RdTools

Release 2.1.3+0.g0749090.dirty

kWh Analytics, Alliance for Sustainable Energy, LLC, SunPower, a

Jan 06, 2022

CONTENTS

RdTools is an open-source library to support reproducible technical analysis of time series data from photovoltaic energy systems. The library aims to provide best practice analysis routines along with the building blocks for users to tailor their own analyses. Current applications include the evaluation of PV production over several years to obtain rates of performance degradation and soiling loss. They also include the capability to analyze systems for system- and subsystem-level availability. RdTools can handle both high frequency (hourly or better) or low frequency (daily, weekly, etc.) datasets. Best results are obtained with higher frequency data.

Full examples are worked out in the notebooks shown in *Examples*.

To report issues, contribute code, or suggest improvements to this documentation, visit the RdTools development repository on github.

CHAPTER

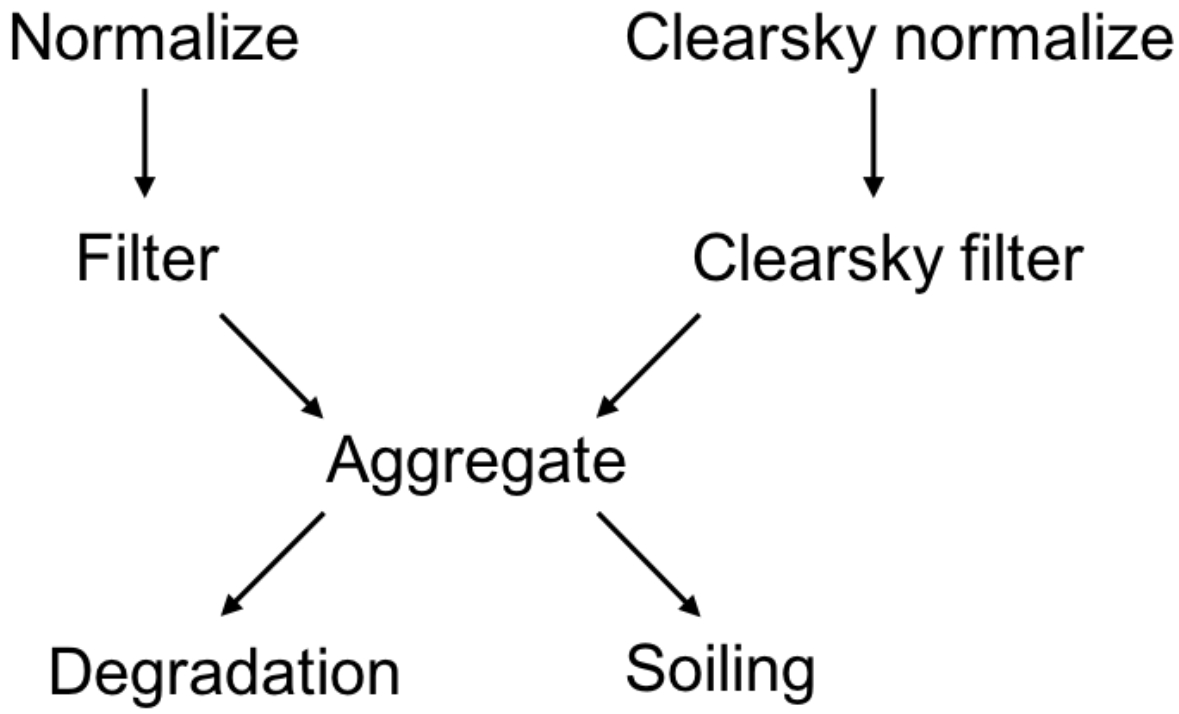
ONE

TRENDS

Both degradation and soiling analyses are based on normalized yield, similar to performance index. Usually, this is computed at the daily level although other aggregation periods are supported. A typical analysis of soiling and degradation contains the following:

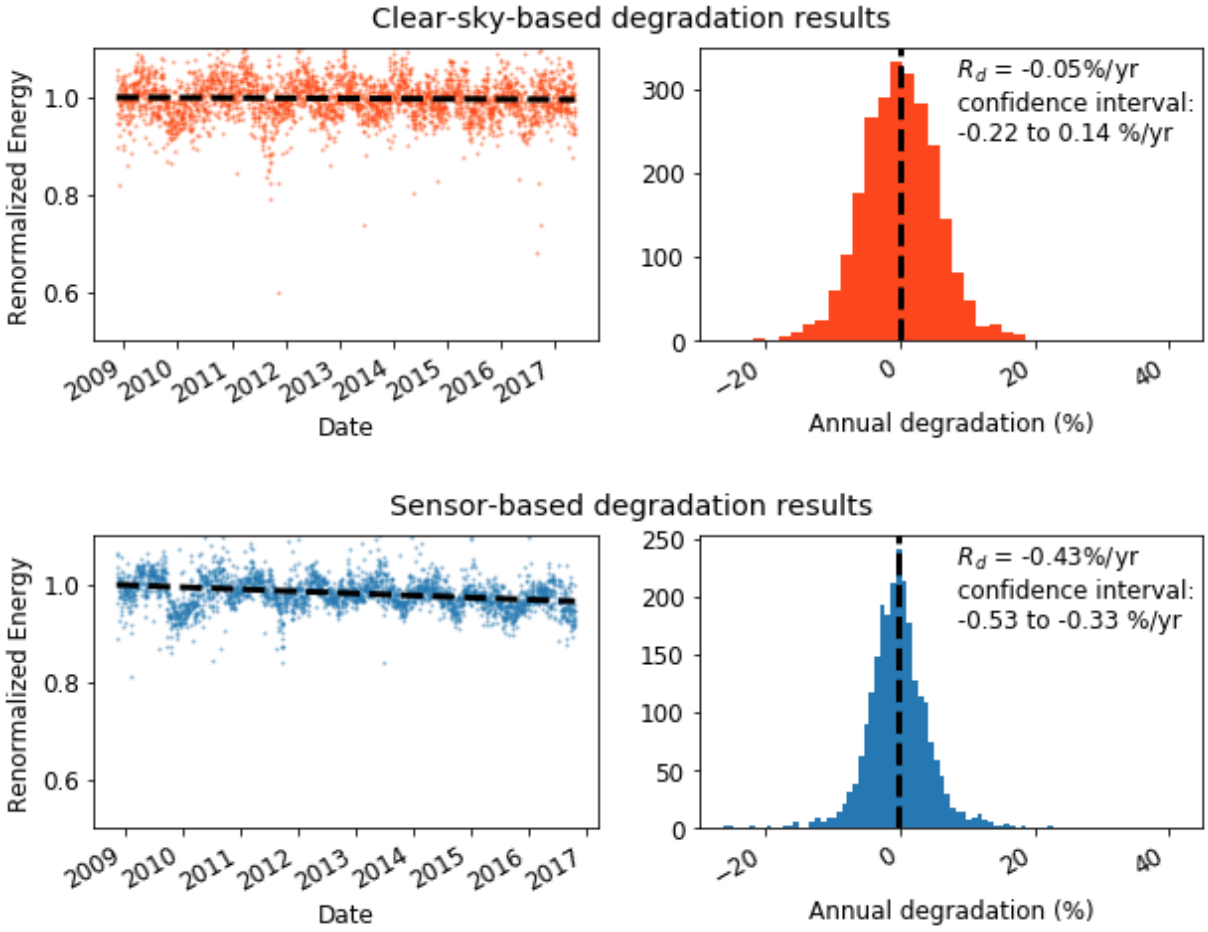
0. Import and preliminary calculations
1. Normalize data using a performance metric
2. Filter data that creates bias
3. Aggregate data
4. Analyze aggregated data to estimate the degradation rate and/or soiling loss

Steps 1 and 2 may be accomplished with the clearsky workflow (see the *Examples*) which can help eliminate problems from irradiance sensor drift.



1.1 Degradation

The preferred method for degradation rate estimation is the year-on-year (YOY) approach (Jordan 2018), available in `degradation.degradation_year_on_year()`. The YOY calculation yields in a distribution of degradation rates, the central tendency of which is the most representative of the true degradation. The width of the distribution provides information about the uncertainty in the estimate via a bootstrap calculation. The *Examples* use the output of `degradation.degradation_year_on_year()` to visualize the calculation.

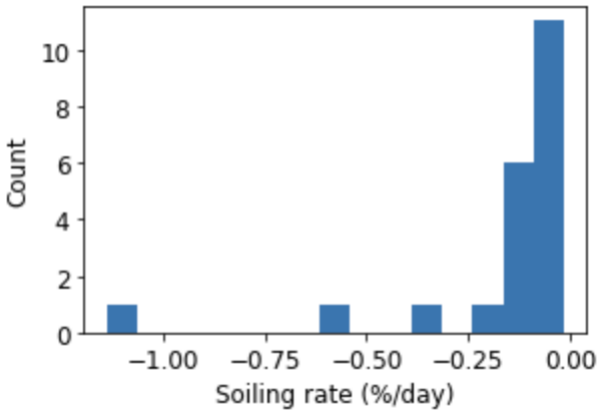


Two workflows are available for system performance ratio calculation, and illustrated in an example notebook. The sensor-based approach assumes that site irradiance and temperature sensors are calibrated and in good repair. Since this is not always the case, a 'clear-sky' workflow is provided that is based on modeled temperature and irradiance. Note that site irradiance data is still required to identify clear-sky conditions to be analyzed. In many cases, the 'clear-sky' analysis can identify conditions of instrument errors or irradiance sensor drift, such as in the above analysis.

The clear-sky analysis tends to provide less stable results than sensor-based analysis when details such as filtering are changed. We generally recommend that the clear-sky analysis be used as a check on the sensor-based results, rather than as a stand-alone analysis.

1.2 Soiling

Soiling can be estimated with the stochastic rate and recovery (SRR) method (Deceglie 2018). This method works well when soiling patterns follow a "sawtooth" pattern, a linear decline followed by a sharp recovery associated with natural or manual cleaning. *soiling.soiling_srr()* performs the calculation and returns the P50 insolation-weighted soiling ratio, confidence interval, and additional information (`soiling_info`) which includes a summary of the soiling intervals identified, `soiling_info['soiling_interval_summary']`. This summary table can, for example, be used to plot a histogram of the identified soiling rates for the dataset.



1.3 TrendAnalysis

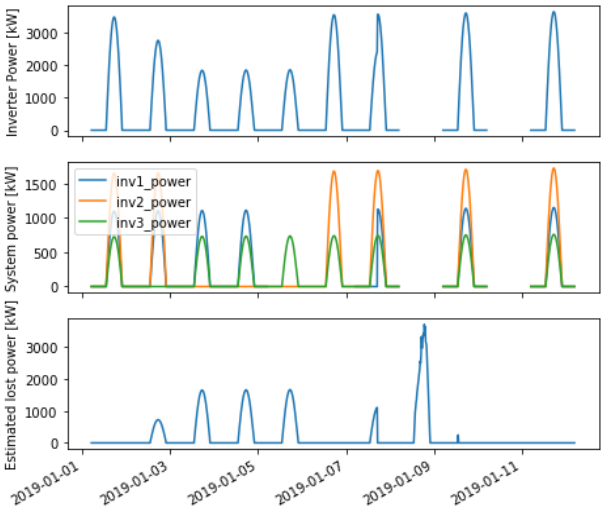
An object-oriented API for complete soiling and degradation analysis including the normalize, filter, aggregate, analyze steps is available in *analysis_chains.TrendAnalysis*. See the TrendAnalysis example for details.

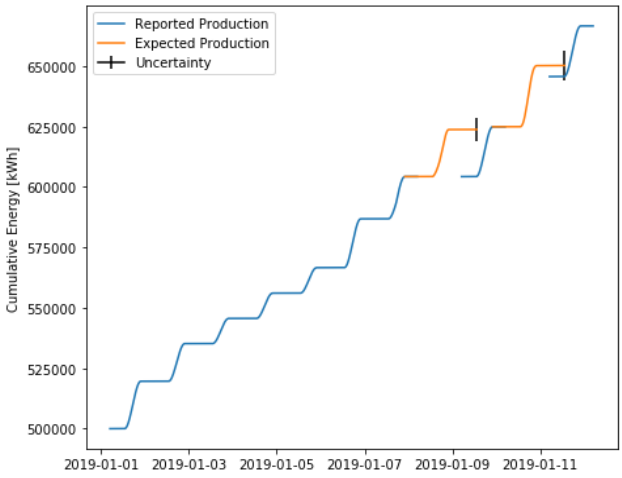
CHAPTER

TWO

AVAILABILITY

Evaluating system availability can be confounded by data loss from interrupted datalogger or system communications. RdTools implements two methods (Anderson & Blumenthal 2020) of distinguishing nuisance communication interruptions from true production outages with the `availability.AvailabilityAnalysis` class. In addition to classifying data outages, it estimates lost production and calculates energy-weighted system availability.





CHAPTER

THREE

INSTALL RDTOOLS USING PIP

RdTools can be installed automatically into Python from PyPI using the command line:

```
pip install rdtools
```

Alternatively it can be installed manually using the command line:

1. Download a release (Or to work with a development version, clone or download the rdtools repository).
2. Navigate to the repository: `cd rdtools`
3. Install via pip: `pip install .`

On some systems installation with `pip` can fail due to problems installing requirements. If this occurs, the requirements specified in `setup.py` may need to be separately installed (for example by using `conda`) before installing `rdtools`.

For more detailed instructions, see the *Developer Notes* page.

RdTools currently is tested on Python 3.7+.

CHAPTER

FOUR

USAGE AND EXAMPLES

Full workflow examples are found in the notebooks in *Examples*. The examples are designed to work with python 3.7. For a consistent experience, we recommend installing the packages and versions documented in `docs/notebook_requirements.txt`. This can be achieved in your environment by first installing RdTools as described above, then running `pip install -r docs/notebook_requirements.txt` from the base directory.

The following functions are used for degradation and soiling analysis:

```
import rdtools
```

The most frequently used functions are:

```
normalization.normalize_with_expected_power(pv, power_expected, poa_global,
                                            pv_input='power')
  '''
  Inputs: Pandas time series of raw power or energy, expected power, and
     plane of array irradiance.
  Outputs: Pandas time series of normalized energy and POA insolation
  '''
```

```
filtering.poa_filter(poa_global); filtering.tcell_filter(temperature_cell);
filtering.clip_filter(power_ac); filtering.logic_clip_filter(power_ac);
filtering.xgboost_clip_filter(power_ac); filtering.normalized_filter(energy_
↪normalized);
filtering.csi_filter(poa_global_measured, poa_global_clearsky);
  '''
  Inputs: Pandas time series of raw data to be filtered.
  Output: Boolean mask where `True` indicates acceptable data
  '''
```

```
aggregation.aggregation_insol(energy_normalized, insolation, frequency='D')
  '''
  Inputs: Normalized energy and insolation
  Output: Aggregated data, weighted by the insolation.
  '''
```

```
degradation.degradation_year_on_year(energy_normalized)
  '''
  Inputs: Aggregated, normalized, filtered time series data
  Outputs: Tuple: `yoy_rd`: Degradation rate
    `yoy_ci`: Confidence interval `yoy_info`: associated analysis data
  '''
```

```
soiling.soiling_srr(energy_normalized_daily, insolation_daily)
  '''
  Inputs: Daily aggregated, normalized, filtered time series data for normalized␣
↪performance and insolation
  Outputs: Tuple: `sr`: Insolation-weighted soiling ratio
    `sr_ci`: Confidence interval `soiling_info`: associated analysis data
  '''
```

```
availability.AvailabilityAnalysis(power_system, power_subsystem,
                                  energy_cumulative, power_expected)
  '''
  Inputs: Pandas time series system and subsystem power and energy data
  Outputs: DataFrame of production loss and availability metrics
  '''
```

CHAPTER

FIVE

DOCUMENTATION

Some RdTools function parameters can take one of several types. For example, the `albedo` parameter of `TrendAnalysis` can be a static value like `0.2` or a time-varying `pandas.Series`. To indicate that a parameter can take one of several types, we document them using the type alises listed below:

`numeric` scalar or `pandas.Series`. Typically int or float dtype.

CHAPTER

SIX

CITING RDTOOLS

To cite RdTools, please use the following along with the version number and the specific DOI coresponding to that version from Zenodo:

- Michael G. Deceglie, Ambarish Nag, Adam Shinn, Gregory Kimball, Daniel Ruth, Dirk Jordan, Jiyang Yan, Kevin Anderson, Kirsten Perry, Mark Mikofski, Matthew Muller, Will Vining, and Chris Deline RdTools, version {insert version}, Compuer Software, https://github.com/NREL/rdtools. DOI:{insert DOI}

The underlying workflow of RdTools has been published in several places. If you use RdTools in a published work, you may also wish to cite the following as appropriate:

- Dirk Jordan, Chris Deline, Sarah Kurtz, Gregory Kimball, Michael Anderson, "Robust PV Degradation Methodology and Application", IEEE Journal of Photovoltaics, 8(2) pp. 525-531, 2018 DOI: 10.1109/JPHOTOV.2017.2779779
- Michael G. Deceglie, Leonardo Micheli and Matthew Muller, "Quantifying Soiling Loss Directly From PV Yield," in IEEE Journal of Photovoltaics, 8(2), pp. 547-551, 2018 DOI: 10.1109/JPHOTOV.2017.2784682
- Kevin Anderson and Ryan Blumenthal, "Overcoming Communications Outages in Inverter Downtime Analysis", 2020 IEEE 47th Photovoltaic Specialists Conference (PVSC). DOI: 10.1109/PVSC45281.2020.9300635
- Kirsten Perry, Matthew Muller and Kevin Anderson, "Performance Comparison of Clipping Detection Techniques in AC Power Time Series," 2021 IEEE 48th Photovoltaic Specialists Conference (PVSC), 2021, pp. 1638-1643, DOI: 10.1109/PVSC43889.2021.9518733

CHAPTER

SEVEN

REFERENCES

- The clear sky temperature calculation, `clearsky_temperature.get_clearsky_tamb()`, uses data from images created by Jesse Allen, NASA's Earth Observatory using data courtesy of the MODIS Land Group.
  - https://neo.sci.gsfc.nasa.gov/view.php?datasetId=MOD_LSTD_CLIM_M
  - https://neo.sci.gsfc.nasa.gov/view.php?datasetId=MOD_LSTN_CLIM_M

Other useful references which may also be consulted for degradation rate methodology include:

- D. C. Jordan, M. G. Deceglie, S. R. Kurtz, "PV degradation methodology comparison — A basis for a standard", in 43rd IEEE Photovoltaic Specialists Conference, Portland, OR, USA, 2016, DOI: 10.1109/PVSC.2016.7749593.
- Jordan DC, Kurtz SR, VanSant KT, Newmiller J, Compendium of Photovoltaic Degradation Rates, Progress in Photovoltaics: Research and Application, 2016, 24(7), 978 - 989.
- D. Jordan, S. Kurtz, PV Degradation Rates – an Analytical Review, Progress in Photovoltaics: Research and Application, 2013, 21(1), 12 - 29.
- E. Hasselbrink, M. Anderson, Z. Defreitas, M. Mikofski, Y.-C.Shen, S. Caldwell, A. Terao, D. Kavulak, Z. Campeau, D. DeGraaff, "Validation of the PVLife model using 3 million module-years of live site data", 39th IEEE Photovoltaic Specialists Conference, Tampa, FL, USA, 2013, p. 7 – 13, DOI: 10.1109/PVSC.2013.6744087.

CHAPTER

EIGHT

DOCUMENTATION CONTENTS

8.1 Examples

This page shows example usage of the RdTools analysis functions.

8.1.1 Degradation and soiling example with clearsky workflow

This jupyter notebook is intended to the RdTools analysis workflow. In addition, the notebook demonstrates the effects of changes in the workflow. For a consistent experience, we recommend installing the specific versions of packages used to develop this notebook. This can be achieved in your environment by running `pip install -r requirements.txt` followed by `pip install -r docs/notebook_requirements.txt` from the base directory. (RdTools must also be separately installed.) These environments and examples are tested with Python 3.7.

The calculations consist of several steps illustrated here:

Import and preliminary calculations

Normalize data using a performance metric

Filter data that creates bias

Aggregate data

Analyze aggregated data to estimate the degradation rate

Analyze aggregated data to estimate the soiling loss

After demonstrating these steps using sensor data, a modified version of the workflow is illustrated using modeled clear sky irradiance and temperature. The results from the two methods are compared at the end.

This notebook works with data from the NREL PVDAQ `[4] NREL x-Si #1` system. Note that because this system does not experience significant soiling, the dataset contains a synthesized soiling signal for use in the soiling section of the example. This notebook automatically downloads and locally caches the dataset used in this example. The data can also be found on the DuraMAT Datahub (https://datahub.duramat.org/dataset/pvdaq-time-series-with-soiling-signal).

[1]:
```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import pvlib
import rdtools
%matplotlib inline
```

[2]:
```
#Update the style of plots
import matplotlib
matplotlib.rcParams.update({'font.size': 12,
                            'figure.figsize': [4.5, 3],
                            'lines.markeredgewidth': 0,
                            'lines.markersize': 2
                            })
# Register time series plotting in pandas > 1.0
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()
```

[3]:
```
# Set the random seed for numpy to ensure consistent results
np.random.seed(0)
```

0: Import and preliminary calculations

This section prepares the data necessary for an `rdtools` calculation. The first step of the `rdtools` workflow is normalization, which requires a time series of energy yield, a time series of cell temperature, and a time series of irradiance, along with some metadata (see Step 1: Normalize)

The following section loads the data, adjusts units where needed, and renames the critical columns. The ambient temperature sensor data source is converted into estimated cell temperature. This dataset already has plane-of-array irradiance data, so no transposition is necessary.

A common challenge is handling datasets with and without daylight savings time. Make sure to specify a `pytz` timezone that does or does not include daylight savings time as appropriate for your dataset.

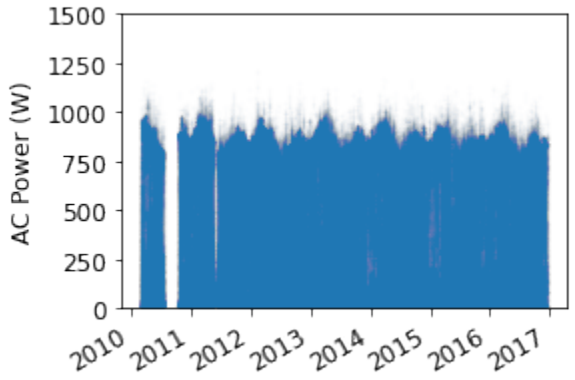
The steps of this section may change depending on your data source or the system being considered. Transposition of irradiance and modeling of cell temperature are generally outside the scope of `rdtools`. A variety of tools for these calculations are available in pvlib.

[4]:
```
# Import the example data
file_url = ('https://datahub.duramat.org/dataset/a49bb656-7b36-'
            '437a-8089-1870a40c2a7d/resource/5059bc22-640d-4dd4'
            '-b7b1-1e71da15be24/download/pvdaq_system_4_2010-2016'
            '_subset_soilsignal.csv')
cache_file = 'PVDAQ_system_4_2010-2016_subset_soilsignal.pickle'

try:
    df = pd.read_pickle(cache_file)
except FileNotFoundError:
    df = pd.read_csv(file_url, index_col=0, parse_dates=True)
    df.to_pickle(cache_file)

df = df.rename(columns = {
    'ambient_temp': 'Tamb',
    'poa_irradiance': 'poa',
})

# Specify the Metadata
meta = {"latitude": 39.7406,
        "longitude": -105.1774,
        "timezone": 'Etc/GMT+7',
        "gamma_pdc": -0.005,
        "azimuth": 180,
        "tilt": 40,
```

(continues on next page)

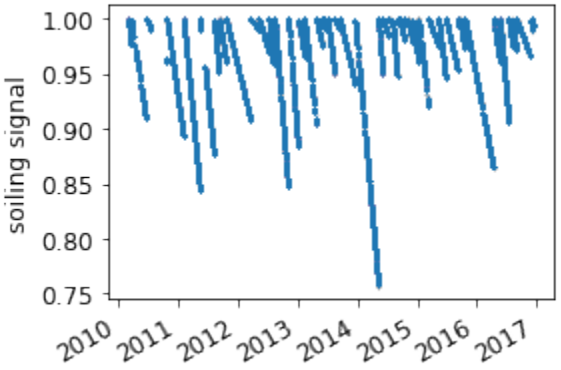
(continued from previous page)

```
        "power_dc_rated": 1000.0,
        "temp_model_params":
        pvlib.temperature.TEMPERATURE_MODEL_PARAMETERS['sapm']['open_rack_glass_
↪polymer']}

df.index = df.index.tz_localize(meta['timezone'])

# There is some missing data, but we can infer the frequency from
# the first several data points
freq = pd.infer_freq(df.index[:10])

# Then set the frequency of the dataframe.
# It is recommended not to up- or downsample at this step
# but rather to use interpolate to regularize the time series
# to its dominant or underlying frequency. Interpolate is not
# generally recommended for downsampling in this application.
df = rdtools.interpolate(df, freq)

# Calculate cell temperature
df['Tcell'] = pvlib.temperature.sapm_cell(df.poa, df.Tamb,
                                          df.wind_speed, **meta['temp_model_params'])

# plot the AC power time series
fig, ax = plt.subplots(figsize=(4,3))
ax.plot(df.index, df.ac_power, 'o', alpha=0.01)
ax.set_ylim(0,1500)
fig.autofmt_xdate()
ax.set_ylabel('AC Power (W)');
```



This example dataset includes a synthetic soiling signal that can be applied onto the PV power data to illustrate the soiling loss and detection capabilities of RdTools. AC Power is multiplied by soiling to create the synthetic 'power' channel

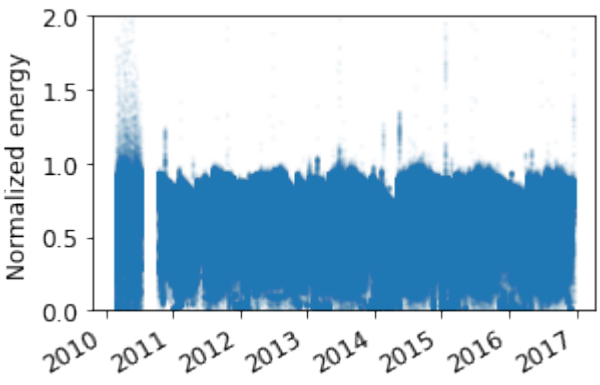
[5]:
```
fig, ax = plt.subplots(figsize=(4,3))
ax.plot(df.index, df.soiling, 'o', alpha=0.01)
#ax.set_ylim(0,1500)
fig.autofmt_xdate()
ax.set_ylabel('soiling signal');

df['power'] = df['ac_power'] * df['soiling']
```



1: Normalize

Data normalization is achieved with `rdtools.normalize_with_expected_power()`. This function can be used to normalize to any modeled or expected power. Note that realized PV output can be given as energy, rather than power, by using an optional key word argument.

