PVRPM

Release 1.5.0

Brandon Silva, Paul Lunis

Feb 23, 2022

TUTORIALS:

1 About **1**

1.1 Installation 1
1.2 Getting Started 2
1.3 PVRPM Assumptions / Limitations 10
1.4 Logic Diagram 11
1.5 Example YAML Configuration 13
1.6 pvrpm 24

2 Indices and tables **27**

CHAPTER

ONE

ABOUT

In photovoltaics (PV), the ability to perform an accurate techno-economic analysis is essential. Creating economically efficient PV systems is beneficial to both consumers and producers alike. This package: Python PhotoVoltaic Reliability Performance Model (PyPVRPM), fills this need. PyPVRPM is a simulation tool that uses NREL's SAM software to model the performance of a PV plant throughout its lifetime. It simulates failures, repairs, and monitoring across the lifespan based on user-defined distributions and values. This allows a more accurate representation of cost and availability throughout a lifetime than SAM's base simulation done from the GUI. By varying repair rates and monitoring solutions, one can compare different configurations to find the most optimal setup for implementing an actual PV power plant.

A few assumptions are taken in the tool, alongside specific ways calculations are made. Please see the logic diagram to understand how the simulation works. Also, view the example configuration to get an idea of setting up your case.

PVRPM Requires a valid SAM case created in its GUI before use. The SAM case provides the information for PVRPM to operate, alongside simulations using its LCOE calculator defined in the case, weather files, etc. Please see the getting started tutorials to learn how to set up your SAM case and PVRPM YAML configuration.

1.1 Installation

This document covers installation and setup of the tool. The tool requires you to build a valid case in SAM, so if you haven't already download and install SAM from here: https://sam.nrel.gov/download.html

Currently, the supported SAM version is 2021.12.02!

SAM can be installed on Windows, MAC, or Linux.

1.1.1 Installation

Requires python >= 3.8

Works on any x64 OS.

Recommended using pip: (Replace *@master* with the branch release name if you want a release version)

```
# for latest development branch
pip install git+https://github.com/FSEC-Photovoltaics/pvrpm-lcoe/@master

# for specific version
pip install git+https://github.com/FSEC-Photovoltaics/pvrpm-lcoe/@vx.x.x
```

Using the wheel file downloaded from https://github.com/FSEC-Photovoltaics/pvrpm-lcoe/releases

```
pip install wheel
pip install pvrpm-x.x.x-py3-none-any.whl
```

Manually:

```
git clone https://github.com/FSEC-Photovoltaics/pvrpm-lcoe
cd pvrpm-lcoe
python setup.py install
```

If you want to build the documentation:

```
git clone https://github.com/FSEC-Photovoltaics/pvrpm-lcoe
cd pvrpm-lcoe
pip install .[docs]
cd docs
make html
```

If you want to run automated tests (will take a while based on compute power):

```
git clone https://github.com/FSEC-Photovoltaics/pvrpm-lcoe
cd pvrpm-lcoe
pip install .[testing]
pytest
```

1.2 Getting Started

Make sure to follow :doc:`Installation <tutorial_1installation>` and install SAM before continuing here!

PVRPM enhances SAM's models for PV systems to obtain a more accurate LCOE than its base simulation. It also allows studying the effects of monitoring and repair techniques on PV systems throughout their lifetime.

To get started, create a new SAM case in the SAM GUI. From there, you **must choose the Detailed Photovoltaic model** for PVRPM to work. This is required because PVRPM need's specific parameters that only exist in this model.

Then, choose your financial model. It must be a financial model that supports lifetime losses. Any financial model under the *Detailed Photovoltaic Model* will work **except** the *LCOE Calculator (FCR Method)* and *No Financial Model*. Please read SAM's documentation for help in setting up the case as it goes into more detail on these models.

Once that is set up, you can download the example configuration and modify it as needed. Below explains from start to finish of how to run a simulation with PVRPM.

1.2.1 Exporting the SAM Case

PVRPM works by reading the information in your SAM case and the PVRPM YAML configuration to properly run the simulation. For the first step, you need to export your SAM case to JSON files, which PVRPM uses to read in your SAM case.

To do this: 1. Open the SAM GUI, then open your `.sam` case file. 2. Once it is open, click the drop-down menu for the case, which is located next to the case name on the top bar. 3. Click `Generate Code`, then `PySAM JSON` (**not** `JSON for Inputs`). 4. A file explorer window will open, allowing you to select a place to save the JSON files. Select an empty folder.

Once this is done, the selected folder will contain a few JSON files. The names of these files will follow the convention of `case-name_module.json` where `case-name` is the name of the case, and `module` is the module that JSON represents.

Pay attention to what modules you have; you'll need to know that for the next part. You can remove the `.h` and `.so` files.

1.2.2 Configuring PVRPM

This will go over every configuration option to set up the case study step by step. The example configuration is also heavily commented on to help with the parameters. Also, please study the logic diagram as it can help when setting up the configuration here. *Also note all of the values listed in examples are entirely arbitrary and do not represent a realistic case.*

You can download the example configuration file `here` or view the example configuration file *here*.

1.2.3 Module Order

These modules correspond to the modules listed in the JSON files you obtained. The modules must be simulated in the correct order for PVRPM to properly run simulations using PySAM (SAM's Python interface). A typical order is in the example, but you may need to modify it depending on your case.

