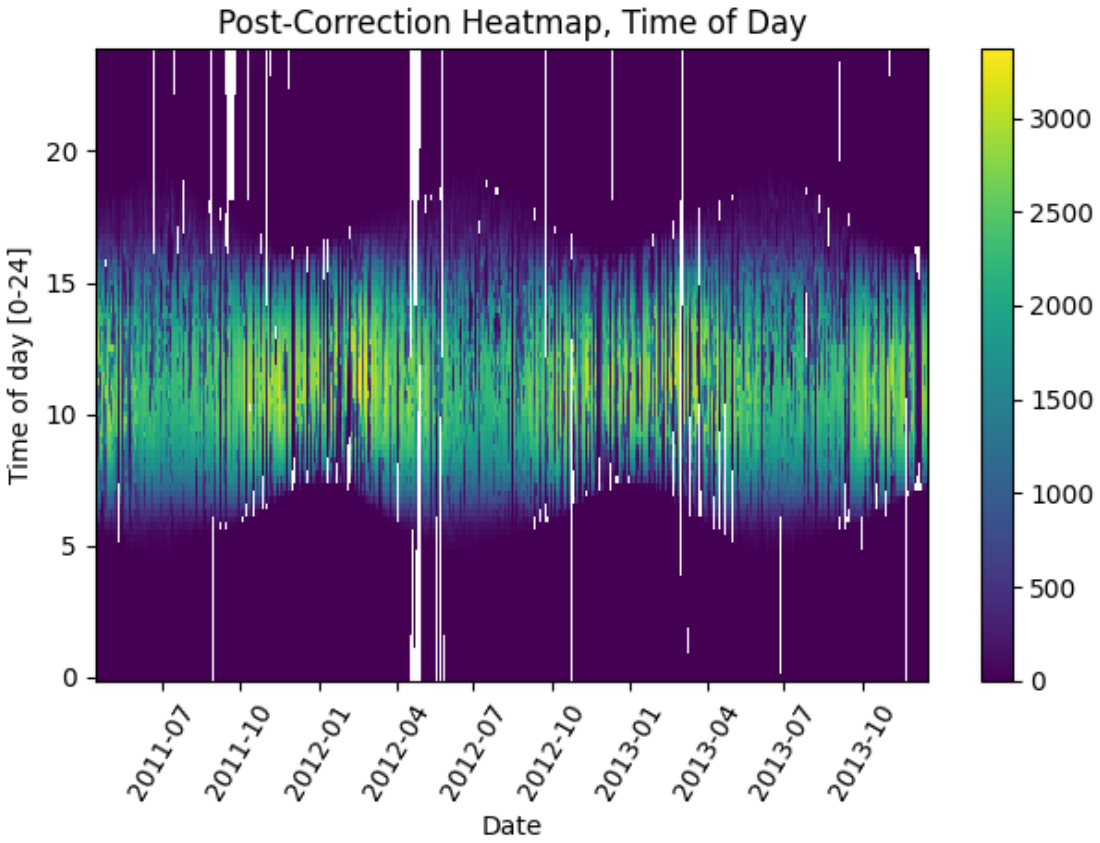
(continues on next page)

(continued from previous page)

```
time_shift_series.plot()
plt.title("Midday Difference Time Shift Series")
plt.xlabel("Date")
plt.ylabel("Midday Difference (Modeled-Measured), Minutes")
plt.tight_layout()
plt.show()

# Plot the heatmap of the irradiance time series
plt.figure()
# Get time of day from the associated datetime column
time_of_day = pd.Series(time_series.index.hour +
                        time_series.index.minute/60,
                        index=time_series.index)
# Pivot the dataframe
dataframe = pd.DataFrame(pd.concat([time_series, time_of_day], axis=1))
dataframe.columns = ["values", 'time_of_day']
dataframe = dataframe.dropna()
dataframe_pivoted = dataframe.pivot_table(index='time_of_day',
                                          columns=dataframe.index.date,
                                          values="values")
plt.pcolormesh(dataframe_pivoted.columns,
               dataframe_pivoted.index,
               dataframe_pivoted,
               shading='auto')
plt.ylabel('Time of day [0-24]')
plt.xlabel('Date')
plt.xticks(rotation=60)
plt.title('Post-Correction Heatmap, Time of Day')
plt.colorbar()
plt.tight_layout()
plt.show()
```


Pre-Correction Heatmap, Time of Day
Time of day [0-24]
0
5
10
15
20
2011-07
2011-10
2012-01
2012-04
2012-07
2012-10
2013-01
2013-04
2013-07
2013-10
Date
3000
2500
2000
1500
1000
500
0


Midday Difference Time Shift Series
Midday Difference (Modeled-Measured), Minutes
20
0
−20
−40
−60
−80
Jul
Jan
2012
Jul
Jan
2013
Jul
Date



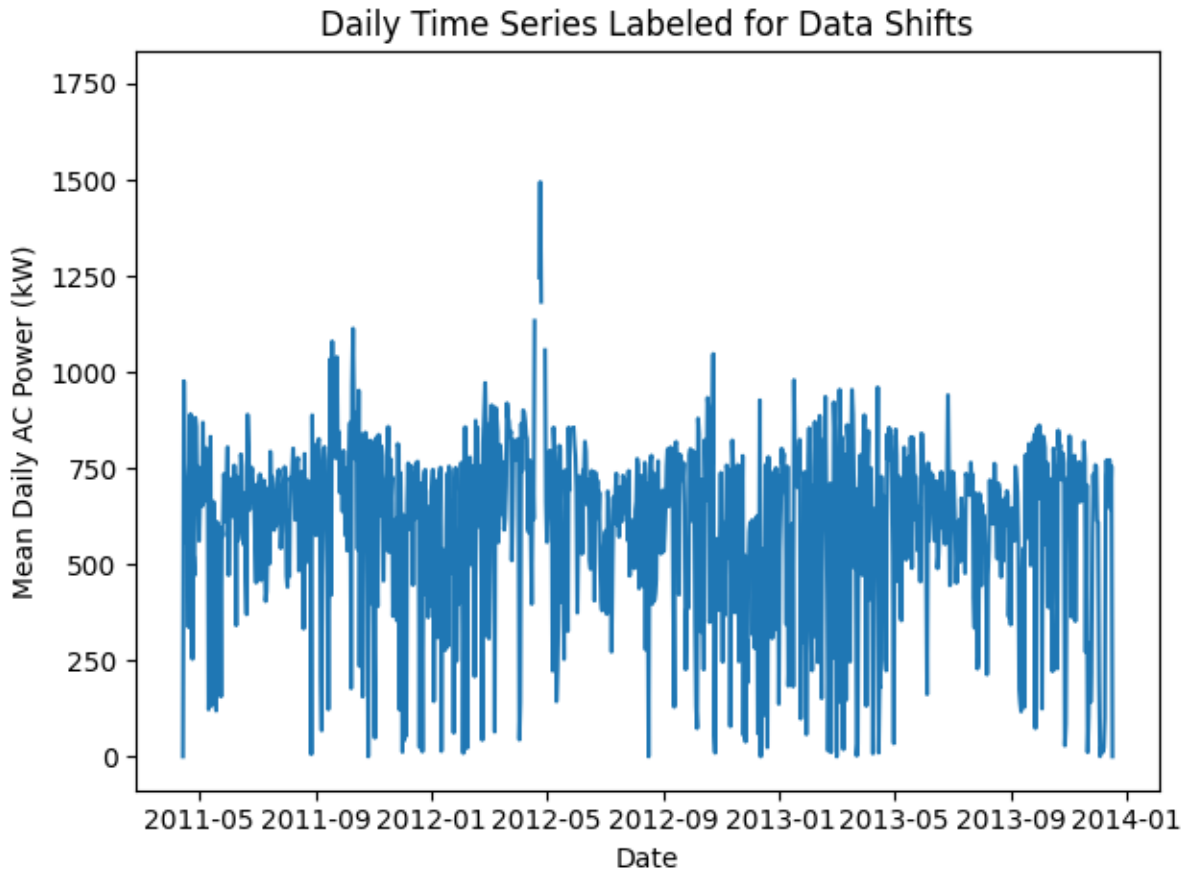
Next, we check the time series for any abrupt data shifts. We take the longest continuous part of the time series that is free of data shifts. We use *pvanalytics.quality.data_shifts.detect_data_shifts()* to detect data shifts in the time series.

```
# Resample the time series to daily mean
time_series_daily = time_series.resample('D').mean()
data_shift_start_date, data_shift_end_date = \
    ds.get_longest_shift_segment_dates(time_series_daily)
data_shift_period_length = (data_shift_end_date -
                            data_shift_start_date).days

# Get the number of shift dates
data_shift_mask = ds.detect_data_shifts(time_series_daily)
# Get the shift dates
shift_dates = list(time_series_daily[data_shift_mask].index)
if len(shift_dates) > 0:
    shift_found = True
else:
    shift_found = False

# Visualize the time shifts for the daily time series
print("Shift Found: ", shift_found)
edges = ([time_series_daily.index[0]] + shift_dates +
         [time_series_daily.index[-1]])
fig, ax = plt.subplots()
```

(continues on next page)

(continued from previous page)

```
for (st, ed) in zip(edges[:-1], edges[1:]):
    ax.plot(time_series_daily.loc[st:ed])
plt.title("Daily Time Series Labeled for Data Shifts")
plt.xlabel("Date")
plt.ylabel("Mean Daily AC Power (kW)")
plt.tight_layout()
plt.show()
```




```
Shift Found:  False
```

Use logic-based and ML-based clipping functions to identify clipped periods in the time series data, and plot the filtered data.

```
# REMOVE CLIPPING PERIODS
clipping_mask = geometric(ac_power=time_series,
                          freq=data_freq)

# Get the pct clipping
clipping_mask.dropna(inplace=True)
pct_clipping = round(100*(len(clipping_mask[
    clipping_mask])/len(clipping_mask)), 4)
if pct_clipping >= 0.5:
```

(continues on next page)