[6]:
```
# Calculate the expected power with a simple PVWatts DC model
modeled_power = pvlib.pvsystem.pvwatts_dc(df['poa'], df['Tcell'], meta['power_dc_rated
↪'],
                                          meta['gamma_pdc'], 25.0 )

# Calculate the normalization, the function also returns the relevant insolation for
# each point in the normalized PV energy timeseries
normalized, insolation = rdtools.normalize_with_expected_power(df['power'],
                                                               modeled_power,
                                                               df['poa'])

df['normalized'] = normalized
df['insolation'] = insolation

# Plot the normalized power time series
fig, ax = plt.subplots()
ax.plot(normalized.index, normalized, 'o', alpha = 0.05)
ax.set_ylim(0,2)
fig.autofmt_xdate()
ax.set_ylabel('Normalized energy');
```

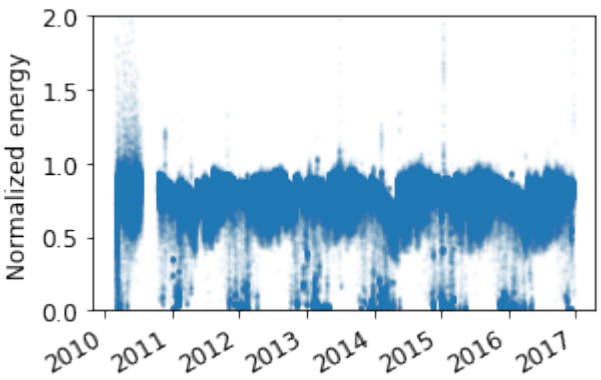


2: Filter

Data filtering is used to exclude data points that represent invalid data, create bias in the analysis, or introduce significant noise.

It can also be useful to remove outages and outliers. Sometimes outages appear as low but non-zero yield. Automatic functions for outage detection are not yet included in `rdtools`. However, this example does filter out data points where the normalized energy is less than 1%. System-specific filters should be implemented by the analyst if needed.

[7]:
```
# Calculate a collection of boolean masks that can be used
# to filter the time series
normalized_mask = rdtools.normalized_filter(df['normalized'])
poa_mask = rdtools.poa_filter(df['poa'])
tcell_mask = rdtools.tcell_filter(df['Tcell'])
# Note: This clipping mask may be disabled when you are sure the system is not
# experiencing clipping due to high DC/AC ratio
clip_mask = rdtools.clip_filter(df['power'], 'quantile')

# filter the time series and keep only the columns needed for the
# remaining steps
filtered = df[normalized_mask & poa_mask & tcell_mask & clip_mask]
filtered = filtered[['insolation', 'normalized']]

fig, ax = plt.subplots()
ax.plot(filtered.index, filtered.normalized, 'o', alpha = 0.05)
ax.set_ylim(0,2)
fig.autofmt_xdate()
ax.set_ylabel('Normalized energy');
```



Filter visualization example: different clipping filters

RdTools provides functions to visualize and tune filters for different applications. In this example, we take a subset of the data, apply an artificial clipping signal, and visualize the results of three different clipping filter methods.

[8]:
```
# Apply an artificial clipping signal to a subset of the data
example_subset = df.iloc[0:15000].copy()
example_subset.loc[example_subset['ac_power']>800,'ac_power'] =800

# Generate clipping masks with each of the available methods
clip_mask_quantile = rdtools.clip_filter(example_subset['ac_power'], 'quantile')
clip_mask_xgboost = rdtools.clip_filter(example_subset['ac_power'], 'xgboost')
clip_mask_logic = rdtools.clip_filter(example_subset['ac_power'], 'logic')
```

```
/rdtools/filtering.py:639: UserWarning: The XGBoost filter is an experimental␣
↪clipping filter that is still under development. The API, results, and default␣
↪behaviors may change in future releases (including MINOR and PATCH). Use at your␣
↪own risk!
  warnings.warn("The XGBoost filter is an experimental clipping filter "
/rdtools/filtering.py:415: UserWarning: The logic-based filter is an experimental␣
↪clipping filter that is still under development. The API, results, and default␣
↪behaviors may change in future releases (including MINOR and PATCH). Use at your␣
↪own risk!
  warnings.warn("The logic-based filter is an experimental clipping filter "
```

[9]:
```
rdtools.tune_filter_plot(example_subset['ac_power'], clip_mask_quantile)
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

[10]:
```
rdtools.tune_filter_plot(example_subset['ac_power'], clip_mask_xgboost)
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

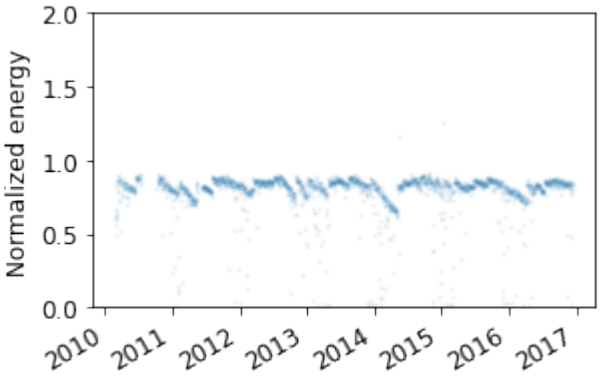
[11]:
```
rdtools.tune_filter_plot(example_subset['ac_power'], clip_mask_logic)
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

3: Aggregate

Data is aggregated with an irradiance weighted average. This can be useful, for example with daily aggregation, to reduce the impact of high-error data points in the morning and evening.

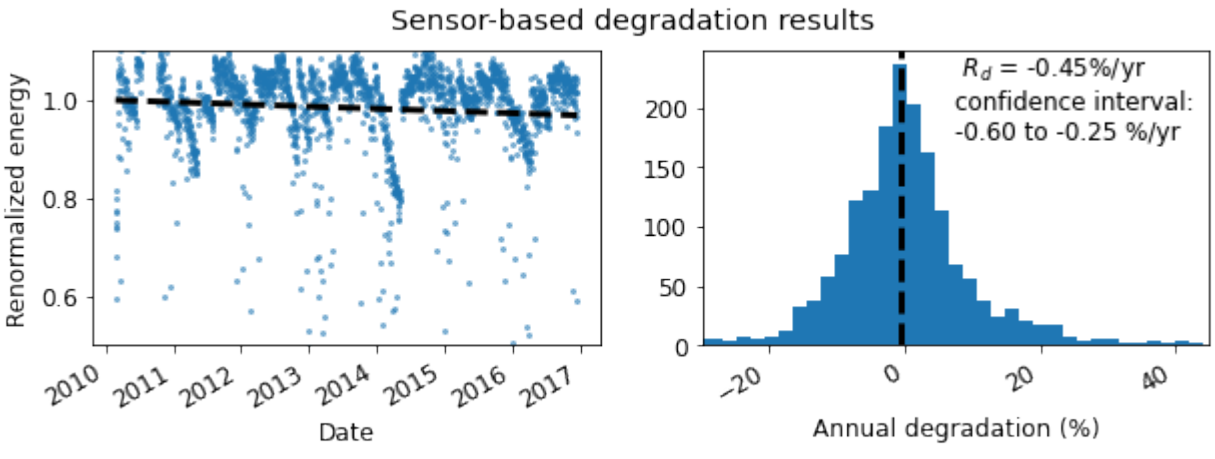
[12]:
```
daily = rdtools.aggregation_insol(filtered.normalized, filtered.insolation,
                                  frequency = 'D')

fig, ax = plt.subplots()
ax.plot(daily.index, daily, 'o', alpha = 0.1)
ax.set_ylim(0,2)
fig.autofmt_xdate()
ax.set_ylabel('Normalized energy');
```



4: Degradation calculation

Data is then analyzed to estimate the degradation rate representing the PV system behavior. The results are visualized and statistics are reported, including the 68.2% confidence interval, and the P95 exceedance value.

[13]:
```
# Calculate the degradation rate using the YoY method
yoy_rd, yoy_ci, yoy_info = rdtools.degradation_year_on_year(daily, confidence_
↪level=68.2)
# Note the default confidence_level of 68.2 is appropriate if you would like to
# report a confidence interval analogous to the standard deviation of a normal
# distribution. The size of the confidence interval is adjustable by setting the
# confidence_level variable.

# Visualize the results

degradation_fig = rdtools.degradation_summary_plots(
    yoy_rd, yoy_ci, yoy_info, daily,
    summary_title='Sensor-based degradation results',
    scatter_ymin=0.5, scatter_ymax=1.1,
    hist_xmin=-30, hist_xmax=45, bins=100
)
```



In addition to the confidence interval, the year-on-year method yields an exceedance value (e.g. P95), the degradation rate that was exceeded (slower degradation) with a given probability level. The probability level is set via the `exceedance_prob` keyword in `degradation_year_on_year`.

[14]:
```
print('The P95 exceedance level is %.2f%%/yr' % yoy_info['exceedance_level'])
```
```
The P95 exceedance level is -0.72%/yr
```

[15]:
```
print(yoy_rd)
print(yoy_ci)
```
```
-0.4540480935921178
[-0.60485954 -0.25286447]
```

5: Soiling calculations

This section illustrates how the aggregated data can be used to estimate soiling losses using the stochastic rate and recovery (SRR) method.[1] Since our example system doesn't experience much soiling, we apply an artificially generated soiling signal, just for the sake of example.

[1] M. G. Deceglie, L. Micheli and M. Muller, "Quantifying Soiling Loss Directly From PV Yield," IEEE Journal of Photovoltaics, vol. 8, no. 2, pp. 547-551, March 2018. doi: 10.1109/JPHOTOV.2017.2784682

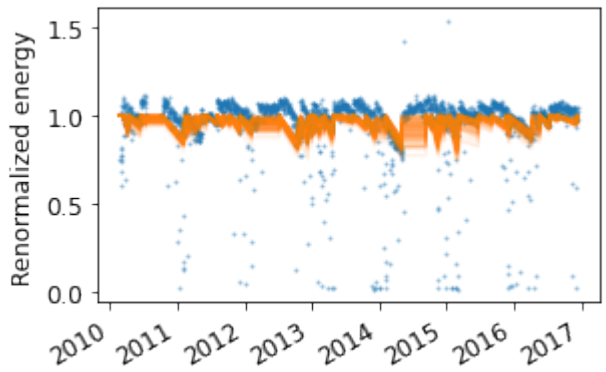
[16]:
```
# Calculate the daily insolation, required for the SRR calculation
daily_insolation = filtered['insolation'].resample('D').sum()

# Perform the SRR calculation
from rdtools.soiling import soiling_srr
cl = 68.2
sr, sr_ci, soiling_info = soiling_srr(daily, daily_insolation,
                                      confidence_level=cl)
```
```
/rdtools/soiling.py:15: UserWarning:

The soiling module is currently experimental. The API, results, and default behaviors␣
↪may change in future releases (including MINOR and PATCH releases) as the code␣
↪matures.

/rdtools/soiling.py:366: UserWarning:

20% or more of the daily data is assigned to invalid soiling intervals. This can be␣
↪problematic with the "half_norm_clean" and "random_clean" cleaning assumptions.␣
↪Consider more permissive validity criteria such as increasing "max_relative_slope_
↪error" and/or "max_negative_step" and/or decreasing "min_interval_length".␣
↪Alternatively, consider using method="perfect_clean". For more info see https://
↪github.com/NREL/rdtools/issues/272
```

[17]:
```
print('The P50 insolation-weighted soiling ratio is %0.3f'%sr)
```
```
The P50 insolation-weighted soiling ratio is 0.949
```

[18]:
```
print('The %0.1f confidence interval for the insolation-weighted'
      ' soiling ratio is %0.3f-%0.3f'%(cl, sr_ci[0], sr_ci[1]))
```
```
The 68.2 confidence interval for the insolation-weighted soiling ratio is 0.944-0.954
```

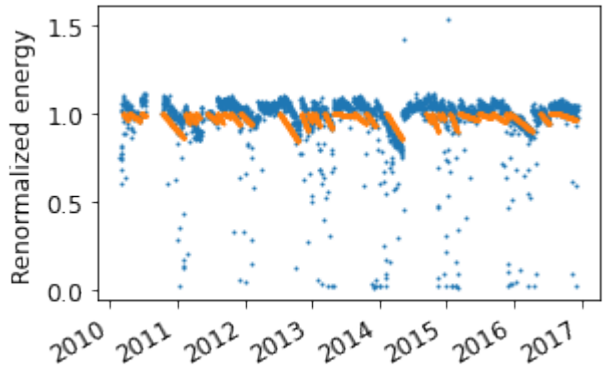
[19]:
```
# Plot Monte Carlo realizations of soiling profiles
fig = rdtools.plotting.soiling_monte_carlo_plot(soiling_info, daily, profiles=200);
```

```
/rdtools/plotting.py:166: UserWarning:

The soiling module is currently experimental. The API, results, and default behaviors␣
↪may change in future releases (including MINOR and PATCH releases) as the code␣
↪matures.
```



[20]:
```
# Plot the slopes for "valid" soiling intervals identified,
# assuming perfect cleaning events
fig = rdtools.plotting.soiling_interval_plot(soiling_info, daily);
```

```
/rdtools/plotting.py:226: UserWarning:

The soiling module is currently experimental. The API, results, and default behaviors␣
↪may change in future releases (including MINOR and PATCH releases) as the code␣
↪matures.
```



[21]:
```
# View the first several rows of the soiling interval summary table
soiling_summary = soiling_info['soiling_interval_summary']
soiling_summary.head()
```

[21]:
```
                      start                       end  soiling_rate  \
0 2010-02-25 00:00:00-07:00 2010-03-03 00:00:00-07:00           0.0
1 2010-03-04 00:00:00-07:00 2010-03-04 00:00:00-07:00           0.0
2 2010-03-05 00:00:00-07:00 2010-03-05 00:00:00-07:00           0.0
3 2010-03-06 00:00:00-07:00 2010-03-06 00:00:00-07:00           0.0
4 2010-03-07 00:00:00-07:00 2010-03-09 00:00:00-07:00           0.0

   soiling_rate_low  soiling_rate_high  inferred_start_loss  \
```

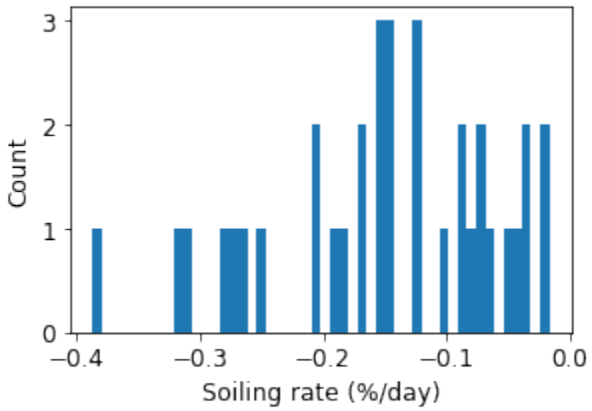
(continues on next page)

(continued from previous page)

```
0                0.0                0.0             0.721703
1                0.0                0.0             0.800385
2                0.0                0.0             0.814419
3                0.0                0.0             1.060073
4                0.0                0.0             1.027141

   inferred_end_loss  length  valid
0           0.721703       6  False
1           0.800385       0  False
2           0.814419       0  False
3           1.060073       0  False
4           1.027141       2  False
```

[22]:
```
# View a histogram of the valid soiling rates found for the data set
fig = rdtools.plotting.soiling_rate_histogram(soiling_info, bins=50)
```

```
/rdtools/plotting.py:266: UserWarning:

The soiling module is currently experimental. The API, results, and default behaviors
↪may change in future releases (including MINOR and PATCH releases) as the code
↪matures.
```



These plots show generally good results from the SRR method. In this example, we have slightly overestimated the soiling loss because we used the default behavior of the `method` key word argument in `rdtools.soiling_srr()`, which does not assume that every cleaning is perfect but the example artificial soiling signal did include perfect cleaning. We encourage you to adjust the options of `rdtools.soiling_srr()` for your application.

[23]:
```
# Calculate and view a monthly soiling rate summary
from rdtools.soiling import monthly_soiling_rates
monthly_soiling_rates(soiling_info['soiling_interval_summary'],
                      confidence_level=cl)
```

[23]:
```
    month  soiling_rate_median  soiling_rate_low  soiling_rate_high  \
0       1            -0.001135         -0.002058          -0.000788
1       2            -0.001623         -0.003788          -0.000830
2       3            -0.001481         -0.002202          -0.000524
3       4            -0.001170         -0.001967          -0.000379
4       5            -0.000401         -0.000939          -0.000199
5       6            -0.000636         -0.001212          -0.000233
6       7            -0.000541         -0.001424          -0.000203
```

(continues on next page)

(continued from previous page)

```
7       8            -0.000652         -0.001669          -0.000272
8       9            -0.000800         -0.001582          -0.000393
9      10            -0.000971         -0.001775          -0.000466
10     11            -0.001314         -0.002744          -0.000514
11     12            -0.001239         -0.002816          -0.000782

    interval_count
0                5
1                7
2                7
3                9
4                5
5                7
6                7
7                8
8                8
9               10
10               8
11              10
```

[24]:
```
# Calculate and view annual insolation-weighted soiling ratios and their confidence
# intervals based on the Monte Carlo simulation. Note that these losses include the
# assumptions of the cleaning assumptions associated with the method parameter
# of rdtools.soiling_srr(). For anything but 'perfect_clean', each year's soiling
# ratio may be impacted by prior years' soiling profiles. The default behavior of
# rdtools.soiling_srr uses method='half_norm_clean'

from rdtools.soiling import annual_soiling_ratios
annual_soiling_ratios(soiling_info['stochastic_soiling_profiles'],
                      daily_insolation,
                      confidence_level=cl)
```

[24]:
```
   year  soiling_ratio_median  soiling_ratio_low  soiling_ratio_high
0  2010              0.963595           0.953683            0.970623
1  2011              0.957843           0.952174            0.962844
2  2012              0.932868           0.914425            0.946839
3  2013              0.963971           0.954543            0.970562
4  2014              0.923057           0.899209            0.945093
5  2015              0.963245           0.949240            0.972810
6  2016              0.958073           0.948505            0.964359
```

Clear sky workflow

The clear sky workflow is useful in that it avoids problems due to drift or recalibration of ground-based sensors. We use `pvlib` to model the clear sky irradiance. This is renormalized to align it with ground-based measurements. Finally we use `rdtools.get_clearsky_tamb()` to model the ambient temperature on clear sky days. This modeled ambient temperature is used to model cell temperature with `pvlib`. If high quality ambient temperature data is available, that can be used instead of the modeled ambient; we proceed with the modeled ambient temperature here for illustrative purposes.

In this example, note that we have omitted wind data in the cell temperature calculations for illustrative purposes. Wind data can also be included when the data source is trusted for improved results

We generally recommend that the clear sky workflow be used as a check on the sensor workflow. It tends to be more sensitive than the sensor workflow, and thus we don't recommend it as a stand-alone analysis.

Note that the calculations below rely on some objects from the steps above

Clear Sky 0: Preliminary Calculations

[25]:
```
# Calculate the clear sky POA irradiance

loc = pvlib.location.Location(meta['latitude'], meta['longitude'], tz = meta['timezone
↪'])
sun = loc.get_solarposition(df.index)
clearsky = loc.get_clearsky(df.index, solar_position=sun)

cs_irradiance = pvlib.irradiance.get_total_irradiance(meta['tilt'], meta['azimuth'],␣
↪sun['apparent_zenith'],
                                                      sun['azimuth'], clearsky['dni'],␣
↪clearsky['ghi'],
                                                      clearsky['dhi'])
df['clearsky_poa'] = cs_irradiance.poa_global
# Renormalize the clear sky POA irradiance
df['clearsky_poa'] = rdtools.irradiance_rescale(df.poa, df.clearsky_poa,
                                                method='iterative')

# Calculate the clearsky temperature
df['clearsky_Tamb'] = rdtools.get_clearsky_tamb(df.index, meta['latitude'],
                                                meta['longitude'])
df['clearsky_Tcell'] = pvlib.temperature.sapm_cell(df.clearsky_poa, df.clearsky_Tamb,
                                                   0, **meta['temp_model_params'])
```

Clear Sky 1: Normalize

Normalize as in step 1 above, but this time using clearsky modeled irradiance and cell temperature

[26]:
```
# Calculate the expected power with a simple PVWatts DC model
clearsky_modeled_power = pvlib.pvsystem.pvwatts_dc(df['clearsky_poa'],
                                                   df['clearsky_Tcell'],
                                                   meta['power_dc_rated'], meta[
↪'gamma_pdc'], 25.0 )

# Calculate the normalization, the function also returns the relevant insolation for
# each point in the normalized PV energy timeseries
clearsky_normalized, clearsky_insolation = rdtools.normalize_with_expected_power(
    df['power'],
    clearsky_modeled_power,
    df['clearsky_poa']
)

df['clearsky_normalized'] = clearsky_normalized
df['clearsky_insolation'] = clearsky_insolation
```

Clear Sky 2: Filter

Filter as in step 2 above, but with the addition of a clear sky index (csi) filter so we consider only points well modeled by the clear sky irradiance model.

[27]:
```
# Perform clearsky filter
cs_normalized_mask = rdtools.normalized_filter(df['clearsky_normalized'])
cs_poa_mask = rdtools.poa_filter(df['clearsky_poa'])
cs_tcell_mask = rdtools.tcell_filter(df['clearsky_Tcell'])

csi_mask = rdtools.csi_filter(df.insolation, df.clearsky_insolation)

clearsky_filtered = df[cs_normalized_mask & cs_poa_mask & cs_tcell_mask &
                       clip_mask & csi_mask]
clearsky_filtered = clearsky_filtered[['clearsky_insolation', 'clearsky_normalized']]
```

Clear Sky 3: Aggregate

Aggregate the clear sky version of of the filtered data

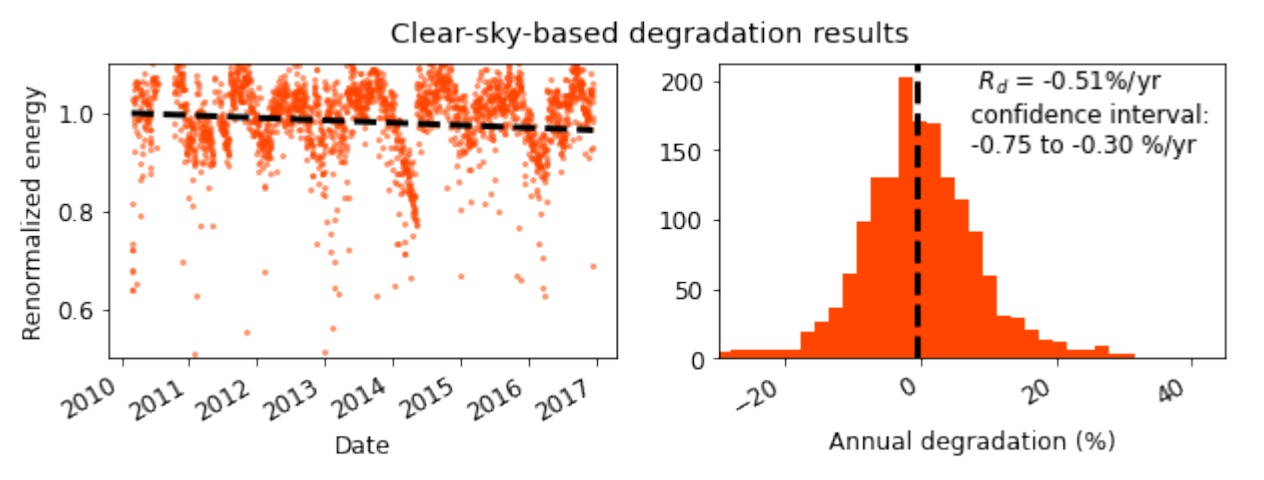
[28]:
```
clearsky_daily = rdtools.aggregation_insol(clearsky_filtered.clearsky_normalized,
                                           clearsky_filtered.clearsky_insolation)
```

Clear Sky 4: Degradation Calculation

Estimate the degradation rate and compare to the results obtained with sensors. In this case, we see that the degradation rate estimated with the clearsky methodology is not far off from the sensor-based estimate.

[29]:
```
# Calculate the degradation rate using the YoY method
cs_yoy_rd, cs_yoy_ci, cs_yoy_info = rdtools.degradation_year_on_year(
    clearsky_daily,
    confidence_level=68.2
)

# Note the default confidence_level of 68.2 is appropriate if you would like to
# report a confidence interval analogous to the standard deviation of a normal
# distribution. The size of the confidence interval is adjustable by setting the
# confidence_level variable.

