

`pv_analyser` Module

Muhammad Elgamal Dec 25, 2022

1 Dependencies, APIs, and code templates

Several APIs were used in the analysis module including the following

1.1 Mathematical and scientific computing packages

NumPy [1] is a Python package that is useful for manipulating multidimensional-array objects called “ndarray”. Such objects can save time for doing complex mathematical computations in the form of matrices. Sorting, addressing or matrix operations can be done easily with NumPy in a few lines of code. Most of used classes in the module have the form of vector operations, interpolation, numerical differentiation/integration, logical addressing and performing statistical operations and they could be performed easily by the help of NumPy.

SciPy [2] is another Python package that has different applications including symbolic computation, dealing with special functions, optimization problems, signal processing, linear algebra, statistics, and file input/output operations. The module uses SciPy as a part of the parameter extraction process to solve complicated nonlinear equations that involve Lambert W function and advanced parameter fitting manipulations, besides its use for saving and manipulating `.mat` files.

Pandas [3] is another Python package used mainly for big-data analytics and artificial-intelligence applications. It is used in several parts of the module to organize large portions of the data especially parsed station reports.

Frechetdist [4] is a Python package for computing the Fréchet distance between two curves which measures the extent at which two curves are so similar. A zero Fréchet distance means both curves coincide, and larger one means greater dissimilarity between both [5]. This package is used in computing the dissimilarity between different IV measurements.

1.2 Weather APIs

The calculation of environmental conditions for a solar panel (i.e., the incident irradiance Irr_T and the cell temperature T_c) depend on actual measurements from the installed site beside modelled data for weather conditions.

Real datasets including the **GHI**, **DNI**, **DHI**, ambient temperature and wind speed are available online for free including

- TMY (Typical Meteorological Year) 2/3[6] :includes data for the United States from 1961 to 2019 and some other regions in the world.
- NSRDB (National Solar Radiation Database) [7], [8]: includes terabytes of data on most of geographical regions in the world up to 2021.
- PVGIS (Photovoltaic Geographical Information System) [9] which is an online dataset provided by the European commission for several regions up to 2020.
- EPW (Energy-Plus Weather)[10]

All aforementioned datasets can be accessed through specific **APIs** provided by PVLib, but unfortunately, they are historical databases and many of them does not have information about Egypt. Additionally, most databases that have live data are proprietary for a subscription fee [11], [12].

So instead, we depended on MeteoStat[13] to retrieve the hourly measurements of ambient temperature, wind speed and pressure. MeteoStat searches multiple online databases for weather of every location on earth up to the time of inquiry. If a data is missing for a specific location or at specific time, MeteoStat performs spatial or time interpolation to find the required value.

In the designed module Temperature inquiry is done as a class method in `pv_analyser.iv_measurement.update_temperature()`. At such method, the windspeed is used to find cell temperature as in (1) and (2). Given that T_m is back temperature of solar panel, a and b are chosen from Table 1

$$T_m = Irr_T \exp[a + b \times WS] + T_a \quad (1)$$

$$T_c = T_m + \frac{Irr_T}{Irr_{STC}} \Delta T \quad (2)$$

Table 1 Thermal correction factors and temperature change of a solar panel in different fixtures

<i>Solar Panel Fixture</i>	<i>a</i>	<i>b</i>	<i>ΔT</i>
<i>Open-rack glass front and back</i>	-3.47	-0.0594	3
<i>Open-rack glass front and polymer back</i>	-3.56	-0.0750	3
<i>Close-mount glass front and back</i>	-2.98	-0.0471	1
<i>Insulated-back glass polymer back</i>	-2.81	-0.0455	0

PVLib [14] is a Python package that can model several photovoltaic systems environmental conditions like finding irradiance values whether through online datasets or by

other empirical models. It also can model several solar panel designs like bifacial panels for instance.

In our module, to find an irradiance for a specific object of the `pv_analyser.iv_measurement` class, the method `update_irradiance()` is used, which uses wind speed and temperature provided by MeteoStat, station location and measurement time to find the solar position using Perez algorithm [15]. The GHI, DNI and DHI are calculated by Perez-Ineichen model [16]–[19] and then total irradiance is calculated. The whole procedure is summarized in Figure 1.