To make the module order easy to set up, go to this website: https://nrel-pysam.readthedocs.io/en/master/Configs.html#sam-simulation-configurations

Then, find the modules you have under `SSC Compute Modules` column in the table; the SAM configuration column should be one of the `Detailed PV Model` rows. Once you find one containing all your modules, put them in the order as they appear in the `SSC Compute Modules` column.

```
module_order: # the order the modules of this case should be executed
              # check to see what modules you use by the name of the json output files
              # typically, this should be Pvsamv1 -> grid -> utiltiy -> others
              # see https://nrel-pysam.readthedocs.io/en/master/Configs.html#sam-
↪simulation-configurations
# Only detailed PV models are supported, which is Pvsamv1, your case must have this␣
↪module
# Also, only LCOE calculators that support lifetime are allowed, PVRPM will check this␣
↪and notify you if your LCOE calculator doesn't support lifetime
  - Pvsamv1
  - Grid
  - Utilityrate5
  - Cashloan
```

1.2.4 Run Setup

Here, set the results folder location. On Windows, use only one backslash to separate directories; you do not need to escape them or spaces. Then set the number of realizations you want to run and the confidence interval for calculating results.

The results folder and number of realizations can be overridden from the command line.

```
### Run Setup ###
# results folder doesn't need to exist, it will be created if it doesnt already
# For windows use single backslash. Do not need to escape spaces in folder names
results_folder: /path/to/results/folder
num_realizations: 2 # number of realizations to run
conf_interval: 90 #XX % confidence interval around the mean will be calculated for␣
↪summary results
```

(continues on next page)

(continued from previous page)

1.2.5 Case Setup

Set up the basics of the case. Set the number of trackers, combiners, and transformers. Set `num_trackers` to `0` if you are not using trackers. The worst-case tracker can be set to true if you are using trackers. This means that failures in tracking components result in them being stuck in the worst way: the module is pointing the opposite of the sun's travel arc.

```
### Case Setup ###
num_combiners: 2 # total number of DC combiner boxes
num_transformers: 1 # total number of Transformers
num_trackers: 2 # total number of trackers

### Financial Inputs ###
present_day_labor_rate: 100 # dollars per hour
inflation: 2.5 # in percent

### Failure Tracker Algorithm ###
use_worst_case_tracker: false
```

1.2.6 Component Level Setup

Each component level requires a setup of failures, monitoring, and repairs. This is required for module, string, combiner, inverter, disconnect, transformer, and grid. However, if you are not setting a component level to fail (`can_fail: false`), you can remove the rest of the sections below it. For example, if string's `can_fail: false`, I can remove the `failure`, `monitoring`, and `repair` sections.

There are many options for types of failures, monitoring, and the component, alongside various combinations to get different behaviors.

Keep in mind the way these operations are as follows:

1. First, using the distributions defined for failures, monitoring, and repairs, a `time_to_failure`, `time_to_detection`, and `time_to_repair` is generated for each component.
2. `time_to_failure` then counts down to 0. Once a failure occurs, `time_to_detection` counts down to 0 (if monitoring is defined). Finally, `time_to_repair` counts to 0, which repairs the component and resets these values.

Component Behaviors

`can_fail` can be set to `true` or `false`, which dictates whether the components in the component level (`module`, `string`, etc.) can fail. If this is `false`, then nothing will fall for this level. You can remove the `failures`, `repairs`, and `monitoring` sections. `can_repair` dictates if the components can be repaired at this level. Typically, leave this `true` if components can fail. `can_monitor` turns on or off component-level monitoring. This signifies some type of monitoring in place; if you want to simulate this type of monitoring, set this to `true`.

Warranty

Components can be set to have a warranty. Components covered under warranty do not incur repair costs when they are repaired. A repaired component resets the warranty to the specified time in this section. For no warranty, remove this section.

Distributions for Failures, Repairs, Monitoring

Every failure, repair, and monitoring mode requires a distribution to be defined that dictates how long until the specified action occurs.

PVRPM has some built-in distributions, where only a mean and standard deviation is needed to model the distribution properly. Under the hood, PVRPM uses `scipy.stats` distributions to generate samples. However, `scipy.stats` documentation for each function is unclear on how to convert the mean and std into usable values for the distribution, which is why PVRPM will do that for you. However, not every single `scipy` distribution is wrapped by PVRPM. These are the distributions wrapped by PVRPM (use these as the distribution option):

- exponential
- normal
- uniform
- lognormal
- weibull

Using these distributions as the `distribution` parameter for any failure, repair, or monitoring only requires you to provide the mean and standard deviation **in days**. The `weibull` distribution also allows you to give the `shape` for this distribution instead of the standard deviation. On the Wikipedia page for the weibull distribution is the parameter `k` and `lambda` is calculated from the mean. Using the `std` option, make sure it is large since weibull distributions have large STDs by design.

If these distributions don't properly model your data, you can use any distribution listed in the scipy.stats module. The `distribution` parameter in the configuration should be set to the function name of the distribution in the `scipy.stats` module. The `parameters` key will then be keyword arguments to the function. Make sure to carefully read scipy's documentation, as each function is different in how you need to define it. Remember, the samples from the distribution represent the number of days before that event occurs for a component.

```yaml
distribution: normal
parameters: # parameters for distribution chosen above, either mean and std for a built␣
↪in distribution or kwargs to the scipy function
  mean: 1460 # years converted to days
  std: 365 # days
```

Failure Setup

PVRPM currently has two failure modes: total failures and concurrent failures. Total failure modes use the shortest time to failure taken from the defined failures as the time it takes for a component to completely fail. Every component gets a different time to failure, depending on the samples drawn from the distribution.

For a total failure setup, the failure requires the distribution, its parameters, the labor time to fix a component, and the cost to repair the component.

Optionally, there are two fraction modes for a failure: `fraction` and `decay_fraction`. Setting the `fraction` will tell PVRPM to fail that fraction (between 0 and 1) of components in the component level consistently throughout the simulation. This means PVRPM will maintain `fraction` of the components with this failure mode throughout the

simulation. Remember, PVRPM will always pick the failure mode in this section **with the shortest time to failure**, so if you set two failures mode, where one is always shorter than the other, then the longer failure mode will never occur, **even if the fraction is defined on the longer failure mode**. The `decay_fraction` also selects `decay_fraction` of the components to fail; however, it decays with each failure. If you set `decay_fraction` to 0.5, then at first, 50 percent of the components will fail with this failure mode, then 25 percent, then 12.5 percent, etc., until it approaches 0, which in reality would mean the number of failures from this mode would be 0 when `decay_fraction` is small enough.