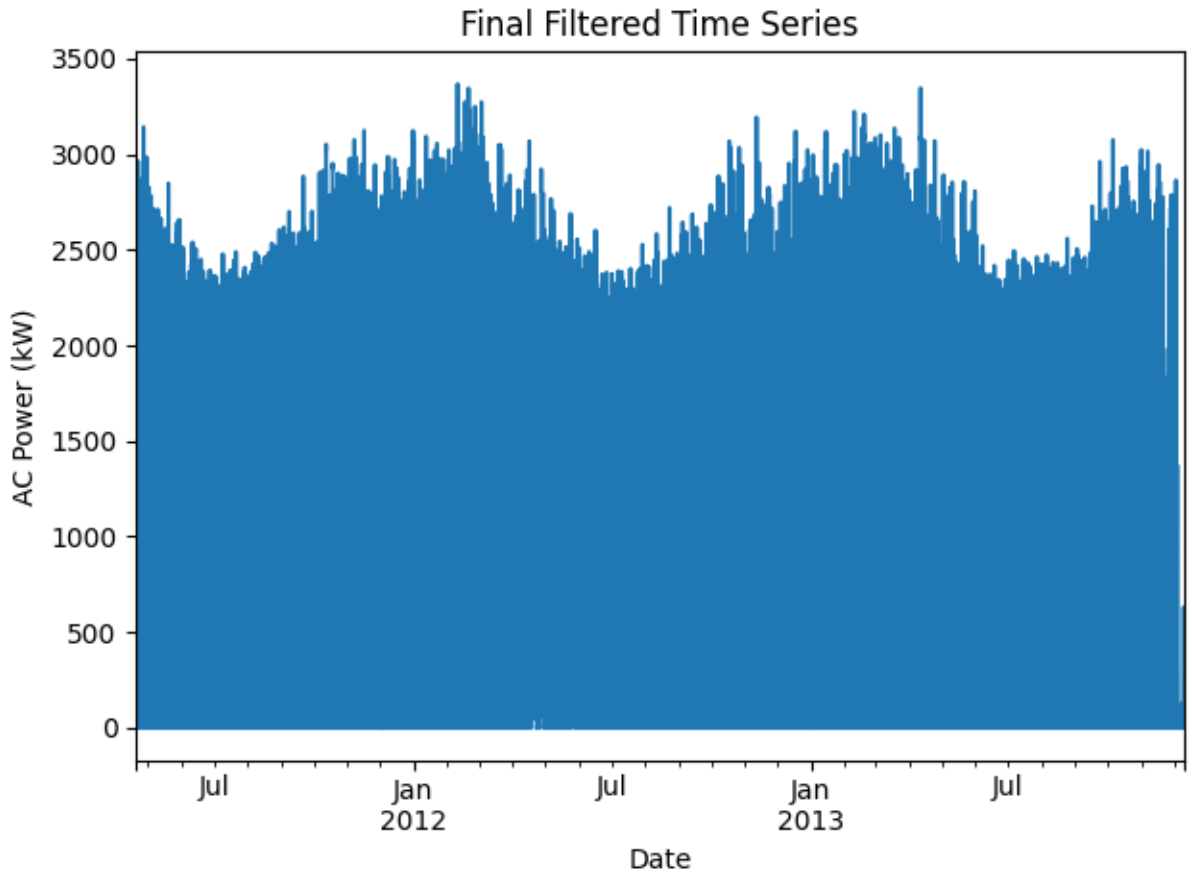
(continued from previous page)

```
    clipping = True
    clip_pwr = time_series[clipping_mask].median()
else:
    clipping = False
    clip_pwr = None

if clipping:
    # Plot the time series with clipping labeled
    time_series.plot()
    time_series.loc[clipping_mask].plot(ls='', marker='o')
    plt.legend(labels=["AC Power", "Clipping"],
               title="Clipped")
    plt.title("Time Series Labeled for Clipping")
    plt.xticks(rotation=20)
    plt.xlabel("Date")
    plt.ylabel("AC Power (kW)")
    plt.tight_layout()
    plt.show()
    plt.close()
else:
    print("No clipping detected!!!")
```

```
No clipping detected!!!
```

We filter the time series to only include the longest shift-free period. We then visualize the final time series post-QA filtering.

```
time_series = time_series[
    (time_series.index >=
     data_shift_start_date.tz_convert(time_series.index.tz)) &
    (time_series.index <=
     data_shift_end_date.tz_convert(time_series.index.tz))]

time_series = time_series.asfreq(data_freq)

# Plot the final filtered time series.
time_series.plot(title="Final Filtered Time Series")
plt.xlabel("Date")
plt.ylabel("AC Power (kW)")
plt.tight_layout()
plt.show()
```



Estimate the azimuth and tilt of the system, based on the power series data. The ground truth azimuth and tilt for this system are 158 and 45 degrees, respectively.

```
# Import the PSM3 data. This data is pulled via the following function in
# PVLib: :py:func:`pvlib.iotools.get_psm3`
file = pvanalytics_dir / 'data' / 'system_50_ac_power_2_full_DST_psm3.parquet'
psm3 = pd.read_parquet(file)
psm3.set_index('index', inplace=True)
psm3.index = pd.to_datetime(psm3.index)

psm3 = psm3.reindex(pd.date_range(psm3.index[0],
                                  psm3.index[-1],
                                  freq=data_freq)).interpolate()
psm3.index = psm3.index.tz_convert(time_series.index.tz)
psm3 = psm3.reindex(time_series.index)
is_clear = (psm3.ghi_clear == psm3.ghi)
is_daytime = (psm3.ghi > 0)

# Trim based on clearsky and daytime values
time_series_clearsky = time_series.reindex(is_daytime.index)[
    (is_clear) & (is_daytime)].dropna()

# Get final PSM3 data
psm3_clearsky = psm3.loc[time_series_clearsky.index]
```

(continues on next page)

(continued from previous page)

```
solpos_clearsky = pvlib.solarposition.get_solarposition(
    time_series_clearsky.index, latitude, longitude)

# Estimate the azimuth and tilt using PVWatts-based method
predicted_tilt, predicted_azimuth, r2 = infer_orientation_fit_pvwatts(
    time_series_clearsky,
    psm3_clearsky.ghi_clear,
    psm3_clearsky.dhi_clear,
    psm3_clearsky.dni_clear,
    solpos_clearsky.zenith,
    solpos_clearsky.azimuth,
    temperature=psm3_clearsky.temp_air,
    azimuth_min=90,
    azimuth_max=275)

print("Predicted azimuth: " + str(predicted_azimuth))
print("Predicted tilt: " + str(predicted_tilt))
```

```
Predicted azimuth: 161.27396222501486
Predicted tilt: 42.20903060025887
```

Look at the daily power profile for summer and winter months, and identify if the data stream is associated with a fixed-tilt or single-axis tracking system.

```
# CHECK MOUNTING CONFIGURATION
daytime_mask = power_or_irradiance(time_series)
predicted_mounting_config = is_tracking_envelope(
    time_series,
    daytime_mask,
    clipping_mask.reindex(index=time_series.index))

print("Predicted Mounting configuration:")
print(predicted_mounting_config.name)
```

```
Predicted Mounting configuration:
FIXED
```

Generate a dictionary output for the QA assessment of this data stream, including the percent stale and erroneous data detected, any shift dates, time shift dates, clipping information, and estimated mounting configuration.

```
qa_check_dict = {"original_time_zone_offset": time_series.index.tz,
                 "pct_stale": pct_stale,
                 "pct_negative": pct_negative,
                 "pct_erroneous": pct_erroneous,
                 "pct_outlier": pct_outlier,
                 "time_shifts_detected": time_shifts_detected,
                 "time_shift_list": time_shift_list,
                 "data_shifts": shift_found,
                 "shift_dates": shift_dates,
                 "clipping": clipping,
                 "clipping_threshold": clip_pwr,
                 "pct_clipping": pct_clipping,
```

(continues on next page)

(continued from previous page)

```
                  "mounting_config": predicted_mounting_config.name,
                  "predicted_azimuth": predicted_azimuth,
                  "predicted_tilt": predicted_tilt}

print("QA Results:")
print(qa_check_dict)
```

```
QA Results:
{'original_time_zone_offset': pytz.FixedOffset(-420), 'pct_stale': 0.7, 'pct_negative':␣
↪0.0, 'pct_erroneous': 0.0, 'pct_outlier': 0.0, 'time_shifts_detected': True, 'time_
↪shift_list': [{'datetime_start': '2011-04-15 00:00:00-07:00', 'datetime_end': '2011-11-
↪06 00:00:00-07:00', 'time_shift': -60.0}, {'datetime_start': '2011-11-06 00:00:00-07:00
↪', 'datetime_end': '2012-03-11 00:00:00-07:00', 'time_shift': 0.0}, {'datetime_start':
↪'2012-03-11 00:00:00-07:00', 'datetime_end': '2012-11-04 00:00:00-07:00', 'time_shift
↪': -60.0}, {'datetime_start': '2012-11-04 00:00:00-07:00', 'datetime_end': '2013-03-10␣
↪00:00:00-07:00', 'time_shift': 0.0}, {'datetime_start': '2013-03-10 00:00:00-07:00',
↪'datetime_end': '2013-11-03 00:00:00-07:00', 'time_shift': -60.0}, {'datetime_start':
↪'2013-11-03 00:00:00-07:00', 'datetime_end': '2013-12-17 00:00:00-07:00', 'time_shift
↪': 0.0}], 'data_shifts': False, 'shift_dates': [], 'clipping': False, 'clipping_
↪threshold': None, 'pct_clipping': 0.0842, 'mounting_config': 'FIXED', 'predicted_
↪azimuth': 161.27396222501486, 'predicted_tilt': 42.20903060025887}
```

Total running time of the script: (0 minutes 23.523 seconds)

Data/Time Shifts

This includes examples for identifying data/capacity/time shifts in time series data.

Data Shift Detection & Filtering

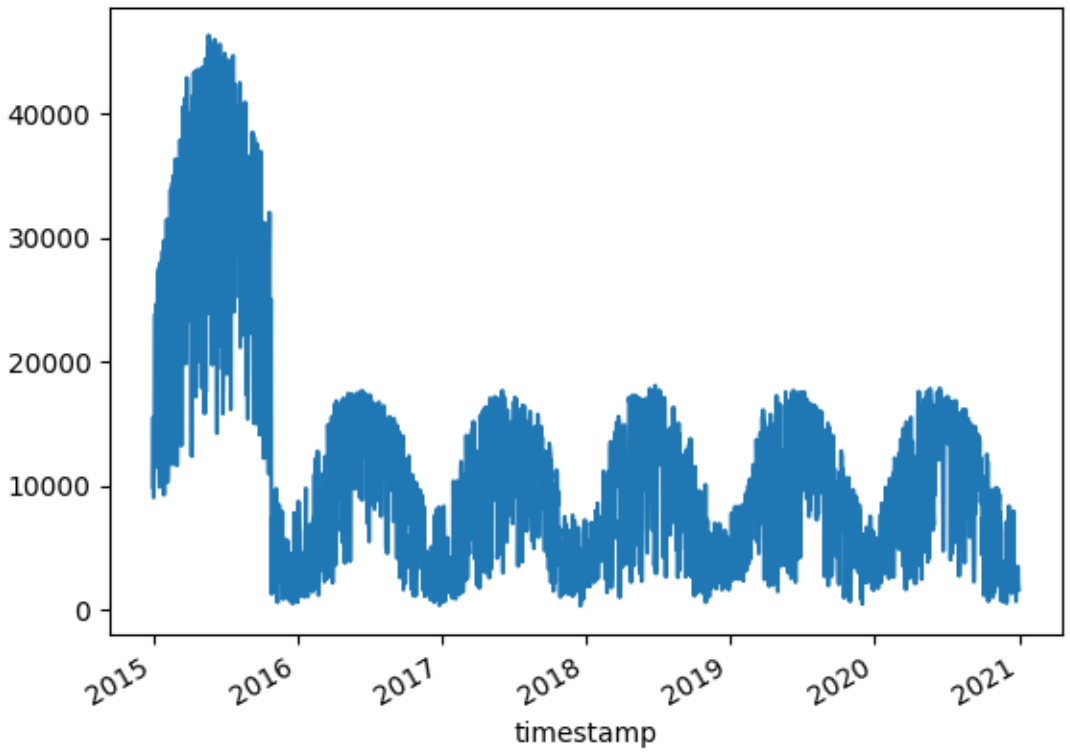
Identifying data shifts/capacity changes in time series data

This example covers identifying data shifts/capacity changes in a time series and extracting the longest time series segment free of these shifts, using *pvanalytics.quality.data_shifts.detect_data_shifts()* and *pvanalytics.quality.data_shifts.get_longest_shift_segment_dates()*.

```
import pvanalytics
import pandas as pd
import matplotlib.pyplot as plt
from pvanalytics.quality import data_shifts as ds
import pathlib
```

As an example, we load in a simulated pvlib AC power time series with a single changepoint, occurring on October 28, 2015.