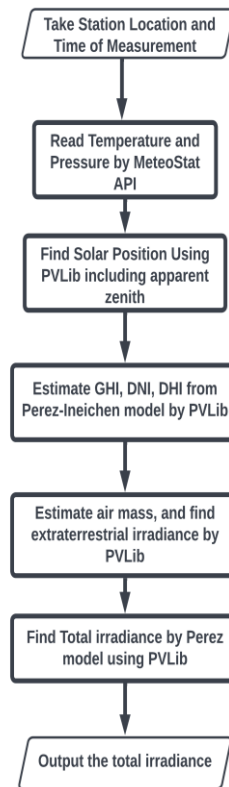


Figure 1 Summary of Irradiance Finding Algorithm

1.3 Additional packages

Matplotlib [20] is used to plot an IV measurement and for graphical purposes. “pickle” and “datetime” are built-in package that help in saving variables, and manipulating time objects respectively

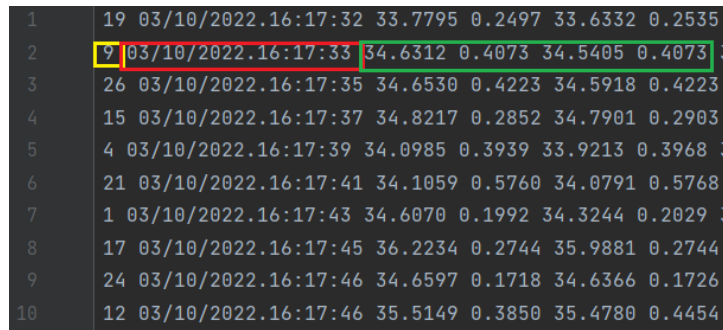
2 Functions

Several functions were defined outside the scope of any used class however, they are

called within module classes or externally.

2.1 Reading a single characterization

`pv_analyser.find_iv(characterization)` is a function that reads a single characterization in a station report and extracts the IV characteristics from it. Figure 2 shows how the station report is formatted. For instance, characterization number 2 is composed panel ID, followed by a time stamp then a sequence of points like this $v_1, i_1, v_2, i_2 \dots$ which includes all reading.



1	19	03/10/2022.16:17:32	33.7795	0.2497	33.6332	0.2535	
2	9	03/10/2022.16:17:33	34.6312	0.4073	34.5405	0.4073	3
3	26	03/10/2022.16:17:35	34.6530	0.4223	34.5918	0.4223	
4	15	03/10/2022.16:17:37	34.8217	0.2852	34.7901	0.2903	
5	4	03/10/2022.16:17:39	34.0985	0.3939	33.9213	0.3968	3
6	21	03/10/2022.16:17:41	34.1059	0.5760	34.0791	0.5768	
7	1	03/10/2022.16:17:43	34.6070	0.1992	34.3244	0.2029	3
8	17	03/10/2022.16:17:45	36.2234	0.2744	35.9881	0.2744	
9	24	03/10/2022.16:17:46	34.6597	0.1718	34.6366	0.1726	
10	12	03/10/2022.16:17:46	35.5149	0.3850	35.4780	0.4454	

Figure 2 Sample station report. Yellow is panel ID, red is timestamp and green is sequence of IV points.

2.2 Reading and saving variables

`pv_analyser.save_variable(object, name)` and `pv_analyser.read_varaiable(name)` are functions that use pickle library to save extracted analysis, import and export data.

2.3 Parsing and analyzing reports

`pv_analyser.parse_station_report()` is a main function that reads station reports in the format shown in Figure 2. The inputs of the function are listed below:

1. **file_name**: name and location of the station report.
2. **full_report**: a logic variable that asks if the user wants to read the full report or portion of it (default **True**).
3. **atspecific**: if the **full_report** is set to **False**. It takes the value '**location**' or '**all-panels-at-instance**' to read characterizations at a specific panel or all panels at specific time duration.
4. **time**: if **full_report** is **False** then time receives two **datetime.datetime** objects for the start and end time of readings. Also, if **atspecific** is '**all-panels-at-instance**' then the first value is for the time at which reading is done. The function retrieves readings with minimum difference in time to the target time.
5. **panel_id**: takes the panel ID which is important if **atspecific** is '**location**'.
6. **extract_parameters**: a logic variable (default **False**) that determines if the user wants to extract industrial parameters of the IV measurement.
7. **condition_extraction**: a logic variable (default **True**) that determines if IV measurement needs to be conditioned prior to parameter extraction.
8. **find_circuit_parameters**: a logic variable (default **False**) that determines if panel circuit parameters need to be evaluated.

9. **cell_count**: an integer variable that defines the number of cells within a panel which is useful when finding the circuit parameters.