A typical setup of failures is to have a long "end of life" failure with a large time, and failures with shorter time to failures with a `fraction` or `decay_fraction`, so some will fail with the shorter failures, and most will fail with the end of life failure.

Concurrent failures work the same way as above, except each failure mode is counted **concurrently**. This means that failure modes defined as concurrent failures **do not have the shortest time picked among the modes; instead, each failure mode will fail the component independent of each other and the total failure mode**. You can view this mode as "partial failures", where failures of this nature happen more often than total failures but cost less and are faster to repair. You can use `fraction` and `decay_fraction` here as needed.

A typical setup for concurrent failure modes is to list routine failures every year or two to a `fraction` of the components.

Total failure mode chooses the quickest time to failure from the different modes, and concurrent failure modes all operate independently of each other; they fail each component independent of other failures. Further note, **when a component is repaired from a *total failure*, all *concurrent failures* get reset** since this is a full replacement, and the partial failures that affected the old component won't affect the new one.

```yaml
failures:
  eol_failures: # this key name can be anything you want
    distribution: normal
    parameters: # parameters for distribution chosen above, either mean and std for a␣
↪built in distribution or kwargs to the scipy function
      mean: 3650 # years converted to days
      std: 365 # days
    labor_time: 2 # in hours
    cost: 322 # in USD
  routine_failures: # this key name can be anything you want
    distribution: normal
    parameters:
      mean: 365 # mean in days, or you can do 1 / (num_failures / year * 365)
      std: 365
    labor_time: 2 # in hours
    cost: 322 # in USD
    fraction: 0.1 # > 0 and < 1, fraction of these components that are normal failures,␣
↪maintained throughout the simulation
  defective_failures: # this key name can be anything you want
    distribution: exponential
    parameters:
      mean: 100 # mean in days, or you can do 1 / (num_failures / year * 365)
    labor_time: 2 # in hours
    cost: 322 # in USD
    decay_fraction: 0.2 # > 0 and < 1, fraction of these components that are defective

concurrent_failures: # this happens all in parallel, independent of each other
  cell_failure: # this key name can be anything you want
    distribution: normal
    parameters: # parameters for distribution chosen above, either mean and std for a␣
↪built in distribution or kwargs to the scipy function
```

(continues on next page)

(continued from previous page)

```yaml
      mean: 365 # years converted to days
      std: 365 # days
    labor_time: 2 # in hours
    cost: 322 # in USD
    decay_fraction: 0.2
  wiring_failure: # this key name can be anything you want
    distribution: normal
    parameters:
      mean: 365 # mean in days, or you can do 1 / (num_failures / year * 365)
      std: 365
    labor_time: 2 # in hours
    cost: 322 # in USD
    fraction: 0.1 # > 0 and < 1, fraction of these components that are normal failures, maintained throughout the simulation
```

Repair Setup

Repairs are much more straightforward. They only need the distribution and its parameters defined for every repair mode. You can either have one repair mode that applies to all failures or a repair mode for each failure mode. You also must list repairs for total failures and concurrent failures separately.

```yaml
repairs:
  all_repairs: # this key name can be anything you want
    distribution: lognormal
    parameters:
      mean: 60 # in days
      std: 20 # in days

concurrent_repairs:
  cell_repair: # this key name can be anything you want
    distribution: lognormal
    parameters:
      mean: 7 # in days
      std: 3 # in days

  wire_repair: # this key name can be anything you want
    distribution: lognormal
    parameters:
      mean: 3 # in days
      std: 3 # in days
```

Monitoring

Multiple monitoring modes are available for components. You can remove any section you are not using. It is also optional; you can disable all monitoring, in which components that fail are immediately repaired. The modes available are:

- Component Level: monitoring at the level of the component, which usually offers quick time to detection.
- Cross Level: monitoring done at a higher level to lower-level components. Meaning inverter monitoring string, combiner, etc.
- Independent: monitoring done independently of any component level, such as drone IR imaging.

Component level monitoring is defined under each component level's configuration. It simply requires distribution and parameters that signify the time to detection in days to detect a failed component in this level.

```yaml
monitoring:
  normal_monitoring: # this key name can be anything you want
    distribution: exponential
    parameters:
      mean: 5
```

Cross-level monitoring is a bit more complex. Alongside the distribution and parameters, some thresholds control how the monitoring works. A `global_threshold` option defines the fraction of components in the monitored level **must fail** before monitoring can detect failed components. This can be seen as enough modules to fail before monitoring at the inverter can start detecting those failures. In PVRPM, this is replicated by the `global_threshold` must be met before `time to detection` counts down. There is also a `failure_per_threshold`, the fraction of **lower level** components that must fail **per upper-level component**. For example, if monitoring at the string with a `failure_per_threshold` of 0.1, then 10 percent of modules under a single string must fail before the string monitoring can detect module failures. Both thresholds can be defined simultaneously, but one must be defined for this monitoring to work.

```yaml
component_level_monitoring:
# lists what monitoring each component level has for levels BELOW it
# this is for cross level monitoring only, for defining monitoring at each level use the␣
↪monitoring distributions above
string: # component level that has the monitoring for levels below
  module: # componenet level that is below's key. same keys used above: module, string,␣
↪combiner, inverter, disconnect, grid, transformer
    global_threshold: 0.2 # fraction on [0, 1] that specifies how many of this component␣
↪type must fail before detection can occur.
                          # this means that until this threshold is met, component␣
↪failures can never be detected
                          # In the simulation, the time to detection doesn't count down␣
↪until this threshold is met, which at that point the compouning function will be used␣
↪along with the distribution as normal
    failure_per_threshold: 0.1 # the fraction of components that must fail per string␣
↪for failures to be detected at that specified string, or if the total number of␣
↪failures reach the global_threshold above
    distribution: normal # distribution that defines how long this monitoring takes to␣
↪detect a failure at this level (independent)
                         # the value calculated from this distribution will be reduced␣
↪by the compounding factor for every failure in this level
    parameters:
      mean: 1200
      std: 365
```