# Visualize the results
clearsky_fig = rdtools.degradation_summary_plots(
    cs_yoy_rd, cs_yoy_ci, cs_yoy_info, clearsky_daily,
    summary_title='Clear-sky-based degradation results',
    scatter_ymin=0.5, scatter_ymax=1.1,
    hist_xmin=-30, hist_xmax=45, plot_color='orangered',
    bins=100);

print('The P95 exceedance level with the clear sky analysis is %.2f%%/yr' %
      cs_yoy_info['exceedance_level'])
```

```
The P95 exceedance level with the clear sky analysis is -0.90%/yr
```

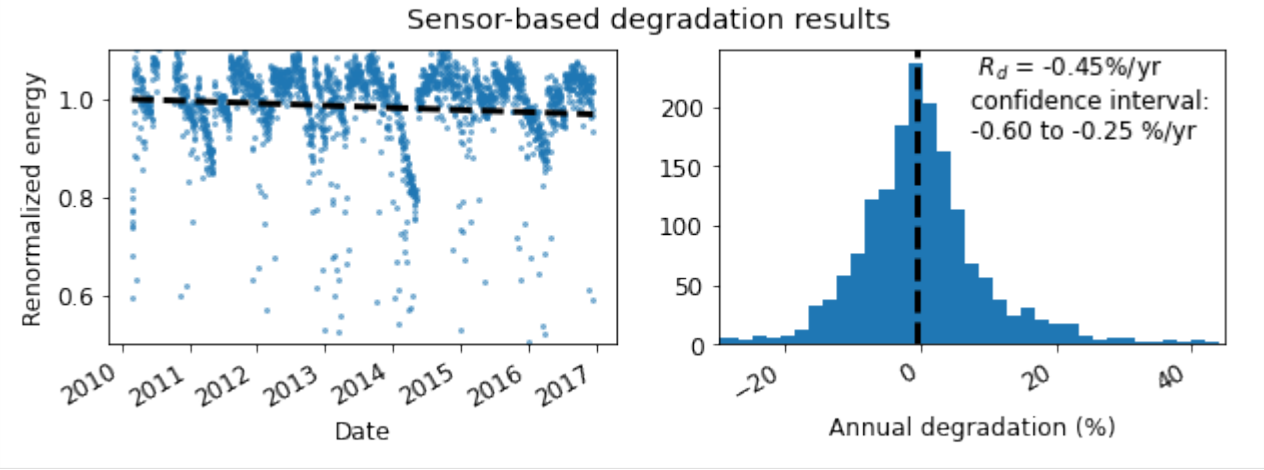


[30]:
```
print(cs_yoy_rd)
print(cs_yoy_ci)
```

```
-0.5089691988110401
[-0.7498017 -0.3010554]
```

[31]:
```
# Compare to previous sensor results
degradation_fig
```

[31]:



8.1.2 TrendAnalysis object-oriented example

This juypter notebook is intended to demonstrate the RdTools analysis workflow as implimented with the `rdtools.TrendAnalysis` object-oriented API. For a consistent experience, we recommend installing the packages and versions documented in `docs/notebook_requirements.txt`. This can be achieved in your environment by running `pip install -r docs/notebook_requirements.txt` from the base directory. (RdTools must also be separately installed.)

The calculations consist of two phases 1. Import and preliminary calculations: In this step data is important and augmented to enable analysis with RdTools. **No RdTools functions are used in this step. It will vary depending on the particulars of your dataset. Be sure to understand the inputs RdTools requires and make appropriate adjustments.**

2. Analysis with RdTools: This notebook illustrates the use of the TrendAnalysis API to perform sensor and clearsky degradation rate calculations along with stocasithc rate and recovery (SRR) soiling calculations.

This notebook works with data from the NREL PVDAQ `[4] NREL x-Si #1` system. Note that because this system does not experience significant soiling, the dataset contains a synthesized soiling signal for use in the soiling section of the example. This notebook automatically downloads and locally caches the dataset used in this example. The data can also be found on the DuraMAT Datahub (https://datahub.duramat.org/dataset/pvdaq-time-series-with-soiling-signal).

[1]:
```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import pvlib
import rdtools
%matplotlib inline
```

[2]:
```
#Update the style of plots
import matplotlib
matplotlib.rcParams.update({'font.size': 12,
                            'figure.figsize': [4.5, 3],
                            'lines.markeredgewidth': 0,
                            'lines.markersize': 2
                            })
# Register time series plotting in pandas > 1.0
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()
```

[3]:
```
# Set the random seed for numpy to ensure consistent results
np.random.seed(0)
```

Import and preliminary calculations

This section prepares the data necessary for an `rdtools` calculation.

A common challenge is handling datasets with and without daylight savings time. Make sure to specify a `pytz` timezone that does or does not include daylight savings time as appropriate for your dataset.

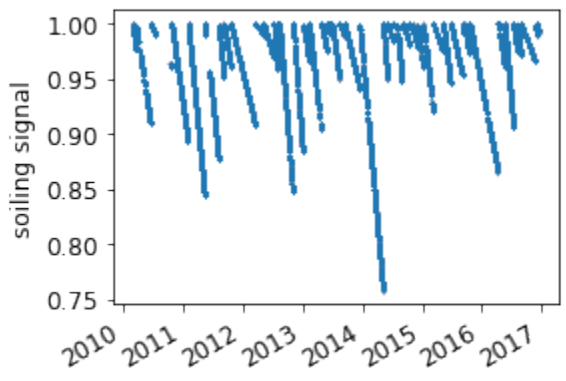
The steps of this section may change depending on your data source or the system being considered. Note that nothing in this first section utilizes the ``rdtools`` library. Transposition of irradiance and modeling of cell temperature are generally outside the scope of `rdtools`. A variety of tools for these calculations are available in pvlib.

[4]:
```
# Import the example data
file_url = ('https://datahub.duramat.org/dataset/a49bb656-7b36-'
            '437a-8089-1870a40c2a7d/resource/5059bc22-640d-4dd4'
            '-b7b1-1e71da15be24/download/pvdaq_system_4_2010-2016'
            '_subset_soilsignal.csv')
cache_file = 'PVDAQ_system_4_2010-2016_subset_soilsignal.pickle'

try:
    df = pd.read_pickle(cache_file)
except FileNotFoundError:
    df = pd.read_csv(file_url, index_col=0, parse_dates=True)
    df.to_pickle(cache_file)
```

[5]:
```
df = df.rename(columns = {
    'ac_power':'power_ac',
    'wind_speed': 'wind_speed',
    'ambient_temp': 'Tamb',
    'poa_irradiance': 'poa',
})

# Specify the Metadata
meta = {"latitude": 39.7406,
        "longitude": -105.1774,
        "timezone": 'Etc/GMT+7',
        "gamma_pdc": -0.005,
        "azimuth": 180,
        "tilt": 40,
        "power_dc_rated": 1000.0,
        "temp_model_params":'open_rack_glass_polymer'}

df.index = df.index.tz_localize(meta['timezone'])

# Set the pvlib location
loc = pvlib.location.Location(meta['latitude'], meta['longitude'], tz = meta['timezone
↪'])

# There is some missing data, but we can infer the frequency from
# the first several data points
freq = pd.infer_freq(df.index[:10])
```

This example dataset includes a synthetic soiling signal that can be applied onto the PV power data to illustrate the soiling loss and detection capabilities of RdTools. AC Power is multiplied by soiling to create the synthetic 'power' channel

[6]:
```
# Plot synthetic soiling signal attached to the dataset
fig, ax = plt.subplots(figsize=(4,3))
ax.plot(df.index, df.soiling, 'o', alpha=0.01)
#ax.set_ylim(0,1500)
fig.autofmt_xdate()
ax.set_ylabel('soiling signal');

df['power'] = df['power_ac'] * df['soiling']
```



Use of the object oriented system analysis API

The first step is to create a `TrendAnalysis` instance containing data to be analyzed and information about the system. Here we illustrate a basic applicaiton, but the API is highly customizable and we encourage you to read the docstrings and check the source for full details.

[7]:
```
ta = rdtools.TrendAnalysis(df['power'], df['poa'],
                           temperature_ambient=df['Tamb'],
                           gamma_pdc=meta['gamma_pdc'],
                           interp_freq=freq,
                           windspeed=df['wind_speed'],
                           power_dc_rated=meta['power_dc_rated'],
                           temperature_model=meta['temp_model_params'])
```

A second step is required to set up the clearsky model, which needs to be localized and have tilt/orientation information.

[8]:
```
ta.set_clearsky(pvlib_location=loc, pv_tilt=meta['tilt'], pv_azimuth=meta['azimuth'],␣
↪albedo=0.25)
```

Once the `TrendAnalysis` object is ready, the `sensor_analysis()` and `clearsky_analysis()` methods can be used to deploy the full chain of analysis steps. Results are stored in nested dict, `TrendAnalysis.results`.

Filters utilized in the analysis can be adjusted by changing the dict `TrendAnalysis.filter_params`.

Here we also illustrate how the aggregated data can be used to estimate soiling losses using the stochastic rate and recovery (SRR) method.1 Since our example system doesn't experience much soiling, we apply an artificially generated soiling signal, just for the sake of example. If the dataset doesn't require special treatment, the 'srr_soiling' analysis type can just be added to the `sensor_analysis` function below. A warning is emitted here because the underlying RdTools soiling module is still experimental and subject to change.

1M. G. Deceglie, L. Micheli and M. Muller, "Quantifying Soiling Loss Directly From PV Yield," IEEE Journal of Photovoltaics, vol. 8, no. 2, pp. 547-551, March 2018. doi: 10.1109/JPHOTOV.2017.2784682

[9]:
```
ta.sensor_analysis(analyses=['yoy_degradation','srr_soiling'])
ta.clearsky_analysis()
```

```
/Users/mdecegli/opt/anaconda3/envs/xgboost/lib/python3.7/site-packages/rdtools/
↪soiling.py:15: UserWarning: The soiling module is currently experimental. The API,␣
↪results, and default behaviors may change in future releases (including MINOR and␣
↪PATCH releases) as the code matures.
  'The soiling module is currently experimental. The API, results, '
/Users/mdecegli/opt/anaconda3/envs/xgboost/lib/python3.7/site-packages/rdtools/
↪soiling.py:366: UserWarning: 20% or more of the daily data is assigned to invalid␣
↪soiling intervals. This can be problematic with the "half_norm_clean" and "random_
↪clean" cleaning assumptions. Consider more permissive validity criteria such as␣
↪increasing "max_relative_slope_error" and/or "max_negative_step" and/or decreasing
↪"min_interval_length". Alternatively, consider using method="perfect_clean". For␣
↪more info see https://github.com/NREL/rdtools/issues/272
  warnings.warn('20% or more of the daily data is assigned to invalid soiling '
```

The results of the calculations are stored in a nested dict, `TrendAnalysis.results`

[10]:
```
yoy_results = ta.results['sensor']['yoy_degradation']
srr_results = ta.results['sensor']['srr_soiling']
```

[11]:
```
# Print the sensor-based analysis p50 degradation rate and confidence interval
print(yoy_results['p50_rd'])
print(yoy_results['rd_confidence_interval'])
```

```
-0.4540480935921178
[-0.60485954 -0.25286447]
```

[12]:
```
# Print the clear-sky-based analysis p50 degradation rate and confidence interval
cs_yoy_results = ta.results['clearsky']['yoy_degradation']
print(cs_yoy_results['p50_rd'])
print(cs_yoy_results['rd_confidence_interval'])
```

```
-0.5089691997338827
[-0.76093848 -0.29500314]
```

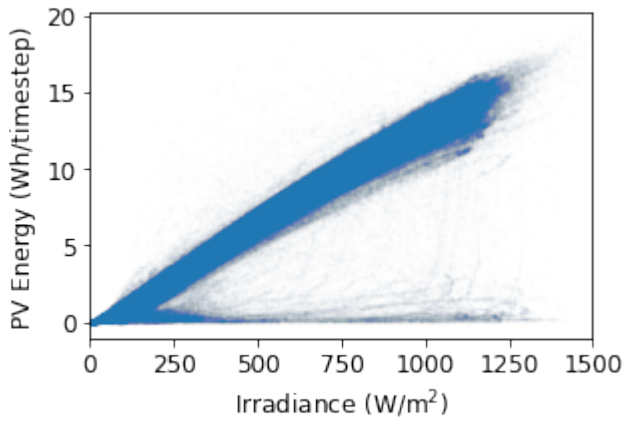
[13]:
```
# Print the p50 inoslation-weighted soiling ration and confidence interval
print(srr_results['p50_sratio'])
print(srr_results['sratio_confidence_interval'])
```

```
0.9493907403580503
[0.94404482 0.95447821]
```

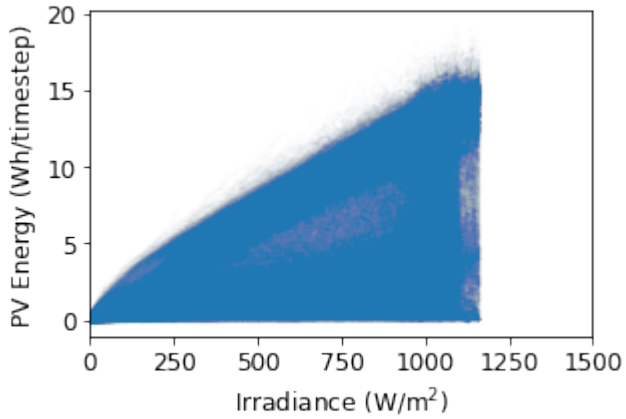
Plotting

The `TrendAnalysis` class has built in methods for making useful plots

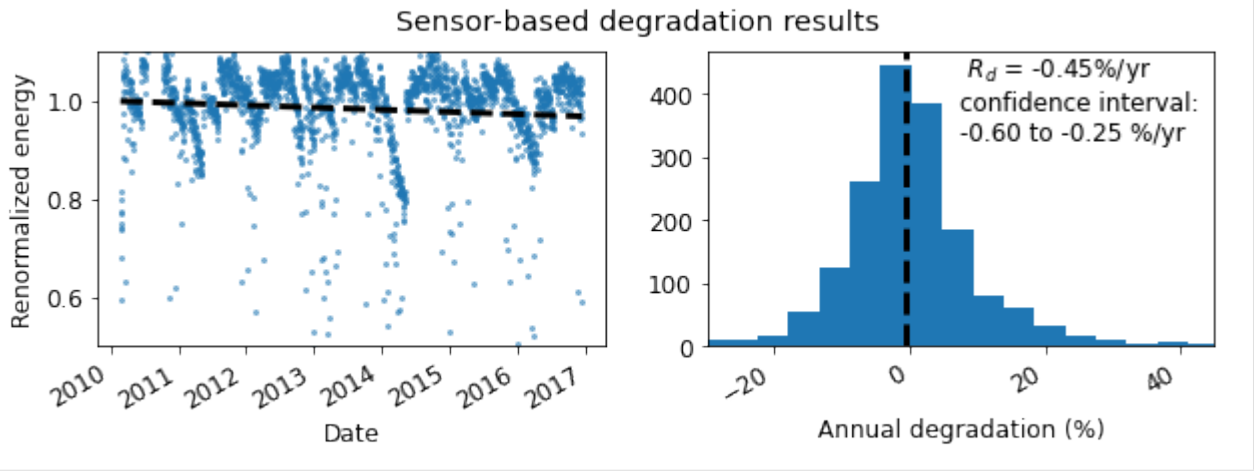
[14]:
```
# check that PV energy is roughly proportional to irradiance
# Loops and other features in this plot can indicate things like
# time zone problems for irradiance transposition errors.
fig = ta.plot_pv_vs_irradiance('sensor');
```



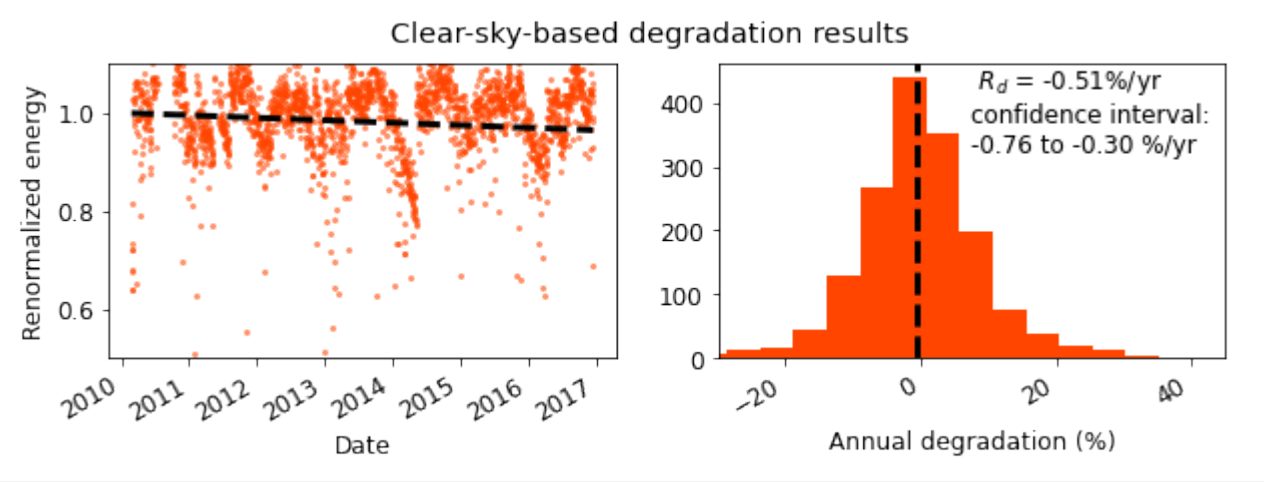
[15]:
```
# Repeat the check for clear-sky irradiance
# For this plot, we expect more points below the main point
# cloud due to cloudy conditions.
fig = ta.plot_pv_vs_irradiance('clearsky');
```



[16]:
```
# Plot the sensor based degradation results
fig = ta.plot_degradation_summary('sensor', summary_title='Sensor-based degradation␣
↪results',
                                  scatter_ymin=0.5, scatter_ymax=1.1,
                                  hist_xmin=-30, hist_xmax=45);
```



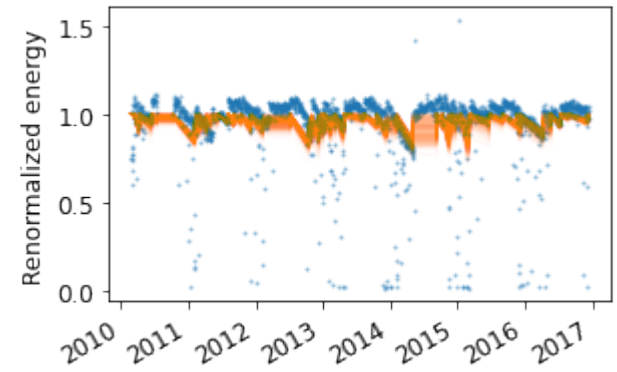
[17]:
```
# Plot the clear-sky-based results
fig = ta.plot_degradation_summary('clearsky', summary_title='Clear-sky-based␣
↪degradation results',
                                  scatter_ymin=0.5, scatter_ymax=1.1,
                                  hist_xmin=-30, hist_xmax=45, plot_color='orangered
↪');
```



The TrendAnalysis class also has built-in methods for plots associated with soiling analysis

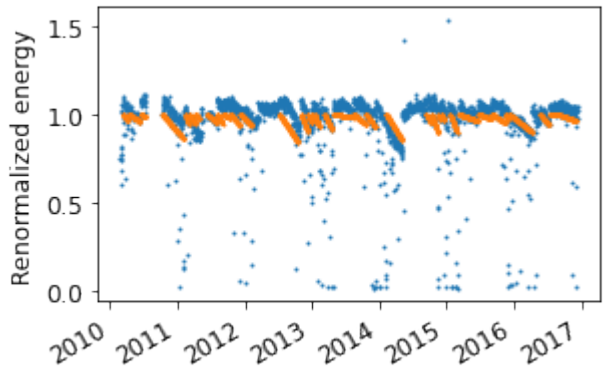
[18]:
```
fig = ta.plot_soiling_monte_carlo('sensor', profile_alpha=0.01, profiles=500);
```

```
/Users/mdecegli/opt/anaconda3/envs/xgboost/lib/python3.7/site-packages/rdtools/
↪plotting.py:166: UserWarning: The soiling module is currently experimental. The API,
↪ results, and default behaviors may change in future releases (including MINOR and␣
↪PATCH releases) as the code matures.
  'The soiling module is currently experimental. The API, results, '
```



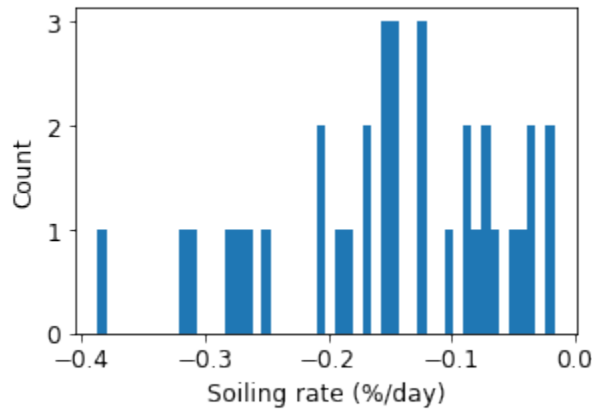
[19]:
```
fig = ta.plot_soiling_interval('sensor');
```

```
/Users/mdecegli/opt/anaconda3/envs/xgboost/lib/python3.7/site-packages/rdtools/
↪plotting.py:226: UserWarning: The soiling module is currently experimental. The API,
↪ results, and default behaviors may change in future releases (including MINOR and␣
↪PATCH releases) as the code matures.
  'The soiling module is currently experimental. The API, results, '
```



[20]:
```
fig = ta.plot_soiling_rate_histogram('sensor', bins=50);
```

```
/Users/mdecegli/opt/anaconda3/envs/xgboost/lib/python3.7/site-packages/rdtools/
↪plotting.py:266: UserWarning: The soiling module is currently experimental. The API,
↪ results, and default behaviors may change in future releases (including MINOR and␣
↪PATCH releases) as the code matures.
  'The soiling module is currently experimental. The API, results, '
```



We can also view a table of the valid soiling intervals and associated metrics

[21]:
```
interval_summary = ta.results['sensor']['srr_soiling']['calc_info']['soiling_interval_
↪summary']
interval_summary[interval_summary['valid']].head()
```

[21]:

	start	end	soiling_rate	soiling_rate_low	soiling_rate_high	inferred_start_loss
6	2010-03-11 00:00:00-07:00	2010-04-08 00:00:00-07:00	-0.001467	-0.004335	0.000000	1.037802
8	2010-04-12 00:00:00-07:00	2010-06-13 00:00:00-07:00	-0.000701	-0.001058	-0.000336	1.012347
10	2010-06-15 00:00:00-07:00	2010-07-13 00:00:00-07:00	-0.000639	-0.001763	0.000000	1.082671
13	2010-10-11 00:00:00-07:00	2011-02-04 00:00:00-07:00	-0.001216	-0.001407	-0.001022	1.064267
16	2011-02-13 00:00:00-07:00	2011-03-03 00:00:00-07:00	-0.003159	-0.005266	-0.001054	1.028467

(continues on next page)

(continued from previous page)

```
    inferred_end_loss  length  valid
6            0.996718      28   True
8            0.968893      62   True
10           1.064776      28   True
13           0.923230     116   True
16           0.971602      18   True
```

Modifying and inspecting the filters

Filters can be adjusted from their default paramters by modifying the attribute `TrendAnalysis.filter_params`, which is a dict where the keys are names of functions in `rdtools.filtering`, and the values are a dict of the parameters for the associated filter. In the following example we modify the POA filter to have a low cutoff of 500 W/m^2 and use an alternate method in the clipping filter.

[22]:
```
# Instantiate a new instance of TrendAnalysis
ta_new_filter = rdtools.TrendAnalysis(df['power'], df['poa'],
                                      temperature_ambient=df['Tamb'],
                                      gamma_pdc=meta['gamma_pdc'],
                                      interp_freq=freq,
                                      windspeed=df['wind_speed'],
                                      temperature_model=meta['temp_model_params'])

# Modify the poa and clipping filter parameters
ta_new_filter.filter_params['poa_filter'] = {'poa_global_low':500}
ta_new_filter.filter_params['clip_filter'] = {'model':'xgboost'}