Additionally, **pv_analyser.parse_station_report()** discards all characterizations that have number of points below a specific threshold stored in the variable **reading_length_threshold** and it assures that all readings have positive current value. The function returns a list of **pv_analyser.iv_measurement** objects related to each characterization.

pv_analyser.analyse_station_across_space() performs space analysis at specific time instance, and if data is missing at this instance, the function interpolates a new reading linearly between the two nearest readings: one before and one after and it takes the following variables as input:

1. **source_file**: destination of station report file
2. **time_of_analysis**: a **datetime.datetime** object that corresponds to the instance we want our analysis at.
3. **panels_included**: a list with all panels at which we want to make a time analysis.
4. **chosen_parameters**: a string list that includes all parameters that we want to include in our analysis possible values are within this list ['isc', 'voc', 'imp', 'vmp', 'mp', 'fillfac', 'effciency', 'irradiance', 'temprature', 'rs', 'rp', 'idark', 'ideality', 'iph'].
5. **time_resolution**: an integer representing the range at which we look for interpolating readings such that the function looks for readings at **time_of_analysis - time_resolution/2** to **time_of_analysis + time_resolution/2**.

pv_analyser.analyse_panel_through_time() is the function that does time analysis. It has inputs **source_file** and **chosen_parameters** like those in **pv_analyser.analyse_station_through_space()**. However, it has extra parameters specifying the panel under analysis (**panel_id**) and two-element integer lists to signify the start and end of the analysis period (**year, month, day, hour** and **min**). Furthermore, it has a **cell_count** input which is used when doing circuit parameter estimation.

Both **pv_analyser.analyse_station_across_space()** and **pv_analyser.analyse_panel_through_time()** return a **pandas.dataframe** object having time as index column and each of **chosen_parameters** list as a column header.

2.4 Running Optimization

As parameter extraction is considered a minimization problem, we define a function that does mathematical optimization called **pv_analyser.run_optimizer()** whose inputs are

1. **func**, a function header that will be passed to the optimizer, the passed function must have a single list input so each index corresponds to a parameter of this function.
2. **x0**, the initial point passed to the optimizer which is useful for finding local minima of optimized functions.
3. **ranges**, a list of slice objects (datatype in python that has start, end and step in one object) for each parameter of the optimized function which is useful for global optimization.

4. **method**, is a string that specifies different methods of optimization that can be applied. It has one of these values: **'brute'**, **'Nelder-Mead'**, **'SLSQP'**, **'L-BFGS-B'** and **'TNC'**.

When the method chosen is **'brute'**, only **ranges** are considered as they limit the search space at which the optimizer is looking for a global minimum. This method is guaranteed to reach a global minimum but with huge number of computations so methods like Nelder-Mead[21], Sequential Least-Square Quadratic Programming (SLSQP) [22], L-BFGS-B [23] and Truncated Newton Constrained algorithm [24] – are used to find a local minimum given an initial value **x0**. The global minimum methods are way faster but can get trapped sometimes preventing them finding the true minimum of some problem. The output of the function is the parameter list at the minimum and the value of function at global minimum.

3 Used classes

Two independent classes exist within the module: **pv_analyser.panel_diode** and **pv_analyser.iv_measurement**. These classes are not inherited by any other class. **panel_diode** is a class with properties that are the circuit parameters of a panel bypass diode. It has also the capability of finding voltage at specific current value through its method **find_voltage**.

3.1 iv_measurement class

This class is the most important class of the analysis module, and we will discuss its internal structure below.

First, its constructor passes the following parameters

1. **measuring_time** and **acquiry_time**: must be **datetime.datetime** objects they signify the time this measurement was taken at or how much does it need to simulate it respectively.
2. **temprature** and **irradiance**: floats for both temperature in C^0 and irradiance in W/m^2
3. **current, voltage**: flattened NumPy arrays of same length.
4. **panel_id**: integer for panel ID.

When the **iv_measurement** is instantiated, it has the following properties

1. **measured_current, current, simulated_current** or **measured_voltage, voltage, simulated_voltage**: they are all flattened NumPy arrays that include measured reading, a conditioned version of the same reading and a circuit simulation with proper SDM parameters that best-fits the reading respectively.
2. **vmin, vmax, voc, isc, imp, iph, rs, rp, idark, ideality** and **effciency**: the minimum voltage at the measured reading, maximum voltage, V_{OC} (can be different if the reading was not conditioned), I_{SC} , I_{MP} , and I_{ph} , R_S , R_P , I_{dark} , n for the extracted SDM parameters and finally η for the taken measurement.
3. **statematrix**: a Python list of NumPy 2D arrays including in this order

$R_S, R_P, I_{ph}, I_{dark}, n$ and temperature matrix. This property stores all the variables required that lead to this IV measurement for a given panel.

Feature Extraction with `iv_measurement.extract_features()`

This method extracts industrial parameters of an IV measurement, condition real measurement and/or extract current modes within a an IV reading.