(continues on next page)

(continued from previous page)

```yaml

combiner:
  string:
      failure_per_threshold: 0.2 # this fraction of strings must fail on a specific␣
↪combiner for detection for those to start
      # failures are not compounded globally in this case, only per each combiner
      distribution: normal # distribution that defines how long this monitoring takes to␣
↪detect a failure at this level (independent)
      parameters:
        mean: 1825
        std: 365
```

Independent monitoring works outside of component levels. It represents monitoring that **detects all failures in any component level** instantly. It can happen statically every set number of days, defined by the `interval`, or at a threshold of failed components. There are a few ways to define this threshold. First, the threshold can be defined as `global_threshold`, which works differently than cross-level monitoring. This value is based on the **DC availability**, meaning the power reaching the inverter. This is calculated using the operating strings and combiners to determine how many modules reach the inverter. With this, combiners and strings are weighted higher than module failures.

The other way to define a threshold is more similar to cross-level monitoring. Using the `failure_per_threshold` sets a threshold of failed components for **each level** that must be reached before monitoring occurs. This uses OR logic, meaning only one level has to drop below this threshold for the independent monitoring for **all levels**.

Finally, you can combine all these arguments together; `interval`, `global_threshold`, and `failure_per_threshold`.

You must specify the labor time for each independent monitoring defined, which is in hours for other parameters. There is also **an optional distribution and parameters** that can be defined as the `time_to_detection` for components under the levels when the independent monitoring occurs. Think of it as the time it takes to complete the independent monitoring. Not setting this means that the `time_to_detection` gets set to zero when independent monitoring occurs.

```yaml
indep_monitoring:
  drone_ir:  # this name can be anything you want!
    interval: 1095 # interval in days when this static monitoring occurs
    cost: 50000 # cost of this monitoring in USD
    labor_time: 1 # in hours
    distribution: normal
    parameters:
      mean: 14
      std: 5
    levels: # the component levels this detects on
      - module
      - string
      - combiner

  drone_ir2: # list as many static monitoring methods as you want
    interval: 365 # this monitoring will happen every 365 days, alongside the threshold.
    # a indep monitoring triggered by a threshold RESETs the countdown to the interval
    global_threshold: 0.1 # if DC availability drops by this threshold amount, then this␣
↪indep monitoring will occur
    # DC availability is the DC power reaching the inverter(s), which is affected by␣
↪combiners, strings, and module failures
    failure_per_threshold: 0.2 # this threshold is PER LEVEL, if the availability of ANY␣
↪of the defined levels drops by this threshold amount, this indep monitoring will occur
```

(continues on next page)

(continued from previous page)

```yaml
    cost: 100000
    labor_time: 1 # in hours
    levels:
      - module
      - combiner
      - string

  drone_ir3: # list as many static monitoring methods as you want
    failure_per_threshold: 0.2 # this threshold is PER LEVEL, people if the availability of ANY of the defined levels drops by this threshold amount, this indep monitoring will occur
    cost: 1500
    labor_time: 1 # in hours
    levels:
      - module
```

1.2.7 Running the simulation

The example configuration provided shows how all these options are defined; please consult it as necessary.

Now that you have your SAM case JSONs, and your PVRPM configuration, you can run the simulation:

```
pvrpm run --case /path/to/directory/with/jsons /path/to/pvrpm/config.yaml
```

You can also parallelize realizations to decrease the overall run time. To use all your CPU cores to run PVRPM:

```
pvrpm run --case /path/to/directory/with/jsons --threads 0 /path/to/pvrpm/config.yaml

PVRPM will alert you to unknown keys in your configuration if you misspelled something and tell you any incorrect or missing parameters you may have.
```

Once the simulation is completed, result graphs and CSV files will be saved to the defined results folder.

1.3 PVRPM Assumptions / Limitations

PVRPM makes a few assumptions in order to be able to run a realistic simulation efficiently. Along with these assumptions arise limitations to what the model can simulate realistically.

To calculate LCOE, NPV, and other data, PVRPM uses SAMs simulation capabilities for PV systems. With all of the configuration for failures, repairs, and monitoring, it generates availability for DC and AC power and the OM yearly cost for the system. These three parameters are what SAM uses in its simulation for each realization to calculate all of the output statistics.

With this, everything PVRPM does must boil down to these three parameters. In doing so, some assumptions must be made to reduce computation complexity. The primary assumption is that failures, repairs, and monitoring are stochastic and can be modeled using statistical distribution. The distribution should consider the many factors that go into these areas, like weather, type of modules, quality of equipment and monitoring, etc., since PVRPM does not simulate these real-world phenomena. Alongside this PVRPM does not account for changes in the system size, it is considered static across the lifetime of the system; that is, the number of components stays the same across the simulation lifetime.

As of right now, failures are assumed to be a total component failure, where availability of the component while failed is zero. In a future release, partial failures will be available to simulate reduced functionality because of a partial failure

of the component but still provide greater than 0 availability in this degraded state. It also assumes that repair costs for these failures do not rise with inflation; only the labor rates rise with yearly inflation rates (except for tracker repair costs, which do rise with inflation). The yearly inflation rate is also set to be the same each year, and new labor costs are calculated only at the beginning of every year. When components are repaired with a warranty, it is assumed that the new component has the full warranty time defined in that component's configuration. Warranties are also only applied on successful repair, so if a warranty runs out before a repair can occur, that component is repaired out of warranty.