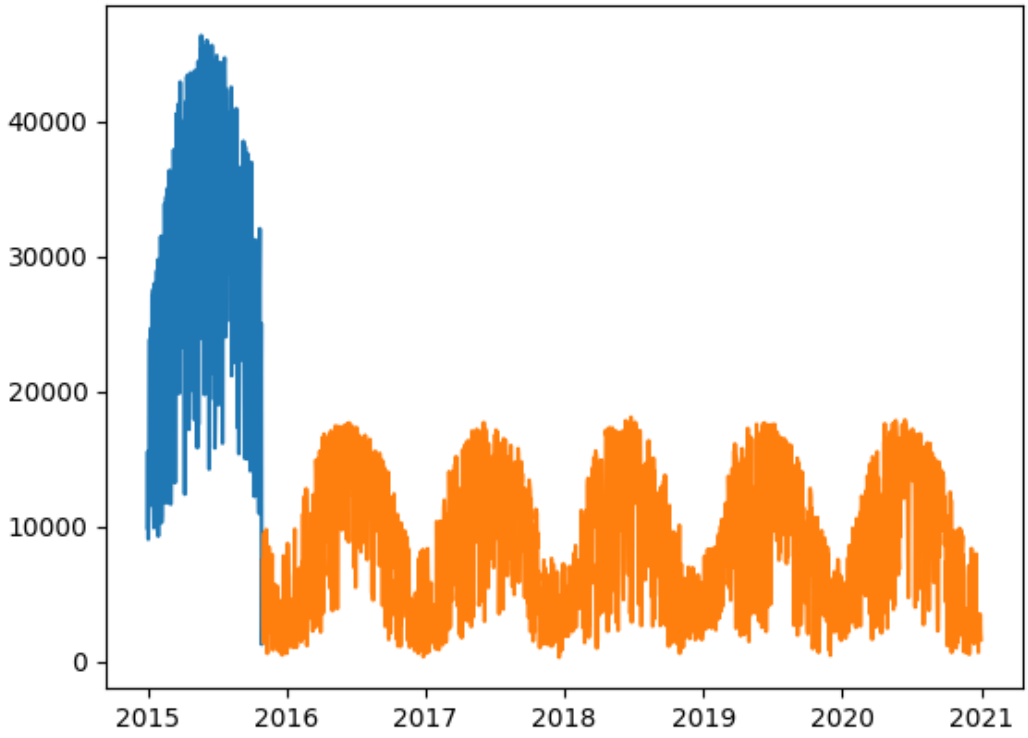
```
pvanalytics_dir = pathlib.Path(pvanalytics.__file__).parent
data_shift_file = pvanalytics_dir / 'data' / 'pvlib_data_shift.csv'
df = pd.read_csv(data_shift_file)
df.index = pd.to_datetime(df['timestamp'])
df['value'].plot()
print("Changepoint at: " + str(df[df['label'] == 1].index[0]))
```

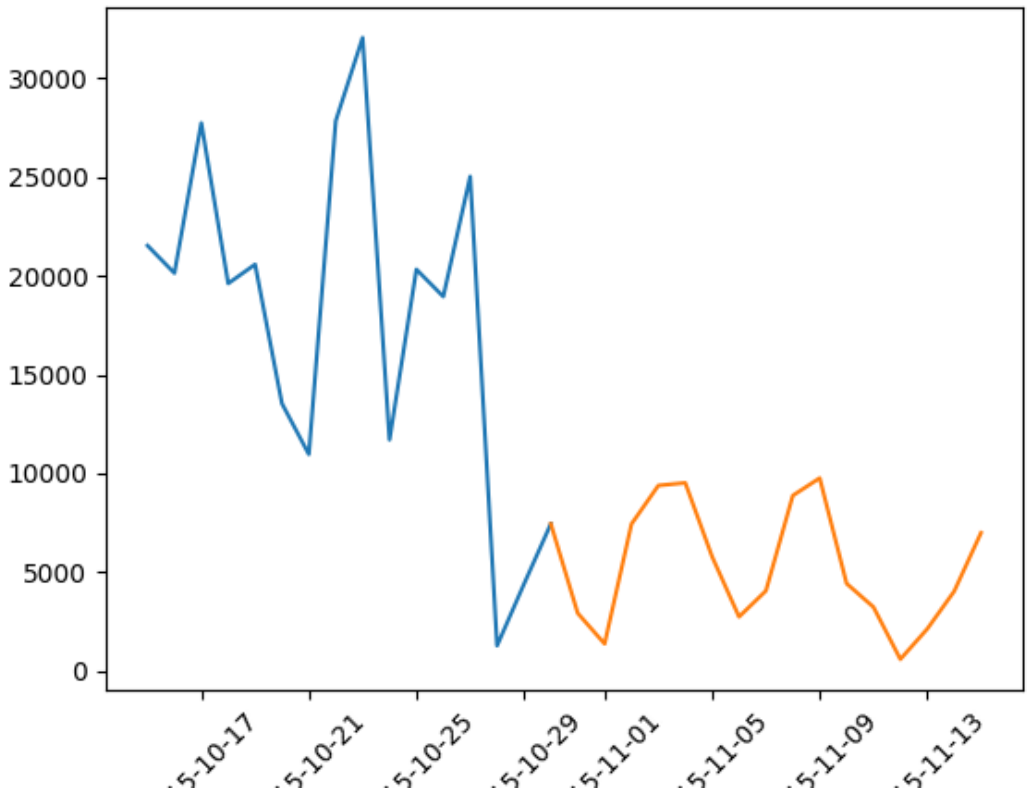



```
Changepoint at: 2015-10-28 00:00:00
```

Now we run the data shift algorithm (with default parameters) on the data stream, using *pvanalytics.quality.data_shifts.detect_data_shifts()*. We plot the predicted time series segments, based on algorithm results.

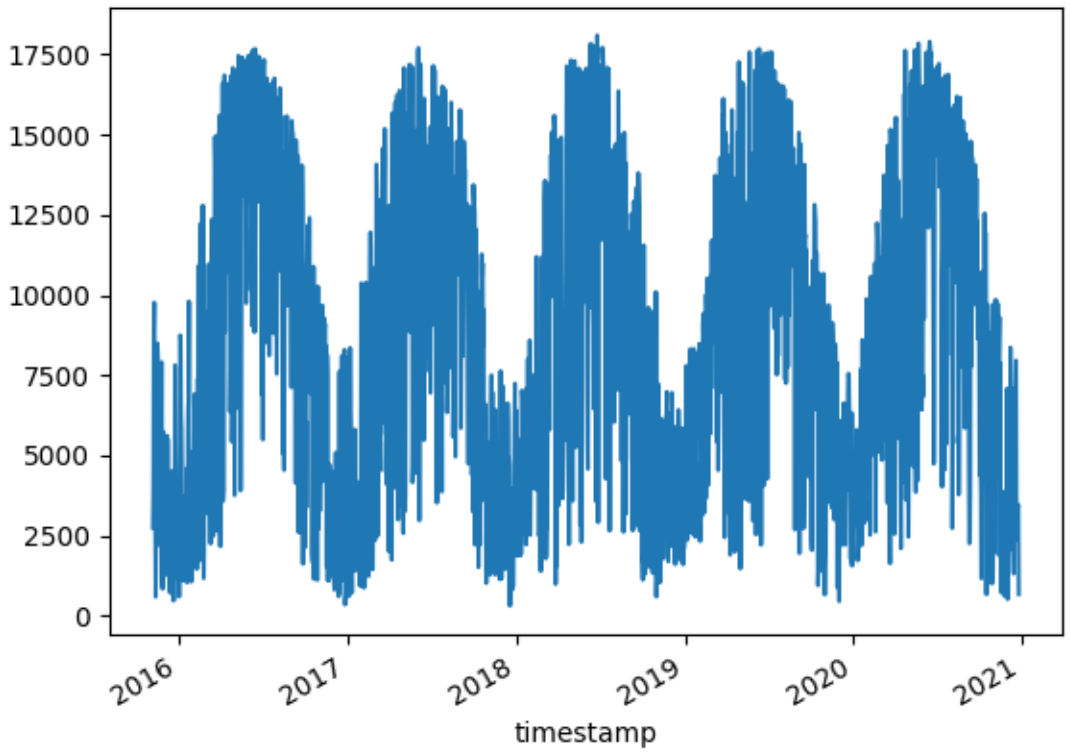
```
shift_mask = ds.detect_data_shifts(df['value'])
shift_list = list(df[shift_mask].index)
edges = [df.index[0]] + shift_list + [df.index[-1]]
fig, ax = plt.subplots()
for (st, ed) in zip(edges[:-1], edges[1:]):
    ax.plot(df.loc[st:ed, "value"])
plt.show()

# We zoom in around the changepoint to more closely show the data shift. Time
# series segments pre- and post-shift are color-coded.

edges = [pd.to_datetime("10-15-2015")] + shift_list + \
    [pd.to_datetime("11-15-2015")]
fig, ax = plt.subplots()
for (st, ed) in zip(edges[:-1], edges[1:]):
    ax.plot(df.loc[st:ed, "value"])
plt.xticks(rotation=45)
plt.show()
```


40000
30000
20000
10000
0
2015
2016
2017
2018
2019
2020
2021



We filter the time series by the detected changepoints, taking the longest continuous segment free of data shifts, using *pvanalytics.quality.data_shifts.get_longest_shift_segment_dates()*. The trimmed time series is then plotted.

```
start_date, end_date = ds.get_longest_shift_segment_dates(df['value'])
df['value'][start_date:end_date].plot()
plt.show()
```



Total running time of the script: (0 minutes 1.057 seconds)

Identifying and estimating time shifts

Identifying time shifts from clock errors or uncorrected Daylight Saving Time.