# Run the YOY degradation analysis
ta_new_filter.sensor_analysis()
```

```
/Users/mdecegli/opt/anaconda3/envs/xgboost/lib/python3.7/site-packages/rdtools/
↪filtering.py:641: UserWarning: The XGBoost filter is an experimental clipping␣
↪filter that is still under development. The API, results, and default behaviors may␣
↪change in future releases (including MINOR and PATCH). Use at your own risk!
  warnings.warn("The XGBoost filter is an experimental clipping filter "
```

[23]:
```
# We can inspect the filter components with the attributes sensor_filter_components␣
↪and clearsky_filter_components
ta_new_filter.sensor_filter_components.head()
```

[23]:
```
                           normalized_filter  poa_filter  tcell_filter  \
2010-02-25 14:16:00-07:00              False       False          True
2010-02-25 14:17:00-07:00               True       False          True
2010-02-25 14:18:00-07:00               True       False          True
2010-02-25 14:19:00-07:00               True       False          True
2010-02-25 14:20:00-07:00               True       False          True

                           clip_filter
2010-02-25 14:16:00-07:00         True
2010-02-25 14:17:00-07:00         True
2010-02-25 14:18:00-07:00         True
2010-02-25 14:19:00-07:00         True
2010-02-25 14:20:00-07:00         True
```

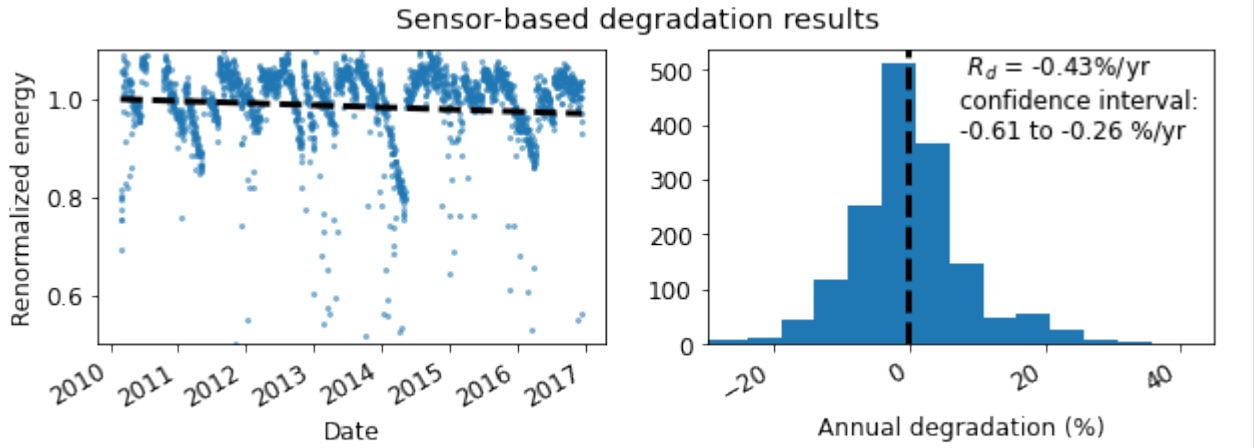
[24]:
```
# and we can inspect the final filter sensor_filter and clearsky_filter
ta_new_filter.sensor_filter.head()
```

[24]:
```
2010-02-25 14:16:00-07:00    False
2010-02-25 14:17:00-07:00    False
2010-02-25 14:18:00-07:00    False
2010-02-25 14:19:00-07:00    False
2010-02-25 14:20:00-07:00    False
Freq: T, dtype: bool
```

[25]:
```
# Visualize the filter
# these visualiztions can take up a lot of memory
# so just look at small subset of the data for this example
rdtools.tune_filter_plot(ta_new_filter.pv_energy['2013/01/01':'2013/01/21'],
                         ta_new_filter.sensor_filter['2013/01/01':'2013/01/21'])
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

[26]:
```
# Visualize the results
ta_new_filter.plot_degradation_summary('sensor', summary_title='Sensor-based␣
↪degradation results',
                                       scatter_ymin=0.5, scatter_ymax=1.1,
                                       hist_xmin=-30, hist_xmax=45);
```



Using externally calculated filters

Arbitrary filters can also be used by setting the `ad_hoc_filter` key of the `TrendAnalysis.filter_params` atribute to a boolean pandas series that can be used as a filter. In this example we filter for "stuck" values, i.e. values that are repeated consecuatively, which can be associated with faulty measurments.

[27]:
```
def filter_stuck_values(pandas_series):
    '''
    Returns a boolean pd.Series which filters out sequentially
    repeated values from pandas_series'
    '''
    diff = pandas_series.diff()
    diff_shift = diff.shift(-1)

    stuck_filter = ~((diff == 0) | (diff_shift == 0))
```

(continues on next page)

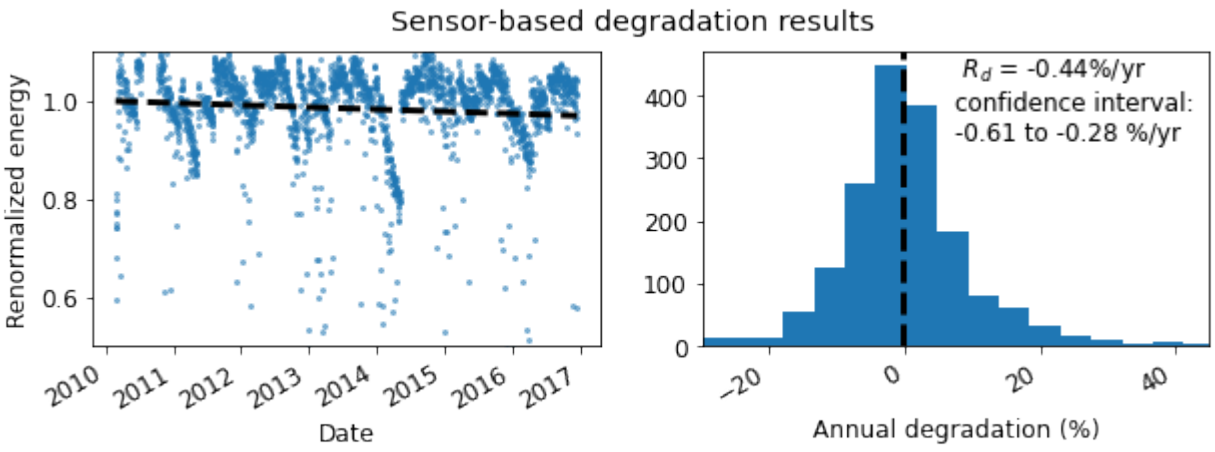
(continued from previous page)

```
    return stuck_filter
```

[28]:
```
# Instantiate a new instance of TrendAnalysis
ta_stuck_filter = rdtools.TrendAnalysis(df['power'], df['poa'],
                                        temperature_ambient=df['Tamb'],
                                        gamma_pdc=meta['gamma_pdc'],
                                        interp_freq=freq,
                                        windspeed=df['wind_speed'],
                                        temperature_model=meta['temp_model_params'])
```

[29]:
```
stuck_filter =  (
                  filter_stuck_values(df['power']) &
                  filter_stuck_values(df['poa']) &
                  filter_stuck_values(df['Tamb']) &
                  filter_stuck_values(df['wind_speed'])
                 )

# reindex onto the same index that will be used for the other filters
stuck_filter = stuck_filter.reindex(ta_stuck_filter.poa_global.index, fill_value=True)

ta_stuck_filter.filter_params['ad_hoc_filter'] = stuck_filter
```

[30]:
```
ta_stuck_filter.sensor_analysis()

# Visualize the results
ta_stuck_filter.plot_degradation_summary('sensor', summary_title='Sensor-based␣
↪degradation results',
                                         scatter_ymin=0.5, scatter_ymax=1.1,
                                         hist_xmin=-30, hist_xmax=45);
```



8.1.3 System availability example

This notebook shows example usage of the inverter availability functions. As with the degradation and soiling example, we recommend installing the specific versions of packages used to develop this notebook. This can be achieved in your environment by running `pip install -r requirements.txt` followed by `pip install -r docs/notebook_requirements.txt` from the base directory. (RdTools must also be separately installed.) These environments and examples are tested with Python 3.7.

RdTools currently implements two methods of quantifying system availability. The first method compares power measurements from inverters and the system meter to distinguish subsystem communication interruptions from true outage events. The second method determines the uncertainty bounds around an energy estimate of a total system outage and compares with true production calculated from a meter's cumulative production measurements. The RdTools `AvailabilityAnalysis` class uses both methods to quantify downtime loss.

These methods are described in K. Anderson and R. Blumenthal, "Overcoming Communications Outages in Inverter Downtime Analysis", 2020 IEEE 47th Photovoltaic Specialists Conference (PVSC).

[1]:
```
import rdtools
import pvlib

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

Quantifying the production impact of inverter downtime events is complicated by gaps in a system's historical data caused by communication interruptions. Although communication interruptions may prevent remote operation, they usually do not result in production loss. Accurate production loss estimates require the ability to distinguish true outages from communication interruptions.

The first method focuses on partial outages where some of a system's inverters are reporting production and some are not. In these cases, the method examines the AC power measurements at the inverter and system meter level to classify each timestamp individually and estimate timeseries production loss. This level of granularity is made possible by comparing timeseries power measurements between inverters and the meter.

Create a test dataset

First we'll generate a test dataset to demonstrate the method. This code block just puts together an artificial dataset to use for the analysis – feel free to skip ahead to where it gets plotted.

[2]:
```
def make_dataset():
    """
    Make an example dataset with several types of data outages for availability␣
↪analysis.

    Returns
    -------
    df_reported : pd.DataFrame
        Simulated data as a data acquisition system would report it, including the
        effect of communication interruptions.
    df_secret : pd.DataFrame
        The secret true data of the system, not affected by communication
        interruptions.  Only used for comparison with the analysis output.
    expected_power : pd.Series
        An "expected" power signal for this hypothetical PV system, simulating a
        modeled power from satellite weather data or some other method.
```

(continues on next page)

(continued from previous page)

```
    (This function creates instantaneous data. SystemAvailability is technically␣
↪designed
    to work with right-labeled averages. However, for the purposes of the example, the
    approximation is suitable.)
    """

    # generate a plausible clear-sky power signal
    times = pd.date_range('2019-01-01', '2019-01-12', freq='15min', tz='US/Eastern',
                          closed='left')
    location = pvlib.location.Location(40, -80)
    clearsky = location.get_clearsky(times, model='haurwitz')
    # just scale GHI to power for simplicity
    base_power = 2.5*clearsky['ghi']
    # but require a minimum irradiance to turn on, simulating start-up voltage
    base_power[clearsky['ghi'] < 20] = 0

    df_secret = pd.DataFrame({
        'inv1_power': base_power,
        'inv2_power': base_power * 1.5,
        'inv3_power': base_power * 0.66,
    })

    # set the expected_power to be pretty close to actual power,
    # but with some autocorrelated noise and a bias:
    expected_power = df_secret.sum(axis=1)
    np.random.seed(2020)
    N = len(times)
    expected_power *= 0.9 - (0.3 * np.sin(np.arange(0, N)/7 +
                             np.random.normal(0, 0.2, size=N)))

    # Add a few days of individual inverter outages:
    df_secret.loc['2019-01-03':'2019-01-05', 'inv2_power'] = 0
    df_secret.loc['2019-01-02', 'inv3_power'] = 0
    df_secret.loc['2019-01-07 00:00':'2019-01-07 12:00', 'inv1_power'] = 0

    # and a full system outage:
    full_outage_date = '2019-01-08'
    df_secret.loc[full_outage_date, :] = 0

    # calculate the system meter power and cumulative production,
    # including the effect of the outages:
    df_secret['meter_power'] = df_secret.sum(axis=1)
    interval_energy = rdtools.energy_from_power(df_secret['meter_power'])
    df_secret['meter_energy'] = interval_energy.cumsum()
    # fill the first NaN from the cumsum with 0
    df_secret['meter_energy'] = df_secret['meter_energy'].fillna(0)
    # add an offset to reflect previous production:
    df_secret['meter_energy'] += 5e5
    # calculate cumulative energy for an inverter as well:
    inv2_energy = rdtools.energy_from_power(df_secret['inv2_power'])
    df_secret['inv2_energy'] = inv2_energy.cumsum().fillna(0)

    # now that the "true" data is in place, let's add some communications␣
↪interruptions:
    df_reported = df_secret.copy()
    # in full outages, we lose all the data:
    df_reported.loc[full_outage_date, :] = np.nan
```

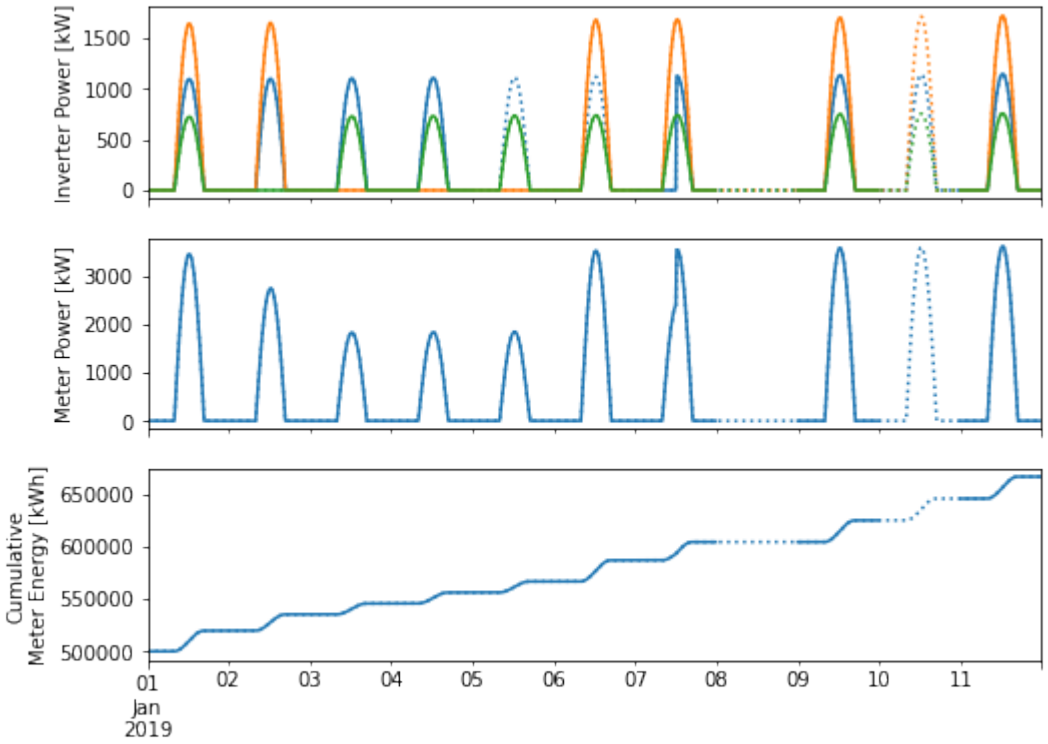
(continues on next page)

(continued from previous page)

```
    # add a communications interruption that overlaps with an inverter outage:
    df_reported.loc['2019-01-05':'2019-01-06', 'inv1_power'] = np.nan
    # and a communication outage that affects everything:
    df_reported.loc['2019-01-10', :] = np.nan

    return df_reported, df_secret, expected_power
```

Let's visualize the dataset before analyzing it with RdTools. The dotted lines show the "true" data that wasn't recorded by the datalogger because of interrupted communications.

[3]:
```
df, df_secret, expected_power = make_dataset()

fig, axes = plt.subplots(3, 1, sharex=True, figsize=(8,6))
colors = plt.rcParams['axes.prop_cycle'].by_key()['color'][:3]

# inverter power
df_secret[['inv1_power', 'inv2_power', 'inv3_power']].plot(ax=axes[0],
                                                           legend=False, ls=':',
                                                           color=colors)
df[['inv1_power', 'inv2_power', 'inv3_power']].plot(ax=axes[0], legend=False)
# meter power
df_secret['meter_power'].plot(ax=axes[1], ls=':', color=colors[0])
df['meter_power'].plot(ax=axes[1])
# meter cumulative energy
df_secret['meter_energy'].plot(ax=axes[2], ls=':', color=colors[0])
df['meter_energy'].plot(ax=axes[2])

axes[0].set_ylabel('Inverter Power [kW]')
axes[1].set_ylabel('Meter Power [kW]')
axes[2].set_ylabel('Cumulative\nMeter Energy [kWh]')
plt.show()
```



Note that the solid lines show the data that would be available in our example while the dotted lines show the true underlying behavior that we normally wouldn't know.

If we hadn't created this dataset ourselves, it wouldn't necessarily be obvious why the meter shows low or no production on some days – maybe it was just cloudy weather, maybe it was a nuisance communication outage (broken cell modem power supply, for example), or maybe it was a true power outage. This example also shows how an inverter can appear to be offline while actually producing normally. For example, just looking at inverter power on the 5th, it appears that only the small inverter is producing. However, the meter shows two inverters' worth of production. Similarly, the 6th shows full meter production despite one inverter not reporting power. Using only the inverter-reported power would overestimate the production loss because of the communication interruption.

System availability analysis

Now we'll hand this data off to RdTools for analysis:

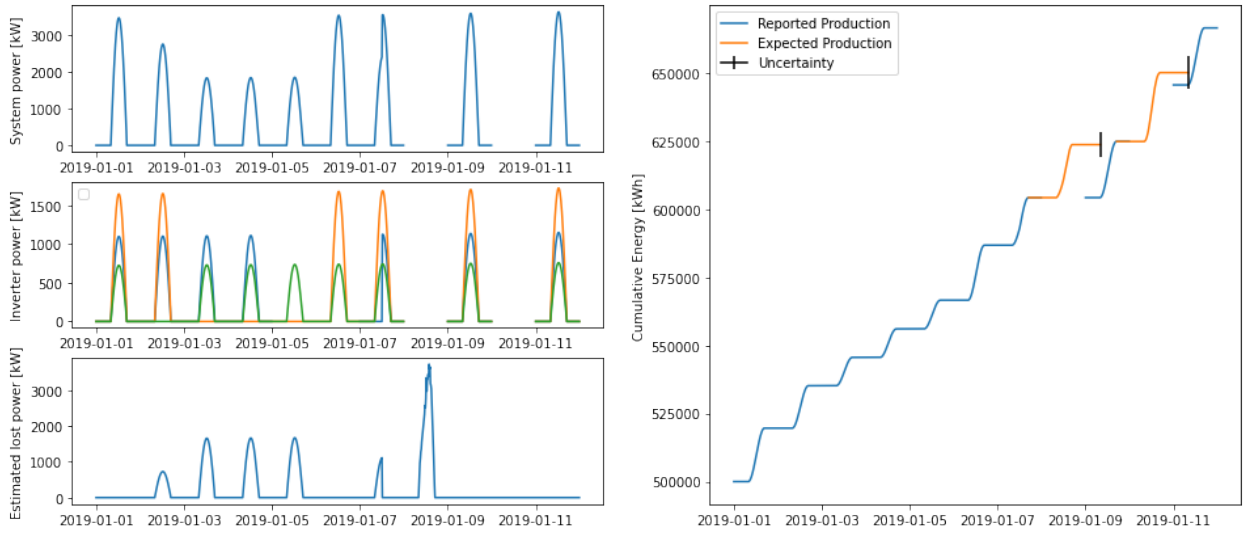
[4]:
```
from rdtools.availability import AvailabilityAnalysis
aa = AvailabilityAnalysis(
    power_system=df['meter_power'],
    power_subsystem=df[['inv1_power', 'inv2_power', 'inv3_power']],
    energy_cumulative=df['meter_energy'],
    power_expected=expected_power,
)
# identify and classify outages, rolling up to daily metrics for this short dataset:
aa.run(rollup_period='D')
```

```
c:\users\kanderso\software\anaconda3\envs\rdtools-dev\lib\site-packages\rdtools\
↪availability.py:18: UserWarning: The availability module is currently experimental.␣
↪The API, results, and default behaviors may change in future releases (including␣
↪MINOR and PATCH releases) as the code matures.
  'The availability module is currently experimental. The API, results, '
```

First, we can visualize the estimated power loss and outage information:

[5]:
```
fig = aa.plot()
fig.set_size_inches(16, 7)
fig.axes[1].legend(loc='upper left');
```

```
c:\users\kanderso\software\anaconda3\envs\rdtools-dev\lib\site-packages\rdtools\
↪plotting.py:386: UserWarning: The availability module is currently experimental.␣
↪The API, results, and default behaviors may change in future releases (including␣
↪MINOR and PATCH releases) as the code matures.
  'The availability module is currently experimental. The API, results, '
No artists with labels found to put in legend.  Note that artists whose label start␣
↪with an underscore are ignored when legend() is called with no argument.
```



Examining the plot of estimated lost power, we can see that the estimated loss is roughly in proportion to the amount of offline capacity. In particular, the loss estimate is robust to mixed outage and communication interruption like on the 5th when only the smallest inverter is reporting production but the analysis correctly inferred that one of the other inverters is producing but not communicating.

RdTools also reports rolled-up production and availability metrics:

[6]:
```
pd.set_option('precision', 3)
aa.results
```

[6]:

	lost_production	actual_production	availability
2019-01-01 00:00:00-05:00	0.000	19606.785	1.000
2019-01-02 00:00:00-05:00	4114.031	15583.450	0.791
2019-01-03 00:00:00-05:00	9396.788	10399.112	0.525
2019-01-04 00:00:00-05:00	9466.477	10476.235	0.525
2019-01-05 00:00:00-05:00	9522.325	10538.040	0.525
2019-01-06 00:00:00-05:00	0.000	20185.784	1.000
2019-01-07 00:00:00-05:00	2859.565	17459.339	0.859
2019-01-08 00:00:00-05:00	19448.084	0.000	0.000
2019-01-09 00:00:00-05:00	0.000	20607.950	1.000
2019-01-10 00:00:00-05:00	0.000	20763.718	1.000
2019-01-11 00:00:00-05:00	0.000	20926.869	1.000

The `AvailabilityAnalysis` object has other attributes that may be useful to inspect as well. The `outage_info` dataframe has one row for each full system outage with several columns, perhaps the most interesting of which are `type` and `loss`.

See `AvailabilityAnalysis?` or `help(AvailabilityAnalysis)` for full descriptions of the available attributes.

[7]:
```
pd.set_option('precision', 2)
# Show the first half of the dataframe
N = len(aa.outage_info.columns)
aa.outage_info.iloc[:, :N//2]
```

[7]:
```
                      start                       end        duration  \
0 2019-01-07 17:00:00-05:00 2019-01-09 08:00:00-05:00 1 days 15:00:00
1 2019-01-09 17:00:00-05:00 2019-01-11 08:00:00-05:00 1 days 15:00:00

   intervals  daylight_intervals  error_lower  error_upper
0        157                  35        -0.24         0.25
1        157                  35        -0.24         0.25
```

[8]:
```
# Show the second half
aa.outage_info.iloc[:, N//2:]
```

[8]:
```
   energy_expected  energy_start  energy_end  energy_actual  ci_lower  \
0         19448.08     604248.74   604248.74           0.00  14819.33
1         25284.75     624856.69   645620.41       20763.72  19266.84

   ci_upper   type      loss
0  24271.15   real  19448.08
1  31555.29  comms      0.00
```

Other use cases

Although this demo applies the methods for an entire PV system (comparing inverters against the meter and comparing the meter against expected power), it can also be used at the individual inverter level. Because there are no subsystems to compare against, the "full outage" analysis branch is used for every outage. That means that instead of basing the loss off of the other inverters, it relies on the expected power time series being accurate, which in this example causes the loss estimates to lose some accuracy. In this case, because the expected power signal is somewhat inaccurate, it causes the loss estimate to be overestimated:

[9]:
```
# make a new analysis object:
aa2 = rdtools.availability.AvailabilityAnalysis(
    power_system=df['inv2_power'],
    power_subsystem=df['inv2_power'].to_frame(),
    energy_cumulative=df['inv2_energy'],
    # okay to use the system-level expected power here because it gets rescaled anyway
    power_expected=expected_power,
)
# identify and classify outages, rolling up to daily metrics for this short dataset:
aa2.run(rollup_period='D')
print(aa2.results['lost_production'])
```