The method **`extract_features`** has an optional input **`condition_measurement`** which is set to **`False`** by default. This option guarantees that the taken measurement is less noisy, uniformly sampled, has few irregularities that make it suitable for further processing.

The conditioning algorithm proceeds as follows:

1. Sort reading from lower voltage to upper voltage (and related current values as well).
2. Define a voltage vector running from 0 to highest voltage in the reading. The number of points in this vector is the optional input **`point_count`** with default value of 1000.
3. Do linear interpolation for the current value at each point of the newly formed voltage vector to be able to uniformly sample the measurement.
4. Numerically differentiate the interpolated current vector with respect to the voltage vector and take the absolute value of the derivative (i.e., compute the absolute instantaneous conductance).
5. After the maximum power point, locate parts of the current vector whose absolute derivative is below a predefined threshold and eliminate them. The conditioning threshold is taken as an optional input to **`extract_features`** and it is called **`conditioning_threshold`** whose default value is 0.05.
6. Take the average of the first fifty points in the current vector and extend them to the left as the short-circuit current value of the reading.
7. For the open-circuit point extrapolate the reading with a polynomial until it intersects with the voltage axis. The default order of this polynomial is taken by the optional input **`fitting_order`** whose default value is 10.

If **`condition_measurement`** is True, then the function checks whether it is required to find current modes or not depending on the option **`find_current_modes`** which is set by default to False. Finding current modes follows the steps below

1. Histogram the conditioned current vector at big number of bins (1000), and find corresponding bin values.
2. Pad the histogram vector with zeros at the end, such that if a peak arises at the last element of the vector, it will be succeeded with a zero (so it is easier to identify as a peak).
3. Fit a polynomial to the histogram vector, to be easier in finding peaks without being trapped in a local noised peak (useful in analyzing a noisy reading).
4. Use **`scipy.signal.find_peaks()`** to find peaks in the histogram vector and choose the biggest n ones such that n is the number of cell strings.
5. Choose either part or all of chosen peaks that have specific relative height to each other.
6. Save chosen peaks at **`current_modes`** property.

All other common industrial parameters are found by means of (3)

$$\left\{ \begin{array}{l} I_{SC} = I | V = 0 \\ V_{OC} = V | I = 0 \\ P_{MP} = P | \frac{dP}{dV} = 0 \\ V_{MP} = V | P = P_{MP} \\ I_{MP} = I | P = P_{MP} \\ FF = \frac{V_{MP} I_{MP}}{V_{OC} I_{SC}} \\ \eta = \frac{P_{MP}/A}{Irr_{max}} \end{array} \right. \quad (3)$$

depending on the properties **voltage** and **current** which can be updated if conditioning has occurred.

Parameter Extraction using `iv_measurement.equivalent_circuit()`

This method extracts the circuit parameters of the SDM model from a given IV measurement and stores the output measurements in the properties **simulated_voltage** and **simulated_current**. Additionally, as sometimes the parameter extraction method fails, it updates the **equivalent_circuit_found** property to **False** if the operation failed.

Two inputs are taken for this function: **N** and **method**. **N** is the number of cells in series for the taken IV reading and method can take either of these values: '**stornelli**' or '**modified_laudani**'. If the method is '**stornelli**', reading is conditioned at the beginning as all calculations depend on the maximum power point and smoothness around and afterward the same procedure proposed in [25] is implemented.

On the other hand, if method is '**modified_laudani**', it is made sure that reading is conditioned, then the value of maximum possible series resistance is obtained from (4)

$$R_S^{max}(n) = \frac{nN_S V_{th}}{I_{MP}} \left[1 + W \left\{ -\exp \left(\frac{V_{OC} - 2V_{MP} - nN_S V_{th}}{nN_S V_{th}} \right) \right\} \right] + \frac{V_{MP}}{I_{MP}} \quad (4)$$

Three internal functions are defined within the scope of the condition of having a '**modified-laudani**' method. The first is **calculate_parameters()** and it takes $\theta = (n, R_S)$ and returns all SDM parameters by the reduced forms given in [26]. The second function is **cost_function()** and it takes the parameters as an input and then it evaluates the maximum absolute error to be minimized. The third function is called **example_function()** and it receives a list of two elements corresponding to $(n, \log(R_S))$ because usually the value of R_S is very small so we can deal better with logarithms to search the parameter space.

example_function() is then passed to **run_optimizer()** with the option **method** equals '**TNC**' and if the value exceeds 10% of the I_{SC} then the algorithm runs brute force optimizer, but this is done very rarely and the parameters minimizing **example_function()** are obtained beside storing the best fit IV characteristics in **simulated_voltage** and **simulated_current** properties.