Time shifts can occur in measured data due to clock errors and time zone issues (for example, assuming a dataset is in local standard time when in fact it contains Daylight Saving Time).

This example uses *shifts_ruptures()* to identify abrupt time shifts in a time series, and estimate the corresponding time shift amount.

```
import pvlib
import pandas as pd
from pvanalytics.quality.time import shifts_ruptures
from pvanalytics.features.daytime import (power_or_irradiance,
                                          get_sunrise, get_sunset)
import matplotlib.pyplot as plt
```

Typically this process would be applied to measured data with possibly untrustworthy timestamps. However, for instructional purposes here, we'll create an artificial example dataset that contains a time shift due to DST.

```
# use a time zone (US/Eastern) that is affected by DST.
# Etc/GMT+5 is the corresponding local standard time zone.
```

(continues on next page)

(continued from previous page)

```
times = pd.date_range('2019-01-01', '2019-12-31', freq='5T', tz='US/Eastern')
location = pvlib.location.Location(40, -80)
cs = location.get_clearsky(times)
measured_signal = cs['ghi']
```

The *shifts_ruptures()* function is centered around comparing the timing of events observed in the measured data with expected timings for those same events. In this case, we'll use the timing of solar noon as the event.

First, we'll extract the timing of solar noon from the measured data. This could be done in several ways; here we will just take the midpoint between sunrise and sunset using times estimated with *power_or_irradiance()*.

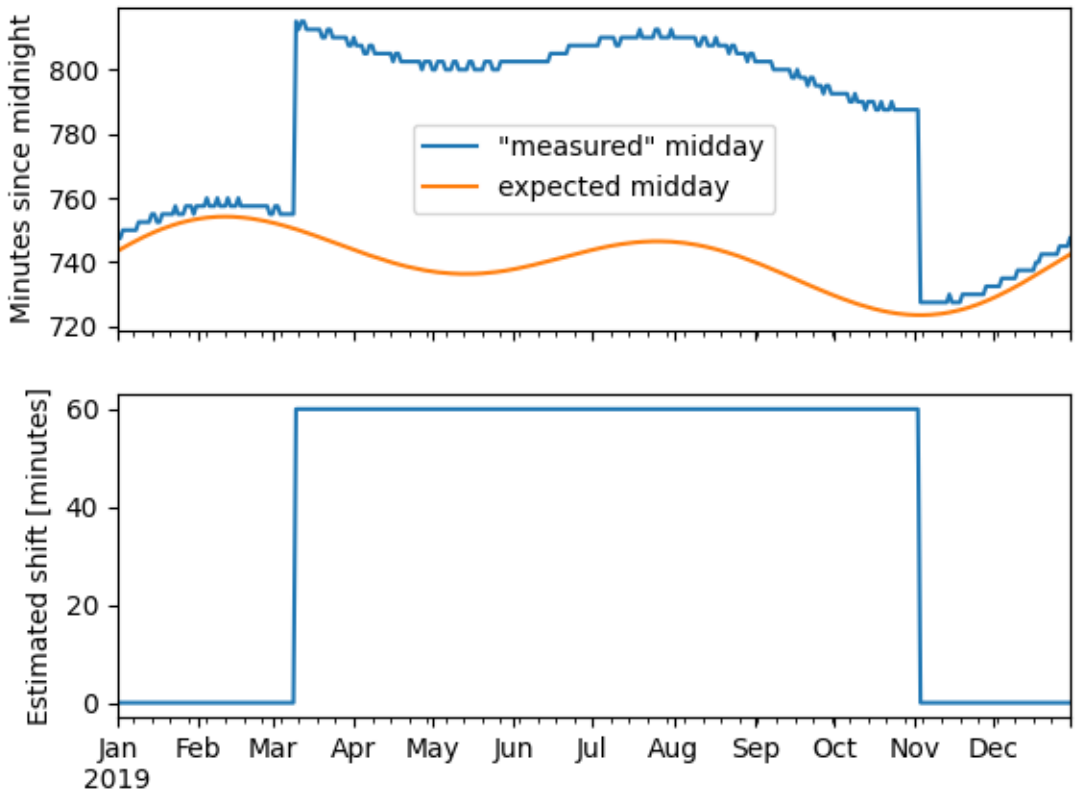
```
is_daytime = power_or_irradiance(measured_signal)
sunrise_timestamps = get_sunrise(is_daytime)
sunrise_timestamps = sunrise_timestamps.resample('d').first().dropna()
sunset_timestamps = get_sunset(is_daytime)
sunset_timestamps = sunset_timestamps.resample('d').first().dropna()


def ts_to_minutes(ts):
    # convert timestamps to minutes since midnight
    return ts.dt.hour * 60 + ts.dt.minute + ts.dt.second / 60


midday_minutes = (
    ts_to_minutes(sunrise_timestamps) + ts_to_minutes(sunset_timestamps)
) / 2
```

Now, calculate the expected timing of solar noon at this location for each day. Note that we use a time zone without DST for calculating the expected timings; this means that if the "measured" data does include DST in its timestamps, it will be flagged as a time shift.

```
dates = midday_minutes.index.tz_localize(None).tz_localize('Etc/GMT+5')
sp = location.get_sun_rise_set_transit(dates, method='spa')
transit_minutes = ts_to_minutes(sp['transit'])
```

Finally, ask ruptures if it sees any change points in the difference between these two daily event timings, and visualize the result:

```
is_shifted, shift_amount = shifts_ruptures(midday_minutes, transit_minutes)

fig, axes = plt.subplots(2, 1, sharex=True)

midday_minutes.plot(ax=axes[0], label='"measured" midday')
transit_minutes.plot(ax=axes[0], label='expected midday')
axes[0].set_ylabel('Minutes since midnight')
axes[0].legend()

shift_amount.plot(ax=axes[1])
axes[1].set_ylabel('Estimated shift [minutes]')
```




```
Text(47.09722222222214, 0.5, 'Estimated shift [minutes]')
```

Total running time of the script: (0 minutes 2.111 seconds)

System

This includes examples for system parameter estimation, including azimuth and tilt estimation, and determination if the system is fixed tilt or tracking.

Detect if a System is Tracking

Identifying if a system is tracking or fixed tilt

It is valuable to identify if a system is fixed tilt or tracking for future analysis. This example shows how to use *pvanalytics.system.is_tracking_envelope()* to determine if a system is tracking or not by fitting data to a maximum power or irradiance envelope, and fitting this envelope to quadratic and quartic curves. The r^2 output from these fits is used to determine if the system fits a tracking or fixed-tilt profile.

```
import pvanalytics
from pvanalytics.system import is_tracking_envelope
from pvanalytics.features.clipping import geometric
from pvanalytics.features.daytime import power_or_irradiance
```

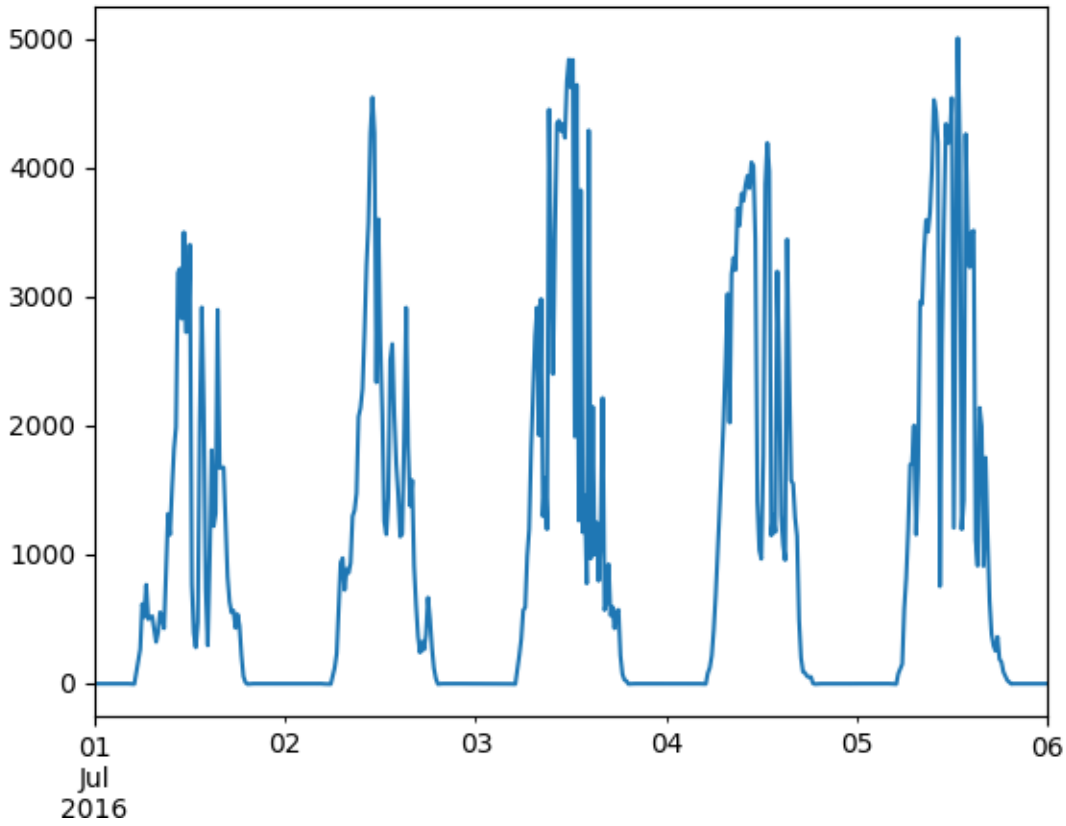
(continues on next page)

(continued from previous page)

```
import pandas as pd
import pathlib
import matplotlib.pyplot as plt
```

First, we import an AC power data stream from the SERF East site located at NREL. This data set is publicly available via the PVDAQ database in the DOE Open Energy Data Initiative (OEDI) (https://data.openei.org/submissions/4568), under system ID 50. This data is timezone-localized. This particular data stream is associated with a fixed-tilt system.

```
pvanalytics_dir = pathlib.Path(pvanalytics.__file__).parent
ac_power_file = pvanalytics_dir / 'data' / \
    'serf_east_15min_ac_power.csv'
data = pd.read_csv(ac_power_file, index_col=0, parse_dates=True)
data = data.sort_index()
time_series = data['ac_power']
time_series = time_series.asfreq('15min')

# Plot the first few days of the time series to visualize it
time_series[:pd.to_datetime("2016-07-06 00:00:00-07:00")].plot()
plt.show()
```



Run the clipping and the daytime filters on the time series. Both of these masks will be used as inputs to the *pvanalytics.system.is_tracking_envelope()* function.

```
# Generate the daylight mask for the AC power time series
daytime_mask = power_or_irradiance(time_series)

# Generate the clipping mask for the time series
clipping_mask = geometric(time_series)
```