```
2019-01-01 00:00:00-05:00        0.00
2019-01-02 00:00:00-05:00        0.00
2019-01-03 00:00:00-05:00     9931.24
2019-01-04 00:00:00-05:00    11453.27
2019-01-05 00:00:00-05:00    11238.57
2019-01-06 00:00:00-05:00        0.00
2019-01-07 00:00:00-05:00        0.00
2019-01-08 00:00:00-05:00     9505.33
2019-01-09 00:00:00-05:00        0.00
```

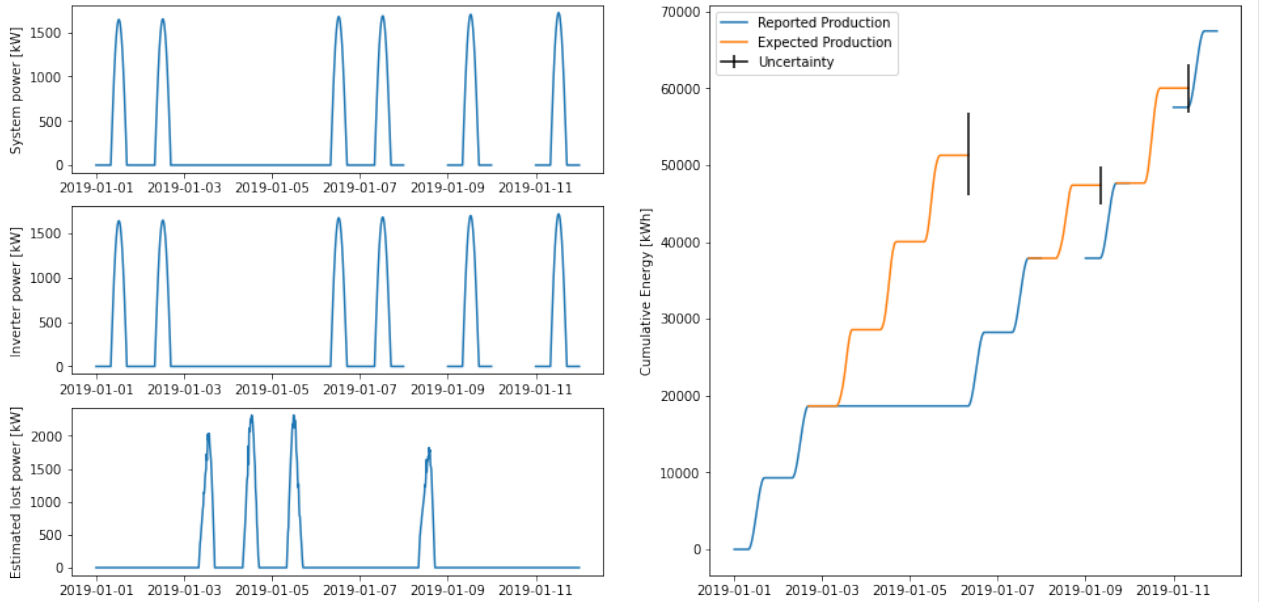
(continues on next page)

(continued from previous page)

```
2019-01-10 00:00:00-05:00        0.00
2019-01-11 00:00:00-05:00        0.00
Freq: D, Name: lost_production, dtype: float64
```

[10]:
```
aa2.plot();
```