Estimating Sunrise/Sunset using `iv_measurement.find_sun_time()`

It is important sometimes to inspect the measurements against the time of measurement across the day, so it becomes important to know the sunrise and the sunset of a given date. PVLib provided `pvlib.solarposition.sun_rise_set_transit_spa()` based on [27] that can estimate clearly the sunrise and the sunset of a specific date. So, a user can grasp if the taken measurements make sense or not. The evaluation of these times depends on the varying solar position, and the location of the station and time zone which are kept as constant in the module. The outputs of this function are stored in `sunrise` and `sunset` properties.

Estimating Incident Irradiance using `iv_measurement.update_irradiance()`

To find the irradiance, one needs to know the location and time of measurement. The location is called from a module constant and stored in `pvlib.location.location` object. It is important to evaluate the temperature and pressure which can be given by the help of MeteoStat to find the apparent solar position. Afterward, given the solar position and the time, one can find all components of the radiation: DNI, DHI, GHI by Perez-Ineichen model [16]–[19]. Additionally, extraterrestrial radiation component is calculated using `pvlib.irradiance.get_extra_radiation()` which depends on [18], [28]–[30]. After all components of the irradiance are calculated (DNI, DHI, GHI, extraterrestrial) and given the albedo of the location defined as a constant of 0.4, the total irradiance is calculated and saved in the property `irradiance`.

Estimating Cell Temperature using `iv_measurement.update_temperature()`

At the beginning it is made sure that the irradiance is estimated. Then both the ambient temperature and wind speed are taken by the MeteoStat API then they are substituted in (1) and (2) to estimate cell temperature. All required constants are stored in the module as in Table 1 for the case ‘open-rack glass front and polymer back’.

Doing Time Interpolation Between Two Readings using `iv_measurement.time_interpolate()`

At sometimes, when analyzing the whole station at once, you may not be able to find all readings at this instance so, you need to interpolate two readings together and finding an intermediate reading. To make this happen, you need to make sure that both readings are sampled at the same voltage points (which can be done by linear interpolation as well), then you linearly interpolate each voltage sample based on the timing of the two IV measurements. The inputs of these functions are two other `pv_analyser.iv_measurement` objects.

Saving to and Reading from .mat files

One needs to save data to a format readable by other programming languages like MATLAB. So important properties of the **iv_measurement** object are stored into a dictionary and got saved by **scipy.io.savemat()** within the method **iv_measurement.save_as_mat()**. Or on the other hand, one can read a **.mat** file with specific entries into an **iv_measurement** object using **iv_measurement.read_from_mat()**. Both functions take a string for the file name to which one needs to read from or write into.

Displaying Measurements and Printing Features

One can print industrial parameters of an IV measurement using **iv_measurement.display_features()** and also a plot of the IV characteristics stored in properties **voltage** and **current** can be done with **iv_measurement.plot()**. The plotting function can be run in a loop but after setting the optional input **several_plots** to **True** and specifying the duration between plots by changing **duration** whose default is 1.3 seconds.

Finding Dissimilarity between Two Measurements

One can calculate the similarity between two IV measurements, which is useful by first normalizing the voltage and current readings over the maximum voltage and current of the reference reading then calculate the Fréchet distance between both curves (only the part that is common between them on voltage axis). This process is done in **iv_measurement.find_dissimilarity()**

3.2 cell class

This class is the parent of both **cell_string** and **panel** classes. The constructor function takes the SDM parameters as inputs then the method **cell.characterize()** does the characterization process.

cell.characterize() takes two inputs **current** - which is a NumPy vector with current readings we want to evaluate our model at and **start_guess** which is an initial value fed to the equation solver while finding the IV characteristics.

Within this function there is a loop over all values in the current axis, then an error function called **SDM(p)** which takes **p** (value of voltage) and tries to minimize the error when subtracting both sides of the IV equation. then the value doing so is taken as an acceptable voltage value. This is done by the help of **scipy.optimize.fsolve()**. Afterward, unnecessary values in the current axis above the photocurrent are removed and data is resampled at equidistant

voltages and saved in the property **measurement** of type **iv_measurement**.

3.3 cell_string class

This class inherits from **cell** class, however, **characterize()** method is implemented differently. In such an implementation it takes the current vector and assigns an empty voltage vector and loops over all the cells of the cell string by defining sub-cells and summing their output voltage exactly as in (5)

$$V_{string,i} = \sum_{k=1}^m V_{cell,k} \text{ and } 1 \leq i \leq m \quad (5)$$

The resulting vector undergoes the same processing as in **cell** class and also gets saved in **measurement** property.