Now, we use *pvanalytics.system.is_tracking_envelope()* to identify if the data stream is associated with a tracking or fixed-tilt system.

```
predicted_mounting_config = is_tracking_envelope(time_series,
                                                 daytime_mask,
                                                 clipping_mask)

print("Estimated mounting configuration: " + predicted_mounting_config.name)
```

```
Estimated mounting configuration: FIXED
```

Total running time of the script: (0 minutes 0.552 seconds)

Infer Array Tilt/Azimuth - PVWatts Method

Infer the azimuth and tilt of a system using PVWatts-based methods

Identifing and/or validating the azimuth and tilt information for a system is important, as these values must be correct for degradation and system yield analysis. This example shows how to use *pvanalytics.system.infer_orientation_fit_pvwatts()* to estimate a fixed-tilt system's azimuth and tilt, using the system's known latitude-longitude coordinates and an associated AC power time series.

```
import pvanalytics
import matplotlib.pyplot as plt
from pvanalytics import system
import pandas as pd
import pathlib
import pvlib
```

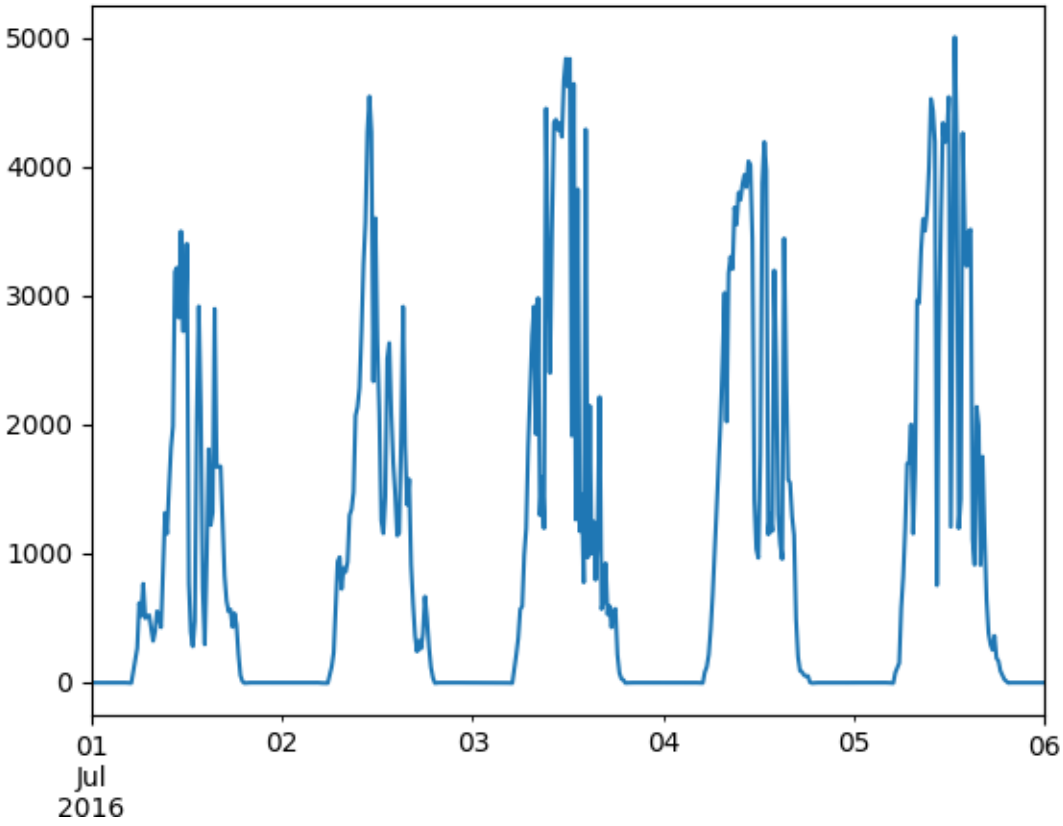
First, we import an AC power data stream from the SERF East site located at NREL. This data set is publicly available via the PVDAQ database in the DOE Open Energy Data Initiative (OEDI) (https://data.openei.org/submissions/4568), under system ID 50. This data is timezone-localized.

```
pvanalytics_dir = pathlib.Path(pvanalytics.__file__).parent
ac_power_file = pvanalytics_dir / 'data' / 'serf_east_15min_ac_power.csv'
data = pd.read_csv(ac_power_file, index_col=0, parse_dates=True)
data = data.sort_index()
time_series = data['ac_power']
time_series = time_series.asfreq('15min')

# Plot the first few days of the time series to visualize it
time_series[:pd.to_datetime("2016-07-06 00:00:00-07:00")].plot()
plt.show()

# Outline the ground truth metadata associated with the system
latitude = 39.742
```

(continues on next page)

(continued from previous page)

```
longitude = -105.1727
actual_azimuth = 158
actual_tilt = 45
```



Next, we import the PSM3 data generated via the `pvlib.iotools.get_psm3()` function, using site latitude-longitude coordinates. To generate the PSM3 data, you must first register for NREL's NSDRB API at the following link: https://developer.nrel.gov/signup/. PSM3 data can then be retrieved using `pvlib.iotools.get_psm3()`. The PSM3 data has been resampled to 15 minute intervals, to match the AC power data.

```
psm3_file = pvanalytics_dir / 'data' / 'serf_east_psm3_data.csv'
psm3 = pd.read_csv(psm3_file, index_col=0, parse_dates=True)
```

Filter the PSM3 data to only include clearsky periods

```
is_clear = (psm3.ghi_clear == psm3.ghi)
is_daytime = (psm3.ghi > 0)
time_series_clearsky = time_series[is_clear & is_daytime]
time_series_clearsky = time_series_clearsky.dropna()
psm3_clearsky = psm3.loc[time_series_clearsky.index]

# Get solar azimuth and zenith from pvlib, based on
# lat-long coords
solpos_clearsky = pvlib.solarposition.get_solarposition(
```

(continues on next page)

(continued from previous page)

```
    time_series_clearsky.index, latitude, longitude)
```

Run the pvlib data and the sensor-based time series data through the *pvanalytics.system.infer_orientation_fit_pvwatts()* function.

```
best_tilt, best_azimuth, r2 = system.infer_orientation_fit_pvwatts(
    time_series_clearsky,
    psm3_clearsky.ghi_clear,
    psm3_clearsky.dhi_clear,
    psm3_clearsky.dni_clear,
    solpos_clearsky.zenith,
    solpos_clearsky.azimuth,
    temperature=psm3_clearsky.temp_air,
)

# Compare actual system azimuth and tilt to predicted azimuth and tilt
print("Actual Azimuth: " + str(actual_azimuth))
print("Predicted Azimuth: " + str(best_azimuth))
print("Actual Tilt: " + str(actual_tilt))
print("Predicted Tilt: " + str(best_tilt))
```

```
Actual Azimuth: 158
Predicted Azimuth: 162.01767819075172
Actual Tilt: 45
Predicted Tilt: 42.14660650908418
```

Total running time of the script: (0 minutes 2.359 seconds)

Weather

This includes examples for weather quality checks.

Weather Limits

Identifying weather values that are within limits.

Identifying weather values that are within logical, expected limits and filtering data outside of these limits allows for more accurate future data analysis. In this example, we demonstrate how to use *pvanalytics.quality.weather.wind_limits()*, *pvanalytics.quality.weather.temperature_limits()*, and *pvanalytics.quality.weather.relative_humidity_limits()* to identify and filter out values that are not within expected limits, for wind speed, ambient temperature, and relative humidity, respectively.

```
import pvanalytics
from pvanalytics.quality.weather import wind_limits, \
    temperature_limits, relative_humidity_limits
import matplotlib.pyplot as plt
import pandas as pd
import pathlib
```

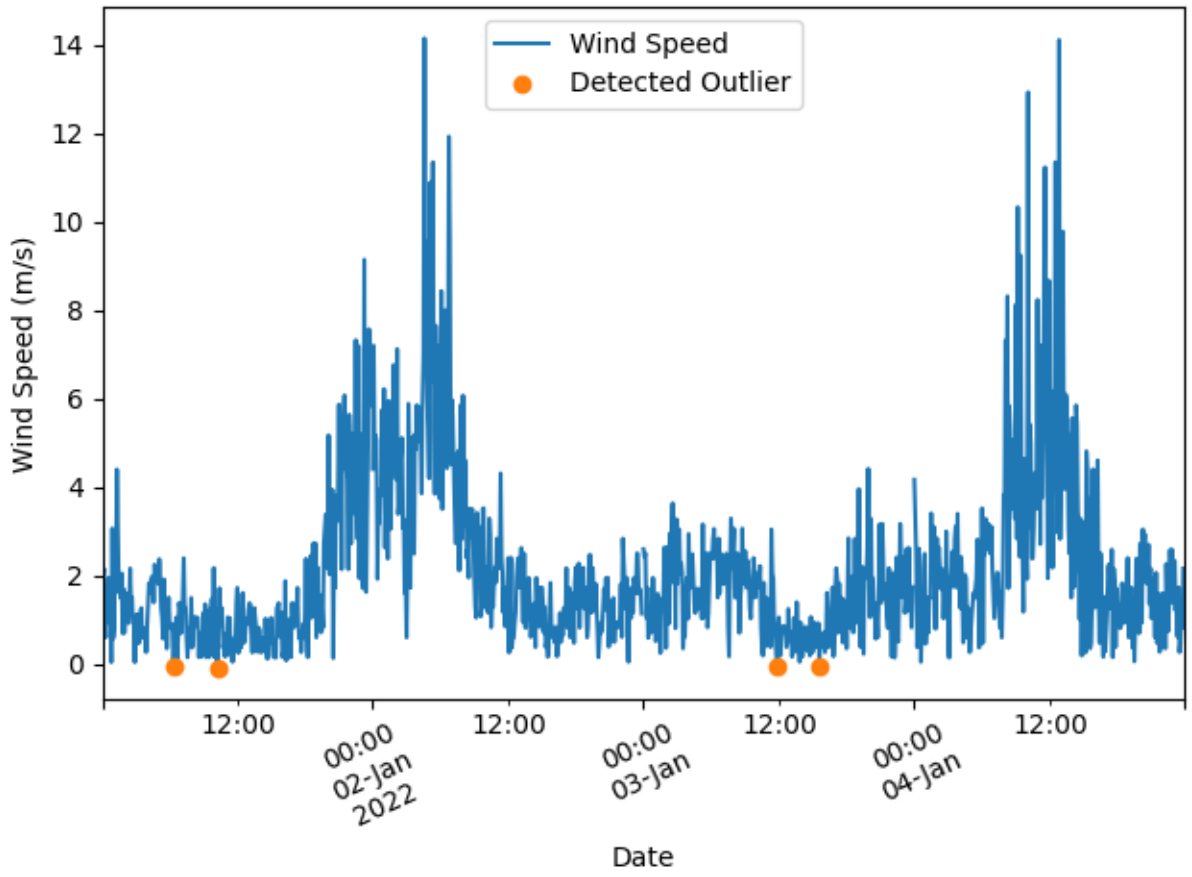
First, we read in the NREL RMIS weather station example, which contains wind speed, temperature, and relative humidity data in m/s, deg C, and % respectively. This data set contains 5-minute right-aligned measurements.

```
pvanalytics_dir = pathlib.Path(pvanalytics.__file__).parent
rmis_file = pvanalytics_dir / 'data' / 'rmis_weather_data.csv'
data = pd.read_csv(rmis_file, index_col=0, parse_dates=True)
print(data.head(10))
```

```
                     Ambient Temperature  ...  Wind Speed
2022-01-01 00:05:00            -10.59725  ...    1.930175
2022-01-01 00:10:00            -10.63128  ...    2.167881
2022-01-01 00:15:00            -10.66532  ...    0.827218
2022-01-01 00:20:00            -10.71636  ...    0.608528
2022-01-01 00:25:00            -10.66532  ...    1.036399
2022-01-01 00:30:00            -10.81846  ...    1.150498
2022-01-01 00:35:00            -10.78443  ...    1.958699
2022-01-01 00:40:00            -10.86950  ...    0.941317
2022-01-01 00:45:00            -10.88652  ...    1.511812
2022-01-01 00:50:00            -10.85249  ...    0.047541

[10 rows x 12 columns]
```