```
c:\users\kanderso\software\anaconda3\envs\rdtools-dev\lib\site-packages\rdtools\
↪plotting.py:386: UserWarning: The availability module is currently experimental.␣
↪The API, results, and default behaviors may change in future releases (including␣
↪MINOR and PATCH releases) as the code matures.
  'The availability module is currently experimental. The API, results, '
```



8.2 API reference

8.2.1 Submodules

RdTools is organized into submodules focused on different parts of the data analysis workflow.

analysis_chains	This module contains functions and classes for object-oriented end-to-end analysis
degradation	Functions for calculating the degradation rate of photovoltaic systems.
soiling	Functions for calculating soiling metrics from photovoltaic system data.
availability	Functions for detecting and quantifying production loss from photovoltaic system downtime events.
filtering	Functions for filtering and subsetting PV system data.
normalization	Functions for normalizing, rescaling, and regularizing PV system data.

continues on next page

Table 1 – continued from previous page

aggregation	Functions for calculating weighted aggregates of PV system data.
clearsky_temperature	Functions for estimating clear-sky ambient temperature.
plotting	Functions for plotting degradation and soiling analysis results.

rdtools.analysis_chains

This module contains functions and classes for object-oriented end-to-end analysis

Classes

TrendAnalysis(pv[, poa_global, ...])	Class for end-to-end degradation and soiling analysis using `sensor_analysis()` or `clearsky_analysis()`

rdtools.degradation

Functions for calculating the degradation rate of photovoltaic systems.

Functions

degradation_classical_decomposition(...[, ...])	Estimate the trend of a timeseries using a classical decomposition approach (moving average) and calculate various statistics, including the result of a Mann-Kendall test and a Monte Carlo-derived confidence interval of slope.
degradation_ols(energy_normalized[, ...])	Estimate the trend of a timeseries using ordinary least-squares regression and calculate various statistics including a Monte Carlo-derived confidence interval of slope.
degradation_year_on_year(energy_normalized)	Estimate the trend of a timeseries using the year-on-year decomposition approach and calculate a Monte Carlo-derived confidence interval of slope.

rdtools.soiling

Functions for calculating soiling metrics from photovoltaic system data.

The soiling module is currently experimental. The API, results, and default behaviors may change in future releases (including MINOR and PATCH releases) as the code matures.

Functions

annual_soiling_ratios(...[, confidence_level])	Return annualized soiling ratios and associated confidence intervals based on stochastic soiling profiles from SRR.
monthly_soiling_rates(soiling_interval_summary)	Use Monte Carlo to calculate typical monthly soiling rates.
soiling_srr(energy_normalized_daily, ...[, ...])	Functional wrapper for *SRRAnalysis*.

Classes

SRRAnalysis(energy_normalized_daily, ...[, ...])	Class for running the stochastic rate and recovery (SRR) photovoltaic soiling loss analysis presented in Deceglie et al.

Exceptions

`NoValidIntervalError`	raised when no valid rows appear in the result dataframe

rdtools.availability

Functions for detecting and quantifying production loss from photovoltaic system downtime events.

The availability module is currently experimental. The API, results, and default behaviors may change in future releases (including MINOR and PATCH releases) as the code matures.

Classes

AvailabilityAnalysis(power_system, ...)	A class to perform system availability and loss analysis.

rdtools.filtering

Functions for filtering and subsetting PV system data.

Functions

clip_filter(power_ac[, model])	Master wrapper for running one of the desired clipping filters.
csi_filter(poa_global_measured, ...[, threshold])	Filtering based on clear-sky index (csi)
logic_clip_filter(power_ac[, mounting_type, ...])	This filter is a logic-based filter that is used to filter out clipping periods in AC power or energy time series.
normalized_filter(energy_normalized[, ...])	Select normalized yield between `low_cutoff` and `high_cutoff`

continues on next page

Table 8 – continued from previous page

poa_filter(poa_global[, poa_global_low, ...])	Filter POA irradiance readings outside acceptable measurement bounds.
quantile_clip_filter(power_ac[, quantile])	Filter data points likely to be affected by clipping with power or energy greater than or equal to 99% of the *quant* quantile.
tcell_filter(temperature_cell[, ...])	Filter temperature readings outside acceptable measurement bounds.
xgboost_clip_filter(power_ac[, mounting_type])	This function generates the features to run through the XGBoost clipping model, runs the data through the model, and generates model outputs.

rdtools.normalization

Functions for normalizing, rescaling, and regularizing PV system data.

Functions

check_series_frequency(series, ...)	Deprecated since version 2.0.0.
delta_index(series)	Deprecated since version 2.0.0.
energy_from_power(power[, target_frequency, ...])	Returns a regular right-labeled energy time series in units of Wh per interval from a power time series.
interpolate(time_series, target[, ...])	Returns an interpolation of time_series, excluding times associated with gaps in each column of time_series longer than max_timedelta; NaNs are returned within those gaps.
irradiance_rescale(irrad, irrad_sim[, ...])	Attempt to rescale modeled irradiance to match measured irradiance on clear days.
normalize_with_expected_power(pv, ...[, ...])	Normalize PV power or energy based on expected PV power.
normalize_with_pvwatts(energy, pvwatts_kws)	Normalize system AC energy output given measured poa_global and meteorological data.
normalize_with_sapm(energy, sapm_kws)	Deprecated since version 2.0.0.
pvwatts_dc_power(poa_global, power_dc_rated)	PVWatts v5 Module Model: DC power given effective poa poa_global, module nameplate power, and cell temperature.
sapm_dc_power(pvlib_pvsystem, met_data)	Deprecated since version 2.0.0.

Exceptions

`ConvergenceError`	Rescale optimization did not converge

rdtools.aggregation

Functions for calculating weighted aggregates of PV system data.

Functions

aggregation_insol(energy_normalized, insolation)	Insolation weighted aggregation

rdtools.clearsky_temperature

Functions for estimating clear-sky ambient temperature.

Functions

get_clearsky_tamb(times, latitude, longitude)	Estimates the ambient temperature at latitude and longitude for the given times using a Gaussian rolling window.

rdtools.plotting

Functions for plotting degradation and soiling analysis results.

Functions

availability_summary_plots(power_system, ...)	Create a figure summarizing the availability analysis results.
degradation_summary_plots(yoy_rd, yoy_ci, ...)	Create plots (scatter plot and histogram) that summarize degradation analysis results.
soiling_interval_plot(soiling_info, ...[, ...])	Create figure to visualize valid soiling profiles used in the SRR analysis.
soiling_monte_carlo_plot(soiling_info, ...)	Create figure to visualize Monte Carlo of soiling profiles used in the SRR analysis.
soiling_rate_histogram(soiling_info[, bins])	Create histogram of soiling rates found in the SRR analysis.
tune_filter_plot(signal, mask[, ...])	This function allows the user to visualize filtered data in a Plotly plot, after tweaking the function's different parameters.

8.2.2 Analysis Chains

Object-oriented end-to-end analysis

analysis_chains.TrendAnalysis(pv[, ...])	Class for end-to-end degradation and soiling analysis using `sensor_analysis()` or `clearsky_analysis()`
analysis_chains.TrendAnalysis.set_clearsky([...])	Initialize values for a clearsky analysis which requires configuration of location and orientation details.
analysis_chains.TrendAnalysis.sensor_analysis([...])	Perform entire sensor-based analysis workflow.
analysis_chains.TrendAnalysis.clearsky_analysis([...])	Perform entire clear-sky-based analysis workflow.
analysis_chains.TrendAnalysis.plot_degradation_summary(...)	Return a figure of a scatter plot and a histogram summarizing degradation rate analysis.
analysis_chains.TrendAnalysis.plot_soiling_rate_histogram(...)	Return a histogram of soiling rates found in the stochastic rate and recovery soiling analysis
analysis_chains.TrendAnalysis.plot_soiling_interval(...)	Return a figure visualizing the valid soiling intervals used in stochastic rate and recovery soiling analysis.
analysis_chains.TrendAnalysis.plot_soiling_monte_carlo(...)	Return a figure visualizing the Monte Carlo of soiling profiles used in stochastic rate and recovery soiling analysis.
analysis_chains.TrendAnalysis.plot_pv_vs_irradiance(case)	Plot PV energy vs irradiance, useful in diagnosing things like timezone problems or transposition errors.

rdtools.analysis_chains.TrendAnalysis

class rdtools.analysis_chains.**TrendAnalysis**(*pv*, *poa_global=None*, *temperature_cell=None*, *temperature_ambient=None*, *gamma_pdc=None*, *aggregation_freq='D'*, *pv_input='power'*, *windspeed=0*, *power_expected=None*, *temperature_model=None*, *power_dc_rated=None*, *interp_freq=None*, *max_timedelta=None*)

Class for end-to-end degradation and soiling analysis using `sensor_analysis()` or `clearsky_analysis()`

Parameters

- **pv** (`pandas.Series`) -- Right-labeled time series PV energy or power. If energy, should *not* be cumulative, but only for preceding time step.
- **poa_global** (`pandas.Series`) -- Right-labeled time series measured plane of array irradiance in W/m^2
- **temperature_cell** (`pandas.Series`) -- Right-labeled time series of cell temperature in Celsius. In practice, back of module temperature works as a good approximation.
- **temperature_ambient** (`pandas.Series`) -- Right-labeled time Series of ambient temperature in Celsius
- **gamma_pdc** (`float`) -- Fractional PV power temperature coefficient
- **aggregation_freq** (`str` or `pandas.tseries.offsets.DateOffset`) -- Pandas frequency specification with which to aggregate normalized PV data for analysis. For

more information, see https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#dateoffset-objects

- **pv_input** (str) -- 'power' or 'energy' to specify type of input used for pv parameter
- **windspeed** (*numeric*) -- Right-labeled Pandas Time Series or single numeric value indicating wind speed in m/s for use in calculating cell temperature from ambient default value of 0 neglects the wind in this calculation
- **power_expected** (pandas.Series) -- Right-labeled time series of expected PV power. (Note: Expected energy is not supported.)
- **temperature_model** (str or dict) -- Model parameters for pvlib.temperature.sapm_cell(). Used in calculating cell temperature from ambient. If string, must be a valid entry for sapm model in pvlib.temperature.TEMPERATURE_MODEL_PARAMETERS. If dict, must have keys 'a', 'b', 'deltaT'. See pvlib.temperature.sapm_cell() documentation for details.
- **power_dc_rated** (float) -- Nameplate DC rating of PV array in Watts. If omitted, pv output will be internally normalized in the normalization step based on it's 95th percentile (see TrendAnalysis._pvwatts_norm() source).
- **interp_freq** (str or pandas.tseries.offsets.DateOffset) -- Pandas frequency specification used to interpolate the input PV power or energy. We recommend using the natural frequency of the data, rather than up or down sampling. Analysis requires regular time series. For more information see https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#dateoffset-objects
- **max_timedelta** (pandas.Timedelta) -- The maximum gap in the data to be interpolated/integrated across when interpolating or calculating energy from power

(not all attributes documented here)

filter_params
parameters to be passed to rdtools.filtering functions. Keys are the names of the rdtools.filtering functions. Values are dicts of parameters to be passed to those functions. Also has a special key *ad_hoc_filter* the associated value is a boolean mask joined with the rest of the filters. filter_params defaults to empty dicts for each function in rdtools.filtering, in which case those functions use default parameter values, *ad_hoc_filter* defaults to None. See examples for more information.

Type dict

results
Nested dict used to store the results of methods ending with *_analysis*

Type dict

__init__(*pv*, *poa_global=None*, *temperature_cell=None*, *temperature_ambient=None*, *gamma_pdc=None*, *aggregation_freq='D'*, *pv_input='power'*, *windspeed=0*, *power_expected=None*, *temperature_model=None*, *power_dc_rated=None*, *interp_freq=None*, *max_timedelta=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(pv[, poa_global, temperature_cell, ...])	Initialize self.
clearsky_analysis([analyses, yoy_kwargs, ...])	Perform entire clear-sky-based analysis workflow.
plot_degradation_summary(case, **kwargs)	Return a figure of a scatter plot and a histogram summarizing degradation rate analysis.
plot_pv_vs_irradiance(case[, alpha])	Plot PV energy vs irradiance, useful in diagnosing things like timezone problems or transposition errors.
plot_soiling_interval(case, **kwargs)	Return a figure visualizing the valid soiling intervals used in stochastic rate and recovery soiling analysis.
plot_soiling_monte_carlo(case, **kwargs)	Return a figure visualizing the Monte Carlo of soiling profiles used in stochastic rate and recovery soiling analysis.
plot_soiling_rate_histogram(case, **kwargs)	Return a histogram of soiling rates found in the stochastic rate and recovery soiling analysis
sensor_analysis([analyses, yoy_kwargs, ...])	Perform entire sensor-based analysis workflow.
set_clearsky([pvlib_location, pv_azimuth, ...])	Initialize values for a clearsky analysis which requires configuration of location and orientation details.

rdtools.analysis_chains.TrendAnalysis.set_clearsky

TrendAnalysis.**set_clearsky**(*pvlib_location=None*, *pv_azimuth=None*, *pv_tilt=None*, *poa_global_clearsky=None*, *temperature_cell_clearsky=None*, *temperature_ambient_clearsky=None*, *albedo=0.25*, *solar_position_method='nrel_numpy'*)

Initialize values for a clearsky analysis which requires configuration of location and orientation details. If optional parameters *poa_global_clearsky*, *temperature_ambient_clearsky* are not passed, they will be modeled based on location and orientation.

Parameters

- **pvlib_location** (*pvlib.location.Location*) -- Used for calculating clearsky temperature and irradiance
- **pv_azimuth** (*numeric*) -- Azimuth of PV array in degrees from north. Can be right-labeled Pandas Time Series or single numeric value.
- **pv_tilt** (*numeric*) -- Tilt of PV array in degrees from horizontal. Can be right-labeled Pandas Time Series or single numeric value.
- **poa_global_clearsky** (*pandas.Series*) -- Right-labeled time Series of clear-sky plane of array irradiance
- **temperature_cell_clearsky** (*pandas.Series*) -- Right-labeled time series of cell temperature in clear-sky conditions in Celsius. In practice, back of module temperature works as a good approximation.
- **temperature_ambient_clearsky** (*pandas.Series*) -- Right-label time series of ambient temperature in clear sky conditions in Celsius
- **albedo** (*numeric*) -- Albedo to be used in irradiance transposition calculations. Can be right-labeled Pandas Time Series or single numeric value.

- **solar_position_method** (str, *default* 'nrel_numpy') -- Optional method name to pass to pvlib.solarposition.get_solarposition(). Switching methods may improve calculation time.

rdtools.analysis_chains.TrendAnalysis.sensor_analysis

TrendAnalysis.**sensor_analysis**(*analyses=['yoy_degradation']*, *yoy_kwargs={}*, *srr_kwargs={}*)

Perform entire sensor-based analysis workflow. Results are stored in self.results['sensor']

Parameters

- **analyses** (list) -- Analyses to perform as a list of strings. Valid entries are 'yoy_degradation' and 'srr_soiling'
- **yoy_kwargs** (dict) -- kwargs to pass to degradation.degradation_year_on_year()
- **srr_kwargs** (dict) -- kwargs to pass to soiling.soiling_srr()

Returns

Return type None

rdtools.analysis_chains.TrendAnalysis.clearsky_analysis

TrendAnalysis.**clearsky_analysis**(*analyses=['yoy_degradation']*, *yoy_kwargs={}*, *srr_kwargs={}*)

Perform entire clear-sky-based analysis workflow. Results are stored in self.results['clearsky']

Parameters

- **analyses** (list) -- Analyses to perform as a list of strings. Valid entries are 'yoy_degradation' and 'srr_soiling'
- **yoy_kwargs** (dict) -- kwargs to pass to degradation.degradation_year_on_year()
- **srr_kwargs** (dict) -- kwargs to pass to soiling.soiling_srr()

Returns

Return type None

rdtools.analysis_chains.TrendAnalysis.plot_degradation_summary

TrendAnalysis.**plot_degradation_summary**(*case*, ***kwargs*)

Return a figure of a scatter plot and a histogram summarizing degradation rate analysis.

Parameters

- **case** (str) -- The workflow result to plot, allowed values are 'sensor' and 'clearsky'
- **kwargs** -- Extra parameters passed to plotting.degradation_summary_plots()

Returns

Return type matplotlib.figure.Figure

rdtools.analysis_chains.TrendAnalysis.plot_soiling_rate_histogram

TrendAnalysis.**plot_soiling_rate_histogram**(*case*, ***kwargs*)

Return a histogram of soiling rates found in the stochastic rate and recovery soiling analysis

Parameters

- **case** (str) -- The workflow result to plot, allowed values are 'sensor' and 'clearsky'
- **kwargs** -- Extra parameters passed to plotting.soiling_rate_histogram()

Returns

Return type matplotlib.figure.Figure

rdtools.analysis_chains.TrendAnalysis.plot_soiling_interval

TrendAnalysis.**plot_soiling_interval**(*case*, ***kwargs*)

Return a figure visualizing the valid soiling intervals used in stochastic rate and recovery soiling analysis.

Parameters

- **case** (str) -- The workflow result to plot, allowed values are 'sensor' and 'clearsky'
- **kwargs** -- Extra parameters passed to plotting.soiling_interval_plot()

Returns

Return type matplotlib.figure.Figure

rdtools.analysis_chains.TrendAnalysis.plot_soiling_monte_carlo

TrendAnalysis.**plot_soiling_monte_carlo**(*case*, ***kwargs*)

Return a figure visualizing the Monte Carlo of soiling profiles used in stochastic rate and recovery soiling analysis.

Parameters

- **case** (str) -- The workflow result to plot, allowed values are 'sensor' and 'clearsky'
- **kwargs** -- Extra parameters passed to plotting.soiling_monte_carlo_plot()

Returns

Return type matplotlib.figure.Figure

rdtools.analysis_chains.TrendAnalysis.plot_pv_vs_irradiance

TrendAnalysis.**plot_pv_vs_irradiance**(*case*, *alpha=0.01*, ***kwargs*)

Plot PV energy vs irradiance, useful in diagnosing things like timezone problems or transposition errors.

Parameters

- **case** (str) -- The plane of array irradiance type to plot, allowed values are 'sensor' and 'clearsky'
- **alpha** (float) -- transparency of the scatter plot
- **kwargs** -- Extra parameters passed to matplotlib.pyplot.axis.plot()

Returns

Return type matplotlib.figure.Figure

8.2.3 Degradation

Functions for calculating the degradation rate of photovoltaic systems.

degradation_classical_decomposition(...[, ...])	Estimate the trend of a timeseries using a classical decomposition approach (moving average) and calculate various statistics, including the result of a Mann-Kendall test and a Monte Carlo-derived confidence interval of slope.
degradation_ols(energy_normalized[, ...])	Estimate the trend of a timeseries using ordinary least-squares regression and calculate various statistics including a Monte Carlo-derived confidence interval of slope.
degradation_year_on_year(energy_normalized)	Estimate the trend of a timeseries using the year-on-year decomposition approach and calculate a Monte Carlo-derived confidence interval of slope.

rdtools.degradation.degradation_classical_decomposition

rdtools.degradation.**degradation_classical_decomposition**(*energy_normalized*, *confidence_level=68.2*)

Estimate the trend of a timeseries using a classical decomposition approach (moving average) and calculate various statistics, including the result of a Mann-Kendall test and a Monte Carlo-derived confidence interval of slope.

Parameters

- **energy_normalized** (pandas.Series) -- Daily or lower frequency time series of normalized system ouput. Must be regular time series.
- **confidence_level** (float, *default* 68.2) -- The size of the confidence interval to return, in percent.

Returns

- **Rd_pct** (float) -- Estimated degradation relative to the year 0 system capacity [%/year]
- **Rd_CI** (numpy.array) -- The calculated confidence interval bounds.
- **calc_info** (dict) -- A dict that contains slope, intercept, root mean square error of regression ('rmse'), standard error of the slope ('slope_stderr'), intercept ('intercept_stderr'), and least squares RegressionResults object ('ols_results'), pandas series for the annual rolling mean ('series'), and Mann-Kendall test trend ('mk_test_trend')

rdtools.degradation.degradation_ols

rdtools.degradation.**degradation_ols**(*energy_normalized*, *confidence_level=68.2*)

Estimate the trend of a timeseries using ordinary least-squares regression and calculate various statistics including a Monte Carlo-derived confidence interval of slope.

Parameters

- **energy_normalized** (pandas.Series) -- Daily or lower frequency time series of normalized system ouput.
- **confidence_level** (float, *default* 68.2) -- The size of the confidence interval to return, in percent.

Returns

- **Rd_pct** (float) -- Estimated degradation relative to the year 0 system capacity [%/year]
- **Rd_CI** (numpy.array) -- The calculated confidence interval bounds.
- **calc_info** (dict) -- A dict that contains slope, intercept, root mean square error of regression ('rmse'), standard error of the slope ('slope_stderr'), intercept ('intercept_stderr'), and least squares RegressionResults object ('ols_results')

rdtools.degradation.degradation_year_on_year

rdtools.degradation.**degradation_year_on_year**(*energy_normalized*, *recenter=True*, *exceedance_prob=95*, *confidence_level=68.2*)

Estimate the trend of a timeseries using the year-on-year decomposition approach and calculate a Monte Carlo-derived confidence interval of slope.

Parameters

- **energy_normalized** (pandas.Series) -- Daily or lower frequency time series of normalized system ouput.
- **recenter** (bool, *default* True) -- Specify whether data is internally recentered to normalized yield of 1 based on first year median. If False, Rd_pct is calculated assuming energy_normalized is passed already normalized to the year 0 system capacity.
- **exceedance_prob** (float, *default* 95) -- The probability level to use for exceedance value calculation, in percent.
- **confidence_level** (float, *default* 68.2) -- The size of the confidence interval to return, in percent.

Returns

- **Rd_pct** (float) -- Estimated degradation relative to the year 0 median system capacity [%/year]
- **confidence_interval** (numpy.array) -- confidence interval (size specified by confidence_level) of degradation rate estimate
- **calc_info** (dict) --
  - *YoY_values* - pandas series of right-labeled year on year slopes
  - *renormalizing_factor* - float of value used to recenter data
  - *exceedance_level* - the degradation rate that was outperformed with probability of *exceedance_prob*

– *usage_of_points* - number of times each point in energy_normalized is used to calculate a degradation slope. 0: point is never used. 1: point is either used as a start or endpoint. 2: point is used as both start and endpoint for an Rd calculation.

8.2.4 Soiling

Functions for calculating soiling metrics from photovoltaic system data.

The soiling module is currently experimental. The API, results, and default behaviors may change in future releases (including MINOR and PATCH releases) as the code matures.

soiling_srr(energy_normalized_daily, ...[, ...])	Functional wrapper for *SRRAnalysis*.
monthly_soiling_rates(soiling_interval_summary)	Use Monte Carlo to calculate typical monthly soiling rates.
annual_soiling_ratios(...[, confidence_level])	Return annualized soiling ratios and associated confidence intervals based on stochastic soiling profiles from SRR.
SRRAnalysis(energy_normalized_daily, ...[, ...])	Class for running the stochastic rate and recovery (SRR) photovoltaic soiling loss analysis presented in Deceglie et al.
SRRAnalysis.run([reps, day_scale, ...])	Run the SRR method from beginning to end.

rdtools.soiling.soiling_srr

rdtools.soiling.**soiling_srr**(*energy_normalized_daily*, *insolation_daily*, *reps=1000*, *precipitation_daily=None*, *day_scale=13*, *clean_threshold='infer'*, *trim=False*, *method='half_norm_clean'*, *clean_criterion='shift'*, *precip_threshold=0.01*, *min_interval_length=7*, *exceedance_prob=95.0*, *confidence_level=68.2*, *recenter=True*, *max_relative_slope_error=500.0*, *max_negative_step=0.05*, *outlier_factor=1.5*)

Functional wrapper for *SRRAnalysis*. Perform the stochastic rate and recovery soiling loss calculation. Based on the methods presented in Deceglie et al. JPV 8(2) p547 2018.

Parameters

- **energy_normalized_daily** (pandas.Series) -- Daily performance metric (i.e. performance index, yield, etc.) Alternatively, the soiling ratio output of a soiling sensor (e.g. the photocurrent ratio between matched dirty and clean PV reference cells). In either case, data should be insolation-weighted daily aggregates.
- **insolation_daily** (pandas.Series) -- Daily plane-of-array insolation corresponding to *energy_normalized_daily*. Arbitrary units.
- **reps** (int, *default* 1000) -- number of Monte Carlo realizations to calculate
- **precipitation_daily** (pandas.Series, *default* None) -- Daily total precipitation. Units ambiguous but should be the same as precip_threshold. Note default behavior of precip_threshold. (Ignored if clean_criterion='shift'.)
- **day_scale** (int, *default* 13) -- The number of days to use in rolling median for cleaning detection, and the maximum number of days of missing data to tolerate in a valid interval. An odd value is recommended.
- **clean_threshold** (float or 'infer', *default* 'infer') -- The fractional positive shift in rolling median for cleaning detection. Or specify 'infer' to automatically use outliers

in the shift as the threshold.

- **trim** (bool, *default* False) -- Whether to trim (remove) the first and last soiling intervals to avoid inclusion of partial intervals
- **method** (str, {'half_norm_clean', 'random_clean', 'perfect_clean'} default 'half_norm_clean') -- How to treat the recovery of each cleaning event
  - 'random_clean' - a random recovery between 0-100%
  - 'perfect_clean' - each cleaning event returns the performance metric to 1
  - 'half_norm_clean' - The starting point of each interval is taken randomly from a half normal distribution with its mode (mu) at 1 and its sigma equal to 1/3 * (1-b) where b is the intercept of the fit to the interval.
- **clean_criterion** (str, {'shift', 'precip_and_shift', 'precip_or_shift', 'precip'} default 'shift') -- The method of partitioning the dataset into soiling intervals
  - 'precip_and_shift' - rolling median shifts must coincide with precipitation to be a valid cleaning event.
  - 'precip_or_shift' - rolling median shifts and precipitation events are each sufficient on their own to be a cleaning event.
  - 'shift', only rolling median shifts are treated as cleaning events.
  - 'precip', only precipitation events are treated as cleaning events.
- **precip_threshold** (float, *default* 0.01) -- The daily precipitation threshold for defining precipitation cleaning events. Units must be consistent with precip.
- **min_interval_length** (int, *default* 7) -- The minimum duration, in days, for an interval to be considered valid. Cannot be less than 2 (days).
- **exceedance_prob** (float, *default* 95.0) -- the probability level to use for exceedance value calculation in percent
- **confidence_level** (float, *default* 68.2) -- the size of the confidence interval to return, in percent
- **recenter** (bool, *default* True) -- specify whether data is centered to normalized yield of 1 based on first year median
- **max_relative_slope_error** (float, *default* 500.0) -- the maximum relative size of the slope confidence interval for an interval to be considered valid (percentage).
- **max_negative_step** (float, *default* 0.05) -- The maximum magnitude of negative discrete steps allowed in an interval for the interval to be considered valid (units of normalized performance metric).
- **outlier_factor** (float, *default* 1.5) -- The factor used in the Tukey fence definition of outliers for flagging positive shifts in the rolling median used for cleaning detection. A smaller value will cause more and smaller shifts to be classified as cleaning events.

Returns

- **insolation_weighted_soiling_ratio** (float) -- P50 insolation weighted soiling ratio based on stochastic rate and recovery analysis
- **confidence_interval** (numpy.array) -- confidence interval (size specified by confidence_level) of degradation rate estimate

- **calc_info** (`dict`) --
  - 'renormalizing_factor' - value used to recenter data
  - 'exceedance_level' - the insolation-weighted soiling ratio that was outperformed with probability of exceedance_prob
  - 'stochastic_soiling_profiles' - List of Pandas series corresponding to the Monte Carlo realizations of soiling ratio profiles
  - 'soiling_ratio_perfect_clean' - Pandas series of the soiling ratio during valid soiling intervals assuming perfect cleaning and P50 slopes
  - 'soiling_interval_summary' - Pandas dataframe summarizing the soiling intervals identified. The columns of the dataframe are as follows:

Column Name	Description
'start'	Start timestamp of the soiling interval
'end'	End timestamp of the soiling interval
'soiling_rate'	P50 Soiling rate for interval, in day^1 Negative value indicates soiling is occurring. E.g. a rate of 0.01 indicates 1% soiling loss per day.
'soiling_rate_low'	Low edge of confidence interval for soiling rate for interval, in day^1
'soiling_rate_high'	High edge of confidence interval for soiling rate for interval, in day^1
'inferred_start_loss'	Estimated performance metric at the start of the interval
'inferred_end_loss'	Estimated performance metric at the end of the interval
'length'	Number of days in the interval
'valid'	Whether the interval meets the criteria to be treated as a valid soiling interval

rdtools.soiling.monthly_soiling_rates

rdtools.soiling.**monthly_soiling_rates**(*soiling_interval_summary*, *min_interval_length=14*, *max_relative_slope_error=500.0*, *reps=100000*, *confidence_level=68.2*)

Use Monte Carlo to calculate typical monthly soiling rates. Samples possible soiling rates from soiling rate confidence intervals associated with soiling intervals assuming a uniform distribution. Soiling intervals get samples proportionally to their overlap with each calendar month.

Parameters

- **soiling_interval_summary** (`pandas.DataFrame`) -- DataFrame describing soiling intervals. Typically from `soiling_info['soiling_interval_summary']` obtained with *rdtools.soiling.soiling_srr()* or *rdtools.soiling.SRRAnalysis.run()* Must have columns `soiling_rate_high`, `soiling_rate_low`, `soiling_rate`, `length`, `valid`,``start``, and `end`.
- **min_interval_length** (`int`, *default* `14`) -- The minimum number of days a soiling interval must contain to be included in the calculation. Similar to the same parameter in *soiling_srr()* and *SRRAnalysis.run()* but with a more conservative default value as a starting point for monthly soiling rate analyses.

- **max_relative_slope_error** (float, *default* 500.0) -- The maximum relative size of the slope confidence interval for an interval to be included in the calculation (percentage).
- **reps** (int, *default* 100000) -- The number of Monte Carlo samples to take for each month.
- **confidence_level** (float, *default* 68.2) -- The size of the confidence interval, as a percentage, to use in determining the upper and lower quantiles reported in the returned DataFrame. (The median is always included in the result.)

Returns

DataFrame describing monthly soiling rates.

Column Name	Description
'month'	Integer month, January (1) to December (12)
'soiling_rate_median'	The median soiling rate for the month over the entire dataset, in units of day^1. Negative value indicates soiling is occurring. E.g. a rate of 0.01 indicates 1% soiling loss per day.
'soiling_rate_low'	The lower edge of the confidence interval for the monthly soiling rate in units of day^1
'soiling_rate_high'	The upper edge of the confidence interval for the monthly soiling rate in units of day^1
'interval_count'	The number of soiling intervals contributing to the monthly calculation. If only a few intervals contribute, the confidence interval is likely to underestimate the true uncertainty.

Return type pandas.DataFrame

rdtools.soiling.annual_soiling_ratios

rdtools.soiling.**annual_soiling_ratios**(*stochastic_soiling_profiles*, *insolation_daily*, *confidence_level=68.2*)

Return annualized soiling ratios and associated confidence intervals based on stochastic soiling profiles from SRR. Note that each year may be affected by previous years' profiles for all SRR cleaning assumptions (i.e. method) except perfect_clean.

Parameters

- **stochastic_soiling_profiles** (list) -- List of pd.Series representing profile realizations from the SRR monte carlo. Typically `soiling_interval_summary['stochastic_soiling_profiles']` obtained with *rdtools.soiling.soiling_srr()* or *rdtools.soiling.SRRAnalysis.run()*
- **insolation_daily** (pandas.Series) -- Daily plane-of-array insolation with DatetimeIndex. Arbitrary units.
- **confidence_level** (float, *default* 68.2) -- The size of the confidence interval to use in determining the upper and lower quantiles reported in the returned DataFrame. (The median is always included in the result.)

Returns

DataFrame describing annual soiling rates.

Column Name	Description
'year'	Calendar year
'soiling_ratio_median'	The median insolation-weighted soiling ratio for the year
'soiling_ratio_low'	The lower edge of the confidence interval for insolation-weighted soiling ratio for the year
'soiling_ratio_high'	The upper edge of the confidence interval for insolation-weighted soiling ratio for the year

Return type pandas.DataFrame

rdtools.soiling.SRRAnalysis

class rdtools.soiling.**SRRAnalysis**(*energy_normalized_daily*, *insolation_daily*, *precipitation_daily=None*)

Class for running the stochastic rate and recovery (SRR) photovoltaic soiling loss analysis presented in Deceglie et al. JPV 8(2) p547 2018

Parameters

- **energy_normalized_daily** (pandas.Series) -- Daily performance metric (i.e. performance index, yield, etc.) Alternatively, the soiling ratio output of a soiling sensor (e.g. the photocurrent ratio between matched dirty and clean PV reference cells). In either case, data should be insolation-weighted daily aggregates.
- **insolation_daily** (pandas.Series) -- Daily plane-of-array insolation corresponding to *energy_normalized_daily*. Arbitrary units.
- **precipitation_daily** (pandas.Series, *default* None) -- Daily total precipitation. (Ignored if clean_criterion='shift' in subsequent calculations.)

__init__(*energy_normalized_daily*, *insolation_daily*, *precipitation_daily=None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(energy_normalized_daily, ...[, ...])	Initialize self.
run([reps, day_scale, clean_threshold, ...])	Run the SRR method from beginning to end.

rdtools.soiling.SRRAnalysis.run

SRRAnalysis.**run**(*reps=1000*, *day_scale=13*, *clean_threshold='infer'*, *trim=False*, *method='half_norm_clean'*, *clean_criterion='shift'*, *precip_threshold=0.01*, *min_interval_length=7*, *exceedance_prob=95.0*, *confidence_level=68.2*, *recenter=True*, *max_relative_slope_error=500.0*, *max_negative_step=0.05*, *outlier_factor=1.5*)

Run the SRR method from beginning to end. Perform the stochastic rate and recovery soiling loss calculation. Based on the methods presented in Deceglie et al. "Quantifying Soiling Loss Directly From PV Yield" JPV 8(2) p547 2018.

Parameters

- **reps** (int, *default* 1000) -- number of Monte Carlo realizations to calculate

- **day_scale** (int, *default* 13) -- The number of days to use in rolling median for cleaning detection, and the maximum number of days of missing data to tolerate in a valid interval. An odd value is recommended.
- **clean_threshold** (float or 'infer', *default* 'infer') -- The fractional positive shift in rolling median for cleaning detection. Or specify 'infer' to automatically use outliers in the shift as the threshold.
- **trim** (bool, *default* False) -- Whether to trim (remove) the first and last soiling intervals to avoid inclusion of partial intervals
- **method** (str, {'half_norm_clean', 'random_clean', 'perfect_clean'} default 'half_norm_clean') -- How to treat the recovery of each cleaning event
  - 'random_clean' - a random recovery between 0-100%
  - 'perfect_clean' - each cleaning event returns the performance metric to 1
  - 'half_norm_clean' - The starting point of each interval is taken randomly from a half normal distribution with its mode (mu) at 1 and its sigma equal to 1/3 * (1-b) where b is the intercept of the fit to the interval.
- **clean_criterion** (str, {'shift', 'precip_and_shift', 'precip_or_shift', 'precip'} default 'shift') -- The method of partitioning the dataset into soiling intervals
  - 'precip_and_shift' - rolling median shifts must coincide with precipitation to be a valid cleaning event.
  - 'precip_or_shift' - rolling median shifts and precipitation events are each sufficient on their own to be a cleaning event.
  - 'shift', only rolling median shifts are treated as cleaning events.
  - 'precip', only precipitation events are treated as cleaning events.
- **precip_threshold** (float, *default* 0.01) -- The daily precipitation threshold for defining precipitation cleaning events. Units must be consistent with self.precipitation_daily
- **min_interval_length** (int, *default* 7) -- The minimum duration for an interval to be considered valid. Cannot be less than 2 (days).
- **exceedance_prob** (float, *default* 95.0) -- The probability level to use for exceedance value calculation in percent
- **confidence_level** (float, *default* 68.2) -- The size of the confidence interval to return, in percent
- **recenter** (bool, *default* True) -- Specify whether data is centered to normalized yield of 1 based on first year median
- **max_relative_slope_error** (float, *default* 500) -- the maximum relative size of the slope confidence interval for an interval to be considered valid (percentage).
- **max_negative_step** (float, *default* 0.05) -- The maximum magnitude of negative discrete steps allowed in an interval for the interval to be considered valid (units of normalized performance metric).
- **outlier_factor** (float, *default* 1.5) -- The factor used in the Tukey fence definition of outliers for flagging positive shifts in the rolling median used for cleaning detection. A smaller value will cause more and smaller shifts to be classified as cleaning events.

Returns

- **insolation_weighted_soiling_ratio** (`float`) -- P50 insolation-weighted soiling ratio based on stochastic rate and recovery analysis
- **confidence_interval** (`numpy.array`) -- confidence interval (size specified by confidence_level) of insolation-weighted soiling ratio
- **calc_info** (`dict`) --
  - 'renormalizing_factor' - value used to recenter data
  - 'exceedance_level' - the insolation-weighted soiling ratio that was outperformed with probability of exceedance_prob
  - 'stochastic_soiling_profiles' - List of Pandas series corresponding to the Monte Carlo realizations of soiling ratio profiles
  - 'soiling_ratio_perfect_clean' - Pandas series of the soiling ratio during valid soiling intervals assuming perfect cleaning and P50 slopes
  - 'soiling_interval_summary' - Pandas dataframe summarizing the soiling intervals identified. The columns of the dataframe are as follows:

Column Name	Description
'start'	Start timestamp of the soiling interval
'end'	End timestamp of the soiling interval
'soiling_rate'	P50 Soiling rate for interval, in day^1 Negative value indicates soiling is occurring. E.g. a rate of 0.01 indicates 1% soiling loss per day.
'soiling_rate_low'	Low edge of confidence interval for soiling rate for interval, in day^1
'soiling_rate_high'	High edge of confidence interval for soiling rate for interval, in day^1
'inferred_start_loss'	Estimated performance metric at the start of the interval
'inferred_end_loss'	Estimated performance metric at the end of the interval
'length'	Number of days in the interval
'valid'	Whether the interval meets the criteria to be treated as a valid soiling interval

8.2.5 System Availability

Functions for detecting and quantifying production loss from photovoltaic system downtime events.

The availability module is currently experimental. The API, results, and default behaviors may change in future releases (including MINOR and PATCH releases) as the code matures.

AvailabilityAnalysis(power_system, ...)	A class to perform system availability and loss analysis.
AvailabilityAnalysis.run([low_threshold, ...])	Run the availability analysis.
AvailabilityAnalysis.plot()	Create a figure summarizing the availability analysis results.

rdtools.availability.AvailabilityAnalysis

class rdtools.availability.**AvailabilityAnalysis**(*power_system*, *power_subsystem*, *energy_cumulative*, *power_expected*)

A class to perform system availability and loss analysis.

This class follows the analysis procedure described in[1], and implements two distinct algorithms. One for partial (subsystem) outages and one for system-wide outages. The *AvailabilityAnalysis.run()* method executes both algorithms and combines their results.

The input timeseries don't need to be in any particular set of units as long as all power and energy units are consistent, with energy units being the hourly-integrated power (e.g., kW and kWh). The units of the analysis outputs will match the inputs.

Parameters

- **power_system** (pandas.Series) -- Timeseries total system power. In the typical case, this is meter power data. Should be a right-labeled interval average (this is what is typically recorded in many DAS).
- **power_subsystem** (pandas.DataFrame) -- Timeseries power data, one column per subsystem. In the typical case, this is inverter AC power data. Each column is assumed to represent a subsystem, so no extra columns may be included. The index must match `power_system`. Should be a right-labeled interval average.
- **energy_cumulative** (pandas.Series) -- Timeseries cumulative energy data for the entire system (e.g. meter). These values must be recorded at the device itself (rather than summed by a downstream device like a datalogger or DAS provider) to preserve its integrity across communication interruptions. Units must match `power` integrated to hourly energy (e.g. if `power` is in kW then `energy` must be in kWh).
- **power_expected** (pandas.Series) -- Expected system power data with the same index as the measured data. This can be modeled from on-site weather measurements if instruments are well calibrated and there is no risk of data gaps. However, because full system outages often cause weather data to be lost as well, it may be more useful to use data from an independent weather station or satellite-based weather provider. Should be a right-labeled interval average.

results

Rolled-up production, loss, and availability metrics. The index is a datetime index of the period passed to *AvailabilityAnalysis.run()*. The columns of the dataframe are as follows:

Column Name	Description
'lost_production'	Production loss from outages. Units match the input power units (e.g. if power is given in kW, 'lost_production' will be in kWh).
'actual_production'	System energy production. Same units as 'lost_production'.
'availability'	Energy-weighted system availability as a fraction (0-1).

Type pandas.DataFrame

loss_system

Estimated timeseries lost power from system outages.

Type pandas.Series

[1] Anderson K. and Blumenthal R. "Overcoming communications outages in inverter downtime analysis", 2020 IEEE 47th Photovoltaic Specialists Conference (PVSC).

loss_subsystem
Estimated timeseries lost power from subsystem outages.

Type pandas.Series

loss_total
Estimated total lost power from outages.

Type pandas.Series

reporting_mask
Boolean mask indicating whether subsystems appear online or not.

Type pandas.DataFrame

power_expected_rescaled
Expected power rescaled to better match system power during periods where the system is performing normally.

Type pandas.Series

energy_expected_rescaled
Interval expected energy calculated from *power_expected_rescaled*.

Type pandas.Series

energy_cumulative_corrected
Cumulative system production after filling in data gaps from outages with estimated production.

Type pandas.Series

error_info
Records about the error between expected power and actual power.

Type pandas.DataFrame

interp_lower, interp_upper
Functions to estimate the uncertainty interval bounds of expected production based on outage length.

Type callable

outage_info
Records about each detected system outage, one row per outage. The primary columns of interest are `type`, which can be either `'real'` or `'comms'` and reports whether the outage was determined to be a real outage with lost production or just a communications interruption with no production impact; and `loss` which reports the estimated production loss for the outage. The columns are as follows:

Column Name	Description
'start'	Timestamp of the outage start.
'end'	Timestamp of the outage end.
'duration'	Length of the outage (*i.e.* `outage_info['end'] - outage_info['start']`).
'intervals'	Total count of data intervals contained in the outage.
'daylight_intervals'	Count of data intervals contained in the outage occurring during the day.
'error_lower'	Lower error bound as a fraction of expected energy.
'error_upper'	Upper error bound as a fraction of expected energy.
'energy_expected'	Total expected production for the outage duration.
'energy_start'	System cumulative production at the outage start.
'energy_end'	System cumulative production at the outage end.
'energy_actual'	System production during the outage (*i.e.*, `outage_info['energy_end'] - outage_info['energy_start']`).
'ci_lower'	Lower bound for the expected energy confidence interval.
'ci_upper'	Lower bound for the expected energy confidence interval.
'type'	Type of the outage ('real or 'comms').
'loss'	Estimated production loss.

Type `pandas.DataFrame`

Notes

This class's ability to detect short-duration outages is limited by the resolution of the system data. For instance, 15-minute averages would not be able to resolve the rapid power cycling of an intermittent inverter. Additionally, the loss at the edges of an outage may be underestimated because of masking by the interval averages.

This class expects outages to be represented in the timeseries by NaN, zero, or very low values. If your DAS does not record data from outages (e.g., a three-hour outage results in three hours of omitted timestamps), you should insert those missing rows before using this analysis.

References

__init__ (*power_system*, *power_subsystem*, *energy_cumulative*, *power_expected*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(power_system, power_subsystem, ...)	Initialize self.
plot()	Create a figure summarizing the availability analysis results.
run([low_threshold, relative_sizes, ...])	Run the availability analysis.

rdtools.availability.AvailabilityAnalysis.run

AvailabilityAnalysis.**run**(*low_threshold=None*, *relative_sizes=None*, *power_system_limit=None*, *quantiles=0.01, 0.99*, *rollup_period='M'*)

Run the availability analysis.

Parameters

- **low_threshold** (float or pandas.Series, *optional*) -- An optional threshold used to naively classify subsystems as online. If the threshold is a scalar, it will be used for all subsystems. For subsystems with different capacities, a pandas Series may be passed with index values matching the columns in power_subsystem. Units must match power_subsystem and power_system. If omitted, the limit is calculated for each subsystem independently as 0.001 times the 99th percentile of its power data.
- **relative_sizes** (dict or pandas.Series, *optional*) -- The production capacity of each subsystem, normalized by the mean subsystem capacity. If not specified, it will be estimated from power data.
- **power_system_limit** (float or pandas.Series, *optional*) -- Maximum allowable system power in the same units as the input power timeseries. This parameter is used to account for cases where online subsystems can partially mitigate the loss of an offline subsystem, for example a system with a plant controller and dynamic inverter setpoints. This constraint is only applied to the subsystem loss calculation.
- **quantiles** (tuple, *default* (0.01, 0.99)) -- (lower, upper) quantiles of the error distribution used for the expected energy confidence interval. The lower bound is used to classify outages as either (1) a simple communication interruption with no production loss or (2) a power outage with an associated production loss estimate.
- **rollup_period** (pandas.tseries.offsets.DateOffset or alias, *default* 'M') -- The period on which to roll up losses and calculate availability.

rdtools.availability.AvailabilityAnalysis.plot

AvailabilityAnalysis.**plot**()

Create a figure summarizing the availability analysis results. The analysis must be run using the *run()* method before using this method.

Returns **fig**

Return type matplotlib.figure.Figure

8.2.6 Filtering

Functions for filtering and subsetting PV system data.

clip_filter(power_ac[, model])	Master wrapper for running one of the desired clipping filters.
quantile_clip_filter(power_ac[, quantile])	Filter data points likely to be affected by clipping with power or energy greater than or equal to 99% of the *quant* quantile.
logic_clip_filter(power_ac[, mounting_type, ...])	This filter is a logic-based filter that is used to filter out clipping periods in AC power or energy time series.

continues on next page

Table 21 – continued from previous page

xgboost_clip_filter(power_ac[, mounting_type])	This function generates the features to run through the XGBoost clipping model, runs the data through the model, and generates model outputs.
csi_filter(poa_global_measured, ...[, threshold])	Filtering based on clear-sky index (csi)
poa_filter(poa_global[, poa_global_low, ...])	Filter POA irradiance readings outside acceptable measurement bounds.
tcell_filter(temperature_cell[, ...])	Filter temperature readings outside acceptable measurement bounds.
normalized_filter(energy_normalized[, ...])	Select normalized yield between `low_cutoff` and `high_cutoff`

rdtools.filtering.clip_filter

`rdtools.filtering.`**`clip_filter`**(*power_ac*, *model='quantile'*, ***kwargs*)

Master wrapper for running one of the desired clipping filters. The default filter run is the quantile clipping filter.

Parameters

- **power_ac** (`pandas.Series`) -- Pandas time series, representing PV system power or energy. For best performance, timestamps should be in local time.
- **model** (`str`, *default* `'quantile'`) -- Clipping filter model to run. Can be 'quantile', 'xgboost', or 'logic'. Note: using the xgboost model can result in errors on some systems. These can often be alleviated by using conda to install xgboost, see https://anaconda.org/conda-forge/xgboost.
- **kwargs** -- Additional clipping filter args, specific to the model being used. Keyword must be passed with value.

Returns Boolean Series of whether to include the point because it is not clipping. True values delineate non-clipping periods, and False values delineate clipping periods.

Return type pandas.Series

rdtools.filtering.quantile_clip_filter

`rdtools.filtering.`**`quantile_clip_filter`**(*power_ac*, *quantile=0.98*)

Filter data points likely to be affected by clipping with power or energy greater than or equal to 99% of the *quant* quantile.

Parameters

- **power_ac** (`pandas.Series`) -- AC power or AC energy time series
- **quantile** (`float`, *default* `0.98`) -- Value for upper threshold quantile

Returns Boolean Series of whether the given measurement is below 99% of the quantile filter.

Return type pandas.Series

rdtools.filtering.logic_clip_filter

rdtools.filtering.**logic_clip_filter**(*power_ac*, *mounting_type='fixed'*, *rolling_range_max_cutoff=0.2*, *roll_periods=None*)

This filter is a logic-based filter that is used to filter out clipping periods in AC power or energy time series. It is based on the method presented in [1]. A boolean filter is returned based on the maximum range over a rolling window, as compared to a user-set rolling_range_max_cutoff (default set to 0.2). Periods where the relative maximum difference between any two points is less than rolling_range_max_cutoff are flagged as clipping and used to set daily clipping levels for the final mask.

Parameters

- **power_ac** (pandas.Series) -- Pandas time series, representing PV system power or energy. For best performance, timestamps should be in local time.
- **mounting_type** (str, *default* 'fixed') -- String representing the mounting configuration associated with the AC power or energy time series. Can either be "fixed" or "single_axis_tracking". Default set to 'fixed'.
- **rolling_range_max_cutoff** (float, *default* 0.2) -- Relative fractional cutoff for max rolling range threshold. When the relative maximum range in any interval is below this cutoff, the interval is determined to be clipping. Defaults to 0.2; however, values as high as 0.4 have been tested and shown to be effective. The higher the cutoff, the more values in the dataset that will be determined as clipping.
- **roll_periods** (int, *optional*) -- Number of periods to examine when looking for a near-zero derivative in the time series derivative. If roll_periods = 3, the system looks for a near-zero derivative over 3 consecutive readings. Default value is set to None, so the function uses default logic: it looks for a near-zero derivative over 3 periods for a fixed tilt system, and over 5 periods for a tracked system with a sampling frequency more frequent than once every 30 minutes.

Returns Boolean Series of whether to include the point because it is not clipping. True values delineate non-clipping periods, and False values delineate clipping periods.

Return type pandas.Series

References

rdtools.filtering.xgboost_clip_filter

rdtools.filtering.**xgboost_clip_filter**(*power_ac*, *mounting_type='fixed'*)

This function generates the features to run through the XGBoost clipping model, runs the data through the model, and generates model outputs.

Parameters

- **power_ac** (pandas.Series) -- Pandas time series, representing PV system power or energy. For best performance, timestamps should be in local time.
- **mounting_type** (str, *default* 'fixed') -- String representing the mounting configuration associated with the AC power or energy time series. Can either be "fixed" or "single_axis_tracking".

Returns Boolean Series of whether to include the point because it is not clipping. True values delineate non-clipping periods, and False values delineate clipping periods.

Return type pandas.Series

References

rdtools.filtering.csi_filter

rdtools.filtering.**csi_filter**(*poa_global_measured*, *poa_global_clearsky*, *threshold=0.15*)

Filtering based on clear-sky index (csi)

Parameters

- **poa_global_measured** (pandas.Series) -- Plane of array irradiance based on measurments
- **poa_global_clearsky** (pandas.Series) -- Plane of array irradiance based on a clear sky model
- **threshold** (float, *default* 0.15) -- threshold for filter

Returns Boolean Series of whether the clear-sky index is within the threshold around 1.

Return type pandas.Series

rdtools.filtering.poa_filter

rdtools.filtering.**poa_filter**(*poa_global*, *poa_global_low=200*, *poa_global_high=1200*)

Filter POA irradiance readings outside acceptable measurement bounds.

Parameters

- **poa_global** (pandas.Series) -- POA irradiance measurements.
- **poa_global_low** (float, *default* 200) -- The lower bound of acceptable values.
- **poa_global_high** (float, *default* 1200) -- The upper bound of acceptable values.

Returns Boolean Series of whether the given measurement is within acceptable bounds.

Return type pandas.Series

rdtools.filtering.tcell_filter

rdtools.filtering.**tcell_filter**(*temperature_cell*, *temperature_cell_low=-50*, *temperature_cell_high=110*)

Filter temperature readings outside acceptable measurement bounds.

Parameters

- **temperature_cell** (pandas.Series) -- Cell temperature measurements.
- **temperature_cell_low** (float, *default* −50) -- The lower bound of acceptable values.
- **temperature_cell_high** (float, *default* 110) -- The upper bound of acceptable values.

Returns Boolean Series of whether the given measurement is within acceptable bounds.

Return type pandas.Series

rdtools.filtering.normalized_filter

rdtools.filtering.**normalized_filter**(*energy_normalized*, *energy_normalized_low=0.01*, *energy_normalized_high=None*)

Select normalized yield between `low_cutoff` and `high_cutoff`

Parameters

- **energy_normalized** (`pandas.Series`) -- Normalized energy measurements.
- **energy_normalized_low** (`float`, *default* `0.01`) -- The lower bound of acceptable values.
- **energy_normalized_high** (`float`, *optional*) -- The upper bound of acceptable values.

Returns Boolean Series of whether the given measurement is within acceptable bounds.

Return type pandas.Series

8.2.7 Normalization

Functions for normalizing, rescaling, and regularizing PV system data.

energy_from_power(power[, target_frequency, ...])	Returns a regular right-labeled energy time series in units of Wh per interval from a power time series.
interpolate(time_series, target[, ...])	Returns an interpolation of time_series, excluding times associated with gaps in each column of time_series longer than max_timedelta; NaNs are returned within those gaps.
irradiance_rescale(irrad, irrad_sim[, ...])	Attempt to rescale modeled irradiance to match measured irradiance on clear days.
normalize_with_expected_power(pv, ...[, ...])	Normalize PV power or energy based on expected PV power.
normalize_with_pvwatts(energy, pvwatts_kws)	Normalize system AC energy output given measured poa_global and meteorological data.
normalize_with_sapm(energy, sapm_kws)	Deprecated since version 2.0.0.
pvwatts_dc_power(poa_global, power_dc_rated)	PVWatts v5 Module Model: DC power given effective poa poa_global, module nameplate power, and cell temperature.
sapm_dc_power(pvlib_pvsystem, met_data)	Deprecated since version 2.0.0.
delta_index(series)	Deprecated since version 2.0.0.
check_series_frequency(series, ...)	Deprecated since version 2.0.0.

rdtools.normalization.energy_from_power

rdtools.normalization.**energy_from_power**(*power*, *target_frequency=None*, *max_timedelta=None*, *power_type='right_labeled'*)

Returns a regular right-labeled energy time series in units of Wh per interval from a power time series. For instantaneous timeseries, a trapezoidal sum is used. For right labeled time series, a rectangular sum is used. NaN is filled where the gap between input data points exceeds `max_timedelta`. Power_series should be given in Watts.

Parameters

- **power** (`pandas.Series`) -- Time series of power in Watts
- **target_frequency** (`pandas.tseries.offsets.DateOffset` or `frequency string`, *default* `None`) -- The frequency of the energy time series to be returned. If omitted, use the median timestep of `power`, or if `power` has fewer than two elements, use `power.index.freq`.
- **max_timedelta** (`pandas.Timedelta`, *default* `None`) -- The maximum allowed gap between power measurements. If the gap between consecutive power measurements exceeds `max_timedelta`, NaN will be returned for that interval. If omitted, `max_timedelta` is set internally to the median time delta in `power`. Ignored when `power` has fewer than two elements.
- **power_type** (`str`, `{'right_labeled', 'instantaneous'}`) -- The labeling convention used in power. Default: 'right_labeled'

Returns right-labeled energy in Wh per interval

Return type pandas.Series

rdtools.normalization.interpolate

rdtools.normalization.**interpolate**(*time_series*, *target*, *max_timedelta=None*, *warning_threshold=0.1*)

Returns an interpolation of time_series, excluding times associated with gaps in each column of time_series longer than max_timedelta; NaNs are returned within those gaps.

Parameters

- **time_series** (`pandas.Series`, `pandas.DataFrame`) -- Original values to be used in generating the interpolation
- **target** (`pandas.DatetimeIndex`, `pandas.tseries.offsets.DateOffset`, `frequency string`) --
  - If DatetimeIndex: the index onto which the interpolation is to be made
  - If DateOffset or frequency string: the frequency at which to resample and interpolate
- **max_timedelta** (`pandas.Timedelta`, *default* `None`) -- The maximum allowed gap between values in `time_series`. Times associated with gaps longer than `max_timedelta` are excluded from the output. If omitted, `max_timedelta` is set internally to two times the median time delta in `time_series`.
- **warning_threshold** (`float`, *default* `0.1`) -- The fraction of data exclusion above which a warning is raised. With the default value of 0.1, a warning will be raised if the fraction of data excluded because of data gaps longer than `max_timedelta` is above than 10%.

Returns

Return type pandas.Series or pandas.DataFrame (matching type of time_series) with DatetimeIndex

---

Note: Timezone information in the DatetimeIndexes is handled automatically, however both `time_series` and `target` should be time zone aware or they should both be time zone naive.

---

rdtools.normalization.irradiance_rescale

`rdtools.normalization.`**`irradiance_rescale`**(*irrad*, *irrad_sim*, *max_iterations=100*, *method='iterative'*, *convergence_threshold=1e-06*)

Attempt to rescale modeled irradiance to match measured irradiance on clear days.

Parameters

- **irrad** (`pandas.Series`) -- measured irradiance time series
- **irrad_sim** (`pandas.Series`) -- modeled/simulated irradiance time series
- **max_iterations** (`int`, *default* `100`) -- The maximum number of times to attempt rescale optimization. Ignored if `method = 'single_opt'`
- **method** (`str`, *default* `'iterative'`) -- The calculation method to use. 'single_opt' implements the irradiance_rescale of rdtools v1.1.3 and earlier. 'iterative' implements a more stable calculation that may yield different results from the single_opt method.
- **convergence_threshold** (`float`, *default* `1e-6`) -- The acceptable iteration-to-iteration scaling factor difference to determine convergence. If the threshold is not reached after `max_iterations`, raise `rdtools.normalization.ConvergenceError`. Must be greater than zero. Only used if `method=='iterative'`.

Returns Rescaled modeled irradiance time series

Return type pandas.Series

rdtools.normalization.normalize_with_expected_power

`rdtools.normalization.`**`normalize_with_expected_power`**(*pv*, *power_expected*, *poa_global*, *pv_input='power'*)

Normalize PV power or energy based on expected PV power.

Parameters

- **pv** (`pandas.Series`) -- Right-labeled time series PV energy or power. If energy, should *not* be cumulative, but only for preceding time step. Type (energy or power) must be specified in the `pv_input` parameter.
- **power_expected** (`pandas.Series`) -- Right-labeled time series of expected PV power. (Note: Expected energy is not supported.)
- **poa_global** (`pandas.Series`) -- Right-labeled time series of plane-of-array irradiance associated with `expected_power`
- **pv_input** (`str`, `{'power' or 'energy'}`) -- Specifies the type of input used for `pv` parameter. Default: 'power'

Returns

- **energy_normalized** (pandas.Series) -- Energy normalized based on power_expected
- **insolation** (pandas.Series) -- Insolation associated with each normalized point

rdtools.normalization.normalize_with_pvwatts

rdtools.normalization.**normalize_with_pvwatts**(*energy*, *pvwatts_kws*)

Normalize system AC energy output given measured poa_global and meteorological data. This method uses the PVWatts V5 module model.

Energy timeseries and poa_global timeseries can be different granularities.

Parameters

- **energy** (pandas.Series) -- Energy time series to be normalized [Wh]. Must be a right-labeled regular time series.
- **pvwatts_kws** (dict) -- Dictionary of parameters used in the pvwatts_dc_power function. See Other Parameters.

Other Parameters

- **poa_global** (pandas.Series) -- Total effective plane of array irradiance [W/m**2].
- **power_dc_rated** (float) -- Rated DC power of array [W]
- **temperature_cell** (pandas.Series, *optional*) -- Measured or derived cell temperature [degrees Celsius]. Time series assumed to be same frequency as *poa_global*. If omitted, the temperature term will be ignored.
- **poa_global_ref** (float, *default* 1000) -- Reference irradiance at standard test condition [W/m**2].
- **temperature_cell_ref** (float, *default* 25) -- Reference temperature at standard test condition [degrees Celsius].
- **gamma_pdc** (float, *default* None) -- Linear array efficiency temperature coefficient [1 / degree Celsius]. If omitted, the temperature term will be ignored.

Note: All series are assumed to be right-labeled, meaning that the recorded value at a given timestamp refers to the previous time interval

---

Returns

- **energy_normalized** (pandas.Series) -- Energy divided by PVWatts DC energy [unitless].
- **insolation** (pandas.Series) -- Insolation associated with each normalized point [Wh/m**2]

rdtools.normalization.normalize_with_sapm

rdtools.normalization.**normalize_with_sapm**(*energy*, *sapm_kws*)

Deprecated since version 2.0.0: The normalize_with_sapm function was deprecated in rdtools 2.0.0 and will be removed in 3.0.0. Use normalize_with_expected_power instead.

Normalize system AC energy output given measured met_data and meteorological data. This method relies on the Sandia Array Performance Model (SAPM) to compute the effective DC energy using measured irradiance, ambient temperature, and wind speed.

Energy timeseries and met_data timeseries can be different granularities.

Warning: The `pvlib_pvsystem` argument must be a `pvlib.pvsystem.LocalizedPVSystem` object, which is no longer available as of pvlib 0.9.0. To use this function you'll need to use an older version of pvlib.

Parameters

- **energy** (`pandas.Series`) -- Energy time series to be normalized in watt hours. Must be a right-labeled regular time series.
- **sapm_kws** (`dict`) -- Dictionary of parameters required for sapm_dc_power function. See Other Parameters.

Other Parameters

- **pvlib_pvsystem** (`pvlib.pvsystem.LocalizedPVSystem object`) -- Object contains orientation, geographic coordinates, equipment constants (including DC rated power in watts). The object must also specify either the `temperature_model_parameters` attribute or both `racking_model` and `module_type` to infer the model parameters.
- **met_data** (`pandas.DataFrame`) -- Measured met_data, ambient temperature, and wind speed. Expected column names are ['DNI', 'GHI', 'DHI', 'Temperature', 'Wind Speed']

Note: All series are assumed to be right-labeled, meaning that the recorded value at a given timestamp refers to the previous time interval

Returns

- **energy_normalized** (`pandas.Series`) -- Energy divided by Sandia Model DC energy.
- **insolation** (`pandas.Series`) -- Insolation associated with each normalized point

rdtools.normalization.pvwatts_dc_power

rdtools.normalization.**pvwatts_dc_power**(*poa_global*, *power_dc_rated*, *temperature_cell=None*, *poa_global_ref=1000*, *temperature_cell_ref=25*, *gamma_pdc=None*)

PVWatts v5 Module Model: DC power given effective poa poa_global, module nameplate power, and cell temperature. This function differs from the PVLIB implementation by allowing cell temperature to be an optional parameter.

Parameters

- **poa_global** (pandas.Series) -- Total effective plane of array irradiance [W/m**2].
- **power_dc_rated** (float) -- Rated DC power of array [W]
- **temperature_cell** (pandas.Series, *optional*) -- Measured or derived cell temperature [degrees Celsius]. Time series assumed to be same frequency as poa_global. If omitted, the temperature term will be ignored.
- **poa_global_ref** (float, *default* 1000) -- Reference irradiance at standard test condition [W/m**2].
- **temperature_cell_ref** (float, *default* 25) -- Reference temperature at standard test condition [degrees Celsius].
- **gamma_pdc** (float, *default* None) -- Linear array efficiency temperature coefficient [1 / degree Celsius]. If omitted, the temperature term will be ignored.

Note: All series are assumed to be right-labeled, meaning that the recorded value at a given timestamp refers to the previous time interval

Returns **power_dc** -- DC power determined by PVWatts v5 equation [W].

Return type pandas.Series

rdtools.normalization.sapm_dc_power

rdtools.normalization.**sapm_dc_power**(*pvlib_pvsystem*, *met_data*)

Deprecated since version 2.0.0: The sapm_dc_power function was deprecated in rdtools 2.0.0 and will be removed in 3.0.0. Use normalize_with_expected_power instead.

Use Sandia Array Performance Model (SAPM) and PVWatts to compute the effective DC power using measured irradiance, ambient temperature, and wind speed. Effective irradiance and cell temperature are calculated with SAPM, and DC power with PVWatts.

Warning: The pvlib_pvsystem argument must be a pvlib.pvsystem.LocalizedPVSystem object, which is no longer available as of pvlib 0.9.0. To use this function you'll need to use an older version of pvlib.

Parameters

- **pvlib_pvsystem** (pvlib.pvsystem.LocalizedPVSystem) -- Object contains orientation, geographic coordinates, equipment constants (including DC rated power in watts). The object must also specify either the temperature_model_parameters attribute or both racking_model and module_type attributes to infer the temperature model parameters.
- **met_data** (pandas.DataFrame) -- Measured irradiance components, ambient temperature, and wind speed. Expected met_data DataFrame column names: ['DNI', 'GHI', 'DHI', 'Temperature', 'Wind Speed']

Note: All series are assumed to be right-labeled, meaning that the recorded value at a given timestamp refers to the previous time interval

Returns

- **power_dc** (pandas.Series) -- DC power in watts derived using Sandia Array Performance Model and PVWatts.
- **effective_poa** (pandas.Series) -- Effective irradiance calculated with SAPM

rdtools.normalization.delta_index

rdtools.normalization.**delta_index**(*series*)

Deprecated since version 2.0.0: The _delta_index function was deprecated in rdtools 2.0.0 and will be removed in 3.0.0.

Takes a pandas series with a DatetimeIndex as input and returns (time step sizes, average time step size) in hours

Parameters **series** (pandas.Series) -- A pandas timeseries

Returns

- **deltas** (pandas.Series) -- A timeseries representing the timestep sizes of series
- **mean** (float) -- The average timestep

rdtools.normalization.check_series_frequency

rdtools.normalization.**check_series_frequency**(*series*, *series_description*)

Deprecated since version 2.0.0: The _check_series_frequency function was deprecated in rdtools 2.0.0 and will be removed in 3.0.0.

Returns the inferred frequency of a pandas series, raises ValueError using series_description if it can't.

Parameters

- **series** (pandas.Series) -- The timeseries to infer the frequency of.
- **series_description** (str) -- The description to use when raising an error.

Returns **freq** -- The inferred index frequency

Return type pandas Offsets string

8.2.8 Aggregation

Functions for calculating weighted aggregates of PV system data.

aggregation_insol(energy_normalized, insolation)	Insolation weighted aggregation

rdtools.aggregation.aggregation_insol

rdtools.aggregation.**aggregation_insol**(*energy_normalized*, *insolation*, *frequency='D'*)

Insolation weighted aggregation

Parameters

- **energy_normalized** (pandas.Series) -- Normalized energy time series
- **insolation** (pandas.Series) -- Time series of insolation associated with each *energy_normalized* point
- **frequency** (str or pandas.tseries.offsets.DateOffset) -- Pandas frequency specification at which to aggregate. Default is daily aggregation. For more information see https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#dateoffset-objects

Returns **aggregated** -- Insolation weighted average, aggregated at frequency

Return type pandas.Series

8.2.9 Clear-Sky Temperature

Functions for estimating clear-sky ambient temperature.

get_clearsky_tamb(times, latitude, longitude)	Estimates the ambient temperature at latitude and longitude for the given times using a Gaussian rolling window.

rdtools.clearsky_temperature.get_clearsky_tamb

rdtools.clearsky_temperature.**get_clearsky_tamb**(*times*, *latitude*, *longitude*, *window_size=40*, *gauss_std=20*)

Estimates the ambient temperature at latitude and longitude for the given times using a Gaussian rolling window.

Parameters

- **times** (pandas.DatetimeIndex) -- A pandas DatetimeIndex, localized to local time
- **latitude** (float) -- Coordinates in decimal degrees.
- **longitude** (float) -- Coordinates in decimal degrees. Positive is east of the prime meridian.
- **window_size** (int, *default* 40) -- The window size in days to use when calculating rolling averages.
- **gauss_std** (int, *default* 20) -- The standard deviation in days to use for the Gaussian rolling window.

Returns clear sky ambient temperature

Return type pandas.Series

Notes

Uses data from images created by Jesse Allen, NASA's Earth Observatory using data courtesy of the MODIS Land Group.

- https://neo.sci.gsfc.nasa.gov/view.php?datasetId=MOD_LSTD_CLIM_M
- https://neo.sci.gsfc.nasa.gov/view.php?datasetId=MOD_LSTN_CLIM_M

8.2.10 Plotting

Functions for plotting degradation and soiling analysis results.

degradation_summary_plots(yoy_rd, yoy_ci, ...)	Create plots (scatter plot and histogram) that summarize degradation analysis results.
soiling_monte_carlo_plot(soiling_info, ...)	Create figure to visualize Monte Carlo of soiling profiles used in the SRR analysis.
soiling_interval_plot(soiling_info, ...[, ...])	Create figure to visualize valid soiling profiles used in the SRR analysis.
soiling_rate_histogram(soiling_info[, bins])	Create histogram of soiling rates found in the SRR analysis.
availability_summary_plots(power_system, ...)	Create a figure summarizing the availability analysis results.
tune_filter_plot(signal, mask[, ...])	This function allows the user to visualize filtered data in a Plotly plot, after tweaking the function's different parameters.

rdtools.plotting.degradation_summary_plots

rdtools.plotting.**degradation_summary_plots**(*yoy_rd*, *yoy_ci*, *yoy_info*, *normalized_yield*, *hist_xmin=None*, *hist_xmax=None*, *bins=None*, *scatter_ymin=None*, *scatter_ymax=None*, *plot_color=None*, *summary_title=None*, *scatter_alpha=0.5*, *detailed=False*)

Create plots (scatter plot and histogram) that summarize degradation analysis results.

Parameters

- **yoy_rd** (float) -- rate of relative performance change in %/yr
- **yoy_ci** (float) -- one-sigma confidence interval of degradation rate estimate
- **yoy_info** (dict) -- a dictionary with keys:
  - YoY_values - pandas series of right-labeled year on year slopes
  - renormalizing_factor - float value used to recenter data
  - exceedance_level - the degradation rate that was outperformed with a probability given by the exceedance_prob parameter in the *degradation.degradation_year_on_year()*
- **normalized_yield** (pandas.Series) -- PV yield data that is normalized, filtered and aggregated
- **hist_xmin** (float, *optional*) -- lower limit of x-axis for the histogram

- **hist_xmax** (float, *optional*) -- upper limit of x-axis for the histogram
- **bins** (int, *optional*) -- Number of bins in the histogram distribution. If omitted, `len(yoy_values) // 40` will be used
- **scatter_ymin** (float, *optional*) -- lower limit of y-axis for the scatter plot
- **scatter_ymax** (float, *optional*) -- upper limit of y-axis for the scatter plot
- **plot_color** (str, *optional*) -- color of the summary plots
- **summary_title** (str, *optional*) -- overall title for summary plots
- **scatter_alpha** (float, *default* `0.5`) -- Transparency of the scatter plot
- **detailed** (bool, *optional*) -- Color code points by the number of times they get used in calculating Rd slopes. Default color: 2 times (as a start and endpoint). Green: 1 time. Red: 0 times.

Note: It should be noted that the yoy_rd, yoy_ci and yoy_info are the outputs from *degradation.degradation_year_on_year()*.

Returns **fig** -- Figure with two axes

Return type matplotlib.figure.Figure

rdtools.plotting.soiling_monte_carlo_plot

rdtools.plotting.**soiling_monte_carlo_plot**(*soiling_info*, *normalized_yield*, *point_alpha=0.5*, *profile_alpha=0.05*, *ymin=None*, *ymax=None*, *profiles=None*, *point_color=None*, *profile_color='C1'*)

Create figure to visualize Monte Carlo of soiling profiles used in the SRR analysis.

Warning: The soiling module is currently experimental. The API, results, and default behaviors may change in future releases (including MINOR and PATCH releases) as the code matures.

Parameters

- **soiling_info** (dict) -- `soiling_info` returned by *soiling.SRRAnalysis.run()* or *soiling.soiling_srr()*.
- **normalized_yield** (pandas.Series) -- PV yield data that is normalized, filtered and aggregated.
- **point_alpha** (float, *default* `0.5`) -- tranparency of the `normalized_yield` points
- **profile_alpha** (float, *default* `0.05`) -- transparency of each profile
- **ymin** (float, *optional*) -- minimum y coordinate
- **ymax** (float, *optional*) -- maximum y coordinate
- **profiles** (int, *optional*) -- the number of stochastic profiles to plot. If not specified, plot all profiles.
- **point_color** (str, *optional*) -- color of the normalized_yield points
- **profile_color** (str, *default* `'C1'`) -- color of the stochastic profiles

Returns fig

Return type matplotlib.figure.Figure

rdtools.plotting.soiling_interval_plot

rdtools.plotting.**soiling_interval_plot**(*soiling_info*, *normalized_yield*, *point_alpha=0.5*, *profile_alpha=1*, *ymin=None*, *ymax=None*, *point_color=None*, *profile_color=None*)

Create figure to visualize valid soiling profiles used in the SRR analysis.

Warning: The soiling module is currently experimental. The API, results, and default behaviors may change in future releases (including MINOR and PATCH releases) as the code matures.

Parameters

- **soiling_info** (dict) -- soiling_info returned by *soiling.SRRAnalysis.run()* or *soiling.soiling_srr()*.
- **normalized_yield** (pandas.Series) -- PV yield data that is normalized, filtered and aggregated.
- **point_alpha** (float, *default* 0.5) -- tranparency of the normalized_yield points
- **profile_alpha** (float, *default* 1) -- transparency of soiling profile
- **ymin** (float, *optional*) -- minimum y coordinate
- **ymax** (float, *optional*) -- maximum y coordinate
- **point_color** (str, *optional*) -- color of the normalized_yield points
- **profile_color** (str, *optional*) -- color of the soiling intervals

Returns fig

Return type matplotlib.figure.Figure

rdtools.plotting.soiling_rate_histogram

rdtools.plotting.**soiling_rate_histogram**(*soiling_info*, *bins=None*)

Create histogram of soiling rates found in the SRR analysis.

Warning: The soiling module is currently experimental. The API, results, and default behaviors may change in future releases (including MINOR and PATCH releases) as the code matures.

Parameters

- **soiling_info** (dict) -- soiling_info returned by *soiling.SRRAnalysis.run()* or *soiling.soiling_srr()*.
- **bins** (int) -- number of histogram bins to use

Returns fig

Return type matplotlib.figure.Figure

rdtools.plotting.availability_summary_plots

rdtools.plotting.**availability_summary_plots**(*power_system*, *power_subsystem*, *loss_total*, *energy_cumulative*, *energy_expected_rescaled*, *outage_info*)

Create a figure summarizing the availability analysis results.

Because all of the parameters to this function are products of an AvailabilityAnalysis object, it is usually easier to use *availability.AvailabilityAnalysis.plot()* instead of running this function manually.

Warning: The availability module is currently experimental. The API, results, and default behaviors may change in future releases (including MINOR and PATCH releases) as the code matures.

Parameters

- **power_system** (pandas.Series) -- Timeseries total system power.
- **power_subsystem** (pandas.DataFrame) -- Timeseries power data, one column per subsystem.
- **loss_total** (pandas.Series) -- Timeseries system lost power.
- **energy_cumulative** (pandas.Series) -- Timeseries system cumulative energy.
- **energy_expected_rescaled** (pandas.Series) -- Timeseries expected energy, rescaled to match actual energy. This reflects interval energy, not cumulative.
- **outage_info** (pandas.DataFrame) -- A dataframe with information about system outages.

Returns **fig**

Return type matplotlib.figure.Figure

See also:

rdtools.availability.AvailabilityAnalysis.plot()

Examples