3.4 panel class

This class embodies the description of physical panel from industrial point of view and from circuit-wise equivalent point of view. For instance, properties like **model_name**, **rows**, **columns**, **cell_string_count** and **area** represent the physical properties of the panel. Also, datasheet parameters are included in properties like **isc**, **voc**, **mp**, **vmp**, **imp**, **efficiency**, **power_thermal_coefficient** and **voc_thermal_coefficient**. Furthermore, some parameters are extracted manually from datasheet like **isc_nonlinearity_factor** and **voc_irradiance_correction_factors**. The I_{SC} nonlinearity factor is α in (6)

$$\alpha = \ln\left(\frac{I_{SC,1}}{I_{SC,2}}\right) / \ln\left(\frac{Irr_1}{Irr_2}\right) \quad (6)$$

and it is calculated by observing the lowest and highest I_{SC} at the lowest and highest irradiance levels. Moreover, the V_{OC} correction factors are the parameters that best fit (7)

$$V_{OC} - V_{OC,STC} - \mu_{voc}\Delta T = c_0 + c_1 \ln(Irr^\nabla) \quad (7)$$

Aside from these properties, datasheet measurement is included into the property **ideal_measurement** and all other measurements of the same panel are stored in **measurements** property which is a list of **iv_measurement** objects. The initial values of all aforementioned parameters are those of the used panel as shown in panel specifications.

Loading a Measurement using **panel.load_measurement()**

The method **load_measurement()** can load measurement from different sources into the panel object properties **ideal_measurement** or **measurements**. If the optional input **find_equivalent_circuit** is set to **True** (while its default is **False**), then the SDM

parameters of the added reading is extracted as well and saved into the saved **iv_measurement** (either in **measurements** or **ideal_measurement** properties).

This function has three other optional inputs:

1. **source_type**: and it can be **'csv'** or **'measurement'**, which allows the loaded measurements to be taken from a **.csv** file or from an **iv_measurement** object
2. **source**: which includes wither a string of the file name or a variable of the **iv_measurement**.
3. **type**: it can be **'STC'** so it will be loaded into the **ideal_measurement** property- or **'taken'** which will append the read measurement into the end of the measurements list.

Characterizing a Panel using **panel.characterize()**

To characterize a panel there are two different scenarios, either to assume all cells have same set of SDM parameters then a very good approximation of the IV characteristics is (8)

$$I = I_{ph} - I_{dark} \left[\exp \left[\frac{e(V + IR_S)}{nN_s kT} \right] - 1 \right] - \frac{V + IR_S}{R_p} \quad (8)$$

but here the SDM parameters given are panel-level parameters so R_S given is the R_S for the whole panel and so on. The other scenario is to simulate where each cell has different set of SDM parameters and in these situations, we depend on (9).

$$V(I) = \left[\sum_{i=1}^z V_i(I) \right] + (n - z) \frac{\alpha kT}{e} \ln \left(\frac{I}{I_s} + 1 \right) \quad (9)$$

Either way, one needs first to pass an **iv_measurement** object containing the current values we want to run our simulation at besides the full set of SDM parameters per cell as they are located physically stored in the property **statematrix** of the passed **iv_measurement** object. The passed object needs to be stored at specific index in measurements property.

The function takes **index**, **start_guess** and **parameters_are_panel_level** as optional inputs with default values of **0**, **2** and **True** respectively. If the number of elements in **measurements[index].statematrix[0]** (or any other element in **statematrix**) exceed

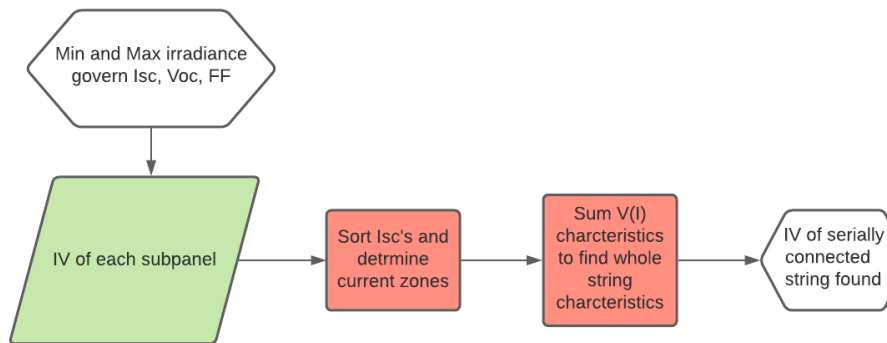


Figure 3 Finding panel irradiance from cell irradiance

one then we consider this as a sign we are doing a cell level simulation and in such scenario, we reshape the dimensions of SDM parameter matrices to comply not with physical placement but with electrical connection, so each row does not represent a physical row but cells in a single cell string. The output of this function is stored in the properties **simulated_voltage** and **simulated_current** of **measurements[index]**.