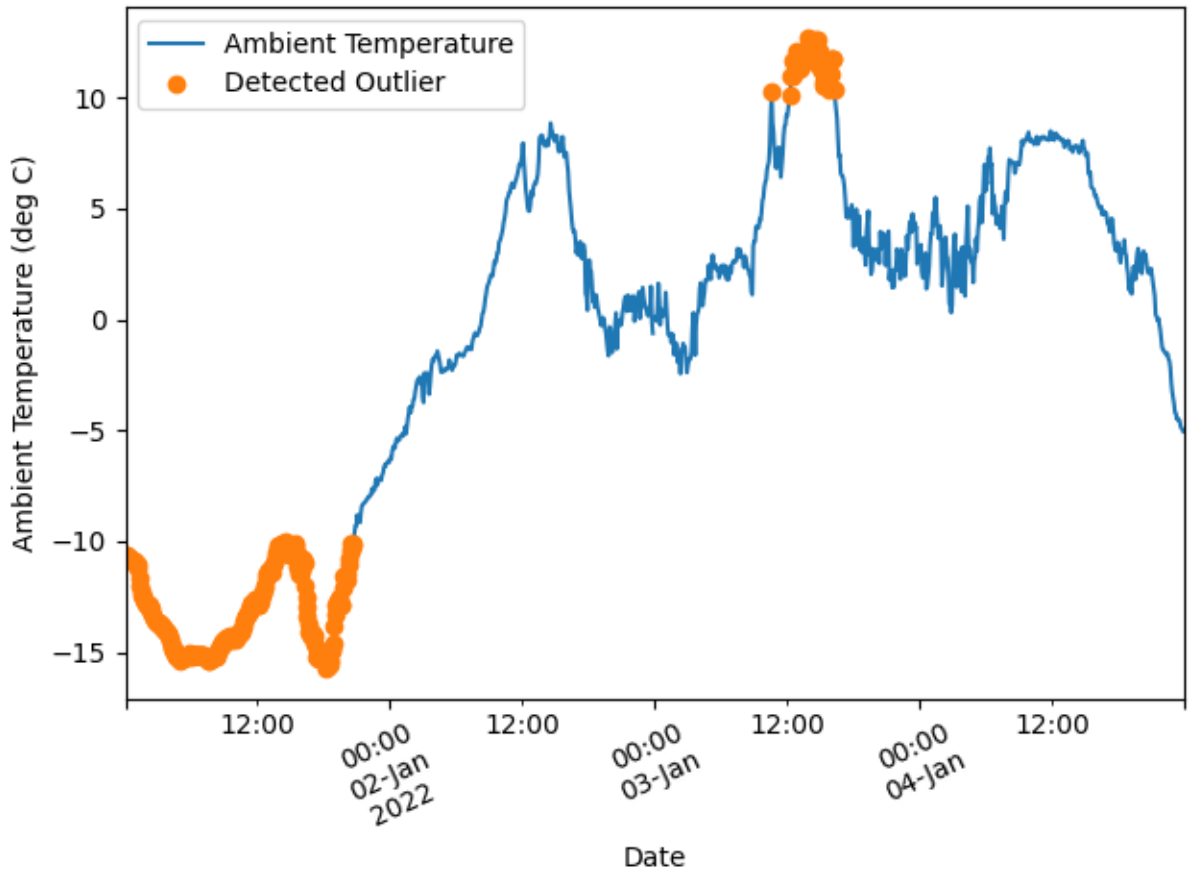
First, we use *pvanalytics.quality.weather.wind_limits()* to identify any wind speed values that are not within an acceptable range. We can then filter any of these values out of the time series.

```
wind_limit_mask = wind_limits(data['Wind Speed'])
data['Wind Speed'].plot()
data.loc[~wind_limit_mask, 'Wind Speed'].plot(ls='', marker='o')
plt.legend(labels=["Wind Speed", "Detected Outlier"])
plt.xlabel("Date")
plt.ylabel("Wind Speed (m/s)")
plt.xticks(rotation=25)
plt.tight_layout()
plt.show()
```



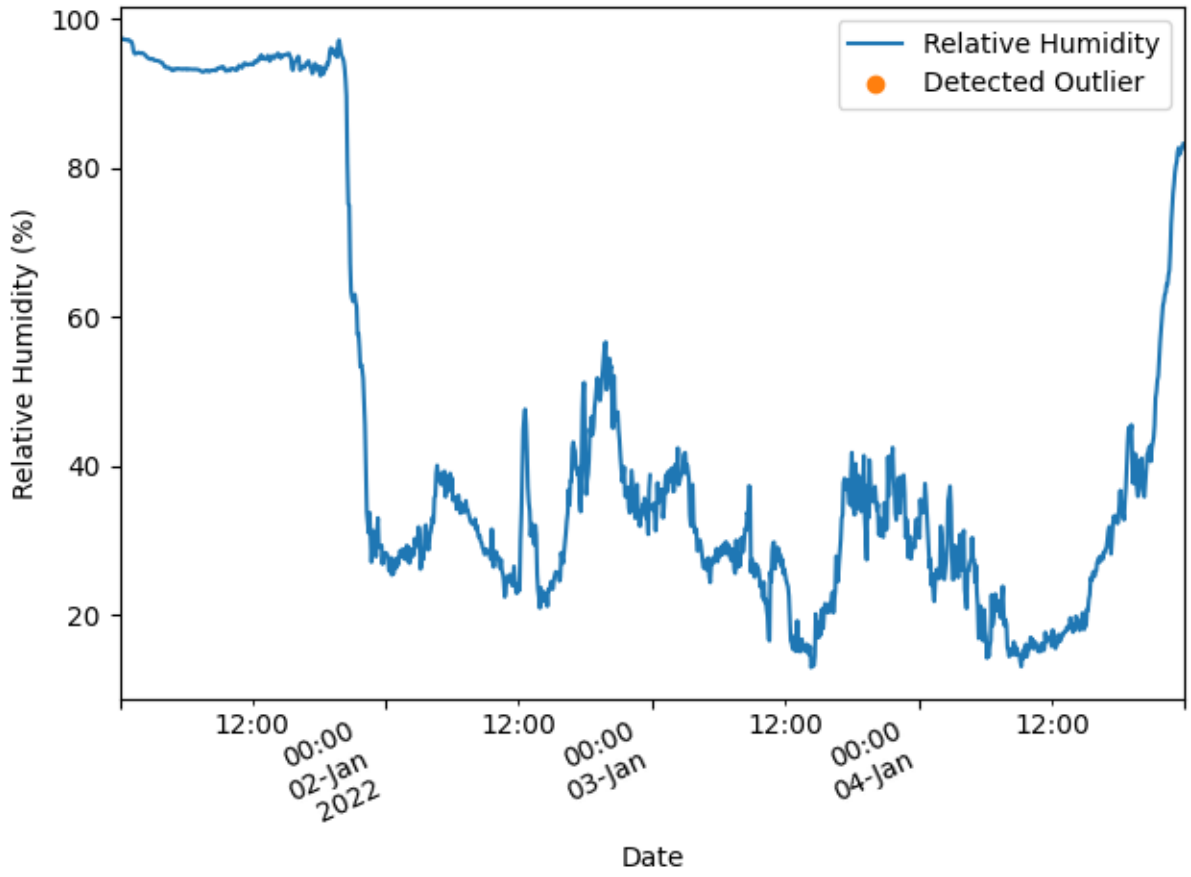
Next, we use *pvanalytics.quality.weather.temperature_limits()* to identify any air temperature values that are not within an acceptable range. We can then filter any of these values out of the time series. Here, we set the temperature limits to (-10,10), illustrating how to use the limits parameter.

```
temperature_limit_mask = temperature_limits(data['Ambient Temperature'],
                                            limits=(-10, 10))
data['Ambient Temperature'].plot()
data.loc[~temperature_limit_mask, 'Ambient Temperature'].plot(ls='',
                                                              marker='o')
plt.legend(labels=["Ambient Temperature", "Detected Outlier"])
plt.xlabel("Date")
plt.ylabel("Ambient Temperature (deg C)")
plt.xticks(rotation=25)
plt.tight_layout()
plt.show()
```



Finally, we use *pvanalytics.quality.weather.relative_humidity_limits()* to identify any RH values that are not within an acceptable range. We can then filter any of these values out of the time series.

```
rh_limit_mask = relative_humidity_limits(data['Relative Humidity'])
data['Relative Humidity'].plot()
data.loc[~rh_limit_mask, 'Relative Humidity'].plot(ls='', marker='o')
plt.legend(labels=['Relative Humidity', "Detected Outlier"])
plt.xlabel("Date")
plt.ylabel('Relative Humidity (%)')
plt.xticks(rotation=25)
plt.tight_layout()
plt.show()
```



Total running time of the script: (0 minutes 0.759 seconds)

Module Temperature Check

Test whether the module temperature is correlated with irradiance.