```
>>> aa = AvailabilityAnalysis(...)
>>> aa.run()
>>> fig = rdtools.plotting.availability_summary_plots(aa.power_system,
...     aa.power_subsystem, aa.loss_total, aa.energy_cumulative,
...     aa.energy_expected_rescaled, aa.outage_info)
```

rdtools.plotting.tune_filter_plot

rdtools.plotting.**tune_filter_plot**(*signal*, *mask*, *display_web_browser=False*)

This function allows the user to visualize filtered data in a Plotly plot, after tweaking the function's different parameters. The plot of signal colored according to mask can be zoomed in on, for an in-depth look.

Parameters

- **signal** (pandas.Series) -- Index of the Pandas series is a Pandas datetime index. Usually this is PV power or energy, but other signals will work.

- **mask** (pandas.Series) -- Pandas series of booleans, where included data periods are marked as True, and omitted-data periods occurs are marked as False. Should have the same detetime index as signal.
- **display_web_browser** (boolean, *default* False) -- When set to True, the Plotly graph is displayed in the user's web browser.

Returns

Return type Interactive Plotly graph, with the masked time series for the filter.

8.3 RdTools Change Log

8.3.1 v2.1.3 (January 6, 2022)

Bug fixes

- Fixed a plotting issue in *rdtools.plotting.availability_summary_plots()* with newer matplotlib versions, as well as an axis labeling error (GH #302)

Requirements

- Added support for python 3.10 and dropped support for python 3.6 (which reached end of life on Dec 23, 2021) (GH #302)
- Bumped several minimum package versions (GH #302):
  - h5py increased to 2.8.0 (released June 4, 2018)
  - pandas increased to 0.23.2 (released July 6, 2018)
  - scipy increased to 1.1.0 (released May 5, 2018)
  - statsmodels increased to 0.9.0 (released May 14, 2018)
- Update pinned versions of several dependencies in `requirements.txt` (GH #302)

Testing

- Drop python 3.6 and add 3.10 to the CI configuration (GH #302)
- Add new `assert_warnings` helper function to `conftest.py` (GH #302)

8.3.2 v2.1.2 (December 22, 2021)

Bug fixes

- XGBoost model saved in a version-stable json. Removes the xgboost version restriction of version 2.1.1. (GH #301, GH #304)

Requirements

- `joblib` removed as requirement (GH #304)
- XGBoost version in `requirements.txt` updated to 1.5.1 (GH #304)

Documentation

- Preferred citation updated (GH #296, GH #300)

8.3.3 v2.1.1 (November 30, 2021)

This patch release temporarily requires `xgboost < 1.5.0`. A future update will allow a wider version range. (GH #301, GH #297)

8.3.4 v2.1.0 (September 17, 2021)

API Changes

- The calculations internal to the soiling SRR algorithm have changed such that consecutive cleaning events are no longer removed. (GH #199, GH #189)
- The calculations internal to the soiling SRR algorithm have changed such that "invalid" intervals are retained at the beginning and end of the dataset for the purposes of the SRR Monte Carlo. Invalid intervals are those that do not qualify to be fit as soiling intervals based on `min_interval_length`, `max_relative_slope_error`, and `max_negative_step`. (GH #199, GH #272)
- The default `day_scale` parameter in soiling functions and methods was changed from 14 to 13. A recommendation to use an odd value along with a warning for even values was also added. (GH #199, GH #189)
- The default `min_interval_length` in soiling functions and methods was changed from 2 to 7. (GH #199)

Enhancements

- New object oriented end-to-end analysis API introduced with the *`TrendAnalysis`* class. Note the behavior has been adjusted since its original introduction in v2.1.0-beta.0. (GH #117, GH #232, GH #233, GH #263, GH #278, GH #285, GH #281)
- Add `sensor_filter_components` and `clearsky_filter_components` to *`TrendAnalysis`* (GH #236, GH #263)
- A new parameter `outlier_factor` was added to soiling functions and methods to enable better control of cleaning event detection. (GH #199)
- Boolean input kwarg `detailed` has been added to *`degradation_summary_plots`* to color-code degradation plots by the number of times data points are used in the degradation distribution. (GH #269, GH #282)
- *`degradation_year_on_year`* adds a new `usage_of_points` entry in the `calc_info` return dictionary. (GH #269, GH #282)
- *`clip_filter()`* updated to allow for different methods of clipping detection with the `model` parameter (GH #200)
- Add new function *`quantile_clip_filter()`* (GH #200).
- Add new function *`logic_clip_filter()`* (GH #200).

- Add new function *xgboost_clip_filter()* (GH #200).
- Add new function *tune_filter_plot()* (GH #200).

Bug fixes

- Unexpected recoveries when using `method=random_clean` in the soiling module have been fixed. (GH #199, GH #234)
- Improved NaN pixel handling in *get_clearsky_tamb()* (GH #274).

Documentation

- Corrected a typo in the *TrendAnalysis* docstring (GH #264)
- Enabled intersphinx so that function parameter types are linked to external documentation (GH #258)

Requirements

- Installation (`setup.py`) now requires `plotly`, `joblib`, `xgboost`, and `scikit-learn`
- Update pinned versions of several dependencies in `requirements.txt` and `docs/notebook_requirements.txt` (GH #289, GH #295)
- Add support for pvlib 0.9 and remove the `tables` dependency added in `v2.1.0b0` (GH #290)

Example Updates

- `TrendAnalysis_example_pvdaq4.ipynb` added
- `degradation_and_soiling_example_pvdaq_4.ipynb` updated to use the same artificial soiling signal imposed in new notebook `TrendAnalysis_example_pvdaq4.ipynb` throughout the analysis.
- `degradation_and_soiling_example_pvdaq_4.ipynb` updated to illustrate new clipping models and filter-tuning plots.

Contributors

- Mark Mikofski (@mikofski)
- Kevin Anderson (@kanderso-nrel)
- Michael Deceglie (@mdeceglie)
- Matthew Muller (@matt14muller)
- Kirsten Perry (@kperrynrel)
- Chris Deline (@cdeline)

8.3.5 v2.0.6 (July 16, 2021)

Bug Fixes

- Fix a failure of *get_clearsky_tamb()* with pandas 1.3.0 (GH #284)
- Change internal casting of timestamps into integers to use `.view()` instead of pandas deprecated `.astype()` method (GH #284)

Requirements

- Update specified versions of bleach in `docs/notebook_requirements.txt` and matplotlib in `requirements.txt` (GH #261)

Contributors

- Kevin Anderson (@kanderso-nrel)
- Michael Deceglie (@mdeceglie)

8.3.6 v2.0.5 (2020-12-30) and v2.1.0-beta.2 (2021-01-29)

Testing

- Add a flake8 code style check to the continuous integration checks (GH #231)
- Moved several pytest fixtures from `soiling_test.py` and `availability_test.py` to `conftest.py` so that they are shared across test files (GH #231)
- Add Python 3.9 to CI testing (GH #249)
- Fix test suite error raised when using pandas 1.2.0 (GH #251)

Documentation

- Organized example notebooks into a sphinx gallery (GH #240)

Requirements

- Add support for python 3.9 (GH #249)
- Update `requirements.txt` versions for numpy, scipy, pandas, h5py and statsmodels to versions that have wheels available for python 3.6-3.9. Note that the minimum versions are unchanged. (GH #249).

Contributors

- Kevin Anderson (@kanderso-nrel)
- Michael Deceglie (@mdeceglie)
- Chris Deline (@cdeline)

8.3.7 v2.0.4 and v2.1.0-beta.1 (December 4, 2020)

Bug Fixes

- Fix bug related to leading NaN values with *energy_from_power()*. This fixed a small normalization error in `degradation_and_soiling_example.ipynb` and slightly changed the clear-sky degradation results (GH #244, GH #245)

Contributors

- Kevin Anderson (@kanderso-nrel)

8.3.8 v2.1.0-beta.0 (November 20, 2020)

Enhancements

- Add new *analysis_chains* module to focus on objected-oriented analysis workflows combining other RdTools modules. Includes *TrendAnalysis* class for sensor- and clear-sky-based soiling and degradation analyses (GH #117).

Requirements

- tables added as a requirement (GH #196).

Example Updates

- New example notebook based on PVDAQ system #4 for the new *TrendAnalysis* analysis work flow (GH #196 and GH #117).
- Update `degradation_and_soiling_example_pvdaq_4.ipynb` example to match best practice, including `pvlib.get_total_irradiance()` in rdtools.interpolate (GH #196 and GH #117).
- Update `degradation_and_soiling_example_pvdaq_4.ipynb` example to use a single `soiling * ac_power` signal (GH #196).

Contributors

- Mike Deceglie (@mdeceglie)
- Kevin Anderson (@kanderso-nrel)
- Chris Deline (@cdeline)

8.3.9 v2.0.3 (November 20, 2020)

Requirements

- Change to `docs/notebook_requirements.txt`: notebook version from 5.7.8 to 6.1.5 and terminado version from 0.8.1 to 0.8.3 (GH #239)

Contributors

- Mike Deceglie (@mdeceglie)
- @dependabot

8.3.10 v2.0.2 (November 17, 2020)

Examples

- degradation_and_soiling_example.ipynb modified to not use `max_timedelta` parameter in *`interpolate()`* and *`energy_from_power()`* in Step 0 (GH #237)

Contributors

- Mike Deceglie (@mdeceglie)

8.3.11 v2.0.1 (October 30, 2020)

Deprecations

- The deprecation of *`pvwatts_dc_power()`* and *`normalize_with_pvwatts()`* has been reversed. (GH #227)

Contributors

- Mike Deceglie (@mdeceglie)
- Kevin Anderson (@kanderso-nrel)

8.3.12 v2.0.0 (October 20, 2020)

Version 2.0.0 adds experimental soiling and availability modules, plotting capability, and includes updates to normalization work flow. This major release introduces some breaking changes to the API. Details below.

API Changes

- The calculations internal to *normalize_with_pvwatts()* and *normalize_with_sapm()* have changed. Generally, when working with raw power data it should be converted to right-labeled energy with *energy_from_power()* before being used with these normalization functions (GH #105, GH #108).
- Remove `low_power_cutoff` parameter in *clip_filter()* (GH #84).
- Many kwargs have changed name (but not input order) to bring nomenclature into closer alignment with the DuraMAT pv-terms project: (GH #185)
  - *aggregation_insol()* first kwarg is now `energy_normalized`.
  - *degradation_year_on_year()*, *degradation_ols()* and *degradation_classical_decomposition()* first kwarg is now `energy_normalized`.
  - *normalized_filter()* input kwargs are now `energy_normalized`, `energy_normalized_low` and `energy_normalized_high`.
  - *poa_filter()* input kwargs are now `poa_global`, `poa_global_low` and `poa_global_high`.
  - *tcell_filter()* input kwargs are now `temperature_cell`, `temperature_cell_low` and `temperature_cell_high`.
  - *clip_filter()* input kwargs are now `power_ac` and `quantile`.
  - *csi_filter()* first two kwargs are now `poa_global_measured`, `poa_global_clearsky`.
  - *normalize_with_pvwatts()* pvwatts_kws dictionary keys have been renamed.
  - *pvwatts_dc_power()* input kwargs are now `poa_global`, `power_dc_rated`, `temperature_cell`, `poa_global_ref`, `temperature_cell_ref`, `gamma_pdc`.
  - *irradiance_rescale()* second kwarg is now `irrad_sim`

Deprecations

- The functions *pvwatts_dc_power()*, *sapm_dc_power()*, *normalize_with_pvwatts()*, and *normalize_with_sapm()* have been deprecated in favor of *normalize_with_expected_power()*. (GH #215)
- *delta_index()* and *check_series_frequency()* (GH #222)

Enhancements

- Add new *soiling* module to implement the stochastic rate and recovery method:
  - Create new class *SRRAnalysis* and helper function *soiling_srr()* (GH #112, GH #168, GH #169, GH #176, GH #208, GH #213)
  - Create functions *monthly_soiling_rates()* and *annual_soiling_ratios()* (GH #193, GH #207)
- Create new module *availability* with the class *AvailabilityAnalysis* for estimating timeseries system availability (GH #131)

- Add new function *normalize_with_expected_power()* (GH #173).
- Add new functions *energy_from_power()* and *interpolate()* (GH #105, GH #108, GH #182, GH #212).
- Add new function *normalized_filter()* (GH #139)
- Add new *plotting* module for generating standard plots (GH #138, GH #131)
- Add parameter `convergence_threshold` to *irradiance_rescale()* (GH #152).

Bug fixes

- Allow `max_iterations=0` in *irradiance_rescale()* (GH #152).

Testing

- Add Python 3.7 and 3.8 to CI testing (GH #135).
- Add CI configuration based on the minimum dependency versions (GH #197)

Documentation

- Create sphinx documentation and set up ReadTheDocs (GH #125).
- Add guides on running tests and building sphinx docs (GH #136).
- Improve module-level docstrings (GH #137).
- Update landing page and add new "Inverter Downtime" documentation page based on the availability notebook (GH #131)

Requirements

- Drop support for Python 2.7, minimum supported version is now 3.6 (GH #135).
- Increase minimum pvlib version to 0.7.0 (GH #170)
- Update requirements.txt and notebook_requirements.txt to avoid conflicting specifications. Taken together, they represent the complete environment for the notebook example (GH #164).
- Add minimum matplotlib requirement of 3.0.0 (released September 18, 2018) (GH #197)
- Increase minimum numpy version from 1.12 (released January 15, 2017) to 1.15 (released July 23, 2018) (GH #197)

Example Updates

- Seed `numpy.random` to ensure repeatable results (GH #164).
- Use *normalized_filter()* instead of manually filtering the normalized energy timeseries. Also updated the associated mask variable names (GH #139).
- Add soiling section to the original example notebook.
- Add a new example notebook that analyzes data from a PV system located at NREL's South Table Mountain campus (PVDAQ system #4) (GH #171).

- Explicitly register pandas datetime converters which were deprecated.
- Add new `system_availability_example.ipynb` notebook (GH #131)

Contributors

- Mike Deceglie (@mdeceglie)
- Kevin Anderson (@kanderso-nrel)
- Chris Deline (@cdeline)
- Will Vining (@wfvining)

8.3.13 v1.2.3 (April 12, 2020)

- Updates dependencies
- Versioneer bug fix
- Licence update

Contributors

- Mike Deceglie (@mdeceglie)

8.3.14 v1.2.2 (October 12, 2018)

Patch that adds author email to enable pypi deployment

Contributors

- Mike Deceglie (@mdeceglie)

8.3.15 v1.2.1 (October 12, 2018)

This update includes automated testing and deployment to support development along with some bug fixes to the library itself, a documented environment for the example notebook, and new example results to reflect changes in the example dataset. It addresses GH #49, GH #76, GH #78, GH #79, GH #80, GH #85, GH #86, and GH #92.

Contributors

- Mike Deceglie (@mdeceglie)
- Adam Shinn (@abshinn)
- Chris Deline (@cdeline)
- nb137 (@nb137)

8.3.16 v1.2.0 (March 30, 2018)

This incorporates changes including:

- Enables users to control confidence intervals reported in degradation calculations (GH #59)
- Adds python 3 support (GH #56 and GH #67)
- Fixes bugs (GH #61 GH #57)
- Improvements/typo fixes to docstrings
- Fixes error in check for two years of data in degradation_year_on_year
- Improves the calculations underlying irradiance_rescale

Contributors

- Mike Deceglie (@mdeceglie)
- Ambarish Nag (@ambarishnag)
- Gregory Kimball (@GregoryKimball)
- Chris Deline (@cdeline)
- Mark Mikofski (@mikofski)

8.3.17 v1.1.3 (December 6, 2017)

This patch includes the following changes:

1. Update the notebook for improved plotting with Pandas v.0.21.0
2. Fix installation bug related to package data

Contributors

- Mike Deceglie (@mdeceglie)
- Chris Deline (@cdeline)

8.3.18 v1.1.2 (November 6, 2017)

This patch includes the following changes:

1. Fix bugs in installation
2. Update requirements
3. Notebook plots made compatible with pandas v.0.21.0

Contributors

- Mike Deceglie (@mdeceglie)

8.3.19 v1.1.1 (November 1, 2017)

This patch:

1. Improves documentation
2. Fixes installation requirements

Contributors

- Mike Deceglie (@mdeceglie)
- Adam Shinn (@abshinn)
- Chris Deline (@cdeline)

8.3.20 v1.1.0 (September 30, 2017)

This update includes the addition of filters, functions to support a clear-sky workflow, and updates to the example notebook.

Contributors

- Mike Deceglie (@mdeceglie)
- Adam Shinn (@abshinn)
- Ambarish Nag (@ambarishnag)
- Gregory Kimball (@GregoryKimball)
- Chris Deline (@cdeline)
- Jiyang Yan (@yjy1663)

8.4 Developer Notes

This page documents some of the workflows specific to RdTools development.

8.4.1 Installing RdTools source code

To make changes to RdTools, run the test suite, or build the documentation locally, you'll need to have a local copy of the git repository. Installing RdTools using pip will install a condensed version that doesn't include the full source code. To get the full source code, you'll need to clone the RdTools source repository from Github with e.g.