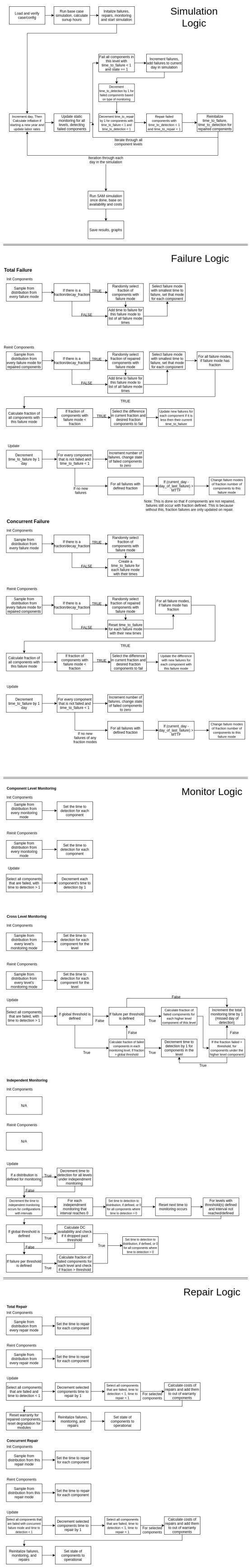
Availability is also based upon the daylight hours for the configured location via the weather file in SAM, except for the grid component level, which should be up 24 hours every day. When availability is lost due to a failure, the availability lost is only the daylight hours lost, not total hours considered failed. PVRPM only considers sun-up hours, not sun-rise or sun-set hours. It also only considers whether the entire hour is sun-up or not, so for example, if most of an hour is sun-up with a sun-set towards the end, the entire hour is still considered sun-up.

During the calculation of when repairs, monitoring, and failures take place, it is assumed that as soon as a failure occurs, either monitoring or repair immediately begins the next day, depending on what is configured for that component. This must be accounted for in the user's configuration of the distributions. If multiple monitoring types are defined for a single component, then the quickest monitoring will override the others. For example, if component-level monitoring is defined for modules and cross-level monitoring from the inverter to the module, whatever has the shortest time to detection. Typically, it would be the component level monitoring.

Finally, each realization is independent of the other realizations in a simulation run. A realization is simply a single "run" of the system for the duration of its lifetime, with the provided configurations.

1.4 Logic Diagram

Here is a set of diagrams to understand how the simulation runs. Each module has a set of functions: init, reinit, and update. Init describes the logic to handle the initial creation at the beginning of the simulation. Reinit describes how to reinitialize components that are repaired. Update gives the logic of what happens on each day of the simulation.


Simulation
Logic
Failure Logic
Monitor Logic
Repair Logic

1.5 Example YAML Configuration

Please read all the comments carefully, there are many options for each section of PVRPM.

```
### Run Setup ###
# results folder doesn't need to exist, it will be created if it doesnt already
# For windows use single backslash. Do not need to escape spaces in folder names
results_folder: /path/to/results/folder
num_realizations: 2 # number of realizations to run
conf_interval: 90 #XX % confidence interval around the mean will be calculated for␣
↪summary results

### Case Setup ###
num_combiners: 2 # total number of DC combiner boxes
num_transformers: 1 # total number of Transformers
num_trackers: 2 # total number of trackers

### Financial Inputs ###
present_day_labor_rate: 100 # dollars per hour
inflation: 2.5 # in percent

### Failure Tracker Algorithm ###
use_worst_case_tracker: false


### Component Information ###
  # The structure of the user-entered component information is as follows:
  #**NOTE: components currently include: module, string, combiner, inverter, disconnect,␣
↪transformer, grid, and (optional) tracker

#    component.
#      name = a string containing the name of the component type, used for error␣
↪reporting
#      can_fail = true if component is allowed to fail
#      can_repair = true if component is allowed to be repaired after failing
#      can_monitor = true if component's failures use component level monitoring
#                    This can be false and replaced with other monitoring methods, see␣
↪below for independent and cross level monitoring sections
#      ### if can_repair, can_monitor or can_fail is false, you can remove their␣
↪respective section below (failures, monitoring and repairs)
#      warranty (can remove this section if no warranty)
#        days = number of days that the warranty is for (e.g. a 20 year warranty would␣
↪be 20 * 365 days)
#      failures (list as many failures as needed)
#        distribution = distribution type of this failure mode
#        parameters = parameters for this failure mode
#          mean: mean in days for this failure mode's distribution
#          std: standard deviation for this failure mode's distribution (not all␣
↪distributions need std)
#        labor_time = number of hours of labor it takes to repair this type of failure
#        cost = parts cost to repair this type of failure
#        cost_per_watt (optional) = USD per watt cost of failure FOR INVERTERS ONLY.␣
↪This will OVERRIDE the cost if specified for a failure
```

(continues on next page)

(continued from previous page)

```
#        by multiplying this value by the inverter size. If you want to set a static␣
↪cost, use the cost parameter as normal
#        fraction (optional) = If fraction is defined, then this failure mode is a␣
↪defective failure mode, and "fraction" represents the fraction of this type of␣
↪component that are defective. This fraction of components with this failure mode is␣
↪maintained throughout the simulation.
#        decay_fraction (optional) = Works similiarly to fraction, except the fraction␣
↪of affected components decays. For example, if 0.5 is defined, then half of the␣
↪components have this failure mode. Once those are repaired, half of the repaired␣
↪components will have this failure mode, and so on.
#      concurrent_failures (these failures are all tracked in parallel, meaning a␣
↪component can fail with these failure modes while another failure mode like above is␣
↪still counting down)
#         list failures same as failures above
#      monitoring: this section specfies how long each failure takes to be detected by␣
↪monitoring. This occurs before repair time begins. This should either be 1 section for␣
↪all failures or a section for each failure
#        distribution = distribution type of monitoring times
#        parameters = parameters of the monitoring distribution
#          mean: mean in days for this monitoring mode's distribution
#          std: standard deviation for this monitoring mode's distribution (not all␣
↪distributions need std)
#      repairs (list a repair for each failure, or only 1 repair for all failures)
#        distribution = distribution type of repair times
#        parameters = parameters of the repair distribution
#          mean: mean in days for this repair mode's distribution
#          std: standard deviation for this repair mode's distribution (not all␣
↪distributions need std)
#        **NOTE: If there is only ONE repair distribution (repair[0]), then ALL failure␣
↪modes will use that distribution! Otherwise, # repair modes must equal # failure modes.
#      concurrent_repairs (repairs for concurrent failure modes, same setup as repairs␣
↪above)
#      degradation (MODULES ONLY) (remove if no degradation) (%/year)

### Distribution types
# PVRPM has some distributions built in, where only a mean and standard deviation is␣
↪needed to
# properly model the failure or repair. Under the hood, PVRPM uses scipy.stats␣
↪distribution
# functions to model these. However, scipy.stats documentation for each function is not␣
↪very
# clear on how to convert the mean and std into usable values for the distribution, which
# is why PVRPM will wrap them for you.

# However, not every single scipy distribution is wrapped by PVRPM. These are the␣
↪distributions wrapped by PVRPM (use these as the distribution option):
#  - exponential
#  - normal
#  - uniform
#  - lognormal
#  - weibull (can also provide shape for this distribution, see below)
# If using one of these distributions, you can simply provide the mean and std in days.
```