Testing the correlation between module temperature and irradiance measurements can help identify if there are issues with the module temperature sensor. In this example, we demonstrate how to use *pvanalytics.quality.weather.module_temperature_check()*, which runs a linear regression model of module temperature vs. irradiance. Model performance is then assessed by the Pearson correlation coefficient. If it meets a minimum threshold, function outputs a True boolean. If not, it outputs a False boolean.

```
import pvanalytics
from pvanalytics.quality.weather import module_temperature_check
from pvanalytics.features.daytime import power_or_irradiance
import matplotlib.pyplot as plt
import pandas as pd
from scipy.stats import linregress
import pathlib
```

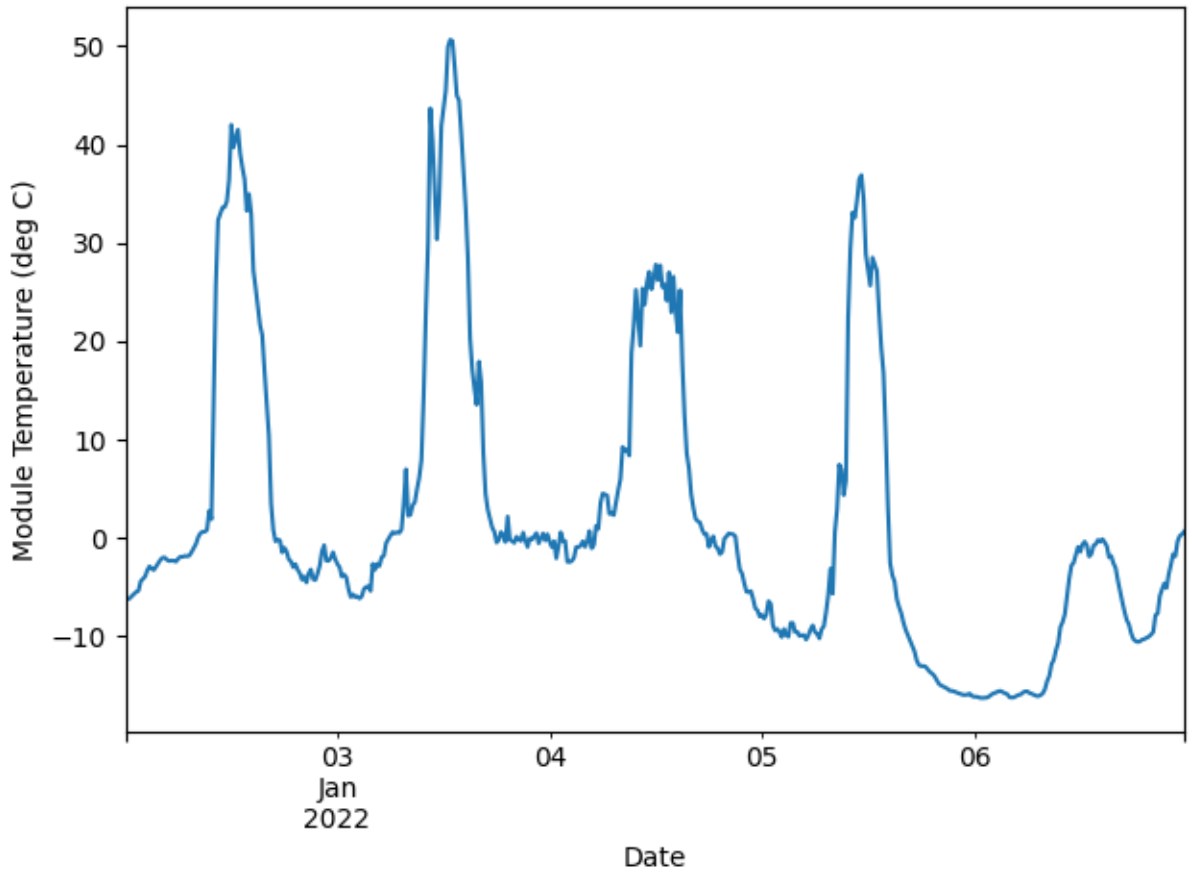
First, we read in example data from the NREL SERF West system, which contains data for module temperature and irradiance under the 'module_temp_1__781' and 'poa_irradiance__771' columns, respectively. This data set contains 15-minute averaged measurements, and is available via the NREL PVDAQ database as system 51.

```
pvanalytics_dir = pathlib.Path(pvanalytics.__file__).parent
serf_east_file = pvanalytics_dir / 'data' / 'serf_west_15min.csv'
data = pd.read_csv(serf_east_file, index_col=0, parse_dates=True)
print(data[['module_temp_1__781', 'poa_irradiance__771']].head(10))
```

```
                     module_temp_1__781  poa_irradiance__771
2022-01-02 00:01:00             -6.4187              -1.9775
2022-01-02 00:16:00             -6.2204              -2.0451
2022-01-02 00:31:00             -6.1505              -2.1464
2022-01-02 00:46:00             -5.9059              -2.1463
2022-01-02 01:01:00             -5.7022              -1.8083
2022-01-02 01:16:00             -5.4755              -1.9941
2022-01-02 01:31:00             -5.3714              -2.0109
2022-01-02 01:46:00             -4.4072              -1.7236
2022-01-02 02:01:00             -4.1609              -1.6221
2022-01-02 02:16:00             -3.9174              -1.6390
```

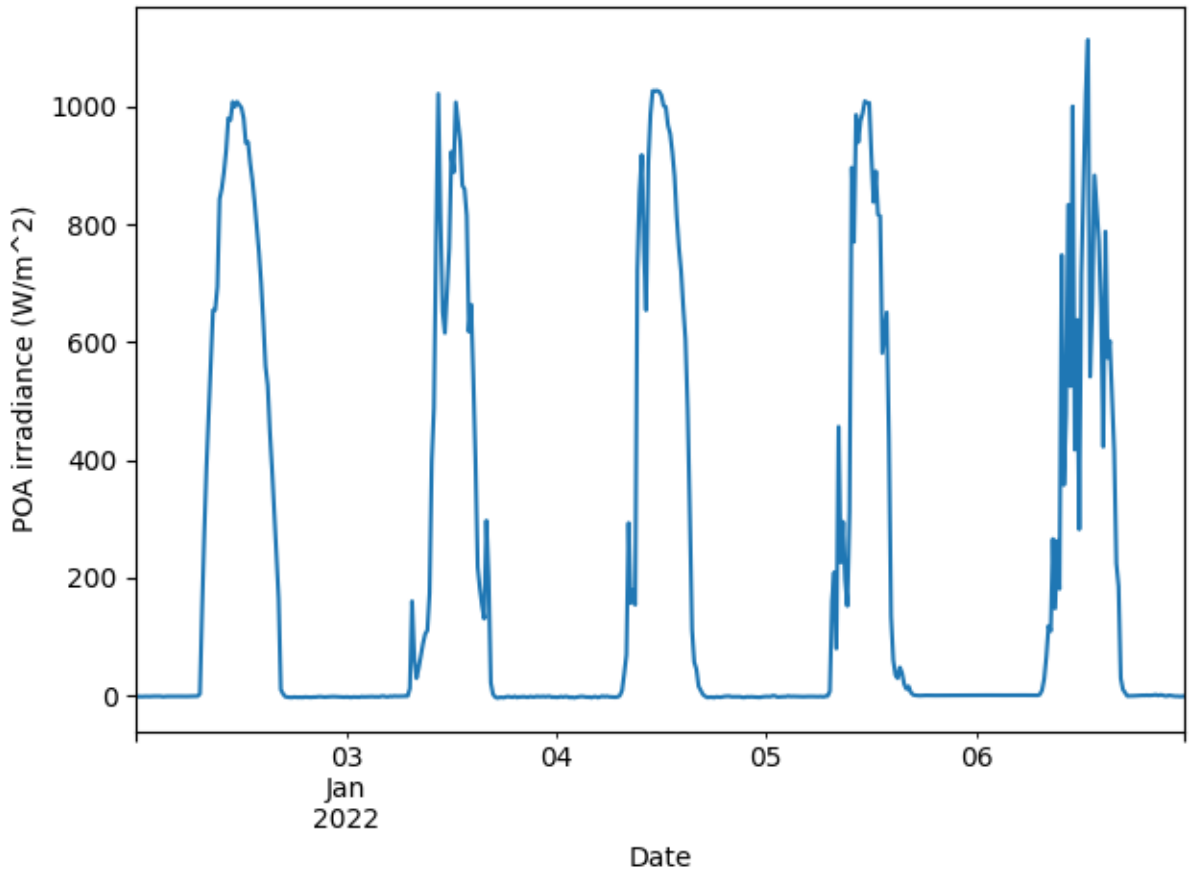
Plot the module temperature to visualize it.

```
data['module_temp_1__781'].plot()
plt.xlabel("Date")
plt.ylabel("Module Temperature (deg C)")
plt.xticks(rotation=25)
plt.tight_layout()
plt.show()
```



Plot the POA irradiance to visualize it.

```
data['poa_irradiance__771'].plot()
plt.xlabel("Date")
plt.ylabel("POA irradiance (W/m^2)")
plt.xticks(rotation=25)
plt.tight_layout()
plt.show()
```



We mask the irradiance time series into day-night periods, and remove any nighttime data to clean up the future regression.

```
predicted_day_night_mask = power_or_irradiance(
    series=data['poa_irradiance__771'], freq='15min')
# Filter out nighttime periods
data = data[predicted_day_night_mask]
```

We then use *pvanalytics.quality.weather.module_temperature_check()* to regress module temperature against irradiance POA, and check if the relationship meets the minimum correlation coefficient criteria.

```
corr_coeff_bool = module_temperature_check(data['module_temp_1__781'],
                                           data['poa_irradiance__771'])
print("Passes correlation coeff threshold? " + str(corr_coeff_bool))
```

```
Passes correlation coeff threshold? True
```

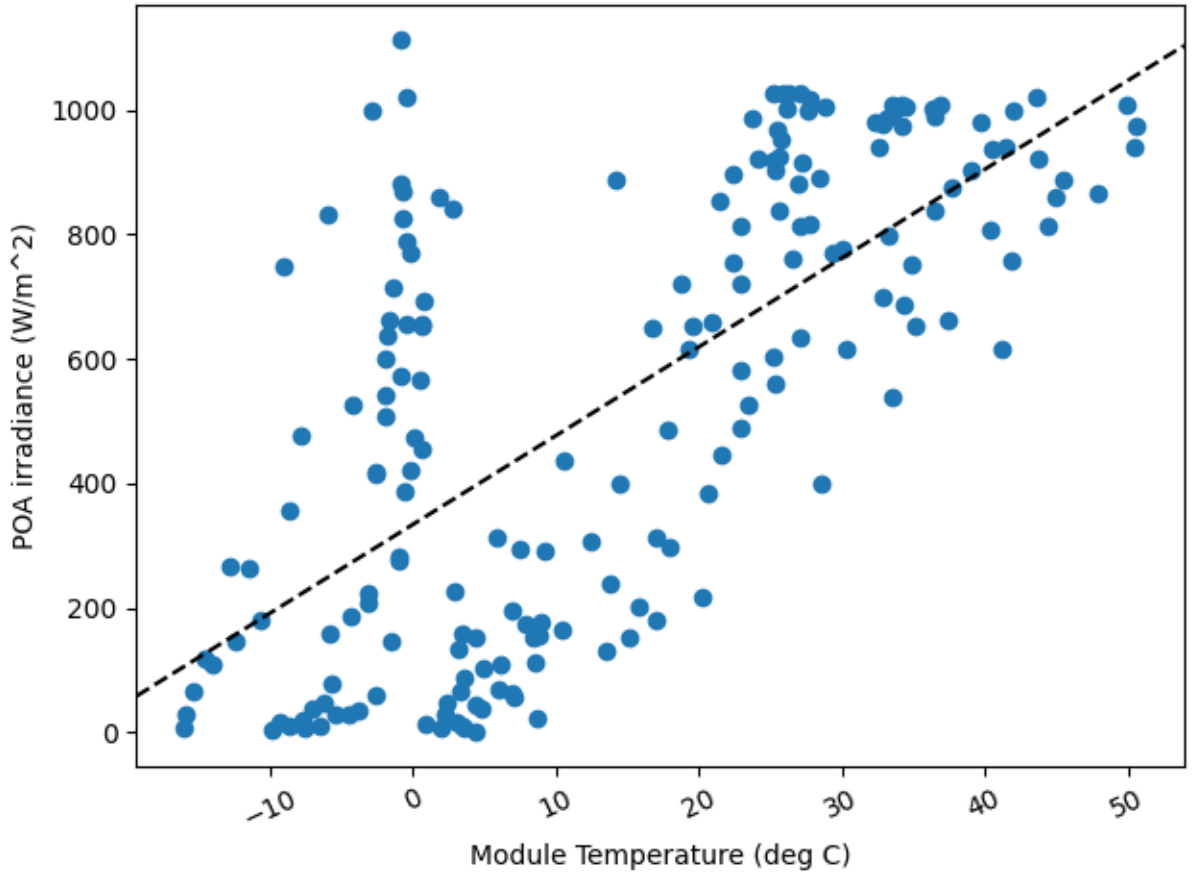
We then plot module temperature against irradiance to illustrate the relationship.

```
data.plot(x='module_temp_1__781',
          y='poa_irradiance__771',
          style='o', legend=None)
data_reg = data[['module_temp_1__781', 'poa_irradiance__771']].dropna()
# Add the linear regression line
```