When simulating a panel from cell level, one needs to sort the minimum photocurrent per each cell string and accordingly we define different current zones and apply (9) freely, a process depicted in Figure 3 where subpanel means a cell string and we assume that a panel when properly operated is a serial string of cells. However, at some instances, the minimum photocurrent in each string can be similar to others so we perturb the photocurrent at such string with error less than 0.1% to be able to use the same formula again even with similar minimum photocurrents.

When simulating a panel as a whole (when **parameters_are_panel_level** is set to **True**), then the situation is much nicer because of solving (8) numerically.

Translating Parameters using **panel.translate_parameters()**

Parameter translation occurs only when the property **ideal_measurement** is full with an **iv_measurement** object. At the beginning we make sure that **ideal_measurement** has SDM parameters extracted and if not we do **ideal_measurement.equivalent_circuit()**. We additionally, depend on the STC parameters embodied within the panel class itself. **panel.translate_parameters()** takes two inputs: **G** and **T** for irradiance and temperature respectively. Equations (10), (11), (12), (13), (14), (15), (16), (17), and (18) to translate parameters.

$$\begin{cases} Irr^\nabla = \frac{Irr}{Irr_{STC}} \\ T^\nabla = \frac{T}{T_{STC}} \\ \Delta T = T - T_{STC} \end{cases} \quad (10)$$

$$I_{ph}(Irr^\nabla, \Delta T) = Irr^\nabla^\alpha a(I_{ph,STC} + \mu_{isc}\Delta T) \quad (11)$$

$$R_p = \frac{R_{p,STC}}{Irr^\nabla} \quad (12)$$

$$R_s = T^\nabla(1 - 0.217 \ln(Irr^\nabla))R_{s,STC} \quad (13)$$

$$n = n_{STC}T^\nabla \quad (14)$$

$$\begin{cases} I_{dark} = \frac{I_{SC} - (C - D)}{EXP_{OC} - EXP_{SC}} \\ I_{ph} = EXP_{OC}I_{dark} + C \end{cases} \quad (15)$$

given $C = V_{OC}/R_P$ and $D = I_{SC}R_S/R_P$.

$$\begin{cases} EXP_{OC} = \exp\left(\frac{V_{OC}}{nN_S V_{th}}\right) \\ EXP_{SC} = \exp\left(\frac{R_S I_{SC}}{nN_S V_{th}}\right) \\ EXP_{MP} = \exp\left(\frac{V_{MP} + R_S I_{MP}}{nN_S V_{th}}\right) \end{cases} \quad (16)$$

$$P_{MP} = Irr^\nabla (P_{MP,STC} + \mu_{mp}\Delta T) \quad (17)$$

$$\eta = \eta_{STC} + \frac{\mu_{mp}\Delta T}{A Irr_{STC}} \quad (18)$$

are solved sequentially to translate the panel-level SDM parameters. Afterward, the translated parameters beside simulation at these new parameters are stored in the new property **translated_measurement**.

References

- [1] S. van der Walt, S. C. Colbert, and G. Varoquaux, "The NumPy array: A structure for efficient numerical computation," *Comput Sci Eng*, vol. 13, no. 2, 2011, doi: 10.1109/MCSE.2011.37.
- [2] P. Virtanen *et al.*, "SciPy 1.0: fundamental algorithms for scientific computing in Python," *Nat Methods*, vol. 17, no. 3, 2020, doi: 10.1038/s41592-019-0686-2.
- [3] W. McKinney, "pandas: a Foundational Python Library for Data Analysis and Statistics," *Python for High Performance and Scientific Computing*, 2011.
- [4] "frechetdist . PyPI." <https://pypi.org/project/frechetdist/> (accessed Dec. 14, 2022).
- [5] HELMUT ALT and MICHAEL GODAU, "Computing the Frechet distance between two polygonal curves," *Int J Comput Geom Appl*, vol. 5, no. 1n2, 1995.
- [6] W. Marion and K. Urban, "User's Manual for TMY2s: Typical Meteorological Years: Derived from the 1961-1990 National Solar Radiation Data Base," 1995.
- [7] E. L. Maxwell, "METSTAT - The solar radiation model used in the production of