(continues on next page)

(continued from previous page)

```yaml
# **For the weibull distribution**: instead of standard deviation (std) you can provide␣
↪the mean and shape
# See here: https://en.wikipedia.org/wiki/Weibull_distribution the shape parameter is k,␣
↪and lambda is calculated by solving for it using the gamma function and the provided␣
↪mean.
# Otherwise, if you provide the STD for the weibull distribution, it should be a large␣
↪number otherwise you'll get values extremely close to the mean only

# You can also override this, and change the distribution to match the function name of␣
↪any of the distributions listed here:
# https://docs.scipy.org/doc/scipy/reference/stats.html
# If you do this, then the "parameters" option will then be a list of kwargs to the␣
↪scipy function you select. For example, if you want to use the gamma distribution:
#  distribution: gamma
#  parameters:
#    a: 1.99
#    scale: 100 # 1 / beta
###

module:
  name: module # can be anything you want
  can_fail: true
  can_repair: true
  can_monitor: true # leave true to use monitoring distributions

  warranty:
    days: 7300 # years converted to days

  failures:
    normal_failures: # this key name can be anything you want
      distribution: normal
      parameters: # parameters for distribution chosen above, either mean and std for a␣
↪built in distribution or kwargs to the scipy function
        mean: 1460 # years converted to days
        std: 365 # days
      labor_time: 2 # in hours
      cost: 322 # in USD
    routine_failures: # this key name can be anything you want
      distribution: normal
      parameters:
        mean: 365 # mean in days, or you can do 1 / (num_failures / year * 365)
        std: 365
      labor_time: 2 # in hours
      cost: 322 # in USD
      fraction: 0.1 # > 0 and < 1, fraction of these components that are normal failures,
↪ maintained throughout the simulation
    defective_failures: # this key name can be anything you want
      distribution: exponential
      parameters:
        mean: 100 # mean in days, or you can do 1 / (num_failures / year * 365)
      labor_time: 2 # in hours
      cost: 322 # in USD
```

(continues on next page)

(continued from previous page)

```
    decay_fraction: 0.2 # > 0 and < 1, fraction of these components that are defective

  concurrent_failures: # this happens all in parallel, independent of each other
    cell_failure: # this key name can be anything you want
      distribution: normal
      parameters: # parameters for distribution chosen above, either mean and std for a␣
↪built in distribution or kwargs to the scipy function
        mean: 365 # years converted to days
        std: 365 # days
      labor_time: 2 # in hours
      cost: 322 # in USD
      decay_fraction: 0.2
    wiring_failure: # this key name can be anything you want
      distribution: normal
      parameters:
        mean: 365 # mean in days, or you can do 1 / (num_failures / year * 365)
        std: 365
      labor_time: 2 # in hours
      cost: 322 # in USD
      fraction: 0.1 # > 0 and < 1, fraction of these components that are normal failures,
↪ maintained throughout the simulation

  monitoring:
    normal_monitoring: # this key name can be anything you want
      distribution: exponential
      parameters:
        mean: 5

    defective_monitoring:
      distribution: normal
      parameters:
        mean: 15
        std: 5

  repairs:
    all_repairs: # this key name can be anything you want
      distribution: lognormal
      parameters:
        mean: 60 # in days
        std: 20 # in days

  concurrent_repairs:
    cell_repair: # this key name can be anything you want
      distribution: lognormal
      parameters:
        mean: 7 # in days
        std: 3 # in days

    wire_repair: # this key name can be anything you want
      distribution: lognormal
      parameters:
        mean: 3 # in days
```

(continues on next page)

(continued from previous page)

```
        std: 3 # in days

  degradation: 20 # modules only, how much a module degrades per year in percent

string:
  name: string
  can_fail: true
  can_repair: true
  can_monitor: true

  failures:
    failure: # this key name can be anything you want
      distribution: exponential
      parameters:
        mean: 182.5 # mean in days, or you can do 1 / (num_failures / year * 365)
      labor_time: 1 # in hours
      cost: 20 # in USD

  monitoring:
    normal_monitoring: # this key name can be anything you want
      distribution: exponential
      parameters:
        mean: 5

  repairs:
    all_repairs: # this key name can be anything you want
      distribution: lognormal
      parameters:
        mean: 7 # in days
        std: 3 # in days

combiner:
  name: combiner
  can_fail: true
  can_repair: true
  can_monitor: true

  failures:
    failure: # this key name can be anything you want
      distribution: normal
      parameters:
        mean: 730
        std: 182.5
      labor_time: 2 # in hours
      cost: 976 # in USD

  monitoring:
    normal_monitoring: # this key name can be anything you want
      distribution: exponential
      parameters:
        mean: 5
```

(continues on next page)