(continues on next page)

(continued from previous page)

```
reg = linregress(data_reg['module_temp_1__781'].values,
                 data_reg['poa_irradiance__771'].values)
plt.axline(xy1=(0, reg.intercept), slope=reg.slope, linestyle="--", color="k")
plt.xlabel("Module Temperature (deg C)")
plt.ylabel("POA irradiance (W/m^2)")
plt.xticks(rotation=25)
plt.tight_layout()
plt.show()

# Print the Pearson correlation coefficient associated with the regression.
print("Pearson Correlation Coefficient: ")
print(reg.rvalue)
```




```
Pearson Correlation Coefficient:
0.6836377995026489
```

Total running time of the script: (0 minutes 0.547 seconds)

2.3 Release Notes

These are the bug-fixes, new features, and improvements for each release.

2.3.1 0.2.0 (February 14, 2024)

Breaking Changes

- Updated function *infer_orientation_fit_pvwatts()* to more closely align with the PVWatts v5 methodology. This includes incorporating relative airmass and extraterrestrial irradiance into the Perez total irradiance model, accounting for array incidence loss (IAM), and including losses in the PVWatts inverter model. Additionally, added optional arguments for bounding the azimuth range in during least squares optimization. (GH147, GH180)
- Updated function *shifts_ruptures()* to align with the methodology tested and reported on at PVRW 2023 ("Survey of Time Shift Detection Algorithms for Measured PV Data"). This includes converting the changepoint detection algorithm from Pelt to Binary Segmentation (which runs much faster), and performing additional processing to each detected segment to remove outliers and filter by a quantile cutoff instead of the original rounding technique. (GH197)

Enhancements

- Added function *get_sunrise()* for calculating the daily sunrise datetimes for a time series, based on the *power_or_irradiance()* day/night mask output. (GH187)
- Added function *get_sunset()* for calculating the daily sunset datetimes for a time series, based on the *power_or_irradiance()* day/night mask output. (GH187)
- Updated function *power_or_irradiance()* to be more performant by vectorization; the original logic was using a lambda call that was slowing the function speed down considerably. This update resulted in a ~50X speedup. (GH186)

Bug Fixes

- `pvanalytics.__version__` now correctly reports the version string instead of raising `AttributeError`. (GH181)
- Compatibility with pandas 2.0.0 (GH185) and future versions of pandas (GH203)
- Compatibility with scipy 1.11 (GH196)
- Updated function *trim()* to handle pandas 2.0.0 update for tz-aware timeseries (GH206)

Requirements

- Advance minimum `pvlib` to 0.9.4, `numpy` to 0.16.0, `pandas` to 1.0.0, and `scipy` to 1.6.0. (GH179, GH185)

Documentation

- Online docs now use `pydata-sphinx-theme` instead of the built-in `alabaster` theme. (GH176, GH178)
- Added PVFleets QA pipeline examples for checking temperature, irradiance, and power streams. (GH201, GH202)
- Added a gallery page for *shifts_ruptures()*. (GH192)

Testing

- Added testing for python 3.11 and 3.12. (GH189, GH204)

Contributors

- Kirsten Perry (@kperrynrel)
- Kevin Anderson (@kanderso-nrel)
- Cliff Hansen (@cwhanse)
- Abhishek Parikh (@abhisheksparikh)
- Quyen Nguyen (@qnguyen345)
- Adam R. Jensen (@adamrjensen)
- Chris Deline (@cdeline)

2.3.2 0.1.3 (December 16, 2022)

Enhancements

- Added function *calculate_component_sum_series()* for calculating the component sum values of GHI, DHI, and DNI, and performing nighttime corrections (GH157, GH163)
- Updated the *stale_values_round()* function with pandas functionality, leading to the same results with a 300X speedup. (GH156, GH158)

Documentation

Added new gallery example pages:

- `pvanalytics.metrics` (GH133, GH153):
  - *performance_ratio_nrel()*
  - *variability_index()*
- `pvanalytics.quality.weather` (GH133, GH151):
  - *temperature_limits()*
  - *relative_humidity_limits()*
  - *wind_limits()*
  - *module_temperature_check()*
- `pvanalytics.quality.irradiance` (GH157, GH163)

  - *calculate_component_sum_series()*

- pvanalytics.system (GH133, GH146):
  - *infer_orientation_fit_pvwatts()*
  - *is_tracking_envelope()*
- Clarified parameter descriptions for pdc0 and pac in *performance_ratio_nrel()* (GH152, GH162).
- Restructured the example gallery by separating the examples into categories and adding README's (GH154, GH155).
- Revised the pull request template (GH159, GH160).

Contributors

- Kirsten Perry (@kperrynrel)
- Cliff Hansen (@cwhanse)
- Josh Peterson (@PetersonUOregon)
- Adam R. Jensen (@adamrjensen)
- Will Holmgren (@wholmgren)
- Kevin Anderson (@kanderso-nrel)

2.3.3 0.1.2 (August 18, 2022)

Enhancements

- Detect data shifts in daily summed time series with *pvanalytics.quality.data_shifts.detect_data_shifts()* and *pvanalytics.quality.data_shifts.get_longest_shift_segment_dates()*. (GH142)

Bug Fixes

- Fix *pvanalytics.quality.outliers.zscore()* so that the NaN mask is assigned the time series index (GH138)

Documentation

Added fifteen new gallery example pages:

- pvanalytics.features.clipping (GH133, GH134):
  - *geometric()*
- pvanalytics.quality.gaps (GH133, GH135):
  - *stale_values_diff()*
  - *stale_values_round()*
  - *interpolation_diff()*
  - *completeness_score()*

  - *complete()*
  - *trim_incomplete()*
- `pvanalytics.quality.outliers` (GH133, GH138):
  - *tukey()*
  - *zscore()*
  - *hampel()*
- `pvanalytics.features.daytime` (GH133, GH139):
  - *power_or_irradiance()*
- `pvanalytics.quality.irradiance` (GH133, GH140):
  - *clearsky_limits()*
  - *daily_insolation_limits()*
  - *check_irradiance_consistency_qcrad()*
  - *check_irradiance_limits_qcrad()*
- `pvanalytics.features.orientation` (GH133, GH148):
  - *fixed_nrel()*
  - *tracking_nrel()*
- `pvanalytics.quality.data_shifts` (GH131):
  - *detect_data_shifts()*
  - *get_longest_shift_segment_dates()*

Other

- Removed empty modules `pvanalytics.filtering` and `pvanalytics.fitting` until the relevant functionality is added to the package. (GH145)

Contributors

- Kirsten Perry (@kperrynrel)
- Cliff Hansen (@cwhanse)
- Kevin Anderson (@kanderso-nrel)
- Will Vining (@wfvining)

2.3.4 0.1.1 (February 18, 2022)

Enhancements

- Quantification of irradiance variability with *pvanalytics.metrics.variability_index()*. (GH60, GH106)
- Internal refactor of *pvanalytics.metrics.performance_ratio_nrel()* to support other performance ratio formulas. (GH109)
- Detect shadows from fixed objects in GHI data using *pvanalytics.features.shading.fixed()*. (GH24, GH101)

Bug Fixes

- Added `nan_policy` parameter to zscore calculation in *pvanalytics.quality.outliers.zscore()*. (GH102, GH108)
- Prohibit pandas versions in the 1.1.x series to avoid an issue in `.groupby().rolling()`. Newer versions starting in 1.2.0 and older versions going back to 0.24.0 are still allowed. (GH82, GH118)
- Fixed an issue with *pvanalytics.features.clearsky.reno()* in recent pandas versions (GH125, GH128)
- Improved convergence in *pvanalytics.features.orientation.fixed_nrel()* (GH119, GH120)

Requirements

- Drop support for python 3.6, which reached end of life Dec 2021 (GH129)

Documentation

- Started an example gallery and added an example for *pvanalytics.features.clearsky.reno()* (GH125, GH127)

Contributors

- Kevin Anderson (@kanderso-nrel)
- Cliff Hansen (@cwhanse)
- Will Vining (@wfvining)
- Kirsten Perry (@kperrynrel)
- Michael Hopwood (@MichaelHopwood)
- Carlos Silva (@camsilva)
- Ben Taylor (@bt-)

2.3.5 0.1.0 (November 20, 2020)

This is the first release of PVAnalytics. As such, the list of "changes" below is not specific. Future releases will describe specific changes here along with references to the relevant github issue and pull requests.

API Changes

Enhancements

- Quality control functions for irradiance, weather and time series data. See `pvanalytics.quality` for content.
- Feature labeling functions for clipping, clearsky, daytime, and orientation. See `pvanalytics.features` for content.
- System parameter inference for tilt, azimuth, and whether the system is tracking or fixed. See `pvanalytics.system` for content.
- NREL performance ratio metric (*pvanalytics.metrics.performance_ratio_nrel()*).

Bug Fixes

Contributors

- Will Vining (@wfvining)
- Cliff Hansen (@cwhanse)
- Saurabh Aneja (@spaneja)

Special thanks to Matt Muller and Kirsten Perry of NREL for their assistance in adapting components from the PVFleets QA project to PVAnalytics.

INDEX

T

V

W

Z