- the national solar radiation data base (NSRDB)," *Solar Energy*, vol. 62, no. 4, 1998, doi: 10.1016/S0038-092X(98)00003-6.
- [8] M. Sengupta, Y. Xie, A. Lopez, A. Habte, G. Maclaurin, and J. Shelby, "The National Solar Radiation Data Base (NSRDB)," *Renewable and Sustainable Energy Reviews*, vol. 89. 2018. doi: 10.1016/j.rser.2018.03.003.
- [9] European Commission, "PVGIS Photovoltaic Geographical Information System." https://joint-research-centre.ec.europa.eu/pvgis-photovoltaic-geographical-information-system_en (accessed Nov. 26, 2022).
- [10] N. Fumo, P. Mago, and R. Luck, "Methodology to estimate building energy consumption using EnergyPlus Benchmark Models," *Energy Build*, vol. 42, no. 12, 2010, doi: 10.1016/j.enbuild.2010.07.027.
- [11] "Vaisala: A global leader in environmental and industrial measurements." <https://www.vaisala.com/en> (accessed Nov. 26, 2022).
- [12] "Solar Irradiance Data | Solargis: weather data and software for solar power investments."
- [13] "MeteoStat: The Weather Record Keeper." <https://meteostat.net/en/> (accessed Nov. 21, 2022).
- [14] "pvlib python -- pvlib python 0.9.3 documentation." <https://pvlib-python.readthedocs.io/en/stable/> (accessed Nov. 26, 2022).
- [15] A. K. Yadav and S. S. Chandel, "Tilt angle optimization to maximize incident solar radiation: A review," *Renewable and Sustainable Energy Reviews*, vol. 23. 2013. doi: 10.1016/j.rser.2013.02.027.
- [16] P. Ineichen and R. Perez, "A new airmass independent formulation for the linke turbidity coefficient," *Solar Energy*, vol. 73, no. 3, 2002, doi: 10.1016/S0038-092X(02)00045-2.
- [17] R. Perez *et al.*, "A new operational model for satellite-derived irradiances: Description and validation," *Solar Energy*, vol. 73, no. 5, 2002, doi: 10.1016/S0038-092X(02)00122-6.
- [18] M. J. Reno, C. W. Hansen, and J. S. Stein, "Global Horizontal Irradiance Clear Sky Models: Implementation and Analysis," *SANDIA REPORT SAND2012-2389 Unlimited Release Printed March 2012*, no. March, 2012.
- [19] J. Remund *et al.*, "Worldwide Linke turbidity information To cite this version :," *Proceedings of ISES Solar World Congress*, 2003.
- [20] "matplotlib -- Visualization with Python." <https://matplotlib.org/> (accessed Nov. 26, 2022).
- [21] F. Gao and L. Han, "Implementing the Nelder-Mead simplex algorithm with adaptive parameters," *Comput Optim Appl*, vol. 51, no. 1, 2012, doi: 10.1007/s10589-010-9329-3.
- [22] D. Kraft, "A Software Package for Sequential Quadratic Programming," *Technical Report DFVLR-FB*, vol. 88, no. 28. 1988.
- [23] C. Zhu, R. H. Byrd, P. Lu, and J. Nocedal, "Algorithm 778: L-BFGS-B: Fortran Subroutines for Large-Scale Bound-Constrained Optimization," *ACM Transactions on Mathematical Software*, vol. 23, no. 4, 1997, doi: 10.1145/279232.279236.
- [24] S. G. Nash, "Preconditioning of Truncated-Newton Methods," *SIAM Journal on Scientific and Statistical Computing*, vol. 6, no. 3, 1985, doi: 10.1137/0906042.
- [25] V. Stornelli, M. Muttillio, T. de Rubeis, and I. Nardi, "A new simplified five-parameter estimation method for single-diode model of photovoltaic panels," *Energies*

(Basel), vol. 12, no. 22, 2019, doi: 10.3390/en12224271.

[26] A. Laudani, F. Riganti Fulginei, and A. Salvini, "High performing extraction procedure for the one-diode model of a photovoltaic panel from experimental I-V curves by using reduced forms," *Solar Energy*, vol. 103, 2014, doi: 10.1016/j.solener.2014.02.014.

[27] I. Reda and A. Andreas, "Solar position algorithm for solar radiation applications," *Solar Energy*, vol. 76, no. 5, 2004, doi: 10.1016/j.solener.2003.12.003.

[28] ASCE-EWRI, "The ASCE standardized reference evapotranspiration equation: ASCE-EWRI Standardization of Reference Evapotranspiration Task Committee Report," *American Society of Civil Engineers*, 2005.

[29] J. A. Duffie, W. A. Beckman, and W. M. Worek, "Solar Engineering of Thermal Processes, 2nd ed.," *J Sol Energy Eng*, vol. 116, no. 1, 1994, doi: 10.1115/1.2930068.

[30] J. Gribbin, "Radiative Processes in Meteorology and Climatology," *Physics Bulletin*, vol. 27, no. 12, 1976, doi: 10.1088/0031-9112/27/12/041.