(continued from previous page)

```
  repairs:
    all_repairs: # this key name can be anything you want
      distribution: exponential
      parameters:
        mean: 3 # in days

inverter:
  name: inverter
  can_fail: true
  can_repair: true
  can_monitor: true

  failures:
    component_failure: # this key name can be anything you want
      distribution: exponential
      parameters:
        mean: 365
      labor_time: 0 # in hours
      cost_per_watt: 0.07 # in USD, cents/watt. This will be multiplied by the inverter_
↪size (listed in SAM). Overrides cost
    routine_failure:
      distribution: exponential
      parameters:
        mean: 365
      labor_time: 0
      cost: 1000 # static cost, not multiplied by inverter size
    catastrophic_failure:
      distribution: normal
      parameters:
        mean: 500
        std: 365.25
      labor_time: 0
      cost_per_watt: 0.35 # in USD, cents/watt. This will be multiplied by the inverter_
↪size (listed in SAM). Overrides cost

  monitoring:
    all_monitoring: # this key name can be anything you want
      distribution: exponential
      parameters:
        mean: 5

  repairs:
    component_repair: # this key name can be anything you want
      distribution: lognormal
      parameters:
        mean: 3 # in days
        std: 1.5
    routine_repair:
      distribution: exponential
      parameters:
        mean: 0.5
    catastrophic_repair:
```

(continues on next page)

(continued from previous page)

```
      distribution: lognormal
      parameters:
        mean: 3
        std: 1.5

disconnect: # A/C disconnect
  name: disconnect
  can_fail: true
  can_repair: true
  can_monitor: true

  failures:
    failure: # this key name can be anything you want
      distribution: weibull
      parameters:
        mean: 1095
        std: 1200 # use a large STD for weibull
      labor_time: 4 # in hours
      cost: 500 # in USD

  monitoring:
    normal_monitoring: # this key name can be anything you want
      distribution: exponential
      parameters:
        mean: 5

  repairs:
    all_repairs: # this key name can be anything you want
      distribution: lognormal
      parameters:
        mean: 1 # in days
        std: 0.5

transformer:
  name: transformer
  can_fail: true
  can_repair: true
  can_monitor: true

  failures:
    failure: # this key name can be anything you want
      distribution: weibull
      parameters:
        mean: 365 # can optionally provide the shape which is K parameter on wikipedia␣
↪page
        shape: 0.3477 # lambda is calculated from the mean
      labor_time: 10 # in hours
      cost: 32868 # in USD

  monitoring:
    normal_monitoring: # this key name can be anything you want
      distribution: exponential
```

(continues on next page)

(continued from previous page)

```
      parameters:
        mean: 5

  repairs:
    all_repairs: # this key name can be anything you want
      distribution: lognormal
      parameters:
        mean: 0.25 # in days
        std: 0.5

grid:
  name: grid
  can_fail: true
  can_repair: true
  can_monitor: true

  failures:
    failure: # this key name can be anything you want
      distribution: weibull
      parameters:
        mean: 100
        shape: 0.75
      labor_time: 0 # in hours
      cost: 0 # in USD

  monitoring:
    normal_monitoring: # this key name can be anything you want
      distribution: exponential
      parameters:
        mean: 5

  repairs:
    all_repairs: # this key name can be anything you want
      distribution: exponential
      parameters:
        mean: 0.5 # in days

tracker: # only required for tracking systems, remove it not using trackers
  name: tracker
  can_fail: true
  can_repair: true
  can_monitor: true

  failures:
    failure: # this key name can be anything you want
      distribution: exponential
      parameters:
        mean: 500
      labor_time: 0 # in hours
      cost: 2000 # in USD

  monitoring:
```

(continues on next page)

(continued from previous page)

```
    normal_monitoring: # this key name can be anything you want
      distribution: exponential
      parameters:
        mean: 5

  repairs:
    all_repairs: # this key name can be anything you want
      distribution: lognormal
      parameters:
        mean: 30 # in days
        std: 10


### Independent Monitoring Practices ###
# This section defines monitoring practices (like IR drone scans) that happen at an␣
↪invertal or threshold with a fix cost to reduce time to detection to 0 (i.e detect all␣
↪failed components which are still in the detection phase from monitoring). or to the␣
↪distribution sample from the defined distribution
# This occurs independent of any component level
# Here you can define the interval, cost, and what component levels this services will␣
↪detect all failures on at each interval
# Remove this section if not using it
# You can also OPTIONALLY define a distribution that states how long it takes after the␣
↪monitoring occurs for the failed components to be detected, instead of it being 0. You␣
↪may omit the distribution to have time to detection go to 0 when monitoring occurs
indep_monitoring:
  drone_ir:  # this name can be anything you want!
    interval: 1095 # interval in days when this static monitoring occurs
    cost: 50000 # cost of this monitoring in USD
    labor_time: 1 # in hours
    distribution: normal
    parameters:
      mean: 14
      std: 5
    levels: # the component levels this detects on
      - module
      - string
      - combiner

  drone_ir2: # list as many static monitoring methods as you want
    interval: 365 # this monitoring will happen every 365 days, alongside the threshold.
    # a indep monitoring triggered by a threshold RESETs the countdown to the interval
    global_threshold: 0.1 # if DC availability drops by this threshold amount, then this␣
↪indep monitoring will occur
    # DC availability is the DC power reaching the inverter(s), which is affected by␣
↪combiners, strings, and module failures
    failure_per_threshold: 0.2 # this threshold is PER LEVEL, if the availability of ANY␣
↪of the defined levels drops by this threshold amount, this indep monitoring will occur
    cost: 100000
    labor_time: 1 # in hours
    levels:
      - module
```

(continues on next page)