```
git clone https://github.com/NREL/rdtools.git
```

from the command line, or using a GUI git client like Github Desktop. This will clone the entire git repository onto your computer.

8.4.2 Installing RdTools dependencies

The packages necessary to run RdTools itself can be installed with `pip`. You can install the dependencies along with RdTools itself from PyPI:

```
pip install rdtools
```

This will install the latest official release of RdTools. If you want to work with a development version and you have cloned the Github repository to your computer, you can also install RdTools and dependencies by navigating to the repository root, switching to the branch you're interested in, for instance:

```
git checkout development
```

and running:

```
pip install .
```

This will install based on whatever RdTools branch you have checked out. You can check what version is currently installed by inspecting `rdtools.__version__`:

```
>>> rdtools.__version__
'1.2.0+188.g5a96bb2'
```

The hex string at the end represents the hash of the git commit for your installed version.

Installing optional dependencies

RdTools has extra dependencies for running its test suite and building its documentation. These packages aren't necessary for running RdTools itself and are only needed if you want to contribute source code to RdTools.

Note: These will install RdTools along with other packages necessary to build its documentation and run its test suite. We recommend doing this in a virtual environment to keep package installations between projects separate!

Optional dependencies can be installed with the special syntax:

```
pip install rdtools[test]  # test suite dependencies
pip install rdtools[doc]   # documentation dependencies
```

Or, if your local repository has an updated dependencies list:

```
pip install .[test]  # test suite dependencies
pip install .[doc]   # documentation dependencies
```

8.4.3 Running the test suite

RdTools uses pytest to run its test suite. If you haven't already, install the testing dependencies (*Installing optional dependencies*).

To run the entire test suite, navigate to the git repo folder and run

```
pytest
```

For convenience, pytest lets you run tests for a single module if you don't want to wait around for the entire suite to finish:

```
pytest rdtools/test/soiling_test.py
```

And even a single test function:

```
pytest rdtools/test/soiling_test.py::test_soiling_srr
```

You can also evaluate code coverage when running the test suite using the coverage package:

```
coverage run -m pytest
coverage report
```

The first line runs the test suite and keeps track of exactly what lines of code were run during test execution. The second line then prints out a summary report showing how much much of each source file was executed in the test suite. If a percentage is below 100, that means a function isn't tested or a branch inside a function isn't tested. To get specific details, you can run `coverage html` to generate a detailed HTML report at `htmlcov/index.html` to view in a browser.

8.4.4 Checking for code style

RdTools uses flake8 to validate code style. To run this check locally you'll need to have flake8 installed (see *Installing optional dependencies*). Then navigate to the git repo folder and run

```
flake8
```

Or, for a more detailed report:

```
flake8 --count --statistics --show-source
```

8.4.5 Building documentation locally

RdTools uses Sphinx to build its documentation. If you haven't already, install the documentation dependencies (*Installing optional dependencies*).

Once the required packages are installed, change your console's working directory to `rdtools/docs/sphinx` and run

```
make html
```

Note that on Windows, you don't actually need the `make` utility installed for this to work because there is a `make.bat` in this directory. Building the docs should result in output like this:

```
(venv)$ make html
Running Sphinx v1.8.5
making output directory...
[autosummary] generating autosummary for: api.rst, example.nblink, index.rst, readme_
↪link.rst
[autosummary] generating autosummary for: C:\Users\KANDERSO\projects\rdtools\docs\
↪sphinx\source\generated\rdtools.aggregation.aggregation_insol.rst, C:\Users\
↪KANDERSO\projects\rdtools\docs\sphinx\source\generated\rdtools.aggregation.rst, C:\
↪Users\KANDERSO\projects\rdtools\docs\sphinx\source\generated\rdtools.clearsky_
↪temperature.get_clearsky_tamb.rst, C:\Users\KANDERSO\projects\rdtools\docs\sphinx\
↪source\generated\rdtools.clearsky_temperature.rst, C:\Users\KANDERSO\projects\
↪rdtools\docs\sphinx\source\generated\rdtools.degradation.degradation_classical_
↪decomposition.rst, C:\Users\KANDERSO\projects\rdtools\docs\sphinx\source\generated\
↪rdtools.degradation.degradation_ols.rst, C:\Users\KANDERSO\projects\rdtools\docs\
↪sphinx\source\generated\rdtools.degradation.degradation_year_on_year.rst, C:\Users\
↪KANDERSO\projects\rdtools\docs\sphinx\source\generated\rdtools.degradation.rst, C:\
↪Users\KANDERSO\projects\rdtools\docs\sphinx\source\generated\rdtools.filtering.clip_
↪filter.rst, C:\Users\KANDERSO\projects\rdtools\docs\sphinx\source\generated\rdtools.
↪filtering.csi_filter.rst, ..., C:\Users\KANDERSO\projects\rdtools\docs\sphinx\
↪source\generated\rdtools.normalization.normalize_with_pvwatts.rst, C:\Users\
↪KANDERSO\projects\rdtools\docs\sphinx\source\generated\rdtools.normalization.
```

(continues on next page)

(continued from previous page)

```
building [mo]: targets for 0 po files that are out of date
building [html]: targets for 4 source files that are out of date
updating environment: 33 added, 0 changed, 0 removed
reading sources... [100%] readme_link
looking for now-outdated files... none found
pickling environment... done
checking consistency... done
preparing documents... done
writing output... [100%] readme_link
generating indices... genindex py-modindex
writing additional pages... search
copying images... [100%] ../build/doctrees/nbsphinx/example_33_2.png
copying static files... done
copying extra files... done
dumping search index in English (code: en) ... done
dumping object inventory... done
build succeeded.

The HTML pages are in build\html.
```

If you get an error like `Pandoc wasn't found`, you can install it with conda:

```
conda install -c conda-forge pandoc
```

The built documentation should be in `rdtools/docs/sphinx/build` and opening `index.html` with a web browser will display it.

8.4.6 Contributing

Community participation is welcome! New contributions should be based on the `development` branch as the `master` branch is used only for releases.

RdTools follows the PEP 8 style guide. We recommend setting up your text editor to automatically highlight style violations because it's easy to miss some issues (trailing whitespace, etc) otherwise.

Additionally, our documentation is built in part from docstrings in the source code. These docstrings must be in NumpyDoc format to be rendered correctly in the documentation.

Finally, all code should be tested. Some older tests in RdTools use the unittest module, but new tests should all use pytest.

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

r

INDEX

Symbols

A

C

D

E

F

G

I

L

M

N

O

P

Q

R

S

T

X