(continued from previous page)

```
      - combiner
      - string

  drone_ir3: # list as many static monitoring methods as you want
    failure_per_threshold: 0.2 # this threshold is PER LEVEL, people if the availability␣
↪of ANY of the defined levels drops by this threshold amount, this indep monitoring␣
↪will occur
    cost: 1500
    labor_time: 1 # in hours
    levels:
      - module

### Cross level component monitoring ###
# This next section will define optional cross level component monitoring. This means␣
↪that you can define monitoring at higher component
# levels for the component levels below it (i.e inverter monitoring modules). These come␣
↪with extra parameters to define failure dependence,
# meaning how more failures contribute to quicker detection times from monitoring. This␣
↪is done by user defined functions and parameters.
# This is only available for all levels except tracker, grid, and module. The monitoring␣
↪defined here is also overrided by monitoring at the level defined above, meaning that␣
↪if you define monitoring for modules above under the module section, monitoring of␣
↪modules defined below will be ignored.
# Also, each level can only be monitored by one higher component level, meaning that if␣
↪you define monitoring of modules at both the string and inverter level, PVRPM will␣
↪only use the monitoring distribution and compounding of the monitoring at the string␣
↪level for modules.

# component_level: The level in which monitors levels below it
#   component_monitoring: The level being monitored, must be a level below the component␣
↪level defined above
#     YOU MUST PROVIDE EITHER GLOBAL_THRESHOLD OR FAILURE_PER_THRESHOLD, or you can␣
↪provide both
#     global_threshold (float): fraction on [0, 1] that defines how many of the␣
↪components must fail across ALL MONITORED COMPONENTS before monitoring can start␣
↪detecting failures. Component failures will never be detected, and therefore not␣
↪repaired, until this fraction of failed components is met.
#     failure_per_threshold (float): fraction on [0, 1] that defines how many components␣
↪must fail under EACH COMPONENT BEING MONITORED AT THIS LEVEL. This means that if there␣
↪are 8 combiners that monitor 64 strings, every combiner will monitor 8 strings, and␣
↪the defined failure_per_threshold must fail under that combiner in order for those␣
↪failures to start being detected, as such you can define a total number of failures␣
↪across all the monitored components and also the number of failures per monitor level␣
↪component.
#     As another example, if there 8 combiners monitoring 64 strings, and enough strings␣
↪fail under combiner 1 to detect those failures, but not enough to meet the global_
↪threshold total, if strings fail under combiner 2 they wont be detected until they␣
↪either reach failure_per_threshold for that combiner or the global global_threshold
#     compounding_function (str): The function used to defined how more failures reduce␣
↪the time to detection from monitoring (NOT IMPLEMENTED)
#     compound_parameters: parameters for the function above, see below for list of␣
↪functions and their parameters (NOT IMPLEMENTED)
```

(continues on next page)

(continued from previous page)

```
#     distribution = distribution type of monitoring times
#        parameters = parameters of the monitoring distribution
#          mean: mean in days for this monitoring mode's distribution
#          std: standard deviation for this monitoring mode's distribution (not all␣
↪distributions need std)

# this section below is optional, remove if you aren't using it
component_level_monitoring:
  # lists what monitoring each component level has for levels BELOW it
  # this is for cross level monitoring only, for defining monitoring at each level use␣
↪the monitoring distributions above
  string: # component level that has the monitoring for levels below
    module: # componenet level that is below's key. same keys used above: module, string,␣
↪combiner, inverter, disconnect, grid, transformer
      global_threshold: 0.2 # fraction on [0, 1] that specifies how many of this␣
↪component type must fail before detection can occur.
                            # this means that until this threshold is met, component␣
↪failures can never be detected
                            # In the simulation, the time to detection doesn't count␣
↪down until this threshold is met, which at that point the compouning function will be␣
↪used along with the distribution as normal
      failure_per_threshold: 0.1 # the fraction of components that must fail per string␣
↪for failures to be detected at that specified string, or if the total number of␣
↪failures reach the global_threshold above
      distribution: normal # distribution that defines how long this monitoring takes to␣
↪detect a failure at this level (independent)
                           # the value calculated from this distribution will be reduced␣
↪by the compounding factor for every failure in this level
      parameters:
        mean: 1200
        std: 365

  combiner:
    string:
        failure_per_threshold: 0.2 # this fraction of strings must fail on a specific␣
↪combiner for detection for those to start
        # failures are not compounded globally in this case, only per each combiner
        distribution: normal # distribution that defines how long this monitoring takes␣
↪to detect a failure at this level (independent)
        parameters:
          mean: 1825
          std: 365

    module: # since module level monitoring is defined for strings, this will be ignored␣
↪as the higher level takes precedant
      global_threshold: 0.4 # fraction on [0, 1] that specifies how many of this␣
↪component type must fail before detection can occur.
      failure_per_threshold: 0.3
      distribution: normal # distribution that defines how long this monitoring takes to␣
↪detect a failure at this level (independent)
      parameters:
        mean: 3650
```

(continues on next page)

(continued from previous page)

```
        std: 365

  inverter:
    combiner:
      global_threshold: 0.1 # fraction on [0, 1] that specifies how many of this␣
↪component type must fail before detection can occur.
      distribution: lognormal
      parameters:
        mean: 365
        std: 365

    string:
      global_threshold: 0.2 # fraction on [0, 1] that specifies how many of this␣
↪component type must fail before detection can occur.
      distribution: exponential
      parameters:
        mean: 3650

    module:
      global_threshold: 0.5 # fraction on [0, 1] that specifies how many of this␣
↪component type must fail before detection can occur.
      distribution: lognormal
      parameters:
        mean: 3650
        std: 365
```

1.6 pvrpm

1.6.1 pvrpm package

Subpackages

pvrpm.core package

Subpackages

pvrpm.core.modules package

Submodules

pvrpm.core.modules.failure module

pvrpm.core.modules.monitor module

pvrpm.core.modules.repair module

Module contents

Submodules

pvrpm.core.case module

pvrpm.core.components module

pvrpm.core.enums module

pvrpm.core.exceptions module

pvrpm.core.logger module

pvrpm.core.simulation module

pvrpm.core.utils module

Module contents

Module contents

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex